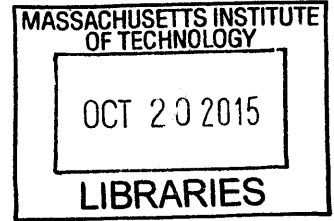


ARCHIVES



Synthetic Tutor:
Profiling Students and Mass-Customizing Learning
Processes Dynamically in Design Scripting Education

by

Ju Hong Park

M. Arch., Harvard University (2005)
B. Eng., Hongik University (1998)

Submitted to the Department of Architecture in Partial
Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Architecture: Design and Computation

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

© 2015 Ju Hong Park. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute paper and electric
copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature redacted

Signature of Author

Design and Computation Group, Department of Architecture
July 31, 2015

Signature redacted

Certified by

Takehiko Nagakura, Ph.D.
Associate Professor of Design and Computation
Thesis Supervisor

Signature redacted

Accepted by

Takehiko Nagakura, Ph.D.
Associate Professor of Design and Computation
Chair, Department Committee on Graduate Students



77 Massachusetts Avenue
Cambridge, MA 02139
<http://libraries.mit.edu/ask>

DISCLAIMER NOTICE

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available.

Thank you.

The images contained in this document are of the best quality available.

Synthetic Tutor

Dissertation Committee

Takehiko Nagakura

Associate Professor of Design and Computation
Massachusetts Institute of Technology
Chair, Department Committee on Graduate Students
Thesis Advisor

Terry Knight

Professor of Design and Computation
Massachusetts Institute of Technology
Reader

George Kocur

Senior Lecturer of Civil and Environmental Engineering
Massachusetts Institute of Technology
Reader

Tina Grotzer

Associate Professor of Education
Harvard Graduate School of Education
Reader

Synthetic Tutor

Synthetic Tutor:
Profiling Students and Mass-Customizing Learning
Processes Dynamically in Design Scripting Education

by

Ju Hong Park

Submitted to the Department of Architecture on July 31, 2015
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Architecture: Design and Computation

ABSTRACT

Artificial intelligence is substituting human intelligence and robots are replacing human workers. Instead of settling for this competitive relationship between humans and machines, this thesis proposes a novel framework in which humans and machines work together to solve the complex problems of design-scripting education, problems which humans or machines alone cannot easily solve. In design education, there are few clear guides and pedagogies that can effectively teach students with diverse educational and professional backgrounds, some of who may need individualized tutoring. This thesis specifically explores applications of artificial intelligence (machine learning and computer vision algorithms) in which humans and machines mutually improve their learning performance. Humans can increase a machine's performance by providing training-data sets that can be a foundation for intelligent decision-making. Machines, on the other hand, can improve humans' learning performance by analyzing human study patterns and providing mass-customized instructions. This thesis illustrates that the developed *Synthetic Tutor* provides novice students with architectural precedents by analyzing their drawings and documents and effectively teaches these students introductory computer programming skills in the context of architectural design. Therefore, this human-machine collaboration has proven an effective framework to solve these ill-structured problems.

Thesis Supervisor: Takehiko Nagakura, PhD
Title: Associate Professor of Design and Computation
Chair, Department Committee on Graduate Students

Synthetic Tutor

Acknowledgement

I would like to gratefully and sincerely thank to my committee members, Takehiko Nagakura, Terry Knight, George Kocur, and Tina Grotzer, for their insightful guidance, patience, understanding and limitless support on my training as a researcher and a thinker.

I am also grateful to William J. Mitchell and Federico Casalegno for providing me with research opportunities at the MIT Design Lab and Mobile Experience Lab, and for supporting my research in the Computation Group throughout my time with the Department of Architecture. I would also like to thank the computation and architectural design faculty, especially George Stiny, Larry Sass, and Mark Goulthrope, for their friendship and mentorship.

I also want to thank my colleagues in the Computation Group and the Mobile Experience Lab. Without them, I would not have made it through the dissertation process. Thanks also go to Rizal Mouslimin, Derek Ham, Rachelle Villalon, Felecia Davis, Sotirios Kotsopoulos, Duks Koschitz, Woong Ki Sung, Vernelle Noel, Katia Zolotovskiy, Leonardo Benuzzi, Sergio Araya, Daniel Cardoso, Onur Yuce Gun, Simon Kim, Namjoo Lee, Carl Yu, Yang Yang, Orkan Telhan, Gaia Scagnetti, Kyoung Jun Lee, Moa Carlsson, Dina El-Zanfaly, and Miho Chu.

Thanks also go to Michelle Addington, Ron Witte, and T. Kelly Wilson at the Harvard University Graduate School of Design for their endless support and being my role models as researchers and architects.

Finally and most importantly, I offer special thanks to my soul mate Jong Jin Lee and son Henry Hyun Park. It was meaningful because of you. I thank my parents Seog-hwi Park, Jung-sook Oh, Won-gu Lee, and Soon-ok Kim for their unyielding devotion and love.

감사합니다. 아버지, 어머니. 언제나.

Synthetic Tutor

Table of Contents

1. INTRODUCTION	11
1.1 Overview	11
1.2 Study Procedures	14
1.3 Summaries.....	20
2. FIELD STUDIES	23
2.1 Educational Theories	23
2.2 Workshop Structure	26
2.3 Findings.....	31
2.4 Conclusions.....	43
3. MACHINE LEARNING	45
3.1 Machine Learning	45
3.2 Clustering / Unsupervised Learning	47
3.3 Classification / Supervised Learning	52
3.4 Image Recognition	54
3.5 Conclusions.....	62
4. SYNTHETIC TUTOR.....	63
4.1 Synthetic Tutor.....	63
4.2 Design-Scripting Education	67
4.3 Profiling and Mass-Customization	71
4.4 Computational Design Feedback	80
5. TESTS AND ANALYSIS	83
5.1 Methodology	83
5.2 Machine Learning Tutor	86
5.3 Computer Vision Tutor	104
6. CONCLUSIONS	113
6.1 Contributions.....	113
6.2 Discussion	116
6.3 Conclusions.....	119
LIST OF FIGURES	121
BIBLIOGRAPHY.....	123
APPENDIX A.....	129
APPENDIX B.....	131
APPENDIX C.....	655

Synthetic Tutor

APPENDIX D.....	657
APPENDIX E.....	659

1. INTRODUCTION

This thesis explores a computational method in which computers and humans work together to solve the complex problems associated with design education, problems which humans or machines cannot easily solve alone. I explore artificial intelligence (machine learning and computer vision algorithms) in which both humans and machines mutually improve their learning performance and increase their capabilities. Throughout this collaborative process, it is shown that machines can improve humans' learning by analyzing their behaviors and supporting customized needs; at the same time, humans can improve a machine's performance by providing increased amounts of data that can be a foundation for the machine's intelligent decision-making.

1.1 Overview

It is widely known that teaching computer programming language to novice students is not easy and that many students have failed to learn in introductory programming courses.¹ Many schools provide online computer programming courses, yet these courses are not always compatible with human learning styles. Students in architecture schools are not exceptions; they also experience difficulties in learning design scripting.

To solve this problem in computer education, I developed an intelligent tutor system called the 'Synthetic Tutor' that can teach computer programming and architectural design by utilizing machine learning and computer vision

¹ Every year about 650,000 students fail out of introductory computer programming courses (Bennedsen and Caspersen 2007).

algorithms. This study hypothesizes that the Synthetic Tutor will have a greater positive effect on the learning performances of its participants than would a non-machine learning tutor, which is currently the prevalent form of online education. With its artificial intelligence, the Synthetic Tutor can analyze participants' learning patterns and provide customized teaching materials. Tailoring and customizing teaching materials in order to make intelligent decisions requires large amounts of user behavior data, so as more students participate over the course of the experiment, the machine tutor will collect more learner-behavior data and its own performance will improve.

This thesis shows that the Synthetic Tutor effectively taught novice students introductory computer programming in the context of architectural design. These students learned how to write a computer program in order to design simple building typologies, and how to draw architectural documents for a residential building project. To review students' projects and provide feedback, computer vision algorithms are developed and tested as an initial attempt to develop a machine tutor that can review students' sketches and models and guide their further design directions as human instructors do.

Although computational technology has successfully extended the boundaries of architectural design, design communities have not taken full advantage of the potential of computational power in design education. The dominant educational method for teaching architectural design is still through person-to-person interaction: small numbers of pupils, guided by their instructors' critiques and suggestions, learn architectural processes by working on design projects. This instructional model, apprenticeship learning, has not changed much since its inception at the *École des Beaux Arts* in the 1860s.

Deviating from this conventional studio teaching method, this study

explores the potential of computational technology to utilize student-generated information in order to provide better education. One good example of utilizing user-generated data can be found in *Amazon's* recommendation system, which suggests additional books based on a user's matrix of past and current behaviors. The recommendation system consists of two filtering components: the first is user profiling, which predicts users' preferences by analyzing their book searches and purchase history. The second filter customizes the users recommendations based on an analysis of similar purchases made by other comparable customers. During design instruction, students also produce a large amount of observable data, including study time, learning patterns, sketches and prototype models.

Computational profiling and customization can solve the problems of design scripting education. A growing body of literature suggests that programming is increasingly valued for its use as a mental tool to enhance heuristic reasoning, abstraction, and decomposition, instead of computational execution (Bundy, 2007; Denning, 2005; Wing, 2006 and 2008). Many pioneers have also explored the potential of using computational power in architecture (Frazer, 1995; Knight, 1994; Mitchell, 1975, 1977, and 1990; Nagakura, 1990 and 1996; Negroponte, 1973 and 1976; Stiny, 1980, 1981, 1985, and 2006; Stiny and Gips, 1972). Their studies clearly show that the rapid development of information and computer technology have let architects experience a 'paradigm shift' (Kuhn, 1996) in design. Negroponte (1973 and 1976) and Mitchell (1975 and 1977) envisioned that designers and computational machines could work together and produce higher-level designs that could not be achieved by designers or machines alone.

However, regardless of the benefits of learning design scripting, I observed clear barriers in teaching design scripting in architecture schools. Unlike

the vast amount of research on the teaching and learning difficulties associated with computer programming, there has been relatively little research regarding programming education intended for design students who have minimal programming experience and relevant knowledge (Amiri, 2011; Burry et al., 2000; Celani, 2002 and 2008; Duarte, 2007; Krawczyk, 2008; Leitao et al., 2010; McCullough et al., 1990; Mitchell, 1977; Mitchell et al., 1987; Wurzer et al., 2011; Yakeley, 2000).

In addition to this research problem, few signature pedagogies and standard curriculums for design exist. Despite the uniqueness and tradition of the design discipline, relatively few programming courses for architects have followed the computer science teaching model. The design curriculum cannot provide enough courses and as much computational knowledge for their students as computer science curriculums do. This lack of research and courses may have excluded design and architecture students from a computer programming education that could benefit them.

1.2 Study Procedures

This study proposes the development of the Synthetic Tutor, including functions such as learner profiling and mass-customized instructions. Students' past and current learning records, such as a series of study patterns over time for given exercises, identify their learning styles and problematic content areas. The statistical analysis of a student's performances on certain design exercises that are effective for other students with similar learning styles makes it possible to offer customized pedagogic solutions for that student.

The developmental process of making the Synthetic Tutor included two educational studies: one qualitative and one quantitative. First, the qualitative

experiments informed my preparation of course materials and curriculums for teaching architecture students introductory computer programming, and helped me to understand students' various learning progress during their study of the provided materials. These experiments also helped me define a computational model of learning patterns and teaching models in later phases. Second, the quantitative experiments enabled me to measure the effect of the Synthetic Tutor at teaching programming as compared to a non-machine learning tutor.

1.2.1 Qualitative Studies: Field Studies

Before I set up a computational model for students and teaching, which would become an essential part of computational instruction, I conducted qualitative studies to understand the nature of design education in the context of an introductory computer programming course. This study also investigated the learning process of students, and it yielded a design-scripting curriculum for architecture design. It was comprised of three educational experiments that were conducted at MIT from the fall semester in 2011 to the fall semester in 2012.

A three-credit computational design workshop, using a project-based and architecture-design-studio-based teaching model, was provided in each semester. The workshop required three hours of lecture and six hours of assignments weekly. During the workshop, students attended lectures and submitted their daily programming code, their three modular projects, and their final projects. Additionally, all forms of correspondence during the semester were collected. Twenty-four students successfully completed the course and five students dropped.

Throughout these workshops, I gained knowledge and experience about students' learning styles while developing effective teaching methods of my own.

The collected data provided a foundation upon which I could establish a computational instruction system in the next phase. Detailed processes and findings are described in Chapter Two.

1.2.2 Quantitative studies

Synthetic Tutor

Based on my teaching experiences and observations of students' progress in the three workshops, I developed a computational tutor and teaching materials using *The Art of COMPUTER GRAPHICS Programming: A Structured Introduction for Architects and Designers* (Mitchell et al., 1987). The book offers historical examples of architectural design and computational processes that connect programming processes and architectural logic with geometric uniqueness. The authors introduce possible problems that could occur during the architectural design process and provide code samples to solve these problems. More importantly, they propose new possibilities in computer programming that extend the boundary of graphic programming into problem-solving procedures. These attempts to develop ways to teach computer programming languages to designers, along with the authors' association of coding structures with structures of architectural design, led me to choose this book as a primary source for my research.

Chapters three and four of *The Art of COMPUTER GRAPHICS Programming* describe the algorithms and processes used to develop an online tutor, giving particular attention to the exploration and implementation of machine learning algorithms. The tested algorithms include both unsupervised and supervised learning. The unsupervised learning could cluster groups of students based on their educational and programming backgrounds, while

supervised learning could customize tutorials using previously identified characteristics in students' learning as well as their real-time learning preferences.

Chapters three and four of this thesis provide information on how to profile participants' learning patterns and provide dynamically customized tutorials that include the normalization of variables and determination of the weight of parameters for user clustering and classification. Additionally, the work describes a computer vision system that can recognize architectural plans and their underlying algorithms and data structures. These systems include global and local feature-based computer vision algorithms and descriptions of how the algorithms calculate and identify image-data similarities.

After using the techniques described in these chapters to develop the Synthetic Tutor quantitative studies were conducted to measure the effectiveness of the tutor in teaching design scripting to architecture students. The first prototype was developed and tested with 87 participants in the summer of 2013. An experiment with two updated variations, one with machine learning algorithms and the other without, was conducted with 78 participants between May and June 2014. Figure 1 shows the different interactions between these participants and the two versions of the Synthetic Tutor.

Architecture students in the United States, Brazil, India, and Singapore participated in the experiments. Participants took the course for two weeks, and their learning patterns and performance changed over the period of the experiment. The main parameters analyzed were participants' studying time (in minutes), number of visits (in page views), sample codes, and project documents.

After the initial quantitative study was complete, a cross-sectional and a longitudinal data analysis were conducted. Overall, two analytical questions were asked: 1) How effective was the machine learning tutor compared to the non-

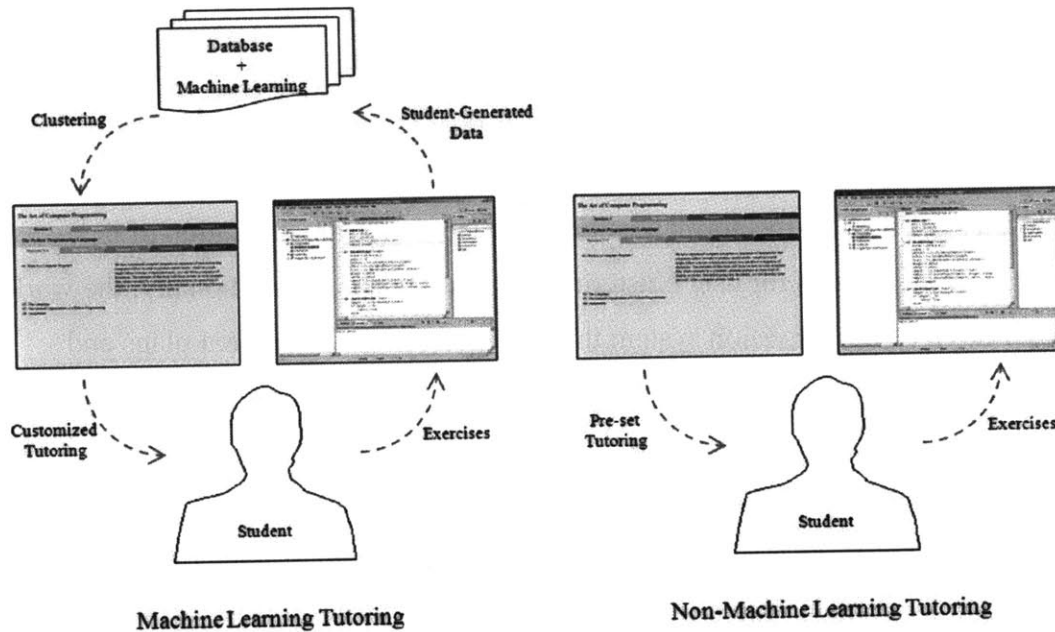


Figure 1. Machine Learning and Non-Machine Learning Tutorial Interactions.

machine learning tutor in terms of participants' learning performance as measured by their study time and the number of programming exercises submitted? 2) What is the feasibility of a computer vision tutor that is able to analyze participants' designs through their hand-drawn sketches or computationally generated architectural representations?

Computer Vision Tutor

Developing a machine learning tutor that can mass-customize instruction is the main work of this thesis. Additionally, this study explores a computational feedback system that utilizes computer vision algorithms to analyze and provide relevant information from a user's drawings. Computer vision-based architectural drawing recognition is the starting point for the computational design process;

these algorithms could be applied to various academic domains and industries.

The developed computer vision (CV) tutor system takes pictures of participants' drawings using a camera attached to a laptop and saves those pictures on a hard drive. The CV tutor then extracts local feature information from these saved images and compares that information with information gained from the training-data sets to calculate similarities among them. The CV tutor finally selects the most similar image in the training-data sets and then uses that image's metadata to search for relevant projects, which the tutor will then provide to the participant. This type of information is a popular kind of instructional feedback in the traditional architecture design studio, particularly during the initial conceptual design process.

Each training-data set includes 320 floor plans from 39 residential buildings designed by 15 different architects; all of this data is used to train the computer vision algorithm. This study developed an image-crawling algorithm to prepare these training-data sets. Image crawling is an automated process to collect image data; it utilizes the image search engines already installed in many search engines. This study can use its image-crawling algorithm to rapidly collect many residential projects and their metadata, which include architects, completion years, locations, and sizes. All of this metadata could then be applied as design feedback.

The main image recognition algorithm developed for the CV tutor is a local feature-based algorithm. The algorithm identifies a set of easily detectable pixels that have high contrast in their brightness or color values compared to that of surrounding pixels. It stores the differences in values in the form of a vector and uses that vector to calculate the 'distances' among images. Two images with a negligible distance are identified as identical. These vector-based transformations and calculations enable the algorithm to identify the semantics of images

regardless of their representational styles and sizes. For example, the algorithm can precisely detect the face of a person in various pictures, regardless of his or her skin tones, camera angles, and light conditions.

The CV tutor exhibits a 97% correct matching rate when it uses the training-data sets. Participants who use their own floor plans and sketches receive relevant architectural precedents from the CV tutor. However, due to the noise and distortions of image files that can occur when participants take pictures of their drawings, the CV tutor's matching rate may decrease. After each test, participants answered survey questions about the performance of the CV tutor and the effectiveness of the feedback they received.

1.3 Summaries

This study proposes a computational design instruction system, the Synthetic Tutor, that can seamlessly teach computer programming and architectural design. The Synthetic Tutor includes a machine learning (ML) tutor that can provide mass-customized teaching of computer programming languages for architecture students. Additionally, within a limited scope and implementation, the tutor can analyze a student's hand-drawn or computer-generated plans and provide that student with relevant architectural projects.

The findings of this study illustrate how collaborations between computers and humans are critical to successful learning. This study identifies methods for human-computer collaboration that will mutually improve both parties' capabilities. In this study, human participants improved their programming capabilities, while at the same time the Synthetic Tutor improved its teaching and customizing performance. Further, the ML tutor proved to be a more effective teacher of programming than the non-ML tutor.

This study has successfully explored computational methods for teaching architecture students design scripting; it has also established collaborative methods between humans and machine instructors for solving highly ill-structured educational problems. The process of developing the intelligent tutor - along with this study's subsequent findings - could be useful for educators who want to build a computational model of their instruction, and especially for those educators teaching difficult subjects or subjects that resist easy transformation into an online format.

Synthetic Tutor

2. FIELD STUDIES ²

The purpose of these qualitative field studies is to explore the educational theory and tendencies by which architecture students learn computer programming. There is plenty of research identifying the difficulties of learning computer programming in computer science schools. However, there are but a small number of studies concerning computer programming education in design contexts. There are even fewer studies in schools of architecture. Accordingly, it is necessary to directly explore the learning process of participants in introductory computer programming and architectural design. This study used a participant-observation research method.

2.1 Educational Theories

The main educational theory explored in these field studies is bricolage. Bricolage is a method of constructing a new object or solution for a problem by using available tools and materials on hand. Real-life examples of bricolage include using broken branches instead of chopsticks to pick up food, sitting on a flat rock instead of a chair, or burning dried leaves for fire while camping. Useful examples in the fine arts are Marcel Duchamp's 'Fountain' (1917) and Andy Warhol's pop art, including 'Marilyn Diptych' (1962) and 'Campbell's Soup' (1968).

In the context of architecture, Le Corbusier's projects, Christopher Alexander's design process, and post-modern architecture could all illustrate the

² Reprinted with permission from The Proceedings of the 2nd International Conference for Design Education Researchers - *Design Learning for Tomorrow*, 14-17 May 2013, Vol. 1, pp 144-155, Copyright 2013, DRS//CUMULUS.

meaning of bricolage. Rowe and Koetter (1984) explain Le Corbusier's design process as the selection and re-assembling of historical elements of architecture into new projects while identifying the project's contexts and redefining the found historical elements. Louridas (1999), meanwhile, has explained that anyone could use Alexander's design patterns to create buildings and cities that fit together as harmoniously as organically grown towns. Alexander's design process identified new design patterns through iterative manipulations of classic patterns. A design could evolve into a new norm by selective overlaying, juxtaposition, combination, and recombination of existing patterns.

In contrast to a modern architecture that utilized monotonous and hierarchical design, postmodern architecture allowed a single project to incorporate pluralistic design approaches and multiple design elements borrowed from historic buildings. Postmodern approaches freely manipulate the original meanings and functions of traditional architectural elements following the characteristics and the site contexts of a project (Louridas 1999). For example, *The Arthur M. Sackler Museum* (1985) by James Stirling displays pluralistic uses of historic elements sensitive to a project's environment. Due to these selective and pluralistic processes, postmodernism is considered bricolage.

Claude Levi-Strauss first described bricolage in his book, *The Savage Mind*, to illustrate the nature of mythological thinking. He explained that mythology was a result of human invention and is composed of previous human experiences. Thus, mythological stories are assemblies of elements from human lives that were re-composed and redefined within a new context. Levi-Strauss (1968) described a *bricoleur* as a problem solver in a primitive tribal society. He had few available tools and limited materials. In contrast, the problems that he needed to solve varied widely. Bricolage was probably a natural result in this

harsh environment. The essential process of bricolage was a dialogue between a bricoleur and his tools and materials. Through this conversation, he could re-conceptualize the purposes of tools and restructure the nature of materials so that they became useful parts of a solution for varying problem contexts within his confined situation.

For the purposes of this study, a bricolage method of instruction should allow diverse ways of learning to program. Much research has focused on students who are not familiar with canonical or hierarchical thinking and are experiencing ‘intellectual wars’ throughout the introductory programming course. These students are in essence forced to become another person, and these experiences lead them to refuse canonical instruction (Turkle and Papert 1990). Architecture students should be able to learn to program by following their mental models of programming as an extension of their design processes. For example, some students understand programming as city planning, some as a fabrication process, some as data processing, and some as the mathematical manipulation of pattern making. Design students may prefer to develop their own ideas instead of solving given problems.

In terms of learning processes, students should experience an evolutionary thinking that they cannot imagine through the dialogues between themselves and a programming language. Students need to organize and reorganize their design thinking and programming repeatedly. In terms of instructional method, students should vicariously experience coding. Accordingly, they can be asked to watch the instructor's live-coding, or the developmental process of an algorithm, instead of simply hearing the instructor’s explanations of its finished forms (McLean and Wiggins 2010). In bricolage instruction, students need to visualize or materialize their design thinking (Stiller 2009).

Lakoff and Johnson (2003) argue that metaphors structure the way people perceive and understand the world around them. This workshop uses the practice of computer programming as a metaphor by which to understand design and the creative design process. The incrementally iterative software design process will provide novice designers with a mental model of design as an evolutionary process, overcoming the prevalent waterfall model.

2.2 Workshop Structure

After surveying the theoretical backgrounds of bricolage and of educational experiments that utilize bricolage in programming education, six educational strategies have been identified to teach introductory computer programming to architectural design students:

1. **Object to Think With:** Students propose projects in which they transform their design ideas into programming language. In contrast to an instructor's typical problem set, student-driven projects increase the level of engagement and personalization. After students propose their initial design ideas, they will continue to develop concrete ideas throughout the workshop. Any radical changes in design ideas are welcomed and encouraged.
2. **Atelier Environment:** Students are asked to learn to program as if they were learning to sketch or paint in an atelier. Spending time programming is the most critical factor when learning to program, as is the case when learning to draw. Structured programming would still be introduced as a framework, as it is in a standard programming course; however, the use of structured programming would be suggested as a template, as if it were an empty

- sketchbook within which students could build their own codes.
3. **Daily Coding Exercises:** Cognitive changes require time. Instead of attempting to foster radical changes in mental models of programming, in this strategy incremental iterations are applied. Students learn programming through daily coding exercises instead of biweekly problem-sets or examinations. The exercises reduce the burden of ‘cognitive wars’ that many novice learners might experience (Turkle and Papert 1990).
 4. **Sketching and Diagramming:** To externalize students’ design thinking and to provide cognitive aids while developing a computer program, this strategy prompts sketching and diagramming exercises, which make students’ coding processes easier by externalizing their design and encouraging analytical understandings of their design thinking.
 5. **Limited Range of Programming Syntax:** This strategy calls for the selection of the Python programming language because of its unusually minimal syntax. Of that small amount of syntax, only the base forms of structured programming are taught. This limited syntax lessens confusion and allows students to reconfigure the meaning of programming syntax in the context of their projects. Students can re-conceptualize the purpose of programming while identifying new uses for the given programming language. Students learn computational concepts through trial-and-error approaches by repeatedly completing short coding assignments. Online reading material is provided for students who seek additional references.

6. Real-time Development: The instructor's real-time coding provides a chance for students to experience the development process of algorithms as if they were writing the code.

The main goal of these strategies is to accommodate students' varying learning styles so that they can have cognitively comfortable experiences while learning computer programming. The workshop consisted of three modules. The main content of the first module was procedural programming, and it included topics such as variables, functions, and structured programming. The second module taught object-oriented programming (OOP), and it covered class, instance, inheritance, and association. The last module taught the software development process using examples of biological systems such as cellular automata, the Lindenmayer system, and the flocking algorithm.

The study was composed of three single semester-long workshops and ran from September 2011 to December 2013 in the form of a series of computational design workshops at the Department of Architecture at the Massachusetts Institute of Technology. Each workshop consisted of two sections with 14 weekly classes that ran three hours each. The first section included a lecture explaining the theory of computer programming and related design theories. To promote design thinking, the lectures presented many architectural precedents that featured elements similar to computational design, such as the recursive garden design of the Taj Mahal (1653), the modular theory of Durand (Villari 1991), and contemporary projects using generative algorithms.

In the second section, the instructor explained programming practices with real-time coding to show computational concepts in demo codes. The instructional goal was to illustrate the developmental process of various

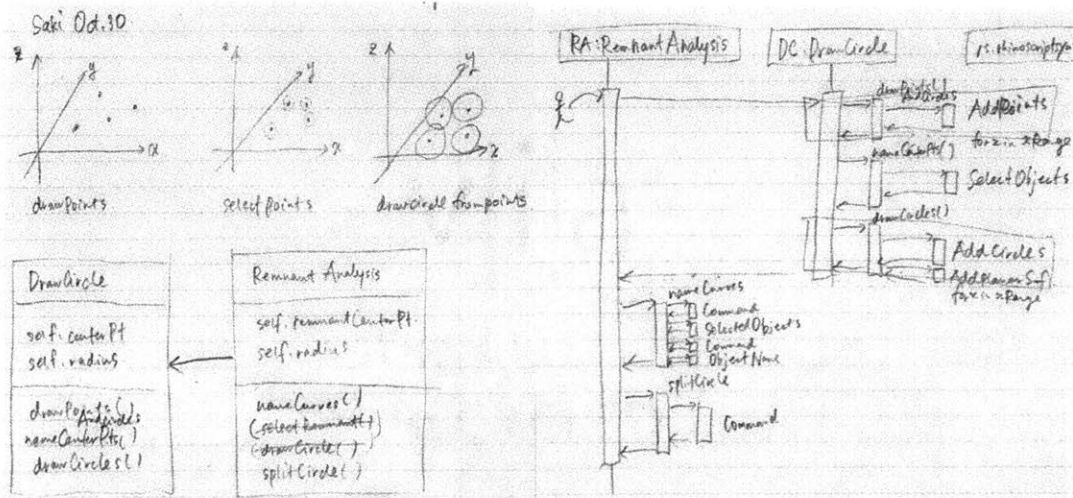


Figure 2. Example of a student’s daily sketches and unified modeling language diagrams.

algorithms, not to construct finished code. To achieve this goal, the instructor showed students demonstrations using developmental diagrams and analytical sketches. At the same time, students were prompted to create their own sketches and diagrams for software design.

After each class, daily coding exercises were assigned (Figure 2 and Figure 3). Students were asked to write short codes every day using a function that is the base module of structured programming. Novice students might take approximately an hour to complete these exercises. Decomposing an assignment from a medium-size program into small chunks of daily exercises helped students to identify their logical errors and programming mistakes more easily. Consequently, students could focus more clearly on their programming. This method prevented students from generating messy code with logical errors that exhausted both the instructor and the students, thus helping the instructor understand students’ work quickly and easily.

Synthetic Tutor

Module 1: Procedural Programming

student	student	student	student	student	student	student
9-20 Mon Sub1927.py		Tue1927a.py		Mon1927a.py		
9-27 Tue Sub1928a.py Sub1928b.py	Jed1928.py	Tue1928a.py Tue1928b.py	Ken1928a.py	Mon1928a.py		Marb1928.py
9-28 Wed Sub1929.py	Jed1929.py	Tue1929.py	Ken1929.py	Mon1929a.py Mon1929b.py Mon1929c.py		Marb1929.py
9-29 Thu Sub1930.py			Ken1930.py	Mon1930.py		Marb1930.py
9-30 Fri Sub1001.py			Ken1001.py	Mon1001.py		
10-1 Sat Sub1002a.py Sub1002b.py	Jed1930.py Jed1001.py Jed1002.py			Mon1002a.py Mon1002b.py Mon1002c.py		Marb1001a.py
10-2 Sun						Marb1001a2.py Marb1001a3.py
10-4 Tue Sub1004.py				Mon1004a.py	Rad1004a.py Rad1004b.py	Marb1004a.py Marb1004b.py
10-5 Wed Sub1005.py	Jed1004.py Jed1005.py			Mon1005a.py Mon1005b.py Mon1005c.py	Rad1005.py	Marb1005.py
10-6 Thu Sub1006a.py Sub1006b.py		Tue1006a.py Tue1006b.py Tue1006c.py	Ken1003.py Ken1004.py Ken1005.py Ken1006.py		Mon1006a.py	Marb1006.py
10-7 Fri Sub1007a.py Sub1007b.py				Mon1007a.py	Rad1006a.py Rad1007.py	
10-8 Sat Sub1008a.py				Mon1008a.py		
10-9 Sun		1009a.py 1009b.py 1009c.py 1009d.py		Mon1009a.py	Laure1009a.py Laure1009b.py Laure1009c.py Laure1009d.py	
10-10 Mon Sub1010a.py Sub1010b.py		1007a.py		Mon1010a.py		
10-11 Tue Sub1011a.py		1011a.py		Mon1011a.py	Laure1011a.py	Marb1011a.py Marb1011b.py
10-12 Wed Sub1012a.py Sub1012b.py Sub1012c.py			Ken1002.py Ken1003.py Ken1004.py Ken1011.py Ken1012.py	Mon1012a.py	Laure1012a.py	Marb1012a.py
10-13 Thu Sub1013a.py Sub1013b.py	Jed1004.py Jed1005.py Jed1006.py Jed1007.py Jed1008.py Jed1009.py Jed1010.py Jed1011.py Jed1012.py	1013a.py	Ken1013.py	Mon1013a.py		Marb1013a.py
10-14 Fri Sub1014a.py Sub1014b.py		1014a.py		Mon1014a.py	Laure1014a.py	

Figure 3. An archive of students' daily coding exercises in the first workshop. This data shows that those students who consistently submitted daily exercises had higher levels of programming capability than students who submitted their exercises sporadically. The length or amount of submitted code influenced students' learning curve less than their frequency of submissions.

The overall purpose of these assignments was to let students incrementally develop a computational model of their non-computational design ideas. They completed small parts of their original design within the range of their limited programming knowledge. Students were first asked to develop an error-free skeletal code, one that was almost empty but still had logical functions that could illustrate the main algorithm of the program using only their limited programming syntax and fluency. Students were then asked to redevelop their code while in the process gaining more programming knowledge (Figure 4).

From the bricolage perspective, students need to bi-directionally redefine their design thinking and their programming structures. On the one hand, they might need to extract only small chunks of the original design and complete code that could be tested quickly. On the other hand, students might selectively re-organize certain parts to test the overall computability of their original design.

Students' programming fluencies were measured in two ways. First, the instructor evaluated students' code to measure whether they could purposefully organize a program using functions, classes, and data structures, and whether they could add comments appropriately. Second, students' final design projects were evaluated to measure whether students could successfully transform their design ideas into programming structures.

2.3 Findings

In total, eighty-five students applied for the three workshops, twenty-nine students were randomly selected, and twenty-four students successfully completed the workshops. These students came from a wide spectrum of academic and professional backgrounds. One had taken an introductory programming course three years ago, and two had minor experience with the visual programming

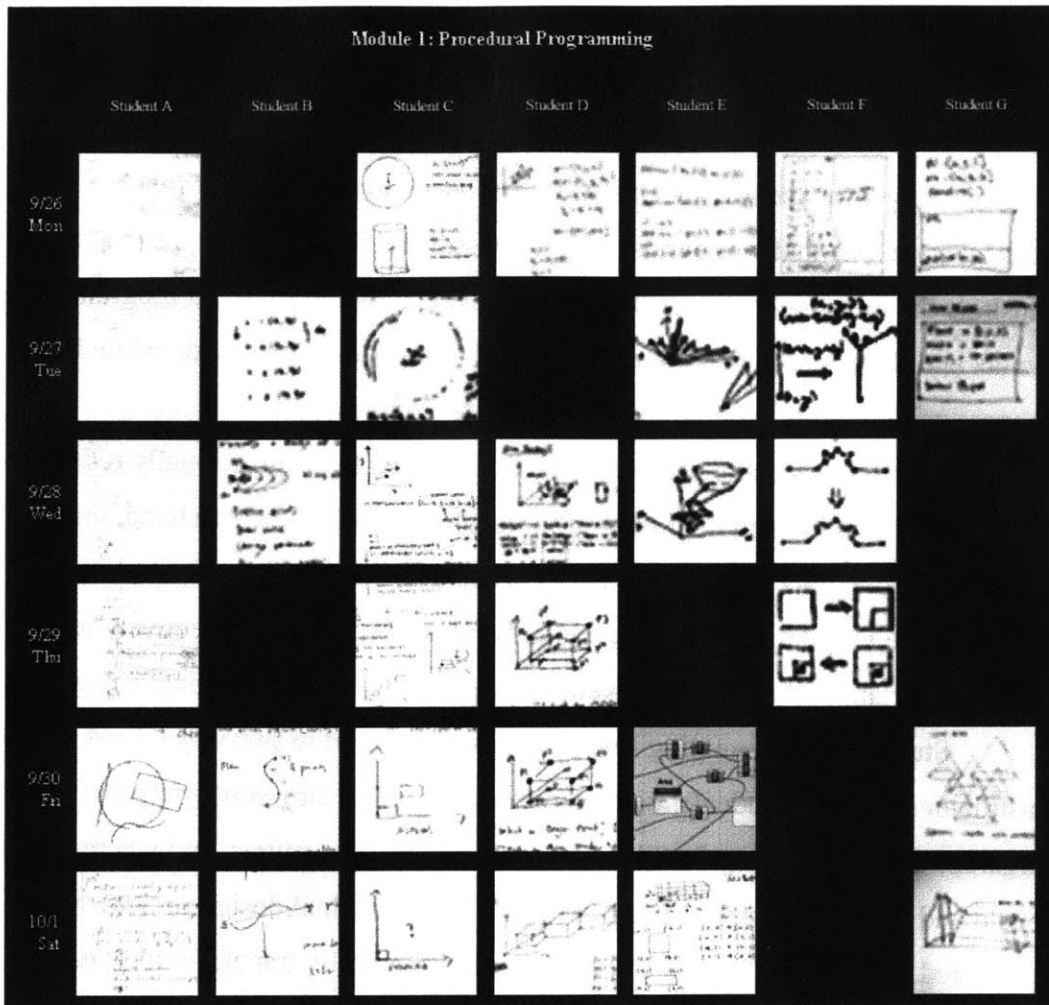


Figure 4. Example of students' daily sketches and diagrams in the first module.

language, Grasshopper. Most students had no background in Python, the target programming language.

In each workshop, most students successfully completed the course, yet at least one student dropped in the middle of each semester. One student stated that the main reason for dropping was her lack of experience using the 3D modeling

software, *Rhinoceros 3D*³, which was the main tool for programming. Another student explained that she had dropped the course because the unfamiliarity of the software substantially increased the difficulty of learning the language.

It is widely reported that novice programmers find object-oriented programming (OOP) difficult to learn (Bennedsen and Caspersen 2007). However, most students involved in the study successfully learned the ideas of abstraction and modularity and wrote working code for their design projects. Two students gained almost instant insight into modular thinking on the first day of learning OOP. It appears that a concrete understanding of modular concepts in procedural programming might improve their ability to learn OOP.

While students naturally showed varying learning processes, their learning curves were highly dependent on the frequency with which they completed coding exercises. A positive relationship between the frequency of code submissions and the number of lines of code was observed. The more frequently students submitted their code, the more lines they could handle; ultimately, students showed they could handle many hundreds of lines of code (Figure 3).

2.3.1 Advantages

Students displayed an increased level of engagement when programming student-driven projects. A direct relationship between students' personal motivation and their development of programming skills was clearly observed. One student used her programming exercises to work on her master's thesis project, which enabled her to focus on her own work; she showed a consistent increase in the complexity and sophistication of her programming. Another student worked on jewelry making, which significantly increased her level of engagement throughout the

³ Rhinoceros 3D is a computer-aided design tool for 3d modeling and visualization. The software's multi-language scripting tools include Python, Visual Basic, and Grasshopper.

workshop. Although physical models were not necessary, she fabricated her project, even going so far as to create two additional projects, neither of which were required. Her motivation drove her to continue her work outside of the boundary of the workshop. This use fabrication exercises as tools to motivate students became a formal exercise in the third workshop.

Another student worked on his architecture design studio project, a simulation of the internal patterns of the urban condition in a building. The number of lines of code he could handle successfully exceeded three hundred, which is impressive, especially considering that this workshop was his first programming course. Most importantly, he extended his coding without increasing its complexity. He later told the instructor that the daily coding exercises with OOP were highly effective and conceptually clear, allowing him to continue his work without difficulty.

For novice students, conventional biweekly problem sets can be difficult. Even when problems are designed for novice programmers, the size of the problem often is still too large for many new students to easily solve. Students' coding may have accumulated syntax and/or logical errors, and their approaches to the problem vary and are not straightforward. The weekly feedback during office hours or lab sessions often comes too late to be timely and may cause frustration for students. Accordingly, students spend many hours trying to identify trivial errors, such as typos and misuses of syntax.

The daily exercises eliminate these errors. Students' learning processes becomes highly effective, and they can speed their progress. Some students need to change their projects frequently while their understanding of programming



Figure 5. Three Students' Four Projects Developmental Processes. Three students' examples show their project developments over three modules (module 0 shows the students' initial project proposal).

increases. These students reconfigure their learning goals and the scope of their projects considering their limited time and programming knowledge. At the same time, some students extended their understanding of programming from this workshop to other classes and combined multiple course projects successfully.

The process of externalizing students' thinking through sketching and diagramming effectively corrects defects in students' code and helps them to identify logical errors. The unified modeling language (UML) was especially useful to students struggling to understand the complex relationships between their programming code and their design ideas. Many students identified and corrected their own errors and were able to independently extend their code while drawing UMLs. Diagramming and analytically sketching ideas made the process of transferring design thinking into programming structure both smooth and transparent.

While students' programming backgrounds were different and their learning curves covered a wide spectrum, their small daily exercises improved their learning processes and freed the instructor to devote more time to supporting the students' diverse learning styles (and less to correcting their coding errors). Understanding the various styles of students' learning, design thinking, and programming logic is a time-consuming process. These varying cognitive processes could not be managed without modular and timely feedback.

2.3.2 Disadvantages

Students who were familiar with canonical thinking showed conceptual misunderstandings of the bricolage-based structure. They needed to have concrete final goals even before they started the workshop. It appeared that they had difficulty understanding the concept of the evolutionary process and could not

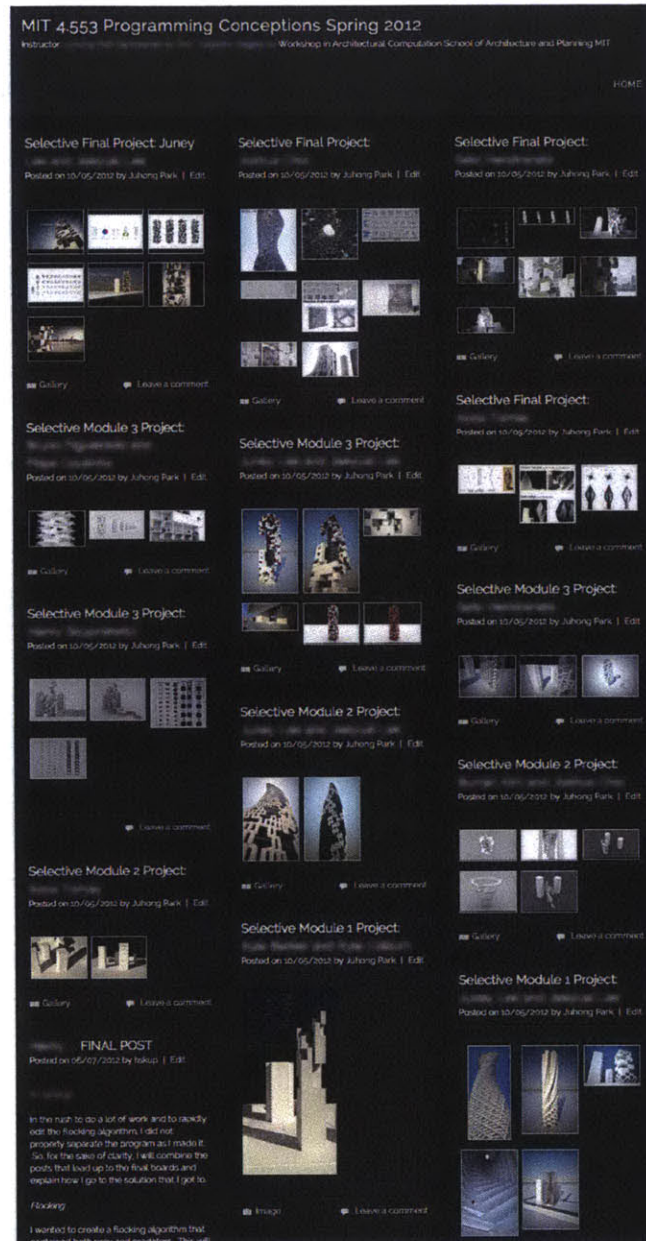


Figure 6. Students' Project Examples.

These are examples of students' projects during the second workshop in the spring of 2012. The workshop used an architectural design (a high-rise residential building in Cambridge, MA) as a main class project. All students' projects and codes were collected on a class blog that externalized and visualized students' programming processes.

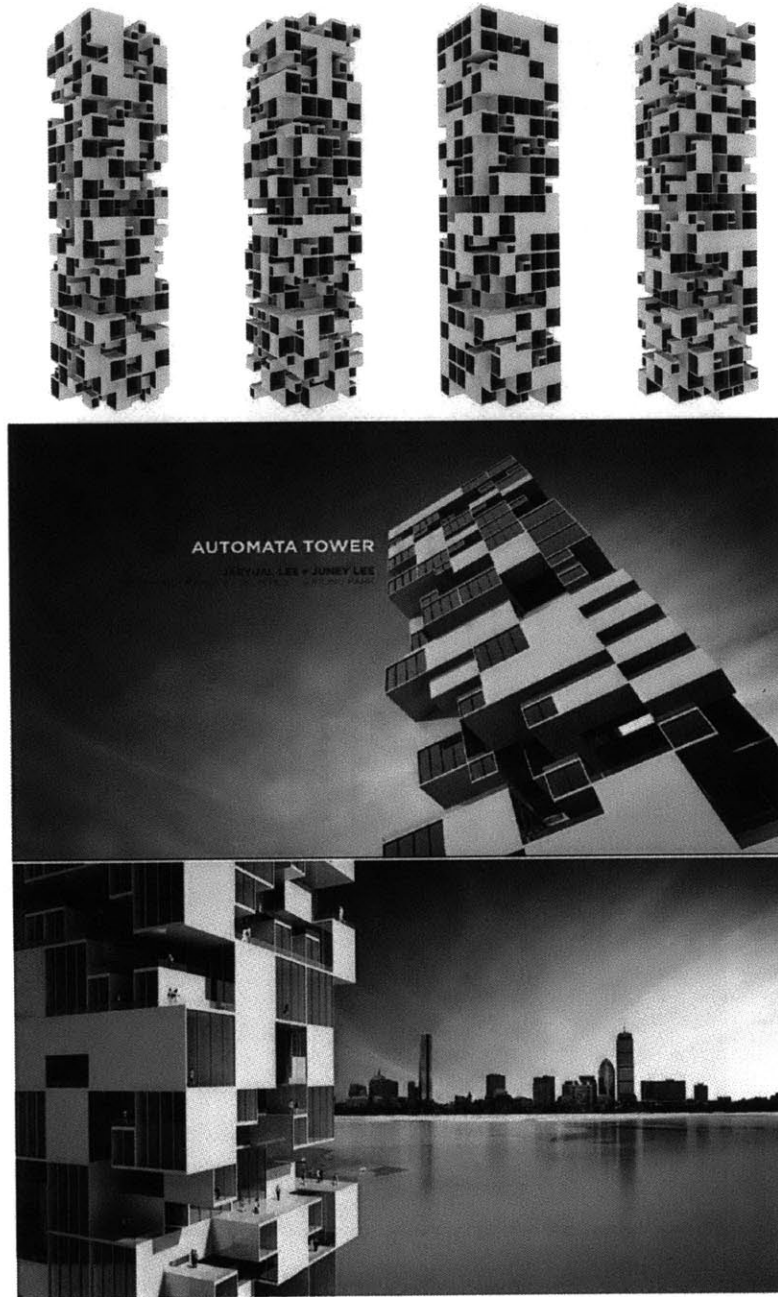


Figure 7. A sample final project completed in the second Spring 2012 workshop. These images show different perspectives of a high-rise building that was designed and generated by using a cellular automata algorithm.



Figure 8. The project archive of the third workshop in the fall of 2012. The third workshop used fabrication as a learning tool, which was inspired by a student who showed a high level of motivation in the first workshop.

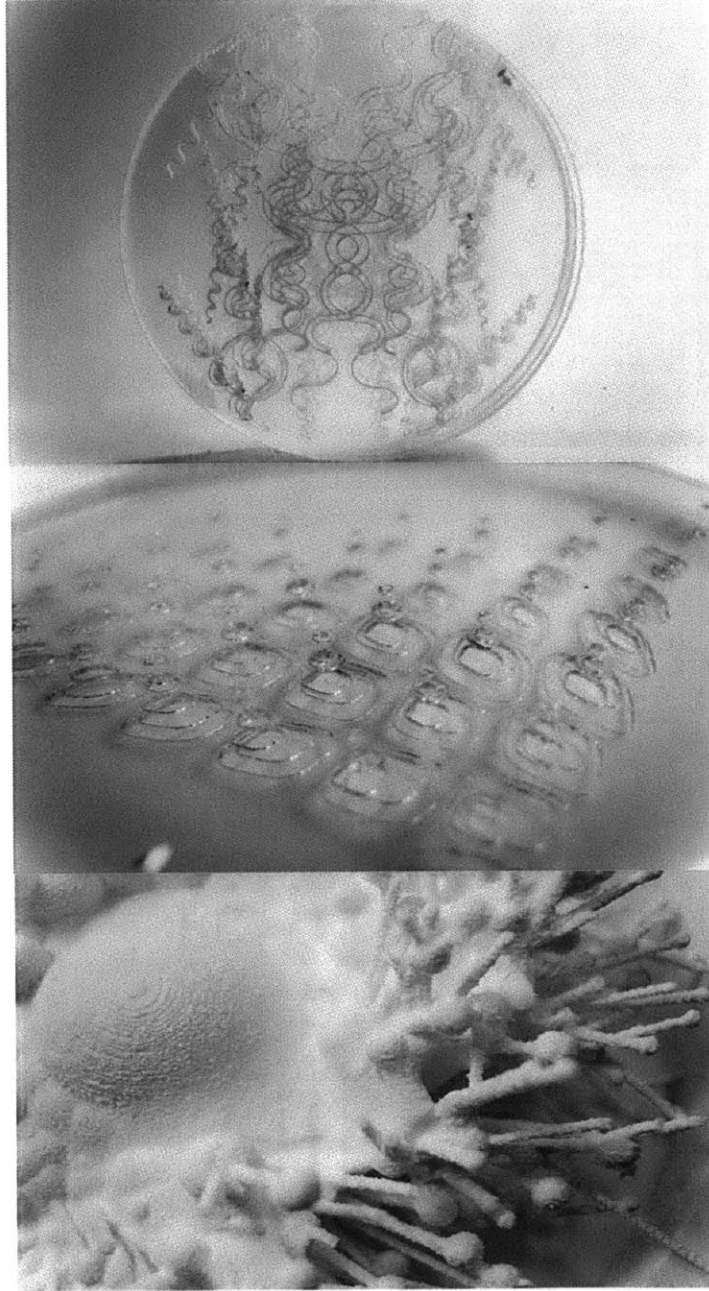


Figure 9. Examples of three students' projects in the third workshop. Students developed algorithms that mimic organic forms found in deep-sea fish and bone structures.

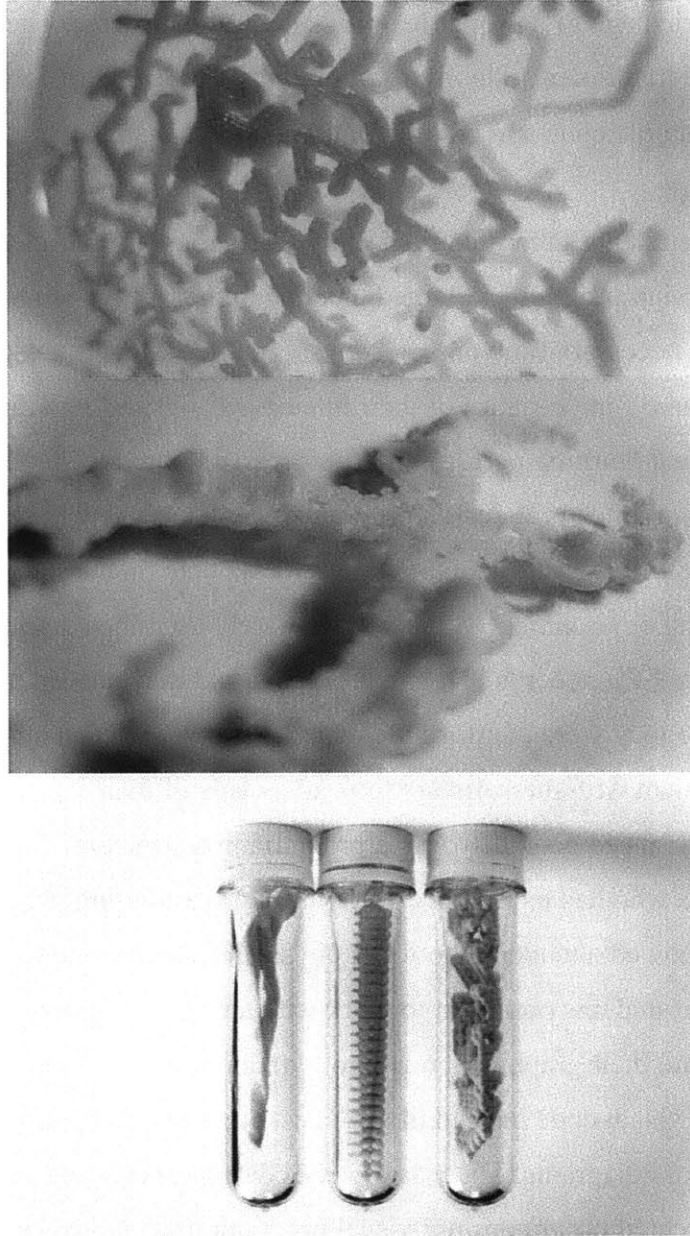


Figure 10. Examples of three students' projects in the third workshop. The top project was to create a bacterial-like growth pattern using the Lyndenmeir system. The middle one aimed to develop a 3D printed deep-sea creature. For the bottom project, a student made a physical visualization of 1) Boston climate data, 2) his own weight information from his first semester, and 3) the cellular automata algorithm, from left to right.

easily accept bricolage as a proper learning and design process. These students, who expected a traditional style of programming lecture, spent more time searching for sample code and reading online tutorials than they did directly learning by doing.

Not all students submitted their daily codes regularly. Students in the architectural design program explained that due to their heavy studio workload they lacked the necessary time on weekdays. First-year students, especially, experienced a hard time because of their intensive core studio requirements. Accordingly, their learning performance progressed relatively slowly compared to students in non-architectural design programs.

Students' initial learning curves were dependent on their familiarity with the programming environment. Students who rarely used the computer-aided software package, Rhinoceros 3D, had problems, and they needed additional exercises before they were comfortable with the software. Other students who were already fluent with the software took advantage of their knowledge by transforming advanced geometry into a programming structure.

Students who had experience with visual programming languages such as *Grasshopper* showed a tendency to resist physical sketching and diagramming exercises. Some students preferred to work with a visual programming language and then translate their graphical code into Python. Some students actually submitted their visual code instead of sketches. For them, the graphical codes *were* sketches and diagrams. These tendencies disappeared when they started to learn object-oriented programming; visual programming languages do not support OOP, and thus translation no longer works. Although this habit of translation slowed some students' acquisition of OOP (as compared to those students who did not have any programming experience), these students' previous

programming knowledge enabled them to accelerate their learning processes during later stages.

2.4 Conclusions

Several questions have been identified. The first is how to extend this pilot study into a standard full semester-long course that could be offered as a part of any architecture school's curriculum. It is necessary to consider additional computer science contents. Fundamental algorithms, numerical modeling, and simulation may all be reasonable fits. The second is how to collaborate with traditional architectural design studio education. The design-scripting instruction could be independent, as it is in other technology and history classes, or it is possible that architecture schools will create collaborative computer programming design studios.

Although developments in online and mobile technology make the time-consuming management of students' code a minor issue, dealing with students' varying learning styles is still not an easy problem for instructors. There is no dominant way to understand the design process, and the spectrum of students' design thinking is extremely wide. The standard engineering and science school-based, so-called problem-solving approach to teaching computer programming may not be an effective way to teach design-school students.

The use of a bricolage approach significantly improved novice programmers' learning of computer programming. The reconfiguration and externalization process of students' various design work and the iterative development of programming structures made student progress transparent, and incremental iteration is at the center of this success. Daily exercises compress debugging and make the error-finding process quick and easy while reducing the

burden on instructors by permitting easy detection and correction of students' errors.

Stiller (2009) used a bricolage approach to introductory programming in a computer science department, but she did not recommend it for students in that kind of department. However, the successful results of this study recommend the inclusion of bricolage in introductory computer programming courses in design schools. Bricolage has historically been used for art and design education, and accordingly it might fit well in architectural design education. Design, as Alexander (1964) maintained, is a problem-setting and problem-defining process. The bricolage approach to teaching computer programming may be a good solution for this unique domain.

3. MACHINE LEARNING

This chapter provides detailed descriptions of the machine learning and computer vision algorithms that are surveyed and utilized in this study⁴. Machine learning is a discipline focusing on developing a system that can learn as humans do. The term 'machine' is used to distinguish the system from living things. I define learning as a process that changes the inner state of a machine or a system and enables a machine or system to do something that it was previously unable to do. I specifically discuss two aspects of learning in this section: acquiring new knowledge from unknown (i.e., unsupervised learning) and improving proficiency based on knowledge already acquired (supervised learning).

3.1 Machine Learning

Learning is generally applicable only to intelligent living animals, including humans (Odaka, 2012). Machine learning (ML) is a branch of artificial intelligence that focuses on developing a computational model of a human-like learning system that can generate behaviors using collected empirical data, such as user interactions and real-time sensor inputs. The goal of ML is to develop an algorithm that by identifying relationships among given complex data can update itself and continue to improve its performance in new environments (Alpaydin, 2004). Examples include algorithms that distinguish spam from emails, select potential clients from previous customers, and control vehicles without human drivers by analyzing real-time camera images.

⁴ These machine learning and computer vision algorithms were developed for this study based on the '*Learning-Library-for-PHP*' (<https://github.com/gburtini/Learning-Library-for-PHP>) and '*OpenCV*' (www.opencv.org) libraries.

Since Rosenblatt (1958) first developed an artificial neural network, Perceptron and Samuel (1959) used the term ‘Machine Learning,’ the ML field has become an essential component in many areas, from manufacturing and robotics to bioinformatics and biometrics. The convergences of statistics and artificial intelligence in the 1980s, and emerging information technologies and data-mining in the 1990s led to a rapid expansion of the field of ML. Computer vision, natural language processing, medical diagnosis, and brain-machine interfaces are just a few ML applications that indicate a bright future in science and engineering (Negnevitsky, 2004).

ML is a relatively new approach in both architectural design and computational learning. Few studies have been conducted that link ML and design education. In the field of educational technology, intelligent tutoring systems (ITS) are considered an ideal instructional model, yet they cause highly challenging problems (Corbett et al., 1997). Most critically, learning is not a single process; rather, it has multifaceted problems, and students have diverse motivations and goals. The range of learners’ cognitive styles varies from subject to subject and changes over time while learners’ cognitive capacities are improving. Additionally, students’ learning patterns are more complex than instructors’ teaching styles (Turkle and Papert, 1992).

Implementing learner modeling involves a great deal of time and substantial data complexity. By collecting learners’ empirical data through channels such as sensor inputs and user interactions, analyzing and categorizing users’ learning patterns, and providing customized feedback in real-time, ML shows a high potential to provide solutions for customized tutoring and individualized education.

However, the majority of research in the field of ML still focuses on

improving the performance of machines, not that of humans. Even though the development of intelligent tutoring systems, especially in linguistics, mathematics, and science education, has been researched for a long time, the use of ML for improving human creativity in arts and design has only recently been highlighted. In arts and design, the focus has been mainly on the role of ML in improving human learners' engagement (Denis, 2004; Mills and Dalgarno, 2007; Vlist et al., 2008) and lessening learning difficulties (Morris and Fiebrink, 2011; Patel, 2010; Widmer 2005).

This study utilizes two machine learning algorithms. The first algorithm identifies students' learning patterns using clustering (unsupervised learning). It finds a pattern of relationships among students' learning behaviors. The second provides a customized tutorial utilizing classification (supervised learning). This algorithm specifically uses the results from clustering to determine the appropriate content to provide for each group of learners.

3.2 Clustering / Unsupervised Learning

Clustering (unsupervised learning) may solve one specific issue that this study encountered: understanding and identifying users' learning patterns without advanced knowledge of their learning styles. This useful algorithm aids in the discovery of unknown patterns that reside in data sets. For example, retail stores can find daily, monthly, and seasonal sales patterns by analyzing their annual sales database. Based on these analyses, store managers can order especially popular items in varying seasons, targeting different age groups and income levels within their clientele.

In this study, clustering was used to identify groups of students who showed similar learning patterns. In the later phases of the study, this information

on learning patterns could be used to provide customized tutorials. Not many design schools teach their students introductory computer programming, and there are few studies on students in architecture schools teaching computer programming; for these reasons, data about these students is difficult to find. Accordingly, this study uses an unsupervised learning algorithm to group students based on their learning patterns without requiring a pre-existing set of data. As a first step, I prepared an initial online workshop to collect participants' behavior information. This experiment was conducted between May and July 2013 at MIT. 86 students from MIT, Harvard University, and Wellesley College joined the workshop through online announcements using a student email list.

I included in my data collection parameters such as participants' daily study time in terms of minutes, their programming and math educational backgrounds, the number of modules they completed, and the number of sample codes they submitted. Based on the analysis and clustering algorithms, I identified five unique learner groups, and this result was used by the classification algorithms to classify participants in the second workshop. Identifying participants' learning patterns requires the collection of user behaviors, the analysis of these behaviors, and the identification of groups of students with similar learning patterns. Before participants begin the tutorial, they answered questions on: 1) educational history, 2) motivation level, 3) previous computer programming experience, 4) any computer science courses a participant took in the past or is currently taking, 5) any college-level mathematics courses a participant took or is currently taking, and 6) their fluency in the 3D software (Rhinceros 3D).

Collectable information on participants' learning patterns measured throughout the workshop includes: 1) a participant's online visiting time, 2) time

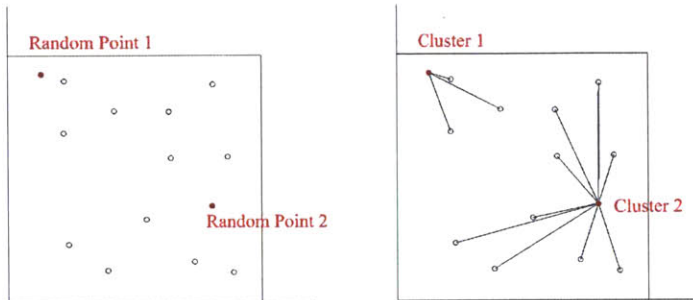


Figure 11. The initial clustering procedures of k-means algorithm. The k-means algorithm first generates random center points (left) and conducts clustering based on the center points (right).

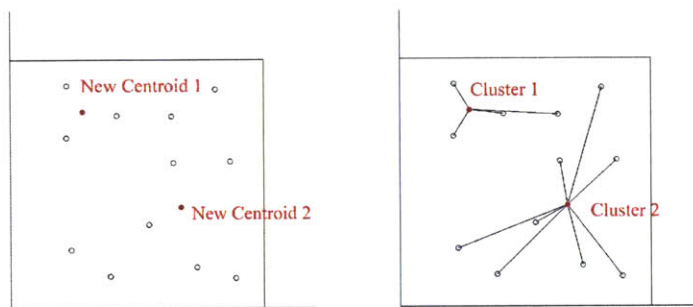


Figure 12. Computing centroids of initially clustered data. After the initial clustering, the algorithm generates updated centroids of data (left) and re-computes updated clustering (right).

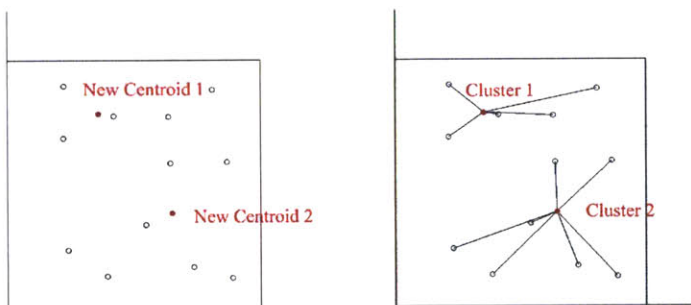


Figure 13. Repeated procedures to compute updated centroids of k-means clustering. The k-means algorithm repeats the process in Figure 12 until updated centroids converge.

spent for each tutorial, 3) whether a participant studied provided sample code, 4) participants' submitted assignments, 5) users' comments on each tutorial, and 6) users' evaluations of each tutorial.

Clustering identifies unknown patterns in a data set by comparing statistical similarities of data values. At first, it uses a present value of k to generate k -number of random center points and computes k -clusters by calculating the distances between collected data and the center points (Figure 11 left). From each data point, the algorithm calculates the distances between the data and each center point, and then it assigns the data to a closest center point (Figure 10, right). After this first iteration, the algorithm computes new centroids for the k -clusters and recalculates the distances between data and centroids (Figure 12). It repeats this process of generating new centroids and computing distances until updated centroids converge (Figure 13).

The k -value increased with the number of participants over the course of the experiment. This increased k -value implied improvements in the precision of clustering of users and identification of users' learning patterns. A clustering with a high k -value does not always mean a precise analysis; rather the finding of k -value with low error would be the target goal of an unsupervised learning algorithm.

This algorithm is able to calculate an unlimited number of variables and identify unlimited groups. In this study, the algorithm computes eight variables⁵ from initial survey questions that quantify users' educational and programming backgrounds (Table 5) in order to initially group participants into k -value groups.

⁵ The survey questions are attached in Appendix A.

Variable	Unit	Data Type
ID	Number	Integer
Program	Number	Integer
Graduate/Undergraduate	Number	Integer
Year	Number	Integer
Programming Experience	Number	Integer
Previous Programming Courses	Number	Integer
Previous Mathematics Courses	Number	Integer
3D Software Experience	Number	Integer
Available Daily Study Time	Number	Integer

Table 1. Initial survey question variables.

Based on these values, the clustering algorithm initially divides participants into k groups.

Variable	Unit	Data Type
ID	Number	Integer
Day 1 Study Time	Minute	Float
Day 2 Study Time	Minute	Float
Day 3 Study Time	Minute	Float
.	.	.
.	.	.
.	.	.
Day 15 Study Time	Minute	Float
Number of Studied Modules	Number	Integer
Exercised Daily Codes	Number	Integer
Daily Assignments	Number	Integer
Final Project	Yes/No	Boolean

Table 2. User behavior variables and their data types collected throughout the workshop. The k-means clustering algorithm uses the normalization values of this study-behavior data.

It then clusters participants by using their behavior data acquired throughout their tutorial experience (Table 6), excluding the independent variable (ID).

3.3 Classification / Supervised Learning

While the above experiment provides information about both participants' behavior parameters and corresponding labels (i.e., group information), the second experiment uses previous participants' behavior data and classification algorithms to identify students' behavior patterns in real-time. Students' learning patterns, including study time, visit numbers, and coding work, are parameters that define students' learner groups. At the beginning of the second experiment, the ML tutor did not have enough data about the students and accordingly it could not recommend customized tutorials. Data from the previous experiment was used to provide customized tutorial sets for participants who joined the study in the early period of the second experiment until the ML tutor could collect enough current student behavior data.

The classification of participants by their learning patterns and their corresponding tutorial contents could support each specific learning style. This customization process (matching sets of tutorials with an appropriate group of participants by using pre-collected data) is known as supervised learning. This second experiment uses classification algorithms alongside supervised learning algorithms. Supervised learning is like learning from an instructor; in this kind of learning, an instructor teaches both labels and related information together, such as the name of an animal and its characteristics. Before it can execute, this algorithm requires a set of pre-acquired data, called a training set, which includes both pre-analyzed parameters and their values.

Using the training set, supervised learning algorithms compute labels for

the newly acquired data. For example, an algorithm used by real estate agents might calculate the value of a house that is newly on the market by comparing the number of rooms and the properties of its address with those of other houses, and then approximating proportional differences from similar houses.

This calculation method, finding an approximate value or category of a new instance by comparing instances of similar properties, is called ‘classification’. The simplest method in statistical modeling to cluster data is ‘regression’. In the case of a sales-decision problem, a manufacturing company may want to send advertisements only to customers who had previously purchased their products, have a certain range of income, or specific types of family members—that is to say, those who are highly likely to buy their products in the future. By analyzing the similarities of their customers’ properties, this company would be able to label each potential buyer and focus the company's advertisements on them.

Spam mail filters provide us with a similar and popular example. The filter compares the properties of emails and then labels them as spam or normal mail. The algorithm first needs a set of emails that are pre-labeled as spam. When a new email arrives, the filter compares each word in the email. For example, a sample spam email may have words such as ‘free’, or ‘coupon’. The filter may need another set of samples, non-spam email, which includes words such as ‘birthday’, ‘special’, ‘prefer’, and ‘want’. A new email with words likes ‘free’ or ‘offer’ could then be labeled as spam.

One simple linear parametric model could be described as below:

$$Y = a X_1 + b X_2 + c X_3 + \dots + j X_N, \quad (1)$$

where Y is the probability of an email being a spam, $X_1 \dots X_N$ are 1 or 0 when an

email includes a corresponding word, and a_j through p_j are the probability of a word which could be in a spam mail. Although many practical models use non-linear parametric models, this linear model could calculate Y . When the training set is small, the calculation of a probability (Y) may have a high rate of error. As the size of training sets increases, the error rate will decrease and the performance of the filter will increase.

In the first experiment, the clustering/supervised-learning algorithm computes information to identify participants' learning patterns and recommend tutorial contents. In the second experiment, this profiling information then becomes a training set for the post-customization process using the classification/supervised-learning algorithm.

3.4 Image Recognition

This study required the development of an automated method for providing instructor-like feedback on participants' designs. In an actual architectural design studio, participants' sketch plans, sections, and perspectives while instructors offer commentary on their work: any corrections on circulation issues, ideas for daylighting or ventilation solutions, suggestions for architectural references, and questions for further consideration. All of these feedback processes require the common initial step of recognizing the meanings of the participants' drawings. Such tasks as understanding shapes and colors, reconstructing 3D geometries, and recognizing objects in image data can be processed and automated by using computer vision technology. Computer vision research is also concerned with the technologies used to extract meaningful patterns from image data, and to reconstruct 3D spatial information from 2D pixel data.

The extraction process in computer vision includes methods for collecting

and pre-processing raw images; pre-processing includes re-sizing, enhancing, and manipulating an image data set. This thesis especially focuses on developing computer vision algorithms that can extract spatial and structural meaning from architectural data, such as plans, sections, or perspectives. In the workshop experiment, computer vision algorithms are used to analyze image data gathered from participants' projects.

CV research goes back to Roberts' study of machine perception in 1963. It has since then developed core technologies for image processing, and its contemporary theories are consistent with recent advances in machine learning, statistics, physics, and robotics. The discipline focuses on teaching a machine how to see things and understand their meaning. Accordingly, the technology explores methods of developing computational models that mimic human perception and recognition systems; this is accomplished through research on the retina and the optic nerve in the eye and the cerebral cortex of the brain. Computer vision applications include (Szeliski, 2010):

1. recognition of faces and objects in imagery,
2. extraction of coordinates from moving objects,
3. tracing motions from a series of films,
4. diagnosing symptoms and diseases from medical images, and
5. searching for identities from hair shapes, skin colors, and clothes.

This study aims to develop an algorithm that can understand participants' designs and provide feedback on their design progress. This algorithm should be able to compare participants' architectural drawings to existing architectural

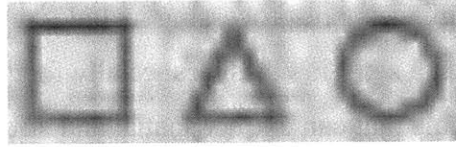


Figure 14. Three shapes as two-dimensional array data types.

drawings. The algorithm should find the existing drawings with the greatest similarity to the participants' drawings, and then provide participants with their matching drawings' metadata.

This study started by exploring global feature-based CV algorithms and extended their range into local feature-based algorithms. Global feature-based CV algorithm, such as Euclid distance-based pattern matching, could recognize overall shapes in image data. Local feature-based pattern matching algorithms analyze image data at more detailed levels, such as patterns of rooms, walls, and windows. The developed local feature-based CV algorithm showed outstanding performance in distinguishing and comparing various architectural plans in various graphic styles. The next section will describe in further detail the machine learning and computer vision algorithms mentioned here.

3.4.1 Euclidian Distance-Based Image Recognition

One simple way for a machine to find similarities among images is k-means clustering. Clustering, as described above in Section 3.2, is a method for finding similarities among instances in a data set with no access to pre-defined data. This algorithm assumes that the dimensions of all images are the same and finds statistical similarities by comparing the values of pixels at the same location in each image. If the images are not the same size, it becomes necessary to pre-process the images in order to map their varying sizes into one common size. For

example, two gray images with a 10 x 10 resolution each have 100 pixels, and each pixel has a value of between 0 and 255 (for an RGB image, each pixel would have three color values). In other words, both gray images have a 10 x 10 matrix with a value ranging between 0 and 255. When a k-means algorithm compares multiple images, it computes each numeric difference between corresponding pixels, and then it calculates the sum or average of all differences. Finally, the algorithm would select the two images with the smallest differences.

Figure 14 shows three images with different shapes. The data type for the three images could be understood as a two-dimensional array. In these images, the values of pixels are zero when the brightness of a pixel is black and 255 when the brightness of a pixel is white. The average distance between two images is calculated as the average of the summation of the Euclidian distances of each pixel in two images.

This algorithm, which calculates similarity using Euclidian-distances, is a fundamental process for advanced computer vision algorithms built to compare complex images. It can effectively recognize primitive shapes, such as triangles, rectangles, or circles, and its underlying calculations are easy to understand. However, although the algorithm shows good computational performance and a low rate of error, it is not an effective method for recognizing complex architectural drawings that include many sub-elements.

3.4.2 Local Feature-Based Image Recognition

Euclidian distance-based image recognition can effectively recognize an overall similarity of images, yet it cannot compare partial components within images. For example, the algorithm may efficiently compare primitive shapes, yet it shows high error rates in comparing multiple elements within images, such as mixtures

of rectangles and triangles against mixtures of rectangles and circles. To compare multiple elements of varying sizes, rotations, and positions in images, the machine tutor needed more sophisticated algorithms than Euclidian distance-based image recognition. To recognize architectural drawings, which can include plans, sections, and perspectives, the machine tutor needed an algorithm that could recognize various elements within drawings, such as rooms, walls, windows, and doors in plans; or stairs, entrances, and ceilings in sections.

The Synthetic Tutor required an algorithm that could compare the similarities in partial images in order to recognize participants' architectural drawings. In computer vision algorithms, these partial images are called local features. Computer algorithms can define local features in many different ways. The simplest method is to use a region with a predefined size and to then sequentially check all these partial regions. For example, if there were an image with 100 x 100 pixels and a predefined partial region-size of 10 x 10, then the algorithms would check the similarities of all 10 x 10 sized images from the top left corner to the bottom right corner, region by region.

One practical algorithm first finds several unique points in an image (called 'interest points') that have high contrast in brightness or color, and then selects a surrounding area as a local feature, instead of selecting all partial regions as local features. This method can significantly reduce the computing load by limiting the areas compared. However, the algorithm used to select unique interest points significantly influences the overall algorithm's performance when recognizing images.

3.4.3 Interest Points

Not all pixels in an image are of use when describing the characteristics of the

image. As such, instead of considering all local regions in an image, it is effective to find a number of unique parts that are critical to understanding an image and to use those unique parts in a comparison. These critical pixels are called ‘interest points.’ There are various algorithms that specify and select these interest points in an image. A pixel whose values differed greatly from those of the surrounding pixels would be a good candidate for an interest point. These value differences are frequently observed at the corners of shapes, including both primitive geometries (such as rectangles, triangles, and hexagons) and building plans, sections, and perspectives.

The algorithmic process to calculate interest points is as follows:

1. For each pixel in an image, select the surrounding 8 pixels: three on the row above, two on the same row (on the left and right of the selected pixel), and three on the row below. If there is no row above or row below, as would occur at the edge of an image, then the value of those missing pixels is considered equal to the value of the selected pixel. This assumption prohibits those pixels on the edge of an image from being interest points.
2. Calculate the differences by comparing the values of the selected pixel with all 8 surrounding pixels.
3. Select the highest difference value among the eight values and mark the value as the highest variance of a pixel.
4. Sort the pixels based on variance.
5. Select the pixels that have variance within the top 25%.
6. Collect the coordinates of these pixels as the interest points. An interest point thusly calculated defines ‘local features’ in the

following section.

3.4.4 Local Features

Local features are partial images that are used to compare the similarities of objects in different images. They are distinctive parts of images that represent the characteristics of objects. These local features are robust indicators used to detect general objects (e.g., cars, people, airplanes, or trees). Detecting general objects is difficult because such objects vary in terms of scale, rotation, position, color, and brightness. To compare images while considering these variants, rather than using direct color or location values, this local-features based CV algorithm first calculates the changes in values in the form of a vector. For example, the vector of changes in colors or brightness makes the detection of an edge of a rectangle easy, even when the rectangle appears in varying locations, sizes, and colors.

The change in brightness values at the corner of a black rectangle on a white background ranges from 255 (white) to 0 (black). A computer vision algorithm can easily detect this high contrast in brightness in both the right and downward directions from the rectangle's top left corner. These two directional changes are maintained even when the size and position of a rectangle are changed and its orientation has rotated; the corner can be easily detected even when the rectangle's color has been changed. This numeric representation of graphic information in the form of a vector is called a 'feature vector'. The algorithm to calculate feature vectors is as follows:

1. For each interest point, the algorithm makes a descriptor that consists of three by three pixels centered on the interest point. The descriptor is composed of 8 radial arrows (vectors) centered on the interest point (north, north-east, east, south-east,

- south, south-west, west, north-west).
2. The algorithm determines the scale of each arrow by calculating the differences in brightness or color values between the interest point and the surrounding pixels.
 3. For each descriptor, the algorithm includes 8 directional vectors, accordingly becoming 128 dimensional vector data (4 x 4 x 8).
 4. The algorithm stores the descriptors of all interest points in an array or a list.

3.4.5 Scale Invariant Feature Transform (SIFT)

Lowe (1999) first developed an algorithm that could identify objects when they were projected in 3D space, as can neurons in the human inferior temporal cortex. Global feature-based human face recognition algorithms (which directly compare faces to other faces) performance poorly when attempting to recognize general objects. Therefore, many face-detection algorithms in current smart cameras and social network services use local feature-based image recognition. A popular example is a face recognition algorithm that compares sizes and colors of facial elements, such as eyes-to-eyes, nose-to-nose, and mouth-to-mouth comparisons. Since this algorithm is able to detect an object regardless of changes in size, rotation, and brightness, it is called the Scale Invariant Feature Transform (SIFT) algorithm.

SIFT compares the local features of objects that have been collected into vector features. The input data is the matrix (or the list, when the matrix is flattened) of vector features; the matrix represents the changes in colors or brightness in 8 radial directions from a given interest point. The differences in

local features in two different images can be calculated in terms of distances, which is similar to Euclidian distance-based clustering.

3.5 Conclusions

This chapter describes the technical algorithms of machine learning and computer vision used in this thesis. Two machine learning algorithms (clustering and classification) are described, and two computer vision algorithms (Euclidian-distance and local feature-based image recognition (SIFT)), are illustrated. The two machine learning algorithms are used to profile participants' learning styles and customize tutoring materials, and the two computer vision algorithms are explored to analyze image data from participants' projects and provide design feedback. In the following chapter, I will describe how these four algorithms are used in the developed Synthetic Tutor, and also how the tutor was operated during the workshop.

4. SYNTHETIC TUTOR

This study proposes a Synthetic Tutor, a computational design instruction system, which can teach computer programming and architectural design seamlessly. The synthetic tutor provides a bricolage instruction in which participants iteratively exercise design scripting to generate architectural design solutions and receive feedback on their solutions. To provide these instructions and feedback, the Synthetic Tutor has two computational tutors: the machine learning (ML) tutor and the computer vision (CV) tutor. The ML tutor consists of design scripting and architectural design exercises to teach designerly ways of programming. The CV tutor provides feedback on participants' design projects after each exercise. This chapter describes algorithms and procedures for developing and testing the ML tutor and the CV tutor. It describes how the ML tutor understands the educational needs of each learner and customizes teaching materials. It illustrates how the ML tutor updates its instructional recommendations throughout a learner's workshop period. The illustration includes the composition of the developed CV tutor explaining the human-machine collaborative learning process to recognize a participant's submitted design project and provide human-like design feedback.

4.1 Synthetic Tutor

The Synthetic Tutor was developed over the course of two experiments. First, I developed online teaching materials and conducted a workshop experiment in which I evaluated the effect of those materials on participants' learning; this occurred between May and July of 2013 (Figure 15 and Figure 16). After

Synthetic Tutor

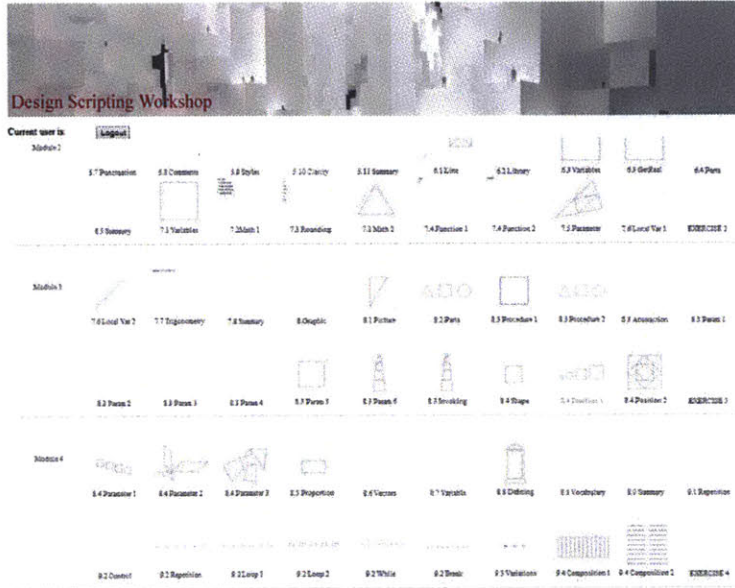


Figure 15. The main page of the online workshop, which shows the first 30 units. The workshop contains teaching materials, including daily exercises.

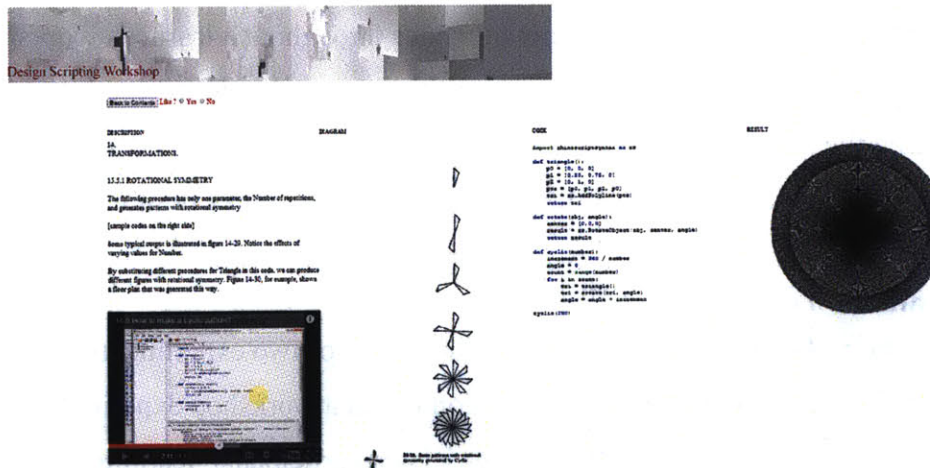


Figure 16. A sample page explaining repetition. Each unit includes short descriptions, a video-lecture, sample codes, and images that show the results of running the sample codes.

completing that workshop, I surveyed participants' reviews, analyzed participants' learning patterns, and developed supervised learning algorithms to analyze data for profiling (clustering) participants' learning styles and educational needs.

Then, from May to June of 2014, I conducted the second experiment using the machine learning tutor and the non-machine learning tutor. In this second experiment, I applied a classification algorithm that used student behavior data from the first experiment to calculate the input parameters for the profiling algorithms.

During these two experiments, the participants followed these overall procedures:

1. **Initial Registration:** A participant registers her user name and password. This process prevents the collection of any identifiable personal information.
2. **Background Survey:** A participant starts the tutorial by completing an introductory background survey. The ML tutor uses the survey results to identify a participant's initial educational needs.
3. **Working with Tutorial Sets:** Based on a participant's initial profiling, the ML tutor recommends for that participant a set of color-coded tutorial modules (out of 200 sets available). Red-titled modules indicate that participants with similar learning styles liked that module⁶. Violet-titled modules indicate that participants spent a sufficient amount time studying that module. Blue titles indicate that users almost skipped that module entirely. A gray-colored title indicates that the participant has already completed that module. It should take a participant an hour each day to study a set of

⁶ 'Like' and 'Dislike' buttons are provided for learners to provide their feedback.

recommended modules. Each tutorial module contains a partial section of the book (*The Art of Computer Programming*), and using various teaching methods (including texts, images, diagrams, and sample codes) the module focuses on explaining only a single computer programming concept. A participant can complete a module within 5-10 minutes. Teaching modules have varying lengths and levels of difficulty. The ML tutor produces various combinations of these teaching modules with various levels of difficulty and provides them to a participant according to his or her preferences and learning performances.

4. **Daily Progress:** There is no guided instruction for participants to follow. On each day, a participant may follow her preferred schedules and freely learn on her own. A participant's learning behaviors (including study time, the number of visits, whether a participant works with the provided sample codes or skips certain modules, and their evaluations of the provided modules) are collected for analysis and become critical data for further learner customization.
5. **Analysis of Learning Patterns:** The ML tutor analyzes how a participant studies the tutorial. The tutor checks whether each module is completed or not. The tutor compares the length of time a participant spent on a module with the average time other participants spent, and it recognizes whether a user spent enough time to study the module or if the participant skipped ahead. The tutor also remembers how many times a participant visited each module to repeat it. The number of visits could be used to identify

whether a participant likes the module. The ML tutor collects data from many participants by using a supervised learning algorithm and compares each participant's learning behaviors with other participants' behaviors.

6. The machine learning tutor uses the collected data from the first experiment to sort all participants into five learner groups. By using a classification algorithm to compare the values of user parameters such as study time, the number of visits, user ratings, and the patterns of these values, the ML tutor can identify the participant's learning style and educational preferences and match the participant with a similar group of participants.
7. Repetitions of steps 3 - 5: A participant repeats this customized tutoring process throughout the workshop or until he or she decides to stop. For as long as the participant is attending the workshop, the machine learning tutor will continue to recommend dynamically selected and customized sets of tutorials based on its analysis of his or her learning patterns.

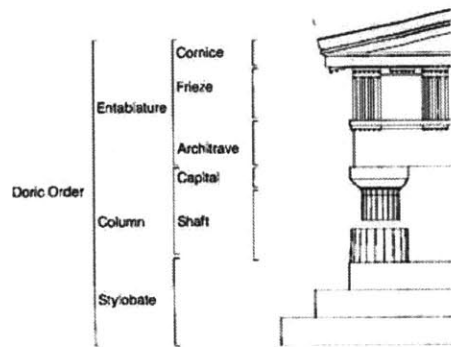
4.2 Design-Scripting Education

When an experienced instructor explains a difficult concept to his/her students, that instructor may at first attempt to explain the concept by invoking a widely known definition. The instructor will then try to identify whether his/her students understand this definition by watching their eyes and body language, and by

12.
HIERARCHICAL STRUCTURE

EXERCISES

1. All of the elements and subsystems of the Doric order have names (fig. 12-33). Draw a tree diagram that depicts this hierarchy. Then write a program, structured in the same way, that generates the order. (Simplify the details where necessary to reduce the task to manageable proportions.)



12-33. Well-defined hierarchy of elements and subsystems—all with traditional names—in the Doric order.

Figure 17. A sample problem set given to participants.

This image shows a sample workshop exercise in which participants can learn programming structure by using a historic architectural example.

looking for changes in their facial expressions. If the instructor sees signs that his/her students are having some difficulty understanding the concept, the instructor may repeat the explanation, or immediately change teaching strategies in order to try to explain the concept with examples, easy descriptions, or the use of metaphors that the student may more readily relate to. If the instructor sees signs of confidence in a student, then they may instead provide more difficult exercises and accelerate their teaching to challenge the student's intelligence. If the instructor encounters signs that a student's learning is slowing down (i.e., that the student may be having difficulty following the material), then the instructor

9.4.1 INCREMENTING A SINGLE POSITION PARAMETER (continued)

It is common for building codes to limit the heights of buildings by specifying the maximum angle `Max_angle` that can be formed at the center of a street (fig. 9-17). As an architect, you might be particularly interested in the maximum floor area that you can fit on a site. Let us assume that the floors of our building are rectangular. The following interactive program reads in values for floor `Length` and `Width`, floor `to_floor` height, and for the constraints `Max_angle` and `Street_width`, then draws the building section and displays the total floor area. Notice the use of two functions: `raster`, which converts feet to raster units so that the section can be drawn to appropriate scale on the screen; and `Max_height`, which calculates the maximum allowable height for given `Street_width` and `Max_angle`. Here is the complete code:

[sample codes on the right side]

Some typical output is illustrated in figure 9-18.

This program introduces an important new idea. There is a function called `Total_area`, which calculates the total floor area of a building and is invoked after the building is drawn. This is an analysis function and is executed to tell us something useful about the object that has been drawn. So the structure of our program is essentially as follows:

```
def declare_function():
    Read values of independent variables
    Calculate values of dependent variables

    Draw the design

    Perform analysis
```

This is not just a graphics program, then; it is a simple example of a computer aided design program. It assists the designer not only by rapidly drawing the building, but also by automatically performing some of the problem solving that is necessary before the building can be drawn, then by automatically performing some of the analysis that is necessary after the building has been drawn. This allows a very rapid trial-and-error design process, as shown by the flow diagram in figure 9-19.

Figure 18. A sample section of workshop modules.

This sample module illustrates an algorithmic approach to architectural design and a trial-and-error design process.

may begin spending more time with a single lesson.

These acceleration and deceleration processes are likely to occur during one-on-one instruction with an experienced instructor. It is easy to observe such versatile teaching practices in everything from physical and sports education to education in music and the arts. It is, however, uncommon to see this synthetic instruction from computer-aided tutors.

CODE	RESULT
<code>import rhinoscriptsyntax as rs</code>	
<code>import math</code>	
<code>def drawRectangle(x,y,length,width):</code>	
<code># calculate vales for x2 and y2</code>	
<code>x2 = x + length</code>	
<code>y2 = y + width</code>	
<code>pt1 = [x,y,0]</code>	
<code>pt2 = [x2,y,0]</code>	
<code>pt3 = [x2,y2,0]</code>	
<code>pt4 = [x,y2,0]</code>	
<code>rs.AddLine(pt1,pt2)</code>	
<code>rs.AddLine(pt2,pt3)</code>	
<code>rs.AddLine(pt3,pt4)</code>	
<code>rs.AddLine(pt4,pt1)</code>	
<code>def getMaxHeight(street_width,max_angle):</code>	
<code>radians = 0.01745</code>	
<code>max_angle = max_angle * radians</code>	
<code>angle_factor = math.cos(max_angle) * math.sin(max_angle)</code>	
<code>max_height = (street_width/2) / angle_factor</code>	
<code>return max_height</code>	
<code># draw floors of highrise</code>	
<code>def drawHighrise(x,y,</code>	
<code>length,thickness,</code>	
<code>floor_to_floor,street_width,</code>	
<code>max_angle):</code>	
<code># calculate max hight of building</code>	
<code>max_height = getMaxHeight(street_width, max_angle)</code>	
<code>total_height = max_height - thickness</code>	
<code>height = 0</code>	
<code># loop to draw floors</code>	
<code>while (height < total_height):</code>	
<code>drawRectangle(x,y,length,thickness)</code>	
<code>height = height + floor_to_floor</code>	
<code>y = y + floor_to_floor</code>	
<code>drawHighrise(100,100,500,20,100,1000,60)</code>	

Figure 19. A sample code showing a generative approach to the design of a high-rise building.

The main content of the Synthetic Tutor came from *The Art of COMPUTER GRAPHICS Programming: A Structured Introduction for Architects and Designers* (Mitchell et al., 1987), drawing most heavily on the second part of the book, ‘Elementary Graphics Programs.’ The book is filled with historical examples of architectural design and computational processes that connect programming processes and architectural logic with geometric uniqueness. The authors introduce problems that could occur during the architectural design processes and provide sample programming codes to solve them. More

importantly, they propose an algorithmic design approach that extends the boundary of graphic programming into problem-solving procedures (Figure 18 and Figure 19).

The book, however, was written twenty-seven years ago, and the main programming language, Pascal, is not widely used in the current architecture community; hence, in the tutorial some outdated chapters were removed and replaced with material teaching a relatively recent educational computer programming language, Python. Following the original book's main theme, though, the proposed tutorial's primary goal is to teach computer programming for the architectural design process.

4.3 Profiling and Mass-Customization

To initially identify learner types, the machine learning tutor uses participants' survey results. The tutor identifies patterns of similar backgrounds within the survey results, looking closely at as participants' current programs, the programming and mathematics courses they had completed, and previous experience with programming and 3D CAD software.⁷ In order to identify these groups without any previously collected data, the machine learning tutor uses a clustering algorithm. The algorithm first calculates the distance between participants' data points, and then it classifies those participants based on the distances between them.

The group number (k-number) begins at one and, over the course of the workshop, increases with the number of participants; as the k-number increases, the tutor can use a greater amount of user data. The first workshop had eighty-seven participants. From these participants, I identified five distinct groups. In the

⁷ The survey questionnaire is attached in Appendix A.

second workshop, I expected to have a similar number of participants⁸ and set the k-number at the power of 2.2 so that the second workshop could also show similar clustering and classification results. The resulting five learner groups were as follows:

1. Group 1: The first group is comprised of extraordinary users. In the entry survey, they stated that they had a high level of motivation. Participants in this group submitted most of the provided exercises; however only a small number of participants (five out of seventy-eight) completed the final architectural design project.
2. Group 2: Users in this group studied consistently and spent five times longer than participants in Group 3. These participants completed most modules and assignments. They spent a relatively long time working on each module, as well as overall. Considering their completion rate, customized learning might not improve the performance of this type of participants; however, it might accelerate their learning curves.
3. Group 3: Participants in this group started off working hard, but after the first two or three days their study time had begun to diminish. This group of participants completed many modules, yet many did not complete the tutorial. Participants in this group might increase their study time, and they could complete the tutorial with proper customization.
4. Group 4: Participants in the fourth group showed a similar

⁸ Some participants did not provide their login names and some identifiers were lost. Due to this missing data, only data from seventy-eight participants was analyzed.

pattern to those in the third group but displayed lower performance in many aspects. Participants in this group worked steadily, studying each module carefully and completing assignments at the end of every chapter, and yet they rarely increased their performance. Interestingly, some participants in this group elongated their study time and became Group 3. However, other participants in this group decreased their study time and developed similarities with Group 5.

5. Group 5: Participants in this group quit studying the workshop even before they finished the first chapter. One possible cause could be that the workshop did not meet their expectations. In this group, I observed one unusual participant who completed the whole workshop in two days. This person had visited all the modules, yet the average study time for each module was less than one minute. It would be highly desirable for this study to develop an ML tutor capable of motivating participants to shift from Group 5 into Group 4, or from Group 4 into Group 3.

Clustering effectively finds unknown patterns among data elements in studies such as this one, where the field of research (computer programming in design education) has only recently begun and few findings exist. Clustering algorithms can identify unknown groups of participants who show similar learning styles and performances. By analyzing the tutorial contents and participants' behaviors, the machine tutor can customize tutorial contents that may improve participants' learning performances.

Similar to search engines like Google, which measure the credibility of a webpage by its reference numbers, and similar Amazon's recommendation engine,

which uses past customers' purchases and product evaluations to establish current customers' purchasing patterns, the machine tutorial uses the learning patterns of previous participants who have showed similar learning patterns to current participants.

The machine learning tutor recognizes whether a participant likes a tutorial module when:

1. a participant evaluates a teaching module highly,
2. a participant spends more than the average amount of time on a teaching module, or
3. a participant visits a teaching module multiple times.

Even though multiple people may learn a computer programming language by reading the same book, they learn through different components of that book. Some participants like to read the textbook from the first page to the last; some like to skim through the book first and then check its detailed contents only after seeing the whole picture; some prefer to work on parts selectively, such as sample codes and experiential coding; some search book chapters for their projects; and some only use the book to compare their knowledge of other programming languages.

For this study user profiling and mass customization starts with an analysis of the book that is the basis for the workshops. The workshop material has different combinations of varying elements, and participants' preferences have shown what combinations of tutorials work together most effectively. Identifying a participant's preferences and customizing the combinations of teaching materials for that a participant does not comprise a static problem.

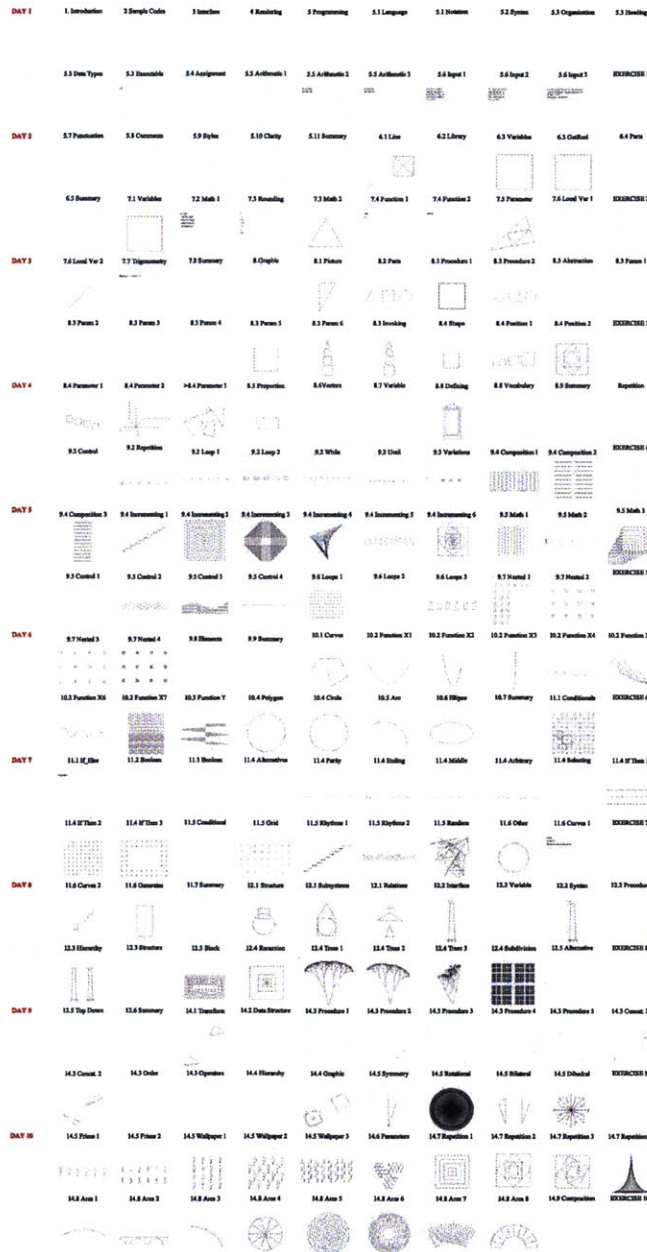


Figure 20. Screen-captured images of the non-ML tutor.⁹
 The non-ML tutor, which simply delivers the tutorial contents.

⁹ Appendix B includes the workshop materials.

Synthetic Tutor



Figure 21. Screen-captured images of the ML tutor (Day 1). The image shows an example the ML tutor’s recommendations for a participant on his first day of the workshop. He was one of Group 3 participants. Highlighted contents were: 5.3 Data Types, 5.7 Punctuation, and Exercise 2.



Figure 22. Screen-captured images from the ML tutor (Day 3). The image shows an example the ML tutor's recommendation for a participant on his third day. Newly highlighted contents included: 9.2 Loop 1, 9.4 Incrementing 3, and 9.7 Nested 1. Recommendations from the first day have disappeared.

Synthetic Tutor

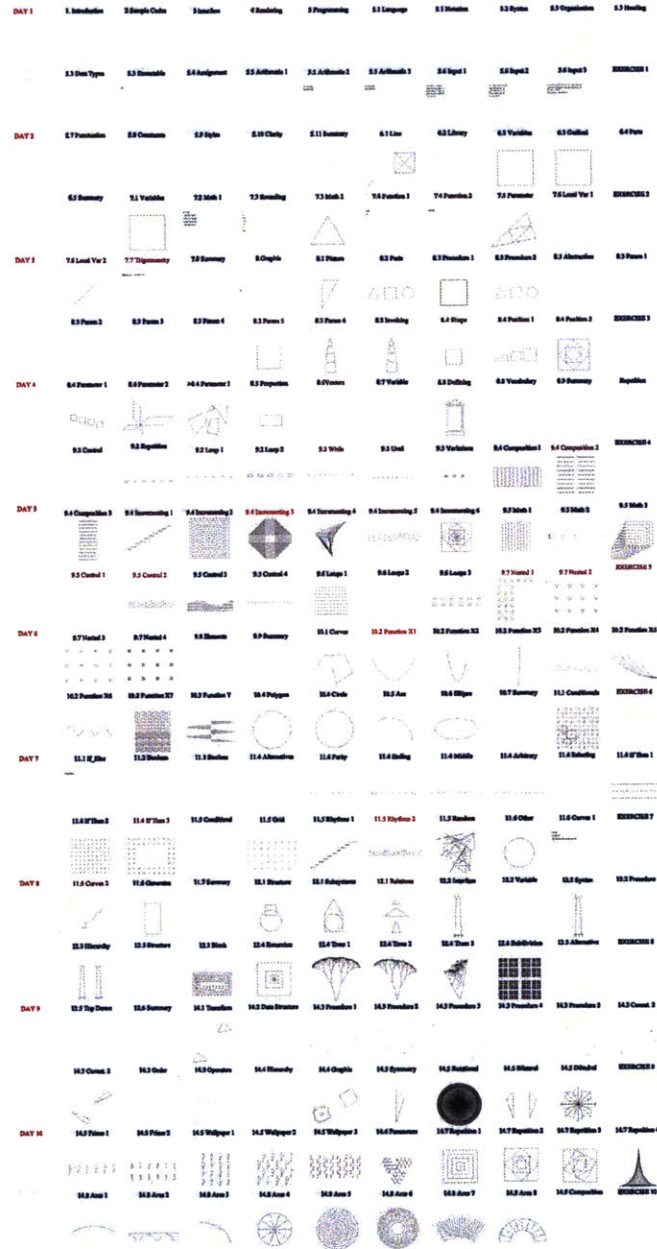


Figure 23. Screen-captured images from the ML tutor (Day 6). The image shows an example the ML tutor's recommendations for a participant on his sixth day. There were only minor changes. Recommendations included: 11.4 If Then 3, 11.5 Rhythm 2, 11.6 Curves 2, and 12.1 Relation.

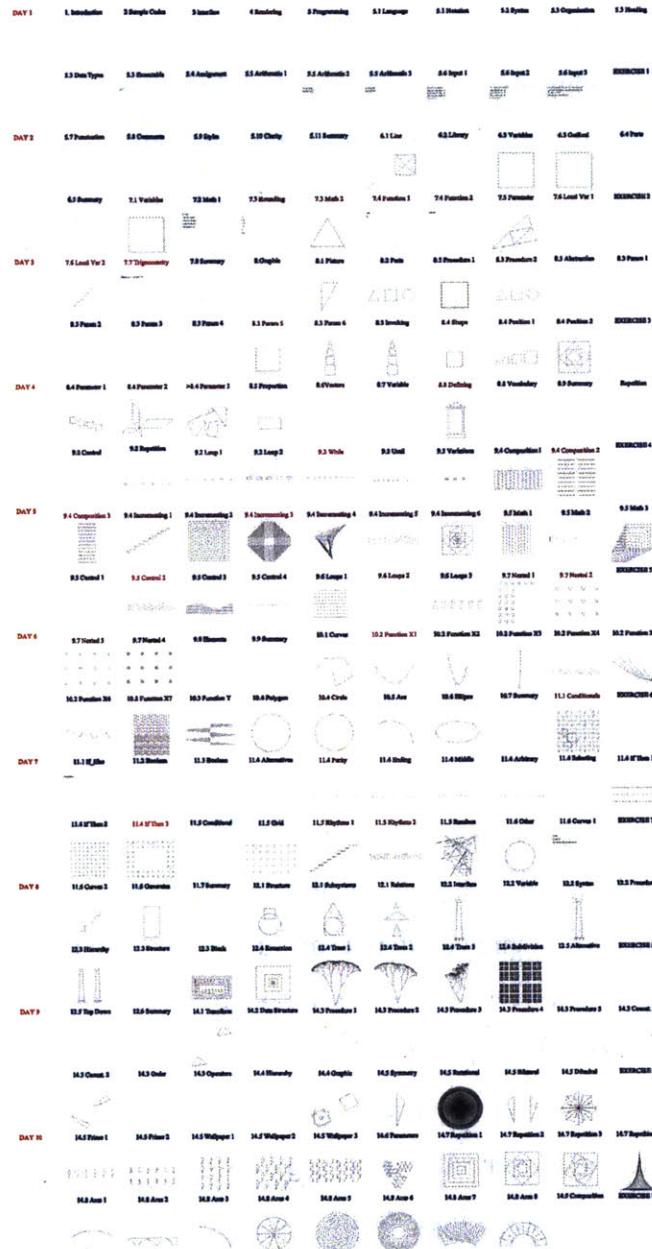


Figure 24. Screen-captured images of the ML tutor (Day 9).
 The image shows an example the ML tutor’s recommendations for a participant on his ninth and final day. The tutor suggested previous modules, including: 9.4 Composition 3, 9.5 Control 2, 10.2 Function X1, and 11.4 If Then 3.

Participants' learner groups change throughout the workshop period, and solutions vary depending on each participant's educational background.

Initial customizations are determined through an analysis of participants' initial survey answers. Subsequently, every time a participant finishes a tutorial module, the participant's study behavior information is updated and the machine learning tutorial also dynamically updates its selection for the next tutorial modules. This customization process occurs collaboratively between the participant's learning behaviors and the tutor's suggestions. Whenever the participant completes a module, the tutor updates that participant's preferences and, at the same time, updates each module with information regarding its level of fitness for that participant's learner group.

4.4 Computational Design Feedback

In conventional architecture design studios, students produce sketches and architectural drawings. Instructors then try to understand students' design intentions and guide them to achieve their design goals. Instructors' design feedback includes instructional sketches and verbal descriptions regarding relevant precedents with which students can further develop their designs. As human instructors provide comments based on their visual inspections and using their architectural knowledge, the computational tutor system developed for this study provides relevant precedents based on its visual recognition of images.

Once a participant submits a drawing of his or her project, the computer vision (CV) tutor analyzes the image data and provides design feedback. In order to generate design feedback, the CV tutor uses an image recognition algorithm, the scale invariant feature transformation (SIFT). SIFT algorithms collect vector information from the images' unique local features (such as corners of rectangles,

any disconnected openings on plans, or the repeated step lines of stairs). The algorithms then compare these vector-transformed features and identify their similarities in terms of Euclidian-distances. The two images with the closest distance are identified as identical projects.

This design feedback system is in its inception, and the current CV tutor can provide only a narrow range of feedback. Ideally, the CV tutor will identify similar historic projects that could be useful to consider for further design developments. The design feedback system is composed of a three-step process. The first step requires a machine training process; the second is an image recognition process that extracts image search keywords using the SIFT algorithm; and the third is an automated image search process using those extracted keywords.

I used an image crawling algorithm to train the computer vision tutor. Image crawling is an automated process that collects image data using the same scripts that image search engines use in many web browsers. Using this image crawling algorithm, I then collected housing projects and added metadata about the projects, such as architects, completion years, locations, and sizes, which could be used for design feedback content. I trained the machine tutor with 320 images from 200 residential projects designed by 15 popular architects. Given a student drawing, the tutor can now identify the most similar image in its database and extract the image's metadata in the form of text. Using this text information, the tutor can then search additional images and provide those results to the student who submitted the drawing.¹⁰

¹⁰ The full list of architects and project names are in Appendix C.

Synthetic Tutor

5. TESTS AND ANALYSIS

This chapter evaluates the effect of the machine learning (ML) tutor and the computer vision (CV) tutor on participants' learning behaviors. In order to evaluate the effect of the ML and the CV tutors, this study conducted an experiment with two types of tutoring systems: one using ML and the other not using ML. The non-ML tutor, as a control group, used identical instructional materials and interface design; however, it did not provide any recommendations or computational feedback as the ML tutor did. I also tested the usefulness of the developed computer vision algorithm-based design feedback system (CV Tutor) that provided computational architectural feedback based on participants' sketches.

5.1 Methodology

In order to evaluate the effectiveness of using machine learning (ML) to teach computer programming, both cross-sectional and longitudinal experiments with two types of tutoring systems were conducted using a two-by-one factorial design. The experiments were conducted at MIT between May and July of 2013, and May and June of 2014. Participants were recruited with flyers posted around the school and via emailed invitations. Seventy-eight participants were randomly assigned to one of the two instructional models (that is, to either the ML tutor or the non-ML tutor¹¹). Both groups were given an entry-survey and a series of tutoring sessions, tests, and exercises. After participants finished their learning experiences with their respective tutorials, a multilevel data analysis with an individual growth

¹¹ I supported the training-data set for the ML tutor with learning-behavior data from participants using both the ML and the non-ML tutor. The non-ML tutor, however, provided no feedback to participants (as shown in Figure 20).

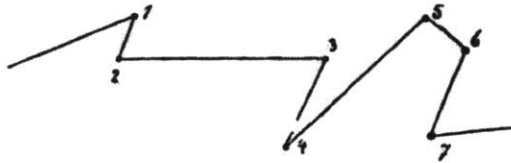
model was conducted (Singer and Willett, 2003). The hypothesis stated that the ML tutor would more effectively help the participants learn computer programming than would typical non-ML computer-aided teaching. Further, the participants who received the ML instruction would perform better over time in the proposed programming exercises than would those who received instructions from the non-ML tutor.

Participating students were either senior-year undergraduates or graduate students in architecture schools. They represented a general group of students who do not have programming experience, yet do have some design studio experience. I invited students from U.S. architecture schools at which architecture students were offered few computer programming courses.

The independent variable (i.e., the predictor) was an instructional model with two different technologies: one with machine learning and one without. Both the ML and the non-ML tutors used HTML webpages with hyperlinks. However, only the ML tutor used dynamically colored text. The non-ML tutor functioned as a control group to examine the effectiveness of the ML tutor as an instructional model. A dummy variable, *ML*, for these two tutors was used. The value of the ML tutor was 1, and the non-ML tutor was 0.

The dependent variables included participants': 1) study time during the workshop, 2) number of modules studied, and 3) number of exercises submitted. After each daily session, students would be asked to write Python code for given exercises (Figure 25 and Figure 26). Classification algorithms using the values of the dependent parameters would then calculate the learning types of each participant; these values fell on a scale running from 1 (for Group 1) to 5 (for Group 5).

5. Below figure shows Paul Klee's active line, limited in its movement by fixed points. Write a program to draw a line of this type.

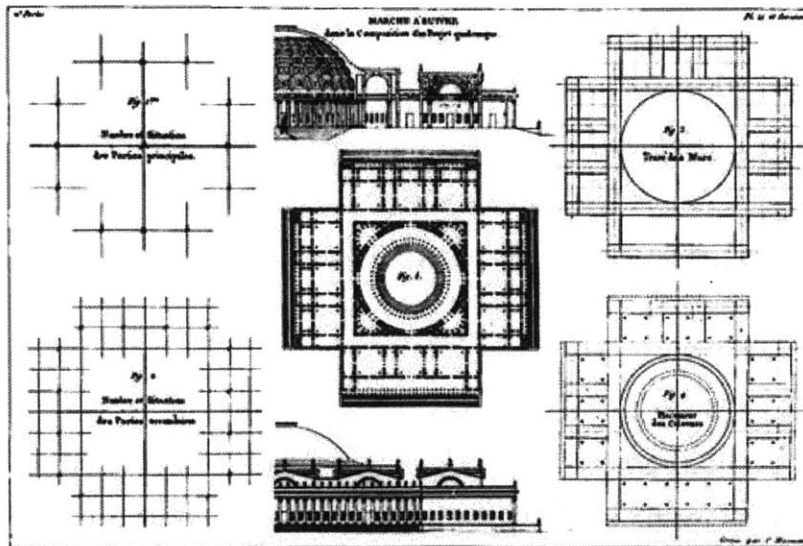


Lines as introduced by Paul Klee in his *Pedagogical Sketchbook*.

Please upload your python file: No file chosen

Figure 25. A sample coding exercise.

4. The French architectural theorist Jean Nicholas-Louis Durand produced many beautiful plates demonstrating how architectural compositions could be understood as -combinations- built up from lower-level vocabulary elements. Figure 12-36 shows an example. Select one of Durand's combinations for careful analysis. What are the parts and subparts? What are the essential spatial relations? What are the design variables? On the basis of your analysis, write a program that generates an interesting series of variants on this architectural theme. Use a top-down programming strategy that parallels the sequence of refinement steps by Durand.



12-36. The combination of vocabulary elements, as illustrated by Durand.

Please upload your python file: No file chosen

Figure 26. A sample exercise to be assigned after the daily session.

5.2 Machine Learning Tutor

The primary data analysis model is a longitudinal data analysis with a multilevel model, especially the individual growth model (Singer and Willett, 2003). This study asks questions such as: How does a participant's learning patterns change over time? How does one individual's learning path compare with other students' learning paths? Can I predict the impact of machine learning algorithms on students' learning while they are taking the proposed tutorials? A longitudinal data analysis can appropriately address these questions. This method was especially effective in this case because participating students improved at various speeds and reached certain proficiencies at different times. This longitudinal study can identify how the predictor (i.e., machine learning) is affecting both within-individual and inter-individual learning patterns.

This multilevel model is composed of a level-1 model (which considers within-individual changes over time) and a level-2 model (which looks at inter-individual differences). The level-1 model uses linear regressions to identify how each student's learning changes over time. The level-2 model identifies how one student's learning pattern differs from other students' patterns by determining whether the intercept and slopes of the average fitted line are systematically correlated with the predictor.

5.2.1 Explorative Data Analysis

In this explorative analysis, the main research question asks how the ML-tutor and the non-ML tutor differently influence the learning behaviors of the study's participants. Participants who used the ML-tutor studied for a longer time, submitted more exercises, visited the workshop more frequently, and accordingly showed more intensive learning patterns than participants who used the non-ML

<i>ID</i>	<i>DAY</i>	<i>ML</i>	<i>Study Time (ST)</i>	<i>Log_e-Study Time (LST)</i>
5	1	0	43.50	3.773
5	2	0	70.00	4.248
5	3	0	75.50	4.324
5	4	0	40.00	3.689
5	5	0	62.50	4.135
6	1	1	127.04	4.845
7	1	0	7.15	1.967
7	2	0	5.90	1.775
8	1	1	91.50	4.516
8	2	1	244.59	5.500
8	3	1	41.32	3.721
8	4	1	106.43	4.667
8	5	1	115.71	4.751
8	6	1	22.62	3.119
8	7	1	26.77	3.287

Table 3. Examples of a person-period data set (ST and LST in minutes).

tutor.

First, I analyzed the data using a level-1 model and identified how each participant's learning behavior changed over the course of the workshop. Next, I conducted a level-2 model analysis to understand how participants who used the ML tutor learned differently from the participants who used the non-ML tutor. In the final step.

I collected between 1 and 15 samples of longitudinal data on 78 participants (Table 3)¹². Every day, the participants studied online tutorials. A time-measuring script recorded participants' daily study time, the number of visits, and their evaluations of each module. The daily assessment period had been determined based on the previous three introductory-programming teaching

¹² The workshop was offered as an online course. Therefore, it was difficult to have balanced data in which each participant had the same number of waves.

	Study Time (min./day)	Log _e Study Time
Mean	111.86	4.139
Median	62.87	4.141
Standard Deviation	153.83	1.071
95% Confidence Interval for Mean	Upper Bound: 131.34 Lower Bound: 92.38	4.274 4.003
Maximum	940.00	6.850
Minimum	3.00	1.100
Range	937.00	5.750
Skew	3.27	-0.006

Table 4. Descriptive statistics for the Daily Study Time and the Daily Log_e Study Time of all participants (n = 242).¹³

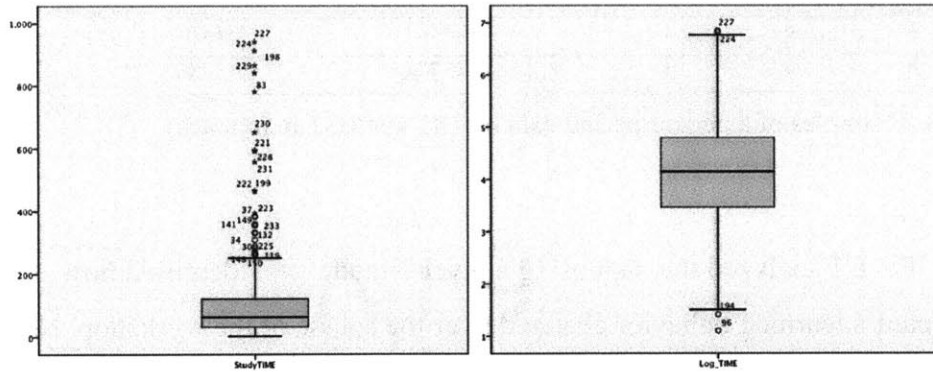


Figure 27. Boxplots of participants’ daily study time in its original scale (left) and in its natural logarithm scale (right).¹⁴

experiences from 2011 to 2013. After attending the daily lectures, most students still needed additional exercises to fully understand the concepts in computer programming. In the previous experiments, students who were writing code showed a significantly improved understanding of programming compared to those students who just watched the instructor’s presentations or listened to

¹³ This study collected non-balanced data, as participants had different numbers of study days.

¹⁴ See the full size graphs in Appendix F for more details.

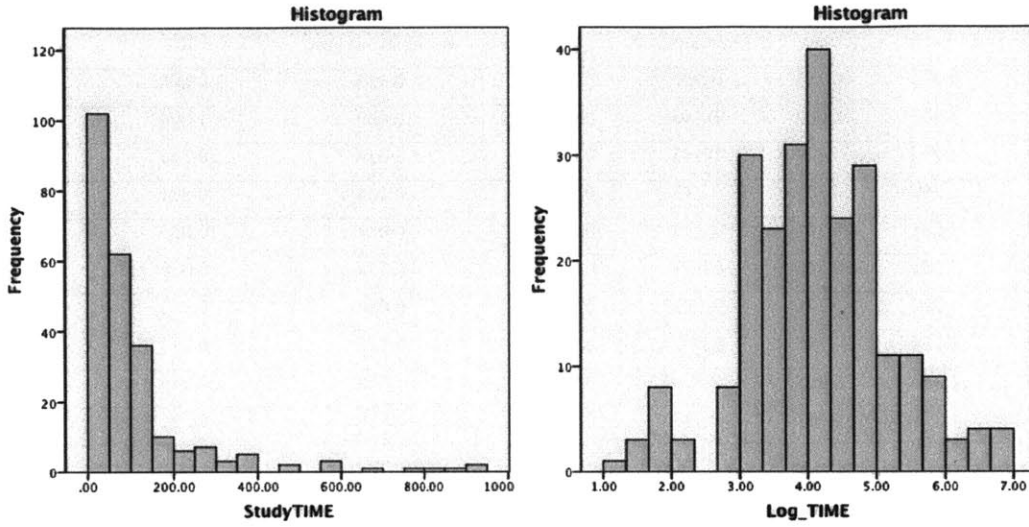


Figure 28. The histogram of a participant’s daily study time in its original scale (left) and in its natural logarithm scale (right).

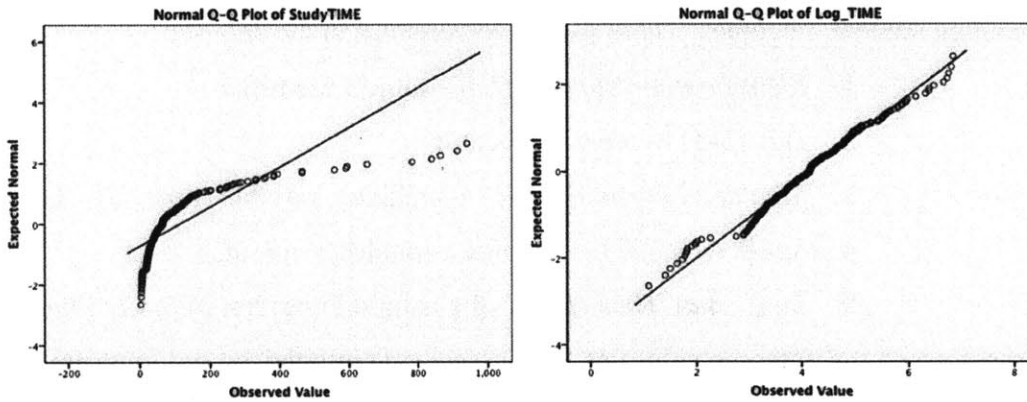


Figure 29. The normality test plots participants’ daily study time in its original scale (left) and in its natural logarithm scale (right). As seen on the right Q-Q (normality test) plot, the normality is improved. The log-transformed sample points lie closer to the diagonal line than the sample points in their original scale.

Synthetic Tutor

ID	Initial status		Rate of change		Residual variance	R ²	ML
	Estimate	se	Estimate	se			
5	4.001	0.255	0.017	0.104	0.325	0.008	0
7	1.967	0.000	-0.192	0.000	0.000	1.000	0
8	4.766	0.423	-0.148	0.089	3.315	0.284	1
19	4.269	0.375	0.033	0.153	0.704	0.081	1
27	3.984	0.209	0.026	0.162	0.052	0.025	0
36	3.511	0.446	0.060	0.345	0.238	0.036	1
47	3.777	0.271	0.143	0.210	0.088	0.316	1
53	3.417	0.145	0.690	0.112	0.025	0.974	1
57	4.882	0.965	-0.320	0.748	1.118	0.155	0
58	4.578	0.567	-0.114	0.303	0.918	0.201	0

Table 5. Randomly selected results from fitting separate within-person exploratory OLS regression models as a function of exponential time.¹⁵

lectures. The data set also includes one potential predictor of study time, *TUTOR*, a dichotomy indicating whether a participant studies with the ML tutor or the non-ML tutor.

A person-period data set (Table 5) was used to systematically measure changing student outcomes. There were five variables in the data set:

1. *Identification Number (ID)* - subject identifier
2. *Day (DAY)* - a time indicator
3. *Machine Learning (ML)* - a predictor variable (time-invariant)
4. *Study Time (ST)* - outcome variables in minutes
5. *Log_e Study Time (LST)* - the natural logarithm of *Study Time*

I transformed the outcome values of *Study Time (ST)* into the natural logarithm of their values. The outcomes in their original scale followed a non-normal distribution. Accordingly, this logarithmic transformation allowed me to assume linearity with *DAY* at level-1, and intuitive interpretation of its analysis (Figure 27,

¹⁵ The results of fitting separate within-person exploratory OLS regression models as a function of time (in their original scale) are attached in Appendix F.

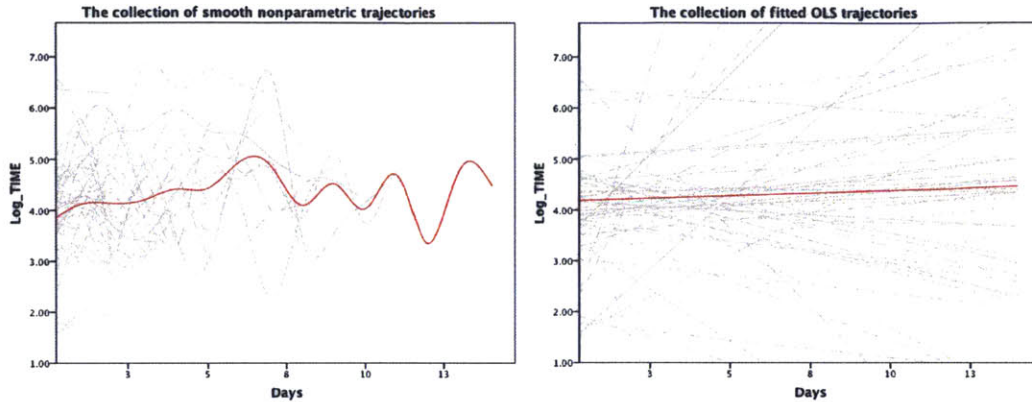


Figure 30. OLS trajectories overlapped in a single plot. The left figure shows the collection of smooth nonparametric ($n = 242$ for 78 participants) and the right the collection of fitted OLS trajectories across participants in the workshop ($n = 202$ for 49 participants). The red lines represent an average change trajectory.

Figure 28, and Figure 29). Having established an overall pattern of learning performance, the next step was to find a parametric model that best fit each individual's learning pattern. The ordinary least squares (OLS) regression was used for the exploratory analysis of the data. The regression provided the simplest illustration of the functional form of the patterns of change, addressing questions such as: does the participant's learning improve, decrease, or remain steady?

Table 1 shows partial-sample results from an OLS-estimation of participants' intercepts and slopes. A detailed calculation of standard errors, residual variance, and R^2 statistics follows in the next sections.

Level-1 Inspection

Figure 31 shows empirical growth plots. It includes OLS-estimated linear trajectories for six randomly selected participants. These trajectories show the

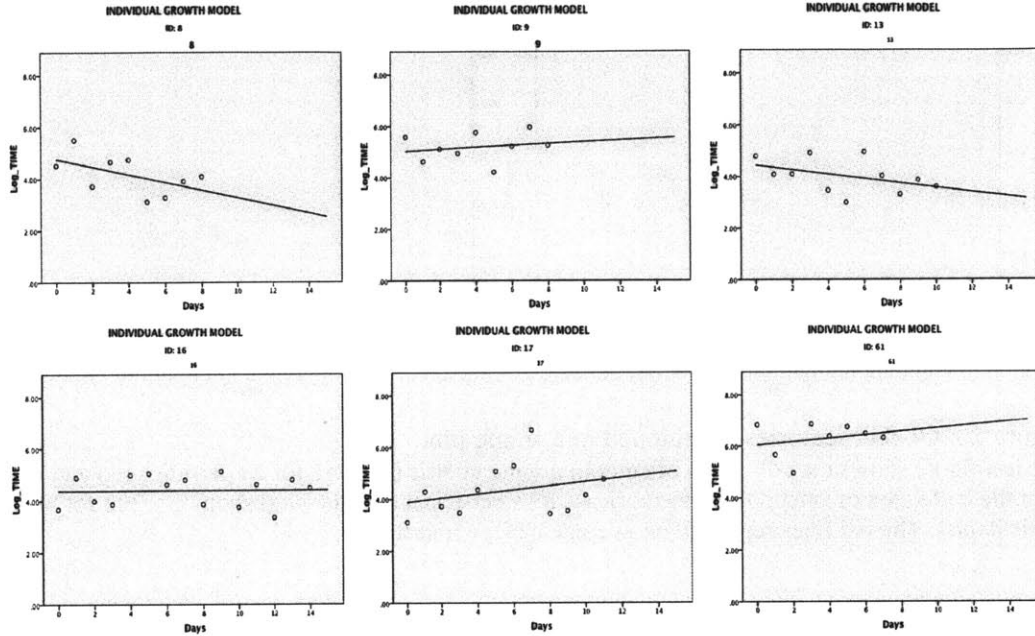


Figure 31. Empirical growth plots for six randomly selected participants in the workshop. These plots show examples of participants' varying learning progress. Among six participants, one shows a significant increase in their study time (ID 17), two show moderate increase (ID 9 and 61), two show significant decrease (ID 8 and 13), and one does not show any change in his/her study time (ID 61). One participant shows high intercepts (ID 61), four show moderate intercepts (ID 8, 9, 13, and 16), and one shows low intercepts (ID 17).

linear relationship between the transformed values of *Study Time* and the values between Days 1 and 15. From this visual inspection, I can confirm that most participants showed a linear relationship between transformed *Study Time* and *DAY* over the course of the workshop.

By overlaying the results of all participants' individual growth patterns in a single plot (Figure 30), an averaged change of all participants is illustrated and the patterns of change of each individual can be compared to the others. Once intercepts and slopes are estimated, descriptive statistics (using means and

		Initial Status		Rate of Change	
		(Natural logarithm)	(min.)	(Natural logarithm / day)	(min./day)
Mean		3.941	51.47	0.038	1.04
Standard Deviation		1.134	3.11	0.439	1.55
95% Confidence	Lower Bound	3.615	37.15	-0.879	0.42
Interval for Mean	Upper Bound	4.266	71.24	0.164	1.18
Minimum		1.410	4.10	-1.630	0.20
Maximum		6.760	862.64	1.830	6.23
Range		5.350	858.55	3.460	6.03

Table 6. Descriptive statistics for the individual growth parameters.

These data were obtained by fitting separate within-person OLS regression models for *Study Time* as a function of exponential time ($n = 202$ from 49 participants' data; any participants' data with fewer than 2 samples are eliminated during analysis).

variances), and univariate summaries (using correlation coefficients) were explored (Table 6). To identify the systemic changes in participants' learning patterns, the effects of machine learning (the predictor) were examined.

The level-1 model analysis illuminates how an individual participant changes his or her study pattern over time. This analysis shows how each student starts the workshop differently and how he or she increases or decreases his or her study time and exercise submission frequencies.

The level-1 submodel for this research (a linear function of *DAY-1*) is:

$$Y_{ij} = [\pi_{0i} + \pi_{1i} (\text{DAY}_{ij} - 1)] + [\varepsilon_{ij}] \quad (2)$$

In equation 2, Y_{ij} is the value of study time for a student i on day j . In this equation, it is assumed that the trajectory is linear in $\text{DAY}_{ij} - 1$. π_{0i} is an intercept and π_{1i} is the slope of the student's learning performance. ε_{ij} is a random measurement of student error.

The average study time on the first day is 130.36 minutes, and its median is 72.10 minutes. The top five percent's study time is 464.50 minutes. Some

Synthetic Tutor

		Study Time (min./day)	Log _e Study Time
Mean		130.36	4.288
Median		72.10	4.278
Standard Deviation		169.14	1.133
95% Confidence Interval for Mean	Upper Bound	158.94	4.479
	Lower Bound	101.79	4.097
Maximum		940.00	6.850
Minimum		3.00	1.100
Range		937.00	5.750
Skew		3.04	-0.342

Table 7. Descriptive statistics for daily Study Time and the daily Log_e Study Time of the ML participants (n = 137).

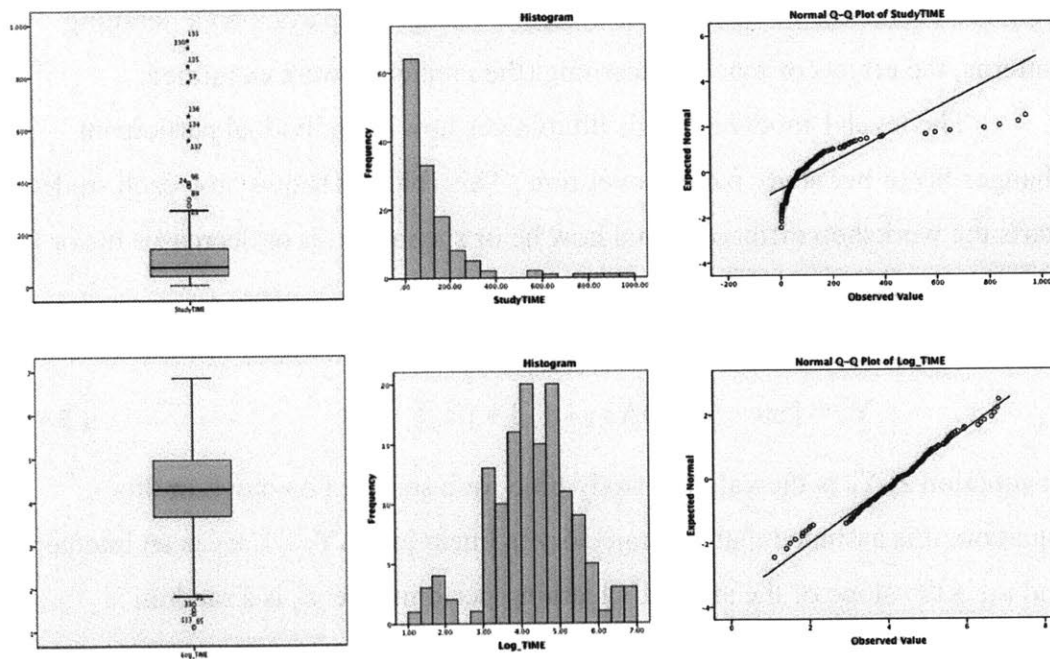


Figure 32. Descriptive statistics for daily Study Time (top) and daily Log_e Study Time (bottom) of the ML participants (n = 137) showing boxplots (left), histograms (middle), and Q-Q (normality test) plots (right).

The six figures above show the improved normality of sample data after log-transformation. See the full-size figures in Appendix F for greater detail.

		Study Time (min./day)	Log _e Study Time
Mean		87.72	3.944
Median		49.00	3.892
Standard Deviation		128.01	0.955
95% Confidence Interval for Mean	Upper Bound	112.49	4.129
	Lower Bound	62.94	3.760
Maximum		865.00	6.760
Minimum		5.64	1.730
Range		859.36	5.030
Skew		3.64	0.477

Table 8. Descriptive statistics for daily Study Time and daily Log_e Study Time of the non-ML participants (n = 105).

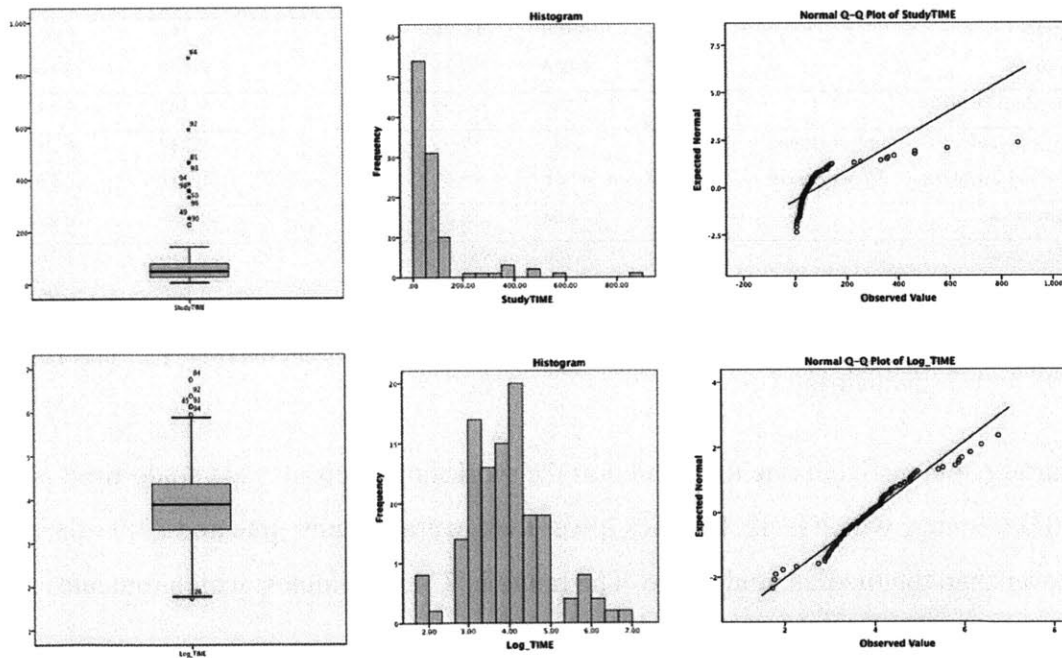


Figure 33. Descriptive statics for daily Study Time (top) and daily Log_e Study Time (bottom) of the non-ML participants (n = 105) showing boxplots (left), histograms (middle), and Q-Q (normality test) plots (right).

The six figures above show the improved normality of sample data after log-transformation. See full-size figures in Appendix F for greater detail.

Synthetic Tutor

		Initial Status		Rate of Change	
		(Natural logarithm)	(min.)	(Natural logarithm / day)	(min./day)
Mean		3.788	44.17	0.232	1.26
Median		3.937	51.26	0.071	1.07
Standard Deviation		1.167	3.21	0.423	1.53
95% Confidence	Lower Bound	3.306	27.28	0.057	1.06
Interval for Mean	Upper Bound	4.270	72.52	0.407	1.50
Minimum		1.410	4.10	-0.150	0.86
Maximum		6.050	424.11	1.830	3.23
Range		4.640	420.02	1.980	5.37

Table 9. Descriptive statistics for individual growth parameters in the ML tutor (n = 137 from 25 participants).

		Initial Status		Rate of Change	
		(Natural logarithm)	(min.)	(Natural logarithm / day)	(min./day)
Mean		4.100	60.34	-0.164	0.85
Median		4.085	59.44	-0.078	0.92
Standard Deviation		1.100	3.00	0.362	1.44
95% Confidence	Lower Bound	3.635	37.90	-0.317	0.73
Interval for Mean	Upper Bound	4.564	95.97	-0.011	0.99
Minimum		1.800	6.05	-1.630	0.20
Maximum		6.760	862.64	0.310	1.36
Range		4.960	856.59	1.940	6.96

Table 10. Descriptive statistics for individual growth parameters in the non-ML tutor (n = 105 from 24 participants).

participants just visit one time and quit the workshop; their average study time is 3.00 minutes, which is 42.45 times lower than average study time and 23.03 times lower than the median study time. The range is 724.44 minutes, which indicates a wide spectrum among participants.

The observed average rate of change is 0.038 per day (on a natural logarithmic scale), meaning a typical participant increases his or her study time

	Test Statistics	
Median Initial Status	4.169	(ML participants, n = 119)
	3.912	(non-ML participants, n = 83)
Mann-Whitney U	253.00	
Asymp. Sig. (2-tailed)	0.347 (> 0.05)	

Table 11. Mann-Whitney U test results for the initial statuses of two groups (ML and non-ML).

	Test Statistics	
Median Rate of Change	0.071	(ML participants, n = 119)
	-0.078	(non-ML participants, n = 83)
Mann-Whitney U	93.50	
Asymp. Sig. (2-tailed)	0.000 (< 0.05)	

Table 12. Mann-Whitney U test results for the rates of change of two groups (ML and non-ML).

1.04 minutes every day until the end of the workshop. The highest daily increase was 1.83, meaning that participant increased his or her study time by 6.23 minutes per day. The lowest increase was -1.630, meaning that participant increased his or her time 0.20 minutes per day.

Level-2 Inspection

After considering individuals' learning behaviors, I analyzed the level-2 model that reveals how participants in the two groups (the ML tutor and the non-ML tutor groups) behave differently (Figure 34). The level-2 sub-model for this research is:

$$\pi_{0i} = \gamma_{00} + \gamma_{01}ML_i + \zeta_{0i} \quad (3)$$

$$\pi_{1i} = \gamma_{10} + \gamma_{11}ML_i + \zeta_{1i} \quad (4)$$

In equations 3 and 4, the π_{0i} (intercept) and the π_{1i} (slope) result from the impact of the predictor (ML) and have their own individual residuals (ζ_{0i} and ζ_{1i}). The

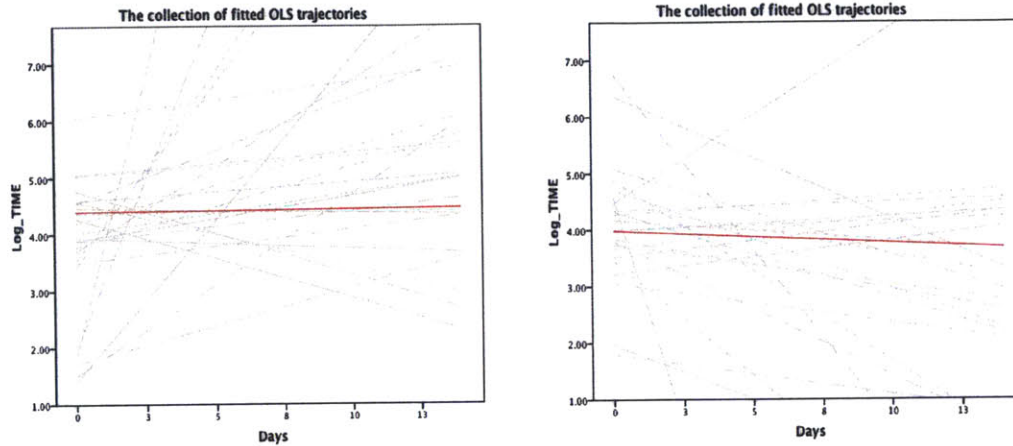


Figure 34. Fitted ordinary least squares (OLS) trajectory of ML (left) and non-ML (right) tutor superimposed on empirical growth plot.

Two red lines represent average population trajectories. (ML: $n = 119$ from 25 participants' data, non-ML: $n = 83$ from 24 participants' data, any participants' data fewer than 2 samples are eliminated during analysis. Their descriptive statistics are in Table 11, Table 12, Figure 32, and Figure 33.

four level-2 parameters (γ_{00} , γ_{01} , γ_{10} , and γ_{11}) are calculated separately by setting ML to 0 and ML to 1 and illuminating the level-2 fixed effects.

The findings of this longitudinal study are that the median initial status (intercept) of the ML tutor participants and the non-ML tutor participants are 4.169 and 3.912 respectively (Table 9 and Table 10). The distributions in the two groups' intercepts do not differ significantly (Table 11)¹⁶. The median rates of change (slopes) of participants who use the ML tutor and the non-ML are 0.071 and - 0.078 respectively (Table 9 and Table 10). The distributions in the two

¹⁶ I conducted Shapiro-Wilk tests of normality, and the result showed that the distribution of the initial status of the ML tutor participants appeared to follow a normal distribution ($p = 0.259 > 0.05$; its null hypothesis was that the data are normally distributed, and it was not rejected); however, the initial status in the non-ML tutor participants did not follow a normal distribution ($p = 0.006 < 0.05$; its null hypothesis was rejected). Accordingly, I used the Mann-Whitney U test instead of an independent t-test to compare the distributions.

groups' slopes differ significantly (Table 12).¹⁷

These findings illuminate the fact that the ML tutor does not influence participants' learning at the beginning of the workshop. However, the ML tutor does improve participants' learning by maintaining their study time through the course of the workshop. This improvement stands in contrast to the non-ML participants, whose study time decreased more rapidly.

5.2.5 Additional Findings

In this section I describe additional cross-sectional analyses of the collected data and illustrate different effects of the ML tutor and the non-ML tutor on the study's participants. I focus on those participants who use the ML and the non-ML tutor similarly on the first day. The ML tutor does not influence these participants' learning at the beginning. However, participants who use the ML tutor increase their study time more rapidly than participants who use the non-ML tutor over the course of the workshop.

The ML tutor successfully influenced the learning behaviors of participants and improved their learning more than did the non-ML tutor. The ML tutor identifies a participant's learning patterns and recommends a set of customized tutorials. In the recommended tutorial set, participants can see how previous participants studied the material, and how many participants liked the tutorial. On the other hand, participants who use the non-ML tutor did not get any extra information. The non-ML tutor participants had to study the tutorials without receiving any feedback, just as they might do for a conventional online course. Accordingly, the learning process of an ML tutor participant showed a

¹⁷ The two distributions of rates of change in the ML and the non-ML tutor did not appear to follow a normal distribution when I tested them using Shapiro-Wilk tests ($p < 0.05$). Accordingly, I also used the Mann-Whitney U test instead of an independent t-test to compare the distributions. See Appendix F.

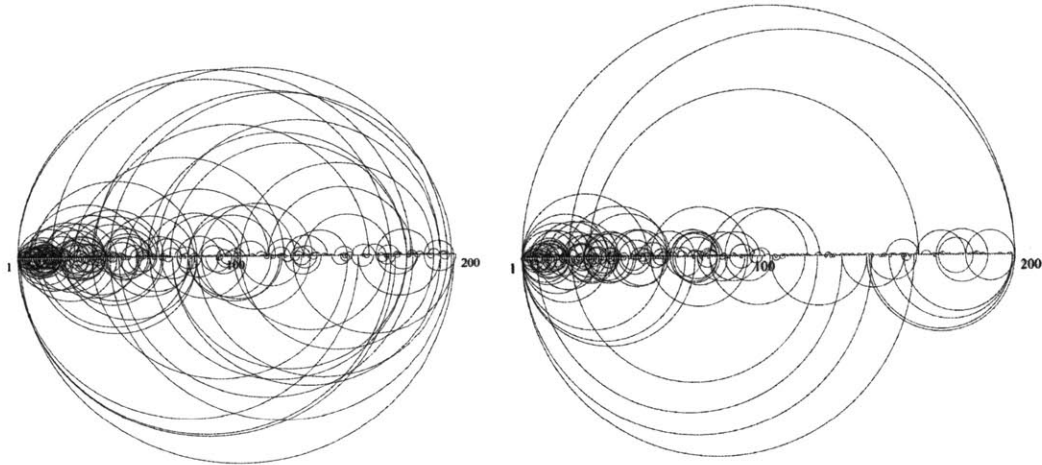


Figure 35. Visiting patterns of participants.

The image on the left shows the visiting patterns of all ML-tutor participants, and the image on the right shows all non-ML tutor participants' visiting patterns. The circles above the horizontal lines indicate that users moved forward from a current module to a higher module. The circles beneath the horizontal lines show that a user moved backward from their current module. Small circles indicate a user's movement to nearby modules, and large circles indicate a user's movement to a more distant module. The largest circle above the horizontal line on the right indicates that a participant moved from module 1 to module 200, visiting the final project immediately after visiting the first.

relatively more dynamic pattern than that of students who used the non-ML tutor (Figure 35).

Participants using the ML tutor visited the tutorial 8,828 times in total, which is 2.26 times more than the 3,902 visits made by participants using the non-ML tutor. The average number of visits made by ML tutor participants was 215.32, while non-ML tutor participants made only 97.55 visits (and the median visiting numbers are 132 and 60 respectively). The ML-tutor groups' standard deviation in visit numbers was 223.8, and the standard deviation for the non-ML tutor group was 97.6 (Figure 36).

An independent-sample t-test was conducted to compare the visit numbers

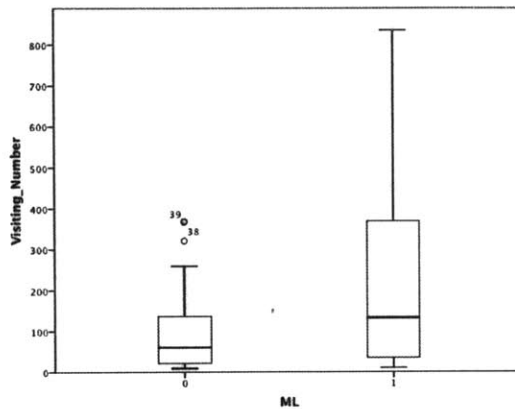


Figure 36. The distribution of the number of visits by participants of non-ML (left; $n = 39$) and ML tutors (right; $n = 39$).

for both groups of participants. The effect of the ML tutor on the number of visits is relatively large. These results suggest that the ML tutor has a considerable effect on participants' visiting number. Specifically, this result suggests that when participants study with the ML tutor, their number of visits increases.

By observing the visualization of a student's learning pattern, I can tell whether that student studies with the ML tutor or the non-ML tutor. In contrast to participants who use the ML tutor to dynamically study the workshop materials, participants who use the non-ML tutor show linear study patterns. Participants who use the ML tutorial re-visit the same tutorial multiple times and complete many tutorials by moving forward and backward (Figure 37, top). Students who learn with the non-ML tutor, on the other hand, complete tutorial sets linearly and rarely re-visit the same tutorial (Figure 37, bottom).

The ML tutor successfully demonstrates the possibility of minimization or full or partial replacement of human instructors in design-scripting education. Based on the daily exercises submitted by the participants, I can conclude

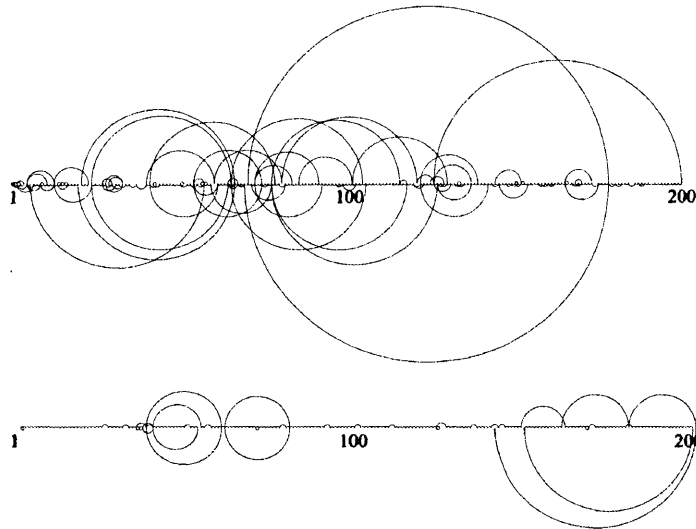


Figure 37. An example of two learning patterns. The top image shows the dynamic learning pattern of a participant who uses the ML tutor. The bottom image shows the linear learning pattern of a participant who uses the non-ML tutor.

that participants successfully learned essential programming concepts, such as functions, repetition, and conditional statements. Participants' submitted code and questions show that not only did they understand the principles of computer programming; they could independently write code for design work (Figure 38). Their clearly organized code used functions, effectively generated complex shapes using repetitions, and controlled boundary conditions. This shows how successfully and independently these participants can learn.

The examples below show a participant's questions regarding workshop exercises. This participant had difficulty understanding the meaning of the exercises. However, as soon as she understood that the exercise simply required participants to draw a triangle, she quickly completed her code work.

```

1  import rhinoscriptsyntax as rs
2  import math
3
4  def drawTriangle1(xc, y1, base, altitude):
5      x1 = xc - (base)
6      x2 = x1 + base
7      y2 = y1 + altitude
8
9      pt1 = [x1,y1,0]
10     pt2 = [x2,y1,0]
11     pt3 = [xc,y2,0]
12
13     rs.AddLine(pt1,pt2)
14     rs.AddLine(pt2,pt3)
15     rs.AddLine(pt3,pt1)
16
17 def drawTriangle2(xc, y1, base, altitude):
18     x1 = xc - (base)
19     x2 = x1 + base
20     y2 = y1 + altitude
21
22     pt1 = [x1,y1,0]
23     pt2 = [x2,y1,0]
24     pt3 = [xc,y2,0]
25
26     rs.AddLine(pt1,pt2)
27     rs.AddLine(pt2,pt3)
28     rs.AddLine(pt3,pt1)
29
30 def myTriangleComposition():
31     xc = 10
32     count = range(1,8)
33     for i in count:
34         drawTriangle1 (xc, 0, 10, 3)
35         drawTriangle2 (xc, 3, 10, 6)
36         xc = xc + 3
37
38
39 myTriangleComposition()

```

Figure 38. A participant's sample code.

The submitted code shows that the participant successfully used multiple functions, input parameters, and repetitions without any instructor assistance. She did, however, make two almost identical functions ('drawTriangle1()' and 'drawTriangle2()'). If she better understood the purpose of using functions, she could eliminate one.

The first is 2.4. I do not understand how to arbitrarily place the coordinates that define the vertices of the triangle before applying math. I would understand if I input two variables and then solved for the height and midpoint to find the apex. I am missing something here.

The second is 3.6. I am not sure how to approach this question. I know the golden ratio formula, but can't figure out how to approach this either.

The third is 3.7. I am not sure what is being asked here.

Participants who used the ML tutor studied more intensively and were

more likely to complete the workshop than participants who used the non-ML tutor. They submit more exercises and visited the tutorials more frequently, and they studied for longer periods than participants who used the non-ML tutor. As opposed to the linear approach taken by the non-ML tutor participants, participants who used the ML tutor completed it dynamically, by navigating the whole tutorial and reviewing previous tutorials iteratively.

5.3 Computer Vision Tutor

The computer vision (CV) tutor is an algorithm designed to search for relevant architectural projects using a captured image of a user's sketch or floor plan. In this section, I describe the developed CV tutor, a test method designed to measure its performance, and the survey results showing what participants think of it. This section also includes the algorithms used by the CV tutor, its hardware settings, and its test environments. Forty-two participants joined the experiment, tested the tutor, evaluated the usefulness of its computational feedback, and compared that feedback with the feedback they received from their previous human instructors.

5.3.1 Experimental Design

During desk critiques, instructors check students' drawings and provide relevant architectural precedents in the forms of images or the instructors' own sketches. The purpose of the computer vision tutor (CV tutor) is to develop an automated design-education system that can support novice students' architectural design processes in a similar way. Specifically, the CV tutor can recognize students' architectural drawings, regardless of their visual styles, and provide design feedback in the form of drawings, model photos, and perspectives. These kinds of feedback are popular in conventional architectural design-studio education.

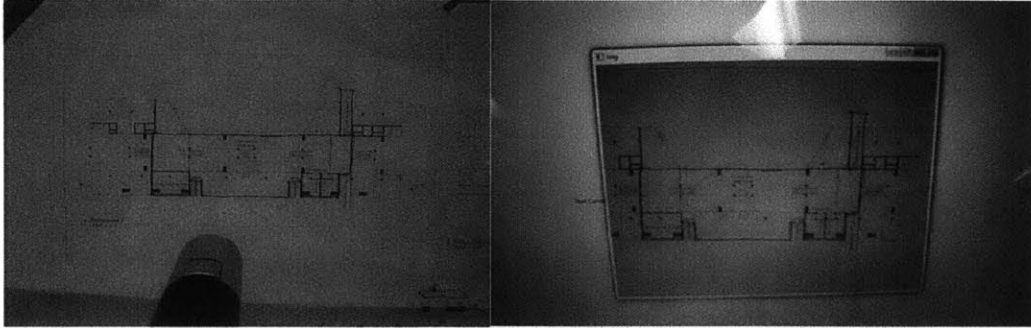


Figure 39. Computer vision tutor process.

The left image shows a user taking a picture of his floor plan, which had been pinned to a wall. The right image shows the captured image of the floor plan on the screen of a computer monitor.

This experiment tests the possibilities of transforming human-oriented education into a computational process. This experiment utilizes a computational tutor in design education and evaluates its performance. The experiment asks participants to evaluate the performance and usefulness of the developed CV tutor, and it also asks them to compare the CV tutor's feedback with human instructor's feedback (Figure 39).

The algorithms used to recognize participants' architectural drawings were developed using the SIFT algorithm. The SIFT algorithm collects the local features of an image and transforms them into vector data. I then provide labels, including the names of architects, the titles of the projects, and additional features of an image and transform them into vector data. I then provide labels, including the names of architects, the titles of the projects, and additional information. The algorithm can then recognize new images regardless of their orientation, scale, and visualization style, and it can match them with the provided labels. The algorithm performs well, with a greater than 98 percent matching rate. It accurately finds the same architectural drawings with different styles, matching

Synthetic Tutor

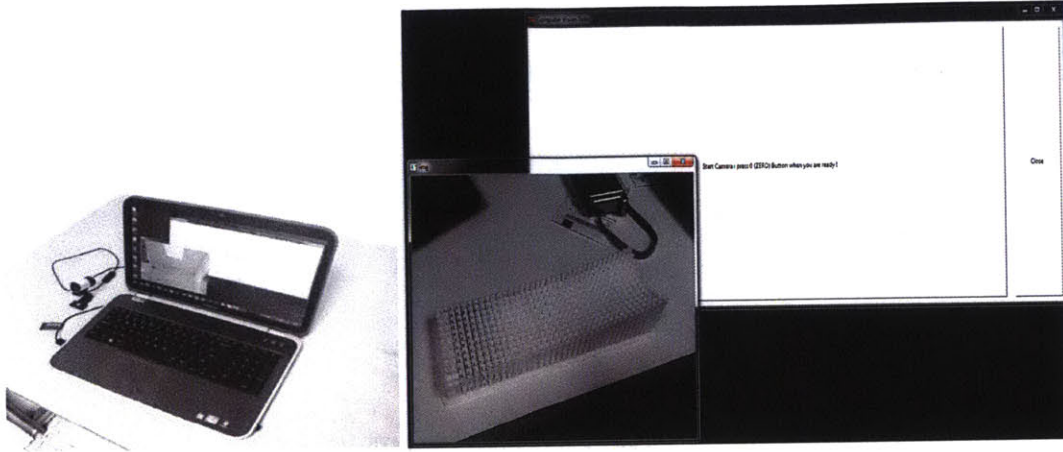


Figure 40. The hardware and software setting of the developed computer vision tutor system.

The left image shows a camera attached to a computer. The right image shows the CV tutor interface, composed of two buttons ('start the camera' and 'close'), and a camera viewer (lower left corner screen) that shows the camera view in real-time.

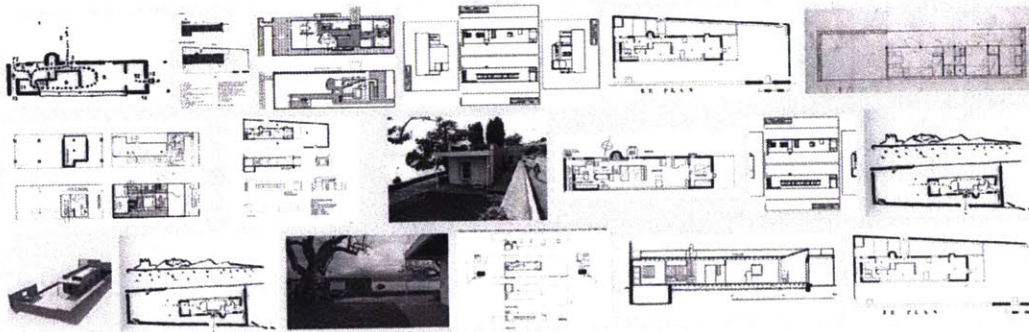


Figure 41 The feedback from the computer vision tutor. The CV tutor has recognized the users' plan's similarity to the floor plan of Le Corbusier's Villa Le Lac, and it shows image-search results for that floor plan.

drawings with hard lines, gray-scale perspectives, or computer-generated color renderings.¹⁸

¹⁸ Section 3.4 provides descriptions of these algorithms.

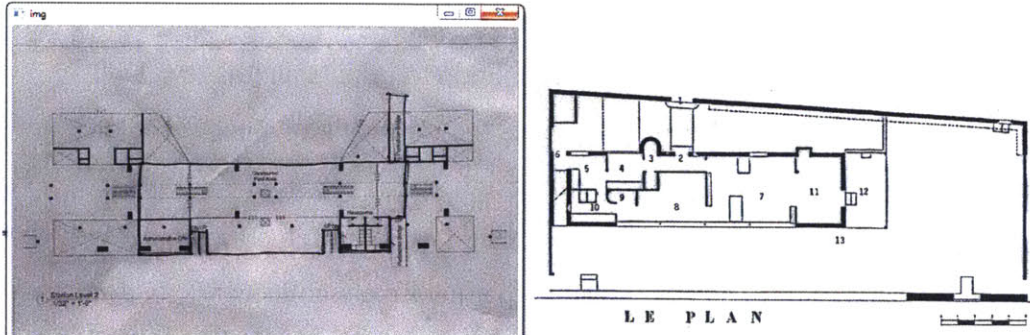


Figure 42. A participant's floor plan and the CV tutor's suggested floor plan. These two images show the result of suggested design feedback. The left image is a participant's floor plan and the right is a floor plan that the CV tutor suggests (Le Corbusier's Villa Le Lac floor plan).

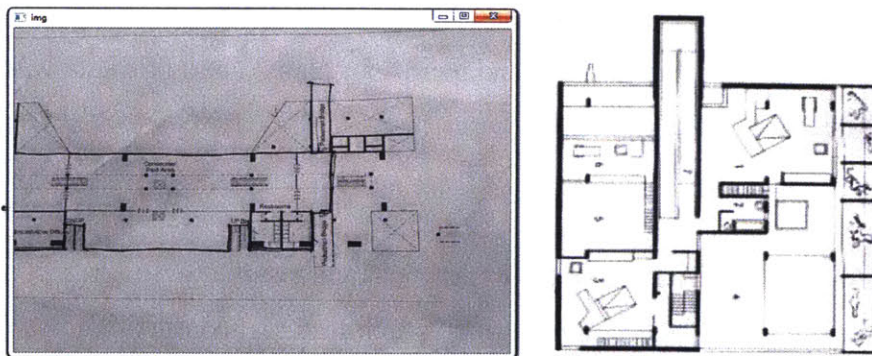


Figure 43. A participant's second attempt and CV tutor's suggested floor plan. A participant takes a picture of the right part of his floor plan and the CV tutor suggested Le Corbusier's Villa Shodhan floor plan.

The CV tutor consists of a high-resolution camera and a computer that runs the CV tutor software (Figure 40). The overall operating sequence is:

- 1) The tutor activates the camera, and it recognizes an image.
- 2) It runs the SIFT algorithm and identifies the most similar image among its training-data sets.
- 3) The software retrieves the image's textual metadata, which

includes project title, architects, construction year, and site.

4) The tutor uses this information to initiate an image search.

5) Finally, the tutor provides the user with the image feedback that had been retrieved by the image search engines.

The CV tutor learns architectural design using a Web crawler. A Web crawler is an automated search algorithm that iteratively collects relevant images based on given search keywords. This machine learning process is a collaboration between human instruction and a machine's automated data collecting and labeling processes (i.e., supervised learning). A human instructor provides a list of architects and their projects, and the CV tutor then gains architectural knowledge from the collected images and their labels. For the CV tutor's initial learning process, I included only housing projects and their floor plans¹⁹, and this data was used to provide design feedback.

5.3.2 Findings

Forty-two students volunteered to join the experiment and test the CV tutor.²⁰ Most participants (40 out of 42) were majoring in architecture, and the other two students were majoring in urban design. They brought their current and previous architectural studio-design projects and showed them to the CV tutor. Many participants used their printed drawings, although some students used digital files (e.g., AutoCad or PDF files) opened on their computer monitors.

The CV tutor recognized participants' drawings and identified the most similar architects' projects. It then searched for images on the Web and showed its search results, including project pictures, models, and drawings. Participants could repeatedly use the CV tutor to get multiple design feedback results. After

¹⁹ Appendix C includes the full list of architects and their projects.

²⁰ The survey questions are attached in Appendix B.

Overall, how do you evaluate the feedback of the CV-Tutor?

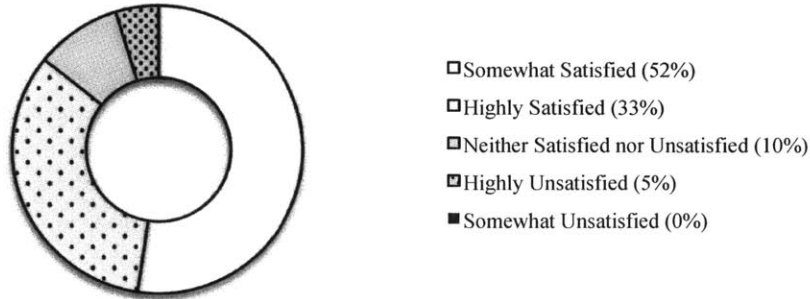


Figure 44. Overall satisfaction rate.

How useful is the feedback to improve your design?

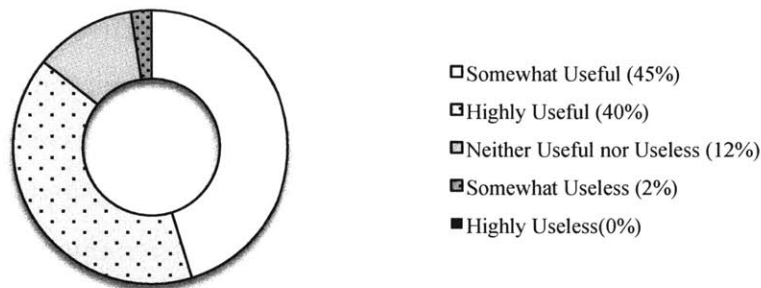


Figure 45. The usefulness of the CV-tutor feedback.

the experiment was finished, participants evaluated the effectiveness of both the CV tutor and their feedback results.

Overall, many participants evaluated the feedback from the CV tutor as satisfactory. Fourteen participants (33%) reported that they were highly satisfied, and 22 participants (52%) were somewhat satisfied. These participants commented that the CV tutor could be useful for their projects, and that they were excited about the potential of using their sketches and drawings for developing their projects (Figure 44).

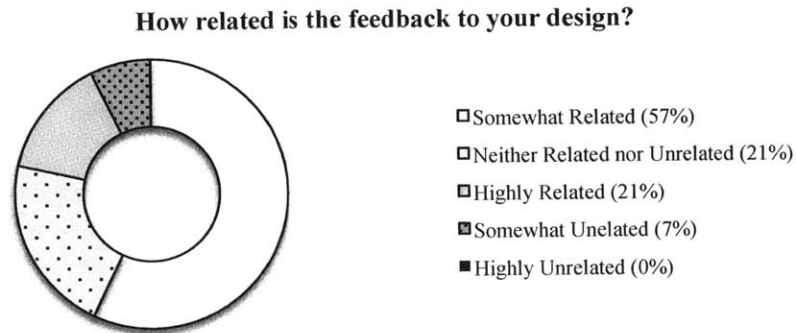


Figure 46. The relevance between the feedback and participants' projects.

Many participants found the design feedback to be highly useful or somewhat useful for their design (17 and 19 out of 42, respectively; Figure 45). Some participants also considered the design feedback to be relevant to their design sketches (3 out of 5). No student thought the feedback was useless or unrelated to his or her design. Eleven participants wrote that the tool could be more effective if it had more data than did the current version (Figure 46). I consider this performance acceptable, and I believe it may be worth developing the CV tutor further and testing it with a larger number of participants.

About half of the participants (20 out of 42) considered the machine design feedback and the instructors' feedback equally useful. Some participants (16 out of 42) evaluated the human instructors' feedback as better than the machine's. Two participants commented that image results were less interesting than an instructor's verbal explanations and dynamic feedback. One student remarked that the lack of human feeling from the tutor made the design feedback less effective. Not surprisingly, human interaction during design education is an important part of the design development process and an architectural studio education (Figure 47).

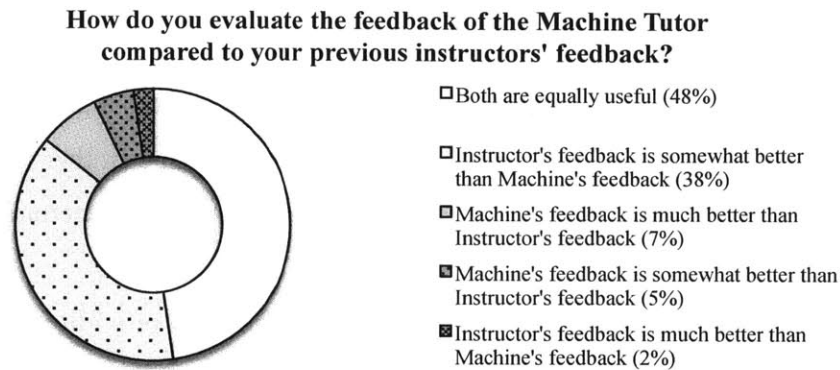


Figure 47. Comparison with human instructors.

Participants, most interestingly, considered the design feedback similar to human design feedback. They commented that they couldn't distinguish some of the design feedback from the CV tutor from that of a human. In a few cases, some participants commented that the computational feedback was highly unexpected when compared to that of a human instructor. At the same time, a small numbers of comments reported that the CV tutor's feedback was related to their design, yet was not as dynamic and offered less insightful information than did their human instructors' feedback.

Unlike text-based searching, image searching is still underdeveloped and therefore less effective. Image search results from various search engines sometimes do not find what they are looking for, even when they use many keywords. An image's metadata frequently does not convey its meaning, and when people do not know appropriate keywords, it can be difficult to find searched-for images.

The CV tutor develops a method for finding relevant images from students' drawings and architectural visualizations. The CV algorithm used in this study

conveys proper search keywords from architectural drawings that humans have provided, and it implements image-search engines to use that retrieved metadata to operate their search processes. The CV tutor's main search algorithm is SIFT, which finds similar images based on the semantics of images rather than visual styles. This allows the CV tutor to recognize participants' architectural plans and provide relevant design feedback in the form of images.

This CV tutor experiment shows the possibility of providing computational design education in architectural-design studios. Design education is one of the areas in which it is difficult to utilize computational automation due to the high level of customization that individual students require. The findings of this study open the possibility of automating customized design education, and may make online education more effective and useful in subjects where high customization is required.

6. CONCLUSIONS

This study explored collaborative human-machine approaches to solving the complex problems associated with design education, problems which humans or machines could not easily solve alone. I proposed and developed the Synthetic Tutor that showed a superior teaching performance to the non-machine learning tutor (such as a conventional online tutor). These results illuminate how the Synthetic Tutor can influence its users: they study a longer time, complete more learning materials, and learn more actively than participants using the non-ML tutor. These findings can be used to develop a new type of intelligent tutor that can profile users' learning needs and provide them with a mass-customized education.

6.1 Contributions

I conducted three workshops to understand the difficulties that students might experience during their learning, and I prepared teaching materials and hands-on exercises as part of those workshops. I researched pedagogies and the *bricolage* educational model for effective teaching. I explored machine learning algorithms that could be used to profile learners and customize teaching materials for individuals with diverse learning curves.

Based on the findings from these three workshops and using the developed Synthetic Tutor, I conducted an experimental design to test the effect of the Synthetic Tutor on participants. I observed and traced how these participants, who had various educational backgrounds and programming experience, applied their learning of design scripting into architectural design projects. The experimental

design shows improvements in learning from both the Synthetic Tutor and the participants. From a human learning perspective, it is significant that this study has found that the Synthetic Tutor changes the learning behavior of participants. After identifying the influence of the Synthetic Tutor on participants' learning, this study attempted to find the causes of these behavioral changes.

In order to accomplish this, the first attempt was to prepare an exit-survey and to ask participants directly about whether they followed the recommended tutorials, whether the recommended contents improved their learning, and how they evaluated the usefulness of these recommendations. The exit-survey, however, was not successful due to the many participants who stopped learning in the middle of the workshop period and did not respond to the survey.

In the second attempt, I analyzed each individual participant's study pattern. For example, as Figure 37 shows, this participant in the ML tutor dynamically moved backward and forward through different modules. When I compared different participants' study patterns, this dynamic pattern is commonly observed regardless of the participants' learning groups. At the same time, the participants who studied with the non-ML tutor commonly showed a linear study pattern.

While participants were improving their learning, the Synthetic Tutor also improving its performance. For this machine learning perspective, the improvement of the tutor could be explained in two different ways. The first is the increasing amount of participants' learning data. On the first day, the ML tutor could only utilize six participants' data. The number of participants increased to 28 by the end of the first week, 45 by the end of second week, and it reached 78 at the end of week 4. In the beginning, the ML tutor could only compute recommendations based on participants' study time and their number of visits.

When more participants joined the workshop, the ML tutor could utilize participants' evaluation data and acquire additional data from the new participants that might improve the quality of recommendations.

The second explanation for the Synthetic Tutor's improved performance is the increasing k-value of the k-means clustering, which makes the recommendation more useful. Within a narrow range of k-values (between 1 and 5), this study shows a high k-value can refine the clustering results. Among the five identified learner groups (see Section 3), participants in Group 1 (extraordinary group) were particularly benefited by the ML tutor. Although there were not many participants in this group, they received useful recommendations from the ML tutor due to their longer study time and larger number of visits than participants in other groups. On the contrary, the participants in Group 4 and Group 5 did not get useful instruction from the ML tutor. Because of their short study time and brief enrollment in the workshop, the ML tutor could not provide them with effective recommendations.

This study also developed computer vision (CV) algorithms to provide computational design feedback. The performance of the implemented CV algorithm showed high precision and low error rates. Participants commented that the feedback from the CV tutor was relevant to their projects, and they stated that although the feedback was not as insightful as that of human instructors, it was as highly useful as the humans' feedback. One participant even stated that the CV tutor's feedback identified a project that he studied during his conceptual design process that no human instructor recognized.

This study illustrates the explorative process I conducted in order to identify students' learning difficulties and find the appropriate teaching materials for these participants. This study illuminates how a human-machine collaborative

process could solve difficult problems in design education. Accordingly, this study provides a new framework for the developmental process and for the role of artificial intelligence in design courses—particularly when those courses still need to be developed, as would be the case for an introductory course in computer programming for architecture students. This new framework could also contribute to current online education, which is heavily content-based, by providing features that could convert static information delivery into an interactive, dynamic, and customized experience.

6.2 Discussion

Statistical analysis shows the effectiveness of the developed Synthetic Tutor. The results illustrate that small changes in online education can significantly improve participants' learning performance. However, there are several identified issues that are outside of this study, and could be highly valuable for further studies.

With more data, this study could have provided in-depth findings on how the Synthetic Tutor's students' improvement related to their educational and motivational backgrounds. Seventy-eight participants joined the experiment, and that was sufficient to analyze the overall impact of the Synthetic Tutor. However, that number was not large enough to analyze the Synthetic Tutor's different impacts on participants in the five learner groups identified by the machine learning algorithms. If the study were to collect more data for each user group, the study could then identify the various impacts of the ML tutor. These findings could provide useful insights for designing effective online courses and interactive machine tutors.

This study collected unbalanced data that consists of between one and fifteen waves from 78 participants. At the beginning of this study, I designed the

experiment to have the same amount of data from each participant: 14 waves during two-week periods. However, the course was offered online and participants showed highly irregular study patterns. Longitudinal data analysis is flexible enough to estimate this unbalanced data; however, balanced data (in which all participants have the same amount of data) could improve the precision of estimation and model fitting. If the workshop could generate balanced data by being offered as a formal online course through a university for a certification or a grade, or if it could have a higher number of participants, then this study could achieve greater precision.

The workshop offers a final assignment to evaluate participants' design-scripting capability. This assignment is taken from a sample question on the architect registration examination (ARE) that asks participants to design a two-story residential building. However, only a small number of students submitted final architectural design projects. Accordingly, this study is not able to prove the effectiveness of the CV tutor's feedback during the workshop; rather, the test had to be conducted separately after completing the experiment. A future project should develop a computational instruction that can teach computer programming language and architectural design seamlessly, and which can provide design feedback from computationally generated architectural drawings.

Three students emailed their feedback on the ML tutor after they completed their learning experience. All three stated that the ML tutor's color-coded suggestions were neither effective nor useful for their learning. Regardless of these low evaluations of the ML tutor's interventions in their learning, this study shows that the learning performances of participants who used the ML tutor were higher than that of those participants who used the non-ML tutor. This

contradiction may imply that there is a high potential for improvement in participants' learning if the ML tutor is continuously researched and improved.

This study could also benefit by including in its teaching materials recent developments in programming languages. I prepared the workshop contents mainly from Mitchell's *The Art of Computer Graphics Programming: A Structured Introduction for Architects and Designers*, which was published in the 1970s. The book lacks content in recent programming knowledge, such as object-oriented programming and visual programming languages. This study may need to include these recent changes in computer programming technologies to accurately research the impact of the Synthetic Tutor on learners' performance.

The high drop rate in conventional online education is a common challenge, and I observed that both the ML tutor and the non-ML tutor did not improve that condition in this study. Many participants stopped their learning at the early stages of the workshop. Providing a graduation certificate or official record of student participation upon completion of the workshop could address the early drop-rate problem and improve the tutor's teaching performance.

Computer vision-based architectural drawing recognition is the starting point for the computational design process and can be applied to various educational domains and industries. For example, in music education, novice musicians could be recorded and given real-time feedback. In athletic training, novice learners' body movements could be corrected with instant camera-based feedback. In medical studies, the CV tutor could be used in screening and treatment, using a visual inspection of the skin or eyes.

The developed CV tutor shows high precision in analyzing participants' hand-drawn plans, and this suggests the possibility of using the tutor to evaluate these drawings and provide relevant project information. If the CV tutor could

read local features in the same way as it reads global features, it could extract useful information from those features and provide in-depth feedback as closely as human instructors do. The perception of architectural sketches in various scales and typologies is a critical capability of human architects. Many professionals can read various forms from a simple dot on a piece of paper. The single dot could be a pipe on a roof, a column on a floor, a spherical glass dome on a greenhouse, or a tubular high-rise building in a city. Human architects' insight comes from their ability to learn from drawings and buildings in various scales and contexts.

These multi-scale and multi-contextual readings of drawings and sketches are crucial to an experienced instructor's methods in an architectural design studio. These capabilities may also enable a future CV tutor to evaluate an architectural design from multiple scales and perspectives and to provide purposeful recommendations from this multilevel analysis. For example, a future CV tutor could individually recommend different sets of projects depending on a student plan's analyzed design issues (e.g., circulation design, spatial organization, or structural behavior). However, the computational algorithms developed in this thesis do not easily implement this spatial insight and are heavily dependent on direct data comparisons. The development of high-level (trans-scale or trans-typological) comparison and clustering could improve the current research in computer vision and introduce an important missing component into the CV tutor.

6.3 Conclusions

Instead of seeing humans and machines in a competitive relationship, this thesis proposes a novel framework in which humans and machines work together to solve the complex problems associated with design education. This thesis specifically explores an application of artificial intelligence (machine learning and

computer vision algorithms) in which both humans and machines mutually improve their learning experiences and capabilities. Humans can increase a machine's performance by providing training-data sets that can be a foundation for intelligent decision-making. Machines can improve humans' learning performance by analyzing their behavioral patterns and providing customized instructions. This study shows that the Synthetic Tutor can improve current on-line education systems, and offer an effective alternative to off-line education by solving inherent problems in human-oriented educational environments.

Design education is a particularly promising area in which both students and instructors could benefit from this study. There are no clear guides or pedagogies that could be used to effectively teach students from all various educational and professional backgrounds, many of whom may need individualized tutoring. Students' levels of motivation, study purposes, and learning styles are all different and are changing in real-time; however, this human-machine collaboration offers an effective framework for solving these ill-structured problems which neither humans nor machines could easily solve alone.

LIST OF FIGURES

Figure 1. Machine Learning and Non-Machine Learning Tutorial Interactions. .	18
Figure 2. Example of a student’s daily sketches and unified modeling language diagrams.....	29
Figure 3. An archive of students’ daily coding exercises in the first workshop...	30
Figure 4. Example of students’ daily sketches and diagrams in the first module.	32
Figure 5. Three Students’ Four Projects Developmental Processes.	35
Figure 6. Students’ Project Examples.....	37
Figure 7. A sample final project completed in the second Spring 2012 workshop.	38
Figure 8. The project archive of the third workshop in the fall of 2012.....	39
Figure 9. Examples of three students’ projects in the third workshop.....	40
Figure 10. Examples of three students’ projects in the third workshop.....	41
Figure 11. The initial clustering procedures of k-means algorithm.....	49
Figure 12. Computing centroids of initially clustered data.....	49
Figure 13 Repeated procedures to compute updated centroids of k-means clustering.....	49
Figure 14. Three shapes as two-dimensional array data types.....	56
Figure 15. The main page of the online workshop, which shows the first 30 units.	64
Figure 16. A sample page explaining repetition.	64
Figure 17 A sample problem set given to participants.	68
Figure 18 A sample section of workshop modules.	69
Figure 19 A sample code showing a generative approach to the design of a high-rise building.	70
Figure 20 Screen-captured images of the non-ML tutor.	75
Figure 21 Screen-captured images of the ML tutor (Day 1).....	76
Figure 22 Screen-captured images from the ML tutor (Day 3).	77
Figure 23 Screen-captured images from the ML tutor (Day 6).	78
Figure 24 Screen-captured images of the ML tutor (Day 9).....	79
Figure 25 A sample coding exercise.	85
Figure 26 A sample exercise to be assigned after the daily session.	85
Figure 27. Boxplots of participants’ daily study time in its original scale (left) and in its natural logarithm scale (right).....	88

Figure 28. The histogram of a participant’s daily study time in its original scale (left) and in its natural logarithm scale (right). 89

Figure 29. The normality test plots participants’ daily study time in its original scale (left) and in its natural logarithm scale (right). As seen on the right Q-Q (normality test) plot, the normality is improved. The log-transformed sample points lie closer to the diagonal line than the sample points in their original scale. 89

Figure 30. OLS trajectories overlapped in a single plot. 91

Figure 31. Empirical growth plots for six randomly selected participants in the workshop. 92

Figure 32. Descriptive statics for daily Study Time (top) and daily Log_e Study Time (bottom) of the ML participants ($n = 137$) showing boxplots (left), histograms (middle), and Q-Q (normality test) plots (right). 94

Figure 33. Descriptive statics for daily Study Time (top) and daily Log_e Study Time (bottom) of the non-ML participants ($n = 105$) showing boxplots (left), histograms (middle), and Q-Q (normality test) plots (right). 95

Figure 34. Fitted ordinary least squares (OLS) trajectory of ML (left) and non-ML (right) tutor superimposed on empirical growth plot. 98

Figure 35. Visiting patterns of participants. 100

Figure 36. The distribution of the number of visits by participants of non-ML (left; $n = 39$) and ML tutors (right; $n = 39$). 101

Figure 37. An example of two learning patterns. 102

Figure 38. A participant’s sample code. 103

Figure 39. Computer vision tutor process. 105

Figure 40. The hardware and software setting of the developed computer vision tutor system. 106

Figure 41 The feedback from the computer vision tutor. 106

Figure 42. A participant’s floor plan and the CV tutor’s suggested floor plan. . 107

Figure 43. A participant’s second attempt and CV tutor’s suggested floor plan. 107

Figure 44. Overall satisfaction rate. 109

Figure 45. The usefulness of the CV-tutor feedback. 110

Figure 46. The relevance between the feedback and participants’ projects. 110

Figure 47. Comparison with human instructors. 111

BIBLIOGRAPHY

- Abbeel, P. (2008). *Apprenticeship learning and reinforcement learning with application to robotic control*. Stanford University, Stanford, CA, USA.
- Alpaydin, E. (2004). *Introduction to Machine Learning*. The MIT Press.
- Argall, B., Chernova, S., Veloso, M., & Browning, B. (2009). A Survey of Robot Learning from Demonstration. *Robotics and Autonomous Systems*, 67, pp. 469-483.
- Amiri, F. (2011). Programming as Design: The Role of Programming in Interactive Media Curriculum in Art and Design. *International Journal of Art & Design Education*, 30(2), pp. 200-210.
- Bundy, A. (2007). Computational Thinking is Pervasive New Kinds of Question ; New Kinds of Answer New Hypotheses ; New Theories New Thinking ; New Angles. *Thinking*, 1(2), 1-3.
- Burry, M., Datta, S. and Anson, S. (2000). Introductory Computer Programming as a Means for Extending Spatial and Temporal Understanding. in *Proceedings of ACADIA 2000*, pp.76-86.
- Celani, M. G. C. (2002). *Beyond analysis and representation in CAD : a new computational approach to design education*. PhD Thesis. MIT. Cambridge.
- Celani, M. G. C. (2008). Teaching CAD Programming to Architecture Students, in *Gestao & Tecnologia de Projetos*, Vol. 3, No 2.
- Chernova, S., & Veloso, M. (2008). Teaching collaborative multi-robot tasks through demonstration. *8th IEEE-RAS International Conference on Humanoid Robots*, 2008. Humanoids, pp. 385-390.
- Churchman, C. W. (1967). Guest Editorial: Wicked Problems. *Management Science*, 14(4), pp. 141-142.
- Coates, A., Abbeel, P., & Ng, A. Y. (2009). Apprenticeship learning for helicopter control. *Commun. ACM*, 52(7), pp. 97-105.
- Cohen, J. (1988). *Statistical power analysis for the behavioral sciences* (2nd ed.). Hillsdale, NJ: Lawrence Earlbaum Associates.

Synthetic Tutor

- Corbet, A., Koedinger, K., & Anderson, J. (1997). *Intelligent Tutoring Systems. Handbook of Human-Computer Interaction*. (2nd Ed.) Elsevier Science B.V.
- Coyne, R., (2005). Wicked problems revisited. *Design Studies*, 26(1), 5–17.
- Denis, G., & Jouvelot, P. (2004). Building the Case for Video Games in Music Education. In *Second International Computer Game and Technology Workshop*, pp. 156-161.
- Denning, P. J. (2005). Is computer science science? *Communications of the ACM*, 48(4), p. 27.
- Duarte, J. (2007). Inserting New Technologies in Undergraduate Architectural Curricula: A Case Study, in *25th eCAADe Conference Proceedings*, pp.423-430.
- Frazer, J. (1995). *An Evolutionary Architecture*. Architectural Association Publications.
- Harel, I. (1988). *Software design for learning: Children's construction of meaning for fractions and Logo programming*. Unpublished doctoral dissertation, MIT, Cambridge, MA.
- Knight, T. W. (1994). *Transformations in Design: A Formal Approach to Stylistic Change and Innovation in the Visual Arts*. Cambridge University Press.
- Kolter, J. Z., Abbeel, P., & Ng, A. Y. (2008). Hierarchical Apprenticeship Learning, with Application to Quadruped Locomotion. Retrieved from <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.4186>.
- Krawczyk, R. J. (2008). *The Codewriting Workbook: Creating Computational Architecture in AutoLISP* (1st ed.). Princeton Architectural Press.
- Kuhn, T. S. (1996). *The Structure of Scientific Revolutions* (3rd ed.). University Of Chicago Press.
- Lackney J. A., (1999). A History of Studio-based Learning Model. Retrieved January 20, 2012, from <http://www.edi.msstate.edu/studio.htm>.
- Leitao, A., Cabccinhas, F. and Martins, S. (2010). Revisiting the Architecture Curriculum: The programming perspective, in *Proceedings of eCAADe 2010*, pp.81-88.
- McCullough, M., Mitchell, W. J., & Purcell, P. (Eds.). (1990). *The Electronic Design Studio: Architectural Education in the Computer Era* (2nd Printing.). The MIT Press.

- Mills, C., & Dalgarno, B. (2007). A conceptual model for game based intelligent tutoring systems. *Citeseer*, pp. 692-702.
- Mitchell, W. J. (1975). The theoretical foundation of computer-aided architectural design. *Environment and Planning B: Planning and Design*, 2(2), pp. 127-150.
- Mitchell, W. J. (1977). *Computer-aided architectural design*. Petrocelli/Charter.
- Mitchell, W. J. (1990). *The Logic of Architecture: Design, Computation, and Cognition*. The MIT Press.
- Mitchell, W. J., Liggett, R. S., & Kvan, T. (1987). *The Art of Computer Graphics Programming: A Structured Introduction for Architects and Designers*. Van Nostrand Reinhold.
- Morris, D., & Fiebrink, R. (2011). Using machine learning to support pedagogy in the arts. Presented at *the CHI 2011 Child-Computer Interaction Workshop* at CHI 2011, Vancouver, May 7, 2011.
- Nagakura, T. (1990). Shape Recognition and Transformation: A Script-Based Approach. Retrieved September 2, 2011, from http://cumincad.scix.net/cgi-bin/works/Show?_id=e8fe&sort=DEFAULT&search=%2fseries%3a%22CAAD%20Futures%22&hits=612.
- Nagakura, T. (1996). *Form-processing: A system for architectural design*. Harvard University, United States. Massachusetts.
- National Council of Architectural Registration Board. (2011). *ARE 4.0 Study Guide: Schematic Design*. Retrieved February 27, 2012, from http://www.ncarb.org/Publications/~media/Files/PDF/Guidelines/ARE_Guidelines.pdf.
- Negnevitsky, M. (2004). *Artificial Intelligence: A Guide to Intelligent Systems* (2nd ed.). Addison Wesley.
- Negroponte, N. (1973). *The Architecture Machine: Toward a More Human Environment*. The MIT Press.
- Negroponte, N. (1976). *Soft Architecture Machines*. The MIT Press.
- Odaka, T. (2012). *First-Time Machine Learning* (in Korean). Hanbit Media Inc.

- Patel, K. (2010). Lowering the barrier to applying machine learning. *Proceedings of the 28th of the international conference extended abstracts on Human factors in computing systems*, CHI EA '10 pp. 2907–2910.
- Perkins, D.N., & Grotzer, T.A. (2005). Dimensions of causal understanding: The role of complex causal models in students' understanding of science. *Studies in Science Education*, 41, pp. 117-166.
- Raudenbush, S. W., & Bryk, A. S. (2001). *Hierarchical Linear Models: Applications and Data Analysis Methods* (2nd ed.). Sage Publications Inc.
- Rittel, H. (1972). *On the planning crisis : systems analysis of the first and second generations*. Berkeley: Institute of Urban and Regional Development.
- Rittel, H. W. J. & Webber, M. M. (1973). Dilemmas in a general theory of planning. *Policy Sciences*, 4(2), pp. 155-169.
- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6), pp. 386-408.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. In Feigenbaum, E.A. & Feldman, J. (Eds), *Computers and Thought*. New York: McGraw-Hill. pp. 71-105.
- Singer, J. D., & Willett, J. B. (2003). *Applied Longitudinal Data Analysis: Modeling Change and Event Occurrence* (1st ed.). Oxford University Press, USA.
- Simon, H. A. (1973). The structure of ill-structured problems. *Artificial Intelligence*, Vol. 4, pp. 181-201.
- Simon, H. A. (1995). Problem forming, problem finding, and problem solving in design. In Collen, A. & Gasparski, W.W. (Eds.), *Design and systems: General applications of methodology* Vol. 3, pp. 245-257.
- Simon, H. A. (1996). *The Sciences of the Artificial* (3rd ed.) The MIT Press.
- Stiny, G. (1980). *Pictorial and Formal Aspects of Shape and Shape Grammars: On Computer Generation of Aesthetic Objects* (1st ed.). Birkhäuser Basel.
- Stiny, G. & Gips, J. (1972). Shape Grammars and the Generative Specification of Painting and Sculpture. *Information Processing*, pp. 1460-1465.

- Stiny, G. (1981). A note on the description of designs. *Environment and Planning B: Planning and Design*, 8(3), pp. 257-267.
- Stiny, G. (1985). Computing with Form and Meaning in Architecture. *Journal of Architectural Education*, 39(1), pp. 7-19.
- Stiny, G. (1990). What designers do that computers should. *The electronic design studio* (pp. 17-30). MIT Press.
- Stiny, G. (2006). *Shape: Talking about Seeing and Doing*. The MIT Press.
- Sutherland, I. E. (2003). Sketchpad: A man-machine graphical communication system (No. UCAM-CL-TR-574). University of Cambridge, Computer Laboratory. Retrieved from <http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf>
- Syed, U., Bowling, M., & Schapire, R. E. (2008). Apprenticeship learning using linear programming. *Proceedings of the 25th International Conference on Machine Learning*, 307, pp1032-1039.
- Turing, A. (1936). On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings of London Math. Soc.*, 2(42), pp. 230-265.
- Turkle, S. & Papert, S. (1992). Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior*, 11(1), pp. 3-33.
- Vlist, B., Westelaken, R., Bartneck, C., Hu, J., Ahn, R., Barakova, E., Delbressine, F., et al. (2008). Teaching Machine Learning to Design Students. In Z. Pan, X. Zhang, A. Rhalibi, W. Woo, & Y. Li (Eds.), *Technologies for E-Learning and Digital Entertainment* Vol. 5093, pp. 206-217.
- Voss, J. & Post, T. (1988). *On The Solving of Ill-Structured Problems*. The Nature of Expertise. Lawrence Erlbaum Associates.
- Walsh, T. J., Subramanian, K., Littman, M. L., & Diuk, C. (2010). Generalizing Apprenticeship Learning across Hypothesis Classes. ICML'10, pp. 1119-1126.
- Weber, E. P., & Khademian, A. M. (2008). Wicked Problems, Knowledge Challenges, and Collaborative Capacity Builders in Network Settings. *Public Administration Review*, 68(2), 334-349.
- Widmer, G. (2005). Studying a creative act with computers: Music performance studies with automated discovery methods. *Musicae Scientiae*, 9(1), pp 11 -30.

Synthetic Tutor

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), pp. 33-35.

Wing, J. M. (2008). Five deep questions in computing. *Communications of the ACM*, 51(1), p.58.

Wurzer, G., Alacam, S. and Loren, W. (2011). How to Teach Architects (Computer) Programming: A Case Study, in *29th eCAADe Conference Proceedings*. pp.51-56.

Yakeley, M. (2000). *Digitally mediated design : using computer programming to develop a personal design process*. PhD Thesis. MIT. Cambridge.

APPENDIX A.

Workshop Survey Questions.

1. What is your current program?

- | | |
|---------------------------|--------------------|
| 1. Architecture | 2. Interior Design |
| 3. Landscape Architecture | 4. Design Studies |

2. Are you a graduate or an undergraduate student?

- | | |
|--------------------------|---------------------|
| 1. Undergraduate student | 2. Graduate student |
|--------------------------|---------------------|

3. What year are you in?

- | | |
|-------------|---------------------|
| 1. 1st Year | 2. 2nd Year |
| 3. 3rd Year | 4. 4th Year |
| 5. 5th Year | 6. 6th or more Year |

4. Do you have any programming experience in: (check all that apply)?

- | | |
|--------------------------------|-----------------|
| 1. Python | 2. Visual Basic |
| 3. Grasshopper (Rhino3D) | 4. C/C++ |
| 5. Java / Processing / Arduino | 6. Other |

5. Have you taken any college level introductory computer programming course?

1. I took multiple computer programming courses.
2. I took one programming course.
3. I am taking a programming course this semester.
4. No I have not. But I have some programming experience.
5. No. I have not. I don't have any programming experience.

6. Have you taken any advanced (college or graduate level) mathematics course before?

1. I took multiple advance mathematics courses.
2. I took one advanced mathematics course.
3. I am taking an advanced mathematics course this semester.
4. No, but I took related course(s).
5. No, I did not take any advanced mathematics courses.

7. What best describes your Rhinoceros 3D experience?

1. No experience.
2. Minimal experience - completed simple 2D drawings.
3. Some experience - made simple 3D models.

Synthetic Tutor

4. Substantial experience - made complex 3D models.
5. Extensive experience - made-models and used the tool for fabrication.

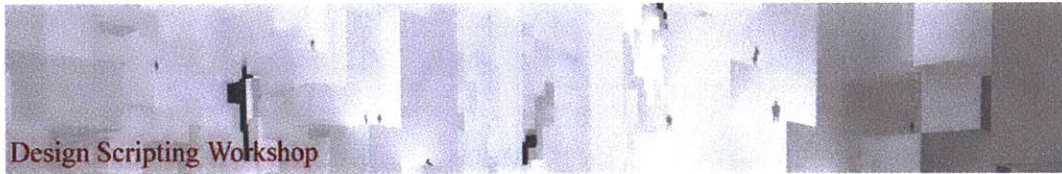
8. How long can you study this workshop every day?

1. More than 2 hours
2. 1 ~ 2 hours
3. 30 min. ~ 1 hour
4. less than 30 min.
5. Not sure.

APPENDIX B.

This appendix includes the whole set of tutorial modules including the entry and content page. The original format uses Hyper Text Markup Language (HTML) and is designed horizontally following a monitor screen's proportion. The layout of tutorials is converted into an image file format to be positioned in this thesis. Accordingly some pages do not fit well and they are rotated to be fit properly.

Entry Page



Thanks for joining the MIT Design Scripting Workshop!
This workshop is an introduction to **Rhino-Python** scripting in the context of architectural design.
It is especially designed for novice programming users. The tutorial contains detailed descriptions and step-by-step sample codes.
You may need to work about an hour every day and can complete the workshop within 10 days.
This tutorial works best with **FIREFOX** or **CHROME** browser (currently not compatible with Internet Explorer).
If you have any questions, comments, or suggestions, please contact at juhong@mit.edu

Please register your **E_MAIL** and **PASSWORD** below.

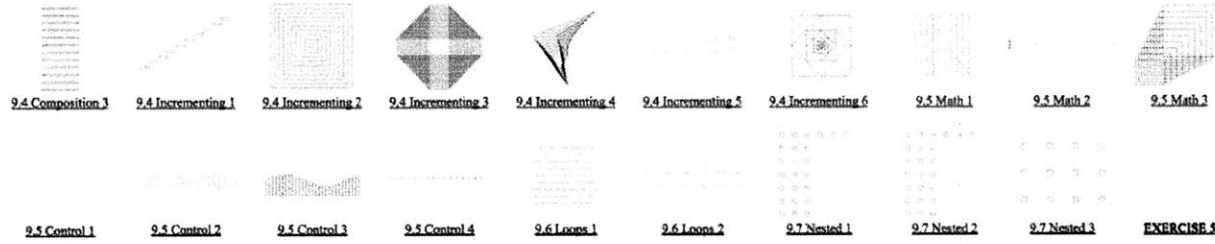
Email:
Password:

If you already registered, please login below.

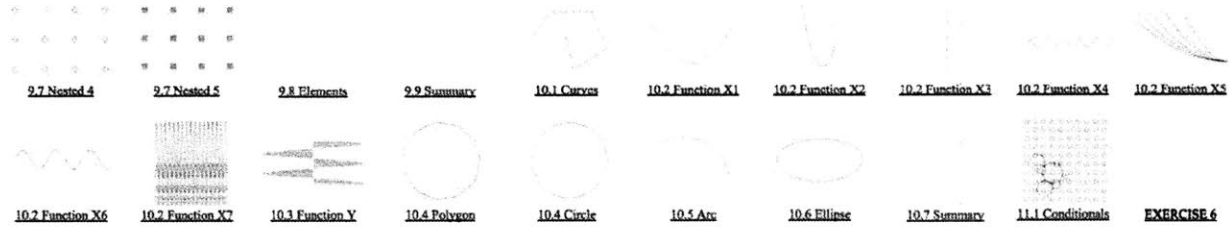
Email:
Password:

DAY 1	<u>1. Introduction</u>	<u>2. Sample Codes</u>	<u>3. Rewards</u>	<u>4. Rendering</u>	<u>5. Programming</u>	<u>5.1. Language</u>	<u>5.1. Notation</u>	<u>5.2. Syntax</u>	<u>5.3. Organization</u>	<u>5.3. Heading</u>
	<u>5.3. Data Types</u>	<u>5.3. Executable</u>	<u>5.4. Assignment</u>	<u>5.5. Arithmetic 1</u>	<u>5.5. Arithmetic 2</u>	<u>5.5. Arithmetic 3</u>	<u>5.6. Input 1</u>	<u>5.6. Input 2</u>	<u>5.6. Input 3</u>	<u>EXERCISE 1</u>
DAY 2	<u>5.7. Punctuation</u>	<u>5.8. Comments</u>	<u>5.9. Styles</u>	<u>5.10. Clarity</u>	<u>5.11. Summary</u>	<u>6.1. Line</u>	<u>6.2. Library</u>	<u>6.3. Variables</u>	<u>6.3. GetReal</u>	<u>6.4. Parts</u>
	<u>6.5. Summary</u>	<u>7.1. Variables</u>	<u>7.2. Math 1</u>	<u>7.3. Rounding</u>	<u>7.3. Math 2</u>	<u>7.4. Function 1</u>	<u>7.4. Function 2</u>	<u>7.5. Parameter</u>	<u>7.6. Local Var 1</u>	<u>EXERCISE 2</u>
DAY 3	<u>7.6. Local Var 2</u>	<u>7.7. Trigonometry</u>	<u>7.8. Summary</u>	<u>8. Graphic</u>	<u>8.1. Picture</u>	<u>8.2. Parts</u>	<u>8.3. Procedure 1</u>	<u>8.3. Procedure 2</u>	<u>8.3. Abstraction</u>	<u>8.3. Param 1</u>
	<u>8.3. Param 2</u>	<u>8.3. Param 3</u>	<u>8.3. Param 4</u>	<u>8.3. Param 5</u>	<u>8.3. Param 6</u>	<u>8.3. Invoking</u>	<u>8.4. Shape</u>	<u>8.4. Position 1</u>	<u>8.4. Position 2</u>	<u>EXERCISE 3</u>
DAY 4	<u>8.4. Parameter 1</u>	<u>8.4. Parameter 2</u>	<u>8.4. Parameter 3</u>	<u>8.5. Proportion</u>	<u>8.6. Vectors</u>	<u>8.7. Variable</u>	<u>8.8. Defining</u>	<u>8.8. Vocabulary</u>	<u>8.9. Summary</u>	<u>9.1. Repetition</u>
	<u>9.2. Control</u>	<u>9.2. Repetition</u>	<u>9.2. Loop 1</u>	<u>9.2. Loop 2</u>	<u>9.2. While</u>	<u>9.2. Break</u>	<u>9.3. Variations</u>	<u>9.4. Composition 1</u>	<u>9.4. Composition 2</u>	<u>EXERCISE 4</u>

DAY 5



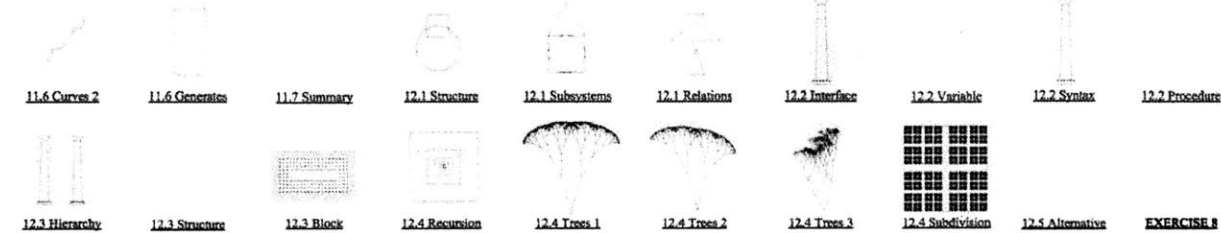
DAY 6

































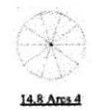







DAY 7



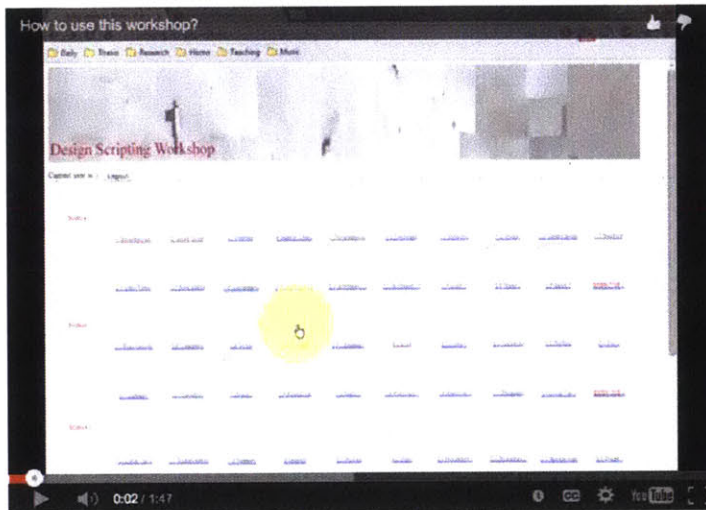
DAY 8



DAY 9									
									
<u>12.5 Top Down</u>	<u>12.6 Summary</u>	<u>14.1 Transform</u>	<u>14.2 Data Structure</u>	<u>14.3 Procedure 1</u>	<u>14.3 Procedure 2</u>	<u>14.3 Procedure 3</u>	<u>14.3 Procedure 4</u>	<u>14.3 Procedure 5</u>	<u>14.3 Concat_2</u>
									EXERCISE 9
<u>14.3 Concat_2</u>	<u>14.3 Order</u>	<u>14.3 Operators</u>	<u>14.4 Hierarchy</u>	<u>14.4 Graphic</u>	<u>14.5 Symmetry</u>	<u>14.5 Rotational</u>	<u>14.5 Bilateral</u>	<u>14.5 Dihedral</u>	EXERCISE 9
DAY 10									
									
<u>14.5 Frieze 1</u>	<u>14.5 Frieze 2</u>	<u>14.5 Wallpaper 1</u>	<u>14.5 Wallpaper 2</u>	<u>14.5 Wallpaper 3</u>	<u>14.6 Parameters</u>	<u>14.7 Repetition 1</u>	<u>14.7 Repetition 2</u>	<u>14.7 Repetition 3</u>	<u>14.7 Repetition 4</u>
									FINAL TEST
<u>14.8 Arcs 1</u>	<u>14.8 Arcs 2</u>	<u>14.8 Arcs 3</u>	<u>14.8 Arcs 4</u>	<u>14.8 Arcs 5</u>	<u>14.8 Arcs 6</u>	<u>14.8 Arcs 7</u>	<u>14.8 Arcs 8</u>	<u>14.9 Composition</u>	FINAL TEST

Module 1: Introduction

INTRODUCTION



This tutorial consists of ten days sessions. Each daily session contains twenty small modules. Each daily work may take about one hour.

Click a text or a thumbnail image to enter the module. Any module with a thumbnail image contains a sample python code that you can run and test. Modules without a thumbnail do not have sample codes.

[7.3 Parameter](#) [7.6 Local Var 1](#)

Each module has four parts: 1. Description, 2. Diagram (figures), 3. Sample Code, and 4. Code Results. To see the result images of sample codes, you need to use a scroll bar on the bottom to see a result image. Some modules only have Descriptions and Figures.

Synthetic Tutor

The screenshot displays the Synthetic Tutor interface with four main panels:

- Description:** Contains text explaining the Fibonacci sequence and its relation to the golden ratio. It includes instructions for the user to write the effect of multiplying the Fibonacci sequence.
- Figures:** Shows a vertical sequence of Fibonacci numbers (1, 1, 2, 3, 5, 8, 13, 21) and their corresponding geometric representations (squares and rectangles) that form a spiral. Below this, there are two diagrams: a square with a cross and a square with a spiral.
- Sample Code:** Displays a snippet of C++ code for calculating the Fibonacci sequence.
- Result Image:** Shows a circular image of a dark, textured sphere with a spiral pattern on its surface.

At the bottom of the interface, there is a navigation bar with a "Back to Contents" button and a feedback prompt: "How do you evaluate this content? Useless 1 2 3 4 5 Highly Useful".

To go back to the main content page, click **Back to Contents** button.

You can alternatively use **Back** button in menu.

You may see a confirmation window. Simply click **Leave this Page** button.



Back to Contents How do you evaluate this content? Useless 1 2 3 4 5 Highly Useful

6. If you have any problems, or are experiencing any delays, just refresh your page by clicking **F5** button. You may need to re-login sometimes.

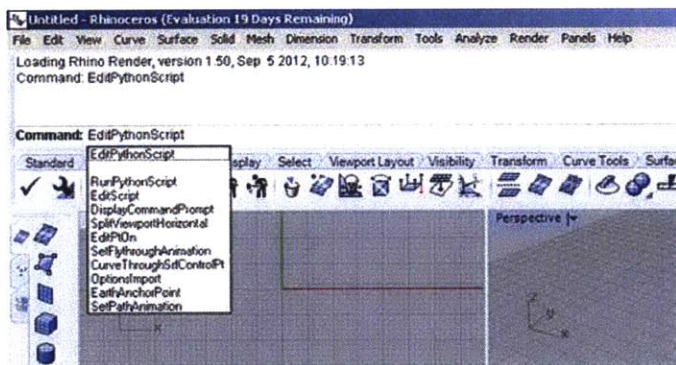
Module 2: Sample Code

EXERCISING SAMPLE CODES

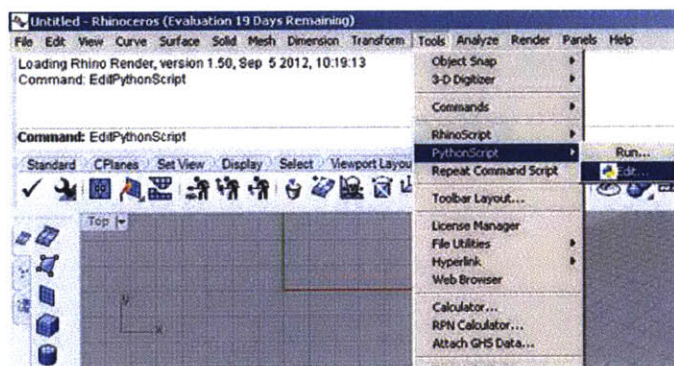
Exercising and running the provided sample codes will significantly improve your learning programming. Here are the instructions for you to follow to run the provided sample codes.

For MAC users, before trying this module, please install the IronPython.mcrhi plug-in. check this link: wiki.mceneel.com/rhino/mac/python.

1. After installing **Rhinoceros 5**, open the software.
2. To open a text editor
 - a. Type **EditPythonScript** on the command line.

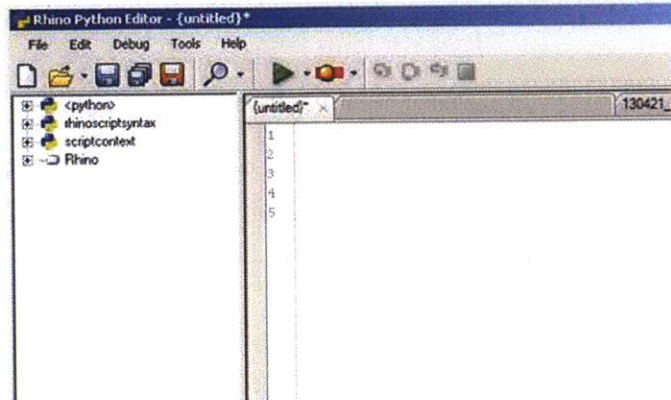


- b. Or, click **Tools / PythonScript / Edit**.



3. You will see the Python Script Editor below.

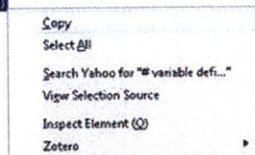
Synthetic Tutor



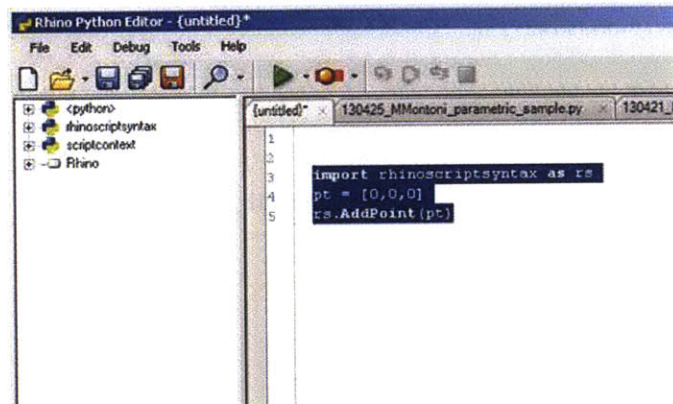
4. You can drag and copy (use **ctrl + c**) sample codes from this tutorial (use the sample code on the right side)

CODE

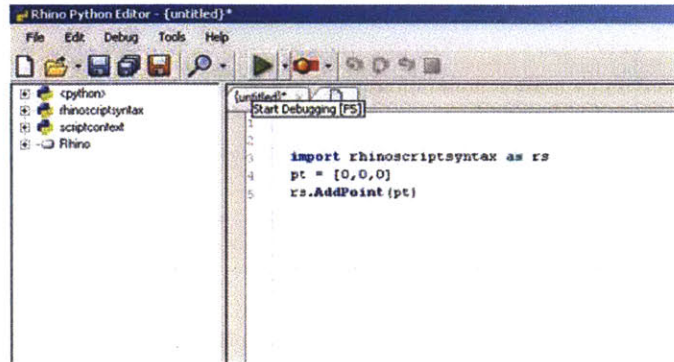
```
import rhinoscriptsyntax as rs
pt = [0,0,0]
rs.AddPoint(pt)
```



5. Then, paste codes into the Rhino Python Editor (use **ctrl + v**).



6. To run the sample code,
 - a. Hit **F5**.
 - b. Or, click the green triangle to start debugging.



7. You can see the generated shapes. If you cannot see anything, just zoom out your screen (use a scroll wheel on your mouse).

Module 3: Interface

INTERFACE

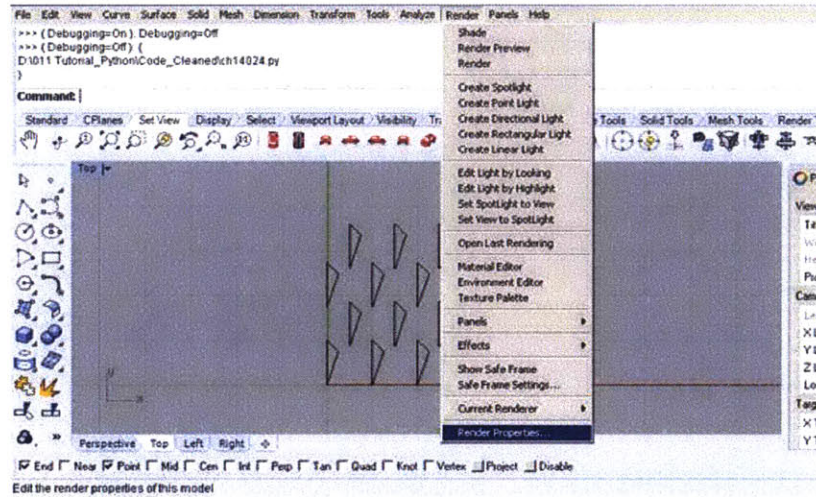
1. The purpose of this workshop tutorial is measuring the effectiveness of teaching materials for novice programmers in the context of architectural design. Please evaluate each contents. Your evaluations are invaluable information that identify the learning performance of this tutorial.
2. This workshop tutorial contains many modular teaching contents. If you click a title or an image, you can access the corresponding content.
3. **[Important] Please use [Goto Index](#) button to go back to the index page. It is important because the button makes this tutorial remember the contents you visited. Please do not use the [Back] button of your browser.**
4. Whenever you finish your learning, please logout from the tutorial.
5. This workshop will finish at May 31, 2013. The upgraded version will be published in this Fall.

Module 4: Rendering

HOW TO MAKE A RENDERED IMAGE OF YOUR SHAPES:

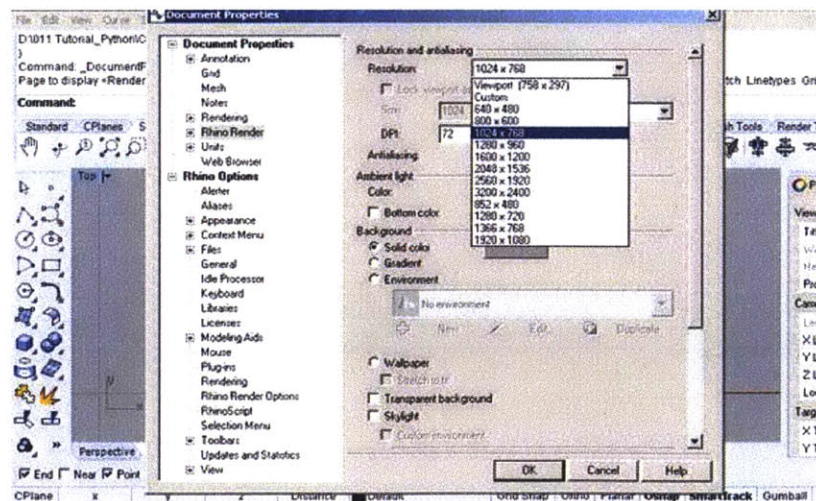
(You may need to render your work to submit a JPG file for daily exercises and the final test project)

1. After generating a form, click Render > Render Properties to change your rendering settings.



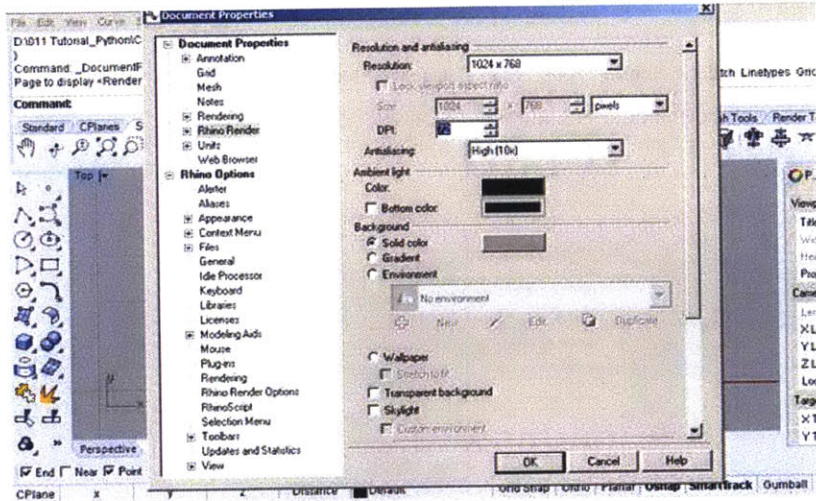
2. To change resolution, click the Resolution box in the Resolution and antialiasing section.

We recommend a maximum resolution of 1024 x 768.

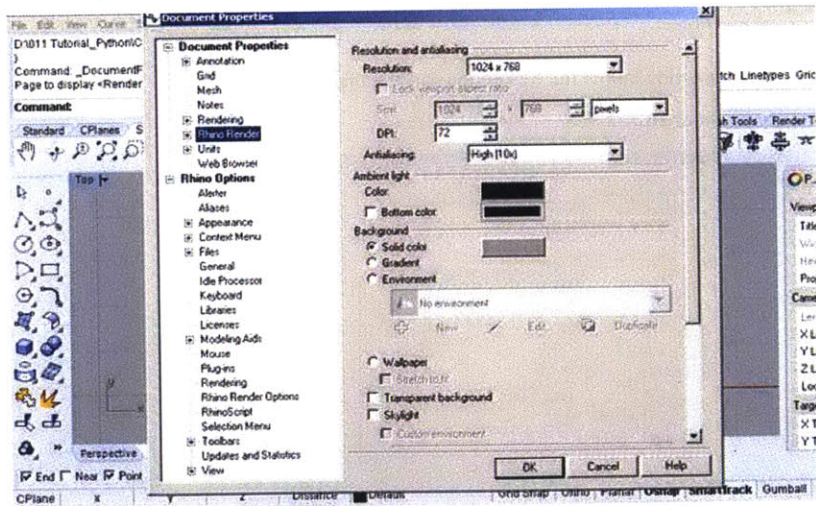


Synthetic Tutor

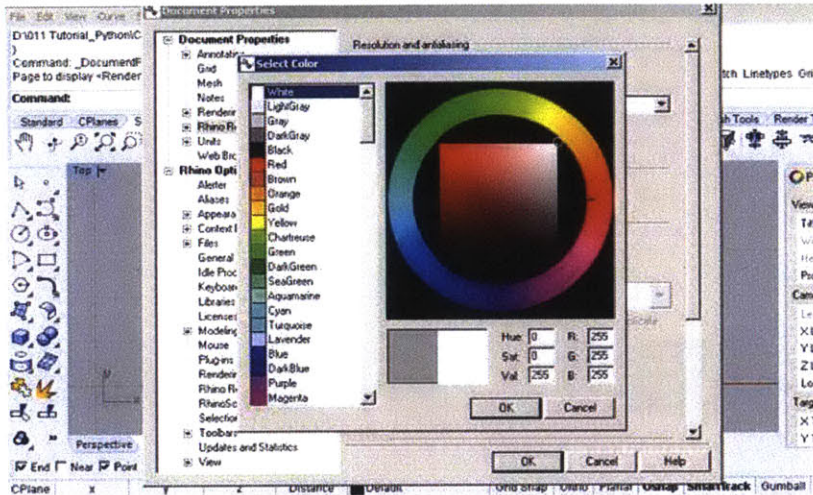
3. Set the value of DPI as 72 and Antialiasing as **High(10x)**



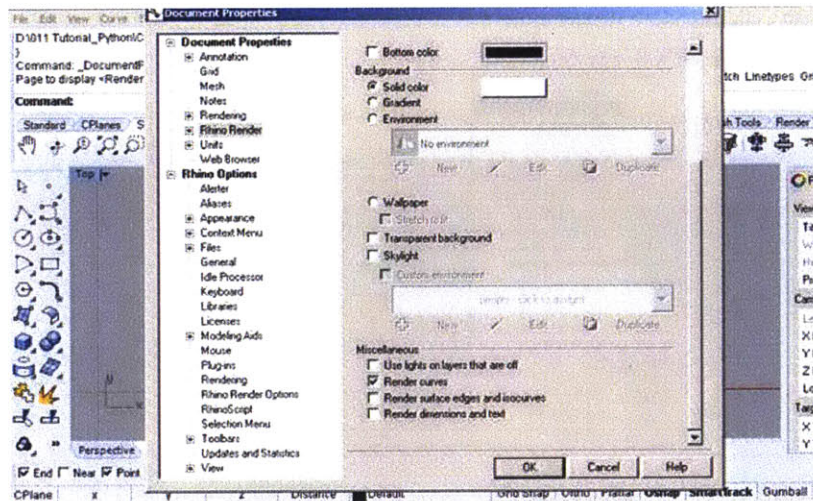
4. To change the background color, click the gray color box next to **Solid Color** under the **Background** section.



5. Select **White** color.

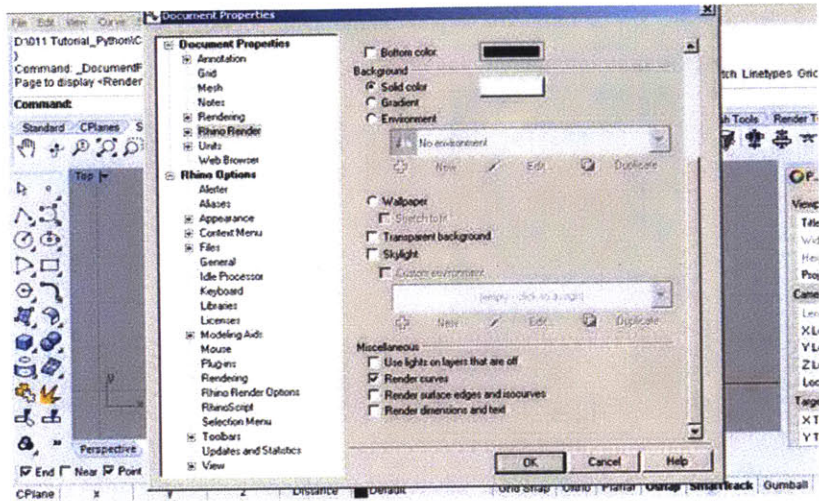


6. To render lines, check the Render curves box under the Miscellaneous section.

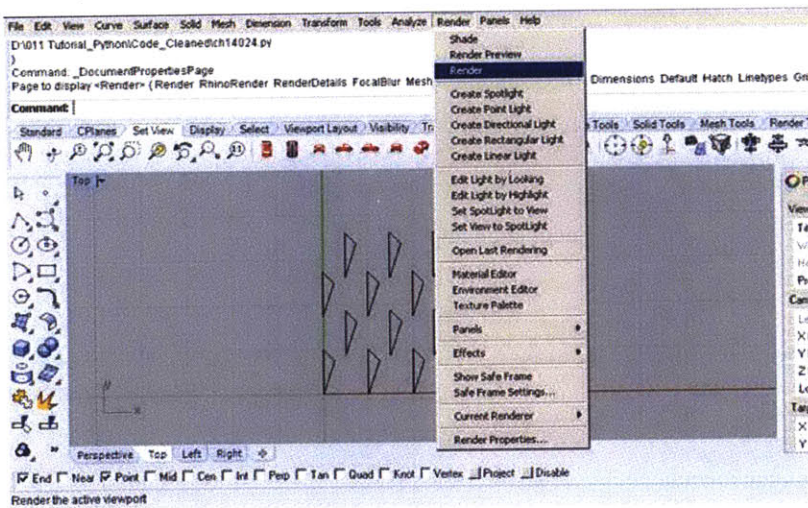


7. Then, select OK and finish settings.

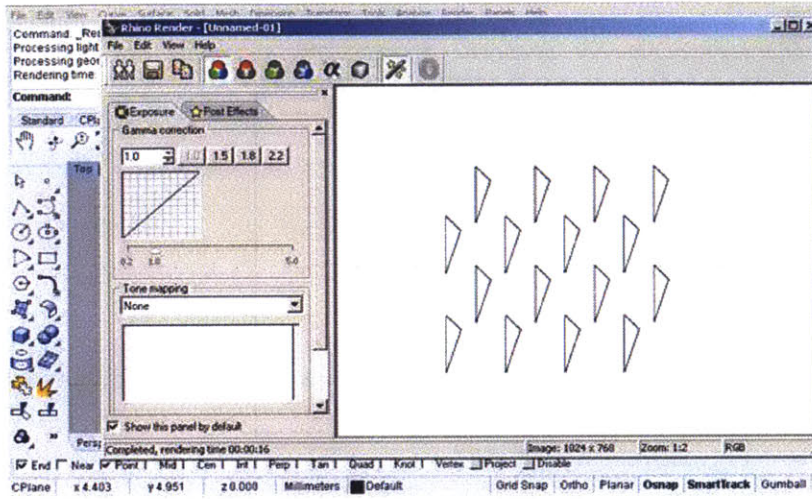
Synthetic Tutor



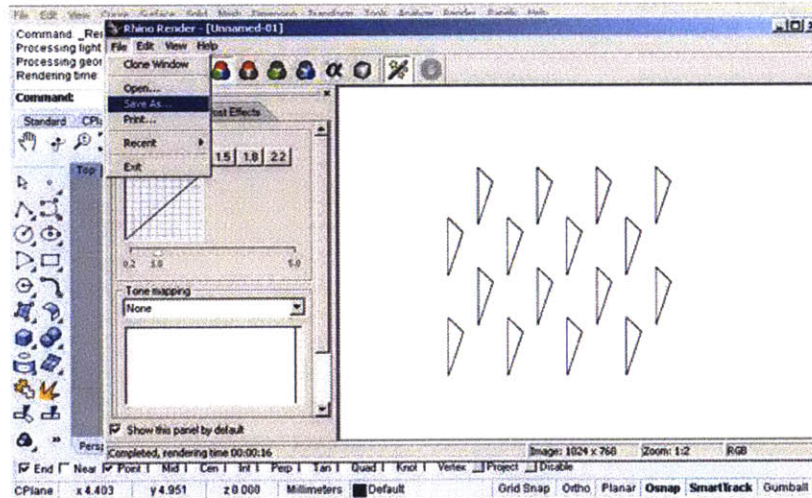
8. Finally, to render your image, click Render > Render menu.



9. You will see the rendered image in the Rhino Render window.

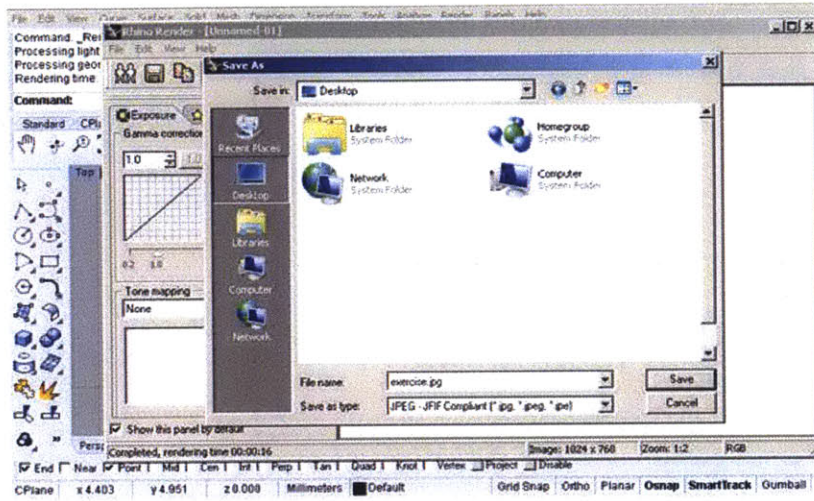


10. To save your rendered image, click File > Save As... menu.



11. Save your file as JPG type under a folder where you can easily remember.

Synthetic Tutor



Module 5: Programming

5. THE PYTHON PROGRAMMING LANGUAGE

5.1 WHAT IS A COMPUTER PROGRAM?

We have introduced computer programs as sequences of instructions that computers follow in order to produce useful results - much as a cook might follow a recipe, a musician a score, or a cab driver a sequence of directions. The remainder of this book will focus on how to write programs that, when executed by a computer, generate pictures on some kind of display or plotter. But before going into the details, we will describe more precisely what a computer program really is.

5.2 THE LANGUAGE

Any computer program is written in some particular language. The language (unlike a natural language such as English) has a precisely specified vocabulary and syntax that must be followed rigorously. The semantic properties are also well defined; any syntactically correct statement causes the computer to perform some specific action. Thus a programming language provides a very precise means of communication and requires you to express yourself exactly; there is no latitude for the vagueness, incompleteness, ambiguities, and errors that we tolerate in everyday speech.

Hundreds of computer languages have been developed for different types of applications, different styles of expression, and different types of computers. Ideas about computing have progressed over the years, so modern programming languages are often more sophisticated than older ones.

Module 6: Language

5. THE PYTHON PROGRAMMING LANGUAGE

5.2.1 NOTATION AND VOCABULARY

Python, like any computer language, has a vocabulary of symbols and reserved words that are used to form meaningful statements. The symbols used in Python are shown in figure 5-1.

They consist of operators and delimiters. The operators +, -, *, and / (add, subtract, multiply, and divide, respectively) are familiar to everybody who has worked with four-function hand calculators. Notice that the asterisk is used instead of a multiplication sign to denote multiplication; this avoids confusion with the letter x and is standard in programming languages. We will explain the other operators later in this chapter, as the need arises. The delimiters function much like punctuation marks in English sentences, or parentheses in mathematical expressions. The reserved words used in Python are listed in figure 5-2. We will explain each of these as we proceed.

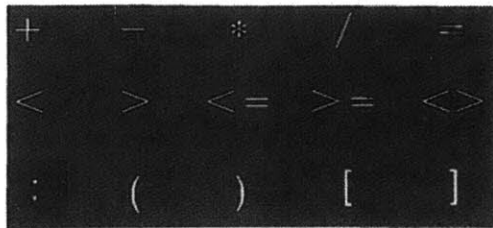
Python statements may contain not only symbols and reserved words, but also identifiers. These are names chosen by the programmer for programs, variables, and other entities to which a user must refer. For example, a program that draws a square might be called `square`, and a variable might be called `x`. Think of memory as an array of pigeonholes containing entities that you will want to manipulate and identifiers as labels placed on the pigeonholes so that you can refer to them by name (fig 5-4).

Under the conventions of Python, an identifier must begin with a letter, which may be followed by a string, of any length and in any sequence, of letters and digits. Since use of longer identifiers often improves the readability of a program, we will use identifiers of up to sixteen characters in this book. Sometimes it is typographically convenient to insert a break in an identifier. Blank spaces are not allowed, however, so you must use an underscore thus: `Villa_Malcontenta`. The following are examples of acceptable identifiers:

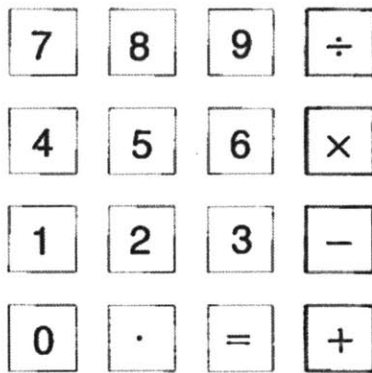
```
x
y
root2
h20
tom_kvan
triangle
very_very_very_long
```

The following are examples of unacceptable identifiers:

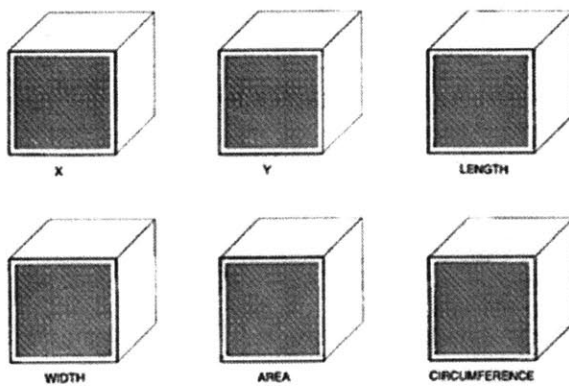
```
2nd (begins with a numeral)
Robin liggett (contains a blank space)
Sheet.3 (contains a symbol)
```



5-1. The symbols



5-2. The keyboard of a four-function hand calculator, showing the four operator keys.



5-4. Identifiers are labels used to refer to entities stored in memory.

Module 7: Notation

5. THE PYTHON PROGRAMMING LANGUAGE

5.2.1 NOTATION AND VOCABULARY continued.

Python also has some standard identifiers, which refer to entities predefined in Python. The list and form of standard identifiers may vary a little among programming languages. In this book, we will use those shown; you should check them against the list for the language that you will be using. We shall return, later, to the uses of these standard identifiers.

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

Standard identifiers List

Python statements may also contain numbers. We shall be concerned here with two types of numbers-integers written in the standard way, for example,

```
1
23
992
1024
-10
```

and real numbers written in decimal notation thus:

```
0.0
3.14159
1000.001
-10.0
```

At least one digit must precede the decimal point, where it is used, and one must follow it, but whole numbers may be written without the decimal point. Do not include commas when writing numbers.

Blanks and line breaks are used in Python as separators. At least one separator must be inserted between any pair of consecutive words or numbers, in order to avoid ambiguity. But separators cannot be inserted within words or numbers. Aside from these necessary restrictions, separators should be used freely to achieve typographic clarity in programs.

Standard Python allows the free use of both uppercase and lowercase characters for legibility and to accommodate stylistic preferences. Some Python programmers confine themselves to lowercase, others to uppercase, and some like to capitalize according to their own rules of typographic style.

We shall also find it convenient to use boldface and italic as well as standard characters. In particular, Python reserved words and standard identifiers will be printed in boldface. Special text editors used for Python programming often do this automatically, which improves the legibility of a program. Interpreters and compilers ignore the distinction when processing Python code.

The following is an example of a simple Python statement that illustrates the conventions of notation and vocabulary that have now been introduced:

```
Sum = 2 + 3
```

This means that the variable Sum is given the value of the result of adding the integers 2 and 3. The components of the expression are:

```
Sum (name of a variable)
= (the assignment operator)
2 (integer number)
+ (the addition operator)
3 (integer number)
```

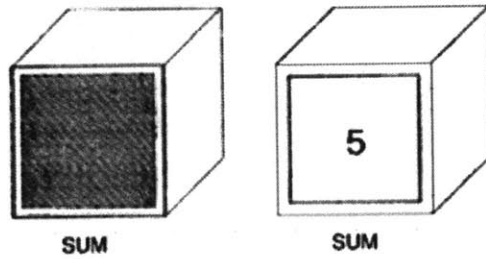
Figure 5-6 illustrates the memory location denoted by SUM before and after execution of this statement. Here is another example:

```
Circumference = Diameter * 3.1417
```

In this case, the components are:

```
Circumference (name of a variable)
= (the assignment operator)
Diameter (name of a variable)
* (the addition operator)
3.1417 (real number)
```

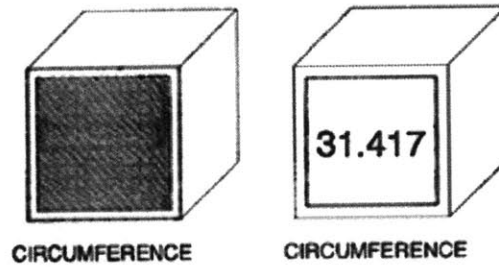
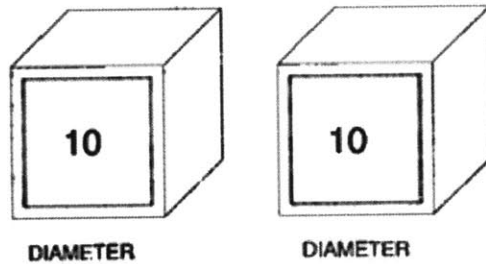
Notice (fig. 5-7) that the value of the variable Circumference after execution of this statement will depend on the value of the variable Diameter before execution.



a. Before execution of the statement.

b. After execution.

5-6. Value stored in the memory location denoted by Sum.



a. Before execution

b. After execution.

5-7. The result of execution of a statement.

Module 8: Syntax

5. THE PYTHON PROGRAMMING LANGUAGE

5.2 THE LANGUAGE

5.2.2 SYNTAX

The symbols, reserved words, identifiers, and numbers that constitute Python statements are put together according to certain rules of syntax. Rules of syntax also govern the ways in which statements themselves may be put together to form larger units of Python code. In other words, Python is a language, like English or French, with its own rules of grammar.

All of our examples will be programs and parts of programs that generate drawings on display devices, and we will approach programming from the viewpoint of the designer or graphic artist. That is, we shall analyze the logic of pictorial composition, then show how graphic compositional rules and principles correspond to constructs available in Python. You will learn to think about drawings in terms of these programming constructs - not only a useful technical skill, but profoundly illuminating in itself.

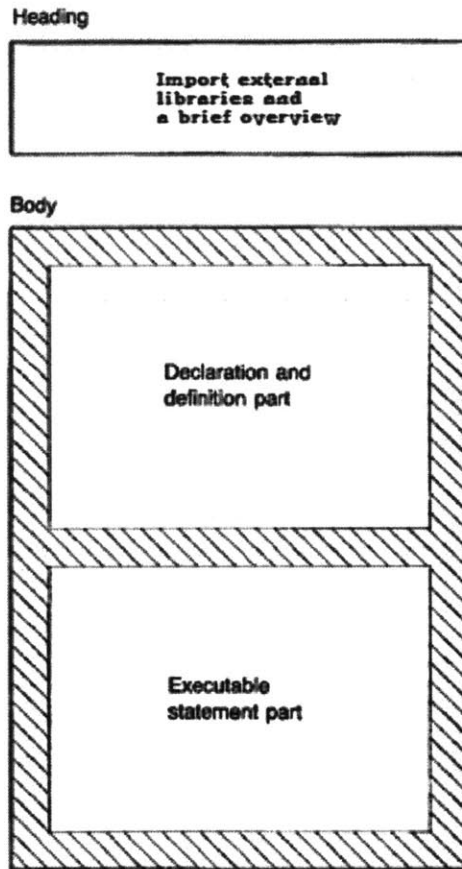
Module 9: Organization

5. THE PYTHON PROGRAMMING LANGUAGE

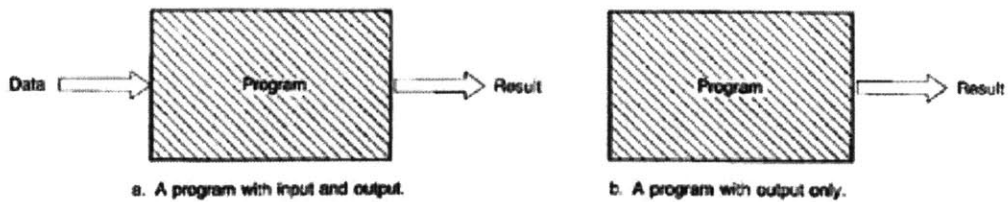
5.3. THE GENERAL ORGANIZATION OF A PYTHON PROGRAM

Any Python program has two essential parts: a description of the data to be operated upon, and a specification of the actions to be performed upon the data to achieve the desired result. Data are described by statements called declarations and definitions, whereas actions are specified by statements called executable statements or commands. In other words, declarations and definitions, like declarative English sentences, describe how something is. But executable statements, like imperative English sentences, specify something to be done. Both declarations and executable statements may be referred to, more generally, as statements.

A Python program is sometimes divided into a heading and a body. The heading includes information about the program and external libraries such as `rhinoscriptsyntax` and `math`. The body, in turn, consists of a declaration and definition part followed by an executable statement part. Declarations and definitions are thus clearly segregated from executable statements. Figure 5-8 illustrates this organization.



5-8. A Python program comprises a heading and a body. The body consists of the declaration and definition parts, followed by an executable statement part.



5-9. Programs and data.

Module 10: Heading

5.
THE PYTHON
PROGRAMMING LANGUAGE

5.3. THE GENERAL ORGANIZATION OF A PYTHON PROGRAM

5.3.1 THE HEADING

Here is an example of a heading:

```
import rhinoscriptsyntax as rs
import math

# The purpose of this program is to draw a window.
# Date: May 20, 2013
# Author: James Kee
```

Headings normally has two information: 1. external libraries and 2. an overview of the program. External libraries start with a reserved word - import - comes first followed by chosen library names (in this case rhinoscriptsyntax and math). A brief overview uses comments that describe basic information about the program such as purpose, date, author, and any information that you may want to remember. # is used to denote that the sentence is a comment. Comment is for human understanding. It will be ignored while other codes are being executed.

Module 11: Data Types

5. THE PYTHON PROGRAMMING LANGUAGE

5.3. THE GENERAL ORGANIZATION OF A PYTHON PROGRAM

5.3.3 PYTHON DATA TYPES

Python recognizes certain very specific types of data that may be stored in memory and operated upon by a program. We have already encountered the type's integer and real. Let us now consider these in more detail. The arithmetic of integers has certain rules, and these are reflected in the definitions of Python operators that can apply to integers. For example, the operators

- * multiply
- + add
- subtract

always yield integers when applied to integers, but the operator

/ divide

will yield a real number.

When at least one of the operands (numbers operated upon) of the multiply, add, or subtract operators is of type real, the result is always a real number. With the divide operator, the result is real even if both operands are integers, and the result turns out to be a whole number. Logical inconsistencies and errors follow if such rules (and there are many more of them, as we shall later see) are not rigorously followed. You will generate an error message if you attempt to inappropriately apply an operator.

Real numbers can have an indefinitely long decimal part, but can be represented only to a finite number of decimal places (that is, to finite precision) in a computer. Thus the value of

1.0 / 3.0

in Python is not the infinite sequence

0.333 . .

but a finite sequence, the length of which may vary from computer to computer.

A third data type in Python is called Boolean. A variable of this type stores the logical values false and true. You cannot apply arithmetic operators, such as * and + to a Boolean variable, and of course you cannot give an integer or real value to a Boolean variable. But you can, for example, form expressions out of Boolean variables and logical operators such as:

- and - logical conjunction
- or - logical disjunction
- not - logical negation

In other words, you can do Boolean logic in Python programs with Boolean variables - just as you can do integer arithmetic with integer variables and real arithmetic with real variables. We shall consider some interesting applications of Boolean variables and logical operators in chapter 11.

Synthetic Tutor

A fourth Python data type is char (character). A variable of this type stores a single text character. Char variables are used in programs that manipulate text. Since this book is mostly concerned with graphics, we will rarely use char variables.

There is much more to be said about Python data types, but this will suffice to introduce them. We will take up the topic again in chapter 13 when we consider the storage of drawings in data structures.

Module 12: Executable

5. THE PYTHON PROGRAMMING LANGUAGE

5.3. THE GENERAL ORGANIZATION OF A PYTHON PROGRAM

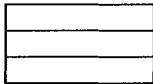
5.3.4 THE EXECUTABLE STATEMENT PART

The executable statement parts of a Python program are sentences that Python executes line by line. In our example, the first two executable statements give values to the variable's Length and Width, respectively. The computer is then instructed to calculate the Area of the rectangle and print the results.

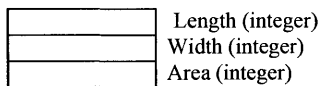
```
Length = 20
Width = 20
Area = Length * Width
```

Note the way that the variables are handled. First they are named in the declaration and definition part. This sets aside space in memory for storing values. Within the executable statement part of a program, variables should be given values. When a variable is given a value, an integer, a real number, a Boolean value, or a char value (depending on the type of variable) is stored in the corresponding location. Once a variable has been given a value, this value may be used in executable statements. When a variable's value is used, the computer looks in the memory location identified by the variable name and retrieves the value stored at that location. Finally, in this example, the statement written (Area) causes the computer to look in the memory location labeled Area and print out what it finds there.

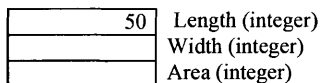
To understand our example program in these terms, imagine three empty memory locations:



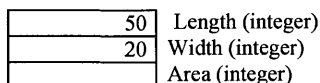
The variable definition part of our program labels each one and specifies the type of data that can be stored there: (Python takes care of this type handling operation internally)



Execution of the statement Length = 50 generates the following result:



Then execution of Width = 20; gives the result:



Next, execution of Area = Length * Width; gives the result:



Synthetic Tutor

50	Length (integer)
20	Width (integer)
1000	Area (integer)

Some basic rules of program organization and use of variables have now been illustrated. They may be summarized as follows:

A variable cannot be given a value or otherwise used in an executable statement until it has been named and typed in the declaration and definition part. If an attempt were made to use a value of a variable that had not yet been declared, the computer would not know where in memory to look for it. If an attempt were made to store a value of a variable that had not yet been declared, the computer would not know where to put it.

The type of a variable in Python actually is flexible and can be changed. A value of one type may be given to a variable of another.

A variable may not be used in an executable statement unless the execution of the statement gives it a value, or unless it has already been given a value by the execution of a previous statement. If it were used without having been given a value, the computer would look in the memory location identified by the variable name and find nothing there.

Certain rules, following from the logical properties of various types of data, must be followed in expressions that manipulate the values of variables.

Module 13: Assignment

5. THE PYTHON PROGRAMMING LANGUAGE

5.4. ASSIGNMENT

Against this background, let us now look more carefully at the sorts of executable statements that appear in our example program. The most fundamental of all executable statements is the assignment statement, which assigns a value defined on the right side of the statement to the variable named on the left side. Here is an example:

```
Length = 50
```

We have, first, the name of a variable, followed by the assignment operator =, then a number. The assignment operator = should not be confused with the equality operator ==, which as we shall see later, has its uses in logical expressions. The assignment operator is pronounced gets, while the equality operator is pronounced equals. In this example, an integer value is assigned to an integer variable. Similarly, a real value can be assigned to a real variable,

```
A_Real = 3.14159
```

a Boolean value can be assigned to a Boolean variable,

```
Beautiful = True  
Maybe = False
```

and a character can be assigned to a char variable,

```
Answer = 'y'
```

Notice that the character must be delimited by single or double quotes, as shown.

Whereas declaration of a variable is like creating and labeling an empty pigeonhole, assignment is the operation of putting something in that pigeonhole.

Module 14: Arithmetic 1

5. THE PYTHON PROGRAMMING LANGUAGE

5.5. ARITHMETIC

Instead of a single number on the right side of an assignment statement, as in the examples above, there may be an arithmetic expression constructed using the arithmetic operators available in Python:

```
Length = 3 + 4 - 5
```

In this case, the expression is first evaluated by the computer, then the result is assigned to the variable. So this statement is equivalent to:

```
Length = 2
```

In general, an arithmetic expression is a rule for calculating a value by applying arithmetic operators according to the standard conventions of algebra. The most fundamental arithmetic operators in Python are those of a four function calculator:

```
* multiply  
/ divide  
+ add  
- subtract
```

```
Expression  
2 + 3 * 4  
3.785 / 9.001  
2 * 3 - 4 * 5  
17 / 3 * 3  
1 / 2
```

Another operator provided by Python, which we sometimes need in integer arithmetic, is % (mod). This function yields the integer division remainder, for example:

```
Expression  
33 % 4  
22 % 10  
10 % 11
```

Mixed mode arithmetic, in which some operands in an arithmetic expression are integer and some are real, is allowed in Python and is sometimes convenient. Here are the rules that govern:

These rules may seem arbitrary at first, but they follow directly from sound mathematical thinking. The integers form a proper subset of the real numbers, so we can always use an integer as an operand. But you can only be sure of an integer result under certain circumstances. If it is not certain that a result will be an integer, then Python takes the safe position and presumes that the result is real.

Parentheses may be used, in the usual way, to group sub expressions- either where this is necessary to define how the expression is to be evaluated, or where it is not strictly necessary, but makes the expression easier to understand. Here are some examples:

Expression

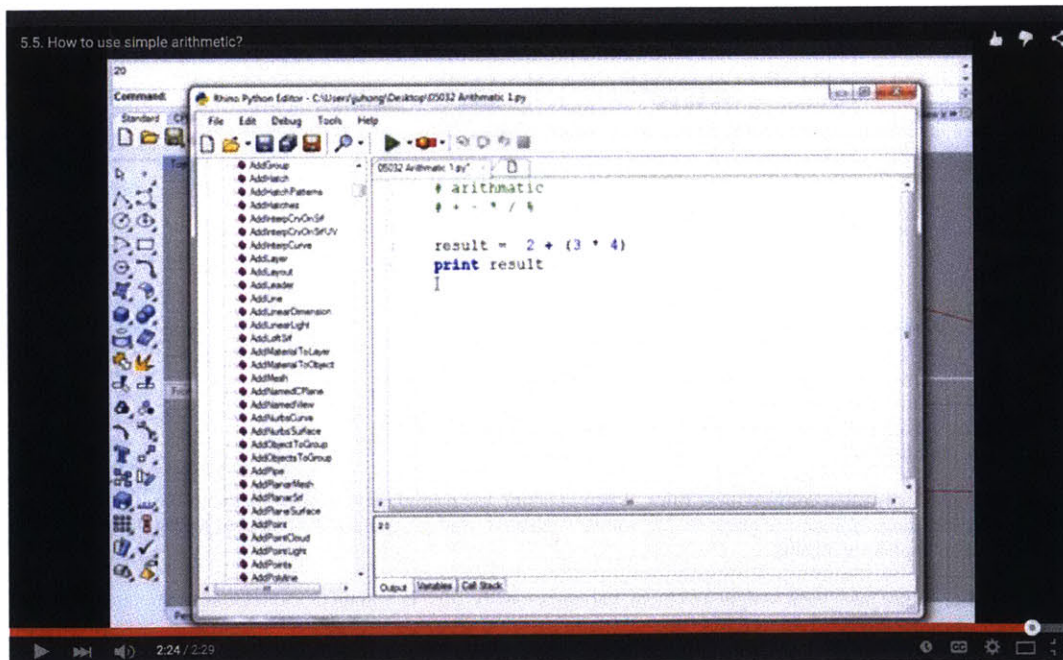
$2 + (3 * 4)$

$(2 + 3) * 4$

$(2 + 3) - (4 * 5)$

$2 * (3 - 4) * 5$

$2 * ((3 - 4) + (5 * 6)) - 7$



Module 15: Arithmetic 2

5. THE PYTHON PROGRAMMING LANGUAGE

5.5. ARITHMETIC (continued)

Certain conventions of operator precedence are followed in the evaluation of expressions. These may be summarized as follows:

The computer scans the expression from right to left.

Whenever an operand has an operator on both sides, for example, the number 2 in the expression $1 + 2 * 3$, a priority scheme is applied. The multiplying operators *, /, and, div all have the same, highest priority. The adding operators + and - both have the same lower priority. Thus, the expression $1 + 2 * 3$ is equivalent to $1 + (2 * 3)$.

Parentheses may be used to remove ambiguity, or to change the priority that would result from the conventions of left-to-right evaluation and the priority scheme. For example, use of parentheses in the expression $1 + (2 * 3)$ merely removes ambiguity, whereas the use of parentheses in the expression $(1 + 2) * 3$ changes the priority.

An expression enclosed within parentheses is evaluated independently of succeeding or preceding operators. Expressions with nested parentheses are evaluated from the inside out. Thus the expression:

$$2 * ((3 - 4) + (5 * 6)) - 7$$

Is first reduced to:

$$2 * (-1 + 30) - 7$$

Then, by left-to-right scanning and operator priority, we get:

$$58 - 7$$

Finally, subtraction yields the result:

$$51$$

This is illustrated by the tree diagram in figure 5-11.

Consecutive multiplication and/or division operations are performed from left to right.

Consecutive addition and/or subtraction operations are performed from left to right.

Novice programmers often confuse these rules. When in doubt, use parentheses for clarification. Never write an expression that leaves you unsure about how it will be evaluated. And never make the reader of a program puzzle about what an expression means.

Provided that values have been assigned to them, variables may be used in arithmetic expressions, for example:

$$\text{Length} * \text{Width} \\ (X + Y) * 5$$

$$2 * P1 * \text{Radius}$$

A name appearing in an expression may also be a constant that has previously been defined.

It is often clearer (and therefore better) to write a sequence of assignment statements with simple arithmetic expressions on the right-hand sides, rather than a single assignment statement with a complicated expression on the right hand side. For example, we might write:

$$\begin{aligned} X &= A + B - C \\ Y &= D * (E - F) \\ Z &= (X + Y) * 5 \end{aligned}$$

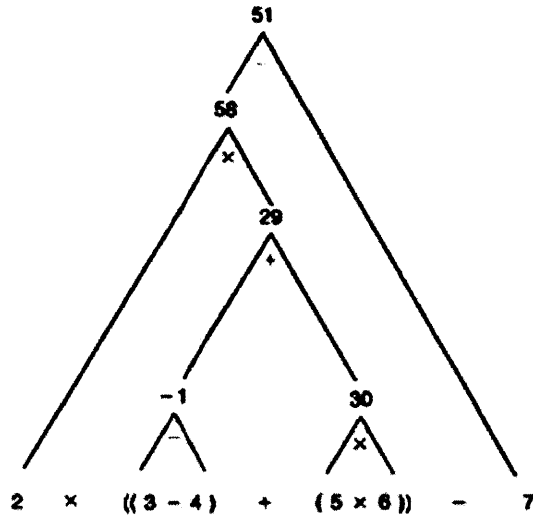
in place of the equivalent:

$$Z = ((A + B - C) + (D * (E - F))) * 5$$

Where there are many parentheses, as in this last example, you have to make sure that they balance. Each left parenthesis must have a corresponding right parenthesis. An expression like the following, with unbalanced parentheses, infringes on the rules of Python syntax, has no clear meaning, cannot be evaluated, and generates an error message:

$$Z = ((A + B - C) + (D * E - F)) * 5$$

In general, as a matter of good programming style, code describing arithmetic calculations should read easily and naturally and should explain itself to the reader. You should express well-known formulas in familiar ways, use parentheses judiciously, and break up complicated expressions.



5-11. Evaluating an arithmetic expression according to the rules of operator precedence.

Synthetic Tutor

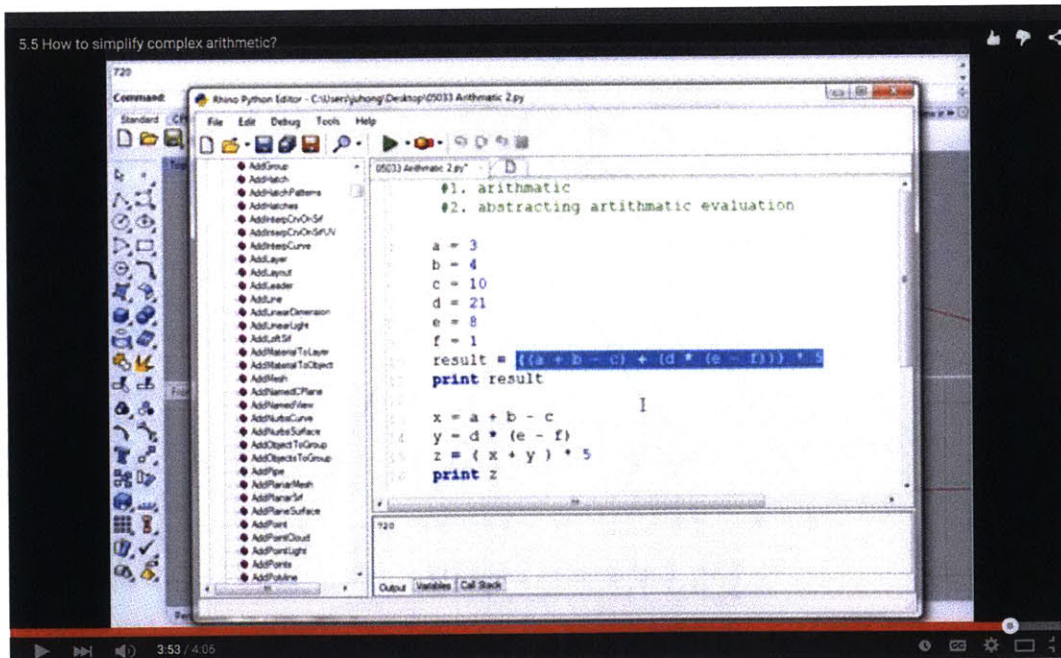
CODE

```
a = 3
b = 4
c = 10
d = 21
e = 8
f = 1
z = ((a + b - c) + (d * (e - f))) * 5
print z

x = a + b - c
y = d * (e - f)
z = (x + y) * 5
print z
```

RESULT

```
307.692307692
355.789230789
```



Module 16: Arithmetic 3

5. THE PYTHON PROGRAMMING LANGUAGE

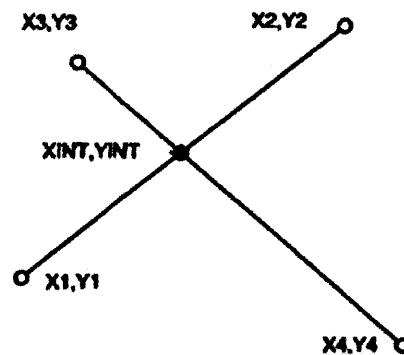
5.5. ARITHMETIC (continued)

So far we have, for clarity, used simple examples to illustrate the use of arithmetic expressions in Python. You do not need a computer to evaluate them. But the point, of course, is that a computer can evaluate even a very complex arithmetic expression extremely rapidly, and that it can repeatedly evaluate an expression for different values of the variables. Here, for example, is a program that does some fairly complicated arithmetic (fig. 5-12);

[sample code on the right side]

To get a feel for the amount of work that the computer must do to execute this program, you should work through the code line by line using a hand calculator. Now imagine executing it thousands of times for different pairs of lines (that is, different initial values for the variables $X_1, Y_1, X_2, Y_2, X_3, Y_3, X_4, Y_4$).

It was a desire to eliminate the drudgery of routine, repetitive arithmetic calculations that motivated the nineteenth-century mathematician Charles Babbage to attempt to construct the first working computer. Draftsmen frequently have to perform routine arithmetic to find the correct coordinates for points and lines in drawings. In the past they used slide rules, now they mostly use pocket calculators. We shall see here how the powerful arithmetic capabilities of Python enable us to automate the draftsman's calculations needed to produce drawings.



5-12. The intersection of two lines.

Synthetic Tutor

CODE

```
x1 = 100
y1 = 200
x2 = 500
y2 = 500
x3 = 200
y3 = 450
x4 = 600
y4 = 100
```

```
b1 = (y2 - y1) / (x2 - x1)
b2 = (y4 - y3) / (x4 - x3)
```

```
a1 = y1 - b1 * x1
a2 = y3 - b2 * x3
```

```
x = (a1 - a2) / (b2 - b1)
y = a1 + b1 * x
```

```
print x
print y
```

RESULT

```
307.692307692
```

```
355.769230769
```

Module 17: Input 1

5.
THE PYTHON
PROGRAMMING LANGUAGE

5.6. INPUT

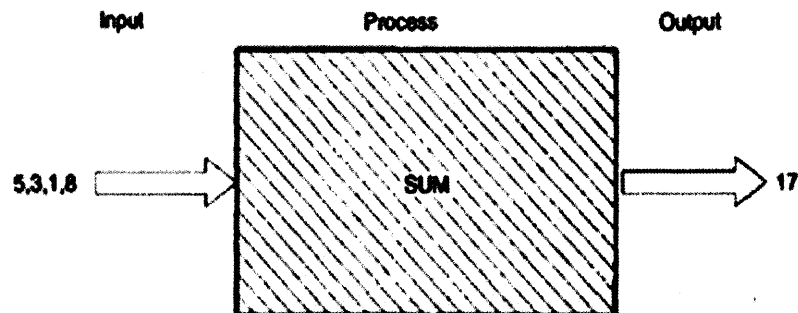
All of the programs that we have considered so far have output but no input. Programs may also be written to process data; these take data as input and operate on it to produce output (fig. 5-13).

A Python input statement to get a value from a user in its simplest form looks like this:

```
import rhinoscriptsyntax as rs
rs.GetInteger()
rs.GetReal()
```

Here is an example of a program that reads in three integers, adds them, and outputs the result:

[Sample codes on the right side]



5-13. A program that processes data—a sequence of numbers is the input, and their sum is the output.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

print 'Type in three integers'

a = rs.GetInteger('Type in the first integer')
b = rs.GetInteger('Type in the first integer')
c = rs.GetInteger('Type in the first integer')

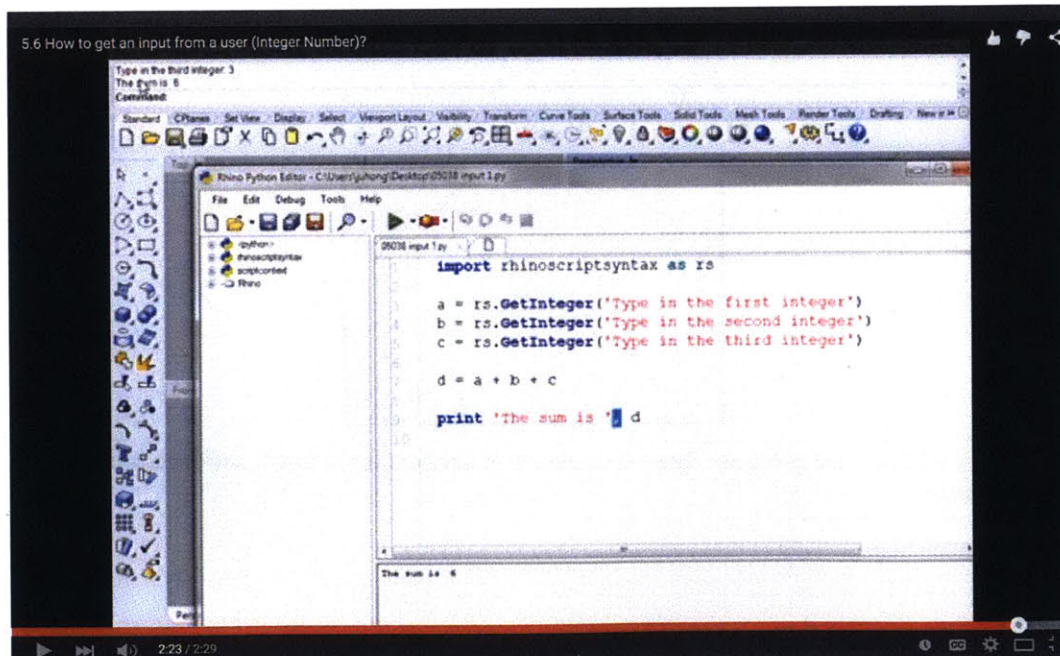
d = a + b + c

print 'The sum is', d
```

RESULT

```
Type in three integers
Type in the first integer: 5
Type in the second integer: 3
Type in the third integer: 1
```

```
The sum is 9
```



Module 18: Input 2

5.
THE PYTHON
PROGRAMMING LANGUAGE

5.6. INPUT (continued)

The GetReal statement may be used for input of real numbers. So we might rewrite our example program as follows:

[sample code on the right side]

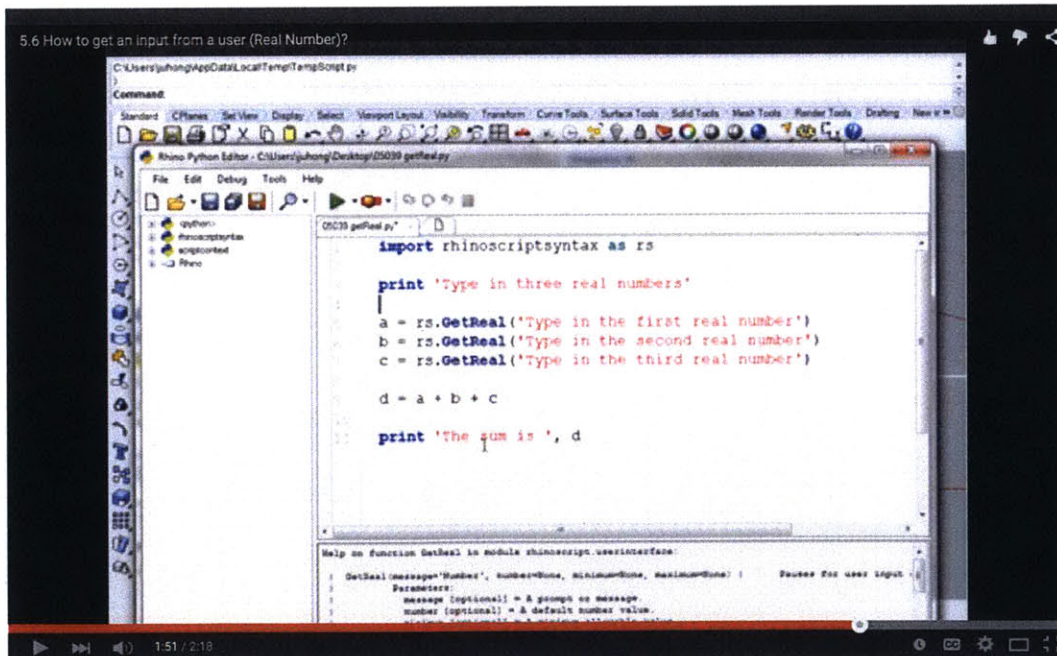
Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
print 'Type in three real numbers'
a = rs.GetReal('Type in the first real')
b = rs.GetReal('Type in the first real')
c = rs.GetReal('Type in the first real')
d = a + b + c
print 'The sum is', d
```

RESULT

```
Type in three real numbers
Type in the first real: 10
Type in the first real: 20
Type in the first real: 30
The sum is 60.0
```



Module 19: input 3

5.
THE PYTHON
PROGRAMMING LANGUAGE

5.6. INPUT (continued)

You can see from these examples how a computer might save a draftsman a great deal of work. Imagine, for instance, that an architectural draftsman is designing a series of elliptical rooms and must know the floor area and perimeter of each. The following program takes as input the lengths of the minor and major axes and prints out the area and perimeter:

[sample code on the right side]

The draftsman can use this program, whenever needed, to perform the tedious, time-consuming, and error-prone process of evaluating the complex formula for each new set of input values.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

pi = 3.1415
maj = rs.GetReal('Type in the length of the major axis')
min = rs.GetReal('Type in the length of the minor axis')

area = pi * min * maj
perimeter = 2 * pi * math.sqrt((min*min + maj*maj) / 2)

print 'The area is ', area
print 'The perimeter is ', perimeter
```

RESULT

```
Type in the length of the major axis of the ellipse: 50
Type in the length of the minor axis of the ellipse: 20
The area is 3141.5
The perimeter is 239.249512121
```

Module 20: Exercise 1

5.
THE PYTHON
PROGRAMMING LANGUAGE

14. EXERCISES

1. Write a program that, when executed, displays your name on the screen.

Please upload your python file: No file chosen

2. Write a program that assigns integer values to the **Length**, **Width**, and **Height** of a rectangular box, calculates Volume and Surface_area, and writes out the values of **all five variables**.

Please upload your python file: No file chosen

3. Modify this program (No.2) to read in the values for **Length**, **Width**, and **Height**.

Please upload your python file: No file chosen

4. The area of a triangle is half its base multiplied by its **height**. Write a program that reads in integer values for these dimensions.

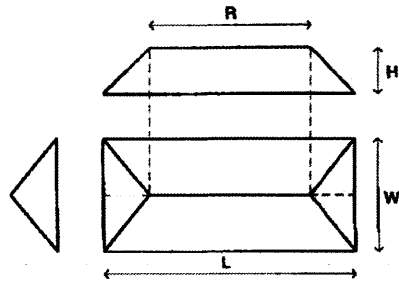
Please upload your python file: No file chosen

5. The formula for the volume of a sphere is $\frac{4}{3} * \pi * R^3$, where R is the radius, and pi is 3.1415. The formula for surface area is $4 * \pi * R^2$. Write a program that reads in a real number value for the radius and writes out values for volume and surface area.
N

Please upload your python file: No file chosen

6. Figure 5-14 shows the plan and two elevations of a hip roof, with associated design variables. Take the cost per square foot of roofing material as an additional variable. Write a program that reads in values for the variables and calculates the cost of a roof.

Synthetic Tutor



5-14. A hip roof.

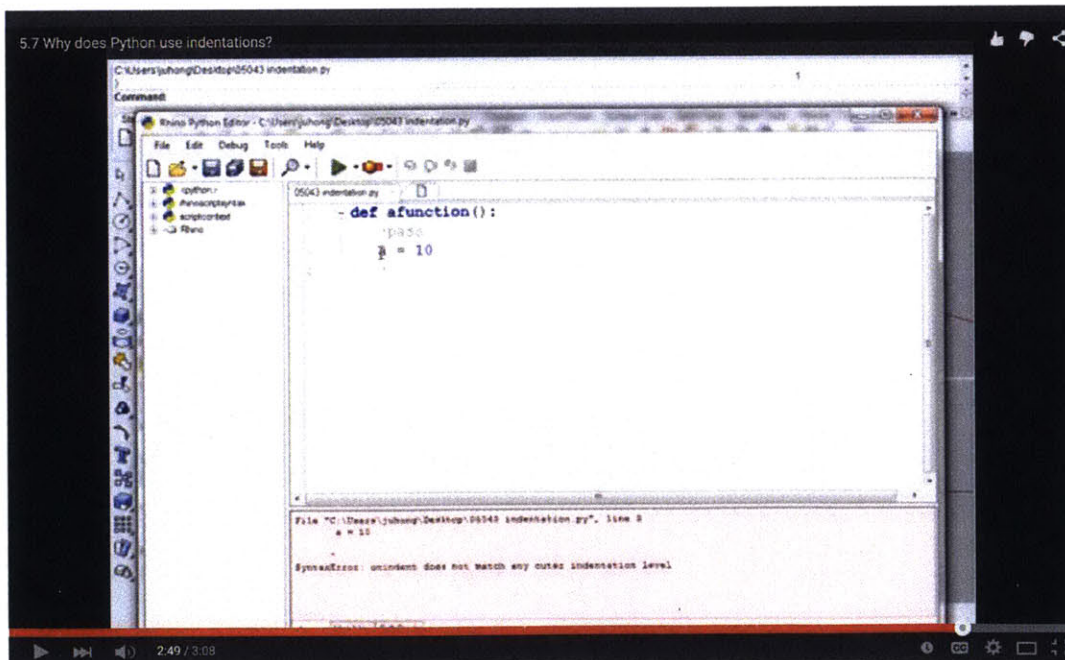
Please upload your python file: No file chosen

Module 21: 5.7 Punctuation

5. THE PYTHON PROGRAMMING LANGUAGE

5.7. PROPER PUNCTUATION

We have now introduced enough of the basic features of Python to enable you to write simple programs. You should try a few to get the feel of programming; the exercises at the end of this chapter offer some suggestions. You must, however, make correct use of the symbols and words that function as delimiters in Python, or errors will result.



Module 22: 5.8 Comments

5.
THE PYTHON
PROGRAMMING LANGUAGE

5.8. EXPLAINING A PROGRAM WITH COMMENTS

You will have noticed in our example programs text after # , like this:

```
# This is a comment
```

Such text is known as a comment. The interpreter or compiler simply ignores everything after # when executing a program. This provides a way to insert explanatory English text into Python code.

Since comments are in English, not in Python, we shall follow the convention of using characters with normal English capitalization and printing them in italics to clearly distinguish them from Python code.

A program should include enough comments to make it self-documenting. In other words, it should be possible to read a program and see immediately what it does and how.

Module 23: 5.9 Styles

5. THE PYTHON PROGRAMMING LANGUAGE

5.9. TYPOGRAPHIC STYLE

Python allows a great deal of freedom in the layout of a program on the page. You should take advantage of this to achieve maximum readability and typographic attractiveness. In particular, make use of blank lines and indentation to clarify the organization of your programs. A standard convention, which we have followed in our examples so far, is to indent, inside a function. As we introduce additional programming constructs, we shall illustrate the associated indentation conventions.

If you work with a general purpose text editor, you must know the indentation conventions and take care to follow them. Some special editors for Python remove this burden by automatically indenting lines.

Module 24: 5.10 Clarity

5. THE PYTHON PROGRAMMING LANGUAGE

5.10. CLARITY AND VERIFIABILITY

Just as there are usually many ways to express the same thought in English, there are usually different but logically equivalent ways to write Python code. Some programmers delight in writing masses of complex, impressive looking statements, the effects of which are almost impossible to decipher. This makes errors difficult to find and correct, causes considerable frustration to a reader trying to figure out what programs do and how they work, and is inconsiderate to those who may later have to work on your programs to modify or correct them. Complex statements should be avoided; a good programmer has a Zen-like reverence for perfect simplicity of expression.

A program must not only work correctly, you must also be able to demonstrate that it does so. To this end, a good general principle to follow is to break down the code into short, easily understandable and verifiable pieces, which reflect the logic of the computation that is to be carried out. Judicious use of functions and procedures-Python constructs that we shall introduce in later chapters-facilitates this.

Be particularly careful about the names you choose for programs, variables, constants, functions, and procedures. Wherever possible, names should be descriptive. (It is infuriating to attempt to read a program in which variable and other names make no sense.) Where there is any possibility of ambiguity in names, write comments to make things clear.

You may have heard from computer enthusiasts that programmers must spend a lot of time debugging-tracking down and eliminating errors in programs. Good programmers do not; they eliminate bugs by programming in a style that minimizes the possibility of errors and that makes finding and correcting any errors that do occur a quick and simple process. In programming, as in other crafts, good style pays off.

Module 25: 5.11 Summary

5.
THE PYTHON
PROGRAMMING LANGUAGE

13. SUMMARY

We have now introduced the basics of Python: how a program is set out; how variables and constants are declared; how numbers may be input; how arithmetic may be performed; and how results may be output. At this point you know enough to write a simple Python program. Try a few before going further to make sure that you understand the fundamentals. In the next chapter, we shall begin to consider programs that generate drawings. Good luck!

Module 26: 6.1 Line

6. PROGRAMS TO GENERATE SIMPLE LINE DRAWINGS

Now that you know the notational conventions of Python and the way to organize a Python program, you can write some elementary programs to generate drawings. We will begin with very simple, two-dimensional line drawings constructed from vectors in a screen coordinate system.

6.1 PRIMITIVE GRAPHIC COMMANDS

We shall be using a command called `rs.AddLine()` to create drawings by using two point lists. To draw a line, first set two points (this type is called, list, which has multiple variables or values in one type. We will frequently use this type to handle multiple point coordinates at once), we write a statement of the form

```
[100, 100, 0]  
[500, 500, 0]
```

To draw a line from the two point coordinates, we write a statement of the form

```
import rhinoscriptsyntax as rs  
rs.AddLine([100, 100, 0],[500,500,0] )
```

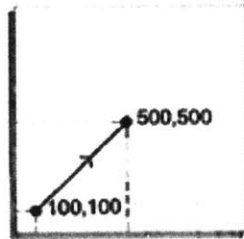
Thus the sequence of statements to draw the square shown in figure 6-10 is written:

```
[sample code : draw a rectangle]
```

The sequence of statements to draw the cross shown in figure 6-11 is written:

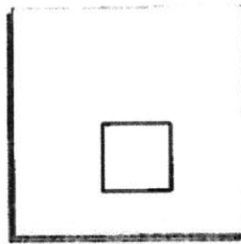
```
[sample code : draw a cross]
```

Adding a line with two points, then, is a primitive marking operation, much as a hand movement with the pencil raised, and a hand movement with the pencil lowered to the paper surface, is the primitive operations in making a pencil drawing. This allows us to construct a drawing directly out of graphic primitives; that is, vector by vector. If the output device is a storage tube, the drawing instrument controlled by two points is, in fact, the electron beam; vectors are actually traced out by the electron beam in the sequence specified by the code. With a refreshed vector display, the effect is to store data describing vectors in the display memory from which the screen image is generated and refreshed. If output is on a raster display, the effect is to write pixel values into the frame buffer, so that vectors mapped onto the raster grid are displayed. In any case, however, it is convenient to think of move and draw as commands for moving the point of a drawing instrument across the surface of the screen.

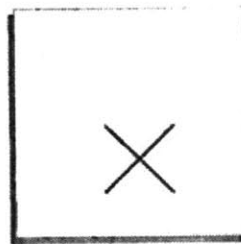


MOVE(100,100);
DRAW(500,500);

6-9. The use of *move* and *draw* commands to draw a vector between two points.



6-10. A square produced by *move* and *draw* operations.



6-11. A cross produced by *move* and *draw* operations.

Synthetic Tutor

CODE

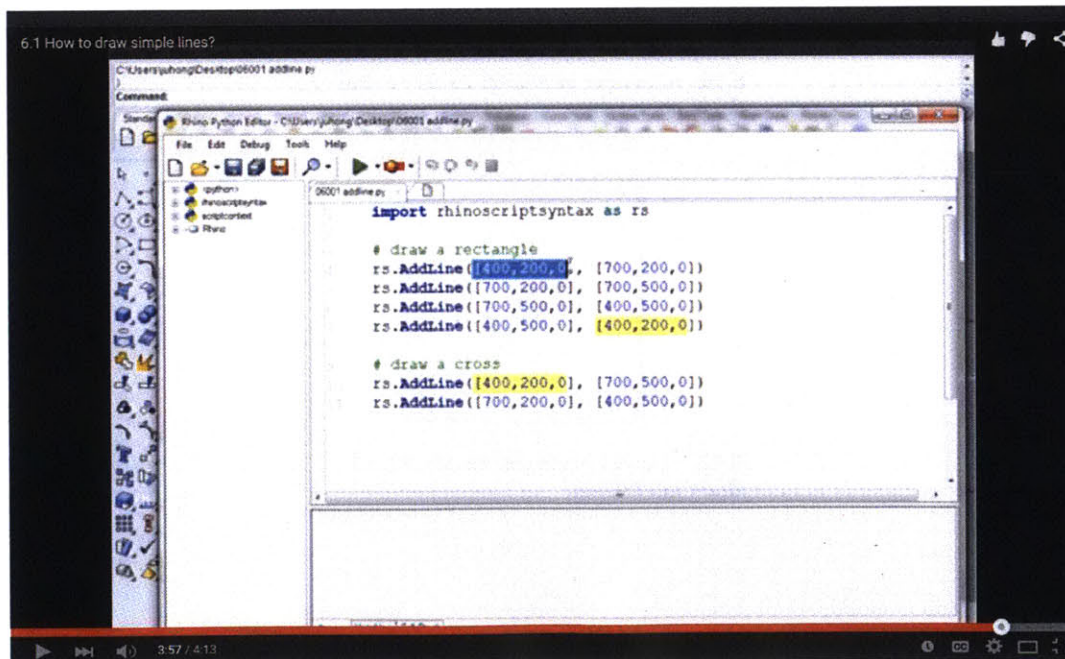
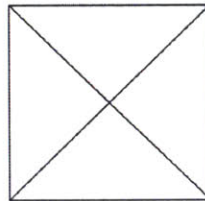
```
import rhinoscriptsyntax as rs

# draw a line
rs.AddLine([100,100,0], [500,500,0])

# draw a rectangle
rs.AddLine([400,200,0], [700,200,0])
rs.AddLine([700,200,0], [700,500,0])
rs.AddLine([700,500,0], [400,500,0])
rs.AddLine([400,500,0], [400,200,0])

# draw a cross
rs.AddLine([400,200,0], [700,500,0])
rs.AddLine([700,200,0], [400,500,0])
```

RESULT



Module 27: 6.2 Library

6. PROGRAMS TO GENERATE SIMPLE LINE DRAWINGS

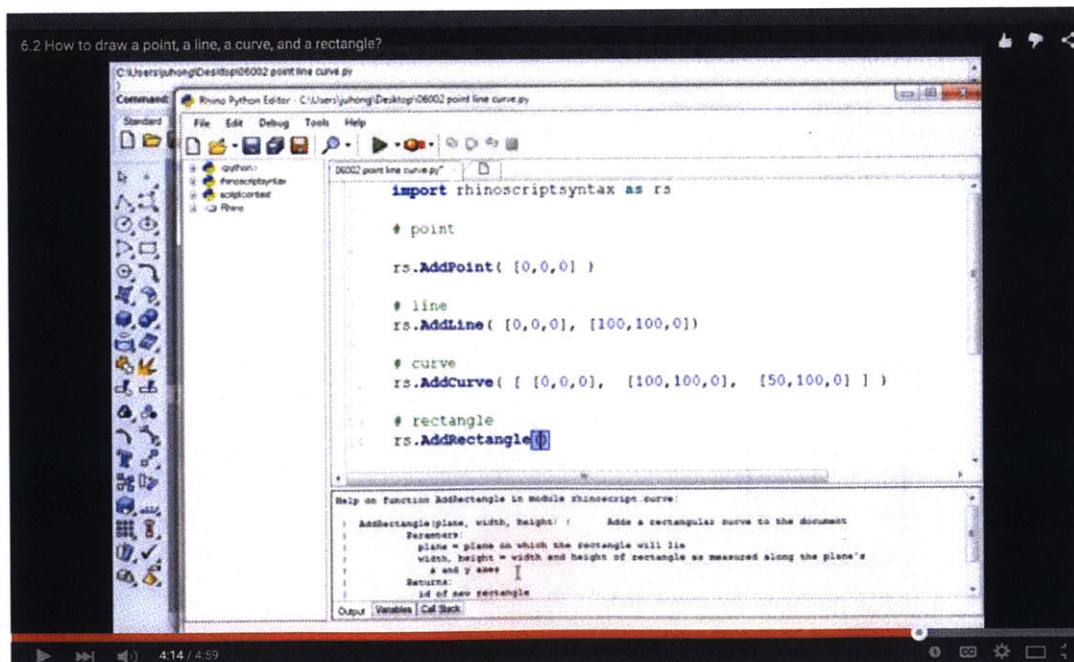
6.2 ANNOUNCEMENT OF EXTERNAL GRAPHICS PROCEDURES

These basic graphic commands - AddPoint, AddLine, AddCurve, AddRectangle - are not part of the Python language itself. Technically they are known as external procedures, and any program that uses them must contain the following announcement at the beginning of a code:

```
import rhinoscriptsyntax as rs
```

These external procedures must be implemented on your computer before you can use them in Python programs. This is done in different ways for different Python systems. rs - is an abbreviation of rhinoscriptsyntax. Graphic commands can be used with . (dot notation). Here show some examples:

```
import rhinoscriptsyntax as rs
rs.AddPoint( [0,0,0] )
rs.AddLine( [0,0,0], [100,100,0] )
rs.AddCurve( [ [0,0,0], [100,100,0], [50,100,0] ] )
rs.AddRectangle( [0,0,0], 5, 20 )
```



Module 28: 6.3 Variables

6. PROGRAMS TO GENERATE SIMPLE LINE DRAWINGS

6.3 A COMPLETE GRAPHICS PROGRAM

We are now in a position to write a very simple but complete graphics program. The following is our example. It announces the graphics tools that will be used (the external procedures), makes the necessary preparations to draw a picture, uses `rs.AddLine()` to produce our example square (see fig. 6-10).

[sample code: draw a rectangle]

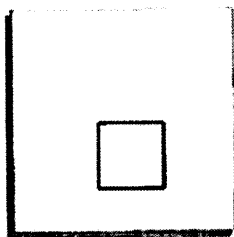
6.3.1 USING VARIABLES

Notice the use of variables in this version of the program. Write variables X1, X2, Y1, and Y2, assign values to these variables. Repeatedly, write variables to make point lists: pt1, pt2, pt3, and pt4, and assign point coordinates within a bracket (making it a list) to the variables.

```
x1 = 400  
x2 = 700  
y1 = 200  
y2 = 500
```

```
pt1 = [x1, y1, 0]  
pt2 = [x2, y1, 0]  
pt3 = [x2, y2, 0]  
pt4 = [x1, y2, 0]  
pt5 = [x1, y1, 0]
```

```
rs.AddLine( pt1, pt2)  
rs.AddLine( pt2, pt3)  
rs.AddLine( pt3, pt4)  
rs.AddLine( pt4, pt5)
```



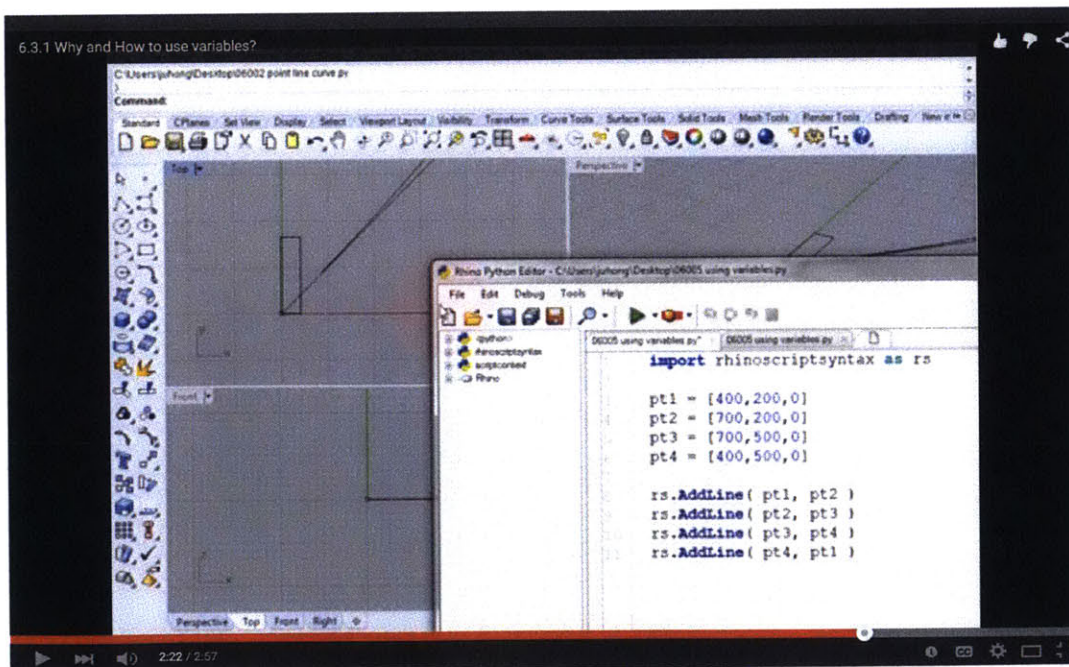
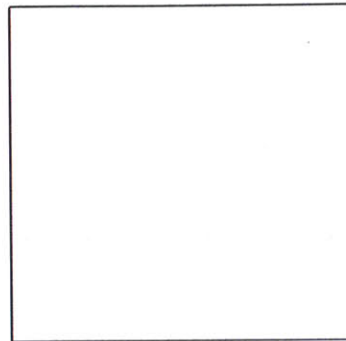
6-10. A square produced by move and draw operations.

CODE

```
import rhinoscriptsyntax as rs

# draw a rectangle
pt1 = [400,200,0]
pt2 = [700,200,0]
pt3 = [700,500,0]
pt4 = [400,500,0]
pt5 = [400,200,0]
rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt4)
rs.AddLine(pt4, pt5)
```

RESULT



Module 29: 6.4 GetReal

6. PROGRAMS TO GENERATE SIMPLE LINE DRAWINGS

6.3 A COMPLETE GRAPHICS PROGRAM

6.3.2 READING IN COORDINATE VALUES

Instead of assigning values to our variables X1, X2, Y1, and Y2 by statements in the code of the program, we might choose to read in these values.

The significant difference here is that values are given to the coordinate variables at execution time, rather than by being specified in the code of the program. The dialogue by which coordinate values were entered appears in the text window, and the corresponding square appears in the drawing window. By re-executing the program and entering different values for the four variables, the user can generate drawings of many different squares and rectangles.

[sample code: draw a rectangle]

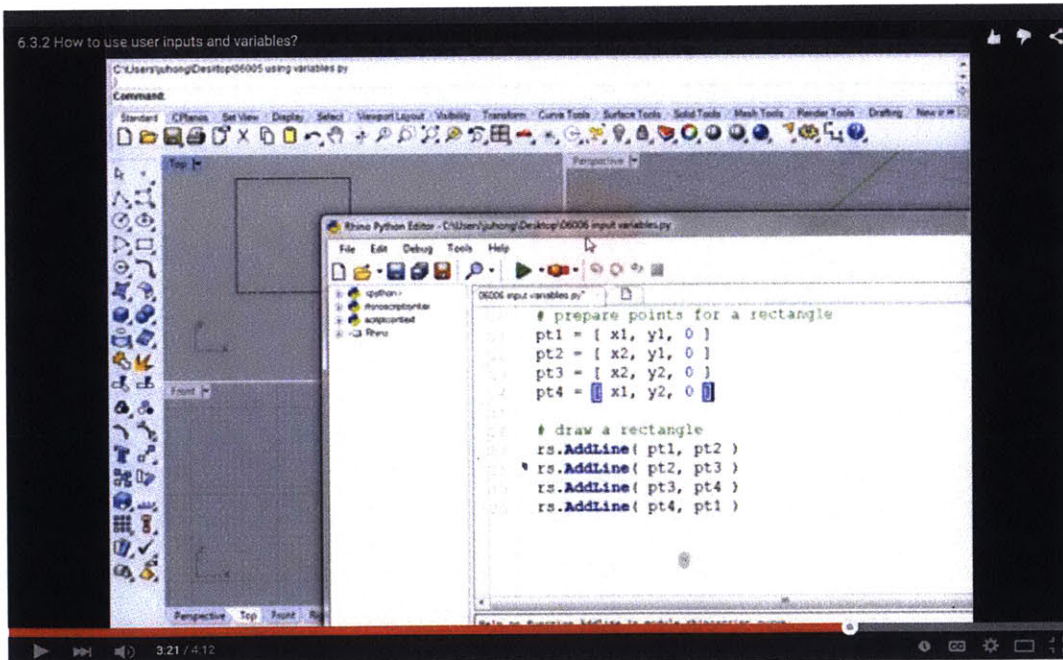
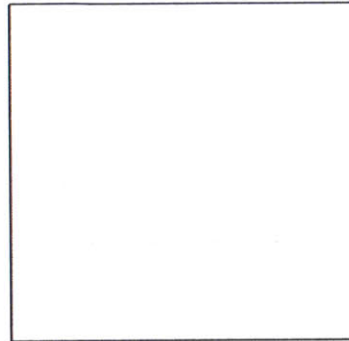
CODE

```
import rhinoscriptsyntax as rs

# get user inputs
x1 = rs.GetReal('Enter real for x1')
y1 = rs.GetReal('Enter real for y1')
x2 = rs.GetReal('Enter real for x2')
y2 = rs.GetReal('Enter real for y2')

# draw a rectangle
pt1 = [x1, y1, 0]
pt2 = [x2, y1, 0]
pt3 = [x2, y2, 0]
pt4 = [x1, y2, 0]
pt5 = [x1, y1, 0]
rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt4)
rs.AddLine(pt4, pt5)
```

RESULT



Module 30: 6.5 Parts

6. PROGRAMS TO GENERATE SIMPLE LINE DRAWINGS

6.4 THE THREE ESSENTIAL PARTS OF A GRAPHIC PROGRAM

It should now be clear that there are two essential parts of a graphic program. First, we must declare the variables, so that space is set aside in memory to store the coordinate data needed to generate our drawing. We must give values to all of these variables, which can either be done with assignment statements or by reading in values. Second, we use these values in `rs.AddLine()` commands to generate the image. For clarity and simplicity, we shall mostly give values to variables by means of assignments in our examples. But keep in mind that you can always replace the assignment statements (=) by reading statements (`rs.GetInteger()` or `rs.GetReal()`) if you want to experiment with the effects of different values. You will find that it is more interesting in your own programming projects to allow for user input in this way. If your computer is equipped with a mouse or graphic tablet, you can probably use these devices for input of coordinate values (`rs.GetPoint()`), and you may want to explore this alternative to typing in numbers.

Module 31: 6.5 Summary

6. PROGRAMS TO GENERATE SIMPLE LINE DRAWINGS

6.5 SUMMARY

You now know how to write the simplest kind of program to generate a line drawing. Before going on, you should do some of the suggested exercises. You will find that it is very tedious to program this way, and you may wonder whether it is worth it. If you know all the coordinates of a figure, why not just draw it yourself, instead of writing down a lot of functions?

The problem, here, is that there is a great deal of effort with little reward. It does not make very much practical sense to program this way; you have to write a relatively long program to generate a relatively simple figure. But, as you learn more sophisticated programming techniques in succeeding chapters, you will be able to write concise, elegant programs to generate large and complex drawings. When you can do this, you can begin to exploit the potential power of computer graphics.

Module 32: 7.1 Variables

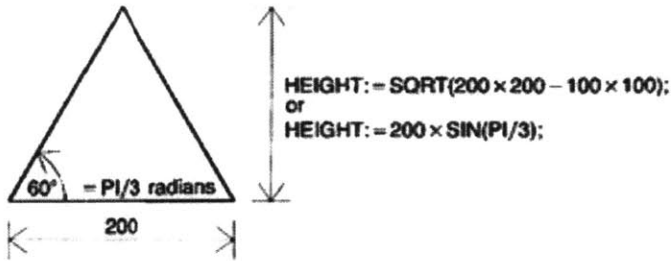
7. COORDINATE CALCULATIONS

So far we have considered graphic programs in which coordinate values are either assigned directly or read in. As you will have found when doing the exercises, you must calculate coordinates to enter, or to type in when prompted, in order to produce a picture. Perhaps you used a pocket calculator to do this. You probably needed something a bit more sophisticated than a four-function calculator, since calculation of coordinates often requires evaluation of trigonometric functions such as sine and cosine, and perhaps others such as square root. Consider, for example, the equilateral triangle of side length 200 shown in figure 7-1. In order to generate it using `rs.AddLine()`, we must know its height to establish the Y coordinate of the apex. From the Pythagorean theory we know that the height will be the square root of 200 squared minus 100 squared, which comes to 173.2050. Alternatively, we can use a trigonometric function and calculate the height as 200 multiplied by the sine of 60 degrees.

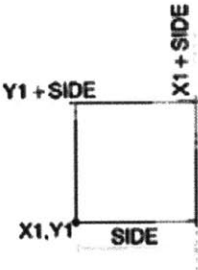
7.1 INDEPENDENT AND DEPENDENT VARIABLES

An alternative to performing such calculations outside the program is to write code to perform them within the program. If we want to draw a square parallel to the coordinate axes, for example, we can give values to `X1`, `Y1`, and `Side`, then calculate all the other vertex coordinates from these (fig. 7-2). That is, we first assign or read in values for our independent variables, then calculate values for dependent variables and finally use the values in move and draw commands. Here is an example of a simple program that does this:

[sample code on the right side]



7-1. An equilateral triangle.



7-2. The vertex coordinates of a square as dependent variables.

Synthetic Tutor

CODE

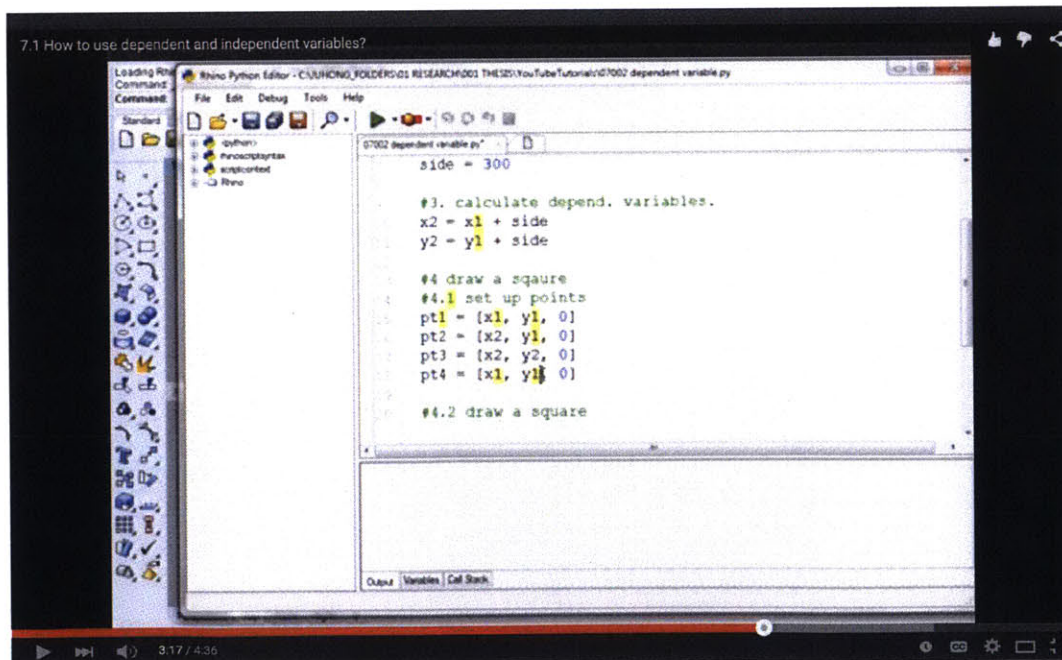
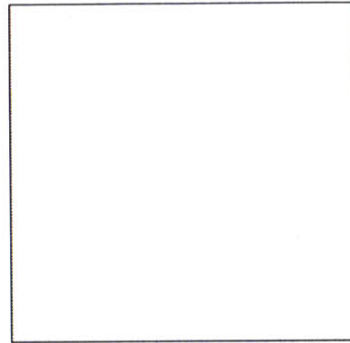
```
import rhinoscriptsyntax as rs

# independent variables
x1 = 400
y1 = 200
side = 300

#dependent variables
x2 = x1 + side
y2 = y1 + side

# draw square
pt1 = [x1, y1, 0]
pt2 = [x2, y1, 0]
pt3 = [x2, y2, 0]
pt4 = [x1, y2, 0]
pt5 = [x1, y1, 0]
rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt4)
rs.AddLine(pt4, pt5)
```

RESULT



Module 33: 7.2 Math 1

7. COORDINATE CALCULATIONS

7.2 STANDARD ARITHMETIC FUNCTIONS

In this example we used the Python's arithmetic operators to obtain the desired result-much as we might use a four-function calculator. In addition to these, for use in more complex calculations, Python provides certain standard arithmetic functions analogous to those of a scientific calculator. To use additional arithmetic functions, we need to import - math - library in your code. To get the square root of 317.835 you write the statement:

```
import math
Height = math.sqrt(317.835)
```

The result of taking the square root of 317.835 is thus assigned to Height.

Different implementations of Python may provide more or less extensive sets of standard arithmetic functions, but these are the basic ones that you will need and that should be available in any implementation:

`abs(X)`: Computes the absolute value of X. The type of the result is the same as that of X.

`math.sqrt(X)`: Computes the square root of X. The type of the result is always real.

`math.sin(X)`: Computes the sine of X, where X is specified in radians. The type of the result is always real.

`math.cos(X)`: Computes the cosine of X, where X is specified in radians. The type of the result is always real.

`math.atan(X)`: Computes the arctangent of X. The result is in radians, and its type is always real.

[sample code on the right side]

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

result = abs(-317.485)
print result

result = math.sqrt(317.835)
print result

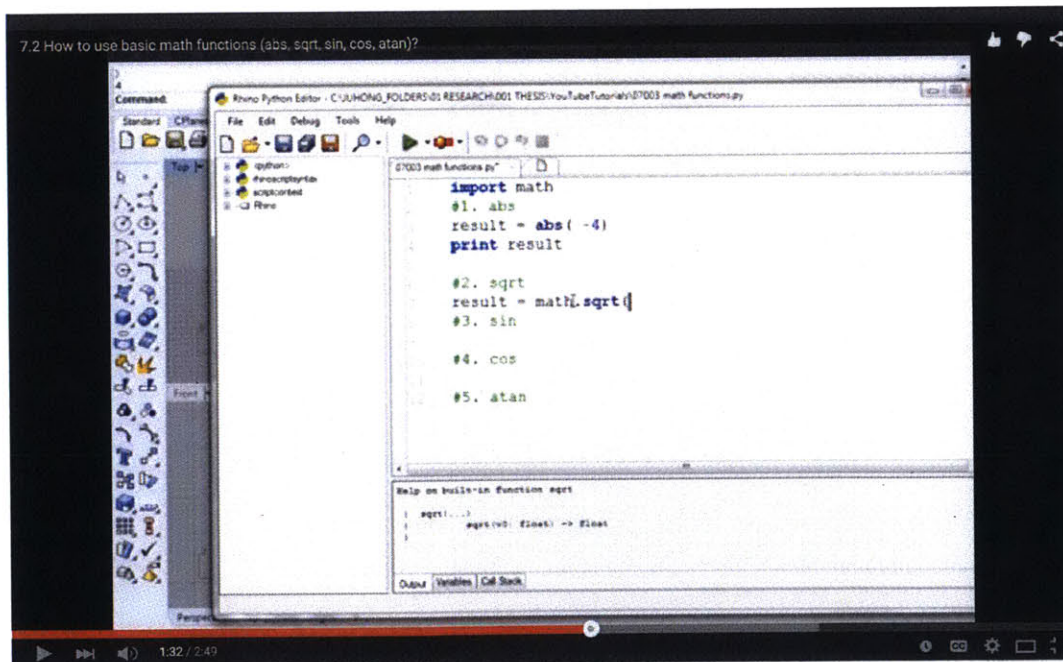
result = math.sin(317.835)
print result

result = math.cos(317.835)
print result

result = math.atan(317.835)
print result
```

RESULT

```
317.485
17.8279275296
-0.509102730554
-0.860705762583
1.56765005058
```



Module 34: 7.3 Rounding

7. COORDINATE CALCULATIONS

7.3 ROUNDING, CEILING AND FLOORING

You will notice that, in many cases, these functions yield real results. Yet you may need an integer to substitute in functions. A real number can be converted to an integer either by truncating it- throwing away the decimal part-or rounding it to the nearest whole number. To perform these operations, Python and Mathematics Library provide transfer functions:

`round(X)`: X must be a real number. The result, of type integer, is the value of X rounded.

`math.ceil(X)`: X must be a real number. The result, of type integer, is the ceiling of X.

`math.floor(X)`: X must be a real number. The result, of type integer, is the floor of X.

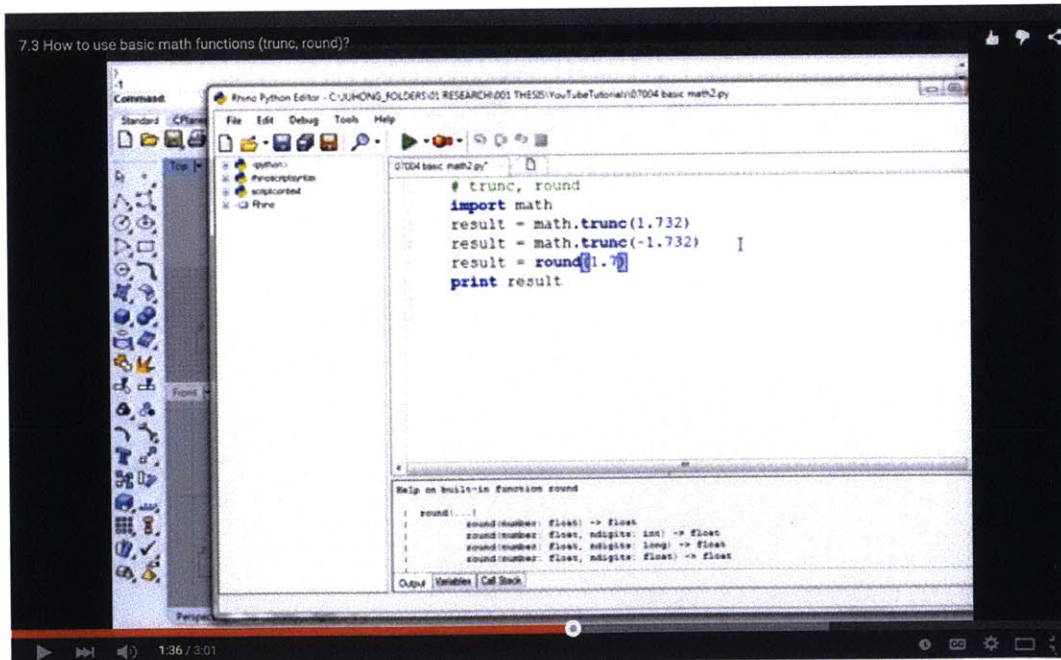
Here are some examples of rounding and truncating and their results:

```
Expression
2 = round(1.732)
2 = math.ceil(1.732)
1 = math.floor(1.732)
```

[sample code on the right side]

Synthetic Tutor

CODE	RESULT
<code>import rhinoscriptsyntax as rs</code>	1
<code>import math</code>	
<code>result = math.trunc(1.732)</code>	-1
<code>print result</code>	2.0
<code>result = math.trunc(-1.732)</code>	-2.0
<code>print result</code>	1.0
<code>result = round(1.732)</code>	0
<code>print result</code>	0.0
<code>result = round(-1.732)</code>	
<code>print result</code>	
<code>result = round(1.01)</code>	
<code>print result</code>	
<code>result = math.trunc(0.0)</code>	
<code>print result</code>	
<code>result = round(0.0)</code>	
<code>print result</code>	



Module 35: 7.3 Math 27.
COORDINATE CALCULATIONS

7.3 ROUNDING, CEILING AND FLOORING (continued)

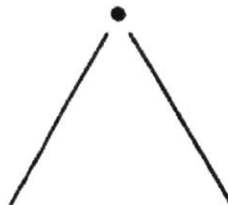
Here is an example of a simple program that uses standard arithmetic functions to compute coordinate values.

[sample code on the right side]

Note that in the above program, the Height of the equilateral triangle calculated to two decimal places is 346.41.



a. Real coordinates rounded to a raster grid.



b. Real coordinates truncated to a raster grid.

7-3. Enlargements of the apex of a triangle.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

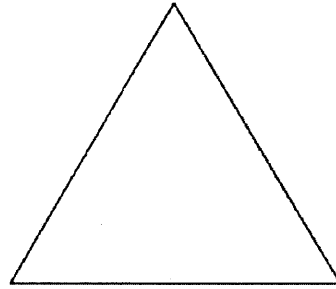
# independent variables
radians = 0.01745
x1 = 300
y1 = 300
side = 400

# dependent variables
x2 = x1 + side
x3 = (x1 + x2) / 2
height = side * math.sin(60 * radians)
y2 = y1 + height

pt1 = [x1, y1, 0]
pt2 = [x2, y1, 0]
pt3 = [x3, y2, 0]
pt4 = [x1, y1, 0]

rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt4)
```

RESULT



Module 36: 7.4 Function 1

7. COORDINATE CALCULATIONS

7.4 WRITING YOUR OWN FUNCTIONS

If Python does not have the function that you need for a calculation, you can write it yourself. The following, for example, is a function that computes the cube of X:

```
def cube ( x ):
    result = x * x * x
    return result
```

The first line is analogous to a program heading. It begins with the reserved word - def -, followed by the identifier (name) of the function-in this case cube. The variable X is referred to as the formal (input) parameter of this function. The formal (input) parameter is named and typed within the parentheses as shown. Then, following a colon (:).

The indented code specifies how the value of the function is to be calculated for a given value of the formal parameter. It must contain at least one assignment statement, assigning a value to the function identifier. The indented statements that follow are called a block. A four space tab indent width is the preferred coding style. If the block statements do not indented properly, Python will cause an error.

A function must be declared before it is invoked in the program-just as a variable must be declared before it can be used in an executable statement. Here is an example of a simple program that calculates the volume of a cube of side length 3:

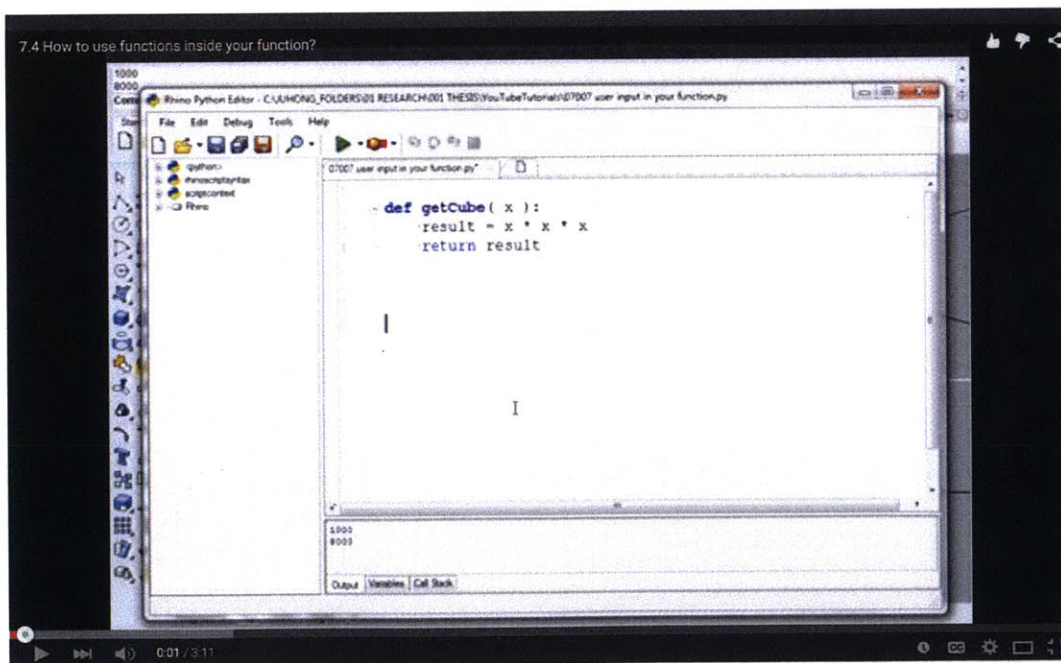
[sample code on the right side]

Notice how the function is invoked-in exactly the same way as Python's built-in functions such as abs() and print(). When it is invoked, it calculates the cube of 3. The result is substituted for the expression getCube(3) and assigned to volume.

The value 3 in this program is referred to as the actual (input) parameter of the function getCube(). When getCube() is invoked, this value is assigned to the formal parameter x. This is the way that the function is given an input value on which to operate.

Synthetic Tutor

CODE	RESULT
<pre>import rhinoscriptsyntax as rs</pre>	1000
<pre>def getCube(x): result = x * x * x return result</pre>	27
<pre>volume = getCube(10) print volume</pre>	
<pre>def getVolume(): volume = getCube(3) print volume</pre>	
<pre>getVolume()</pre>	



Module 37: 7.4 Function 2

7. COORDINATE CALCULATIONS

7.4 WRITING YOUR OWN FUNCTIONS (continued)

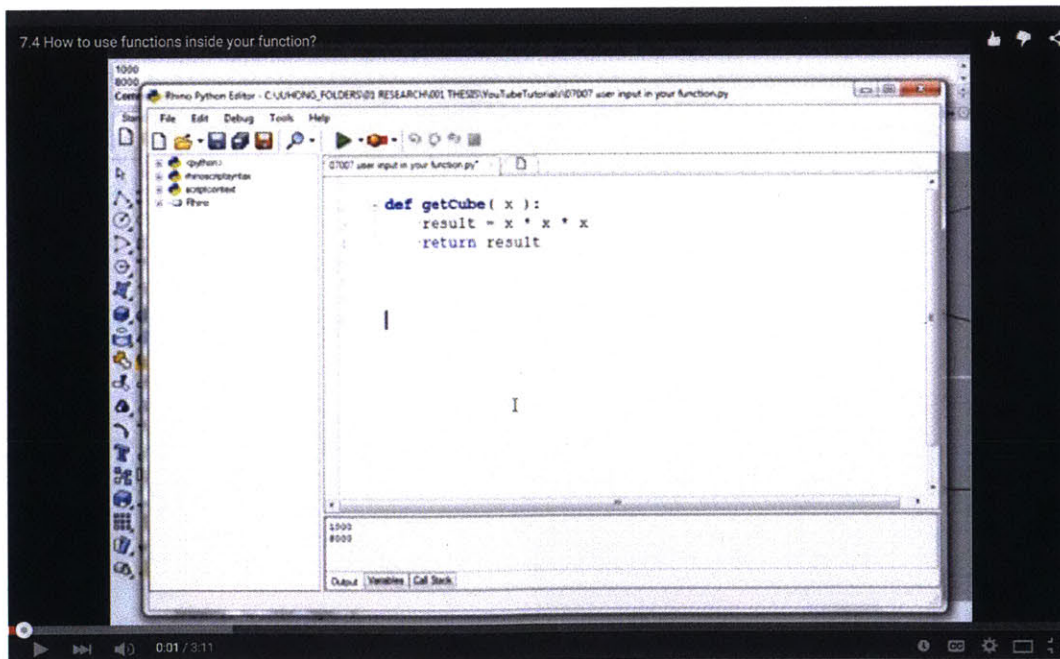
The actual parameter need not be a value; it might be a variable to which a value has been assigned. This is illustrated in the following program, which reads in the side length of a cube then calculates the volume:

[sample code on the right side]

In essence, declaring a function in this way allows us to use a short, meaningful name in our program in place of a longer, more complex, and less meaningful expression. Once the function is declared, we can invoke it repeatedly in a program, without having to write it out in full each time, or to worry about the details of its internal operation. The construct of a function, then, provides us with a very powerful abstraction mechanism for use in programming.

Synthetic Tutor

CODE	RESULT
<pre>import rhinoscriptsyntax as rs def getCube(x): result = x * x * x return result def getVolumeInput(): side = rs.GetInteger('Enter integer for side length') volume = getCube(side) print volume getVolumeInput()</pre>	1000000



Module 38: 7.5 Parameter

7.

COORDINATE CALCULATIONS

7.5 FUNCTIONS WITH SEVERAL PARAMETERS

In computer graphics programming, we are often interested in functions that return an X coordinate or a Y coordinate of a point in a drawing. Consider, for example, the bisection of a line defined by its endpoints (X1, Y1) and (X2, Y2) (fig. 7-4). The following function, which computes the midpoint between any two integers Low and High, might be used to find the midpoint coordinates:

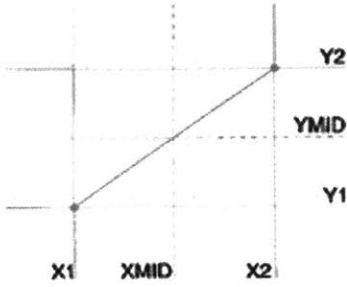
The code to calculate the midpoint (Xmid, Ymid) using this function is as follows:

```
Xmid = getMidPoint(X1,X2);  
Ymid = getMidPoint(Y1,Y2);
```

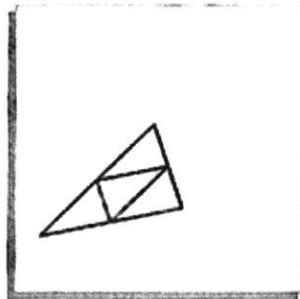
Notice that Midpoint has two formal (input) parameters, Low and High, and that each invocation of Midpoint lists two actual parameters. A function may, in fact, have as many formal (input) parameters as we wish. Whenever the function is invoked, there must be an actual parameter for each formal parameter. The correspondence is established by position in the parameter list; the value of the first actual parameter is assigned to the first formal parameter, that of the second actual parameter to the second formal parameter, and so on.

The following program illustrates use of this Midpoint function to draw a smaller triangle within a larger triangle (fig. 7-5):

[sample code on the right side]



7-4. The bisection of a line defined by its endpoints.



7-5. A drawing produced by Nest_triangles.

CODE

```
import rhinoscriptsyntax as rs

def getMidPoint(low, high):
    midpoint = (low + high) / 2
    return midpoint

def drawNestTriangle():
    # set coordinates of outer triangle
    x1 = 100
    y1 = 200
    x2 = 500
    y2 = 600
    x3 = 600
    y3 = 300

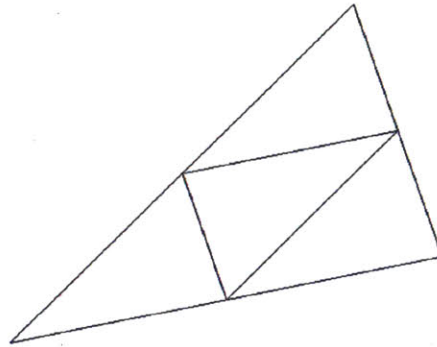
    # calculate midpoints of sides
    x12 = getMidPoint(x1, x2)
    y12 = getMidPoint(y1, y2)
    x23 = getMidPoint(x2, x3)
    y23 = getMidPoint(y2, y3)
    x31 = getMidPoint(x3, x1)
    y31 = getMidPoint(y3, y1)

    # set points
    pt1 = [x1, y1, 0]
    pt2 = [x2, y2, 0]
    pt3 = [x3, y3, 0]
    pt12 = [x12, y12, 0]
    pt23 = [x23, y23, 0]
    pt31 = [x31, y31, 0]

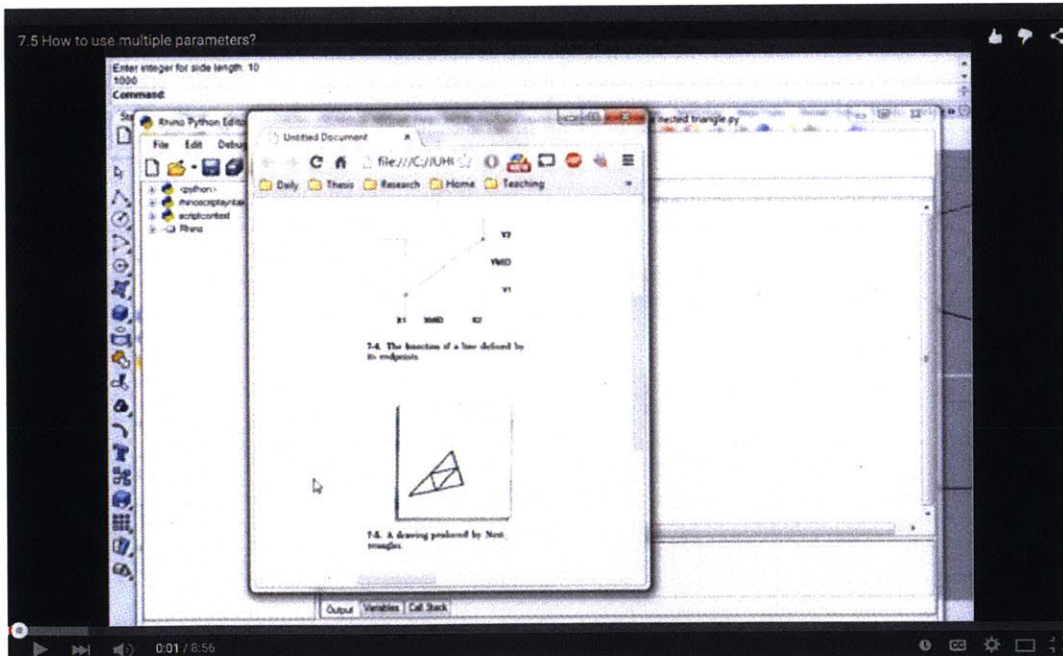
    # draw outer triangle
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt1)

    # draw inner triangle connecting midpoints
    rs.AddLine(pt12, pt23)
    rs.AddLine(pt23, pt31)
    rs.AddLine(pt31, pt12)
```

RESULT



drawNestTriangle()



Module 39: 7.6 Local Variable 1

7. COORDINATE CALCULATIONS

7.6 LOCAL VARIABLES

Now consider the simple coordinate calculation problem illustrated in figure 7-6. We have a point (X1, Y1), an angle Theta, and a Length; we want to calculate the coordinates (X2, Y2). We know from elementary trigonometry that the Base of the triangle is given by the formula $\text{Length} * \cos(\text{Theta})$, so a function to return X2 can be written as follows:

[sample code on the right side]

Note that we are invoking functions from within a function; the Python standard function `cos` is invoked from within X2. In this way, you can also invoke functions that you have declared yourself. Also note that since the angle Theta is expressed as a real number of radians, the function X2 has both integer and real formal parameters in its parameter list. The `cos` function returns a real value, and since the result of multiplying an integer by a real number is real, the value of the expression $\text{Length} * \cos(\text{Theta})$ is real.

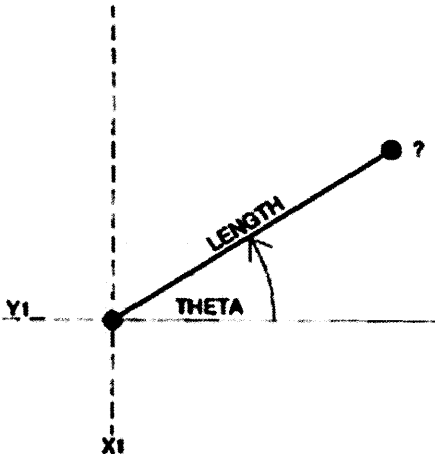
Finally, note that we have declared a real variable `Base` within the function. Whenever a variable is declared within a function, it is local to that function. In effect, it does not exist outside the function; it cannot be referenced from outside the function; its value is unknown outside the function; and in fact, it exists only during the time that the function is activated. After the function returns its result value and terminates, the program has no way of knowing that the local variable ever existed. The functions formal parameters are all local in this same sense.

If we wanted to eliminate the local variable `Base`, we could easily do so by rewriting the function like this:

[sample code on the right side]

Here, instead of using the variable `Base`, we have used the expression $\text{Length} * \cos(\text{Theta})$. In general, we can use expressions as actual parameters when we invoke procedures. This is more concise, but it usually makes programs more difficult to follow. In most cases it is better to introduce local variables, break complicated expressions down into several separate lines, and provide explanatory comments at each step, as we did in our first version of this function.

You can now see that strict rules govern the use of functions. The local nature of the formal parameters, and of variables declared within the function, ensures that nothing done to these variables during the execution of the function can have any effect on anything outside the function. That is, the internal workings of a function are completely insulated from the rest of the program. This discipline may seem restrictive, but you will find that it clarifies your programs and keeps you out of trouble.



7-6. A coordinate calculation problem to find $X2$, given $X1$, $Y1$, $Theta$, and $Length$.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

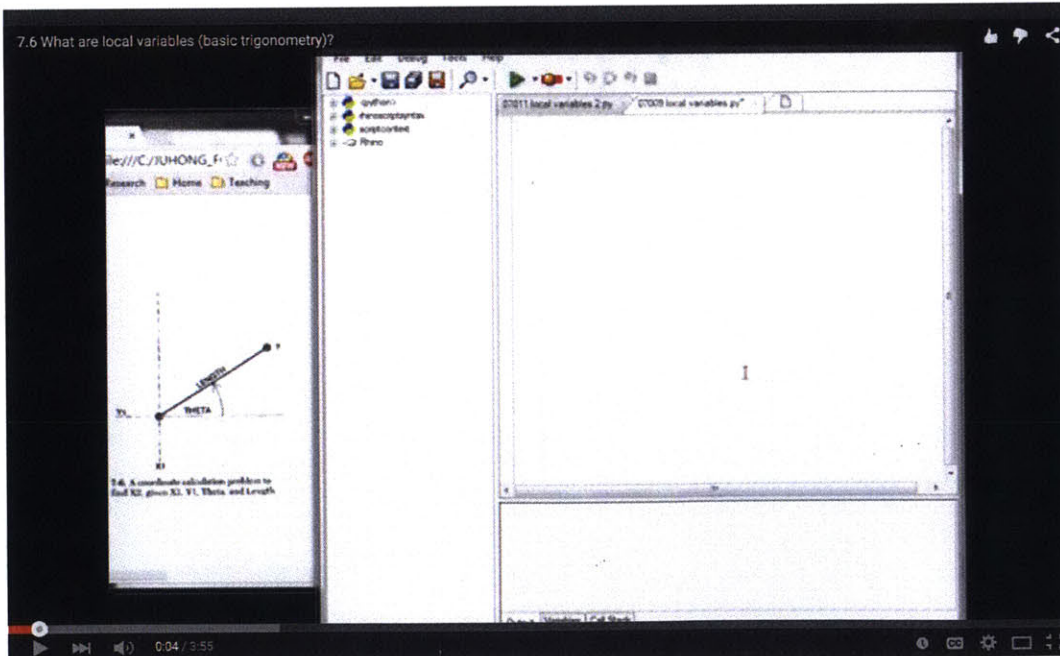
def X2_ver1():
    x1 = 0
    length = 100
    theta = 30

    base = length * math.cos(theta)
    # round and add to x1
    x2 = x1 + base
    return x2

def X2_ver2():
    x1 = 0
    length = 100
    theta = 30

    # round and add to x1
    x2 = x1 + length * math.cos(theta)
    return x2
```

RESULT



Module 40: Exercise

6. PROGRAMS TO GENERATE SIMPLE LINE DRAWINGS

6.6 EXERCISES

1. Draw two figures defined by the following two sets of coordinates.

```
[ 200, 200, 0 ]
[ 600, 200, 0 ]
[ 300, 300, 0 ]
[ 500, 300, 0 ]
[ 200, 200, 0 ]
```

```
[ 400, 250, 0 ]
[ 200, 200, 0 ]
[ 600, 200, 0 ]
[ 400, 250, 0 ]
[ 300, 300, 0 ]
[ 500, 300, 0 ]
[ 400, 250, 0 ]
```

Please upload your python file: No file chosen

2. Write a program to draw an equilateral triangle.

Please upload your python file: No file chosen

3. Write a program to draw your initials.

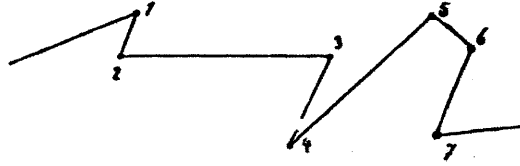
Please upload your python file: No file chosen

4. Write a program to read in the coordinates of the **vertices of a triangle**, then draw the triangle.

Please upload your python file: No file chosen

5. Below figure shows Paul Klees active line, limited in its movement by fixed points. Write a program to draw a line of this type.

Synthetic Tutor



Lines as introduced by Paul Klee in his *Pedagogical Sketchbook*.

Please upload your python file: No file chosen

Module 41: 7.6 Local Variable 2

7. COORDINATE CALCULATIONS

7.6 LOCAL VARIABLES (continued)

To conclude, here is a program that uses functions X2 and Y2 to calculate coordinates, then draws a line of length Length at angle Theta from point

[sample code on the right side]

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def X2(x1, length, theta):
    # calculate real value for base
    base = length * math.cos(theta)

    # round and add to x1
    x2 = x1 + base
    return x2

def Y2(y1, length, theta):
    # calculate real value for base
    height = length * math.sin(theta)

    # round and add to y1
    y2 = y1 + height
    return y2

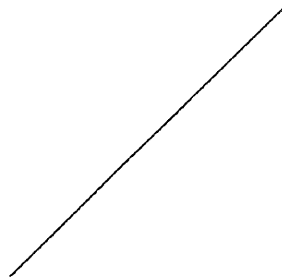
def drawLine():
    # set variables defining line
    radians = 0.01745
    angle = 45
    x1 = 300
    y1 = 200
    length = 400

    # calculate dependent variables
    theta = angle * radians
    x = X2(x1, length, theta)
    y = Y2(y1, length, theta)

    # draw line
    pt1 = [x1, y1, 0]
    pt2 = [x, y, 0]
    rs.AddLine(pt1, pt2)

drawLine()
```

RESULT



Module 42: 7.7 Trigonometry

7. COORDINATE CALCULATIONS

7.7 FUNCTIONS FOR TRIGONOMETRY AND ANALYTIC GEOMETRY

In technical drafting it is often necessary to use theorems of trigonometry and analytic geometry in order to find points needed in construction of figures. If we want to write programs to find points automatically, it is useful to have at our disposal a library of functions to evaluate the relevant formulas. Only a few of those that we need are built into Python. You will recall, from the discussion earlier in the chapter, for example, that standard Python does not have enough math library. You will probably need to write a few functions of your own for use later as building blocks of your graphic programs.

You may find it more convenient, for example, to specify angles in degrees rather than radians. A function to perform the conversion is:

[sample code on the right side]

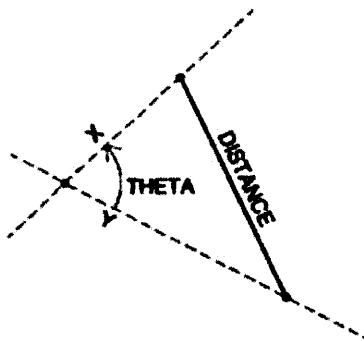
Here is an example of something a bit more complicated. Surveyors often find a distance by triangulation (fig. 7-10). They measure an angle Theta and distances X and Y to obtain data necessary to calculate Distance. The function-needed to obtain Distance may be written:

[sample code on the right side]

So a useful program to read in Theta (in degrees) and the distances X and Y can be written as follows:

[sample code on the right side]

One subtle technical point should be noted here the formal parameters Theta, X, and Y of Distance have the same identifiers as the corresponding actual parameters. However, the formal parameters are local to the procedure Distance and are not the same thing as the actual parameters. When Distance is invoked, the current values of the actual parameters are assigned to the corresponding formal parameters. But, if Distance assigned a value to one of its formal parameters, this would have no effect on the actual parameters; a function can never create a side effect by assigning values to its formal parameters. In other words, the formal parameters of a function are like valves that let data into a function, but do not let modified data back out.



7-10. The principle of triangulation.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def radians(degree):
    pi = 3.1415
    radians = degree * 180 / pi
    return radians

def distance(theta, x, y):
    dist = math.sqrt(x*x + y*y - 2*x*y*math.cos(theta))
    return dist

def triangulation():
    # prompt for and read in data
    angle = rs.GetReal('enter angle in degrees')
    length_1 = rs.GetReal('enter first length')
    length_2 = rs.GetReal('enter second length')

    # convert angle
    theta = radians(angle)

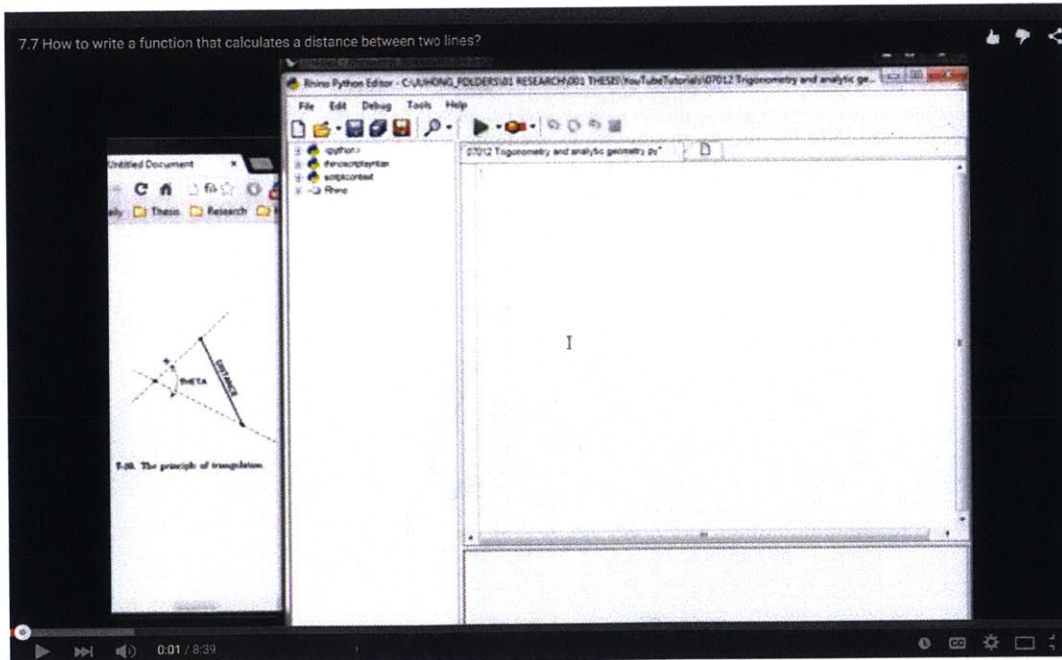
    # calculate distance
    dist = distance(theta, length_1, length_2)

    # write out result
    print 'distance = ', dist

triangulation()
```

RESULT

```
distance = 81.5416847127
```



Module 43: 7.8 Summary

7. COORDINATE CALCULATIONS

7.8 SUMMARY

We have seen, in this chapter, how Python standard functions and functions that you write yourself can be used to clarify, simplify, and shorten programs that carry out complicated calculations. You should make sure that you thoroughly understand the rules for declaration and invocation of functions as well as for the passing of data. You should structure your programs logically by breaking them down into short, easily understandable functions.

You will find that this modularization helps you to write correct programs and to quickly trace and fix errors that do occur. The insulation around a function localizes the effect of any error within it and prevents the propagation of mysterious side effects. You will also find that building up a library of carefully written, generalized functions provides you with reusable program modules that can serve as building blocks in later programs.

The functions that you build for calculating coordinates will become your tools for correctly sizing and placing elements of a graphic composition. As such, they will encode both technical knowledge (of how to calculate the tangent of an angle or the midpoint of a line, for example) and aesthetic rule - in particular, rules of proportion.

Module 44: 8. Graphic

8. GRAPHIC VOCABULARIES

We began our discussion of line drawings by observing that they are composed of individual primitive marks, such as pencil or pen strokes. For our purposes here, we have taken the primitive mark to be a vector—a straight line segment defined by the coordinates of its endpoints.

A drawing is made by executing a sequence of primitive marking operations, such as movements of a pencil held in the hand. Thus you can specify a picture by giving a sequence of commands to execute primitive marking operations one after the other. This works, although it is very tedious—much like instructing, over the telephone, a draftsman who cannot understand any task more complicated than drawing a single line.

Module 45: 8.1 Picture

8.

GRAPHIC VOCABULARIES

8.1 PICTURES AS PROGRAMS

We have now seen how a computer can be programmed in exactly the same laborious, step-by-step way to generate a picture composed of vectors. You write a sequence of points and commands. When the computer executes these commands, one after the other, the specified drawing is generated on the screen. In other words, a picture is encoded as a Python program that specifies the sequence of primitive marking operations (`rs.AddLine()`) needed to create it.

If we want to refer to a picture, we usually give it a name, such as square, triangle, squiggle, north elevation, or Mona Lisa. It makes sense to give the same name to the program that generates the picture. In Python, this name becomes the heading of the program.

In summary, then, we have seen that you can write a Python program to generate a line drawing as follows:

- Import rhinoscriptsyntax library as rs, so you can use built-in functions such as `rs.AddLine()`
- Declare the constants and variables that will be needed, much as you might provide a draftsman with a scratchpad to jot down numbers and an electronic calculator with a useful array of function keys.
- Assign or read in values for the independent variables.
- Calculate values of any dependent variables.
- Write rhino's built-in commands/functions to generate the picture.

You must always have values for independent variables before you can calculate values for dependent variables, and you must always have X and Y coordinate values before you can execute a move or draw. One logical way to organize code, then, is in distinct steps, as in the following code to draw an equilateral triangle.

[sample code on the right side]

The choice between these alternatives, and among possible combinations of the two, is a matter of programming style, not of Python syntax. You should always choose the clearest, most expressive method.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

# assign values to independent variables
x1 = 300          # bottom-left vertex coordinate
y1 = 300          # bottom-left vertex coordinate
side = 400        # side length

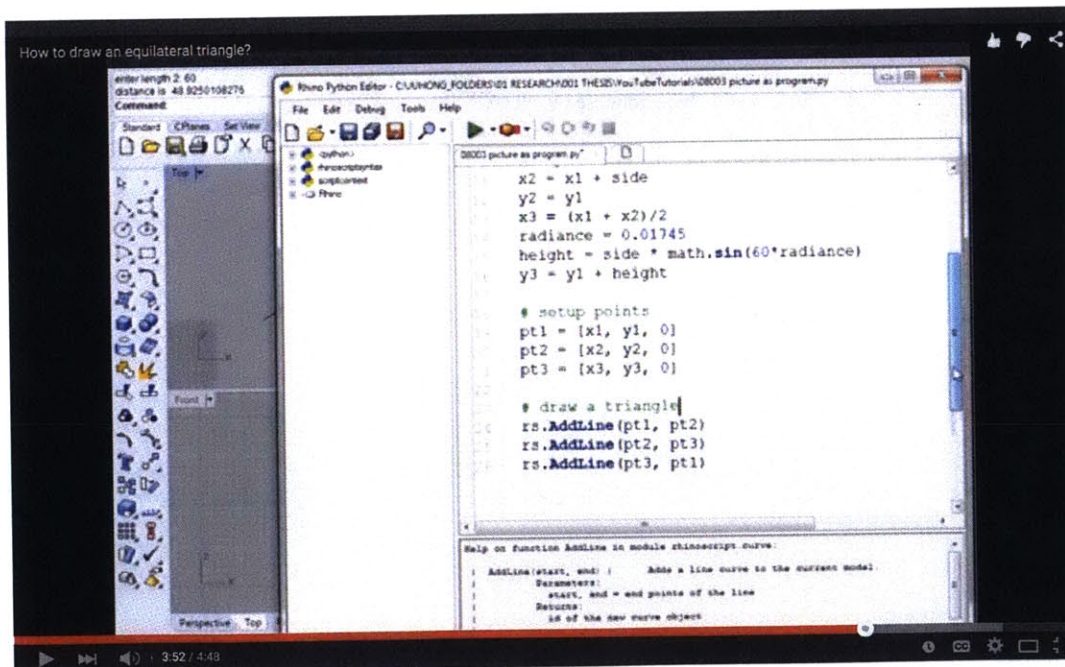
# calculate values of dependent variables
x2 = x1 + side    # bottom right vertex coordinate
x3 = (x1 + x2) / 2 # apex x-coordinate

radianc = 0.01745
height = side * math.sin(60 * radianc)
y2 = y1 + height # apex h-coordinate

pt1 = [x1, y1, 0]
pt2 = [x1, y2, 0]
pt3 = [x3, y2, 0]

# draw triangles
rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt1)
```

RESULT



Module 46: 8.2 Parts

8. GRAPHIC VOCABULARIES

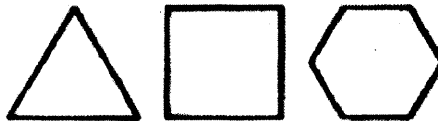
8.2 PARTS OF PICTURES AND PARTS OF PROGRAMS

So far we have considered only very simple figures, composed of a few vectors. When we look at more complex pictures, we usually find that they have a number of distinct parts. Figure 8-1, for example, is composed of a triangle, a square, and a hexagon.

It is convenient to break down a graphic program to draw such a picture in corresponding distinct parts—one part of the program draws a triangle, one a square, and one a hexagon. The program then has an internal structure that reflects the structure of the image. (This mirroring of structure is a very important general principle, and we will come back to it later). One way to do this is simply to insert appropriate points in a list using brackets (like `pt1 = [400, 200, 0]`) and list statements as follows:

[sample code on the right side]

Thus we can immediately see what each group of points and statements do.



8-1. A picture composed of three distinct parts: a triangle, a square, and a hexagon.

Synthetic Tutor

CODE

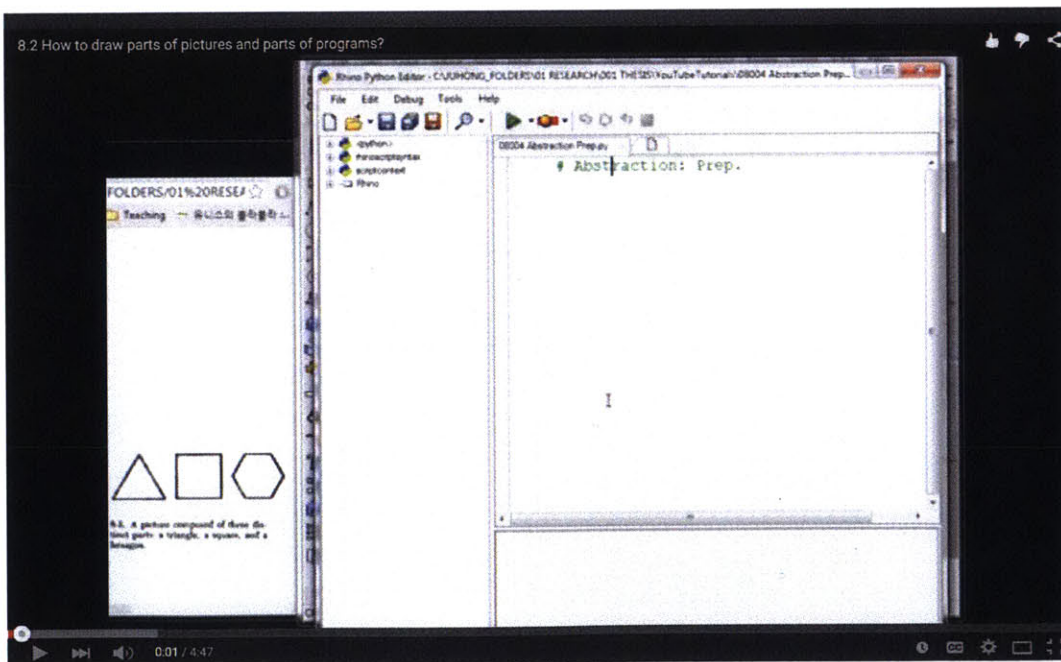
```
import rhinoscriptsyntax as rs
import math

# draw a triangle
pt1 = [100,200,0]
pt2 = [300,200,0]
pt3 = [200,400,0]
rs.AddLine( pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt1)

# draw a square
pt1 = [400,200,0]
pt2 = [600,200,0]
pt3 = [600,400,0]
pt4 = [400,400,0]
rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt4)
rs.AddLine(pt4, pt1)

# draw a hexagon
pt1 = [758,200,0]
pt2 = [874,200,0]
pt3 = [932,300,0]
pt4 = [874,400,0]
pt5 = [758,400,0]
pt6 = [700,300,0]
rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt4)
rs.AddLine(pt4, pt5)
rs.AddLine(pt5, pt6)
rs.AddLine(pt6, pt1)
```

RESULT



Module 47: 8.3 Procedure 1

8. GRAPHIC VOCABULARIES

8.3 PROCEDURES

A more sophisticated approach is to write a separate procedure for each part. A procedure in Python has a similar organization as a program, with a heading, a declaration part, and an executable part. For example, a procedure to draw a square looks very much like a program to draw a square:

[sample code on the right side]

Note, however, that this reflects the fact that a procedure is a distinct part of a program, not a complete program in itself.

As you can see, Python does not actually distinguish between procedures and functions (In this workshop, procedures and functions are used interchangeably). Like functions, procedures are first declared, then used (invoked) within a program. The following example, in which the program Draw square invokes the procedure Square, illustrates the logical distinction between declaration and invocation (You can think a procedure as a function without an output).

[sample code on the right side]

The procedure Square is declared by writing the code to draw a square and then invoked by name in the executable part of the program. There are a couple of rules to be followed here. Just as you cannot use a tool until you have it in your hand, you cannot invoke a procedure (just as you cannot invoke a function), unless you have already declared it. In our example, declaration of the procedure Square, in effect, makes a tool to draw a square available to the computer. Then, by giving the command Square in the executable part of the program, we instruct the computer to use this tool. Once a procedure has been declared, it may be invoked any number of times, just as a tool, once in hand, may be used over and over again.

Synthetic Tutor

CODE

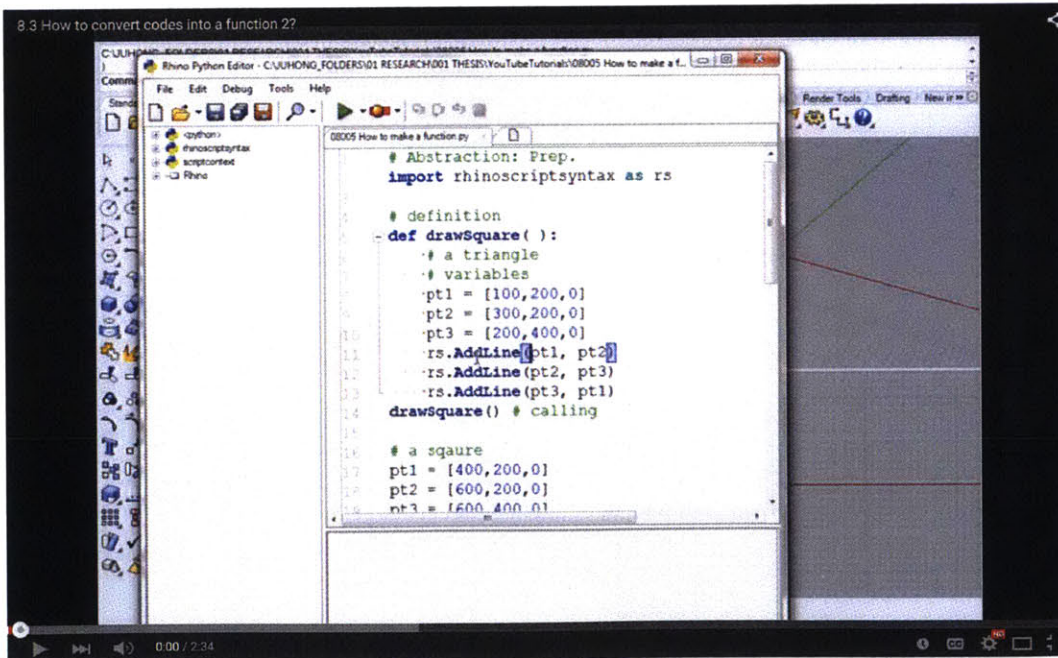
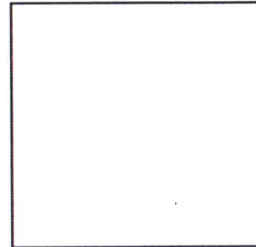
```
import rhinoscriptsyntax as rs
import math

pt1 = [400,200,0]
pt2 = [600,200,0]
pt3 = [600,400,0]
pt4 = [400,400,0]
rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt4)
rs.AddLine(pt4, pt1)

def drawSquare():
    pt1 = [400,200,0]
    pt2 = [600,200,0]
    pt3 = [600,400,0]
    pt4 = [400,400,0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

drawSquare()
```

RESULT



Module 48: 8.3 Procedure 2

8. GRAPHIC VOCABULARIES

8.3 PROCEDURES (continued)

To draw our example picture, now, we might declare procedures Square, Triangle, and Hexagon, then invoke them to create the picture:

[sample code on the right side]

The invocations now consist of commands at a higher level than `rs.AddLine()`. Instead of specifying primitive marks (points) one by one, it specifies more complicated figures consisting of sets of primitive marks. The procedures that have been declared tell how these more complicated figures are put together out of primitive marks. In other words, the construct of a procedure provides us with an extremely powerful means of abstraction we can associate a name with an arbitrarily complex arrangement of vectors.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

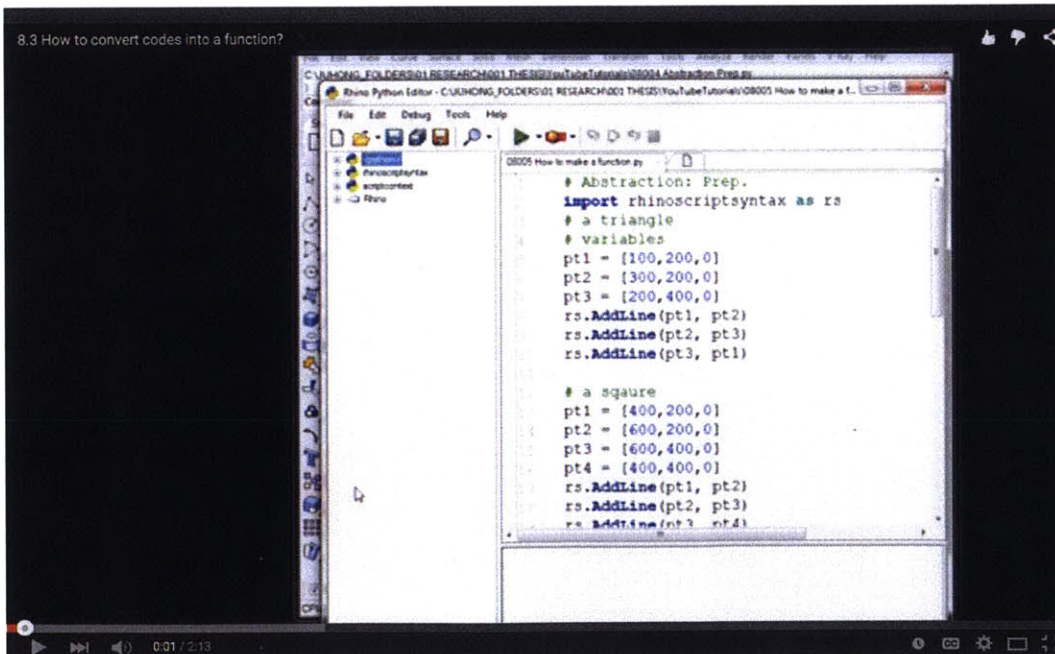
def drawSquare():
    pt1 = [400, 200, 0]
    pt2 = [600, 200, 0]
    pt3 = [600, 400, 0]
    pt4 = [400, 400, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawTriangle():
    pt1 = [100, 200, 0]
    pt2 = [300, 200, 0]
    pt3 = [200, 400, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt1)

def drawHexagon():
    pt1 = [758, 200, 0]
    pt2 = [874, 200, 0]
    pt3 = [932, 300, 0]
    pt4 = [874, 400, 0]
    pt5 = [758, 400, 0]
    pt6 = [700, 300, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt5)
    rs.AddLine(pt5, pt6)
    rs.AddLine(pt6, pt1)

drawTriangle()
drawSquare()
drawHexagon()
```

RESULT



Module 49: 8.3 Abstraction

8. GRAPHIC VOCABULARIES

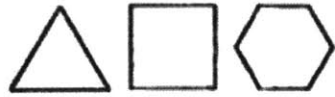
8.3 PROCEDURES

8.3.1 THE ART OF ABSTRACTION

In the example that we have been discussing, we have broken our drawing down, in the most obvious way, into a triangle, a square, and a hexagon. But there are many other possible ways to separate this picture into parts. Figure 8-2 shows just a few of these possibilities. We could, of course, write a procedure to generate any one of these parts.

The crucial point here is that a way of breaking a picture down into parts is not somehow given, but represents a choice among alternatives—usually many of them. We choose the method that seems most natural and useful to us. In our example, we recognized familiar, closed, symmetrical figures that have well-known English names and did not consider the more bizarre possibilities.

Sometimes, however, the choice is not so obvious. Figure 8-3 shows a simple line figure and a variety of plausible and interesting ways of breaking it into parts. The method that is chosen is the result of artistic vision, not of the mechanics of computer technology. The structuring of a Python graphics program as a set of procedures merely reflects a commitment to a particular vision and makes it explicit.

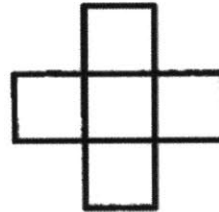


a. The original picture.

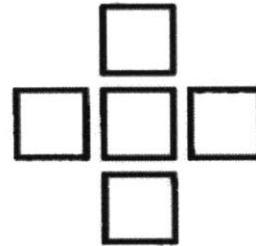


b. Several different ways to break it into distinct parts.

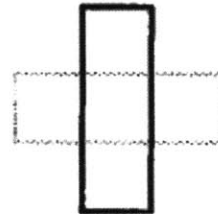
8-2. Alternative decompositions of a picture.



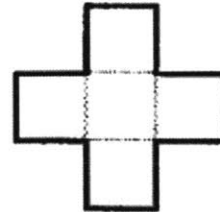
a. The original composition.



b. The composition viewed as four squares connected at their corners.



c. The composition viewed as two overlapping rectangles.



d. The composition viewed as a square superimposed on a cross.

8-3. A variety of ways to see the same composition.

Module 50: 8.3 Parameter 1

8. GRAPHIC VOCABULARIES

8.3 PROCEDURES

8.3.2 PARAMETERIZATION OF GRAPHIC ELEMENTS

If the unintelligent human draftsman that we considered at the beginning of this chapter at least understood the concepts of a triangle, square, and hexagon, we could give him concise telephone instructions that sound much like the main part of our program. We could say:

Draw a triangle
Draw a square
Draw a hexagon

But he would ask some questions:

- . How big is the square?
- . Where is it placed on the surface?
- etc.

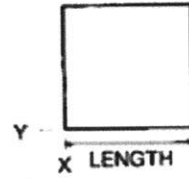
The issue that emerges here is the distinction between the essential and accidental properties of an object. All squares share certain properties:

- . Four straight sides
- . Parallel and equal opposite sides
- . 90-degree vertex angles

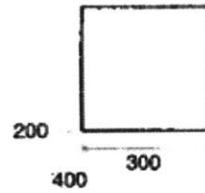
These properties are essential to a square they define a square. But size and position are accidental properties they may vary from square to square. So, if somebody tells you to draw a square, you know what shape to make it, but you must determine how big it will be and where to put it. In other words, the essential properties of an object are those that it shares with all others of the same type, whereas accidental properties may vary among the instances of the type and must be specified to identify a particular instance.

To depict a general type of graphic element (rather than one of its instances), a diagram that shows the essential properties is used. It is labeled with variable names and dimension lines, according to the standard conventions of technical drafting, to identify the accidental properties. Figure 8-4a, for example, diagrams a square with sides parallel to the coordinate axes and indicates that X and Y (coordinates of the bottom-left corner) as well as side Length are accidental properties. That is, the diagram stands for all the squares, of any size, that may be instantiated in the coordinate system. A particular instance of this type can be specified by substituting values for variable names (fig. 8-4b), as in an unsealed architectural drawing, or by redrawing the element to the correct scale as indicated by these values (fig. 8-4c).

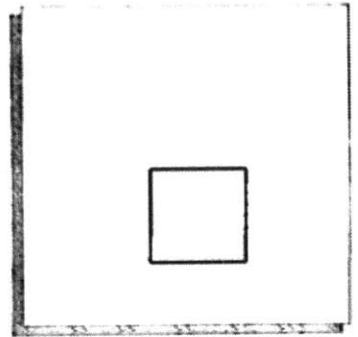
a. A type diagram given variable names.



b. An instance specified by substituting values for variable names.



c. The same instance redrawn correctly to scale.



8-4. A square with sides parallel to the axes of the coordinate system.

Module 51: 8.3 Parameter 2

8. GRAPHIC VOCABULARIES

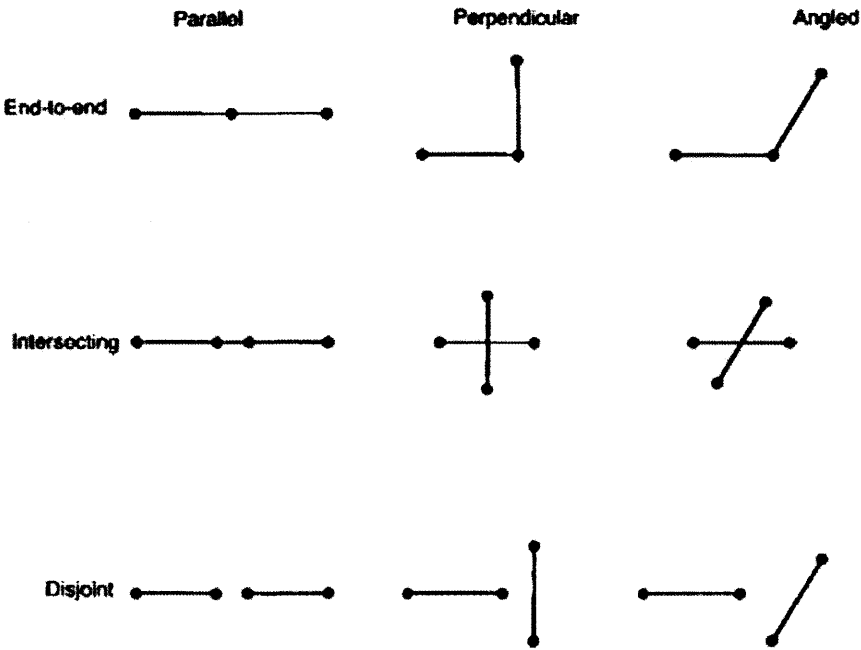
8.3 PROCEDURES

8.3.2 PARAMETERIZATION OF GRAPHIC ELEMENTS

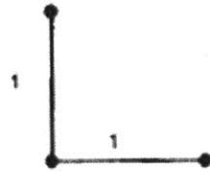
The type diagram depicts the spatial relation between vectors that characterize the type (that are essential and found in all instances). These relations are of connection, direction, and length. A pair of vectors may have any one of three connectivity relations (fig. 8-5). They may be connected end-to-end, intersect, or be disjoint. We also commonly recognize three possible relations of direction. Vectors may be parallel, perpendicular, or angled in relation to each other.

The lengths of pairs of vectors may be related in particular ratios. The Pythagorean philosophers, for example, and following them Renaissance architects, like Andrea Palladio, attached particular aesthetic importance to ratios of small whole numbers such as in figure 8-6:

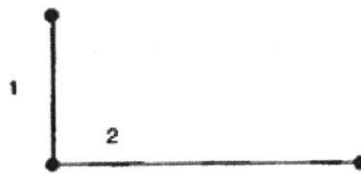
- . 1:1 identity
- . 1:2 octave
- . 2:3 fifth
- . 3:4 fourth



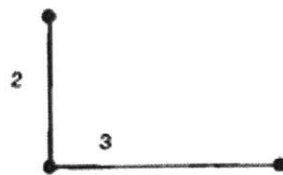
8-5. The relations of connectivity and angle between vectors.



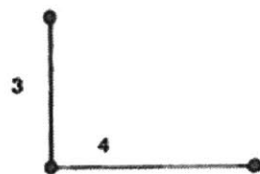
a. Identity.



b. Octave.



c. Fifth.



d. Fourth.

8-6. The lengths of vectors related by ratios of small whole numbers.

Module 52: 8.3 Parameter 3

8. GRAPHIC VOCABULARIES

8.3 PROCEDURES

8.3.2 PARARNETERIZATION OF GRAPHIC ELEMENTS

These ratios often govern Palladio's plans and elevations (fig. 8-7). Coordinates of endpoints may also be related in specified ratios (fig. 8-8). This amounts to specifying a ratio of the lengths of "invisible" vectors.

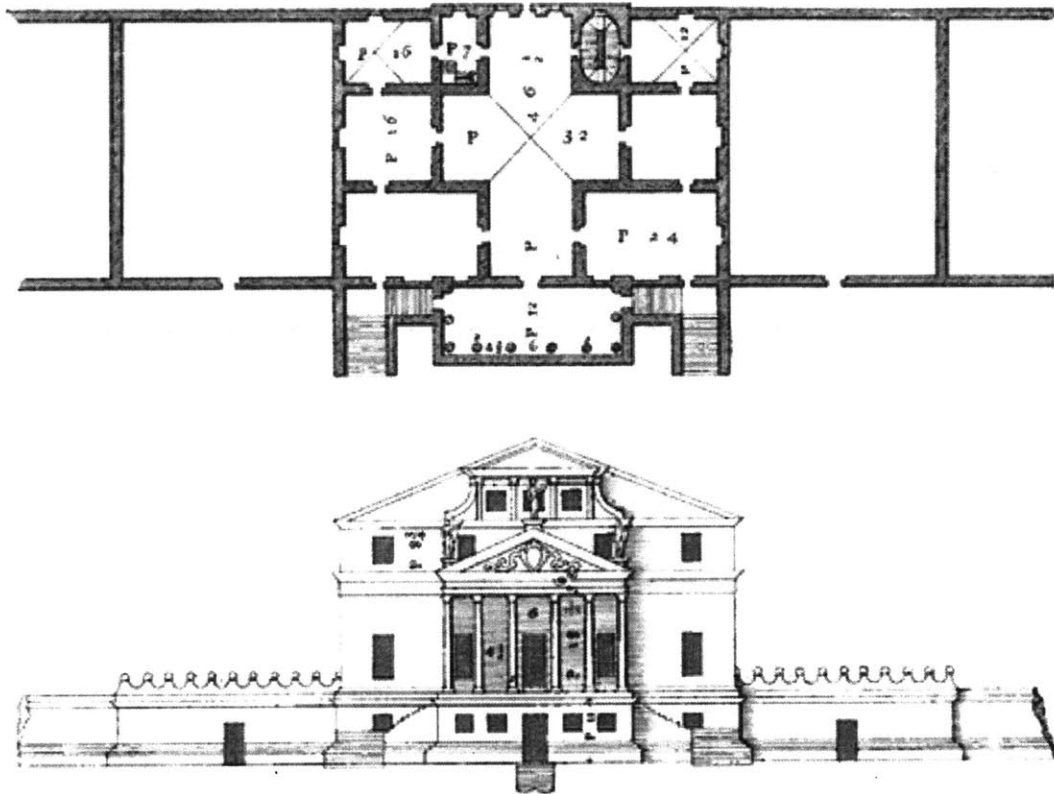
A square, then, is characterized by the following connectivity relations (fig. 8-9a):

- . side 1 is connected to side 2
- . side 2 is connected to side 3
- . side 3 is connected to side 4
- . side 4 is connected to side 1
- . no sides intersect

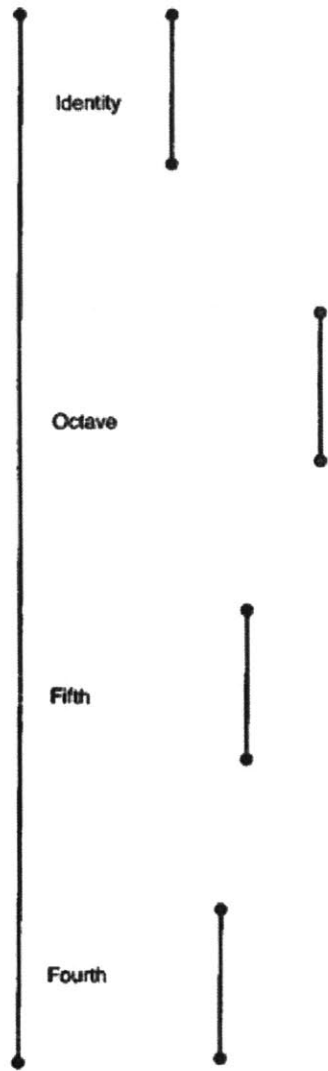
It also has the following relations of direction (fig. 8-9b):

- . side 1 is parallel to side 3
- . side 2 is parallel to side 4
- . side 1 is perpendicular to side 2
- . side 2 is perpendicular to side 3
- . side 3 is perpendicular to side 4
- . side 4 is perpendicular to side 1

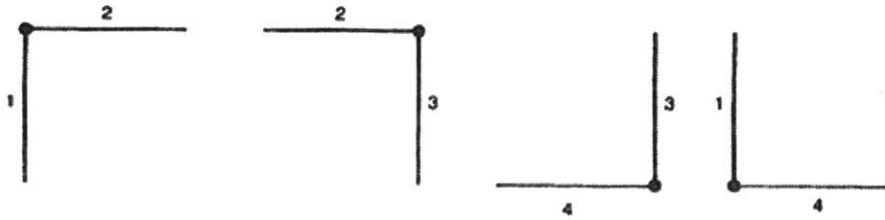
Finally, the lengths of all sides are related in the ratio 1:1 (fig. 8-9c).



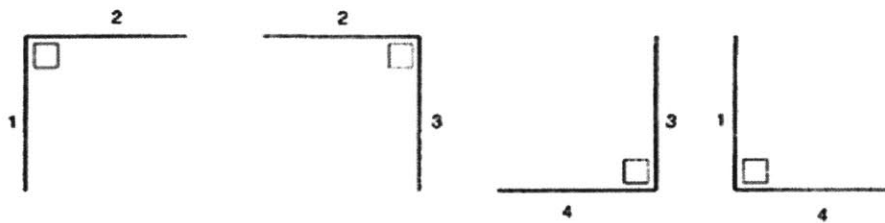
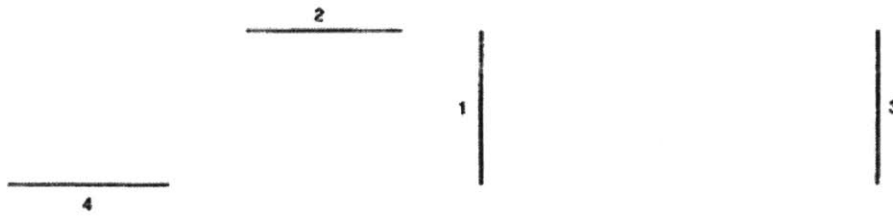
8-7. Plan and elevation of the Villa Malcontenta, from Isaac Ware's edition of Andrea Palladio's *Four Books of Architecture*. Important ratios of lengths are marked.



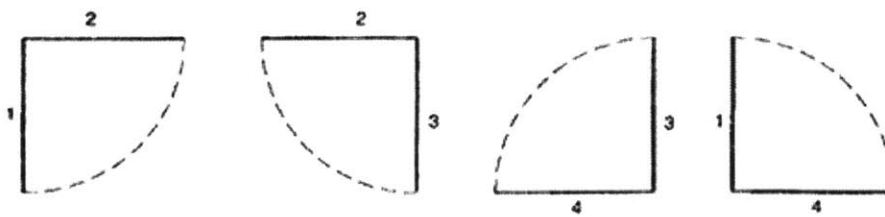
8-8. Ratios of the length of a vector to the distance from a parallel vector.



a. Connectivity.



b. Direction (parallels and perpendiculars).



c. Ratios of side lengths.

8-9. Relations of vectors that characterize a square.

Module 53: 8.3 Parameter 4

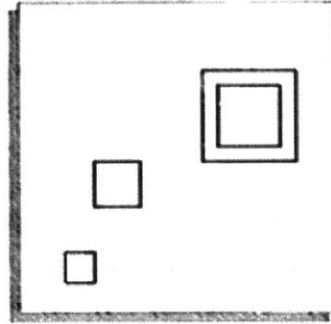
8. GRAPHIC VOCABULARIES

8.3 PROCEDURES

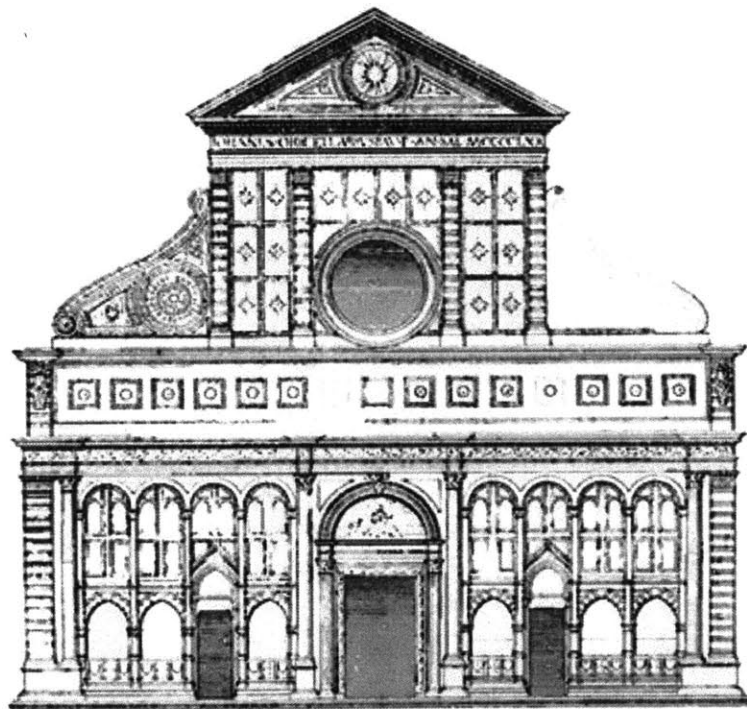
8.3.2 PARARNETERIZATION OF GRAPHIC ELEMENTS

The variables associated with the diagram are called the parameters of the graphic element. Each parameter has a name (for example, Length), a type (for example, integer or real number), and a range (for example, integers from 0 to 1,000). A graphic element with parameters is called a parameterized or parametric element. When specific values from their ranges are assigned to the parameters, a particular instance is defined. Figure 8-10 illustrates some of the many possible instances of our square, all defined by assigning values to the parameters.

Composition of a motif can be understood, then, as a process of specifying essential relations of connectivity, direction, and ratio in a set of vectors. All instances of the same type of motif will consist of vectors related in the same way. This formalizes, in a fashion suitable for our purposes here, a famous definition of design given by the Renaissance architect and theorist Leon Battista Alberti in his *Ten Books on Architecture*: the right and exact adapting and joining together of the lines and angles which compose and form the face of the building (fig. 8-11).



8-10. Squares instantiated within a screen coordinate system.



8-11. "A right and exact adapting and joining together the lines and angles which compose and form the face of the building": a composition of parallels, perpendiculars, simple whole number ratios, and instances of squares by Alberti.

Module 54: 8.3 Parameter 5

8. GRAPHIC VOCABULARIES

8.3 PROCEDURES

8.3.3 PARAMETERIZED PROCEDURES

Just as we used procedures Square, Triangle, and Hexagon earlier to generate the corresponding figures, we can use parameterized procedures to generate instances of parametric objects, such as our square (fig. 8-12).

The first step in writing a parameterized procedure is to list the formal parameters in parentheses after the procedure name and declare their types. For example, we might begin a parameterized procedure to draw a square:

```
def Square (X, Y, Length):
```

Next we express our commands/functions (`rs.AddLine()`) in terms of the three formal parameters (fig. 8-13). The coordinates X and Y are given directly by the formal parameters. The coordinate X_2 is given by $X + \text{Length}$, and the coordinate Y_2 is given by $Y + \text{Length}$. Using these coordinate values, we can write the procedure as follows:

[sample code on the right side]

Notice that the code within this procedure produces any square with sides parallel to the coordinate axes, (as specified by the parameter values) and only such squares, so it expresses the essence of the type Square. This is done by means of arithmetic expressions and assignments that define the appropriate connectivity, direction, and ratio-of-length relations. Then each `rs.AddLine` generates a line that is spatially related in the appropriate way to its predecessors. The first `rs.AddLine` produces a line starting at point $(X, Y, 0)$, parallel to the X axis, and of the specified Length (fig. 8-14a). The second produces a second line connected to the end of the first, perpendicular to it, and in a Length ratio of 1:1 (fig. 8-14b). The third produces a third line, connected to the end of the second, perpendicular to it, and once again, in a Length ratio of 1:1 (fig. 8-14c). Finally, the fourth `rs.AddLine` produces a fourth line that connects the end of the third line back to the beginning of the first (fig. 8-14d).

The formal parameter list, on the other hand, specifies whatever it is that we want to vary about squares—the accidental properties of squares that we want to control. If we want to make a composition out of squares, the formal parameters establish our graphic variables. In this case, the parameters are x , y , and Length, corresponding to the variables shown on our diagram.

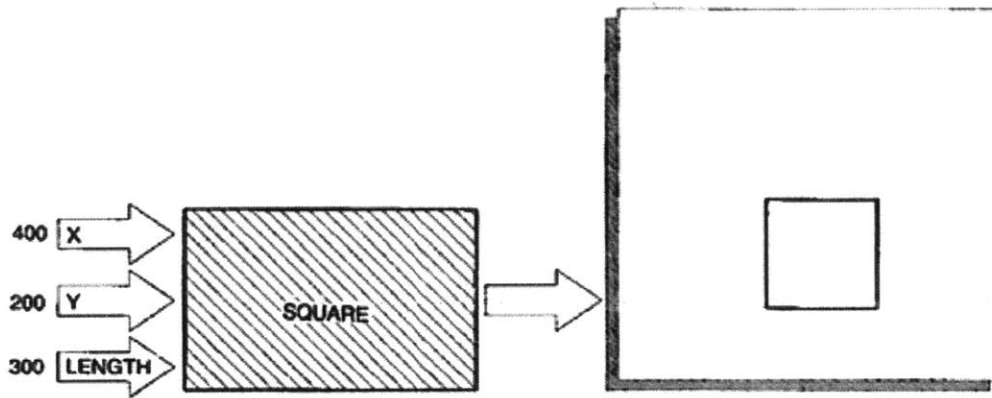
Once such a parameterized procedure has been declared, it may be invoked with defined parameter values to generate an instance. The following statements first assign values to X , Y , and Length, then invoke Square with these values as actual parameters to generate the corresponding instances:

```
x = 400  
y = 200  
Length = 300  
Square(x, y, length)
```

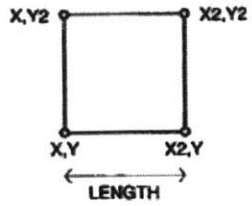
Instead of using separate assignment statements, we can more concisely write:

```
Square (400, 200, 300)
```

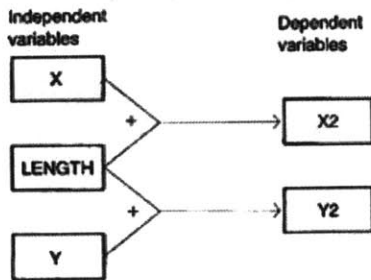
That is, we can specify values for X , Y , and Length directly in the list of actual parameters.



8-12. The action of a parameterized procedure to draw a square; parameter values are passed in, and the corresponding instance is passed out.

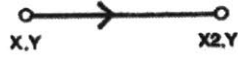


a. The type diagram.

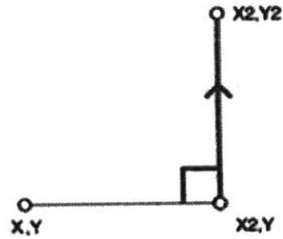


b. Independent and dependent variables.

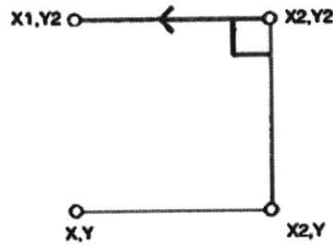
8-13. Vertex coordinates of a square expressed in terms of X, Y, and Length.



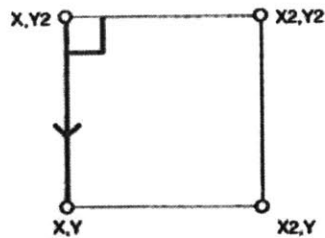
a. First vector drawn from point (X, Y), parallel to the X axis, and of specified Length.



b. Second vector connected to end of first, perpendicular to it, and in ratio 1:1.



c. Third vector drawn in the same relationship to the second as the second is to the first.



d. Fourth vector connects to the beginning of the first.

8-14. The procedure Square spatially relates four vectors.

CODE

```
import rhinoscriptsyntax as rs

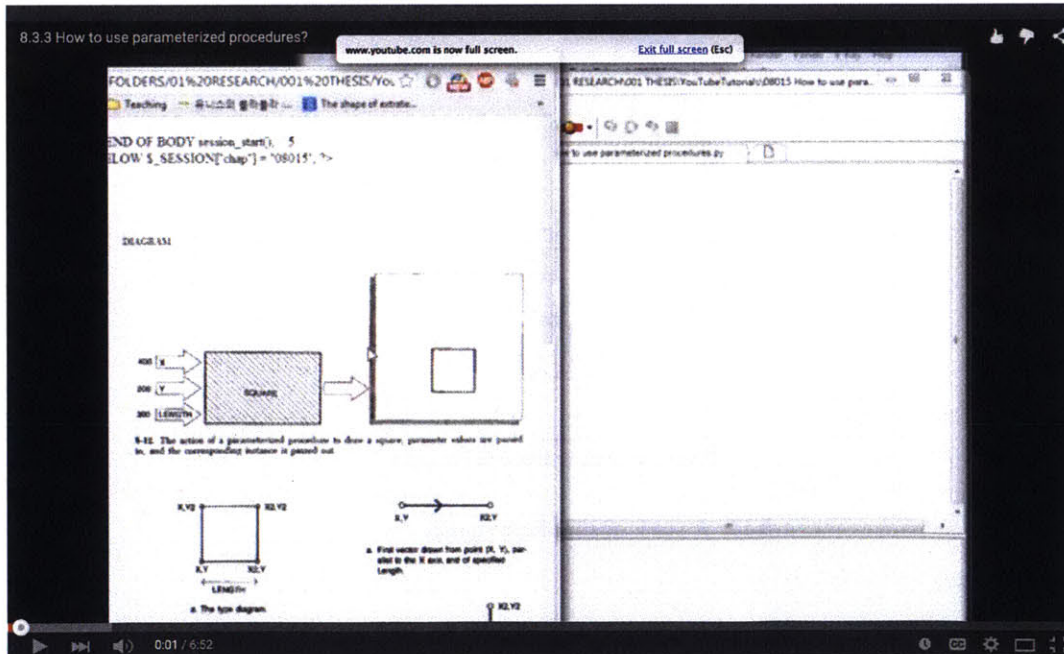
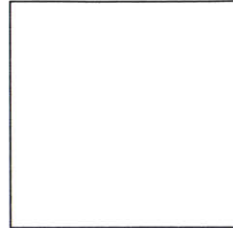
def drawSquare(x, y, length):
    # calculate values for x2 and y2
    x2 = x + length
    y2 = y + length

    # set the points of four corners
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]

    # draw line
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

drawSquare(10, 10, 100)
```

RESULT



Module 55: 8.3 Parameter 6

8. GRAPHIC VOCABULARIES

8.3 PROCEDURES

8.3.3 PARAMETERIZED PROCEDURES (continued)

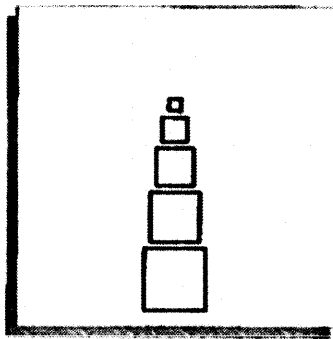
If we repeatedly invoke Square, with different X, Y, and Length values, we produce a picture composed of many instances of squares.

[sample code on the right side]

We could produce exactly the same result with the following, which specifies each line directly by a rs.AddLine:

[sample code on the right side]

But this program is lengthy and difficult to follow, and unlike the program that invokes the parameterized procedure Square, it does not take advantage of our knowledge that the picture is composed entirely of instances of squares with sides parallel to the coordinate axes. So the program that invokes Square is more concise and tells us more about the structure of the drawing that it generates.



8-15. A stack of shrinking squares.

CODE

```
import rhinoscriptsyntax as rs

def drawSquare(x, y, length):
    # calculate values for x2 and y2
    x2 = x + length
    y2 = y + length

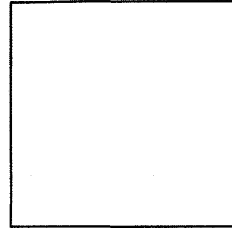
    # set the points of four corners
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]

    # draw line
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def main():
    drawSquare(400, 50, 200)
    drawSquare(420, 270, 160)
    drawSquare(440, 450, 120)
    drawSquare(460, 590, 80)
    drawSquare(480, 690, 40)

main()
```

RESULT



Synthetic Tutor

```
def drawFiveSquares():
    # draw the first square with side = 200
    x = 400
    y = 50
    x2 = x + 200
    y2 = y + 200
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

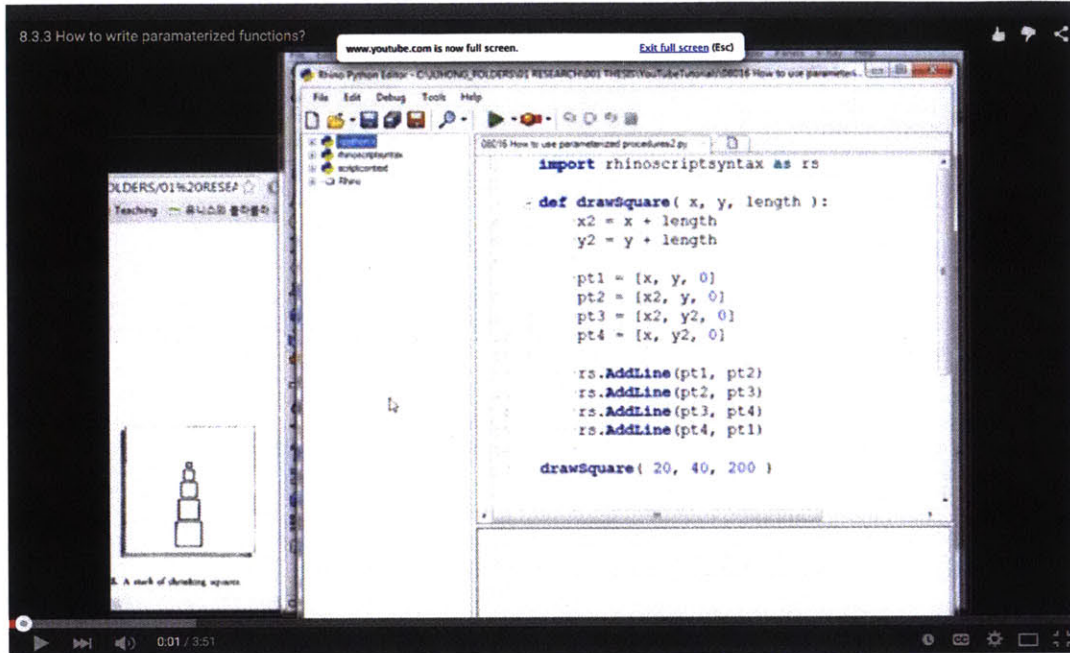
    # draw the first square with side = 160
    x = 420
    y = 270
    x2 = x + 160
    y2 = y + 160
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

    # draw the first square with side = 120
    x = 440
    y = 450
    x2 = x + 120
    y2 = y + 120
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

    # draw the first square with side = 80
    x = 460
    y = 590
    x2 = x + 80
    y2 = y + 80
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

    # draw the first square with side = 40
    x = 480
    y = 690
    x2 = x + 40
    y2 = y + 40
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

drawFiveSquares()
```



Module 56: 8.3 Invoking

8. GRAPHIC VOCABULARIES

8.3 PROCEDURES

8.3.4 INVOKING PROCEDURES

The concept of a parameterized graphic procedure and its uses should now be clear. Before going on, though, it will be useful to pause and summarize the rules that must be followed when invoking procedures. These are much like the rules that apply to the invocation of functions.

We have seen, first, that a procedure is invoked by giving its name and the actual parameters. This causes the values of the actual parameters to be assigned to the corresponding formal parameters. The procedure is then executed, and the computer returns to process the next statement following the procedure invocation. The values of the actual parameters can be specified directly in the list as

```
drawSquare (400,200,300)
```

or by assigning values to variables that are then listed as actual parameters, for example:

```
x = 400  
Y = 200  
Length = 300  
drawSquare (X, Y, Length)
```

In this latter case, these variables must be declared in the main program and are local to it. It is not necessary for these variables to have the same identifiers as the formal parameters of the procedure drawSquare. For example, we might write:

```
X_Corner = 400  
Y_Corner = 200  
Side = 300  
drawSquare (X_Corner, Y_Corner, Side)
```

It is the positions these variables occupy in the actual parameter list that are important. In this case the value assigned to the variable X_Corner will be passed to the formal parameter X the value assigned to the variable Y_Corner will be passed to the formal parameter Y and the value assigned to Side will be passed to the formal parameter Length. The only requirement is that the type of the variable must match the type of the corresponding parameter. In this case X_Corner, Y_Corner, and Side must all be declared as integer variables.

The following program illustrates these points by showing yet another way to create the stack of squares in figure 8-15:

[sample code on the right side]

Note that in order to draw a different set of stacked squares, one need change only the initially assigned values of the variables (X_Corner, Y_Corner, Side, and Increment). Notice also that several statements are repeated each time a square is drawn. In chapter 9, we will see how to eliminate this redundancy.

CODE

```

import rhinoscriptsyntax as rs

def drawSquare(x, y, length):
    # calculate values for x2 and y2
    x2 = x + length
    y2 = y + length

    # set the porints of four corners
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]

    # draw line
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawFiveSquaresIncremental():
    x = 400
    y = 50
    side = 200
    increment = 20

    # draw the first square
    drawSquare(x, y, side)

    # draw the second square
    x = x + increment
    y = y + side + increment
    side = side - 2*increment
    drawSquare(x, y, side)

    # draw the third square
    x = x + increment
    y = y + side + increment
    side = side - 2*increment
    drawSquare(x, y, side)

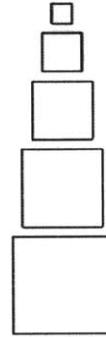
    # draw the fourth square
    x = x + increment
    y = y + side + increment
    side = side - 2*increment
    drawSquare(x, y, side)

    # draw the fifth square
    x = x + increment
    y = y + side + increment
    side = side - 2*increment
    drawSquare(x, y, side)

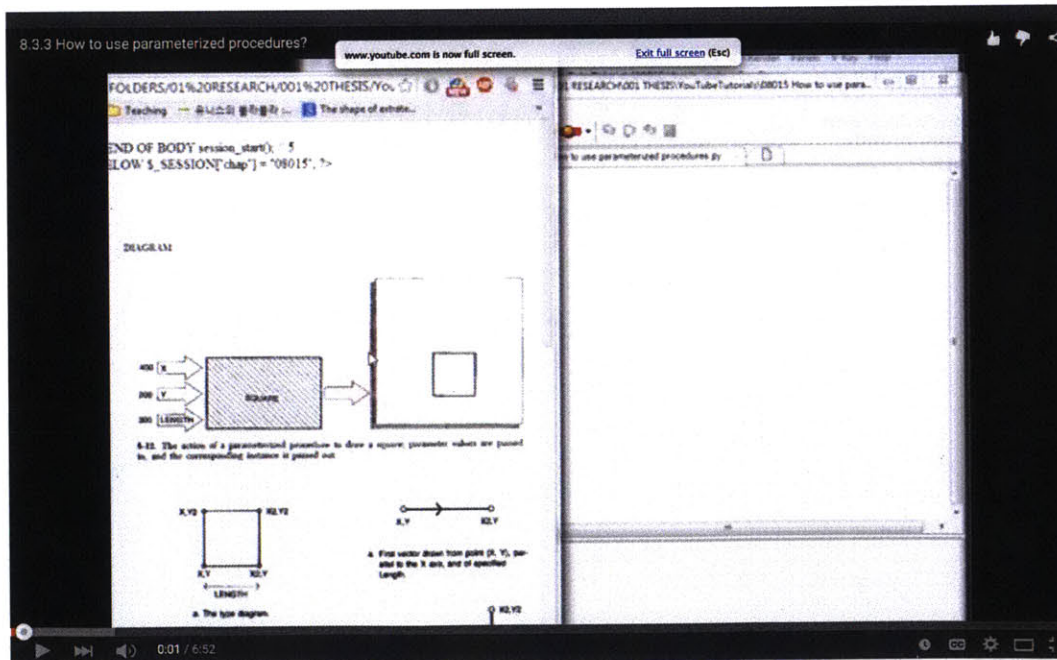
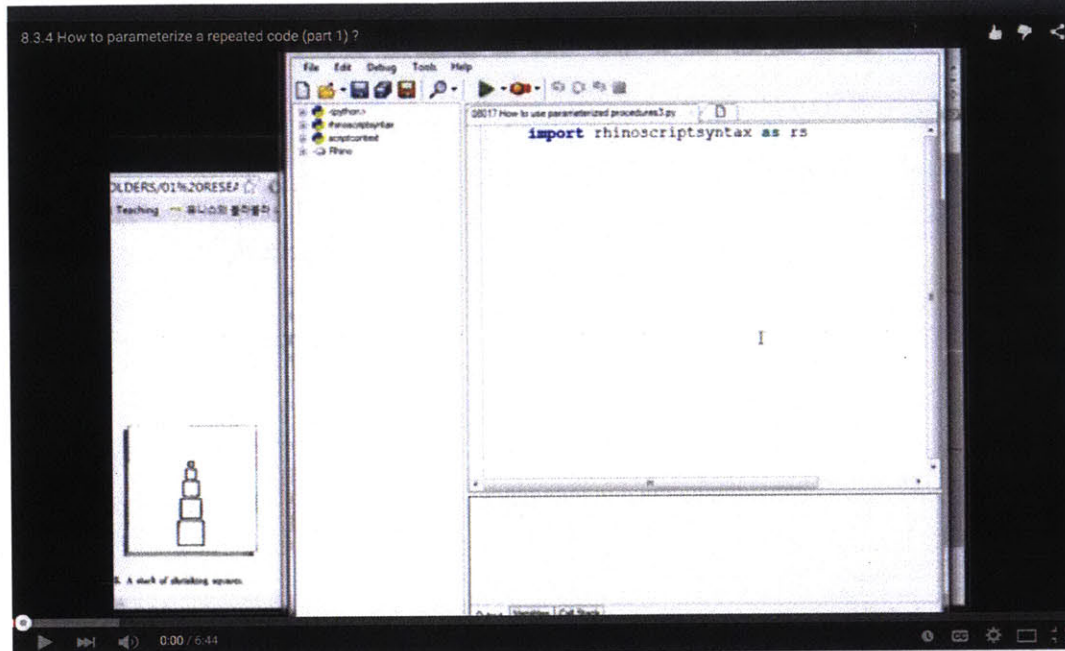
drawFiveSquaresIncremental()

```

RESULT



Synthetic Tutor



Module 57: 8.4 Shape

8. GRAPHIC VOCABULARIES

8.4 SHAPE AND POSITION PARAMETERS

In the remainder of this chapter we shall assume that you have a sound grasp of parameterized procedures that generate a line figure, and we shall concentrate on the logic of parameterization. Specifically, how do you choose and express the parameters of a graphic element?

It is useful to begin by distinguishing between the shape and position parameters of a graphic element. Shape parameters control size, proportion, and other such properties, whereas position parameters control the location on the drawing surface. In Square, for example, Length is a shape parameter and X and Y are position parameters (fig. 8-16).

8.4.1 ALTERNATIVE PARAMETERIZATION SCHEMES

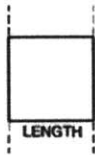
Usually there are different, though mathematically equivalent, ways to specify shape and position parameters. For example, we could locate a square by its center point, rather than its bottom-left corner (fig. 8-17) and rewrite procedure `drawSquareFromCenter()` as follows:

[sample code on the right side]

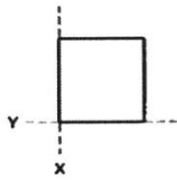
Or we might want to specify squares by giving the coordinates of the top-right corner and the length of the diagonal (fig. 8-18). In this case, `drawSquareFromCorner()` could be written:

[sample code on the right side]

Choice of the parameterization scheme depends on the intended compositional use of the motif. If you intend to fit squares together with edges aligned (fig. 8-19a), you will probably find it convenient to specify position by corner coordinates and shape by side length. Concentric nesting (fig. 8-19b), on the other hand, is easier if position is specified by center coordinates. And corner-to-corner diagonal connection (fig. 8-19c) is easier if shape is specified by the length of the diagonal. If you intend to use a motif in several different ways, you may find it convenient to write a procedure for each.

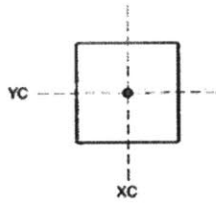


a. Shape parameter of a square.

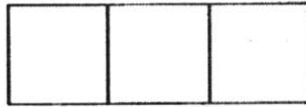


b. Position parameters of a square.

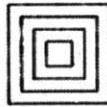
8-16. Shape and position parameters.



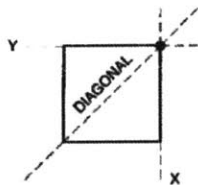
8-17. A square located by its center point.



a. Edges of squares related.



b. Centers of squares related.



c. Diagonals of squares related.

8-18. Another way to parameterize a square.

8-19. Different parameterization schemes facilitate the formation of different relationships between squares.

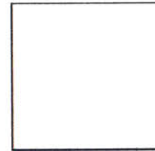
CODE

```
import rhinoscriptsyntax as rs

def drawSquareFromCenter(xc, yc, length):
    x1 = xc - (length/2)
    x2 = x1 + length
    y1 = yc - (length/2)
    y2 = y1 + length
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

drawSquareFromCenter(100, 100, 200)
```

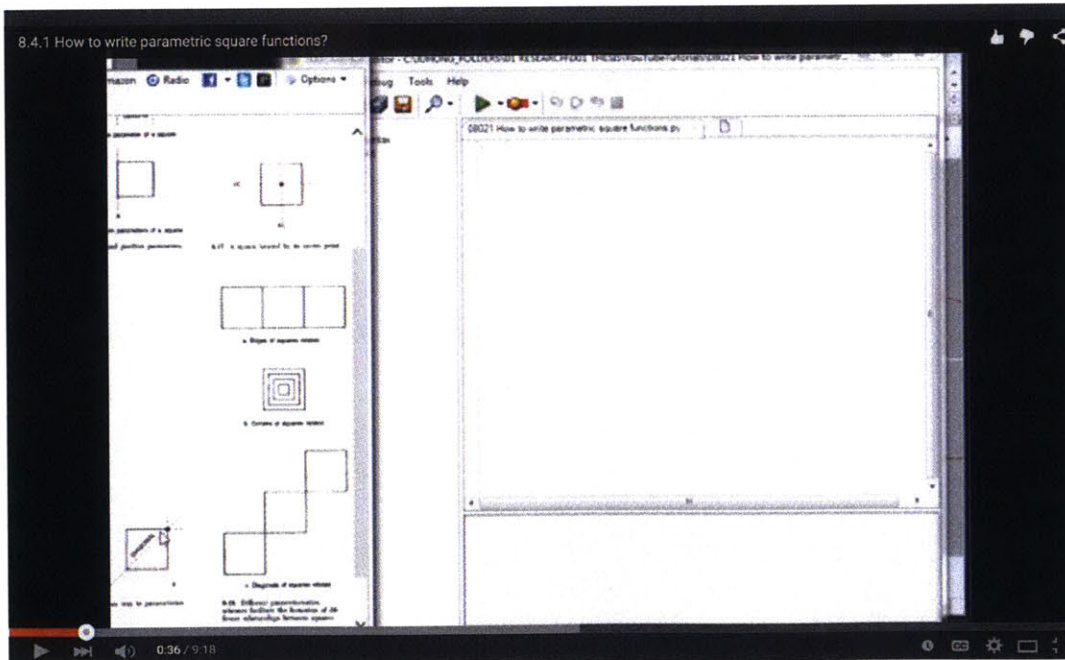
RESULT



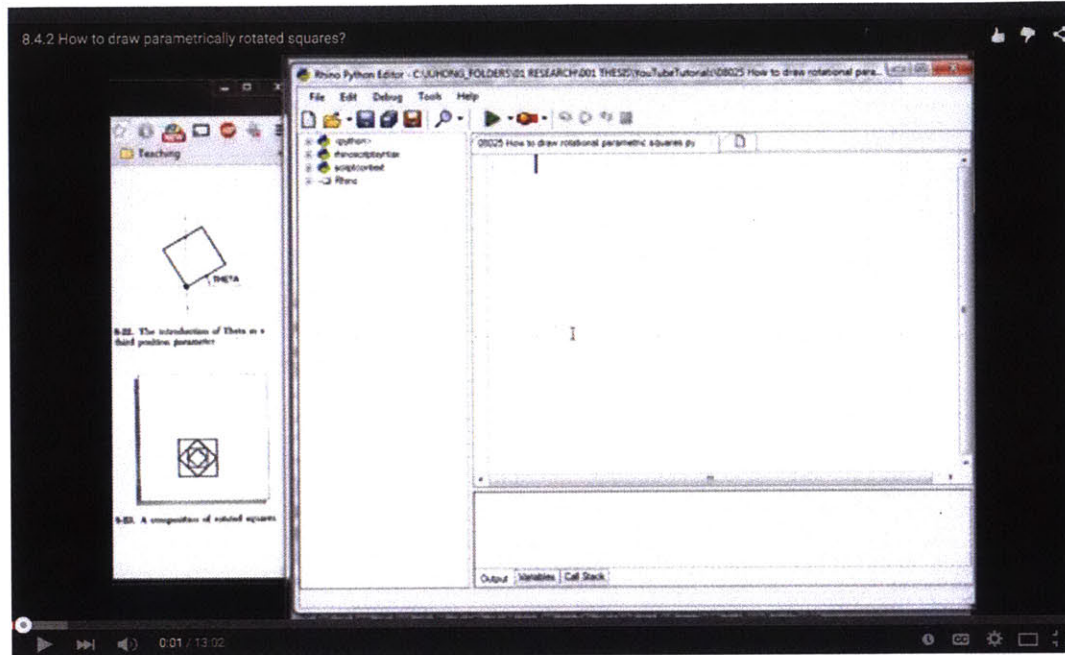
```
import rhinoscriptsyntax as rs
import math

def drawSquareFromCorner(xd, yd, diagonal):
    radiance = 0.01745
    theta = 45 * radiance
    side = diagonal * math.sin(theta)
    x = xd - side
    y = yd - side
    pt1 = [x, y, 0]
    pt2 = [xd, y, 0]
    pt3 = [xd, yd, 0]
    pt4 = [x, yd, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

drawSquareFromCorner(100,100,200)
```



Synthetic Tutor



Module 58: 8.4 Position 1

8. GRAPHIC VOCABULARIES

8.4 SHAPE AND POSITION PARAMETERS

8.4.2 CHOOSING POSITION PARAMETERS

We can establish many or few position parameters to a graphic element, depending on the number of degrees of freedom that we want in positioning the element within the boundaries of the screen coordinate system. In Square we have used two position parameters, X and Y, so that we have two degrees of freedom, which enables us to move instances in directions parallel to the x and Y axes. But we could have fewer. In the following procedure, drawSquare(), Y is made a constant within the procedure, and the only position parameter is X:

[sample code on the right side]

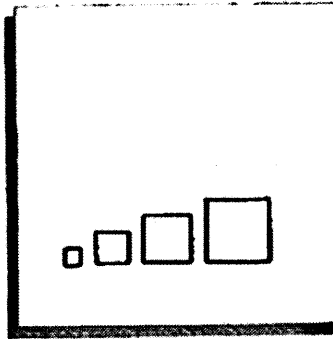
In other words, a certain Y coordinate becomes an essential property of the element, and we have fewer design variables. We can invoke this procedure to generate compositions of aligned squares. The following statements, for example, generate the composition shown in figure 8-20:

```
drawSquare (150, 50)
drawSquare (250,100)
drawSquare (400,150)
drawSquare (600,200)
```

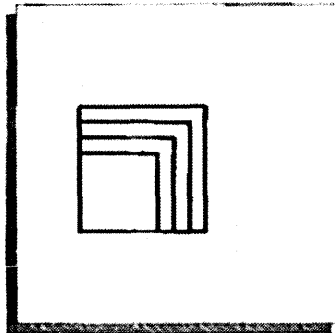
The next version of a procedure to draw a square has no position parameters:

[sample code on the right side]

This procedure draws VaryingSquare() starting at a certain point, and Length is the only design variable. The following statements use it to generate the composition shown in figure 8-21:



8-20. A composition of aligned squares.



8-21. A composition in which squares are fixed at their bottom-left corner, but side lengths vary.

CODE

```

import rhinoscriptsyntax as rs
import math

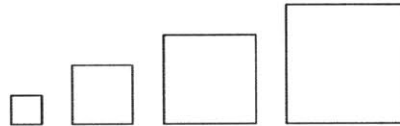
def drawSquare(x, length):
    y = 200
    # calculate values for x2 and y2
    x2 = x + length
    y2 = y + length
    # set the porints of four corners
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    # draw line
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawAlignedSquare():
    drawSquare(150, 50)
    drawSquare(250, 100)
    drawSquare(400, 150)
    drawSquare(600, 200)

drawAlignedSquare()

```

RESULT



```

import rhinoscriptsyntax as rs

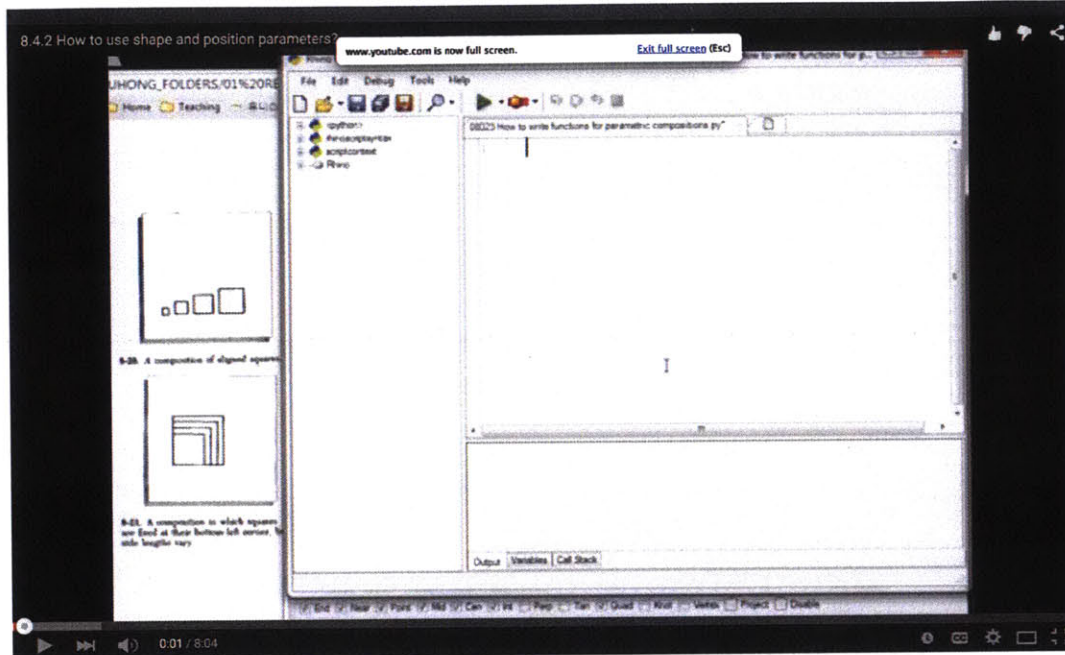
def drawSquare(length):
    x = 200
    y = 300
    # calculate values for x2 and y2
    x2 = x + length
    y2 = y + length
    # set the porints of four corners
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    # draw line
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawVaryingSquare():
    drawSquare(250)
    drawSquare(300)
    drawSquare(350)
    drawSquare(400)

drawVaryingSquare()

```

Synthetic Tutor



Module 59: 8.4 Position 2

8. GRAPHIC VOCABULARIES

8.4 SHAPE AND POSITION PARAMETERS

8.4.2 CHOOSING POSITION PARAMETERS (continued)

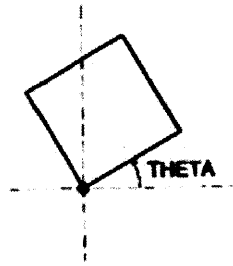
Another possibility is to define not only X and Y as position parameters, but also an angle Theta of rotation from the X axis (fig. 8-22). It is thus no longer an essential property that a square has sides parallel to the coordinate axes, and we have an additional position variable to work with in design. The code of the procedure now becomes more complicated and involves use of some trigonometry:

[sample code on the right side]

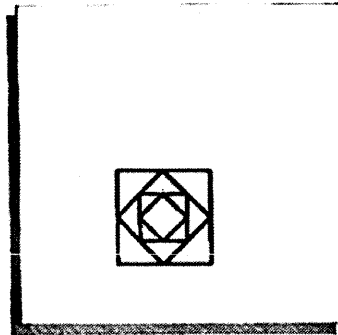
(Do not be too intimidated by this. Later we shall explore much more convenient ways to handle rotations.)

The following code assigns values to all three position parameters and the shape parameter Length, for each instance, and generates the composition 8-22. The introduction of Theta as a shown in figure 8-23: third position parameter.

The effect of introducing additional position parameters is much like that of introducing additional drawing instruments. If you have only a parallel rule and a ninety-degree set square, you can only construct squares parallel to the coordinate axes. But, if you introduce an adjustable drafting triangle, or a drafting machine, you can construct squares at any angle.



8-22. The introduction of Theta as a third position parameter.



8-23. A composition of rotated squares.

CODE

```

import rhinoscriptsyntax as rs
import math

def drawSquare(x, y, length, theta):
    k = 0.01745
    # calculate values for x2 and y2
    x2 = x + length * math.cos(theta * k)
    y2 = y + length * math.sin(theta * k)

    theta = theta + 45
    diagonal = math.sqrt(length * length * 2)
    x3 = x + diagonal * math.cos(theta * k)
    y3 = y + diagonal * math.sin(theta * k)

    theta = theta + 45
    x4 = x + length * math.cos(theta * k)
    y4 = y + length * math.sin(theta * k)

    # set the points of four corners
    pt1 = [x, y, 0]
    pt2 = [x2, y2, 0]
    pt3 = [x3, y3, 0]
    pt4 = [x4, y4, 0]

    # draw line
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawRotatedSquares():
    # draw an outer square
    x1 = 200
    y1 = 200
    length1 = 500
    drawSquare(x1, y1, length1, 0)

    # draw rotated second square
    x2 = x1 + (length1 / 2)
    length2 = round(length1/math.sqrt(2.0))
    drawSquare(x2, y1, length2, 45)

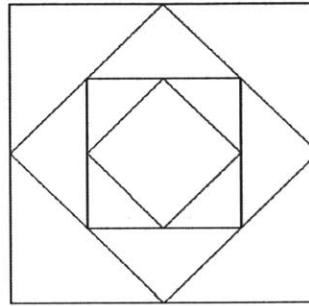
    # draw an outer square
    x3 = x1 + (length1 / 4)
    y3 = y1 + (length1 / 4)
    length3 = round(length2/math.sqrt(2.0))
    drawSquare(x3, y3, length3, 0)

    # draw an outer square
    length4 = round(length3/math.sqrt(2.0))
    drawSquare(x2, y3, length4, 45)

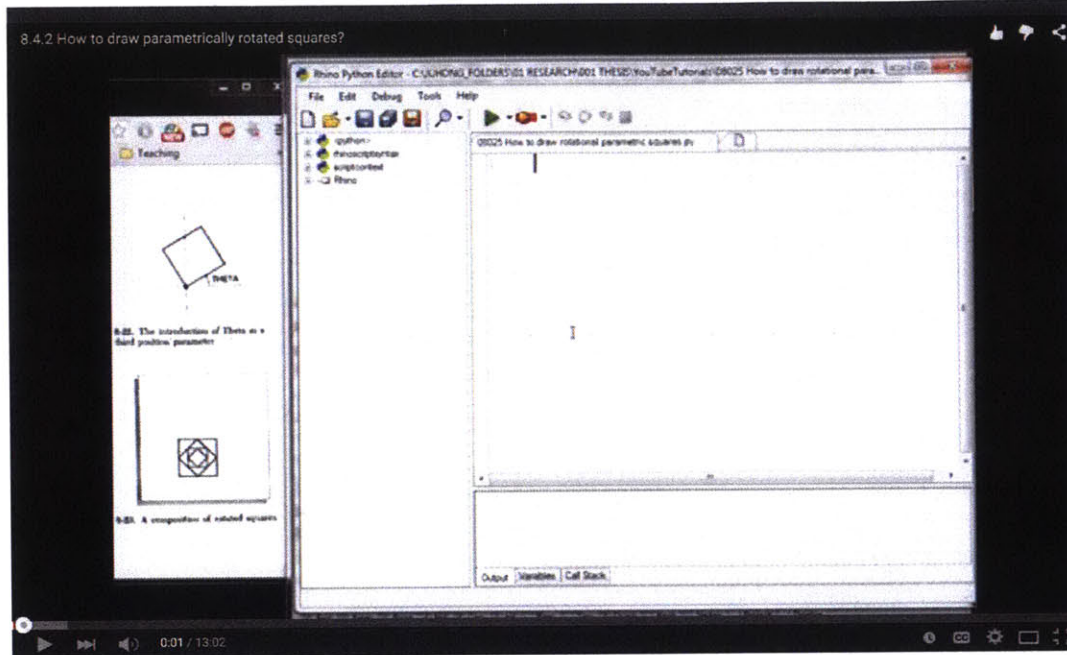
drawRotatedSquares()

```

RESULT



Synthetic Tutor



Module 60: Exercise 3

7. COORDINATE CALCULATIONS

7.9 EXERCISES

1. Python does not have a function to calculate the reciprocal of an integer. Write one.

Please upload your python file: No file chosen

2. Python does not have an exponentiation function. Write one.

Please upload your python file: No file chosen

3. Write a function that calculates the area of a circle of **specified radius**.

Please upload your python file: No file chosen

4. Write a function that calculates the perimeter of a circle of **specified radius**.

Please upload your python file: No file chosen

5. Write a function to return the mean of two real numbers.

Please upload your python file: No file chosen

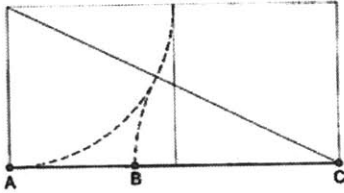
6. A line is divided into the so-called golden ratio when $AB/BC = BC/AC$ (fig. 7-11a). Figure 7-11b shows a geometric construction for dividing a line in this ratio. Write a function that accepts the coordinates of A and C, and returns the X coordinate of B.

Write a second function to return the Y coordinate.

Synthetic Tutor



a. The ratio.



b. Its geometric construction.

7-11. A line divided in the golden ratio.

Please upload your python file: No file chosen

7. Consider the following function:

```
def change_Y(Y):  
    result = (1024 - Y) / 2  
    return result
```

What would be the effect of using Change_Y to change every Y coordinate of a drawing in a 1,024 by 1,024 screen coordinate system?

Please upload your python file: No file chosen

Module 61: 8.4 Parameter 1

8. GRAPHIC VOCABULARIES

8.4 SHAPE AND POSITION PARAMETERS

8.4.3 CHOOSING SHAPE PARAMETERS

We can also establish many or few shape parameters, with corresponding degrees of freedom to vary the shapes of instances. By definition, a square has only one shape parameter, which we have taken to be Length. We might even eliminate this, as in the following procedure that draws squares of fixed size at different positions parallel to the coordinate axes:

[sample code on the right side: `drawSquare()`]

Our only design decisions now are about values of the position parameters X and Y for each instance. The composition shown in fig. 8-24 was generated by the following statements, which assign values to these parameters for each instance:

[sample code on the right side: `drawSquareWithPositions()`]

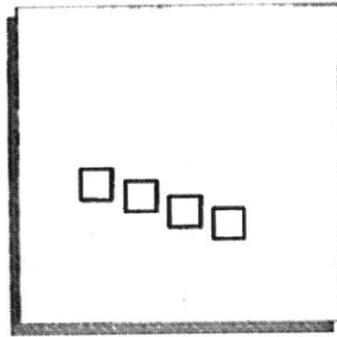
A rectangle, by definition, has two shape parameters, which we usually think of as Length and Width (fig. 8-25). We can modify our Square procedure to become a Rectangle procedure:

[sample code on the right side: `drawRectangle()`]

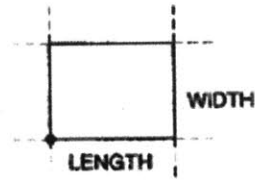
Note that this represents a generalization of the idea of a square. Rectangle can still produce squares if we choose to relate Length and Width by assigning each the same value, but it can also produce rectangles that are not squares. In other words, the square is a specialized subtype (with a more restrictively defined essence) of the rectangle. The following statements, which invoke Rectangle, generate the composition shown in figure 8-26.

[sample code on the right side: `composeRectangles()`]

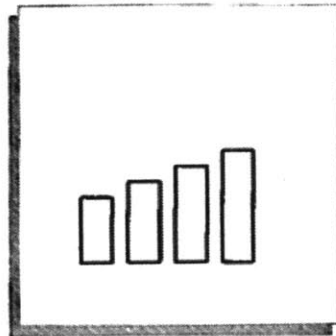
A rectangle with sides parallel to the coordinate axes can also be parameterized in terms of the coordinates of its diagonally opposite corners (fig. 8-27).



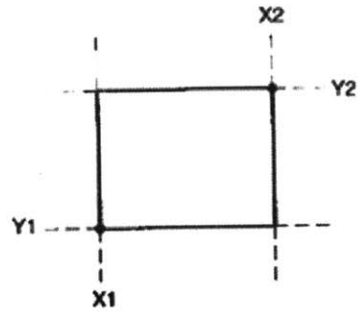
8-24. A composition in which the squares are of fixed size but vary in position.



8-25. A rectangle with sides parallel to the coordinate axes. Length and Width are the two shape parameters.



8-26. A simple composition of rectangles.



8-27. A rectangle parameterized by the coordinates of its diagonally opposite corners.

CODE

```

import rhinoscriptsyntax as rs

def drawSquare(x, y):
    length = 100

    # calculate values for x2 and y2
    x2 = x + length
    y2 = y + length

    # set the porints of four corners
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]

    # draw lines
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawSquareWithPositons():
    drawSquare(200,400)
    drawSquare(340,360)
    drawSquare(480,320)
    drawSquare(620,280)

drawSquareWithPositons()

import rhinoscriptsyntax as rs

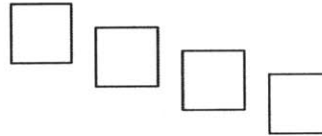
def drawRectangle(x, y, length, width):
    x2 = x + length
    y2 = y + width
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def composeRectangles():
    drawRectangle(200,200,100,200)
    drawRectangle(350,200,100,250)
    drawRectangle(500,200,100,300)
    drawRectangle(650,200,100,350)

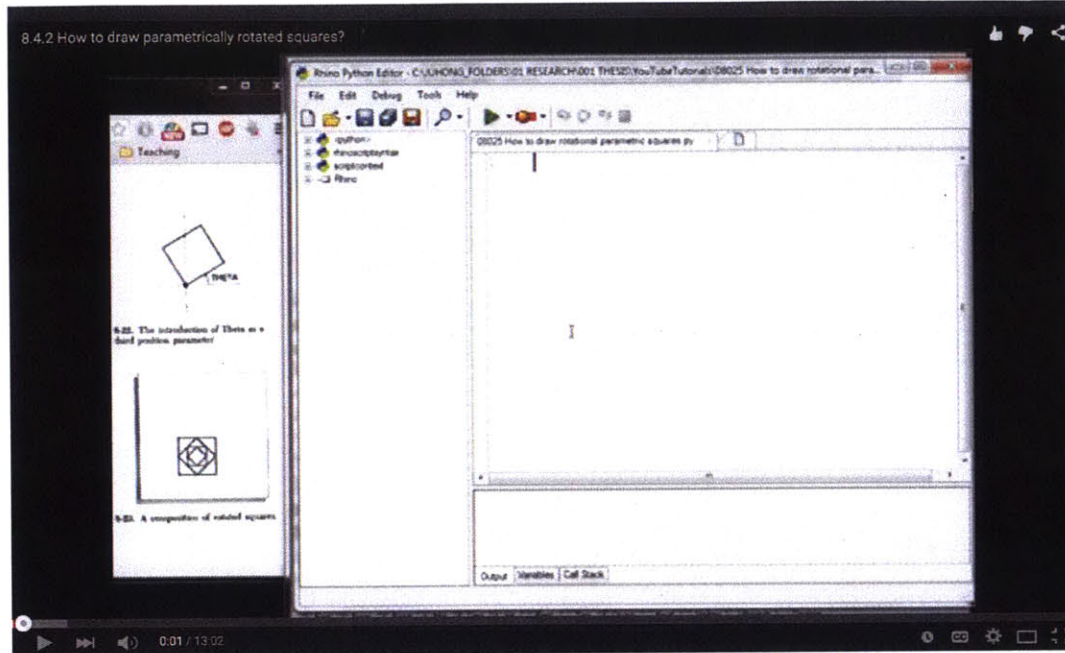
composeRectangles()

```

RESULT



Synthetic Tutor



Module 62: 8.4 Parameter 2

8. GRAPHIC VOCABULARIES

8.4 SHAPE AND POSITION PARAMETERS

8.4.3 CHOOSING SHAPE PARAMETERS (continued)

Introduction of the additional shape parameter Shear angle (fig. 8-28) yields the type parallelogram. The following procedure draws parallelograms:

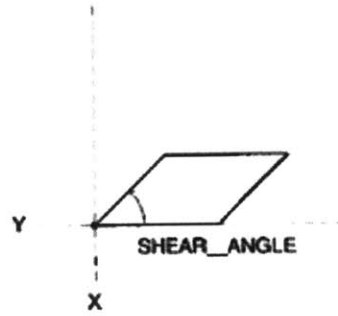
[sample code on the right side]

This represents another generalization. Parallelogram will draw rectangles when Shear is set to 90 degrees, and it will draw squares when Shear is 90 degrees and Length has the same value as Width.

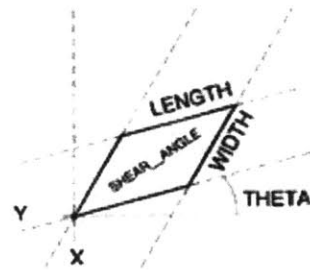
Here, for the ambitious, is a procedure with X, Y, Theta, Length, Width, and Shear_angle (fig. 8-29) as parameters:

[sample code on the right side]

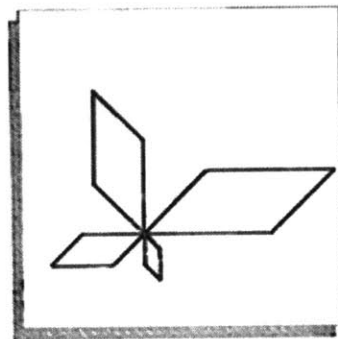
This involves some fairly complicated trigonometry. The following statements invoke it to generate the composition of rotated parallelograms shown in figure 8-30:



8-28. A parallelogram with two sides parallel to the X axis.



8-29. A rotated parallelogram.



8-30. A composition of rotated parallelograms.

CODE

```

import rhinoscriptsyntax as rs
import math

def drawParallelogram(x, y, length, width, shear_angle):
    radiance = 0.01745
    shear_angle = shear_angle * radiance
    x2 = x + length
    x4 = x + width * math.cos(shear_angle)
    x3 = x4 + length
    y2 = y + width * math.sin(shear_angle)

    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x3,y2,0]
    pt4 = [x4,y2,0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

drawParallelogram(10, 10, 100, 100, 45)

import rhinoscriptsyntax as rs
import math

def drawParallelogram(x, y, length, width, shear_angle, theta):
    radiance = 0.01745
    shear_angle = shear_angle * radiance
    theta = theta * radiance

    x1 = width * math.cos(shear_angle)
    y1 = width * math.sin(shear_angle)
    length_x1 = length + x1
    diagonal = math.sqrt(y1 * y1 + length_x1 * length_x1)
    angle = math.atan(y1 / (length + x1))

    x1 = x + length * math.cos(theta)
    y1 = y + length * math.sin(theta)
    x2 = x + diagonal * math.cos(theta + angle)
    y2 = y + diagonal * math.sin(theta + angle)
    x3 = x + width * math.cos(shear_angle + theta)
    y3 = y + width * math.sin(shear_angle + theta)

    pt1 = [x,y,0]
    pt2 = [x1,y1,0]
    pt3 = [x2,y2,0]
    pt4 = [x3,y3,0]

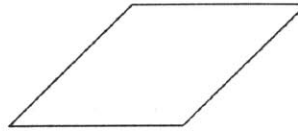
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawRotatedParallelogram():
    drawParallelogram(400,300,400,200,45,0)
    drawParallelogram(400,300,300,150,45,90)
    drawParallelogram(400,300,200,100,45,180)
    drawParallelogram(400,300,100,50,45,270)

drawRotatedParallelogram()

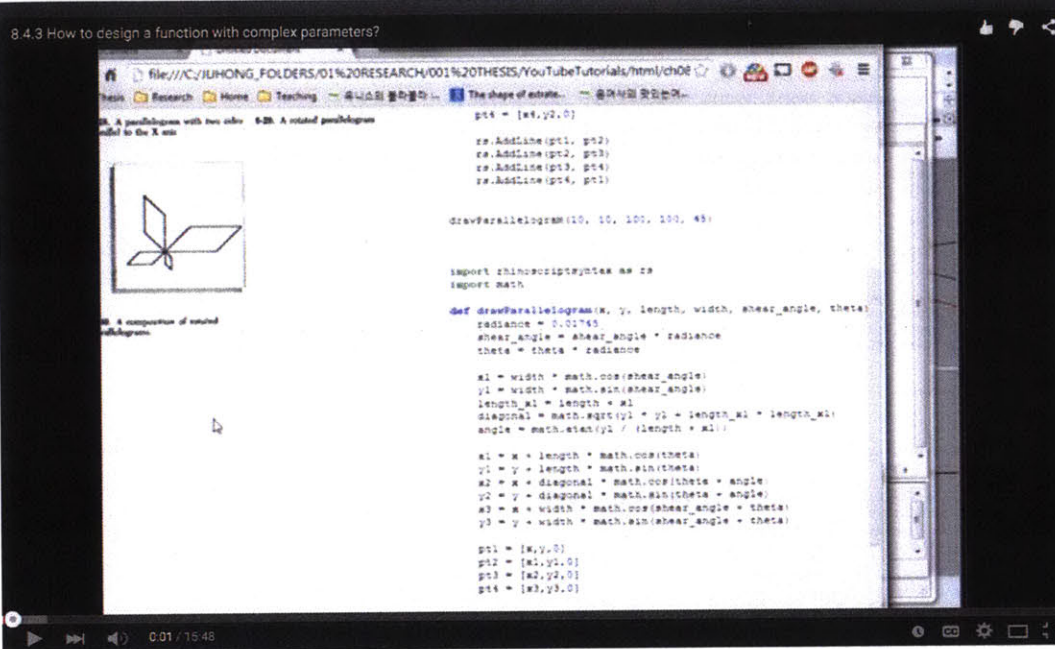
```

RESULT



Synthetic Tutor

8.4.3 How to design a function with complex parameters?



```
pt4 = [x4,y2,0]

rs.AddLine(pt1, pt2)
rs.AddLine(pt2, pt3)
rs.AddLine(pt3, pt4)
rs.AddLine(pt4, pt1)

DrawParallelogram(10, 10, 100, 100, 45)

import rhinoscriptsyntax as rs
import math

def drawParallelogram(x, y, length, width, shear_angle, theta):
    radianse = 0.01745
    shear_angle = shear_angle * radianse
    theta = theta * radianse

    x1 = width * math.cos(shear_angle)
    y1 = width * math.sin(shear_angle)
    length_x1 = length * x1
    diagonal = math.sqrt(y1 * y1 + length_x1 * length_x1)
    angle = math.atan(y1 / (length * x1))

    x1 = x + length * math.cos(theta)
    y1 = y + length * math.sin(theta)
    x2 = x + diagonal * math.cos(theta + angle)
    y2 = y + diagonal * math.sin(theta + angle)
    x3 = x + width * math.cos(shear_angle + theta)
    y3 = y + width * math.sin(shear_angle + theta)

    pt1 = [x,y,0]
    pt2 = [x1,y1,0]
    pt3 = [x2,y2,0]
    pt4 = [x3,y3,0]
```

Module 63: 8.4 Parameter 3

8. GRAPHIC VOCABULARIES

8.4 SHAPE AND POSITION PARAMETERS

8.4.4 THE MAXIMUM POSSIBLE NUMBER OF PARAMETERS

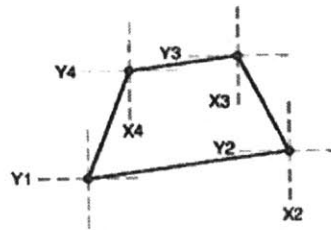
We can generalize the idea of a four-sided figure to the ultimate by allowing the coordinates of each vertex to be set independently (fig. 8-31). Thus we have eight parameters, giving us eight degrees of freedom. The following program generates a composition of such figures (see fig. 8-32):

[sample code on the right side]

Notice that the procedure `drawFourside()` does not necessarily generate polygons. It can also generate bow tie-figures (fig. 8-33).

In general, the maximum number of parameters for a figure composed of n vectors is $4n$ (fig. 8-34a). If the vectors are connected head to tail (Klee, active line, limited in its movement by fixed points) the maximum is $2(n + 1)$ (fig. 8-34b). If they are connected to form an n -sided figure, the maximum is $2n$ (fig. 8-34c). Our task in parameterizing a procedure to draw a figure composed of n vectors, therefore, is to choose some appropriate number of degrees of freedom between 0 and $4n$, then to work out a convenient way to define the parameters and express the code of the procedure in terms of them.

Figures with few degrees of freedom are subtypes of figures with more. We have seen, for instance, that the square is a subtype of the rectangle, and the rectangle of the parallelogram. Thus hierarchies of types and subtypes emerge (fig. 8-35).



8-31. A quadrilateral.



a. Composition of four vectors.



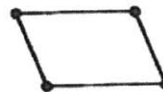
b. Four-sided figure.



c. Four-sided polygon.



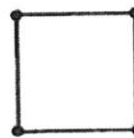
d. Trapezoid.



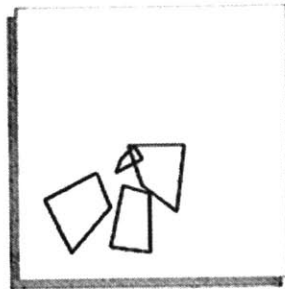
e. Parallelogram.



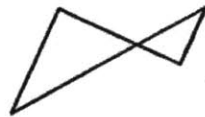
f. Rectangle.



g. Square

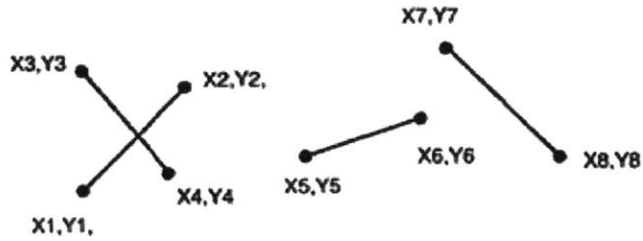


8-32. A composition of quadrilaterals.

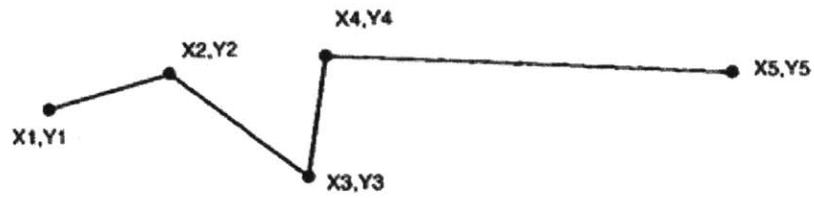


8-33. A "bow-tie" figure.

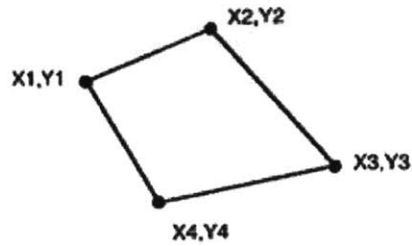
8-35. A typical hierarchy of types and subtypes.



a. A figure composed of 4 vectors can have $4n = 16$ parameters.



b. A chain of 4 vectors can have $2(n + 1) = 10$ parameters.



c. A 4-sided figure can have $2n = 8$ parameters.

8-34. The maximum number of parameters for a figure composed of n vectors.

Synthetic Tutor

CODE

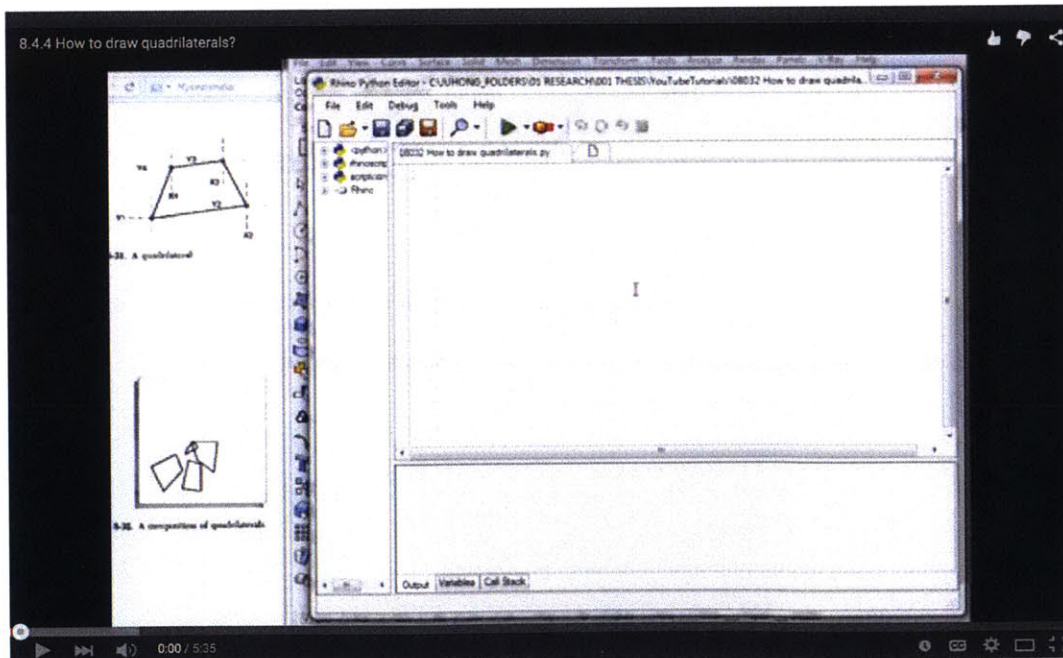
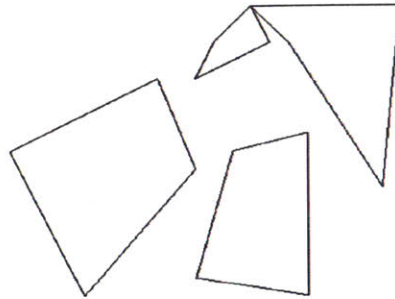
```
import rhinoscriptsyntax as rs
import math

def drawFourside(x1, y1, x2, y2, x3, y3, x4, y4):
    pt1 = [x1,y1,0]
    pt2 = [x2,y2,0]
    pt3 = [x3,y3,0]
    pt4 = [x4,y4,0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawCompositionQuadrilaterals():
    drawFourside(200,100,350,275,300,400,100,300)
    drawFourside(350,125,500,100,500,325,400,300)
    drawFourside(600,250,625,500,425,500,475,450)
    drawFourside(350,400,450,450,425,500,375,450)

drawCompositionQuadrilaterals()
```

RESULT



Module 64: 8.5 Proportion8.
GRAPHIC VOCABULARIES

8.5 RULES OF PROPORTION

Where there are n vectors and we choose d degrees of freedom for a figure, $4n-d$ coordinate values must be generated within the procedure. This can be done either by setting coordinates to constants, or by making coordinates dependent on the parameters. Such dependencies often express rules of proportion. For example, this procedure incorporates the rule:

$$\text{Width} = \text{Length} / 2$$

and generates rectangles of proportion 2:1.

[sample code on the right side]

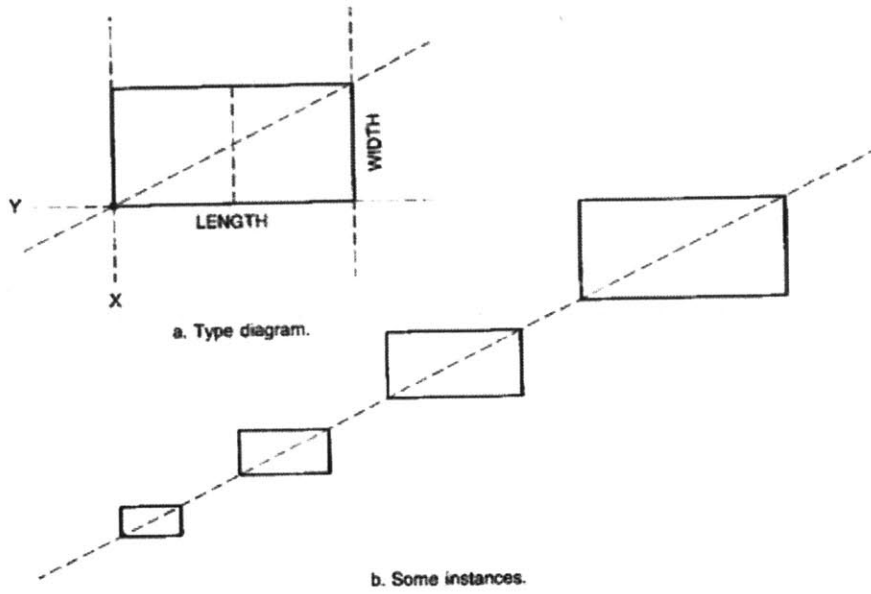
The next procedure incorporates the rule:

$$\text{Width} = (3 * \text{Length}) / 5$$

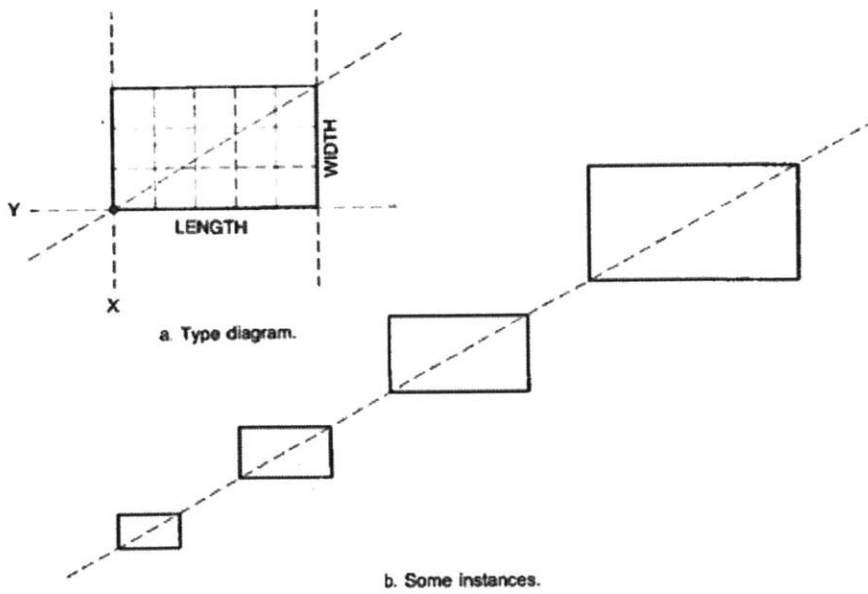
and generates rectangles of proportion 5:3.

[sample code on the right side]

Both of these procedures have the same parameters X , Y , and Length , but the different functions relating Length and Width make them different subtypes of the type Rectangle .



8-36. A rectangle of proportion 2:1.



8-37. A rectangle of proportion 5:3.

CODE

```
import rhinoscriptsyntax as rs
import math

def drawRectangleTwoToOne(x, y, length):
    # calculate width as function of length
    width = length / 2

    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width

    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

drawRectangleTwoToOne(100,100,100)

import rhinoscriptsyntax as rs
import math

def drawRectangleFiveToThree(x, y, length):
    # calculate width as function of length
    width = (3 * length) / 5

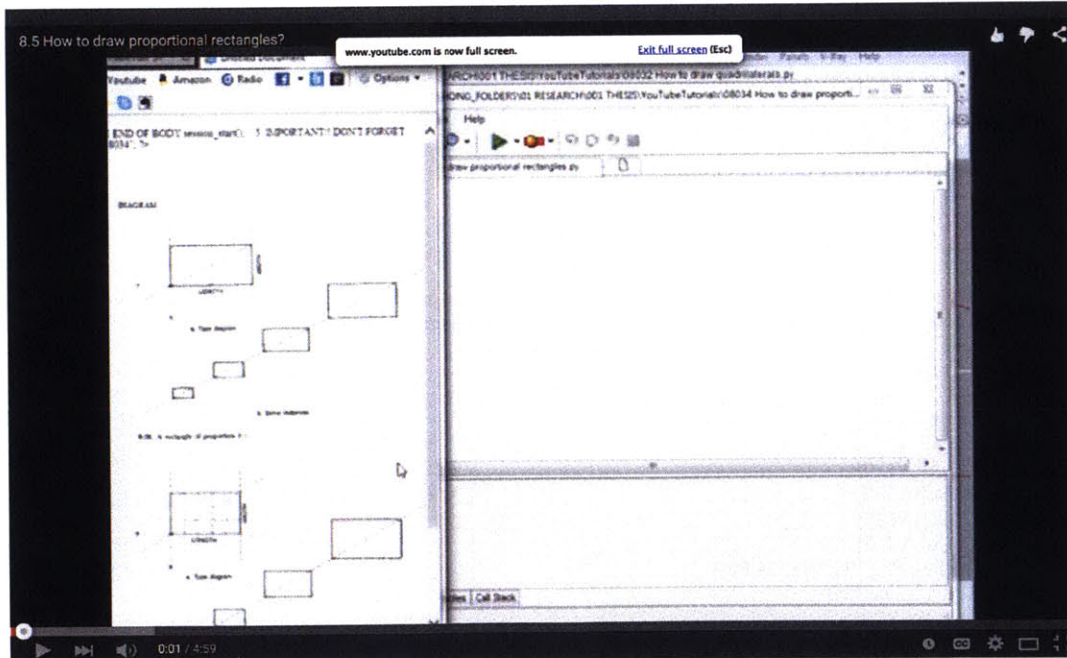
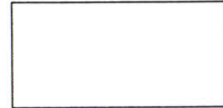
    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width

    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

drawRectangleFiveToThree(100,100,100)
```

RESULT



Module 65: 8.6 Vectors

8. GRAPHIC VOCABULARIES

8.6 WAYS TO CONNECT VECTORS

Types of motifs can be constructed systematically by considering all the different ways to make a connected figure out of n vectors (fig. 8-38). Two vectors can be connected end to end. There are three ways to connect three vectors. With four vectors the possibilities expand. There is, as one might expect, a combinatorial explosion as the number of vectors grows (see Harary 1969). The essential point, though, is that the number of possibilities for a given number of vectors is finite, and you can systematically enumerate all of them. If vectors are the atoms, these connected figures are the molecules of line drawings.

The pattern of connection of vectors in a graphic element is expressed in the generating procedure by the sequencing of points (Pts is a nested list of four points and each point is a list of three coordinates). Let us consider, for example, the shapes composed of three vectors. Code for a triangle (fig. 8-39a) may be expressed:

```
import rhinoscriptsyntax as rs

Pt0 = [x1, y1, 0]
Pt1 = [x2, y2, 0]
Pt2 = [x3, y3, 0]
Pt3 = [x1, y1, 0]
Pts = [Pt0, Pt1, Pt2, Pt3]
rs.AddPolyline(Pts)
```

Note that six variables are used to save x, y coordinates here ($x1, y1, x2, y2, x3,$ and $y3$), indicating that there are six potential degrees of freedom. Code for a chain of three vectors (fig. 8-39b), which has eight potential degrees of freedom, may be expressed:

```
import rhinoscriptsyntax as rs

Pt0 = [x1, y1, 0]
Pt1 = [x2, y2, 0]
Pt2 = [x3, y3, 0]
Pt3 = [x4, y4, 0]
Pts = [Pt0, Pt1, Pt2, Pt3]
rs.AddPolyline(Pts)
```

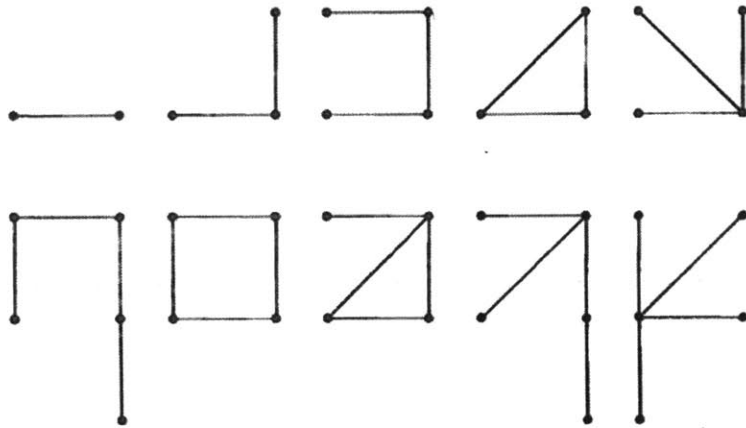
The only difference is that the last vector does not connect back to the point ($X1, Y1$), but ends at independently established coordinates ($X4, Y4$). Finally, code to draw the radial pattern (fig. 8-39c) may be expressed:

```
import rhinoscriptsyntax as rs

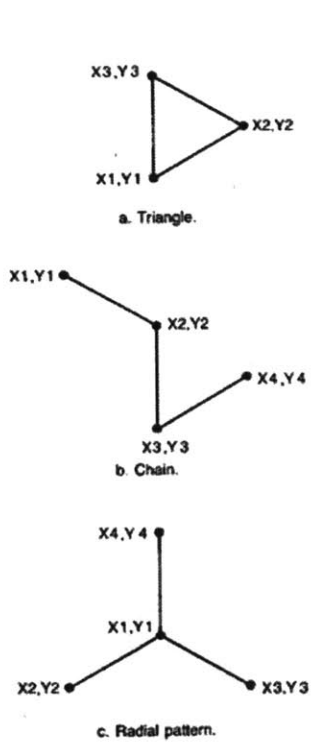
Pt0 = [x1, y1, 0]
Pt1 = [x2, y2, 0]
Pt2 = [x1, y1, 0]
Pt3 = [x3, y3, 0]
Pt4 = [x1, y1, 0]
Pt5 = [x4, y4, 0]
Pts = [Pt0, Pt1, Pt2, Pt3, Pt4, Pt5]
rs.AddPolyline(Pts)
```

Once the pattern of connectivity is established, we might want to restrict ourselves to some subtype by introducing

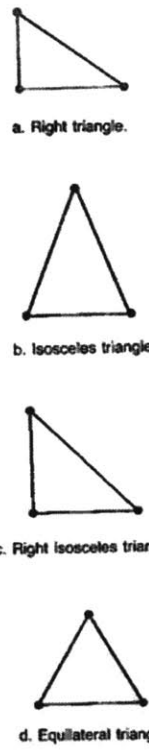
direction and ratio of length relations. In the triangular pattern, for example, we might require a pair of sides to be perpendicular, defining a right triangle (fig. 8-40a). Or we might require a pair of sides to be of equal length to produce an isosceles triangle (fig. 8-40b). If we require a pair of sides to be both perpendicular and of equal length, we obtain a right isosceles triangle (fig. 8-40c). And if we require all sides to be of equal length, we produce an equilateral triangle (fig. 8-40d).



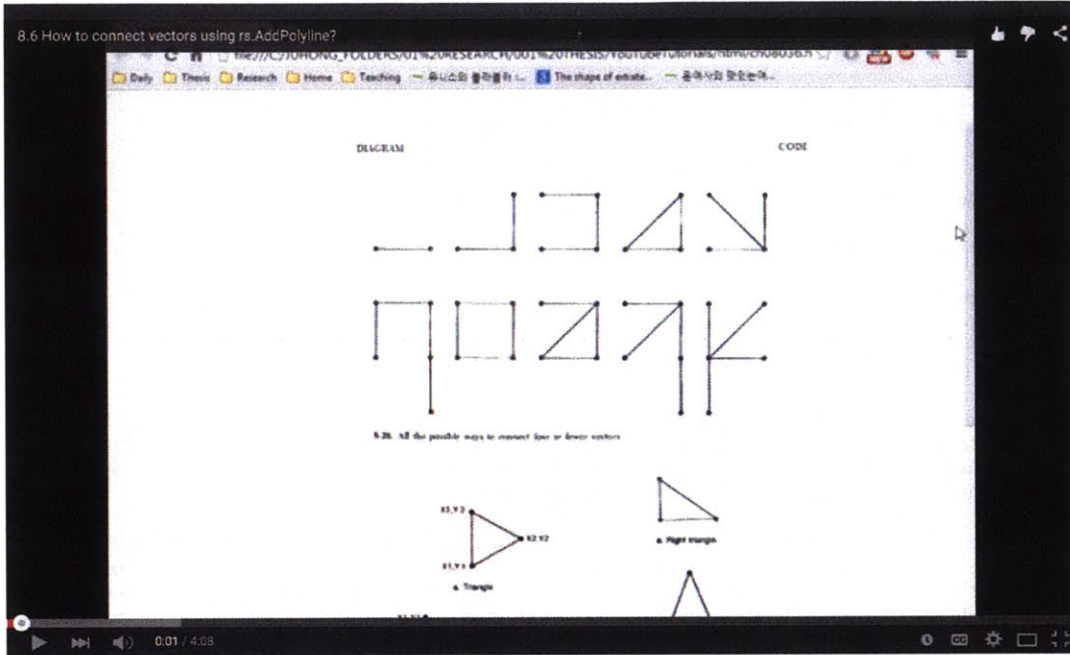
8-38. All the possible ways to connect four or fewer vectors.



8-39. Patterns of three vectors.



8-40. Subtypes of the triangle.



Module 66: 8.7 Variables

8. GRAPHIC VOCABULARIES

8.7 How Much Variation Is Needed?

How much variation should there be within a type? More precisely:

- . How many degrees of freedom?
- . How should these degrees of freedom be expressed in terms of parameters and dependencies?
- . What should be the ranges of the parameters?
- . What should be the increments of variation within these ranges?

At one extreme, a type might have only one instance. At the other extreme, for a figure of n vectors, there might be $4n$ parameters, each with a range of 1,024 values, yielding 1,024 instances. This, however, is a reduction to absurdity we are back to having n independent vectors. Unless we can see some particular advantage in composing vectors n at a time, instead of one at a time, we do not gain anything.

The trade-off, then, is that if we write procedures with few shape and position parameters, we will have few graphic variables to manipulate in generating a composition, and we will be able to generate only highly structured, disciplined drawings of a very specific kind. But if we allow too many shape and position parameters, then the method of thinking in terms of types and instances begins to lose its power.

In fact, most of the drawings that we want to generate do turn out to be structured and disciplined in identifiable ways. The point is to recognize the rules that apply, or that we want to apply, and to express these rules concisely and elegantly in Python code. Conversely, you should avoid coding in restrictions that, work against your graphic intentions. Once again, the fundamental issues here are not ones of computer technology, but of an artists or a designers stylistic choices.

Module 67: 8.8 Defining

8. GRAPHIC VOCABULARIES

8.8 DEFINING VOCABULARIES OF GRAPHIC ELEMENTS

So far we have considered examples of programs that generate compositions from one type of graphic element, such as a square, a rectangle, a trapezoid, or a triangle. But there is nothing to stop us from declaring procedures to draw several different types of graphic elements, then instantiating these to create a composition. The next program, for example, draws the composition of rectangles and triangles that is illustrated in figure 8-42.

[sample code on the right side]

A graphic vocabulary may now be defined as a set of graphic element types that can be instantiated to generate drawings. Each element in the vocabulary is defined by the declaration of a parameterized procedure, and an instance is created whenever the procedure is invoked by name with appropriate actual parameters. In our example, then, we have a vocabulary consisting of rectangle and triangle.

We have seen that the parameters to a procedure that generates a vocabulary element establish the design variables associated with that element. There may be many or few of these, depending upon whether you want many features of an instance to be -given- or whether you want to be able to vary many features from instance to instance.

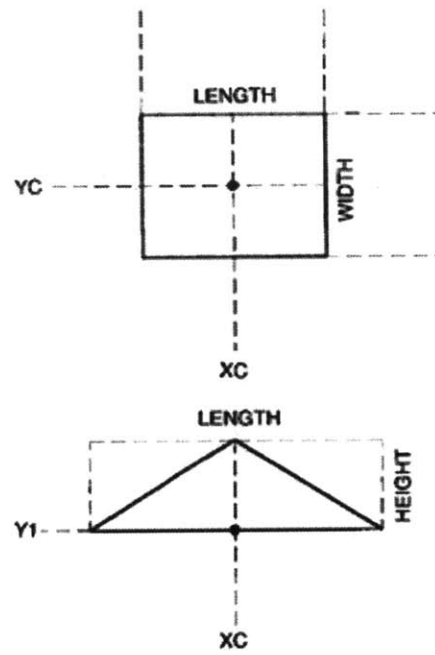
The code within a procedure that generates a vocabulary element defines the essential properties. `rsAddLine` express how vectors are connected. By setting constants and writing code that makes coordinate values dependent on the parameters, you establish essential dimensional properties. And, by declaring subranges, you make it essential to comply with certain limits.

We have also seen that you can take a systematic approach to defining a vocabulary element. First, decide how the vectors are connected. Next, decide what degrees of freedom you want in shaping and positioning instances, and develop a convenient scheme for expressing the shape and position parameters. Write expressions making coordinates in `rs.AddLine` dependent on these parameters. Finally, express any dimensioning discipline that you want to impose by means of constants and subranges.

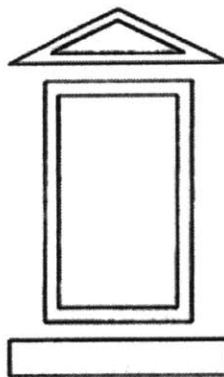
Once the vocabulary has been declared, you can specify a composition by writing commands in the main program to create instances. The composition can very easily be modified by adding or deleting such commands or changing parameter values.

There are several ways to organize the code. You might first assign or read values for all the independent variables, then calculate values for all dependent variables, and finally make a procedure after you have established all the necessary parameter values. Alternatively, you might read and calculate parameter values immediately before you need to pass them into a graphics procedure, or you might use some combination of the two alternatives.

Each approach has its advantages and disadvantages. Strict segregation of the three steps keeps clear the logical distinction between independent and dependent design variables, but makes it more difficult to trace through the text of the program how a value is derived for a particular shape or position parameter. Conversely, calculation of values as needed usually makes it easier to trace the derivation of a parameter value, but obscures the distinction between independent and dependent variables. You must decide what is most important in a given program and choose an approach accordingly.



a. Vocabulary elements.



b. Composition of instances.

8-42. A composition of rectangles and triangles.

CODE

```

import rhinoscriptsyntax as rs

def drawRectangle(xc, yc, length, width):
    x1 = xc - (length / 2)
    x2 = x1 + length
    y1 = yc - (width / 2)
    y2 = y1 + width

    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawTriangle(xc, y1, base, altitude):
    x1 = xc - (base / 2)
    x2 = x1 + base
    y2 = y1 + altitude

    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [xc, y2, 0]

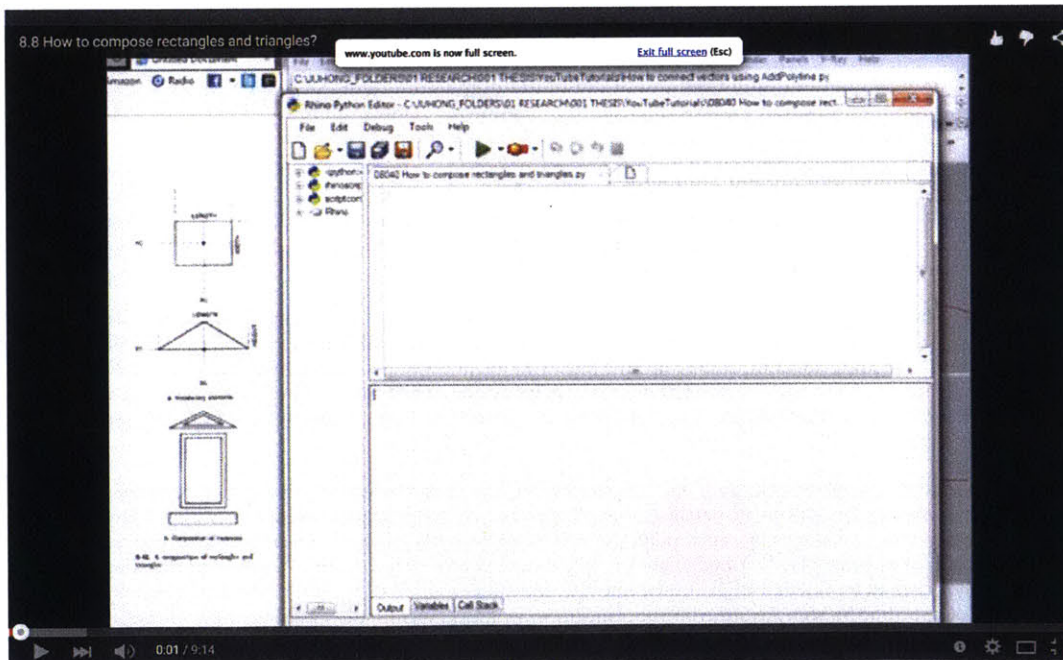
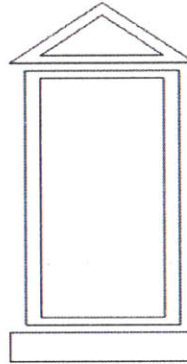
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt1)

def drawWindow():
    drawRectangle(450, 125, 300, 50)
    drawRectangle(450, 375, 200, 400)
    drawRectangle(450, 375, 250, 424)
    drawTriangle(450, 612, 200, 67)
    drawTriangle(450, 600, 300, 100)

```

```
drawWindow()
```

RESULT



Module 68: 8.8 Vocabulary

8. GRAPHIC VOCABULARIES

8.8 DEFINING VOCABULARIES OF GRAPHIC ELEMENTS

8.8.1 EXAMPLES OF GRAPHIC VOCABULARIES

A graphic vocabulary allows us to avoid writing long, tedious lists of instructions at the vector-by-vector level. It, instead, allows us to write instructions at the vocabulary element-by-vocabulary element level. We make use of our knowledge of the structure of the drawings that we want to produce in order to achieve this, and the structure of the program expresses this knowledge. Thus the program becomes shorter, more elegant, more informative, and easier to change.

If you want to write graphics programs, then, one of the first steps is to put together an appropriate library of graphics procedures. These can be used as building blocks for producing programs very quickly.

You must, of course, think carefully about the kind of graphic vocabulary that you will need. It should be appropriate to your particular graphic purposes and extensive and flexible enough to allow you to accomplish the results that you want, yet it should not be too large. Let us consider some examples.

If you want to generate compositions of simple geometric figures, you might put together the following vocabulary (fig. 8-43):

- Line
- Circle
- Arc
- Rectangle
- Square

At this point you can write procedures for all of these except Circle and Arc. We will introduce procedures for drawing circles and arcs in chapter 10.

To create lettering, you might define procedures to instantiate each element of an alphabet. You might allow only position parameters X and Y (fig. 8-44a) or you might allow size to be varied as well by introducing one shape parameter (fig. 8-44b). You might also allow compressed or extended versions by introducing a fourth parameter, allowing height and width to be varied independently (fig. 8-44c). Finally, you might allow the detailed adjustment of each character form (figure 8-44d).

Where the graphic task is to draw furniture layouts, you will need a vocabulary of furniture types: chairs, desks, tables, and so on. Or, where the task is to draw architectural elevations, you will need a vocabulary of architectural elements: doors, windows, columns, arches, and the like. You might develop different vocabularies for elevations in different architectural styles, such as classical or gothic. A landscape architect might develop a vocabulary of different types of trees. To draw people you might define a parameterized man, a parameterized woman, and a parameterized child.

When you establish a graphic vocabulary, the consideration of how instances will be put together to produce compositions will suggest appropriate parameterization schemes. An architect composing a floor plan, for example, usually wants to specify columns by center point (XC, YC) and Width (fig. 8-45a). It is usually most convenient to specify walls by start point (X1, Y1), endpoint (X2, Y2), and thickness T (fig. 8-45b). Windows in elevation are conveniently specified by Width, Height, center line XC, and sill height Sill (fig. 8-45c). And arch voussoirs might reasonably be specified by arch Radius, voussoir Thickness, voussoir Angle, and center-line angle Theta (fig. 8-45d).



a. Vector.



b. Circle.



c. Arc.



d. Rectangle.



e. Square.



f. Right triangle.

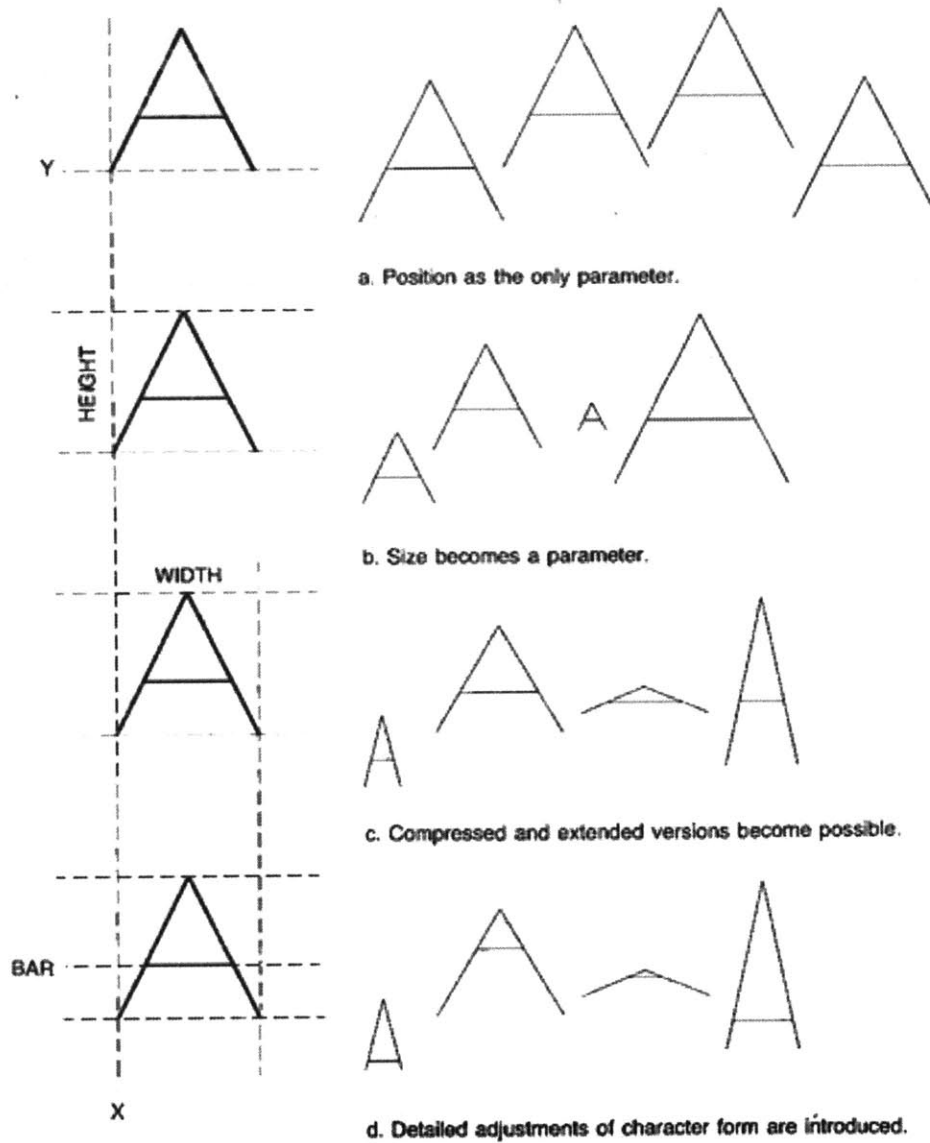


g. Equilateral triangle.

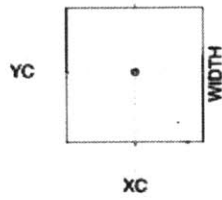


h. Triangle.

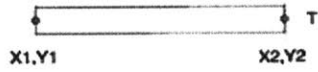
8-43. A vocabulary for generating compositions of simple geometric figures.



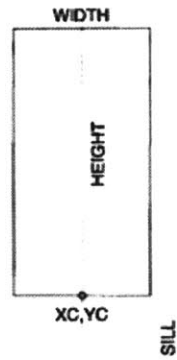
8-44. The parameterization of a character.



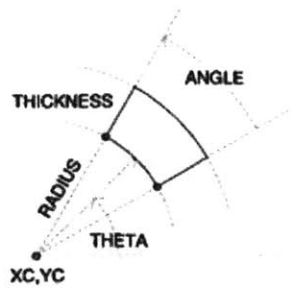
a. Column in plan.



b. Wall in plan.



c. Window in elevation.



d. Arch voussoir.

8-45. Parameterization schemes for some architectural vocabulary elements.

Module 69: 8.9 Summary

8. GRAPHIC VOCABULARIES

8.9 SUMMARY

We have introduced the fundamentally important idea of a graphic vocabulary expressed as a set of parameterized procedures. We have seen, too, how such procedures may be invoked within a program to generate a composition. The nature and extent of your graphic vocabulary and appropriate schemes for parameterization of its elements are established by carefully analyzing the ways that the drawings you intend to produce can be decomposed, the ways that these parts can be classified into types according to their commonalities and differences, and the ways that instances are varied and related in compositions.

Module 70: 9.1 Repetition

9. REPETITION

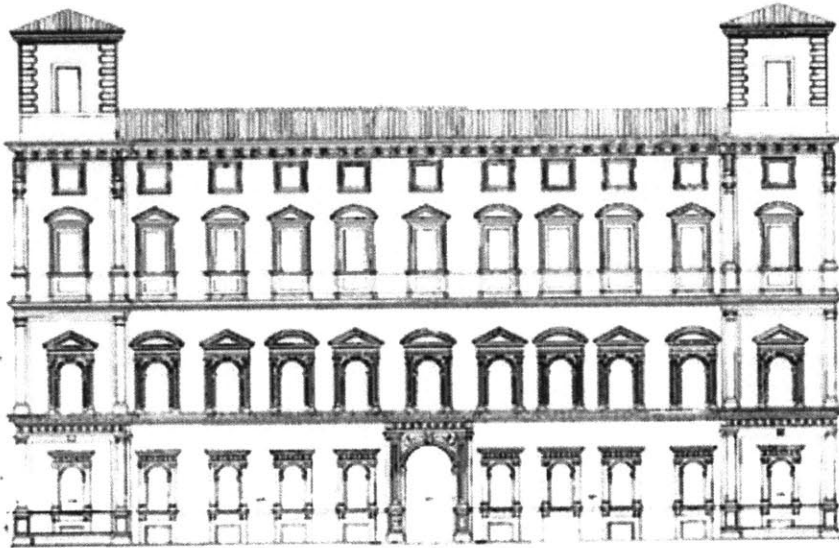
We began by looking at pictures simply as sets of vectors, and we saw how they could be generated by sequences of procedures. Then, in the last chapter, we saw that pictures may have more structure; they are often composed of instances of elements from some limited vocabulary of graphic types.

To write a Python program to draw a picture that is structured in this way, you must first declare the vocabulary elements that you intend to use. This is done by declaring a set of parameterized procedures in the declaration part of the program. Then, in the executable part of the program, you specify what vocabulary elements are to be instantiated (by giving procedure names) and the shape and location of each instance (by assigning values to the actual parameters of procedures). This is analogous to graphic design by first creating then using some vocabulary of templates or stencils, or to architectural design by first creating then using some kit of parts—an industrialized component building system, for example.

In other words, the procedure declarations establish the parts of the picture; the variable declarations to the main program establish the design variables that must be assigned values in order to create a picture; and the code (invoked procedures) actually specify how the parts are to be put together in a particular graphic composition. By varying this code, you can create different compositions from the same vocabulary.

9.1 PRINCIPLES OF REGULAR COMPOSITION

Now let us consider a particular kind of composition: the plan or elevation of a building (fig. 9-1). We can see immediately that, in this example, there is a limited vocabulary of standard columns, windows, and so on. Furthermore, the composition is not merely a random collection of instances. On the contrary, it is very highly ordered; it exhibits regular repetition of elements, consistencies of proportion, and numerous symmetries. Not all drawings are as highly ordered as this, but most do display at least some obvious regularities.



9-1. An ordered composition using a limited vocabulary (Palace in the Piazza di Sora, Rome, as depicted in Paul Letarouilly's *Edifices de Rome Moderne*)

Module 71: 9.2 Control

9. REPETITION

9.2 USE OF CONTROL STRUCTURES TO EXPRESS COMPOSITIONAL RULES

If we were to write a Python procedure to generate this drawing, we would need to express the regularities of the composition in the code that specify the composition. How might this be done?

A good way to approach this problem is to consider how you might execute the drawing by hand, with pen on a piece of tracing paper. You could begin at the bottom-left corner and draw the lines one by one, until you reach the top-right corner-but you would not. Almost certainly (especially after reading the last chapter) you would begin by making templates for repeating vocabulary elements, such as windows and columns, so that these could be instantiated rapidly by tracing. Then you might recognize that whole bays repeat. This suggests first using the templates to create a bay, then slipping the bay along and tracing it repeatedly to produce the whole elevation. Probably you would first lay out regular grid lines to control placement. There are special conditions at the end bays and at the center, so you would have to handle these a little differently. You might notice, too, that many elements are bilaterally symmetrical. For these you need only a half template, which can be flipped over to produce a mirror image.

An important general principle emerges. Analyzing the principles of the composition's organization (its vocabulary, the nesting of smaller units within larger units, the regular repetition of units, the symmetries that appear, and the special conditions that must be handled) allows you to organize your work intelligently. This would enable you to produce the drawing much more quickly and easily than would otherwise be possible. Similarly, in writing the Python code to generate a graphic composition, we should take advantage of our insights into the principles of the composition in order to minimize our effort. The use of control structures available in Python enables us to do this very effectively.

The role of control structures in a program is to express the flow of control-the sequence in which operations (for example, instantiation of vocabulary elements) are executed. So far, without explicitly discussing it, we have been making use of the most elementary control structure-textual sequence. That is, operations are executed, one after another, in the sequence in which they are written in the text of the main program. If we want to change the sequence in which operations are performed, we simply change the sequence of statements in the main program.

More precisely, Python syntax provides the possibility of writing not just single statements, but compound statements as well. A compound statement begins with `def`, and the component statements within it are executed in the same sequence as they are written. This, for example, is a compound statement:

```
def function( ):
    rs.AddLine([100,100,0], [500,100,0])
    rs.AddLine([500,100,0], [300,300,0])
    rs.AddLine([300,300,0], [100,100,0])
```

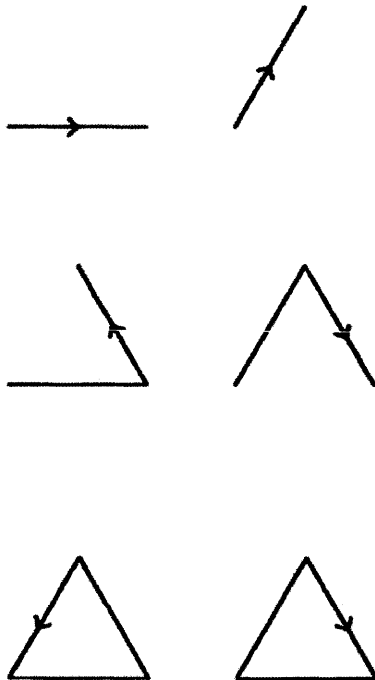
The result is a triangle, with sides drawn in counterclockwise order (fig.9-2a). If we were to flip the first and second draw statements, as follows, we would obtain the same triangle, but it would now be drawn in clockwise order (fig. 9-2b)

```
def function( ):
    rs.AddLine([100,100,0], [300,300,0])
    rs.AddLine([300,300,0], [500,100,0])
    rs.AddLine([500,100,0], [100,100,0])
```

For typographic clarity, it is customary to align the components of a compound statement and indent them, as shown in the examples above. This enables us to read a compound statement easily as a sequence.

We can, then, always encode a picture as a compound statement or invocations of procedures that instantiate vocabulary elements. Note, too, that the body of any Python program, any function, or any procedure takes the form of a single compound statement.

Python, however, provides several other useful control structures in addition to the compound statement. We shall be concerned, in the next few chapters, with structures of repetition, branching, nesting, and recursion and with using these to express principles of regular graphic composition. Intelligent use of these structures not only enables us to write very concise and elegant graphic programs to generate large and apparently complicated compositions, but also to clearly portray the regularities of a composition in the text of its generating program.



- a. Counterclockwise from left vertex.
- b. Clockwise from left vertex.

9-2. Sequences for drawing a triangle.

Module 72: 9.2 Repetition

9. REPETITION

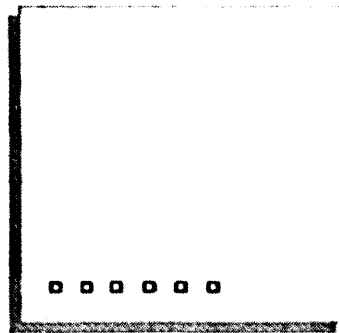
9.2 USE OF CONTROL STRUCTURES TO EXPRESS COMPOSITIONAL RULES

9.2.1 REGULAR REPETITION

Let us begin by considering the row of columns in plan, illustrated in figure 9-3. This is created by regular repetition of a vocabulary element, a specific number of times, along an axis.

To generate this composition, we might first declare a procedure to draw a square parallel to the coordinate axes, then invoke it in the main program the required number of times to generate the instances, arranged in the appropriate way. The program looks like this:

It is not difficult to identify the textual redundancy in our main program. The only thing that changes in each successive statement is the value of the x parameter. It would be much more concise, elegant, and expressive of the composition's organization simply to say that the square is to be instantiated a specified number of times, starting X at a specified value and incrementing it by a specified amount at each successive step in the process.



9-3. A row of six square columns (drawn in plan) in the screen coordinate system.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def drawSquare(x,y,length):
    # calculate width as function of length
    width = length/2

    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width

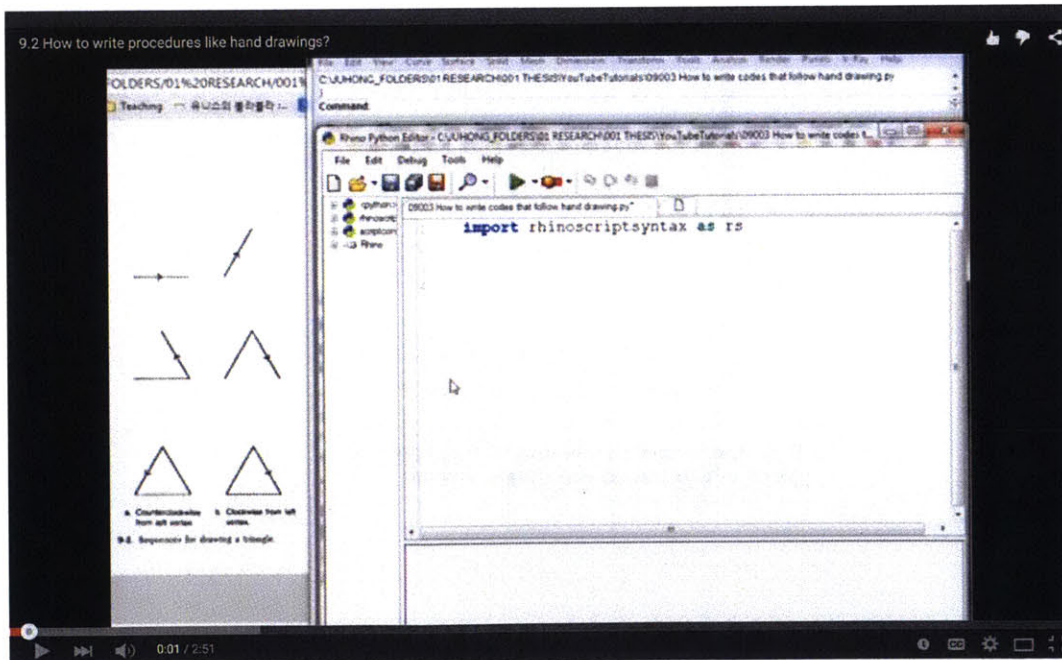
    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def drawRowSquares():
    drawSquare(100,100,30)
    drawSquare(200,100,30)
    drawSquare(300,100,30)
    drawSquare(400,100,30)
    drawSquare(500,100,30)
    drawSquare(600,100,30)

drawRowSquares()
```

RESULT



Module 73: 9.2 Loop 1

9. REPETITION

9.2 USE OF CONTROL STRUCTURES TO EXPRESS COMPOSITIONAL RULES

9.2.2 THE COUNTED LOOP

Python provides a control structure that enables us to write the program in just this way. It is called the counted loop or the for statement. Here is our main program rewritten with a for statement

Before a for statement, make a list that has a range of numbers. At the right sample code, `count = range(1, 7)` generates a list of numbers from 1 to 6 (not 7). Accordingly, `count` becomes a list, `[1,2,3,4,5,6]`. The for statement begins with the reserved word - `for`. This is followed by the name of the control variable, in this case - `i`. Next comes - `in` (checking operator) - with a variable of a list (count in this sample code), checking whether a value is in the list (in the sample code, `count`). The for statement is ending with colon (`:`) such as a function. In this case, we begin counting at 1 and end with counting at 6. It all means - for `i` given an initial value of 1 and incrementing to 6, do the following compound statement. Notice that the compound statements are indented. Everything within the indented block is within the loop; everything else in the program is outside the loop.

The loop is controlled by the initial and final values specified for the control variable. For our purposes here, we shall assume that the control variable, the initial value, and the final value are always of type integer. (The control variable may, more generally, be of any ordinal data type, but we shall not make use of ordinal data types other than integers in this context). It usually makes sense to begin a count at 1, so that the for statement looks something like this

```
for i in range(1, 7):
```

But we could begin the count at 0

```
for i in range(0, 6):
```

We could even begin at a negative integer

```
for i in range(-2, 3):
```

It is possible and often necessary to use variables like this

```
for i in range( Initial, Final+1 ):
```

In this case, of course, `Initial` and `Final` must be declared of type integer, and they must be assigned values before the for loop is processed. It is even possible to use integer-valued expressions and functions in the following manner

```
initial = A / B
final = math.sin( C ) + 1
for i in range( initial, final ):
```

This can become very confusing, however, and we do not recommend it. Where such expressions are used, they are evaluated once when the loop commences.

Here is yet another variant

```
for i in range( 10, 1, -1 ):
```

Synthetic Tutor

In this case the count is down to a lower final value from a higher initial value. The usefulness of this will become apparent when we consider functions of the control variable.

What happens if, in counting to a value, the initial value is greater than the final value, or in counting down to a value, the initial value is less than the final value?

Notice that the variable `X` in our example program is assigned an initial value outside the loop. It is then incremented the required amount by the following statement within the loop

```
X = X + 100
```

The initial value of `X` specifies where the first instance of square is to be placed, and the assignment within the loop controls the placement of each successive instance relative to its predecessor.

You should study figure 9-4 carefully. It traces the execution of this loop by showing, at each iteration, the values of the control variable - `i` - and the position parameter `X`, and the state of the drawing.

In order to avoid confusion, you must pay careful attention to the values of variables such as `i` and `X` on exit from the loop. The convention followed by Python is that the value of a control variable, such as `i`, is undefined on exit from a `for` statement. This means that you cannot use the variable again before resetting its value. The value of `X` on exit is the last value that was assigned to it. You may use this value in subsequent parts of the program, or you may choose to reset it to another value.

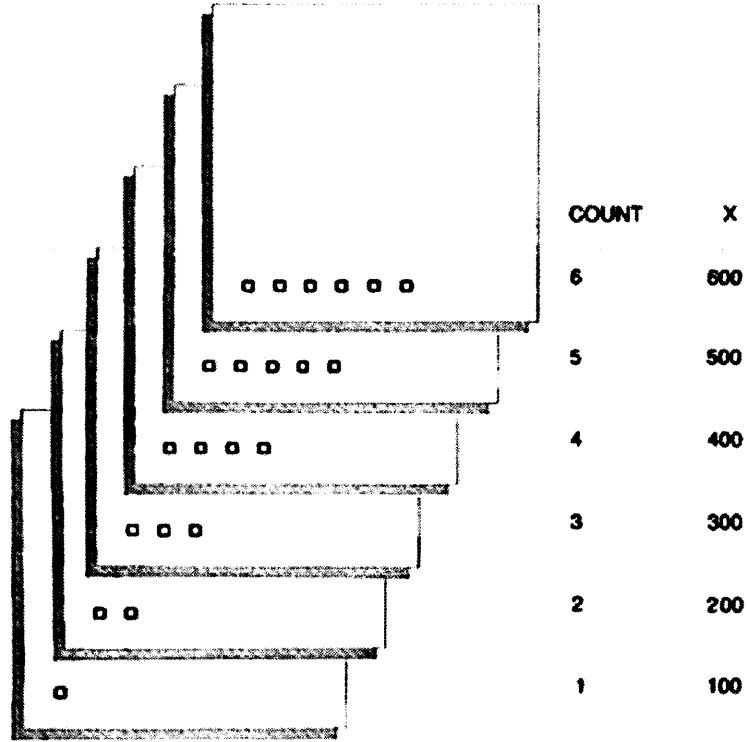
Another rule that must be followed to avoid confusion is that the value of a control variable, such as `i`, cannot be changed by a statement within a counted loop; the computer would lose track of where it was in the count. You can, however, use the value of the control variable within a counted loop, for example, in something like

```
DISTANCE = i * 100
```

Finally, for loops that include only one statement may be written in simplified form

```
for i in range(1, 10):  
    print i
```

The inner block must be indented.



9-1. States of the drawing, with values of Count and values of X at each iteration.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def drawSquare(x,y,length):
    # calculate width as function of length
    width = length/2

    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width

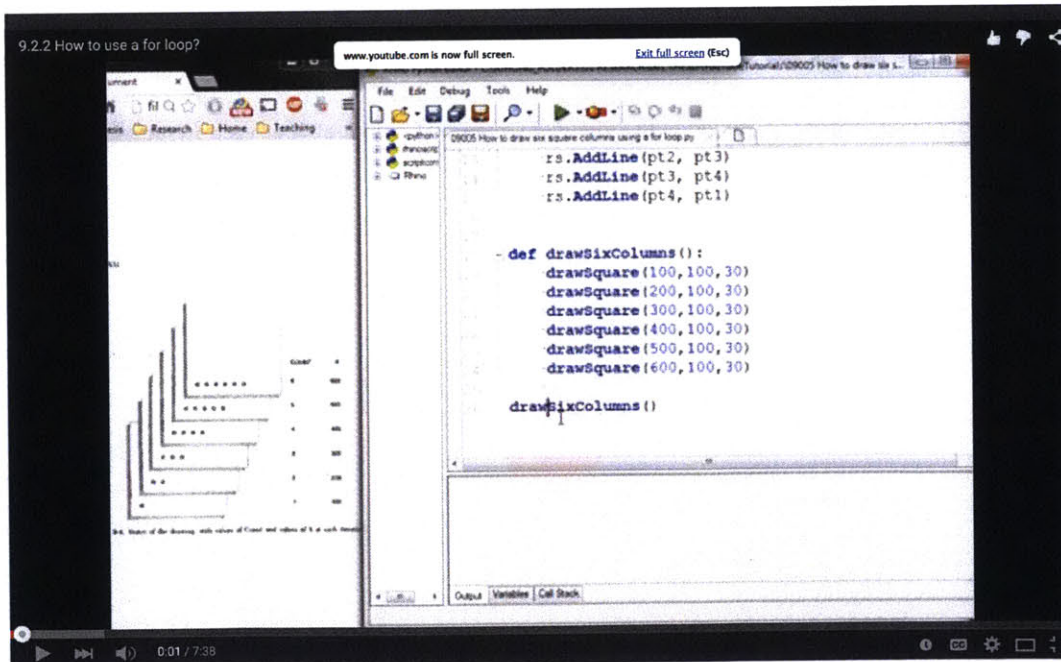
    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def drawRowSquares():
    x = 100
    count = range(6)
    for i in count:
        drawSquare(x,100,30)
        x = x + 100

drawRowSquares()
```

RESULT



Module 74: 9.2 Loop 2

9. REPETITION

9.2 USE OF CONTROL STRUCTURES TO EXPRESS COMPOSITIONAL RULES

9.2.4 THE WHILE STATEMENT AND CONTROL EXPRESSIONS

The use of a control variable to keep count of repetitions, as in a for statement, is not the only way to control a loop. Python provides another possibility: the while statement.

Here is an example of a loop (to draw a row of squares) that uses the while statement:

[sample codes on the right side]

The while statement consists of the reserved word while, followed by an expression, then ending with colon, and finally the statement that is to be executed repeatedly (the use of parentheses is optional).

Here it is the control expression `Total_X <= 1,023` that controls the loop: This control expression is constructed using Python's Boolean operators:

```
and : conjunction
or  : disjunction
not : negation
==  : equal (two equals)
<> : unequal
<  : less
>  : greater
<= : less or equal
>= : greater or equal
```

The syntax of the expression is the normal syntax used in Boolean logic. Here are some examples:

```
X == Y
X <> Y
X < Y
X >= Y
(X <= Y) OR (A > B)
```

Integer, Real, Boolean, and character variables may all appear in these statements. However, you must be careful about type. It would be meaningless, for example, to state that a real variable is equal to a character variable. When such a control expression is evaluated, the result is a Boolean value: true or false. Thus we say that the type of the control expression is Boolean. The variables in a control expression must all have defined values when the while statement is entered. The statement is evaluated before each iteration. If it is true, then the statement within the loop is executed, if it is false, the loop is exited. In other words, the loop is repeated while the control expression is true. To put this in yet another way, the loop is repeated until the control expression is false. If the control expression is false at the beginning, then the loop is not executed at all.

This construct makes sense, of course, only if the control expression can (sometimes or always) be expected to be true at the beginning, and if something happens within the loop that can change the value of the control expression to false. If nothing within the loop can make the control expression false, the loop can never be exited. In our example, the control expression is:

```
TOTAL_X <= 1023
```


Synthetic Tutor

The value of `Total_X` is incremented within the loop, so we can be sure that it will eventually become equal to or greater than 1,023, resulting in exit from the loop. The effect of this is to draw a row of squares while we have space on the screen to do so; that is, until we run out of space at the right-hand edge of the screen (in this case drawing nine squares as shown in fig. 9-6).

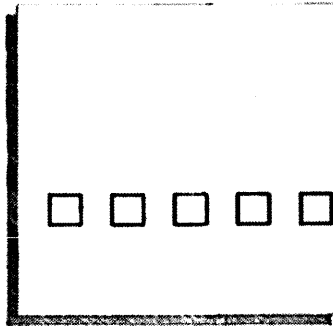
When the while statement is evaluated for the tenth time the value of `Total_X` is 1,030, so the expression

$$\text{TOTAL_X} \leq 1023$$

now evaluates to false, and the loop is exited. If the expression

$$x \leq 1023$$

had been used to control the loop, it would have evaluated to true on the tenth iteration, and a tenth square intersecting the right-hand side of the screen would have been drawn.



9-5. Another row of squares, produced by varying the parameters.

CODE

```
import rhinoscriptsyntax as rs

def drawSquare(x,y,length):
    # calculate width as function of length
    width = length/2

    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width

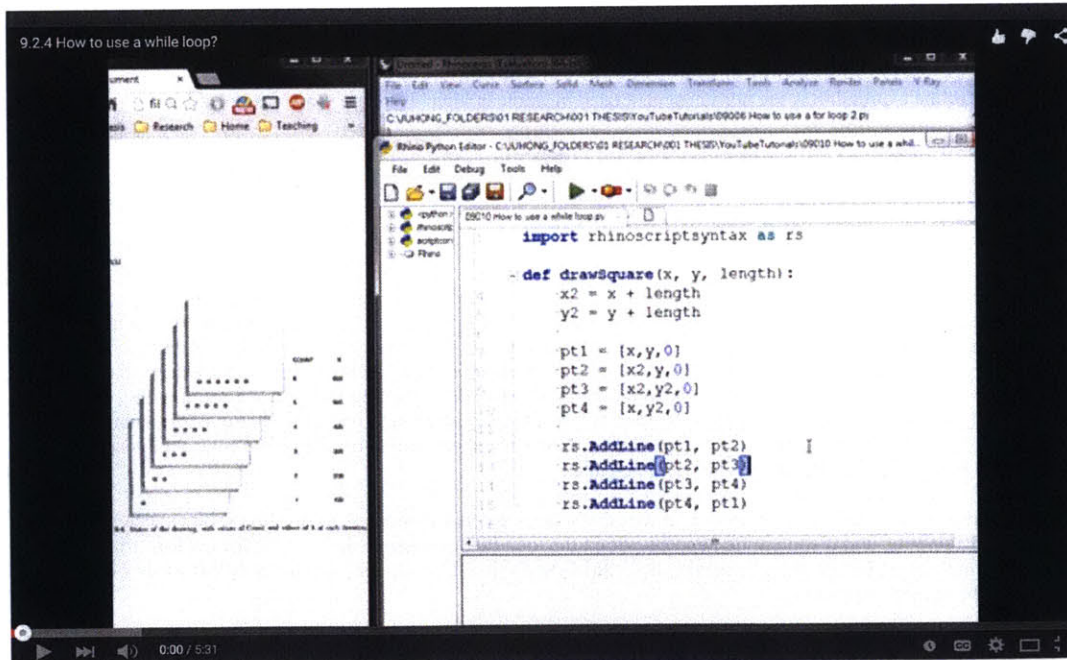
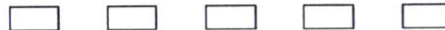
    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def drawRowSquares(x, x_increment, y, length, num):
    count = range(num)
    for i in count:
        drawSquare(x,y,length)
        x = x + x_increment

drawRowSquares(100,200,300,100,5)
```

RESULT



Module 75: 9.2 While

9. REPETITION

9.2 USE OF CONTROL STRUCTURES TO EXPRESS COMPOSITIONAL RULES

9.2.4 THE WHILE STATEMENT AND CONTROL EXPRESSIONS

The use of a control variable to keep count of repetitions, as in a for statement, is not the only way to control a loop. Python provides another possibility: the while statement.

Here is an example of a loop (to draw a row of squares) that uses the while statement:

[sample codes on the right side]

The while statement consists of the reserved word while, followed by an expression, then ending with colon, and finally the statement that is to be executed repeatedly (the use of parentheses is optional).

Here it is the control expression `Total_X <= 1,023` that controls the loop: This control expression is constructed using Python's Boolean operators:

```
and : conjunction
or  : disjunction
not : negation
==  : equal (two equals)
<> : unequal
<  : less
>  : greater
<= : less or equal
>= : greater or equal
```

The syntax of the expression is the normal syntax used in Boolean logic. Here are some examples:

```
X == Y
X <> Y
X < Y
X >= Y
(X <= Y) OR (A > B)
```

Integer, Real, Boolean, and character variables may all appear in these statements. However, you must be careful about type. It would be meaningless, for example, to state that a real variable is equal to a character variable. When such a control expression is evaluated, the result is a Boolean value: true or false. Thus we say that the type of the control expression is Boolean. The variables in a control expression must all have defined values when the while statement is entered. The statement is evaluated before each iteration. If it is true, then the statement within the loop is executed, if it is false, the loop is exited. In other words, the loop is repeated while the control expression is true. To put this in yet another way, the loop is repeated until the control expression is false. If the control expression is false at the beginning, then the loop is not executed at all.

This construct makes sense, of course, only if the control expression can (sometimes or always) be expected to be true at the beginning, and if something happens within the loop that can change the value of the control expression to false. If nothing within the loop can make the control expression false, the loop can never be exited. In our example, the control expression is:

```
TOTAL_X <= 1023
```

The value of `Total_X` is incremented within the loop, so we can be sure that it will eventually become equal to or greater than 1,023, resulting in exit from the loop. The effect of this is to draw a row of squares while we have space on the screen to do so; that is, until we run out of space at the right-hand edge of the screen (in this case drawing nine squares as shown in fig. 9-6).

When the while statement is evaluated for the tenth time the value of `Total_X` is 1,030, so the expression

```
TOTAL_X <= 1023
```

now evaluates to false, and the loop is exited. If the expression

```
x <= 1023
```

had been used to control the loop, it would have evaluated to true on the tenth iteration, and a tenth square intersecting the right-hand side of the screen would have been drawn.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def drawSquare(x,y,length):
    # calculate width as function of length
    width = length/2

    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width

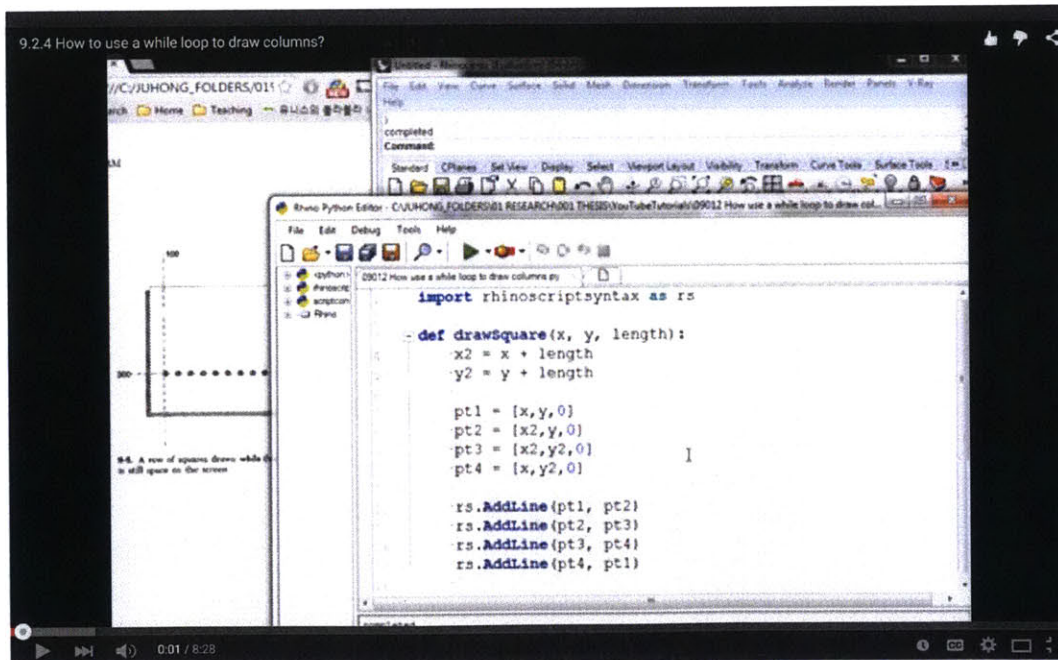
    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def drawRowSquares():
    x = 100
    total_x = x + 30
    while (total_x < 1023):
        drawSquare(x,300,30)
        x = x + 100
        total_x = x + 30

drawRowSquares()
```

RESULT



Module 76: 9.2 Break

9. REPETITION

9.2 USE OF CONTROL STRUCTURES TO EXPRESS COMPOSITIONAL RULES

9.2.5 THE WHILE BREAK STATEMENT

A third way to control a loop in Python is to use a while - break statement. Here is an example of a loop to draw a row of squares controlled in this way (fig. 9-8):

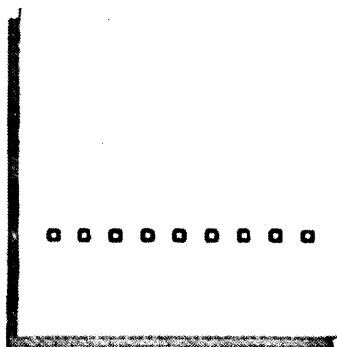
[sample codes on the right side]

The sequence of statements between the reserved words while and break is always executed at least once.

Execution of the loop is controlled by the Boolean expression that follows if. This is evaluated after every iteration. Initially we expect this expression to be true. Something happens within the loop that can make it false. When evaluation shows it to be false, the loop is exited.

It is generally feasible to write both a while and a while-break version of a loop. Choice of one or another is a matter of clarity of expression, and of whether or not we want to guarantee that there will always be at least one iteration.

Notice that the control expressions used in our examples of while and while-break loops must be evaluated at every iteration. In these examples, there are few iterations, and the control expression is simple, so the amount of computation required for this is insignificant. But it is possible to write very complex control expressions, and if there are many iterations, the amount of computation can then become significant. It is good programming practice, then, to keep control expressions as simple as possible.



9-8. A row of squares drawn until there is no space left on the screen.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def drawSquare(x,y,length):
    # calculate width as function of length
    width = length/2

    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width

    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

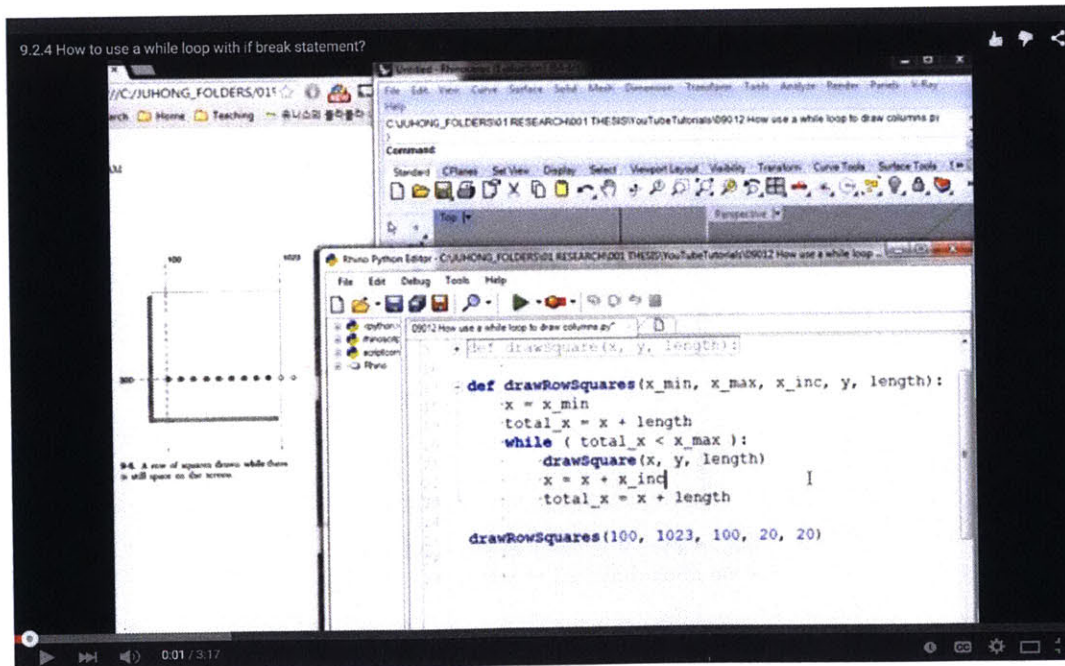
    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def drawRowSquares():
    x = 100
    total_x = x + 30

    while True:
        drawSquare(x,300,30)
        x = x + 100
        total_x = x + 30
        if (total_x > 1000):
            break

drawRowSquares()
```

RESULT



Module 77: 9.3 Variations

9. REPETITION

9.3 PROGRAMS TO EXPLORE VARIATIONS

Code to read in parameter values and draw the corresponding instance of a motif may be placed within a while - break loop as follows:

[sample codes on the right side]

Satisfactory must be declared as a char variable. The advantage of this arrangement is that a user of the program can keep cycling through the loop until a satisfactory variant of the motif is obtained.

You can always convert a program that generates a drawing of a motif into a program that allows you to explore variants of the motif by introducing a repeat until loop in this way. We suggest that from now on you do so in your own programming projects.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def drawSquare(x,y,length):
    # calculate width as function of length
    width = length/2

    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width

    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def drawRowSquares():
    x = 100

    while True:
        drawSquare(x,300,30)
        x = x + 100
        type = rs.GetString('type y or n and Enter')
        if (type == 'y'):
            break

drawRowSquares()
```

RESULT



Module 78: 9.4 Composition 1

9. REPETITION

9.4 THE COMPOSITIONAL USES OF REPETITION

Now that we have established the basic modes of repetition provided by Python, we can go on to consider various types of repetitive graphic compositions, and how these may be generated using loops.

We shall begin by considering compositions made by incrementing a single position parameter of a vocabulary element within a loop. Then we shall go on to consider incrementing two and three position parameters. Next, we shall look at incrementing a single shape parameter of a vocabulary element. Finally, we shall put all of this together and consider incrementing multiple parameters of a vocabulary element within a loop. Various kinds of mathematical progressions (arithmetic progressions, geometric progressions, and so on) are frequently used to structure architectural and graphic compositions. We shall consider how such progressions may be constructed within loops. We shall ultimately look beyond single loops to see what happens when we combine loops. There are two cases of this: compound loops and nested loops.

Always, though, we shall be concerned with two basic compositional issues. What is it that remains constant in the repeating motif from instance to instance? And what is the pattern of change from instance to instance across the composition? In other words, what are the variables, and what are their increments?

9.4.1 INCREMENTING A SINGLE POSITION PARAMETER

In all the examples provided so far, we have been incrementing a single position parameter—the X coordinate of the bottom-left corner of our column—within a loop. This produces a horizontal row of regularly spaced columns (fig. 9-9a). An obvious alternative is to keep the X coordinate constant, while incrementing the Y coordinate, to generate a vertical row (fig. 9-9b).

```
def incrementing():
    Y = 100
    for i in range(1, 7):
        square(300, Y, 30)
        Y = Y + 100
```

A third possibility is to increment the rotation of the column about some specified center point to produce a regularly spaced ring of columns (fig. 9-9c). You can write code to do this, using trigonometric functions, but it will be better to defer detailed consideration of this kind of composition until after we have discussed circles and arcs (in the next chapter) and the rotation transformation (in chapter 14).

It is worth noting that the most ancient architectural compositions that we know are based on this elementary idea of incrementing a single position parameter—perhaps to create settings for processional rituals. The Megaliths of Carnac, in Brittany, are regular rows of stones (fig. 9-10a), and at Stonehenge, the stones are regularly spaced around circles (fig. 9-10b).

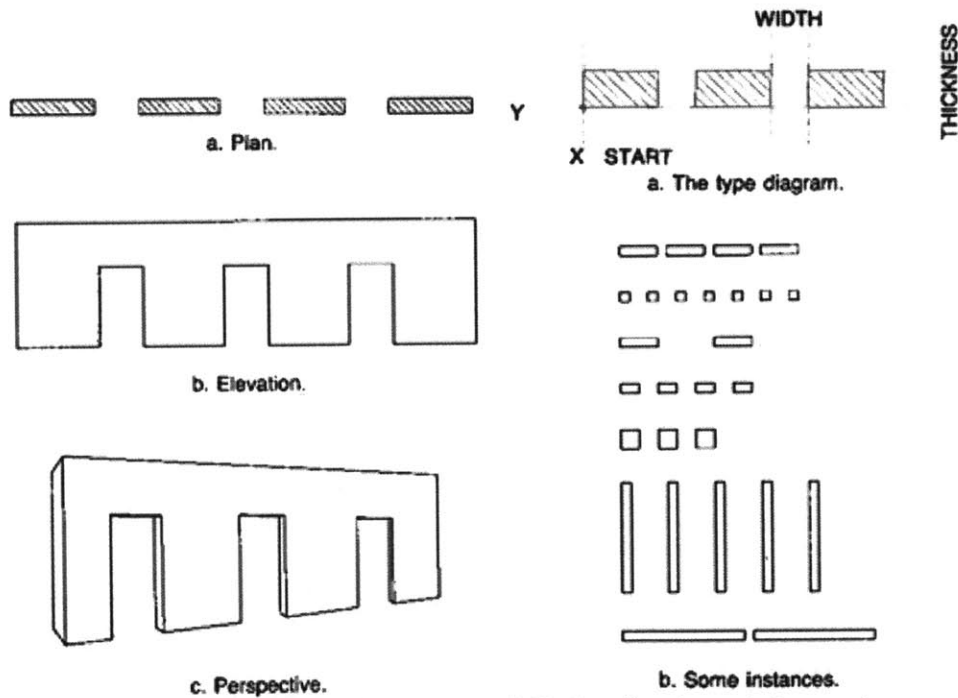
Architects have often made use of the device of regularly spaced rows of openings in wall planes. This shows up both in plan (fig. 9-11a) and in elevation (fig. 9-11b). The following procedure instantiates this architectural type. It takes as parameters the Y coordinate of the wall, its starting X coordinate, its ending X coordinate, the wall thickness, the number of openings, and the width of each opening (fig. 9-12a).

[sample codes on the right side]

Notice how this procedure automatically calculates the X increment that is to be used at each step. Notice, too, the

trouble that will be caused by invoking this procedure with parameter values that specify too many openings, or openings that are too wide so that they overlap. For the moment you will have to avoid such values. Later, in chapter 11, we shall see how to write code to check automatically for -illegal- parameter values and respond appropriately. Figure 9-12b shows some results generated by invoking this procedure with different parameters.

If you look closely at these results, you can discover an interesting change in the architectural meaning of the type as the values of the parameters vary. Where the voids are narrow relative to the solids, we interpret the object as a regularly pierced wall. Conversely, where the voids are wide relative to the solids, we interpret it as a colonnade—that is, as a regularly subdivided opening. If wall thickness is larger than the spacing of openings, we read parallel walls running in the perpendicular direction. There are also versions where the reading is ambiguous.



9-11. A regularly spaced row of openings in a wall plane.

9-12. A wall with regularly spaced openings.



a. Horizontal row.

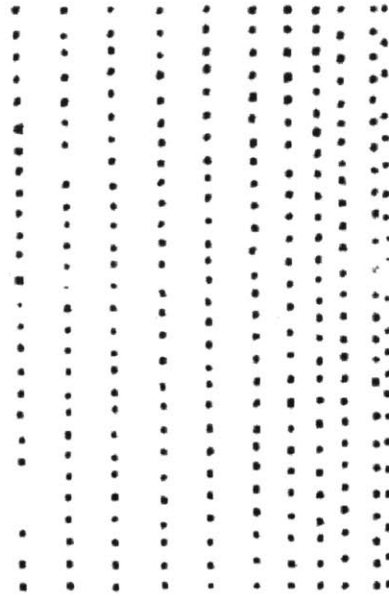


b. Vertical row.



c. Circle.

9-9. Simple compositions of squares generated by incrementing a single position parameter.



a. Rows of stones at Carnac, Brittany.



b. The circles of Stonehenge.

9-10. Megalithic compositions produced by the regular repetition of elements.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

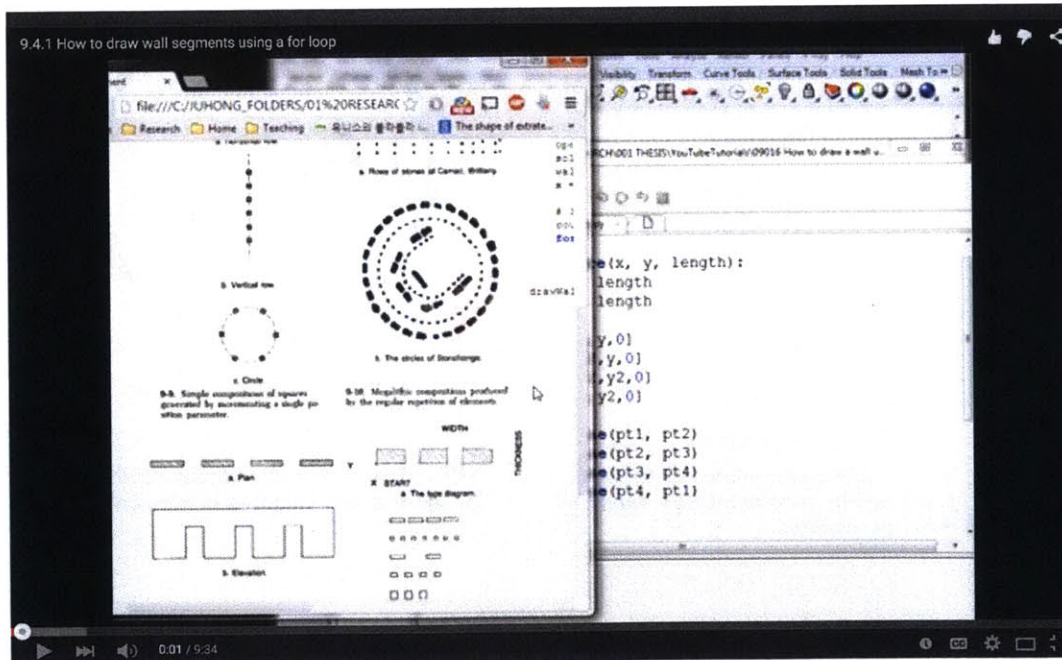
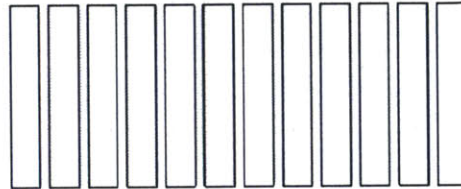
def drawRectangle(x,y,length,width):
    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width
    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]
    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def drawWall(y, x_start, x_end, thick, width, num):
    # calculate length of wall segments
    length = x_end - x_start
    open = width * num
    solid = length - open
    wall_segment = solid / num
    x = x_start

    # loop to draw wall segments
    count = range(num)
    for i in count:
        drawRectangle(x, y, wall_segment, thick)
        x = x + wall_segment + width

drawWall(100,100,600,200,10,12)
```

RESULT



Module 79: 9.4 Composition 2

9. REPETITION

9.4 THE COMPOSITIONAL USES OF REPETITION

9.4.1 INCREMENTING A SINGLE POSITION PARAMETER (continued)

Another common architectural device, based upon the idea of incrementing a single position parameter, is that of the *enfilade*-a sequence of parallel wall planes with aligned openings. The following procedure takes the parameters defining a boundary rectangle, the wall thickness, the width of the opening, and the spacing between walls as parameters (fig. 9-13a) and fits an *enfilade* within the specified rectangle:

The architectural logic followed here is that the last wall should occur just before the boundary of the rectangle is reached. Figure 9-13b illustrates some examples of *enfilades* specified by different sets of parameter values. (The boundary rectangle is not drawn by this procedure, but is shown for clarity in the illustrations. We have also added *poche* in the traditional manner.)

Notice that if the boundary rectangle is too small, no wall will be drawn. If we wanted to follow a different architectural logic, and guarantee that at least one wall would always be drawn, we could use a *while-break* instead of a *while* loop. We might also handle the relation of the last wall thickness to the boundary rectangle in a different way, by substituting the control expression

$$Y \leq Y_MAX$$

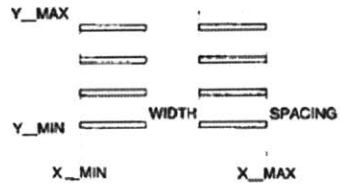
for the control expression

$$Y + THICKNESS \leq Y_MAX$$

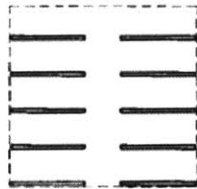
This allows the last wall to overlap the end of the boundary rectangle (fig.9-14), but never to be completely outside the boundary rectangle.

Now consider a row of rectangular wall panels shown in elevation as illustrated in fig. 9-15. It is clear that you can control widths, numbers of repetitions, and the termination condition in several different ways, depending on what is the most appropriate architectural principle to follow. You can let width vary and have your procedure automatically fit a specified number of panels into the boundary rectangle using a *for* loop (fig. 9-15b). If you keep width constant, then you can use a *while* or *while-break* loop, so that there is *undershoot* at the right side (fig. 9-15c). Alternatively, you can control the loop, so that there is an *overshoot* at the right side (fig. 9-15d). The procedure can be modified so that the *undershoot* is at the left side (fig. 9-15e), or so that the *overshoot* is at the left side (fig. 9-15f). A final pair of possibilities is to divide the necessary *undershoot* in half at both ends (fig. 9-15g), or to do the same with *overshoot* (fig. 9-15h).

High-rise office buildings typically consist of parallel, regularly repeated floor planes (fig. 9-16). These can be drawn in section by a procedure that takes location coordinates for the ground floor, floor length, floor thickness, floor-to-floor height, and the number of floors as the parameters (fig. 9-16a). Alternatively, an architect might find it more convenient to use floor-to-ceiling height as a parameter (fig. 9-16b). Should you start with a floor plane (fig. 9-16c), or with a floor-to-floor space (fig. 9-16d)? That is a question of architectural principle that you must resolve before writing the procedure. You must then express your resolution appropriately in the code. Similarly, should you count the last (top) plane as a floor (fig. 9-16e), or as a roof (fig. 9-16f)?

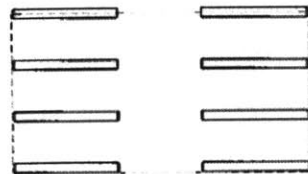


a. The type diagram.

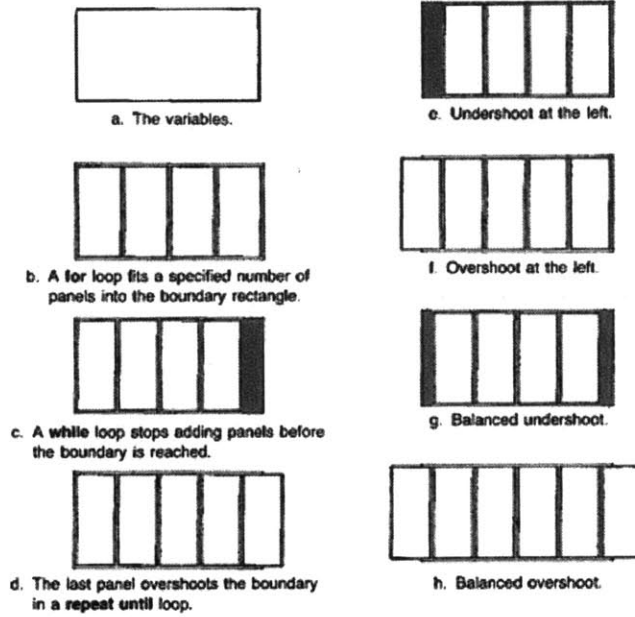


b. Instances of enfilades fitted within specified rectangles.

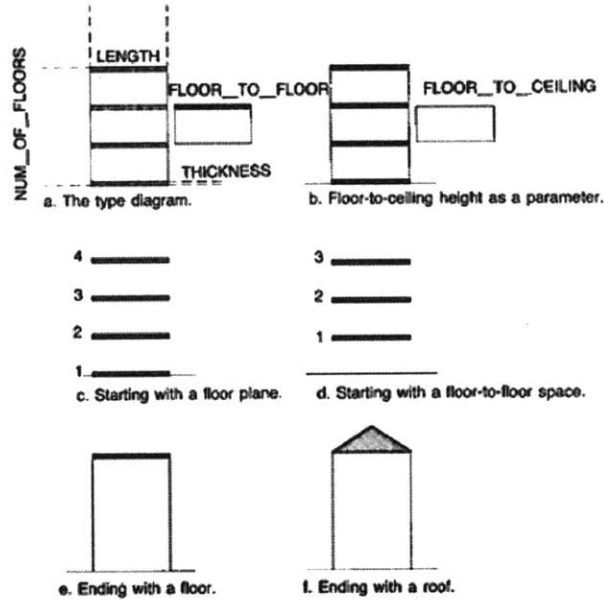
9-13. The use of a while loop to generate an enfilade.



9-14. The use of a repeat until loop to generate an enfilade.

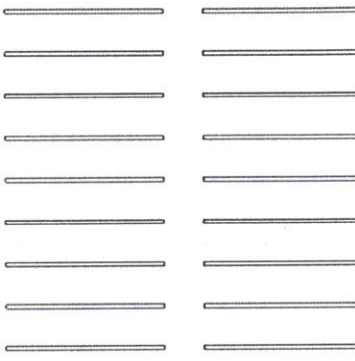


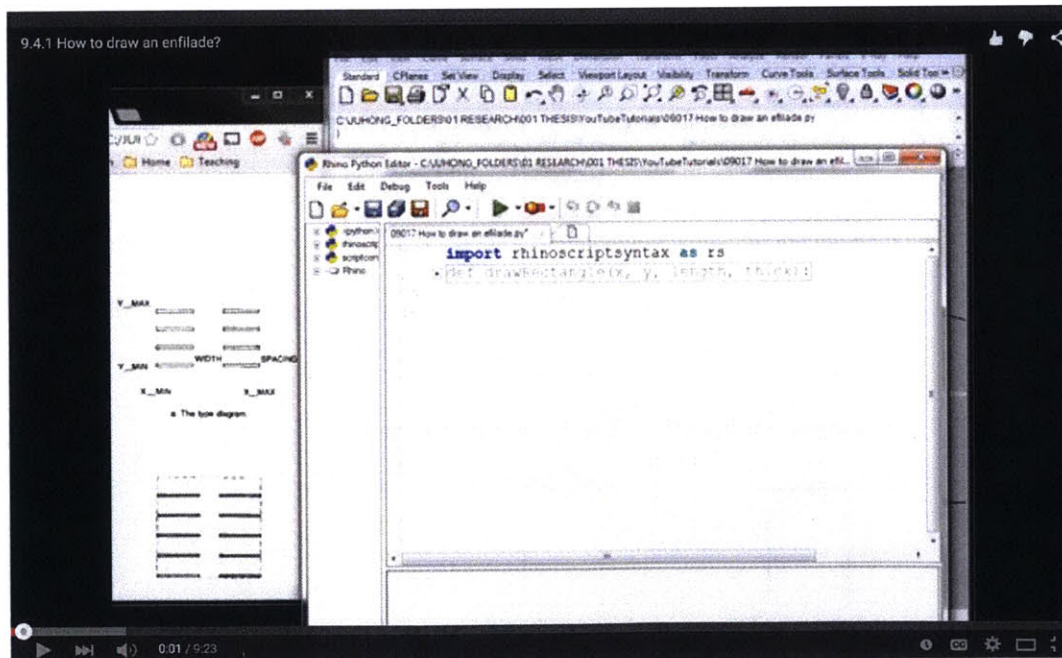
9-15. A row of rectangular wall panels with different conditions governing termination.



9-16. A schematic section of a high-rise office building.

Synthetic Tutor

<pre>CODE import rhinoscriptsyntax as rs def drawRectangle(x,y,length,width): # calculate vales for x2 and y2 x2 = x + length y2 = y + width pt1 = [x,y,0] pt2 = [x2,y,0] pt3 = [x2,y2,0] pt4 = [x,y2,0] rs.AddLine(pt1,pt2) rs.AddLine(pt2,pt3) rs.AddLine(pt3,pt4) rs.AddLine(pt4,pt1) def drawEnfilade(x_min, y_min, x_max, y_max, thick,width, spacing): # calculate length of wall segment gap = (x_max - x_min) - spacing length = gap /2 inc = thick + spacing x1 = x_min x2 = x_max - length y = y_min # loop to place parallel wall planes while (y + thick) < y_max): drawRectangle(x1,y,length,thick) drawRectangle(x2,y,length,thick) y = y + inc drawEnfilade(100,100,1000,1000,10,600,100)</pre>	<p>RESULT</p> 
--	--



Module 80: Exercise 4

8.
GRAPHIC VOCABULARIES

8.10 EXERCISES

1. Take the graphic programs that you have developed so far and create procedures from them, so the figures that they draw can be used as vocabulary elements. Put them together in a program that generates a simple composition.

Please upload your python file: No file chosen

2. The following procedure generates a simple line figure.

```
import rhinoscriptsyntax as rs

def cross_box (xc, yc, length, width):

    x1 = xc - length / 2
    y1 = yc - width / 2
    x2 = x1 + length
    y2 = y1 + width

    pt0 = [x1, y1, 0]
    pt1 = [x2, y1, 0]
    pt2 = [x2, y2, 0]
    pt3 = [x1, y1, 0]
    pt4 = [x2, y2, 0]
    pt5 = [x1, y2, 0]
    pt6 = [x2, y1, 0]
    pts = [pt0, pt1, pt2, pt3, pt4, pt5, pt6]

    rs.AddPolyline(pts)
```

The following invocations generate instance of this figure:

```
cross_box( 450, 50, 200, 50)
cross_box( 400, 200, 100, 200)
cross_box( 500, 200, 100, 200)
cross_box( 400, 400, 100, 100)
cross_box( 500, 400, 100, 100)
```

Draw these instances.

3. Many graphic artists and designers have been fascinated by the generation of highly disciplined but interesting compositions, using nothing more than a square as the vocabulary element. Think about the graphic variables that you might want to use in producing such a composition, and write a Square procedure with these as the formal parameters. Use this procedure in a program to generate a composition.

Please upload your python file: No file chosen

Synthetic Tutor

4. Repeat this exercise with progressively more general types of four-sided figures: rectangle, parallelogram, and polygon.

Please upload your python file: No file chosen

5. Try the same exercise with triangles. Remember that **there are four** recognized subtypes: equilateral, right, isosceles, and scalene. Do you want to use all of these? **Should they all** be parameterized in the same way?

Please upload your python file: No file chosen

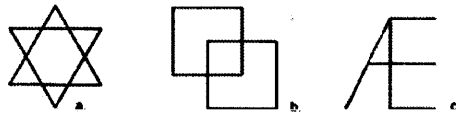
6. Consider the uppercase characters needed to draw your **initials**. How might these be parameterized to allow a wide variety of alternatives? Write the necessary **procedures, and use** them in programs to draw monograms.

Please upload your python file: No file chosen

7. Take one of your procedures to draw a graphic element, and use it in an interactive program that reads in values for the parameters, then displays the corresponding instance. Use the program to generate a series of variations of the motif.

Please upload your python file: No file chosen

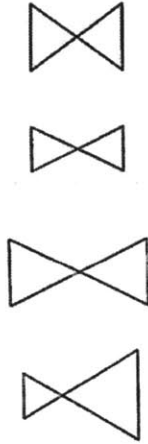
8. The Star of David (fig. 8-46a) is a well-known two-dimensional line figure. We usually think of it as two superimposed triangles, but there are many less obvious ways to decompose it. Think of an interesting decomposition, write appropriately parameterized procedures to generate each of the parts, and use these in programs to generate variations on the Star of David. You may find it interesting to repeat this exercise for different decompositions and different parameterization schemes. Repeat the exercise for the line drawings shown in figures 8-46b and 8-46c.



8-46. Some geometric figures to be decomposed.

Please upload your python file: No file chosen

9. Figure 8-47 shows four instances of a type. What properties are constant from instance to instance? What properties vary? Write a procedure to generate figures of this type.



8-47. Some instances of a type.

Please upload your python file: No file chosen

10. Figure 8-48 shows a well-known floor plan by Mies van der Rohe. Write a parameterized procedure to generate the basic vocabulary element. Use this in a program to replicate the plan. Then use it in programs to produce variations on this theme.



8-48. Mies van der Rohe's "Project for a Brick Country House," 1923.

Please upload your python file: No file chosen

11. Clothing designers often establish a motif, then produce a series of parametric variants to fit the motif to

Synthetic Tutor

different figures. What are the usual parameters of hats, shirts, trousers, and shoes? What varies when a new value is given to one of these parameters? What remains constant? Other designed artifacts (for example, steel beams, bathtubs, and pen nibs) are also produced in series of parametric variants. Write an analysis of the role of parametric variation in design(write your analysis as comments(with #) in a python file).

Please upload your python file: No file chosen

12. Metafont system for typographic design (Knuth 1986) is one of the most sophisticated explorations of the idea of parametric variation by computer yet to appear. Study its characteristics carefully, compare it to more traditional tools for typographic design, and write a critique (write your critique as comments(with #) in a python file).

Please upload your python file: No file chosen

Module 81: 9.4 Composition 3

9. REPETITION

9.4 THE COMPOSITIONAL USES OF REPETITION

9.4.1 INCREMENTING A SINGLE POSITION PARAMETER (continued)

It is common for building codes to limit the heights of buildings by specifying the maximum angle `Max_angle` that can be formed at the center of a street (fig. 9-17). As an architect, you might be particularly interested in the maximum floor area that you can fit on a site. Let us assume that the floors of our building are rectangular. The following interactive program reads in values for floor `Length` and `Width`, `floor_to_floor` height, and for the constraints `Max_angle` and `Street_width`, then draws the building section and displays the total floor area. Notice the use of a function: `getMaxHeight`, which calculates the maximum allowable height for given `Street_width` and `Max_angle`. Here is the complete code:

[sample codes on the right side]

Some typical output is illustrated in figure 9-18.

This program introduces an important new idea. There is a function called `Total_area`, which calculates the total floor area of a building and is invoked after the building is drawn. This is an analysis function and is executed to tell us something useful about the object that has been drawn. So the structure of our program is essentially as follows:

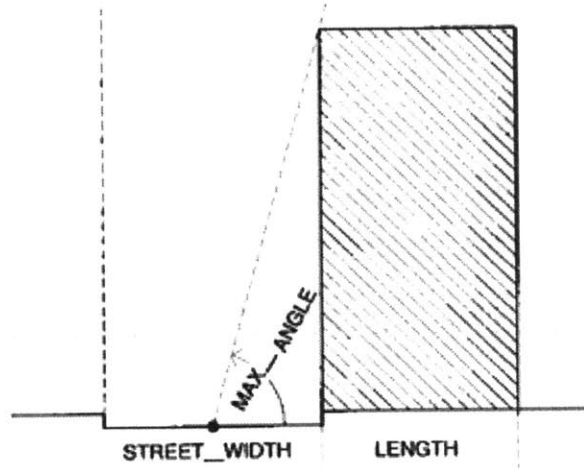
```
def declare_function():
    Read values of independent variables
    Calculate values of dependent variables

    Draw the design

    Perform analysis
```


This is not just a graphics program, then; it is a simple example of a computer aided design program. It assists the designer not only by rapidly drawing the building, but also by automatically performing some of the problem solving that is necessary before the building can be drawn, then by automatically performing some of the analysis that is necessary after the building has been drawn. This allows a very rapid trial-and-error design process, as shown by the flow diagram in figure 9-19.

We shall generally restrict our attention to computer graphics, rather than explore the much larger topic of computer-aided design. But it is useful to remember that, in practice, graphics procedures are often embedded in computer-aided design programs, and that the code of a computer-aided design program basically consists of problem-solving functions and procedures, graphics procedures, and analysis functions and procedures.

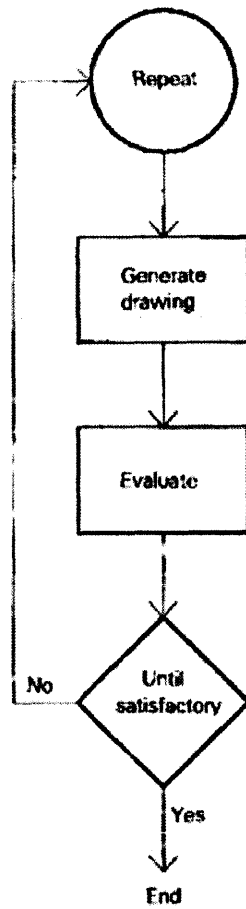


9-17. Max_angle and street_width as parameters controlling the height of a building.

```
ENTER LENGTH OF FLOOR:
200
ENTER WIDTH OF FLOOR:
150
ENTER THICKNESS OF FLOOR:
7
ENTER FLOOR_TO_FLOOR HEIGHT
35
ENTER STREET WIDTH:
300
ENTER MAXIMUM ANGLE:
56.5
TOTAL FLOOR AREA IS: 180000
```



9-18. An interactive program to draw and analyze a high-rise office building.



9-19. A flow diagram of a trial-and-error design process.

CODE

```
import rhinoscriptsyntax as rs
import math

def drawRectangle(x,y,length,width):
    # calculate vales for x2 and y2
    x2 = x + length
    y2 = y + width
    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]
    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def getMaxHeight(street_width,max_angle):
    radians = 0.01745
    max_angle = max_angle * radians
    angle_factor = math.cos(max_angle) * math.sin(max_angle)
    max_height = (street_width/2) / angle_factor
    return max_height

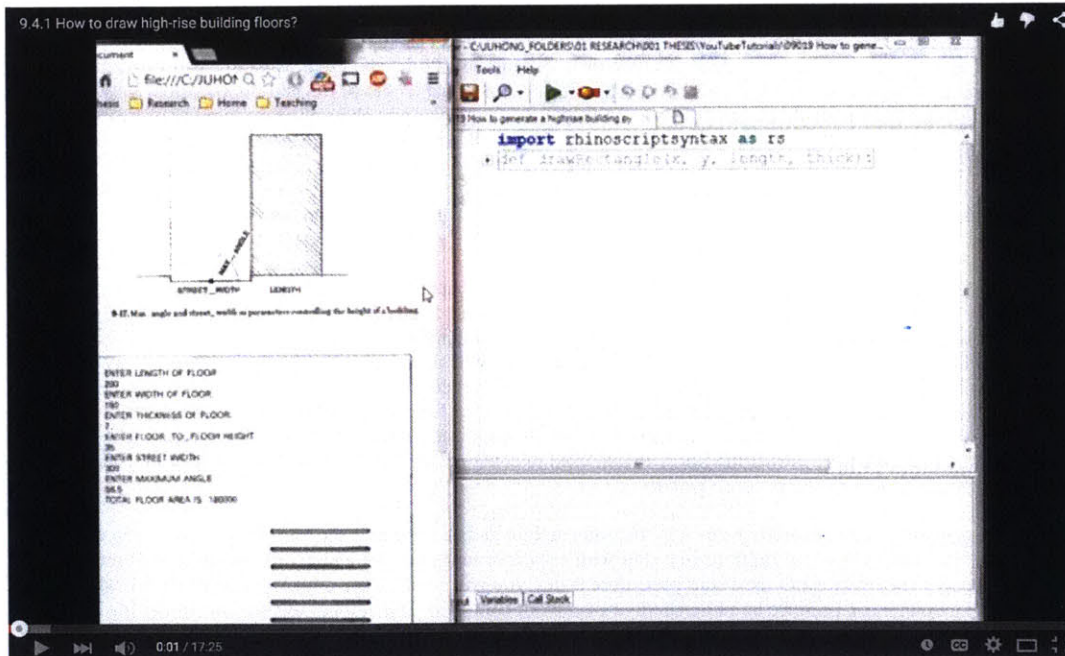
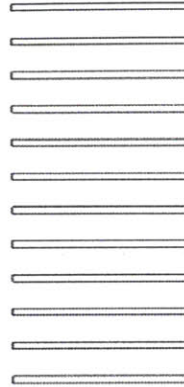
# draw floors of highrise
def drawHighrise(x,y,
                length,thickness,
                floor_to_floor,street_width,
                max_angle):

    # calculate max hight of building
    max_height = getMaxHeight(street_width, max_angle)
    total_height = max_height - thickness
    height = 0

    # loop to draw floors
    while (height < total_height):
        drawRectangle(x,y,length,thickness)
        height = height + floor_to_floor
        y = y + floor_to_floor

drawHighrise(100,100,500,20,100,1000,60)
```

RESULT



Module 82: 9.4 Incrementing 1

9. REPETITION

9.4 THE COMPOSITIONAL USES OF REPETITION

9.4.2 INCREMENTING TWO POSITION PARAMETERS

Figure 9-20. shows some section drawings of stairs. They were generated by the following procedures

[sample code on the right side]

The parameters of this procedure specify a starting point, the dimensions of the tread, the X and Y increments at each step, and the number of stairs. That is, we now increment two position parameters at each iteration.

A number of important architectural variables are involved here. The principal ones are (fig. 9-21)

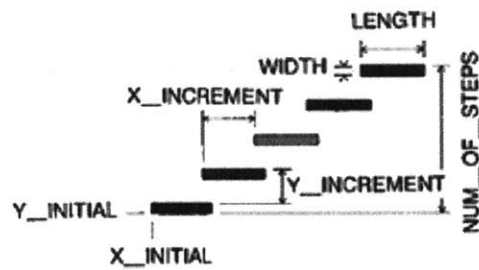
- . tread length
- . tread width
- . number of treads
- . number of risers
- . tread increment
- . riser increment
- . tread/riser increment ratio
- . angle formed with horizontal
- . length of run
- . floor-to-floor height

These are interrelated by functions in obvious ways, so not all of them can be taken as independent. You can parameterize a stair procedure in a variety of ways, then, depending on how you might want to use it in generating architectural drawings. You might reasonably assume, for example, that length of run and floor-to-floor height will usually be givens, so you would express these as parameters. Then you might choose to take number of treads as an independent design variable, so this too would be a parameter to the procedure. These three parameters, together with tread length and width and starting coordinates, are sufficient to define fully an instance. Alternatively, you might take the number of risers, riser increment, and tread increment as parameters. There are other reasonable possibilities as well in any case, there will be seven independent design variables, and others will become dependent.

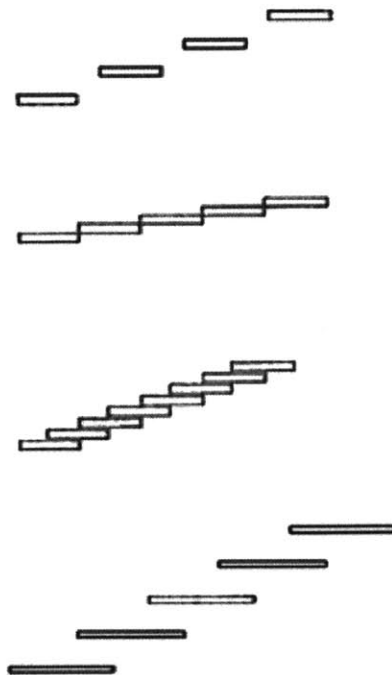
Not only are there different ways to parameterize stair procedures, but also different ways to handle termination. Do you want to begin with a tread or with a riser? Similarly, do you want to end with a tread or with a riser? Figure 9-22 shows the four possible combinations of cases. The choices that you make will have implications for how the length of run and floor-to-floor height are defined and calculated. They will also determine whether you will always have an odd number of treads, or an even number, or whether you can get either one. The question can be of some architectural importance. It was a rule of Roman temple architecture, for example, that there should always be an odd number of steps. In the context of actual construction, the termination rules determine where the change of material from that used to construct the floors to that used to construct the treads takes place, which can have important structural and aesthetic consequences.

The risers in stairs should generally be exactly the same height and all the treads the same length. Anything else is dangerous. This means that you cannot end a stair with a short or long riser or tread in order to make it fit into a specified space. The implication for a stair procedure is that you will always have a for loop, rather than a while or a repeat until loop that can produce an undershoot or overshoot condition. The number of iterations of the for loop might be specified directly, or it might be calculated from information about length of run, floor-to-floor height, and either tread increment or riser increment.

There is an important lesson to be drawn from these detailed analyses of procedures to draw simple, repetitive architectural compositions. If you want to write really useful procedures to draw repetitive compositions, you must think very carefully about the logic of parameterization and about the desired end conditions. The Python distinction between for, while, and repeat until loops provides a useful framework for this and enables you to express the principles that you have chosen to follow in a clear and explicit way.

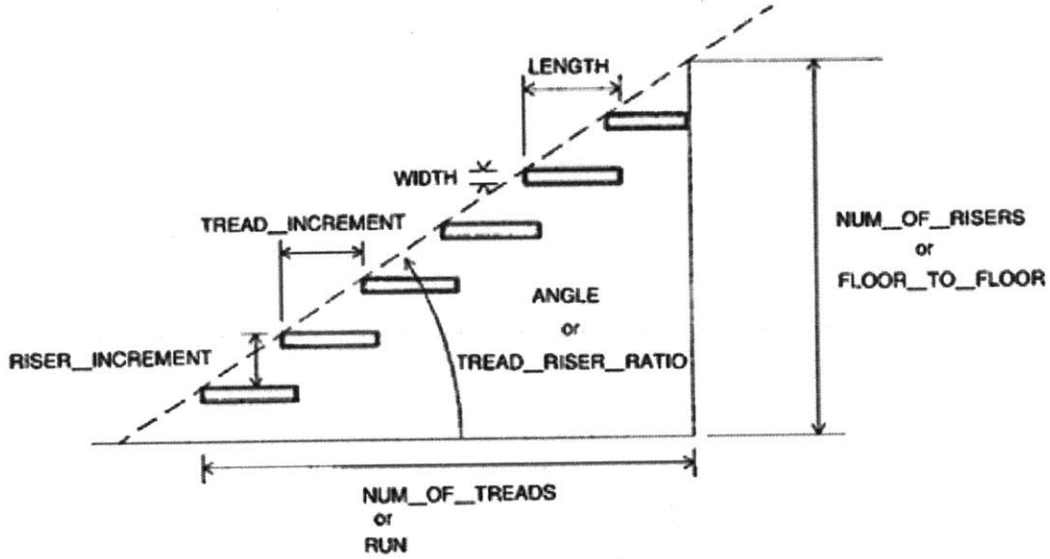


a. Type diagram.

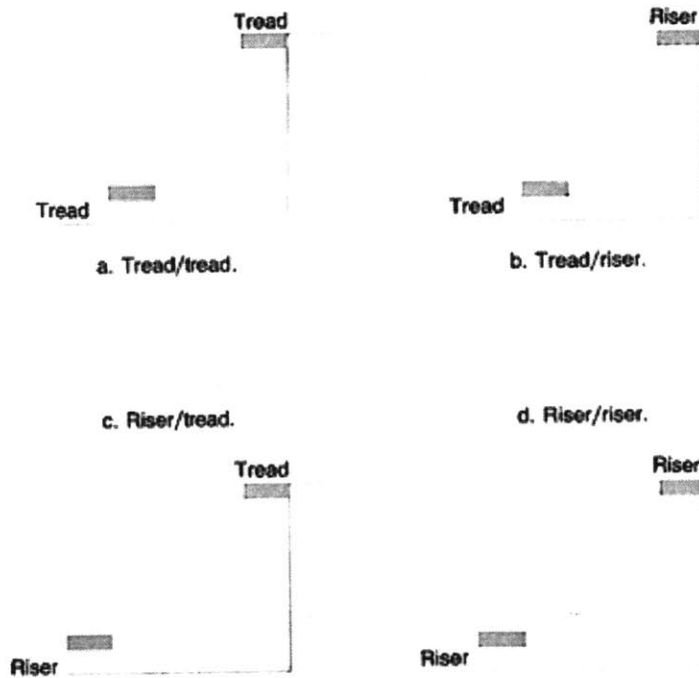


b. Instances.

9-20. Section drawings of stairs.



9-21. The principal architectural variables associated with a stair.



9-22. Possible starting and finishing conditions for a square.

CODE

```
import rhinoscriptsyntax as rs

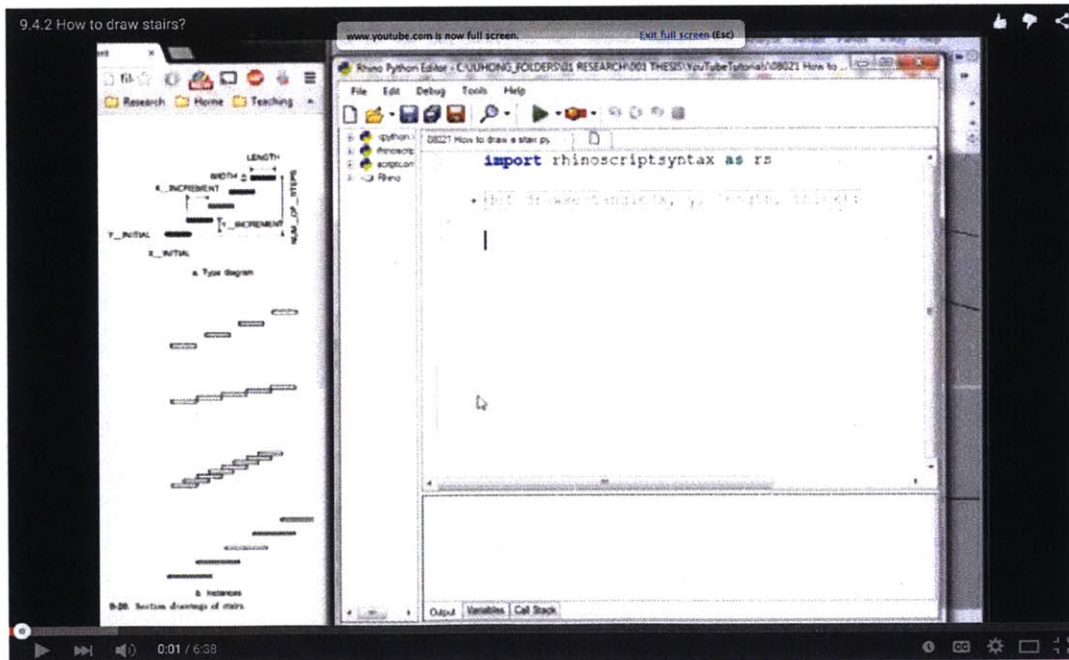
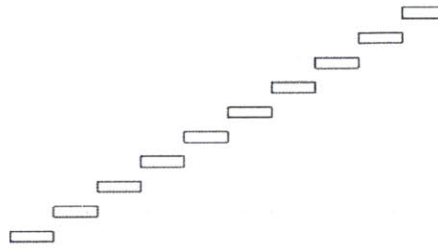
def drawRectangle(x, y, length, width):
    x2 = x + length
    y2 = y + width
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawStairs(x_initial, y_initial,
              length, width,
              x_increment, y_increment,
              num_of_steps):

    x = x_initial
    y = y_initial
    count = range(num_of_steps)
    for i in count:
        drawRectangle(x, y, length, width)
        x = x + x_increment
        y = y + y_increment

drawStairs(10, 10, 12, 3, 12, 7, 10)
```

RESULT



Module 83: 9.4 Incrementing 2

9. REPETITION

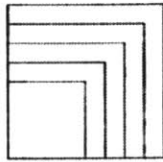
9.4 THE COMPOSITIONAL USES OF REPETITION

9.4.3 INCREMENTING A SINGLE SHAPE PARAMETER

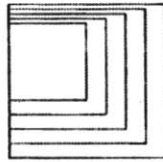
Another very interesting class of compositions can be generated by keeping position constant, but incrementing a single shape parameter at each iteration. The following procedure, for example, takes as parameters coordinates of a starting point, a starting side length, a length increment, and the number of repetitions incorporates a for loop and generates compositions of concentric squares as illustrated in figure 9-23.

[sample code on the right side]

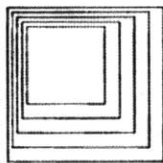
We have used here a procedure Square that locates a square by its center point, but the fixed point might be anywhere. Figure 9-24 illustrates examples of compositions of squares generated by taking the fixed point at a corner, at an arbitrary point along an edge, at an arbitrary point inside the initial square, and at an arbitrary point outside the square. Similar games can be played with triangles (fig. 9-25).



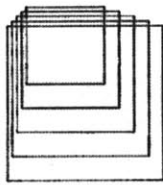
a. At a corner.



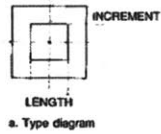
b. On an edge.



c. Inside.



d. Outside

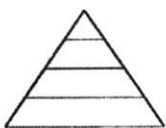


a. Type diagram



b. Some instances.

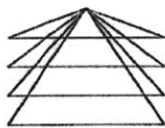
9-23. A composition of nested squares.



a. Equilateral triangles fixed at apex.



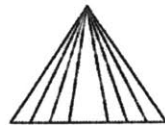
b. Equilateral triangles fixed at center of base.



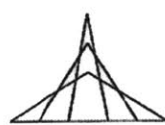
c. Isosceles triangles of constant width fixed at apex.



d. Isosceles triangles of constant width fixed at center of base.



e. Isosceles triangles of constant height fixed at apex.



f. Height and width vary, with fixed point at center of base.

9-25. Compositions of triangles about different fixed points.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def square(x_center, y_center, length):
    pass
    x1=x_center - length/2
    y1=y_center - length/2

    # calculate vales for x2 and y2
    x2 = x1 + length
    y2 = y1 + length

    pt1 = [x1,y1,0]
    pt2 = [x2,y1,0]
    pt3 = [x2,y2,0]
    pt4 = [x1,y2,0]

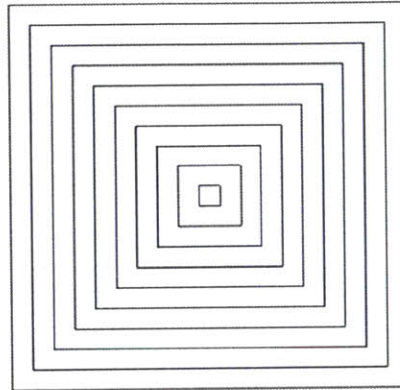
    rs.AddLine(pt1,pt2)
    rs.AddLine(pt2,pt3)
    rs.AddLine(pt3,pt4)
    rs.AddLine(pt4,pt1)

def nestSquares(x_center, y_center,
               length, increment,
               repetition):

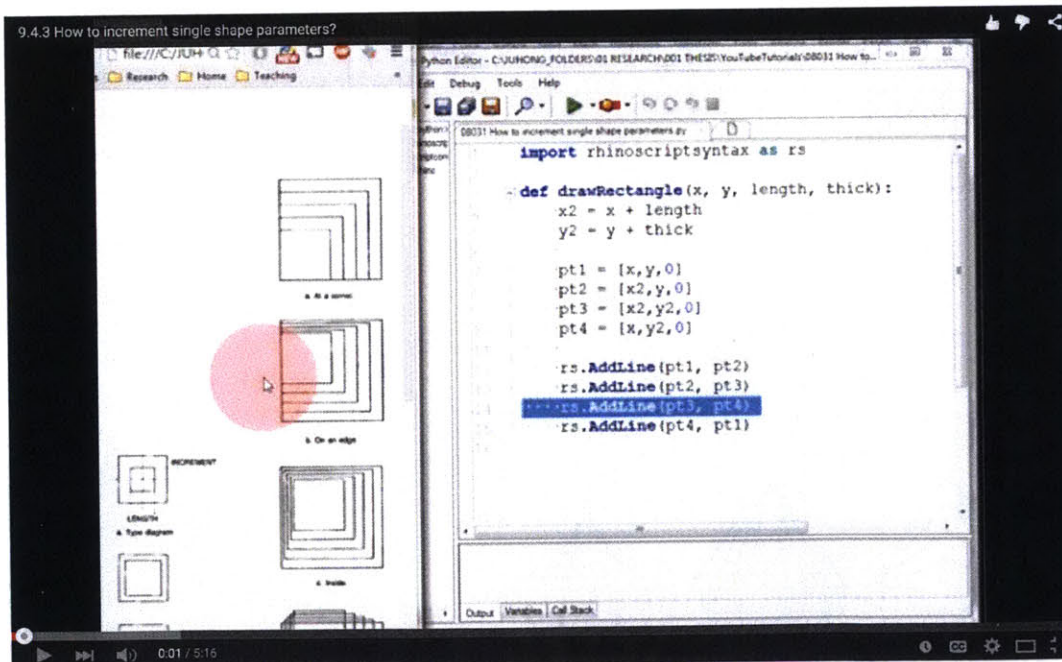
    count = range(repetition)
    for i in count:
        square(x_center, y_center, length)
        length = length + increment * 2

nestSquares(0, 0, 100, 100, 10)
```

RESULT



9.4.3 How to increment single shape parameters?



```
import rhinoscriptsyntax as rs

def drawRectangle(x, y, length, thick):
    x2 = x + length
    y2 = y + thick

    pt1 = [x,y,0]
    pt2 = [x2,y,0]
    pt3 = [x2,y2,0]
    pt4 = [x,y2,0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)
```

Module 84: 9.4 Incrementing 3

9. REPETITION

9.4 THE COMPOSITIONAL USES OF REPETITION

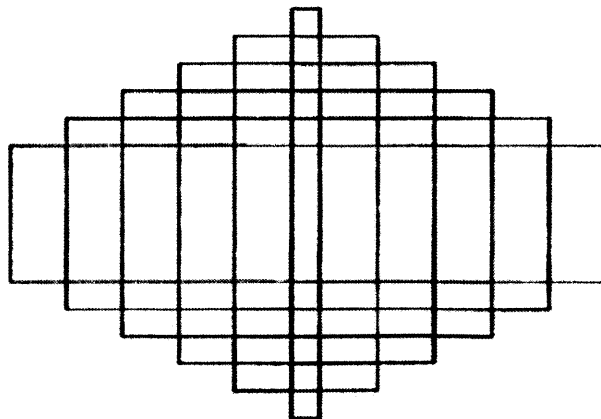
9.4.4 INCREMENTING MULTIPLE SHAPE PARAMETERS

We saw in chapter 8 that a figure can have not just one, but many shape parameters. Where there are multiple shape parameters, any number of these may be incremented within a loop.

We have seen, for instance, that a rectangle parallel to the coordinate axes has two shape parameters Length and Width. The following iterative procedure takes as parameters an initial Length and Width, a Length and Width increment, and a lower limit on the value of Length

[sample code on the right side]

At each iteration, this procedure subtracts the increment from Length (that is, it decrements), and it adds the increment to Width. The control structure is a while loop, controlled by the lower bound on Length. It generates the kind of composition shown in figure 9-26.



9-26. Composition produced by incrementing Length and Width of concentric rectangles.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

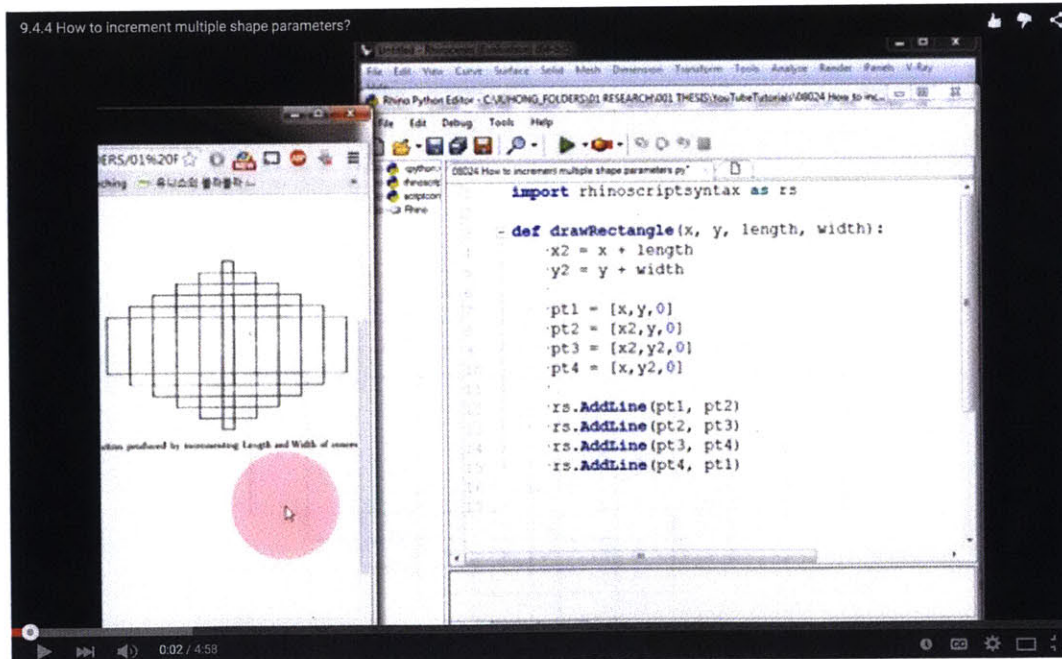
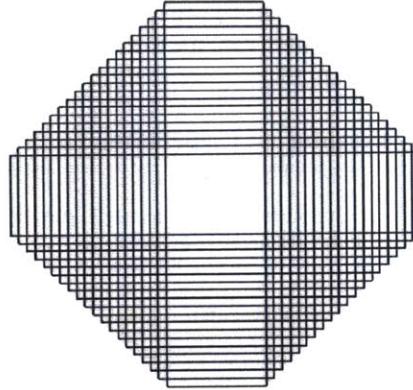
def drawRectangle(x, y, length, width):
    x2 = x + length
    y2 = y + width
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawCrossRectangle(x, y,
                      length, width,
                      length_inc, width_inc,
                      limit):
    x1 = x
    y1 = y

    while ( length > limit ):
        drawRectangle(x1, y1, length, width)
        x1 = x1 + length_inc
        y1 = y1 - width_inc
        length = length - length_inc * 2
        width = width + width_inc * 2

drawCrossRectangle(0,0, 500,100, 10,10, 100)
```

RESULT



Module 85: 9.4 Incrementing 4

9. REPETITION

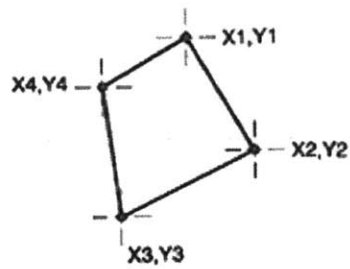
9.4 THE COMPOSITIONAL USES OF REPETITION

9.4.4 INCREMENTING MULTIPLE SHAPE PARAMETERS

Now consider the general four-sided object shown in figure 9-27a. It has eight shape parameters the X and Y coordinates of the four vertices. We can associate an increment, which may be a positive or a negative number, with each of these. The following procedure with seventeen parameters (including a parameter controlling the number of iterations) changes all eight shape variables at each iteration

[sample code on the right side]

Some of the many compositions that this procedure can generate are shown in figure 9-27b. Notice that, as the quadrilateral changes with each iteration, each vertex moves along a straight line. The angle of this line and the direction of movement are determined by the relation between the X and Y increments.



a. Parameters of the quadrilateral element.



b. Some instances.

9-27. Compositions produced by incrementing vertex coordinates of a quadrilateral.

CODE

```
import rhinoscriptsyntax as rs

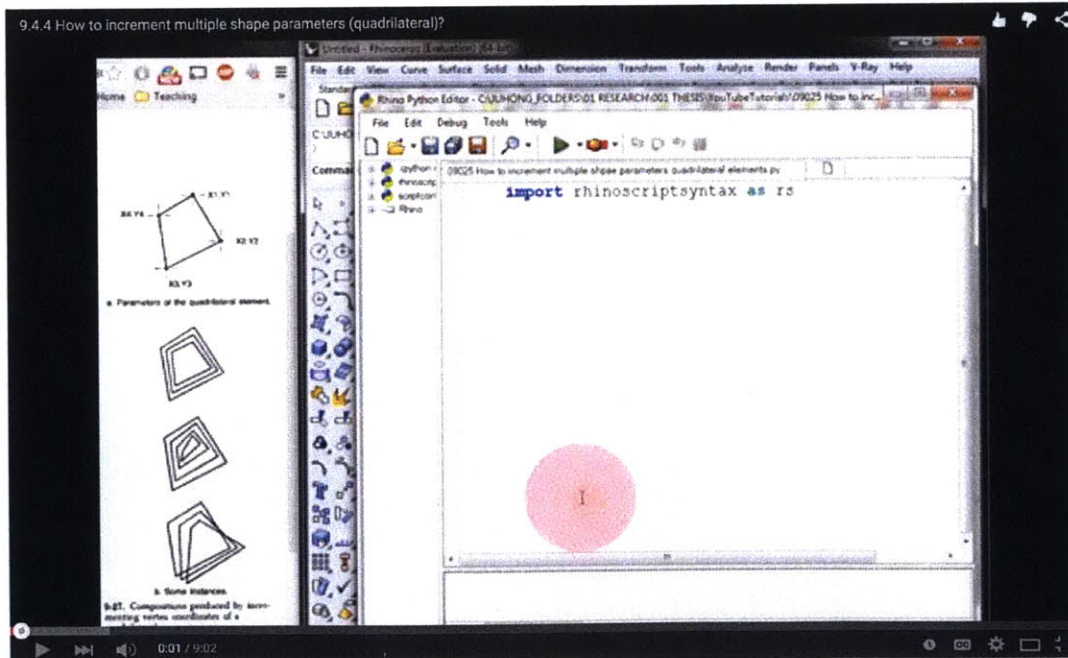
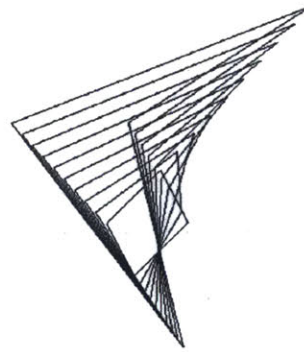
def four_side(p0,p1,p2,p3):
    points = [p0,p1,p2,p3,p0]
    rs.AddPolyline(points)

def drawNestFigure(x1, y1,
                  x2, y2,
                  x3, y3,
                  x4, y4,
                  x1_inc, y1_inc,
                  x2_inc, y2_inc,
                  x3_inc, y3_inc,
                  x4_inc, y4_inc,
                  repetitions):
    count = range(repetitions)
    for i in count:
        p0 = [x1, y1, 0]
        p1 = [x2, y2, 0]
        p2 = [x3, y3, 0]
        p3 = [x4, y4, 0]
        four_side(p0,p1,p2,p3)

        x1 = x1 + x1_inc
        y1 = y1 + y1_inc
        x2 = x2 + x2_inc
        y2 = y2 + y2_inc
        x3 = x3 + x3_inc
        y3 = y3 + y3_inc
        x4 = x4 + x4_inc
        y4 = y4 + y4_inc

drawNestFigure(0, 0,
              100, 100,
              50, 200,
              -50, 100,
              10, -15,
              -12, 21,
              31, 34,
              -19, 21,
              10)
```

RESULT



Module 86: 9.4 Incrementing 5

9. REPETITION

9.4 THE COMPOSITIONAL USES OF REPETITION

9.4.5 INCREMENTING BOTH POSITION AND SHAPE PARAMETERS

Sometimes in nature we find that the shapes of instances of some type of object vary systematically with position. In a grove of trees, for example, the trees in the interior grow tall and narrow in order to reach the light, whereas trees on the outside are shorter (fig. 9-28). Alternatively, instances may be sorted and arranged spatially according to shape. Children might be lined up for a photograph, for example, in strict order from the shortest to the tallest.

In architecture it is very common for position and shape to vary regularly in a correlated way. A regular row of columns supporting a pitched roof must grow in height (fig. 9-29a). Structural logic suggests that columns will also become thicker as they get taller (fig. 9-29b).

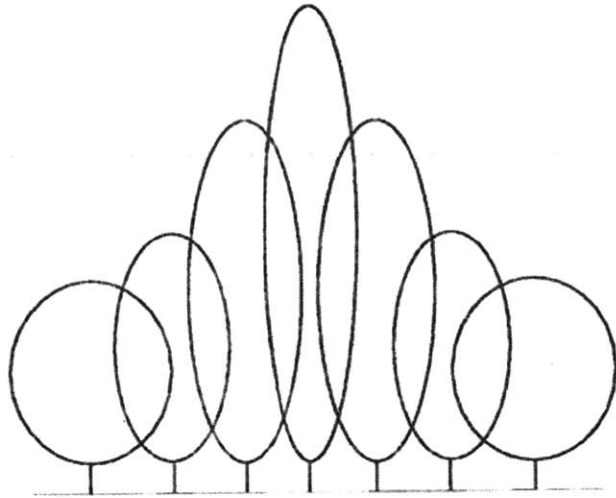
[sample code on the right side]

The logic of this kind of composition can often be captured with a procedure that increments both shape and position parameters within a loop. The following simple procedure, for example, generates column rows of the type shown in figure 9-29b

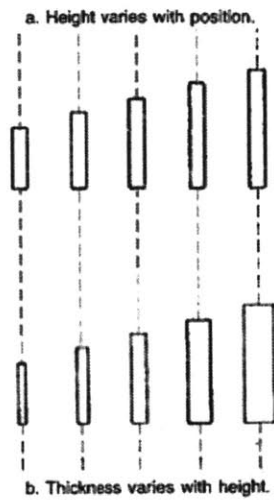
Note that the parameters specify initial position and dimensions, center-to-center column spacing, number of columns, height increment, and thickness increment. You should be able to think of other, perhaps more convenient, parameterization schemes.

Here is an analogous procedure to draw a stepped pyramid in elevation

[sample code on the right side]



9-25. A grove of trees; interior trees grow taller and narrower in order to reach light.



9-29. A row of columns supporting a pitched roof.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

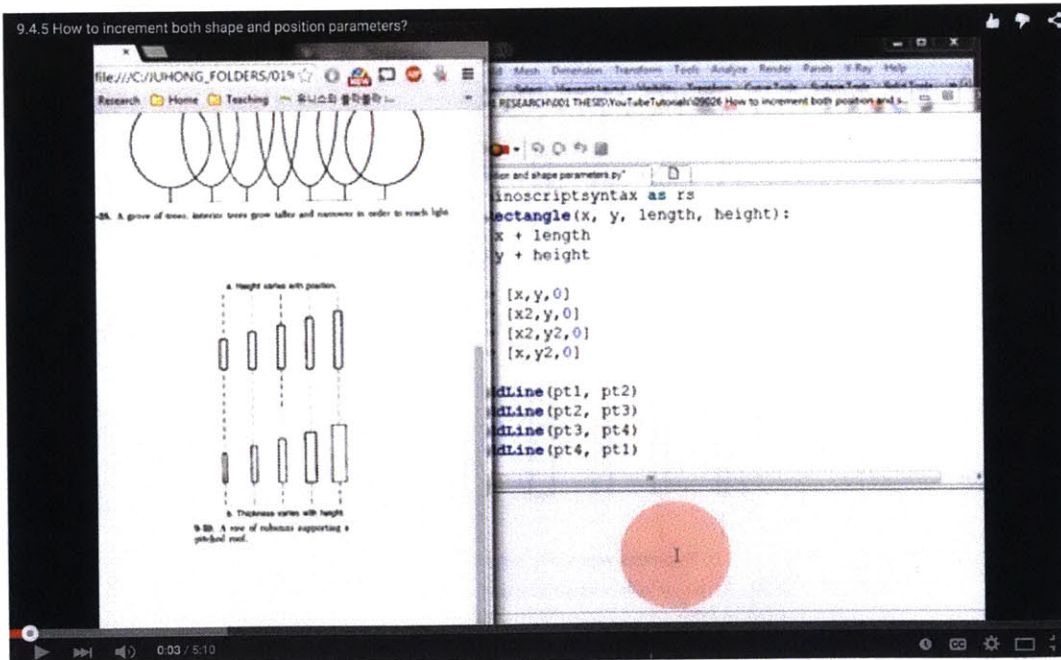
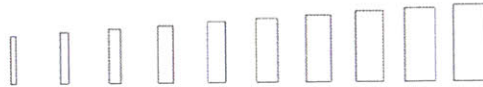
def drawColumn(x, y, height, thickness):
    x2 = x + thickness
    y2 = y + height
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawRowColumns(x, y, height, thickness,
                  spacing, height_inc, thickness_inc,
                  num_of_columns):

    count = range(num_of_columns)
    for i in count:
        drawColumn(x, y, height, thickness)
        x = x + spacing
        height = height + height_inc
        thickness = thickness + thickness_inc

drawRowColumns(0, 0, 3000, 300,
              3000, 200, 200,
              10)
```

RESULT



Module 87: 9.4 Incrementing 6

9. REPETITION

9.4 THE COMPOSITIONAL USES OF REPETITION

9.4.5 INCREMENTING BOTH POSITION AND SHAPE PARAMETERS

Here the parameters specify length, width and position of the lowest layer, and the amount by which the length of each successive layer is to be decremented. Iteration continues until the length of the layer becomes zero (fig. 9-30).

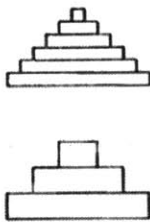
We considered, in chapter 7, a function to find the Midpoint of a line. We can use this function in the following iterative procedure that takes parameters specifying center point coordinates, an initial side length, and the total number of squares to be nested and produces compositions like the one shown in figure 9-31.

[sample code on the right side]

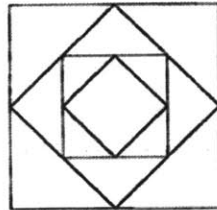
Notice what happens if we specify too many squares a square of side length 1 will be repeated until the loop terminates.

Another interesting way to look at this figure is as a square that is rotated through a quarter circle and scaled down by a factor of $\sqrt{2}$ at each iteration. That is, shape varies together with rotation, rather than with translation of the element, as in our earlier examples. The beautiful radial spiral shown in figure 9-32 is generated in a very similar way. Here a line with a fixed endpoint is rotated and lengthened by a fixed increment at each iteration.

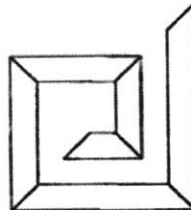
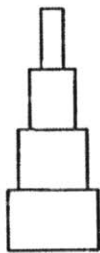
Finally, figure 9-33 illustrates a type of floor plan very much like that of Le Corbusier's famous Spiral Museum. The repeating element is a 45-degree trapezoid. Plans of this type can be generated by a procedure that lengthens the element by an appropriate increment, translates its origin forward, and rotates it through a half circle at each iteration.



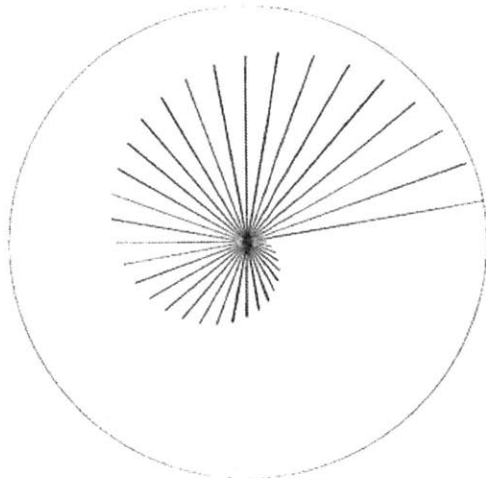
9-30. Stepped pyramids



9-31. A pattern of nested squares



9-33. A spiral formed by repeating instances of a 45-degree trapezoid.



9-32. A radial spiral.

CODE

```

import rhinoscriptsyntax as rs

def MidPoint(x1, x2):
    mid = (x1 + x2)/2
    return mid

def drawNestSquares(x_center, y_center,
                  length, num_squares):

    x1 = x_center - (length/2)
    y1 = y_center - (length/2)
    x2 = x1
    y2 = y1 + length
    x3 = x2 + length
    y3 = y2
    x4 = x3
    y4 = y1

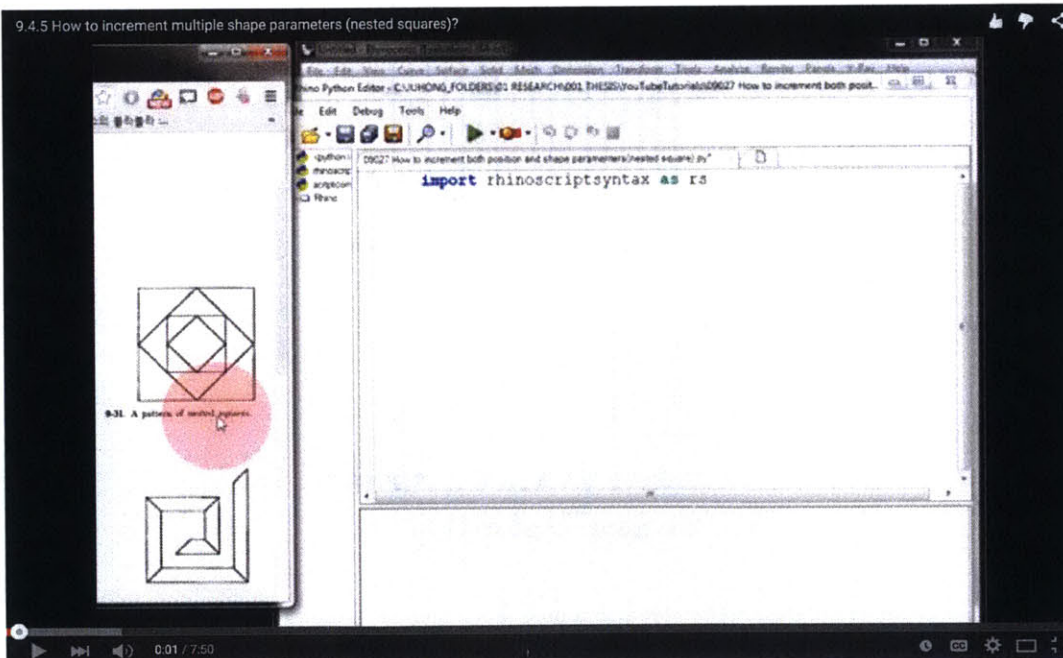
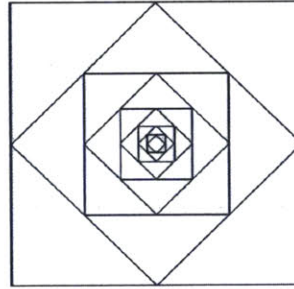
    count = range(num_squares)
    for i in count:
        pt1 = [x1, y1, 0]
        pt2 = [x2, y2, 0]
        pt3 = [x3, y3, 0]
        pt4 = [x4, y4, 0]
        pts = [pt1, pt2, pt3, pt4, pt1]
        rs.AddPolyline(pts)

        new_x1 = MidPoint(x1, x2)
        new_y1 = MidPoint(y1, y2)
        x2 = MidPoint(x2, x3)
        y2 = MidPoint(y2, y3)
        x3 = MidPoint(x3, x4)
        y3 = MidPoint(y3, y4)
        x4 = MidPoint(x4, x1)
        y4 = MidPoint(y4, y1)
        x1 = new_x1
        y1 = new_y1

drawNestSquares(0, 0,
               1000, 10)

```

RESULT



Module 88: 9.5 Math 1

9. REPETITION

9.5 CONSTRUCTING MATHEMATICAL PROGRESSIONS

So far we have focused mostly on repetitive compositions in which an element dimension is changed by a specified increment or decrement at each iteration, or in which a line of elements is lengthened by a fixed amount whenever a new element is added. In these compositions, dimensions form arithmetic progressions. Here is a procedure, for example, that generates a row of equally spaced vertical lines.

[sample code on the right side]

Let us assume that the initial value assigned to X is 10. At each iteration, a new value is assigned to X by the following statement

$$X = X + \text{INCREMENT}$$

Let us further assume that Increment is set to the value 10. Values taken by X will now be in the arithmetic progression

10, 20, 30, 40, 50, 60, . .

The result is the pattern shown in figure 9-34a.



a. X coordinates form an arithmetic progression.



b. X coordinates form a geometric progression.

9-34. The spacing of parallel lines.

CODE

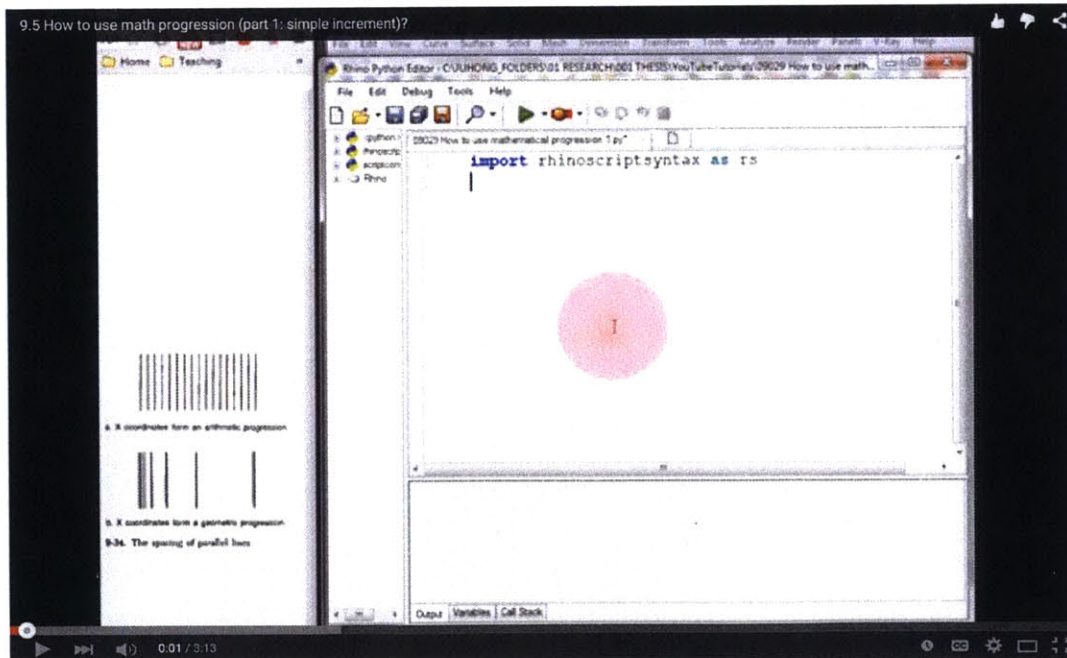
```
import rhinoscriptsyntax as rs

def drawLinesRatio(start_x, start_y,
                  length, ratio, number):
    x = start_x
    y = start_y + length
    count = range(number)

    for i in count:
        pt0 = [x, start_y, 0]
        pt1 = [x, y, 0]
        rs.AddLine(pt0, pt1)
        x = x + ratio

drawLinesRatio(2, 0, 100, 2, 10)
```

RESULT



Module 89: 9.5 Math 2

9. REPETITION

9.5 CONSTRUCTING MATHEMATICAL PROGRESSIONS

We do not need to restrict ourselves to arithmetic progressions, however. We can use geometric progressions of dimensions, in which each successive term is in a specified ratio to its predecessor. Consider, for example, this procedure to generate another row of vertical lines

[sample code on the right side]

Let us once again assume that the initial value assigned to X is 10. Now, at each iteration, a new value is assigned to X by the statement

```
X = X * RATIO
```

Let us assume, further, that Ratio is set to the value 2. Values taken by X will now be in the geometric progression

10, 20, 40, 80, 160, 320, . .

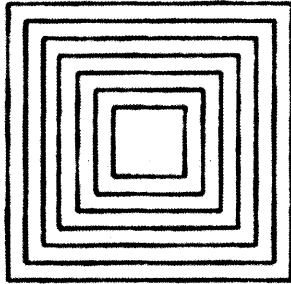
The result is the pattern shown in figure 9-34b.

In this example, Ratio is of type integer. But it is more generally useful to make it of type real. The assignment statement then becomes

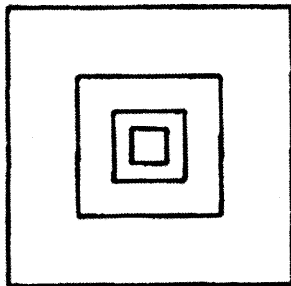
```
X = ROUND(X * RATIO)
```

The sizes of objects, too, may increase in either arithmetic or geometric progression. The next procedure, for example, generates concentric squares with side lengths forming an arithmetic progression (fig. 9-35a).

Nest squares can be modified slightly to generate concentric squares with side lengths forming a geometric progression (fig. 9-35b).



a. Side lengths in arithmetic progression.



b. Side lengths in geometric progression.

9-35. Compositions of concentric squares.

Module 90: 9.5 Math 3

9. REPETITION

9.5 CONSTRUCTING MATHEMATICAL PROGRESSIONS

9.5.1 EXPRESSIONS CONTAINING THE CONTROL VARIABLE

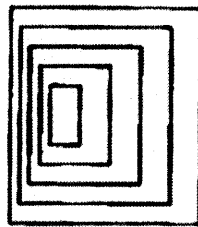
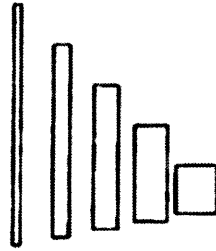
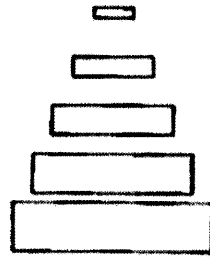
Where the values of more than one parameter of some object are changed within a loop, different expressions may assign the successive values to each one of these. This produces differential change. In the following procedure, for example, different expressions assign values to X, Y, Length, and Width of a rectangle.

[sample code on the right side]

Figure 9-36 illustrates some results produced by the execution of this procedure. In summary, the Python code to construct a sequence of instances of some type of object, with position or shape values forming arithmetic or geometric progressions, generally looks like this

Assign initial values to position and shape variables. Loop control statement (FOR or WHILE)

```
def loop_control_statement():  
    Draw instance with current position and shape values.  
    Assign new values to position and shape variables using  
    incrementing variables.
```



9-36. Some results produced by the procedure `Vary_rectangles`.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def drawRectangle(x1,y1,length,width):
    x2 = x1+length
    y2 = y1+width

    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]

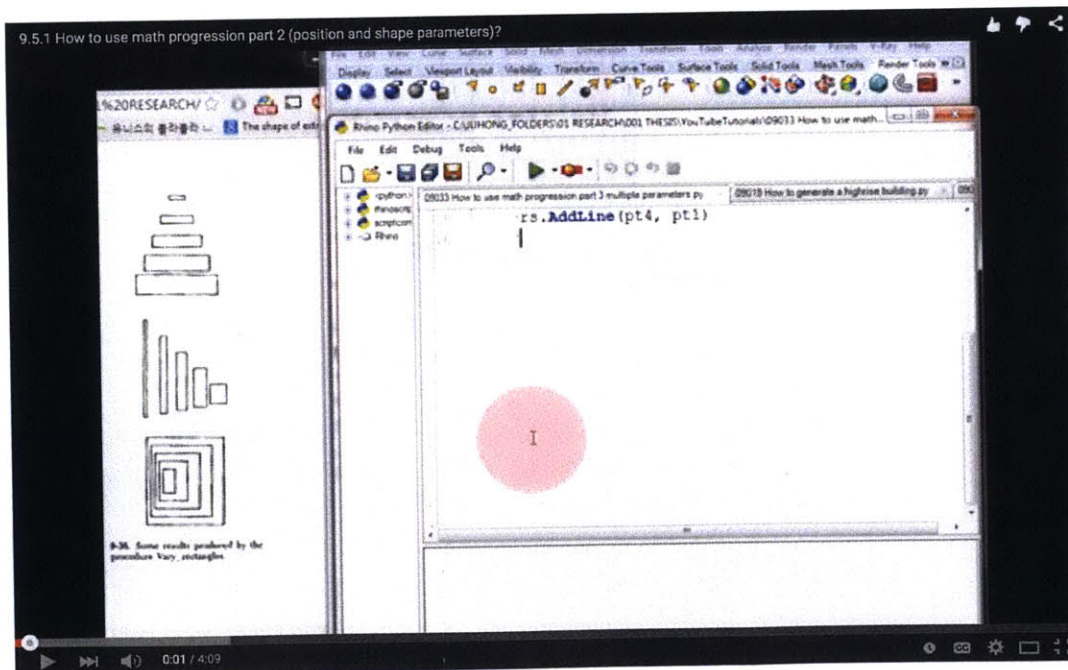
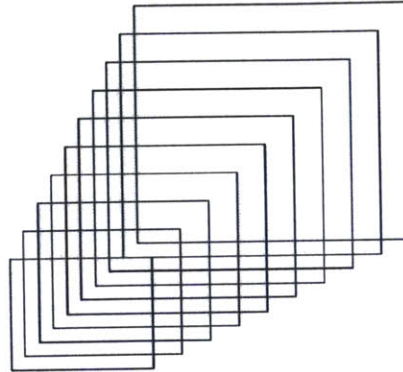
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawVaryingSquares(x, y, length, width,
                      inc_x, inc_y,
                      inc_length, inc_width,
                      number):

    count = range(number)
    for i in count:
        drawRectangle(x, y, length, width)
        x = x + inc_x
        y = y + inc_y
        length = length + inc_length
        width = width + inc_width

drawVaryingSquares(0,0, 1000,800, 100,100,
                  100,100, 10)
```

RESULT



Module 91: 9.5 Control 1

9. REPETITION

9.5 CONSTRUCTING MATHEMATICAL PROGRESSIONS

9.5.1 EXPRESSIONS CONTAINING THE CONTROL VARIABLE

When we employ an assignment of the form

$$X = X + \text{INCREMENT}$$

within a loop to construct an arithmetic progression, or an assignment of the form

$$X = X * \text{RATIO}$$

within a loop to construct a geometric progression, we make the position or the shape of each instance of the repeating element depend upon that of its predecessor. In other words, we directly specify how shape or position changes from one instance to the next. An alternative approach is to specify the nature of the change indirectly, by making the value of a position or shape variable depend upon the value of a for loop's control variable.

Where i is the control variable of a for loop, we can, for example, generate arithmetic progressions by employing assignments of the form

$$X = i * \text{INCREMENT}$$

We can make a value of a position or shape variable grow exponentially, rather than arithmetically or geometrically, by using assignments of the form.

$$X = \text{EXPONENT}(i, \text{POWER})$$

Where exponent is an integer function that raises i to the integer exponent specified by Power. For example, if i begins at 1, and Power is set at 2, successive values of X are

i	1	2	3	4
X	1	4	9	16

If Power is set at 3, then successive values of X are

i	1	2	3	4
X	1	8	27	64

The Exponent function can generate very large values even when i is small, so you must take care. If Power is 2, for example, Exponent will reach a value of 32,768 when i is 15 this is larger than the largest integer representable in many Python systems.

[sample codes on the right side]

Synthetic Tutor

CODE

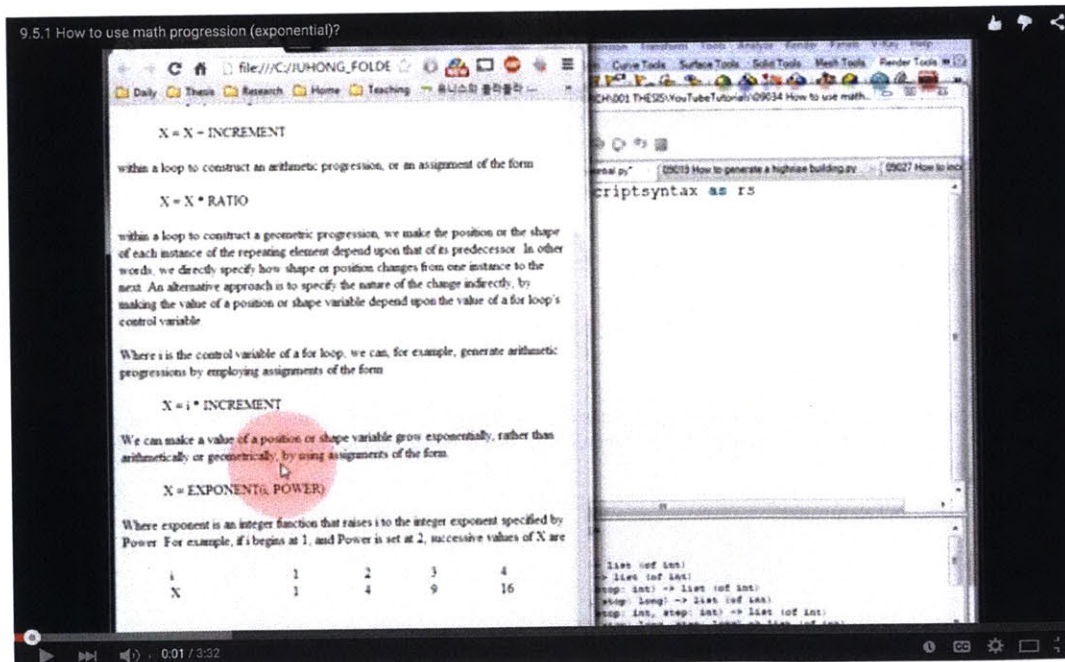
```
import rhinoscriptsyntax as rs

def getExponent (base, power):
    e = base
    n = power - 1
    count = range(n)
    for i in count:
        e = e * base
    exponent = e
    return exponent

exp = getExponent(2,5)
print exp
```

RESULT

32



Module 92: 9.5 Control 2

9. REPETITION

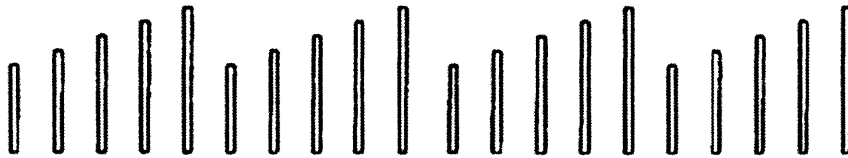
9.5 CONSTRUCTING MATHEMATICAL PROGRESSIONS

9.5.1 EXPRESSIONS CONTAINING THE CONTROL VARIABLE

Sometimes it is useful to use periodic functions of the control variable. Consider, for example, a row of columns to support a saw tooth roof (fig. 9- 37). The following procedure employs the % (mod) function to generate compositions of this type:

[sample codes on the right side]

Note that the % (mod) function is used to produce the remainder that results from dividing a dividend by a divisor. For example, the result of $5 \% (\text{mod}) 2$ is 1 the remainder when dividing 5 by 2 is 1. In our example, we used the % (mod) function to determine the start of a period.



9-37. A row of columns, in a sawtooth pattern, generated with the mod function.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

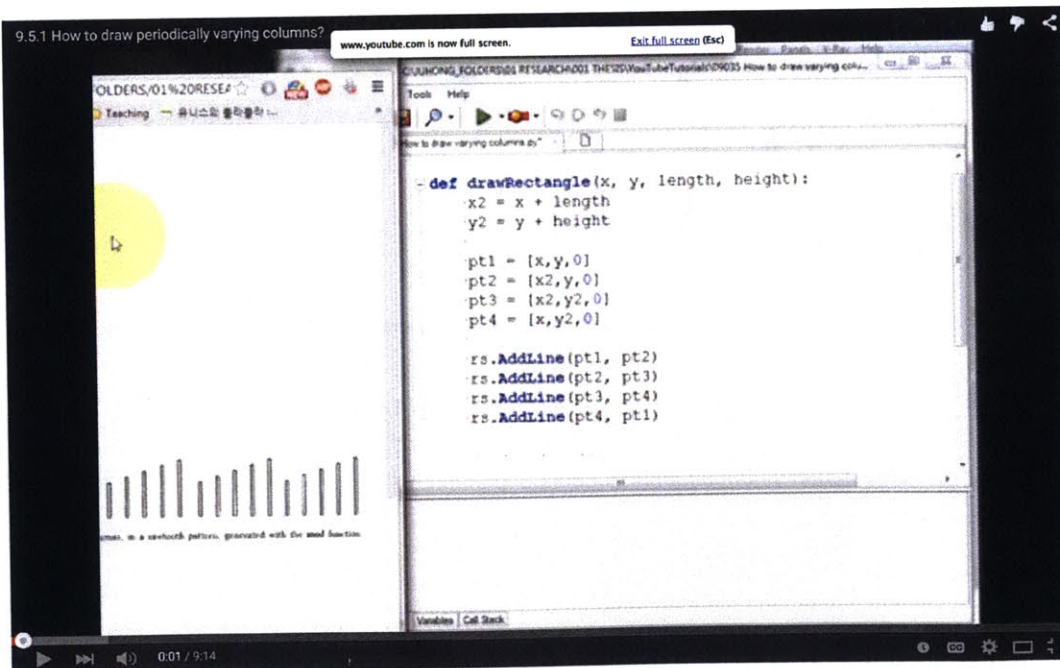
def drawColumns(x1, y1, height, thickness):
    x2 = x1 + thickness
    y2 = y1 + height
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawRowColumns(x_start, y, min_height,
                  thickness, height_inc, spacing,
                  period, num_columns):

    height = min_height
    x = x_start
    count = range(1, num_columns)
    for i in count:
        drawColumns(x, y, height, thickness)
        mod = i % period
        height = min_height + mod * height_inc
        x = x + spacing

drawRowColumns(0, 0, 1000, 200, 100, 500, 5, 20)
```

RESULT



Module 93: 9.5 Control 3

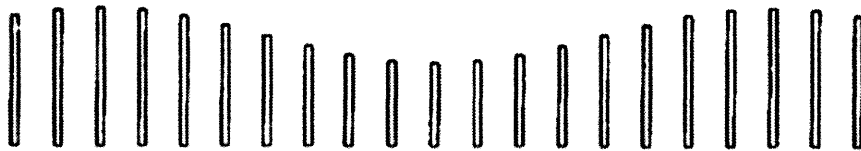
9. REPETITION

9.5 CONSTRUCTING MATHEMATICAL PROGRESSIONS

9.5.1 EXPRESSIONS CONTAINING THE CONTROL VARIABLE

In figure 9-38 the roof takes a sine curve rather than a saw tooth form. To generate this type of composition, we simply modify our procedure to employ the sin function

[sample codes on the right side]



9-38. A row of columns generated with the sin function.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

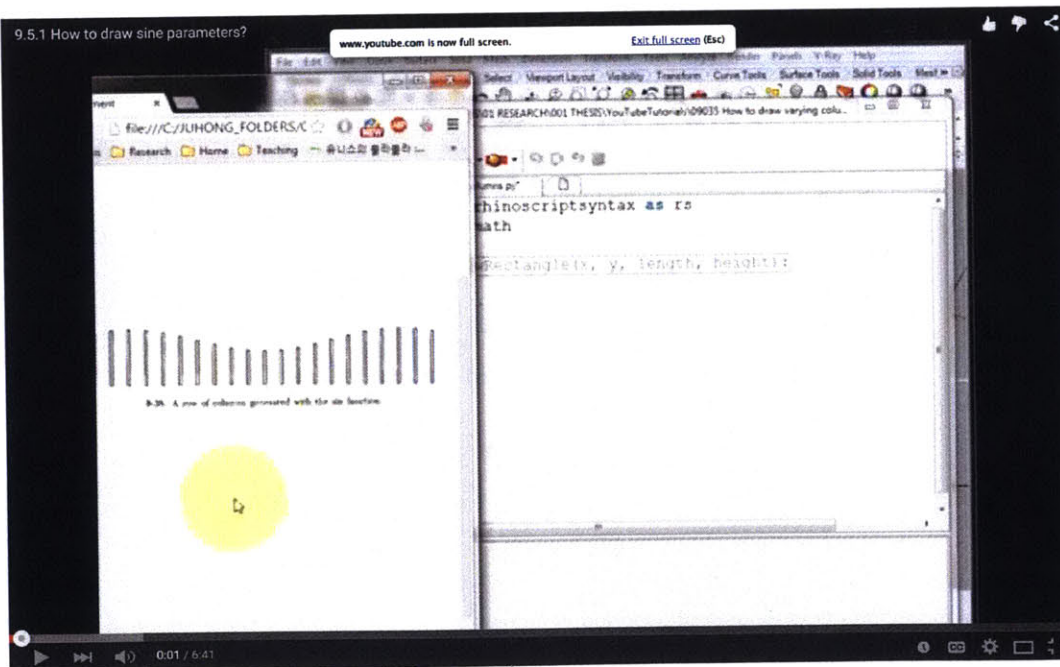
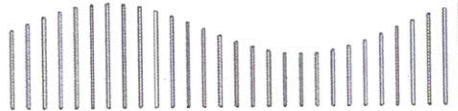
def drawColumns(x1, y1, height, thickness):
    x2 = x1 + thickness
    y2 = y1 + height
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawSineColumns(x_start, y,
                   min_height, thickness,
                   height_inc, spacing,
                   period, num_columns):

    radians = 0.01745
    angle = 180 / period * radians
    theta = 0
    x = x_start
    count = range(1, num_columns)
    for i in count:
        height = min_height + height_inc * math.sin(theta)
        drawColumns(x, y, height, thickness)
        x = x + spacing
        theta = i * angle

drawSineColumns(0, 0,
                3000, 100,
                1000, 600,
                12, 30)
```

RESULT



Module 94: 9.5 Control 4

9. REPETITION

9.5 CONSTRUCTING MATHEMATICAL PROGRESSIONS

9.5.1 EXPRESSIONS CONTAINING THE CONTROL VARIABLE

Our earlier procedure for drawing a row of squares can be modified to make X dependent on the control variable index (see `index` in the sample code). This allows us to draw portions of a row of squares by starting and finishing index at different values. The parameters to this modified procedure now include the initial and final values of `index`.

[sample codes on the right side]

Figure 9-39a shows the resulting row of squares when the initial and final values of `index` are set at 1 and 9 respectively. Figure 9-39b shows just a portion of the row when the loop is specified to go from 3 to 6.

Should you add an increment to the value of X at each iteration, or should you make X depend on the last value of the control variable `index` (as in these modified versions)? If integer arithmetic is used to calculate the value of X , the choice is one of style. If real arithmetic is used, however, there is an important practical difference - round off errors will cumulate as increments are added to X at each iteration, but they will not if X depends on the value of `index`. Just as a craftsman must watch out for cumulative error when fitting elements together in a row, we must do the same when writing programs to generate repetitive compositions.

A wide variety of expressions containing the control variable may be employed within loops to construct repetitive compositions. But the code always takes the same general form.

```
def general_form():
    for index in range(initial, final):
        Evaluate expressions containing - index - to assign values to position and shape variables
        Draw instance with current position and shape values
```

Synthetic Tutor

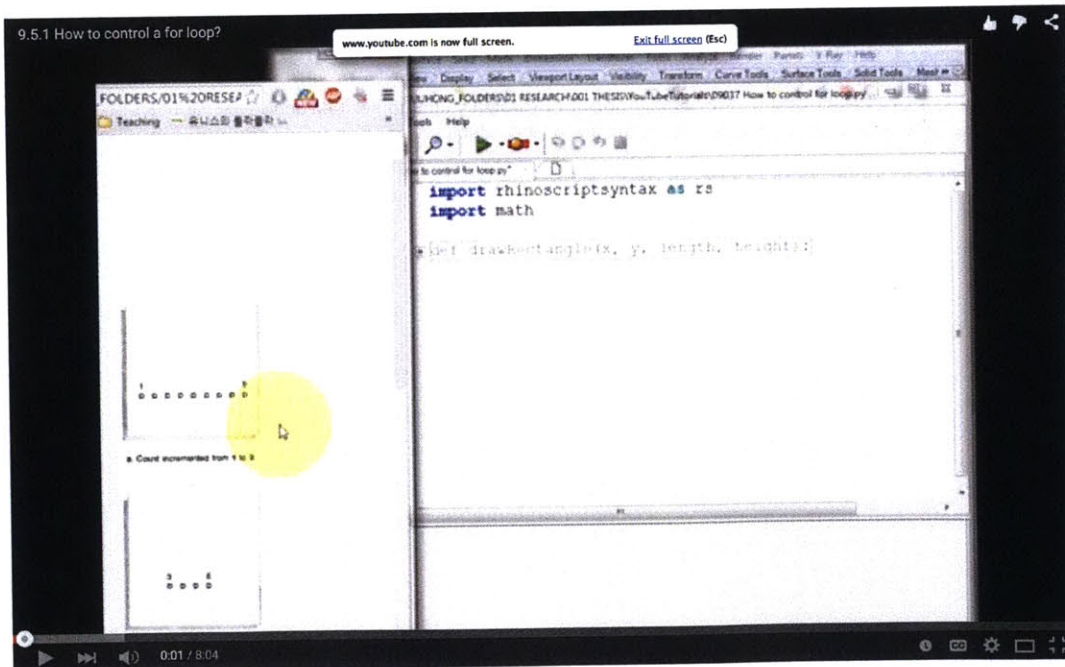
```
CODE
import rhinoscriptsyntax as rs
import math

def drawSquare(x_center,y_center,length):
    x1 = x_center - length/2
    y1 = y_center - length/2
    x2 = x1 + length
    y2 = y1 + length
    #draw square
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawRowSquares(x_initial,x_increment,y,length,initial,final):
    indices=range(initial,final)
    for index in indices:
        x=x_initial + (index-1)*x_increment
        drawSquare(x,y,length)

drawRowSquares(0,300,3000,100,3,20)
```

RESULT



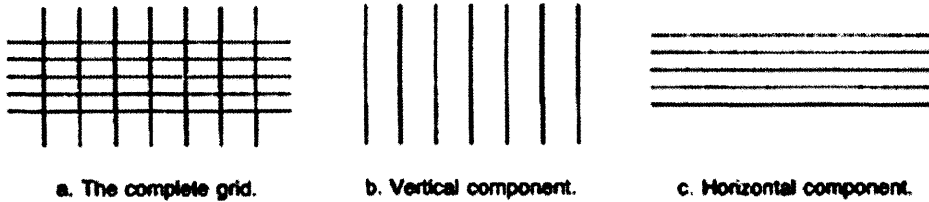
Module 95: 9.6 Loop 1

9.
REPETITION

9.6 SEQUENCES OF LOOPS

So far in this chapter we have considered repetitive compositions that are generated by single loops. But not all repetitive compositions can be produced in this way. The grid shown in figure 9-41a, for example, cannot. We can decompose it, however, into overlaid vertical and horizontal parallel lines (figs. 9-41b,c). Here is a simple procedure to generate parallel horizontal lines.

[sample codes on the right side]



9-41. The decomposition of a grid of lines.

Synthetic Tutor

CODE

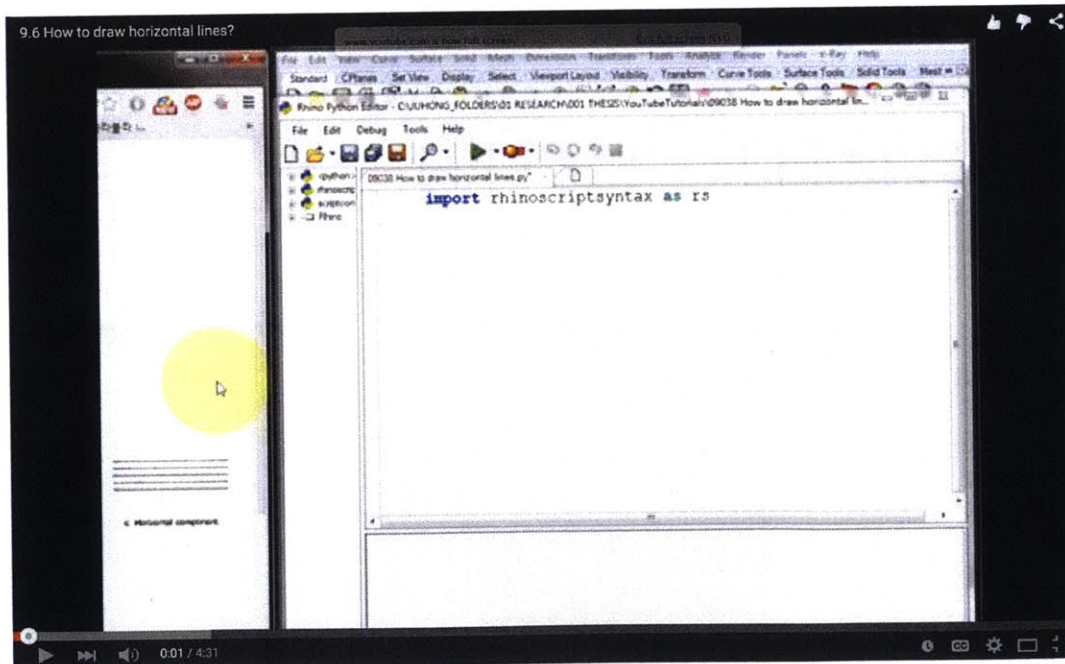
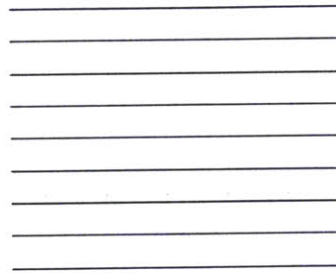
```
import rhinoscriptsyntax as rs

def drawHorizontals(start_x_h, start_y_h,
                    length_h, spacing_h, number_h):

    x = start_x_h + length_h
    y = start_y_h
    count = range(1, number_h)
    for i in count:
        pt1 = [start_x_h, y, 0]
        pt2 = [x, y, 0]
        rs.AddLine(pt1, pt2)
        y = y + spacing_h

drawHorizontals(0, 0,
                1000, 100, 10)
```

RESULT



Module 96: 9.6 Loop 2

9. REPETITION

9.6 SEQUENCES OF LOOPS

Now consider the simple frieze pattern shown in figure 9-42. It can be decomposed into several simpler repeating patterns, and each of these patterns can be generated by a procedure that executes a loop. The following program, which invokes the appropriate procedures one after the other, generates the complete frieze:

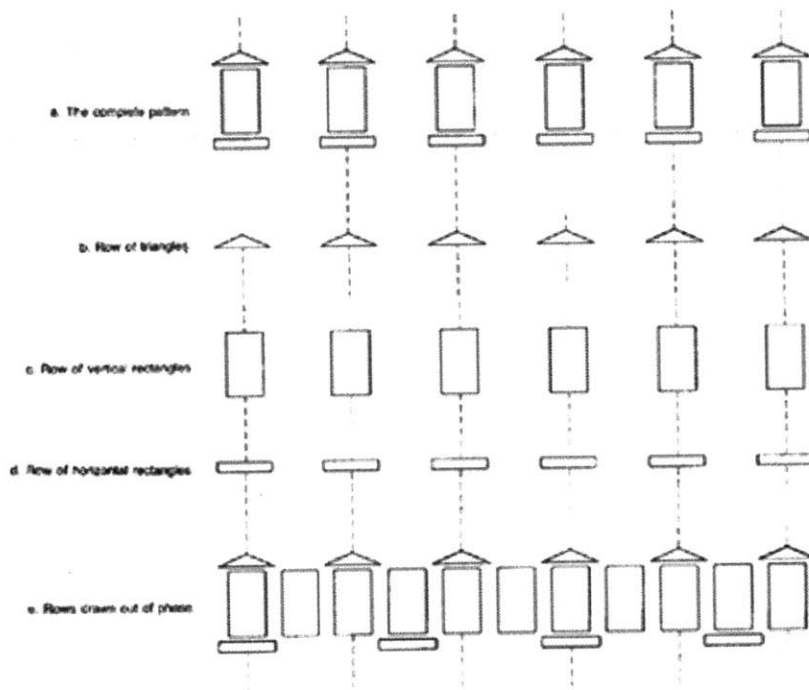
[sample codes on the right side]

Notice that the sequence in which the procedures are invoked controls the sequence in which the component patterns are actually drawn on the screen, but does not have any effect on the final drawing. If we were to change the sequence of procedure invocations in our program to the following,

```
ROW_TRIANGLES (50, 200, 60, 20, 60, 6)
ROW_RECTANGLES (60, 115, 40, 80, 80, 6)
ROW_RECTANGLES (50, 100, 60, 10, 60, 6)
```

The effect would be to reverse the order in which the component patterns are drawn on the screen, but the end result would be exactly the same.

In this program, we have taken care that the overlaid rows of triangles and rectangles are in phase with each other. By passing different values for the parameter Spacing into the procedures Row_rectangle and Row_triangle, we can also generate compositions in which the rows are out of phase (fig. 9-42e).



9-42. The decomposition of a frieze pattern.

CODE

```

import rhinoscriptyntax as rs

def drawRectangle(x1, y1, length, width):
    x2 = x1 + length
    y2 = y1 + width
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawTriangle(x1, y1, base, altitude):
    x2 = x1 + base
    x3 = x1 + base / 2
    y2 = y1 + altitude
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x3, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt1)

def drawRowRectangles(start_x, y, length, width,
                      spacing, number):
    x = start_x
    count = range(number)
    for i in count:
        drawRectangle(x, y, length, width)
        x = x + spacing + length

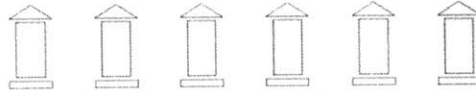
def drawRowTriangles(start_x, y, base,
                    altitude, spacing, number):
    x = start_x
    count = range(number)
    for i in count:
        drawTriangle(x, y, base, altitude)
        x = x + spacing + base

def frieze():
    drawRowRectangles(50, 100, 60, 10, 60, 6)
    drawRowRectangles(60, 115, 40, 80, 80, 6)
    drawRowTriangles(50, 200, 60, 20, 60, 6)

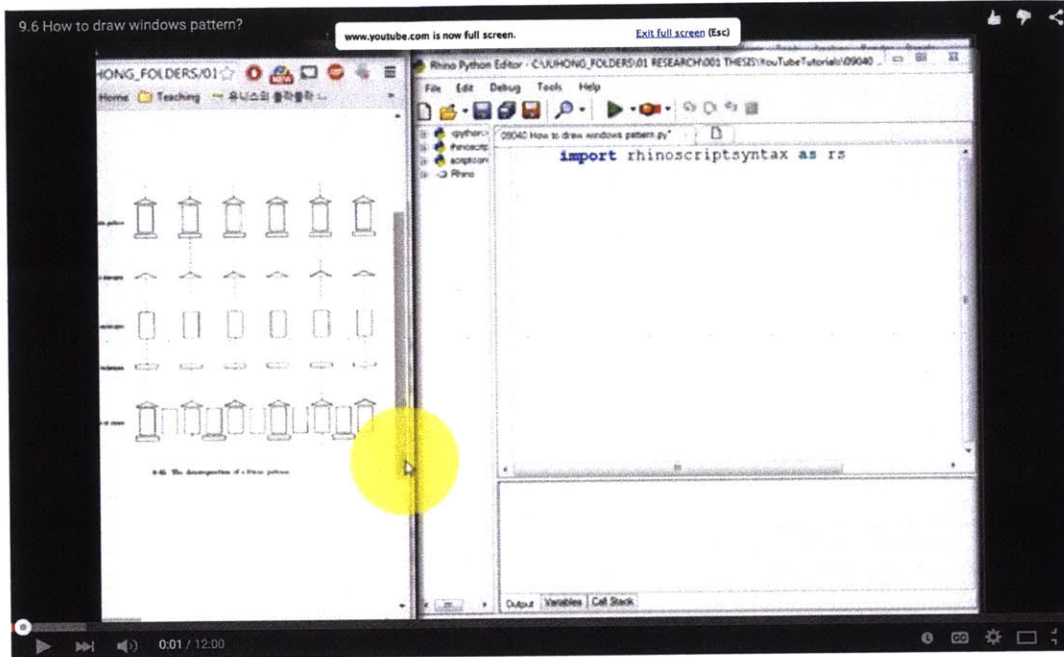
frieze()

```

RESULT



Synthetic Tutor



Module 97: 9.7 Nested 1

9.
REPETITION

9.7 NESTED LOOPS

Figure 9-43 illustrates a rectangular column grid, which often appears in plan drawings of buildings. To draw it, we might first write a procedure to generate a row of columns as follows

[sample codes on the right side]

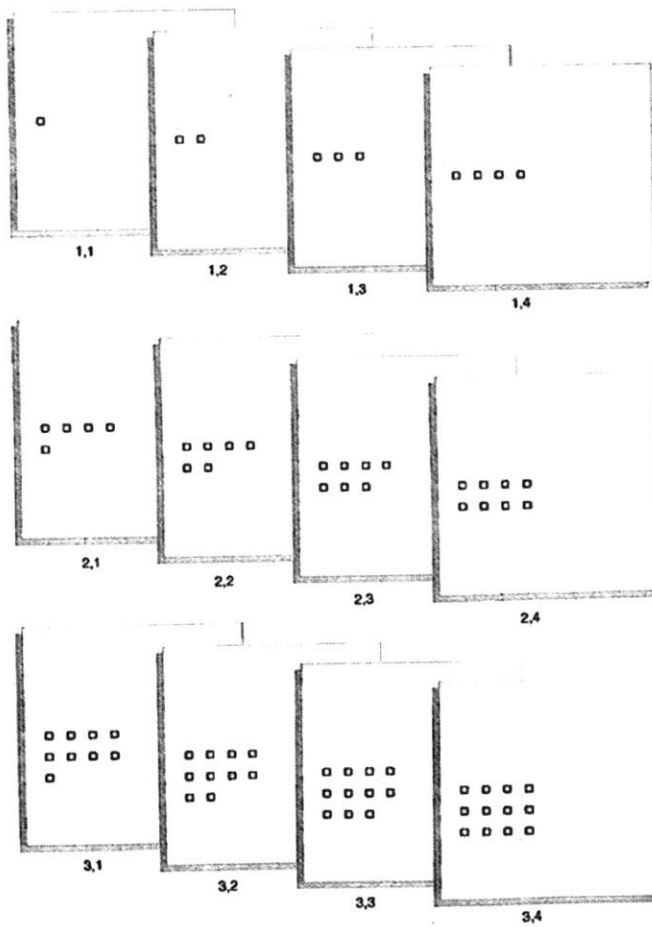
We might then invoke this procedure the requisite number of times.

```
ROW (100,500,30,100,4)  
ROW (100,400,30,100,4)  
ROW (100,300,30,100,4)
```

The effect is to draw rows, one after the other.



9-43. A grid of columns drawn in the screen coordinate system.



9-44. States of the drawing at each iteration, as it is generated by nested loops.

CODE

```
import rhinoscriptsyntax as rs

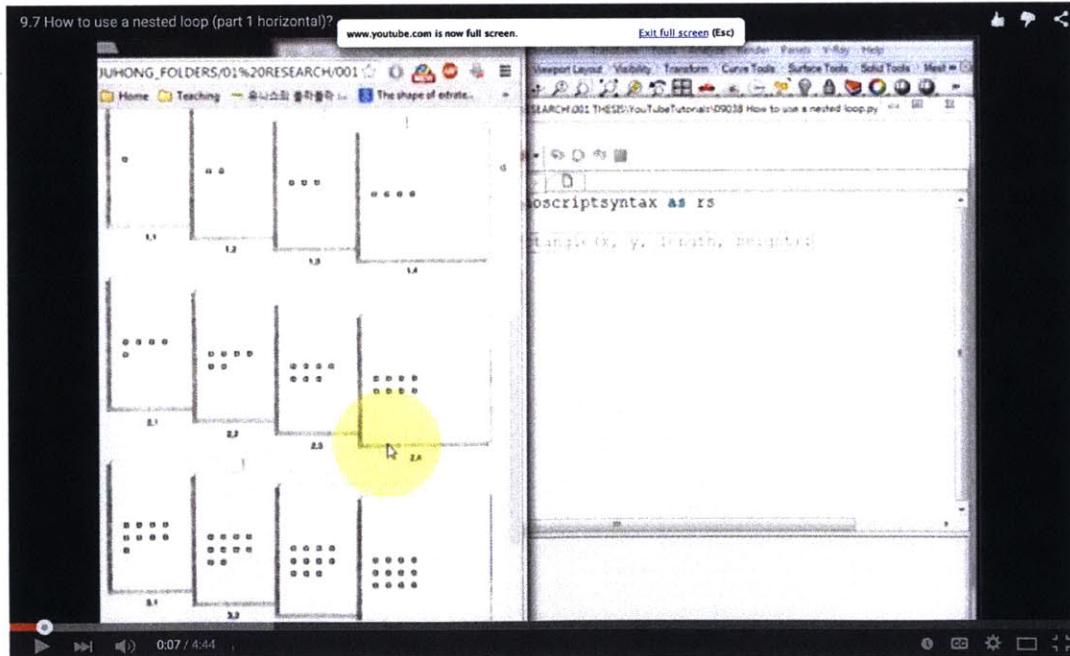
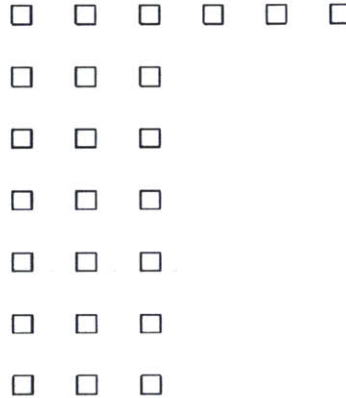
def drawColumn(x_center, y_center, diameter):
    x1 = x_center - diameter/2
    y1 = y_center - diameter/2
    x2 = x1 + diameter
    y2 = y1 + diameter
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawRow(initial_x, y, diameter,
            spacing, num_columns):

    x = initial_x
    count = range(num_columns)
    for i in count:
        drawColumn(x, y, diameter)
        x = x + spacing

drawRow(500, 600, 30, 100, 6)
```

RESULT



Module 98: 9.7 Nested 2

9. REPETITION

9.7 NESTED LOOPS

A more concise way to express this, however, is to invoke the Row procedure from within a loop.

The effect is to nest one loop within another. The inner loop repeatedly invokes the procedure Column to produce horizontal rows, and the outer loop repeatedly invokes the procedure Row to produce the grid. Figure 9-44 shows the step-by-step construction of the grid along with the values that the counters of the inner and outer loops have at each step. You should study this carefully before going on.

It is good practice to indent as shown this visually expresses the idea of nesting and clearly distinguishes the inner from the outer loop. This approach should be employed only when the inner loop does something very simple and straightforward.

CODE

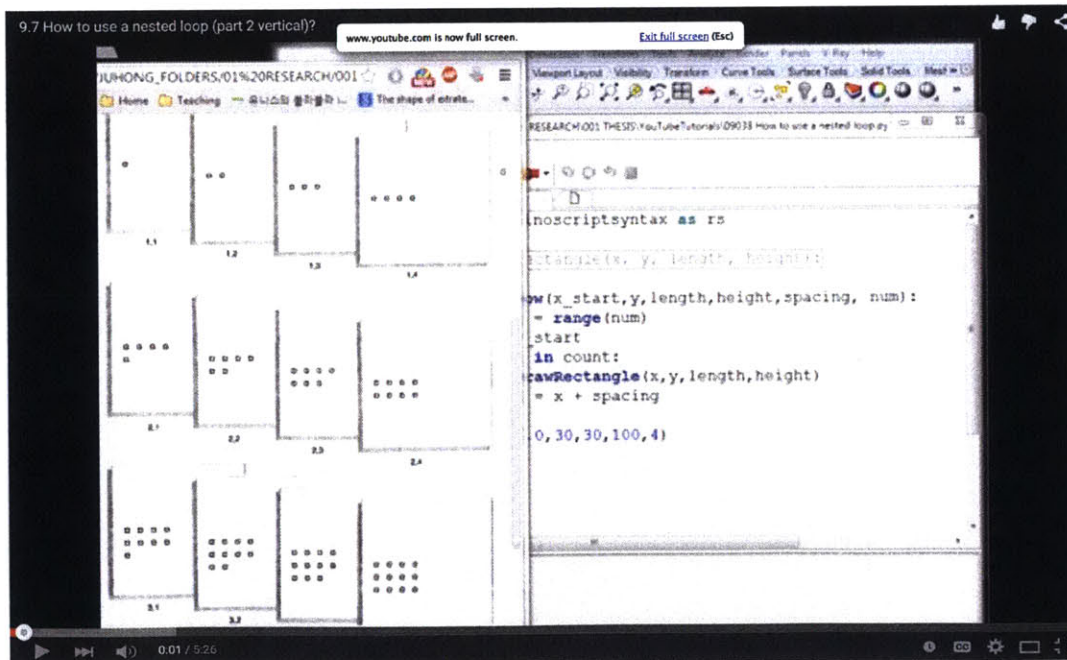
```
import rhinoscriptsyntax as rs

def drawColumn(x_center, y_center, diameter):
    x1 = x_center - diameter/2
    y1 = y_center - diameter/2
    x2 = x1 + diameter
    y2 = y1 + diameter
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def nestedLoop():
    x = 500
    y = 500
    count_rows = range(3)
    count_columns = range(6)
    for rowIndex in count_columns:
        for colIndex in count_rows:
            drawColumn(x, y, 30)
            x = x + 100
        y = y - 100
        x = 500

nestedLoop()
```

RESULT



Module 99: 9.7 Nested 3

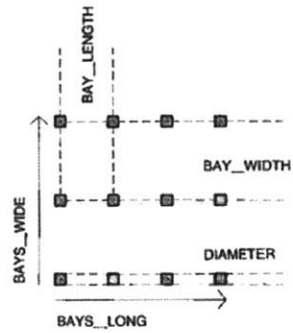
9. REPETITION

9.7 NESTED LOOPS

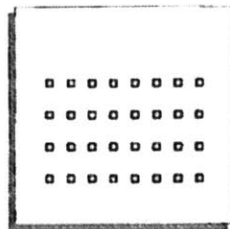
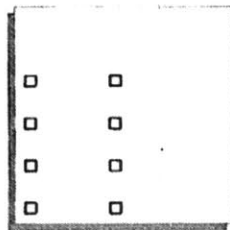
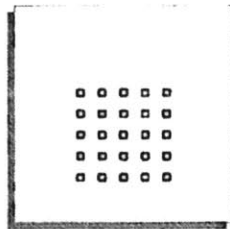
We are now in a position to write a very general program to draw rectangular column grids. The design variables are shown in figure 9-45a.

[sample codes on the right side]

This program first prompts for and reads in values for the design variables, then executes the loops to draw the required number of columns of the specified size and at the appropriate locations. Some of the many possible results are illustrated (fig. 9-45b).



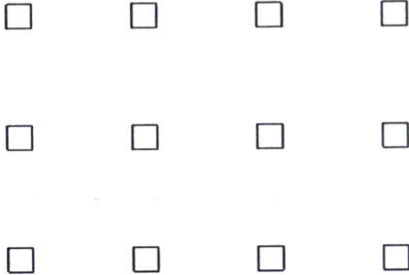
a. The type diagram.

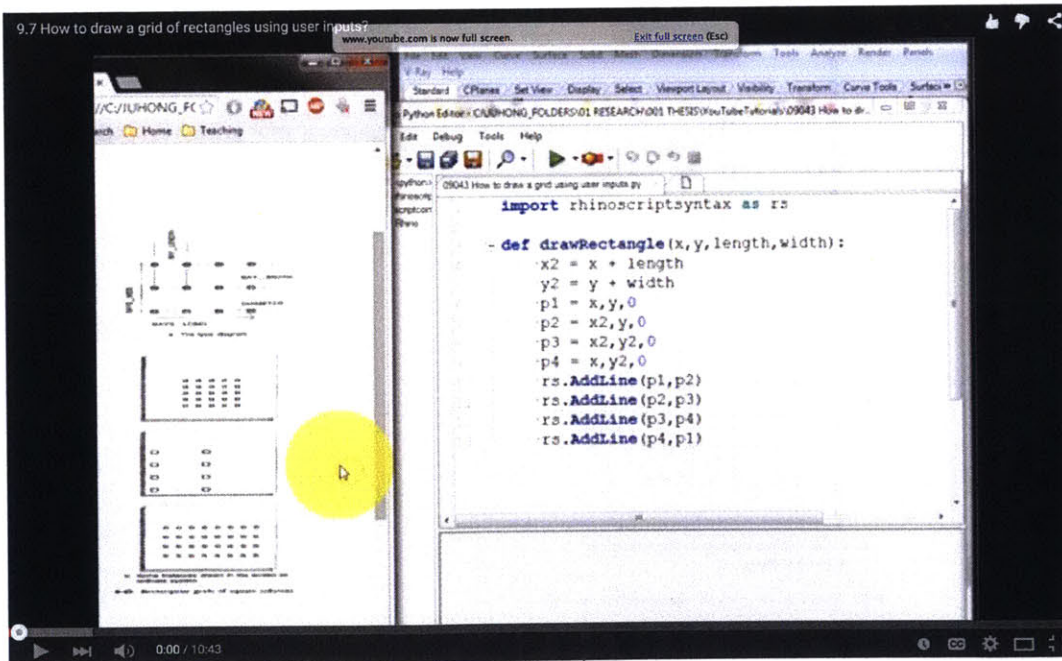


b. Some instances drawn in the screen coordinate system.

9-45. Rectangular grids of square columns.

Synthetic Tutor

CODE	RESULT
<pre>import rhinoscriptsyntax as rs def drawSquare(x_center, y_center, length): x1 = x_center - length/2 y1 = y_center - length/2 x2 = x1 + length y2 = y1 + length pt1 = [x1, y1, 0] pt2 = [x2, y1, 0] pt3 = [x2, y2, 0] pt4 = [x1, y2, 0] rs.AddLine(pt1, pt2) rs.AddLine(pt2, pt3) rs.AddLine(pt3, pt4) rs.AddLine(pt4, pt1) def grid(): x_center = rs.GetReal('enter x coordinate of center of grid', 0) y_center = rs.GetReal('enter y coordinate of center of grid', 0) diameter = rs.GetReal('enter column diameter', 40) bays_long = rs.GetInteger('enter number of bays long', 4) bays_wide = rs.GetInteger('enter number of bays wide', 3) bay_length = rs.GetReal('enter bay length', 200) bay_width = rs.GetReal('enter bay width', 200) print 'working...' y = y_center count_rows = range(1, bays_wide+1) count_columns = range(1, bays_long+1) for rowIndex in count_rows: x = x_center for columnIndex in count_columns: drawSquare(x, y, diameter) x = x + bay_length y = y + bay_width print 'finish drawing' grid()</pre>	



Module 100: Exercise 59.
REPETITION

9.10 EXERCISES

1. Assume a procedure to draw a square positioned by X, Y at the bottom left corner of side length Length. Execute the following code by hand. Record the value taken by each variable at each iteration, and draw the graphic output on grid paper. Write your values as comments in a python file.

a.

```
x = 100
y = 100
length = 10
```

```
for count in range( 1, 6 ):
    square( x, y, length )
    x = x + 2 * length
    length = length * 2
```

b.

```
x = 100
y = 100
length = 200
```

```
while ( length < 20 ):
    square( x, y, length )
    length = length / 2
    x = x + length
    y = y + length
```

c.

```
x_initial = 100
y_initial = 100
length_initial = 100
```

```
for count in range( 1, 4 ):
    x = x_initial
    y = y_initial
    length = length_initial
```

```
while ( length < 10 ):
    square( x, y, length )
    length = length - 10
    x = x + 5
    y = y + 5
    x_initial = x_initial + 100
    y_initial = y_initial + 100
```

Please upload your python file: No file chosen

2. Assuming a Rectangle procedure positioned by X, Y at the bottom-left corner with Length and Width defining horizontal and vertical dimensions respectively, draw the picture that would be produced by the following code.

Synthetic Tutor

```
import rhinoscriptsyntax as rs

x_initial = 100
y_initial = 100
length = 100
width = 30
nx = 8

y = y_initial

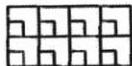
while ( nx > 0 ):
    x = x_initial

    for count_x in range( 1, nx ):
        rectangle( x, y, length, width )
        x = x + length

    nx = nx - 1
    x_initial = x_initial + length / 2
    y = y + width
```

Please upload your python file: No file chosen

3. Write procedures to draw the types of grids of squares shown in figure 9-49.



9-49. Some grids of squares.

Please upload your python file: No file chosen

4. Consider radial patterns of the type illustrated in figure 9-50. Write a general procedure to generate them.



9-50. A radial pattern.

Please upload your python file: No file chosen

5. Figure 9-51 illustrates some common types of building sections. Consider how each one should be parameterized, and write procedures to generate them.



9-51. Some schematic sections for multistory buildings.

Please upload your python file: No file chosen

6. There are various standard patterns for placement of the chords in a roof truss. Some of the most common are shown in figure 9-52. Choose one of these and, taking span, depth, and the number of subdivisions as parameters, write a procedure to lay out the chords correctly and draw the truss.

Module 101: 9.7 Nested 4

9. REPETITION

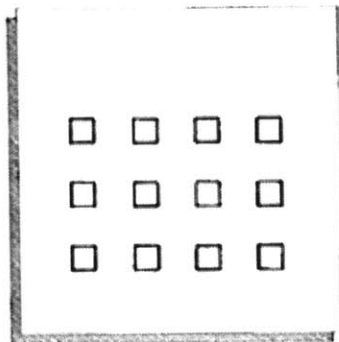
9.7 NESTED LOOPS

It is simple to modify this program to draw another type of column at the grid locations it is only necessary to substitute a different procedure for Square. The following procedure, for example, draws a cross-shaped column.

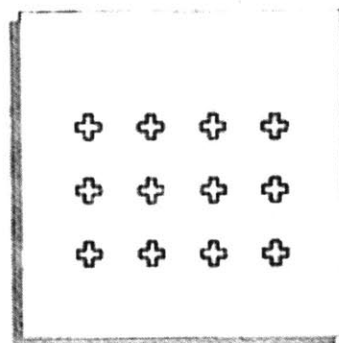
[sample codes on the right side]

If this is substituted in the column grid program, then the type of drawing shown in figure 9-46 is generated.

The object that is drawn at grid locations may itself be generated by a loop. Figure 9-47, for example, shows a grid of nested squares.

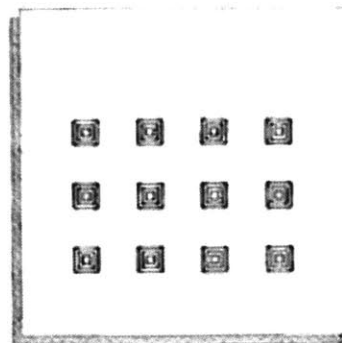


a. Grid of square columns.

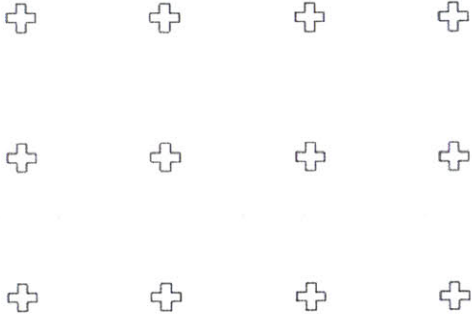


b. Cross-shaped columns substituted at the same locations.

9-46. Substitution of motifs within a loop.



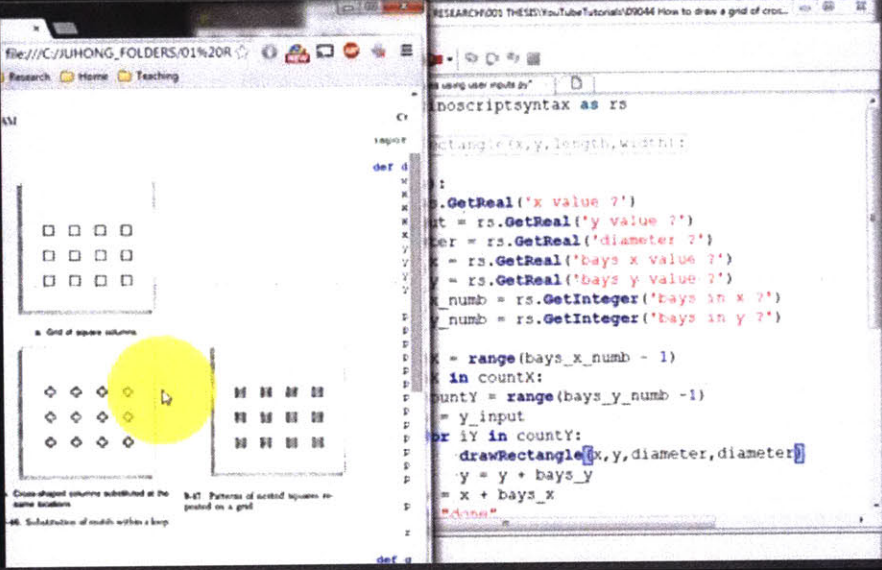
9-47. Patterns of nested squares repeated on a grid.

CODE	RESULT
<pre> import rhinoscriptsyntax as rs def drawCross(x,y,length): width = length / 3 x0 = x - (length / 2) x1 = x0 + width x2 = x0 + (2 * width) x3 = x0 + length y0 = y - (length / 2) y1 = y0 + width y2 = y0 + (2 * width) y3 = y0 + length pt1 = [x0, y1, 0] pt2 = [x0, y2, 0] pt3 = [x1, y2, 0] pt4 = [x1, y3, 0] pt5 = [x2, y3, 0] pt6 = [x2, y2, 0] pt7 = [x3, y2, 0] pt8 = [x3, y1, 0] pt9 = [x2, y1, 0] pt10 = [x2, y0, 0] pt11 = [x1, y0, 0] pt12 = [x1, y1, 0] pt13 = [x0, y1, 0] pts = [pt1, pt2, pt3, pt4, pt5, pt6, pt7, pt8, pt9, pt10, pt11, pt12, pt13] rs.AddPolyline(pts) def grid (): x_center = rs.GetReal('enter_x_coordinate of center of grid',0) y_center = rs.GetReal('enter_y_coordinate of center of grid',0) diameter = rs.GetReal('enter column diameter',40) bays_long = rs.GetInteger('enter number of bays long',4) bays_wide = rs.GetInteger('enter number of bays wide',3) bay_length = rs.GetReal('enter bay length',200) bay_width = rs.GetReal('enter bay width',200) print 'working...' y = y_center count_rows=range(1, bays_wide+1) count_columns=range(1, bays_long+1) for rowIndex in count_rows: x = x_center for columnIndex in count_columns: drawCross(x, y, diameter) x = x + bay_length y = y + bay_width print 'finish drawing' grid() </pre>	

Synthetic Tutor

9.7 How to draw a grid of crosses?

www.youtube.com is now full screen. Exit full screen (Esc)



```
def draw_grid_of_crosses():  
    rs = RaschSyntax()  
    rs.draw_rectangle(x, y, length, width):  
    def draw_cross(x, y, diameter):  
        rs.draw_circle(x, y, diameter)  
        rs.draw_circle(x, y, diameter)  
    x = rs.get_real('x value ?')  
    y = rs.get_real('y value ?')  
    diameter = rs.get_real('diameter ?')  
    bays_x = rs.get_real('bays x value ?')  
    bays_y = rs.get_real('bays y value ?')  
    x_num = rs.get_integer('bays in x ?')  
    y_num = rs.get_integer('bays in y ?')  
    x = range(bays_x - 1)  
    for countX in range(bays_x - 1):  
        countY = range(bays_y - 1)  
        y = y_input  
        for iY in countY:  
            draw_rectangle(x, y, diameter, diameter)  
            y = y + bays_y  
            x = x + bays_x  
    done
```

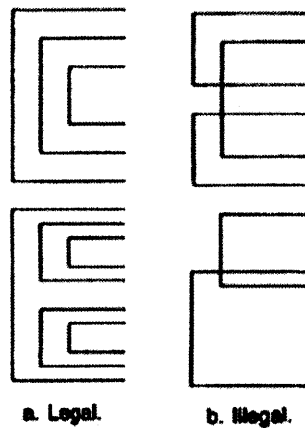
Module 102: 9.7 Nested 59.
REPETITION

9.7 NESTED LOOPS

The inner loop of this program invokes the procedure Square to generate nested squares. The middle loop invokes the procedure Nested_squares to generate rows of nested squares. Then the outer loop invokes the procedure Row to generate the grid. Alternatively (but less clearly), the nesting could be expressed using begin and end statements and indenting as follows.

[sample codes on the right side]

In general, you can nest loops as deeply as you want to generate repetitive compositions such as regular rows, stacks, nested objects, and combinations of these. The only restrictions are that successive loops cannot overlap, and an inner loop must always be contained completely within an outer loop. The syntactic conventions of Python make it difficult to write code that violates these rules. Figure 9-48a illustrates some examples of legal nesting, whereas figure 9-48b illustrates some examples of illegal nesting. They are illegal because they could result in ambiguous values of control variables or expressions.



9-48. Nesting of loops.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs

def drawSquare(x_center, y_center, diameter):
    x1 = x_center - diameter/2
    y1 = y_center - diameter/2
    x2 = x1 + diameter
    y2 = y1 + diameter
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def drawNestedSquares(x,y,diameter,increment,nest):
    indices=range(nest)
    for index in indices:
        drawSquare(x,y,diameter)
        diameter=diameter-increment

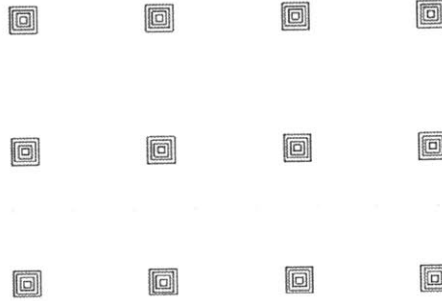
def grid():
    x_center = rs.GetReal('enter x coordinate of center of grid',0)
    y_center = rs.GetReal('enter y coordinate of center of grid',0)
    diameter = rs.GetReal('enter column diameter',40)
    bays_long = rs.GetInteger('enter number of bays long',4)
    bays_wide = rs.GetInteger('enter number of bays wide',3)
    bay_length = rs.GetReal('enter bay length',200)
    bay_width = rs.GetReal('enter bay width',200)
    print 'working...'

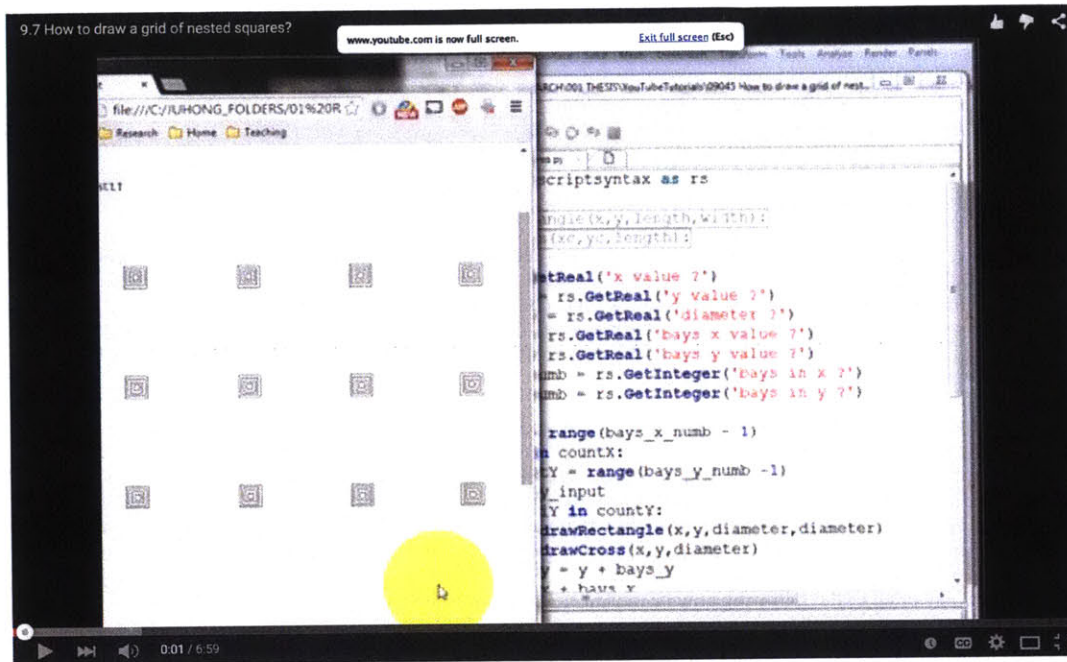
    y = y_center
    count_rows=range(1, bays_wide+1)
    count_columns=range(1, bays_long+1)
    for rowIndex in count_rows:
        x = x_center
        for columnIndex in count_columns:
            drawNestedSquares(x, y, diameter, 10, 4)
            x = x + bay_length
            y = y + bay_width

    print 'finish drawing'

grid()
```

RESULT





Module 103: 9.8 Elements

9. REPETITION

9.8 VOCABULARY ELEMENTS WITH REPETITION PARAMETERS

It should now be obvious that the use of loops allows us to declare graphic vocabulary elements that have not only position and shape parameters, but also repetition parameters. That is, the amount of repetition may vary from instance to instance. Consider our procedure Row, to generate a row of square columns in the program Grid. The position parameters are the coordinates of the starting point. There is a shape parameter controlling the diameter of the column. Then there is a repetition parameter-Num_columns-controlling the number of columns. Finally, there is a parameter controlling the spacing of columns.

Where for loops are employed, repetition parameters may specify initial and final values for the control variable. Where while or repeat until loops are employed, repetition parameters may specify values for variables that appear in the control expressions. Where an arithmetic progression is employed, the initial value may be treated as a design variable and so may the value of the increment. Similarly, the initial values and ratios that define geometric progressions may be treated as design variables. Where functions of the control variable are employed, the values of any coefficients may also be treated as design variables.

In summary, any or all of the following may be treated as design variables and represented as parameters of procedures that generate repetitive compositions

- . position of the entire composition
- . shape variables of the repeating element
- . initial and final values of control variables in for loops
- . values for variables in control expressions for while and repeat until loops
- . initial values and increments for arithmetic progressions
- . initial values and ratios for geometric progressions
- . values for coefficients appearing in functions of control variables

Module 104: 9.9 Summary

9. REPETITION

9.9 SUMMARY

In this chapter we have made use of the knowledge that a composition may be generated not only from a limited vocabulary, but that there may also be regular repetition of vocabulary elements. Where this is the case, we do not need to explicitly specify each instance of the repeating element. It suffices to specify what element is repeated, where the repetition starts, what changes at each step, and where the repetition ends. We can do this by writing a loop from within which a procedure to generate the repeating element is invoked.

We have seen that Python provides two kinds of loops the counted for loop, and the while loop that is controlled by the evaluation of expressions. These can be used to write concise programs that generate simple repetitive compositions. More complicated repetitive compositions can be generated by writing loops that execute one after the other, or that are nested one within the other.

The first step in writing a program to generate a repetitive composition is to identify the repeating graphic element and declare an appropriately parameterized procedure to generate it. This procedure can then be invoked from within a loop (or a structure of nested loops). Consideration of how you want to control the way that the loop begins and ends and the number of iterations should indicate whether a for or a while is most appropriate. You may want the successive values of the repeating element's shape and position parameters to form arithmetic or geometric progressions.

In this case, you can initialize values outside the loop and calculate new values from old values at each iteration. Alternatively, you may want successive values of the shape and position parameters to be functions of the control variable. In this case, you must evaluate these functions at each iteration.

Whereas Python provides concise, convenient ways to express rules of repetition, establishment of those rules is up to the designer, who must decide if there are practical or aesthetic reasons to repeat some elements within a composition, and if so, how the repeating pattern should be started and terminated, what shape or position properties should change at each iteration, and what should remain the same.

Module 105: 10.1 Curves

10. CURVES

We have taken a straight line segment to be the primitive element of graphic composition we have considered drawings as sets of straight line segments in the picture plane and we have treated composition as the relation of straight line segments by connection, angle, ratio, and repetition. But this does not seem an adequate framework for analysis of a composition like Botticelli's Birth of Venus (fig. 10-1) the only straight line is the horizon (parallel to the x axis), and all the others are curves. Different types of curves are used to construct the shell, waves, hair, drapery, foliage, and outlines of human bodies, and the composition relates curves to each other and to the horizon line. Sharp inflections contrast with shallow sweeps curves accompany each other or diverge in the shell, there are waves and hair repetitions with variation from instance to instance.

Ezra Pound once wrote, with considerable scorn "Give your draughtsman sixty-four stencils of 'Botticelli's most usual curves.' And will he make you a masterpiece?" We cannot guarantee the masterpiece, but we can approach the composition of curves by considering different types of curves, parameterized procedures to generate these types, and visually important relations between curves of various types.

10.1 THE APPROXIMATION OF CURVES BY STRAIGHT SEGMENTS

Consider the smooth curve shown in figure 10-2a. Since we have no primitive to draw a line of continuously varying slope, we cannot write a Python program to replicate it exactly. But we can approximate it, more or less closely, with sequences of vectors generated by draw commands. Here is a procedure that produces the coarse approximation illustrated in figure 10-2b

And here is a procedure that produces the finer approximation illustrated in figure 10-2c:

[sample codes on the right side]

We can ultimately make the approximation as close as the resolution of our display device allows. How many points are needed for an adequate approximation? The answer will depend on the amount of information needed for recognition of the end of the curve (fig. 10-3), the aesthetic importance of smooth curves, and the available resolution. You should experiment.

Fortunately, many of the types of curves of most interest to us—circles, arcs of circles, ellipses, parabolas, and so on—do not need to be described in this cumbersome, point-by-point fashion. It is possible to write loops that repeatedly invoke functions to generate X- and Y-coordinate values of successive, closely spaced points on the curve. These values can then be used in draw commands to generate a vector approximation of the curve. In this chapter, we shall explore this approach to generating the kinds of curves that we most often need in drawings.



10-1. Sandro Botticelli, *The Birth of Venus* c. 1480, Uffizi Gallery, Florence.



a. A smooth curve.



b. A coarse approximation with straight line segments.



c. A finer approximation with straight line segments.

10-2. Approximations of a curve.



10-3. The curve of a human profile represented by successively greater amounts of information.

Synthetic Tutor

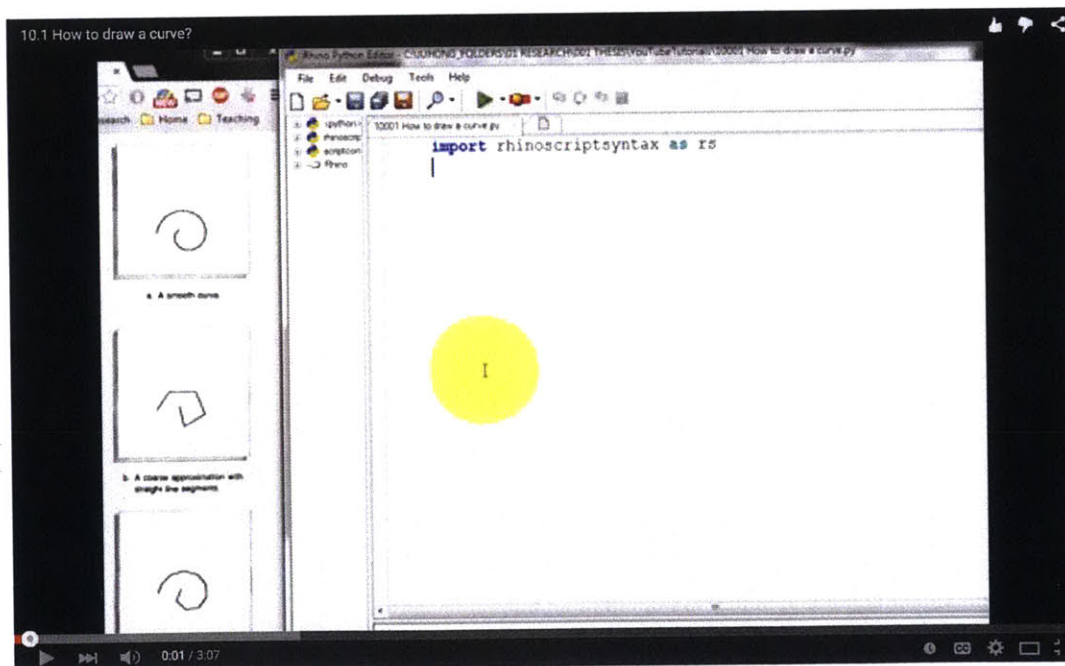
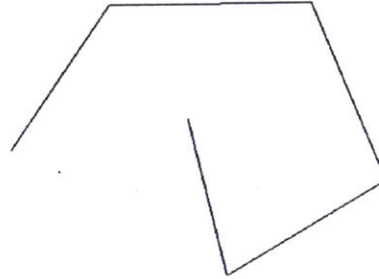
CODE

```
import rhinoscriptsyntax as rs

def drawCurve():
    pt1 = [481, 354, 0]
    pt2 = [518, 195, 0]
    pt3 = [677, 290, 0]
    pt4 = [605, 472, 0]
    pt5 = [404, 471, 0]
    pt6 = [306, 324, 0]
    points = [pt1, pt2, pt3, pt4, pt5, pt6]
    rs.AddPolyline(points)

drawCurve()
```

RESULT



Module 106: 10.2 Function X110.
CURVES

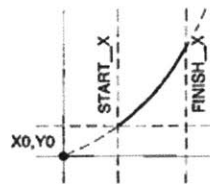
10.2 FUNCTIONS OF X

In many cases, the Y coordinate of a point on a curve can simply be represented as a function of the X coordinate. A sequence of points can then be generated by a loop that increments the value of X through some range and invokes the appropriate function at each iteration to return the corresponding Y value. The following procedure uses the standard Python function `math.sqrt()` (which computes the square of a number) in this way to generate a parabolic curve

[sample codes on the right side]

Notice how this procedure is parameterized. There are parameters that specify the starting X coordinate, the finishing X coordinate, the location of the curve in the screen coordinate system, and the number of segments. Some examples of output are given in figure 10-4.

A very broad class of curves, which includes the straight line and the parabola, is the class of polynomial curves. A straight line is a polynomial of degree one, and a straight line composed of a specified number of segments is generated by this procedure (figure 10-5a)

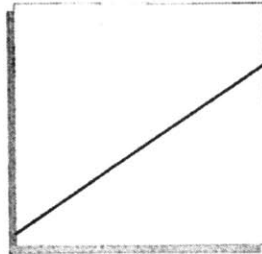


a. Type diagram.

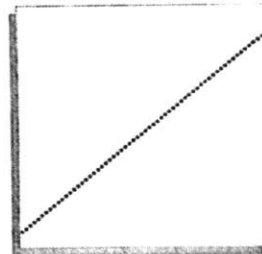


b. Examples of instances.

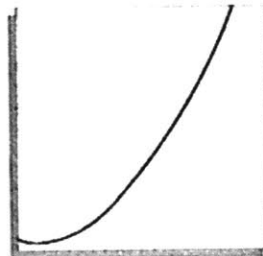
10-4. A parabola generated using the function `sqr`.



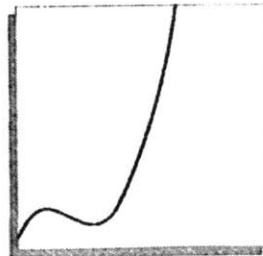
a. Straight line (degree one).



b. Dotted straight line.



c. Parabola (degree two).



d. Cubic (degree three).

10-5. Type diagrams for polynomials.

CODE

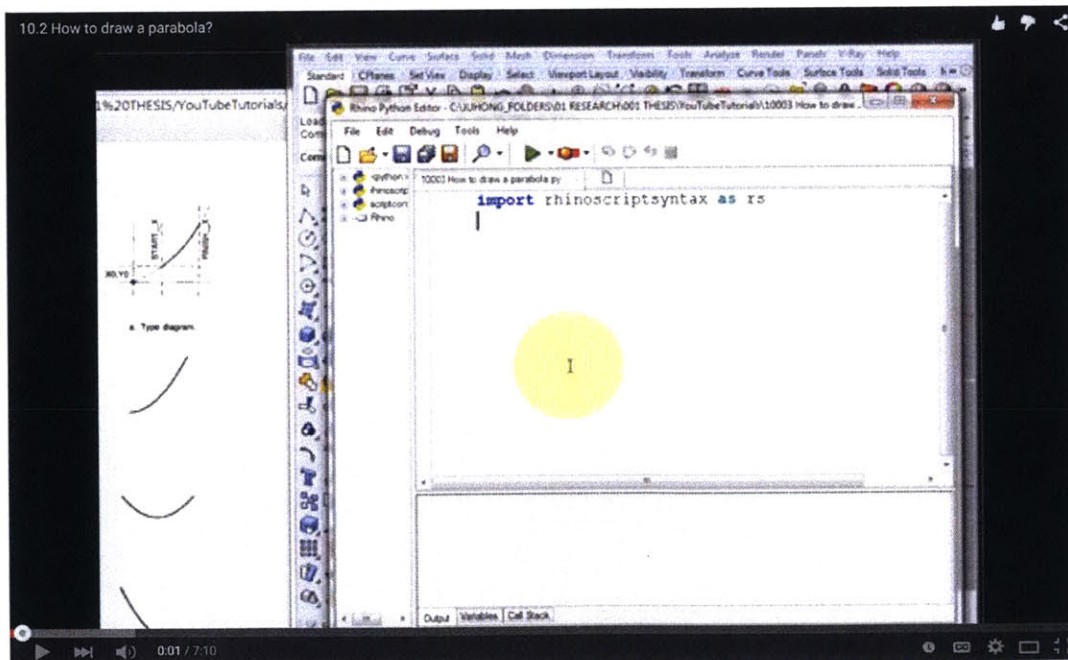
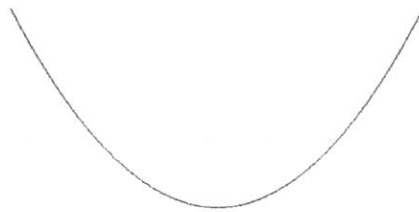
```
import rhinoscriptsyntax as rs

def parabola(x0, y0,
            start_x, finish_x,
            n_segments):

    x_increment = (finish_x - start_x) / n_segments
    x = start_x
    y = x * x
    pt0 = [x, y, 0]
    count = range(n_segments)
    for i in count:
        x = x + x_increment
        y = x * x
        pt1 = [x + x0, y + y0, 0]
        rs.AddLine(pt0, pt1)
        pt0 = pt1

parabola(0,0,-1,1,100)
```

RESULT



Module 107: 10.2 Function X2

10. CURVES

10.2 FUNCTIONS OF X

This procedure is pointless (or rather, generates too many points) as it stands, because we could simply draw a vector from the first point to the last without generating those between. But it can easily be converted into a useful procedure to draw a dotted line (fig. 10-5b), as follows

[sample codes on the right side]

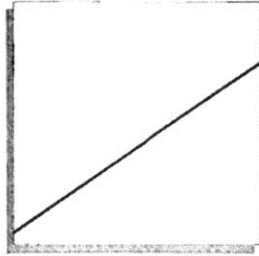
A parabola is a polynomial of degree two and is generated by the following procedure (fig. 10-5c)

[sample codes on the right side]

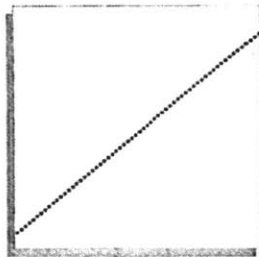
Here the value of Y is assigned by the statement

$$Y = A0 + A1 * X + A2 * \text{EXPONENT}(X,2)$$

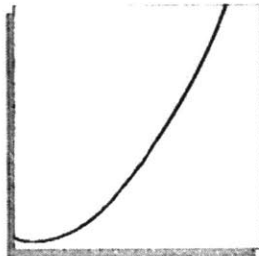
Exponent is the exponentiation function that was introduced in chapter 9. The coefficients A0, A1, and A2 determine the location in the coordinate system and the precise shape of an instance.



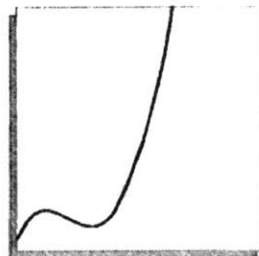
a. Straight line (degree one).



b. Dotted straight line.



c. Parabola (degree two).



d. Cubic (degree three).

10-5. Type diagrams for polynomials.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs

def exponent(base, power):
    e = base
    count = range(1, power)
    for i in count:
        e = e * base
    return e

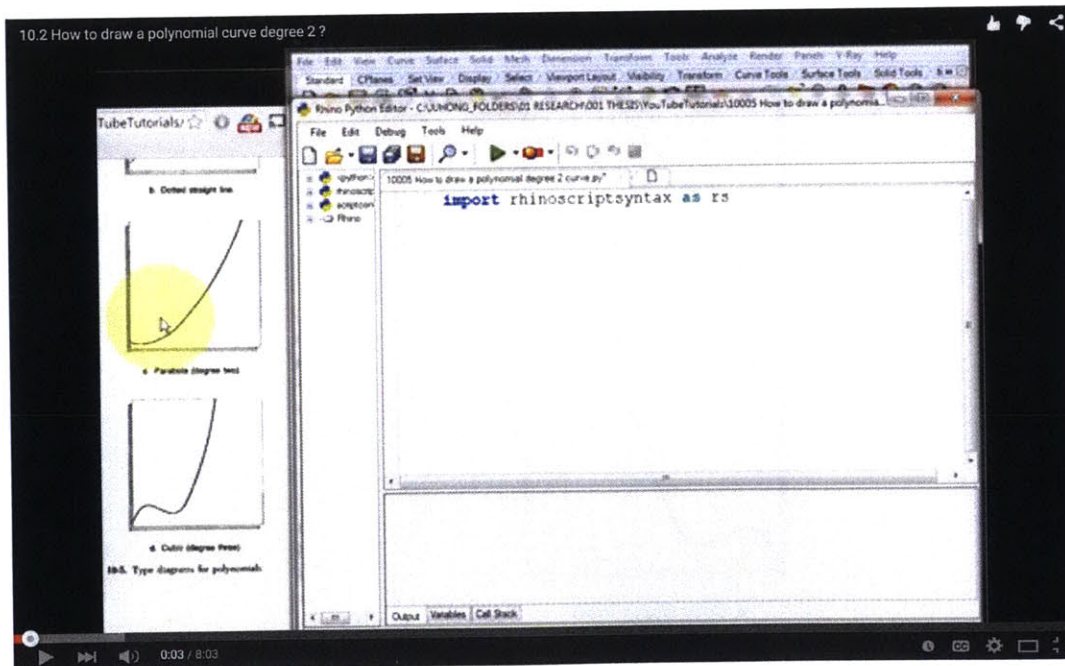
def poly_2(a0, a1, a2,
           x0, y0, start_x, finish_x,
           n_segments):

    x_increment = (finish_x - start_x) / n_segments
    x = start_x
    y = a0 + a1 * x + a2 * exponent(x,2)
    pt0 = [x + x0, y + y0, 0]

    count = range(n_segments)
    for i in count:
        x = x + x_increment
        y = a0 + a1 * x + a2 * exponent(x,2)
        pt1 = [x + x0, y + y0, 0]
        rs.AddLine(pt0, pt1)
        pt0 = pt1

poly_2(0.123, -3.43, 0.13,
       0, 3, -10, 30,
       30)
```

RESULT



Module 108: 10.2 Function X3

10.
CURVES

10.2 FUNCTIONS OF X

In similar fashion, a polynomial of degree three is generated by this next procedure (fig. 10-5d)

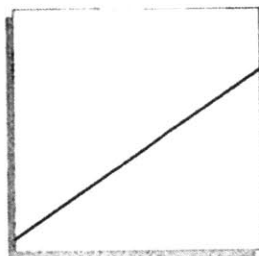
[sample codes on the right side]

In this case, the value of Y is assigned by the statement

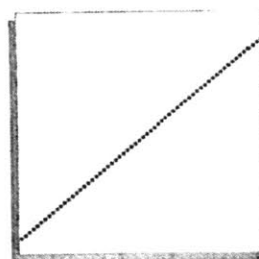
$$Y = A0 + A1 * X + A2 * \text{EXPONENT}(X, 2) \\ + A3 * \text{EXPONENT}(X, 3)$$

Now there are four coefficients A0, A1, A2, and A3.

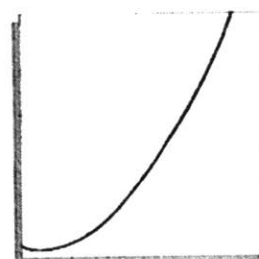
Similarly, for a fourth degree polynomial, we would have coefficients A0, A1, A2, A3, and A4 as well as a correspondingly expanded assignment statement. For a fifth degree polynomial we would have six such coefficients, and so on. Note that if some of the coefficients of a higher-degree polynomial are set to zero, a lower-order curve is generated. So we could, if we wished, employ one general polynomial procedure with n parameters to draw polynomial curves of any degree up to n. However, this would be inconvenient for use in drawing lower-order polynomials.



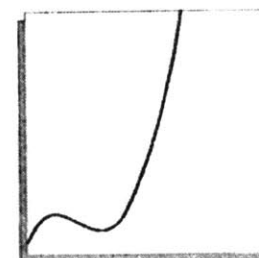
a. Straight line (degree one).



b. Dotted straight line.



c. Parabola (degree two).



d. Cubic (degree three).

10-5. Type diagrams for polynomials.

```

CODE
import rhinoscriptsyntax as rs

def exponent(base, power):
    e = base
    count = range(1, power)
    for i in count:
        e = e * base
    return e

def poly_3(a0, a1, a2, a3,
          x0, y0, start_x, finish_x,
          n_segments):

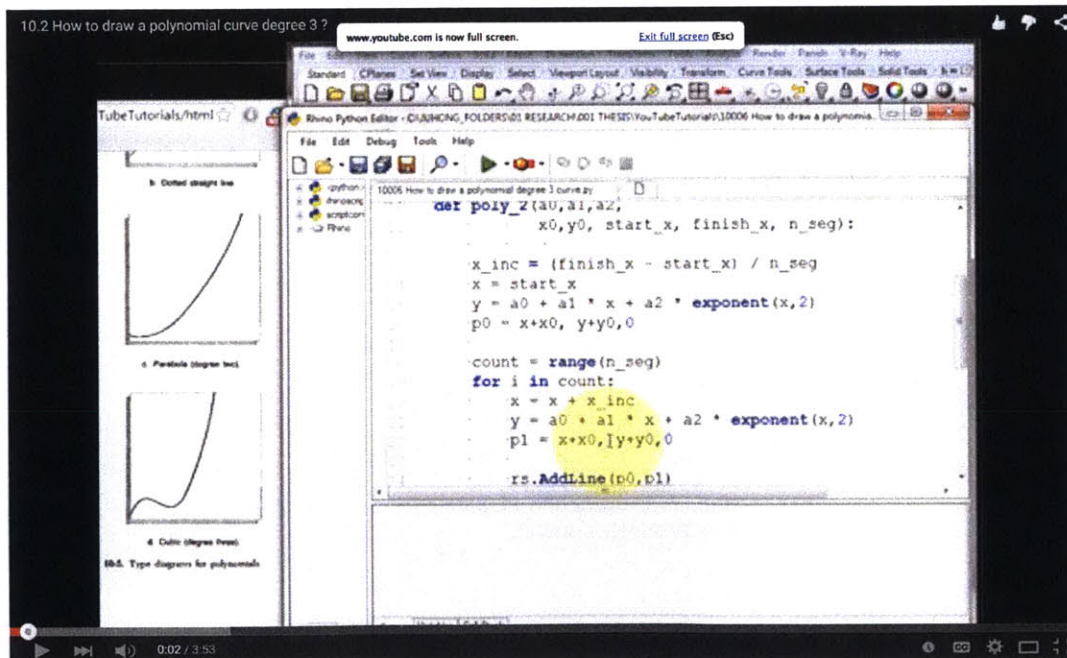
    x_increment = (finish_x - start_x) / n_segments
    x = start_x
    exp1 = exponent(x,2)
    exp2 = exponent(x,3)
    y = a0 + a1 * x + a2 * exp1 + a3 * exp2
    pt0 = [x + x0, y + y0, 0]

    count = range(n_segments)
    for i in count:
        x = x + x_increment
        exp1 = exponent(x,2)
        exp2 = exponent(x,3)
        y = a0 + a1 * x + a2 * exp1 + a3 * exp2
        pt1 = [x + x0, y + y0, 0]
        rs.AddLine(pt0, pt1)
        pt0 = pt1

poly_3(3, 0.23, 0.2, 0.3,
      0, 3, -5, 10,
      30)

```

RESULT



Module 109: 10.2 Function X4

10. CURVES

10.2 FUNCTIONS OF X

Polynomial procedures can be invoked from within loops to produce repetitive compositions of curves. The following procedure, for example, invokes Poly_2 to generate a pattern like Botticelli's waves (fig. 10-6).

[sample codes on the right side]



**10-6. A composition of parabolas
like Botticelli's waves.**

Module 110: 10.2 Function X5

10. CURVES

10.2 FUNCTIONS OF X

A pattern like Botticelli's shell (fig. 10-7) is generated by:

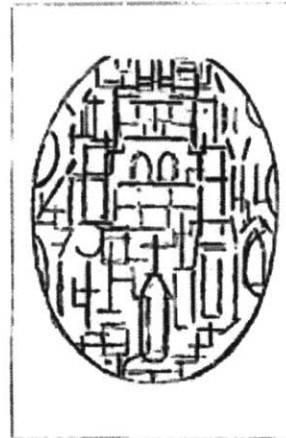
[sample codes on the right side]

In the case of the waves, position varies from instance to instance, whereas shape remains constant. In the shell, position is the same, whereas shape varies.

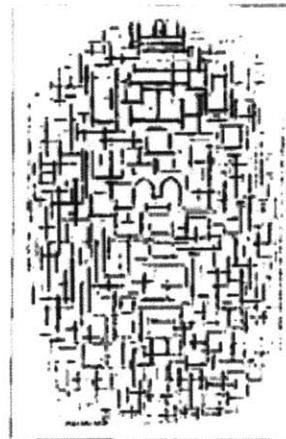
Are polynomials adequate approximations of Botticelli's curves, or are they too "mechanical"? This depends on the desired level of abstraction. If you want to emphasize similarity, repetition, and the clear, simple relation of graphic elements, then simple polynomials, with few parameters, are appropriate. But if you want to inflect curves in complex ways and create subtle variation from instance to instance, you will need more complex polynomials with more parameters, giving you more degrees of freedom. The works of Piet Mondrian explore this issue early paintings use a vocabulary of several different types of curves in a rich variety of relations, but later paintings reduce this to straight lines in just a few simple relations. At the same time, his early palette was wide and was developed to create complex relationships of tone, hue, and saturation. He then reduced his palette to black, white, and saturated primaries. Sometimes this process of simplification is illustrated particularly strikingly by successive versions of the same motif. The sketch of a church in figure 10-8a, for example, contains many curved lines, and there is considerable variation from line to line. The later version, shown in figure 10-8b, is reduced almost completely to a grid like pattern of horizontal and vertical straight segments fewer graphic variables are used and simpler relations are formed.



10-7. A composition of parabolas like Botticelli's shell.



a. Sketch of an architectural motif, with several different types of curves related in complex ways.



b. A later version consisting almost entirely of straight lines in †, L, and T relationships.

10-8. Levels of abstraction in the work of Piet Mondrian (1917).

```

CODE
import rhinoscriptsyntax as rs

def exponent(base, power):
    e = base
    count = range(1, power)
    for i in count:
        e = e * base
    return e

def poly_2(a0, a1, a2,
          x0, y0, start_x, finish_x,
          n_segments):

    x_increment = (finish_x - start_x) / n_segments
    x = start_x
    y = a0 + a1 * x + a2 * exponent(x,2)
    pt0 = [x + x0, y + y0, 0]

    count = range(n_segments)
    for i in count:
        x = x + x_increment
        y = a0 + a1 * x + a2 * exponent(x,2)
        pt1 = [x + x0, y + y0, 0]
        rs.AddLine(pt0, pt1)
        pt0 = pt1

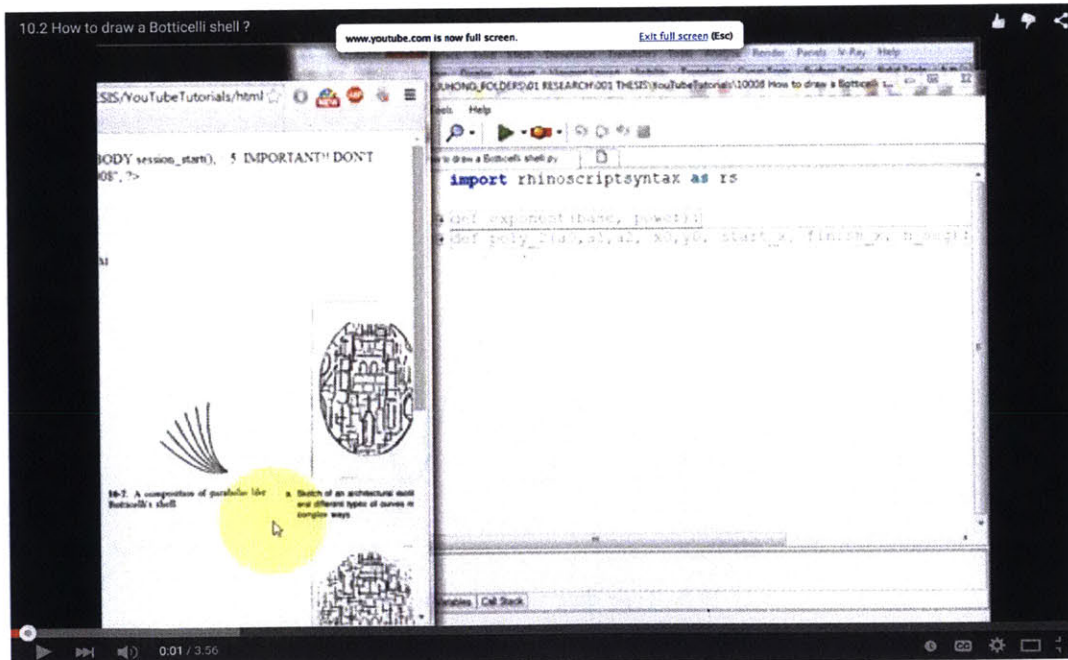
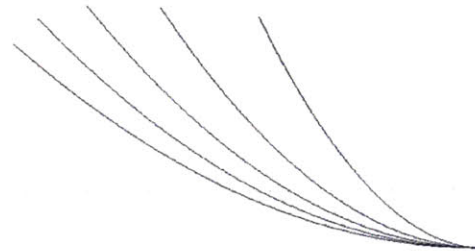
def shell():
    # start drawing
    a0 = 0
    a1 = 0
    a2 = 0.012
    inc = 40
    start_x = -90
    finish_x = 0
    x0 = 300
    y0 = 10

    count = range(1, 6)
    for i in count:
        poly_2(a0, a1, a2, x0, y0, start_x, finish_x, 20)
        start_x = start_x - inc
        inc = inc - 10
        a2 = a2 * ( 1 - 1 / (i + 1) )

shell()

```

RESULT



Module 111: 10.2 Function X6

10. CURVES

10.2 FUNCTIONS OF X

Polynomials are not the only kinds of functions that can be employed to generate curves. Trigonometric functions can also be used. Our next procedure illustrates this by drawing a sine curve (fig. 10-9), the coordinates of which are found by evaluating the Python's `math.sin()` function.

[sample codes on the right side]

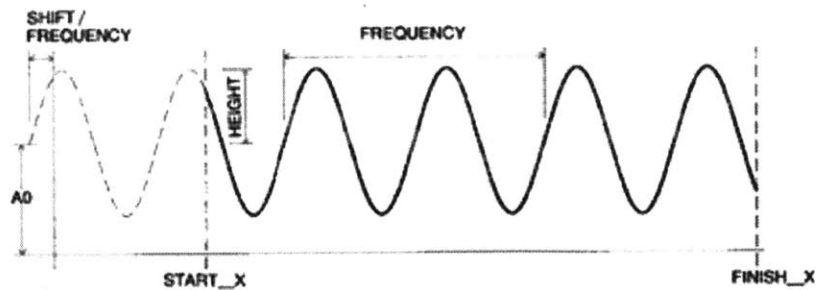
Here the value is assigned by the statement

$$Y = AC + HEIGHT * SIN(THETA)$$

Where

$$THETA = (FREQUENCY * X + SHIFT) * RADIANS$$

Height specifies the amplitude, Frequency specifies the number of cycles for a given range of X, Shift specifies the displacement from the origin (where a value of 0 gives a standard sine curve), and A0 is the vertical displacement of the origin.



10-9. Type diagram for a sine curve.

CODE

```
import rhinoscriptsyntax as rs
import math

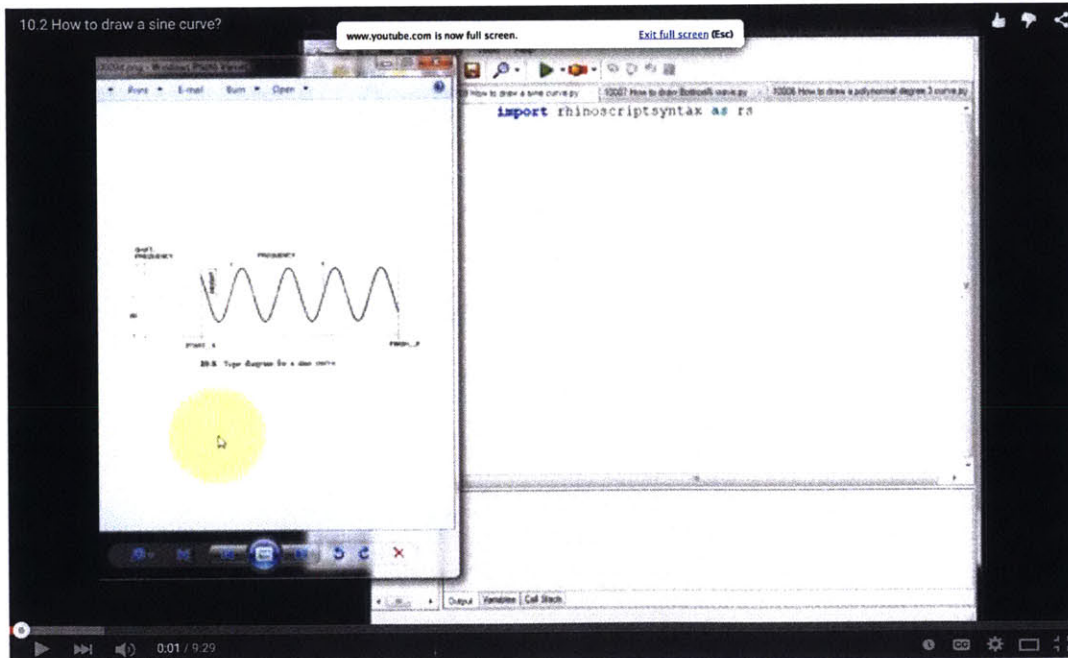
def sine_curve(x0, y0, start_x, finish_x,
              n_segments, height, frequency,
              shift, a0):

    radians = 0.01745
    x_increment = (finish_x - start_x) / n_segments
    x = start_x
    theta = (frequency * x + shift) * radians
    y = a0 + (height * math.sin(theta))
    pt0 = [x + x0, y + y0, 0]

    count = range(n_segments)
    for i in count:
        x = x + x_increment
        theta = (frequency * x + shift) * radians
        y = a0 + (height * math.sin(theta))
        pt1 = [x + x0, y + y0, 0]
        rs.AddLine(pt0, pt1)
        pt0 = pt1

sine_curve(0, 0, 0, 100,
          100, 10, 10,
          10, 0)
```

RESULT



Module 112: 10.2 Function X7

10. CURVES

10.2 FUNCTIONS OF X

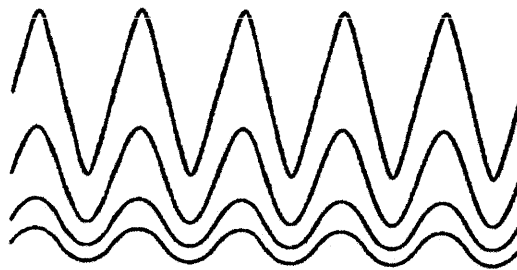
The following code invokes Sine_curve within a loop to generate a repetitive composition

[sample codes on the right side]

Figure 10-10 shows some output. The axes of the curves are parallel and evenly spaced, and they are all in phase, but height varies. More complex compositions could be produced by varying additional parameters within the loop.

As these examples illustrate, there is a general form for code to plot curves that are described as functions of X

[sample codes on the right side]



**10-10. A repetitive composition
of sine curves.**

```

CODE
import rhinoscriptsyntax as rs
import math

def sine_curve(x0, y0, start_x, finish_x,
              n_segments, height, frequency,
              shift, a0):

    radians = 0.01745
    x_increment = (finish_x - start_x) / n_segments
    x = start_x
    theta = (frequency * x + shift) * radians
    y = a0 + (height * math.sin(theta))
    pt0 = [x + x0, y + y0, 0]

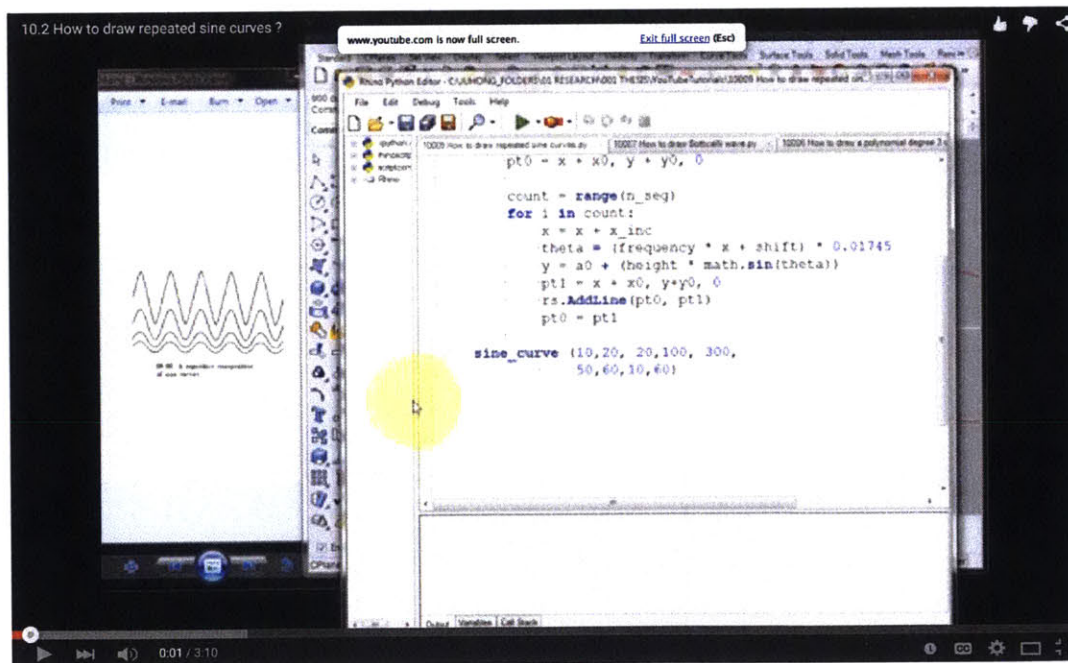
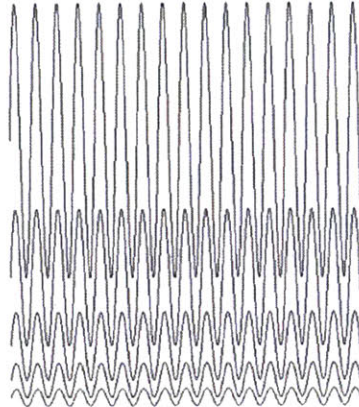
    count = range(n_segments)
    for i in count:
        x = x + x_increment
        theta = (frequency * x + shift) * radians
        y = a0 + (height * math.sin(theta))
        pt1 = [x + x0, y + y0, 0]
        rs.AddLine(pt0, pt1)
        pt0 = pt1

def sine_curve_repeat():
    y0 = 20
    height = 25
    count = range(5)
    for i in count:
        sine_curve(10, y0, 0, 1000, 1000, height, 6, 0, 0)
        height = height * 2
        y0 = y0 + height

sine_curve_repeat()

```

RESULT



Module 113: 10.3 Function Y

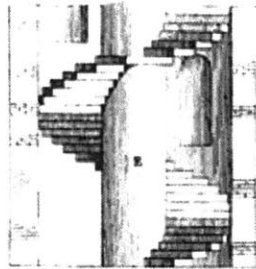
10. CURVES

10.3 FUNCTIONS OF Y

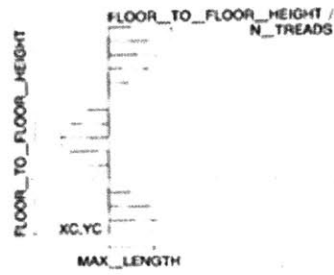
A similar approach can be taken to curves in which X is a function of Y. Figure 10-11, for example, shows an elevation of spiral stairs the outer ends of the treads trace out sine curves. Simplified figures of this type are generated by the following procedure (fig. 10-12)

[sample codes on the right side]

Figure 10-13 shows a beautiful section drawing of an elliptical spiral stair by Paul Letarouilly. It is a composition of instances of vertical sine curves, contrasted with the regular horizontal rhythms of the treads and the vertical rhythms of columns and balusters. Its richness is generated by overlaying variations on the theme of the vertical sine curve.



10-11. Circular spiral stairs, from Isaac Ware's edition of Andrea Palladio's *Four Books of Architecture*.

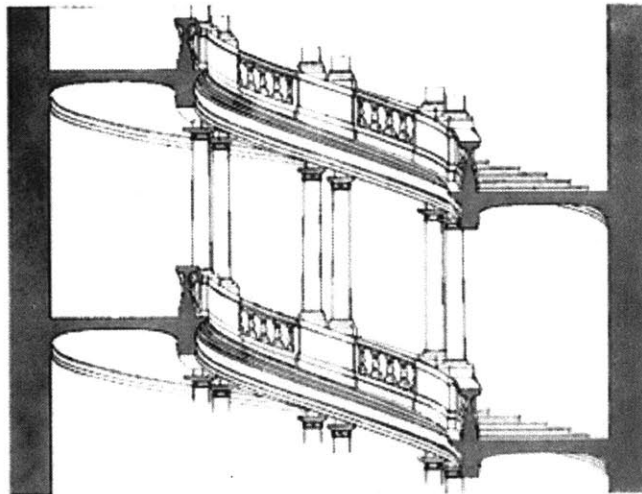


a. Type diagram.



b. Instances.

10-12. Simplified spiral stairs.



10-13. A section of the elliptical stair at the Barberini Palace, Rome, from Paul Letarouilly's *Edifices de Rome Moderne*.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

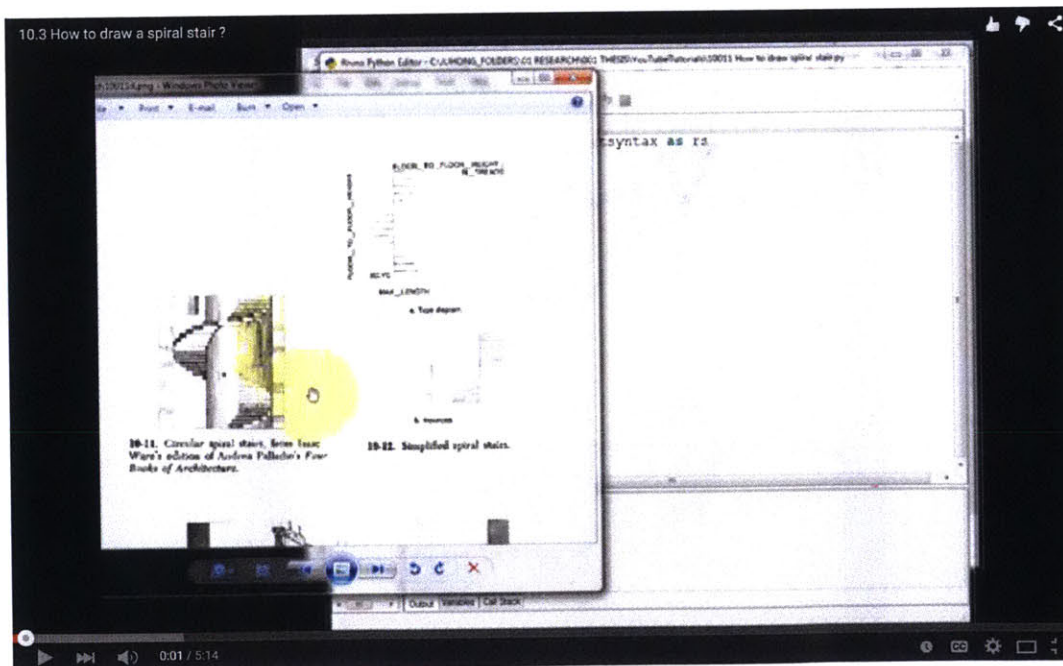
def spiral_stair(xc, yc,
                floor_to_floor_height, n_treads,
                max_length, frequency):

    radians = 0.01745
    y_increment = floor_to_floor_height / n_treads
    y = yc

    count = range(n_treads)
    for i in count:
        theta = frequency * y * radians
        x = xc + (max_length * math.sin(theta))
        pt0 = [xc, y, 0]
        pt1 = [x, y, 0]
        rs.AddLine(pt0, pt1)
        y = y + y_increment

spiral_stair(100,100,100,50,120,8)
```

RESULT



Module 114: 10.4 Polygon

10. CURVES

10.4 REGULAR POLYGONS AND CIRCLES

The most frequently used type of curve in architecture and most other fields of design is the circle. For our purposes, a circle may be regarded as a regular polygon with a sufficiently large number of sides to appear smooth. It is useful, then, to have a procedure that draws a regular polygon with any specified number of sides, of any specified radius, and centered at any specified point (X, Y) (fig. 10-14).

We cannot simply evaluate a function of X to find the Y coordinates of points on a circle, because for any given value of X there will always be two Y coordinates (fig. 10-15). We must therefore find some other approach.

Figure 10-16 illustrates the principle that we will employ. An Angle is measured from a vertical line. We know, from elementary trigonometry, that the X coordinate of a point on the circle is assigned by

$$X_COORD = XC + RADIUS * MATH.COS(ANGLE)$$

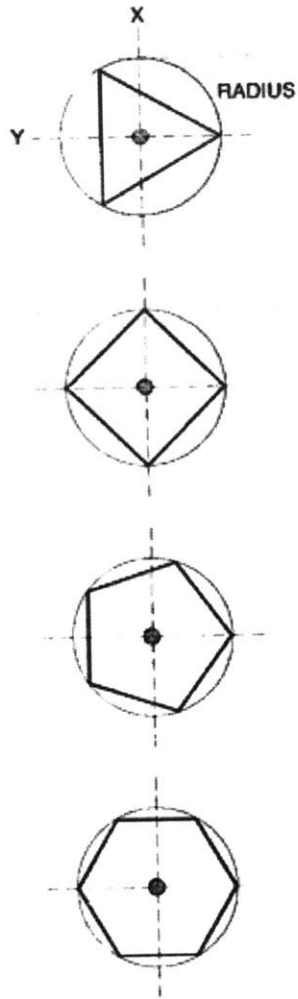
Similarly, the Y coordinate is assigned by

$$Y_COORD = YC + RADIUS * MATH.SIN(ANGLE)$$

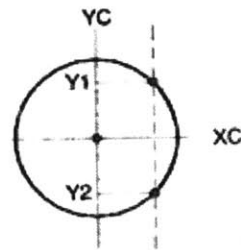
So, to produce an n-sided regular polygon, we need a procedure that executes a loop from 1 to N_sides, calculates a value for Angle at each iteration, and calculates X_coord and Y_coord

[sample codes on the right side]

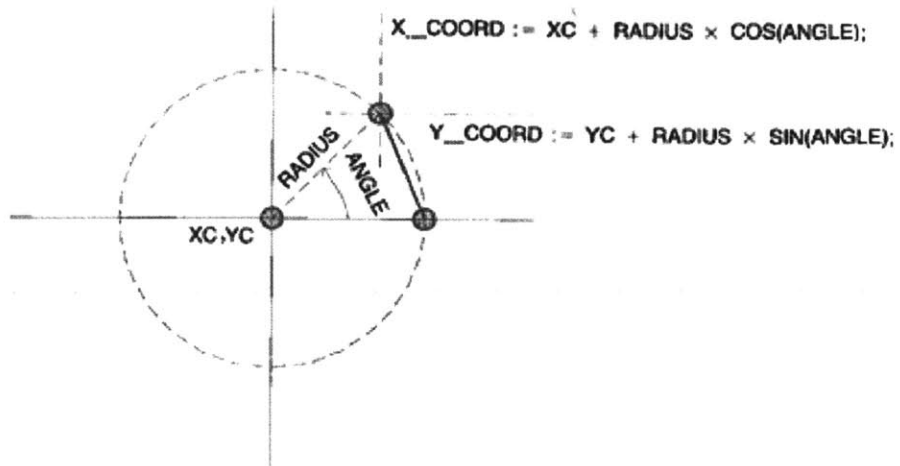
Some examples of output from this procedure are shown in figure 10-17. You should experiment with it to find how many sides you need to draw smooth-looking circles of various sizes. You should avoid specifying more sides than you really need, because this will result in unnecessary computation and will slow down the process of displaying a circle.



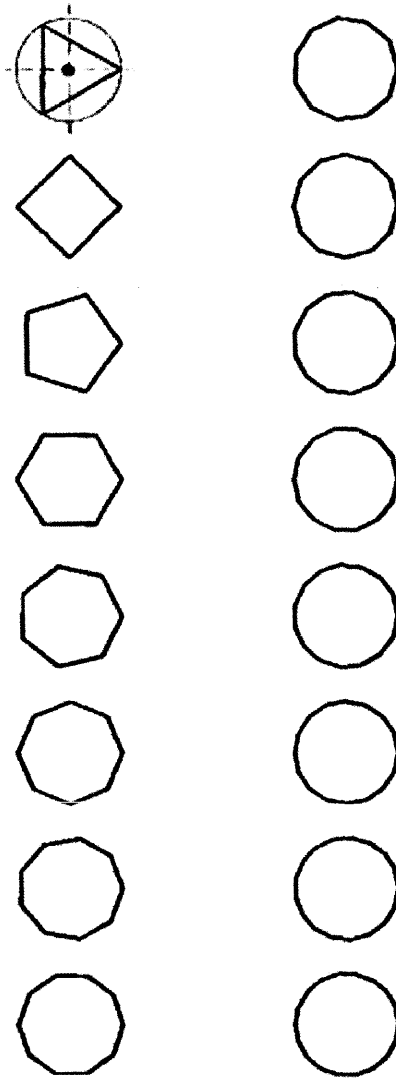
10-14. The parameterization of a regular polygon by center coordinates, radius, and number of sides.



10-15. For any X coordinate on the circumference of a circle, there are two Y-coordinate values.



10-16. The geometric construction for drawing a regular polygon.



10-17. Polygons with increasing numbers of sides that will eventually approximate a circle.

```

CODE
import rhinoscriptsyntax as rs
import math

def polygon(xc, yc, radius, n_sides):
    ptList = []
    radians = 0.01745
    increment = (360 / n_sides) * radians
    angle = 0
    x_coord = xc + radius
    pt0 = [x_coord, yc, 0]
    ptList.append(pt0)

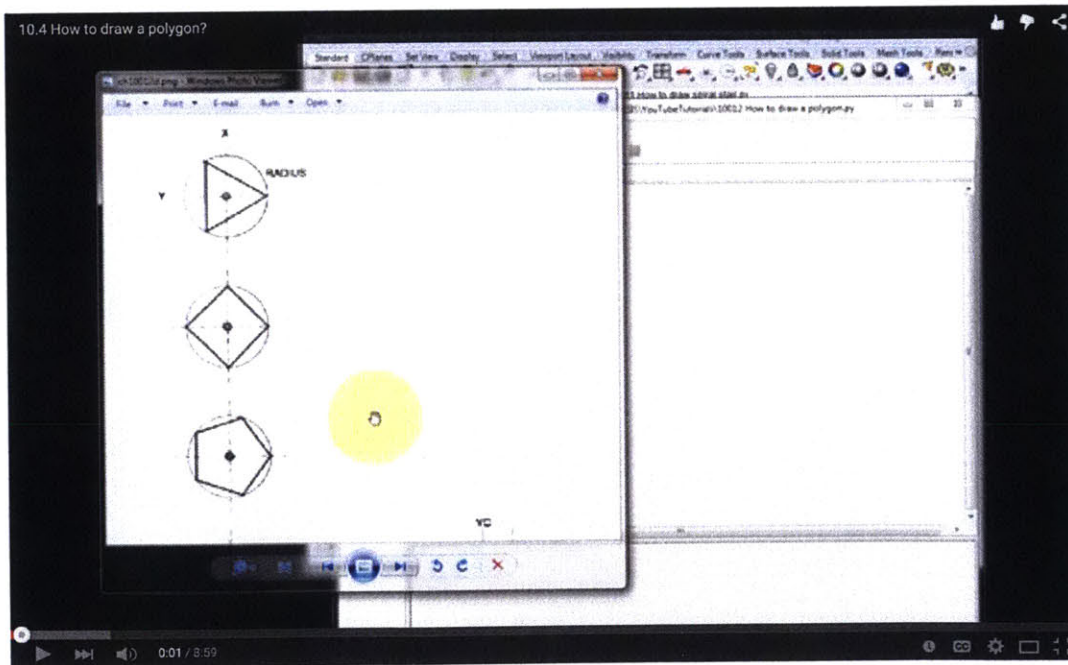
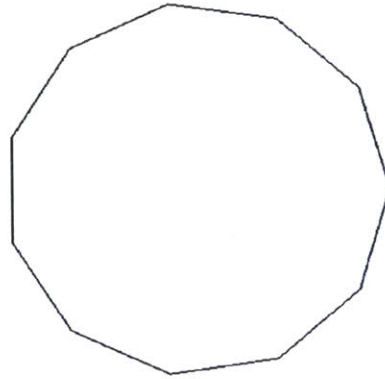
    count = range(1, n_sides+1)
    for i in count:
        angle = angle + increment
        x_coord = xc + (radius * math.cos(angle))
        y_coord = yc + (radius * math.sin(angle))
        pt = [x_coord, y_coord, 0]
        ptList.append(pt)

    rs.AddPolyline(ptList)

polygon(1,1,100,11)

```

RESULT



Module 115: 10.4 Circle

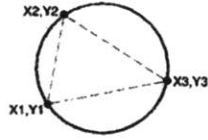
10. CURVES

10.4 REGULAR POLYGONS AND CIRCLES

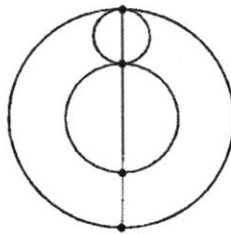
It is usually most convenient to specify a circle by its center point and radius, but you may want to specify a circle by the coordinates of three points on its perimeter (fig. 10-18). Here is a procedure that takes these coordinate values as parameters and invokes our previous circle procedure to generate the required result

[sample codes on the right side]

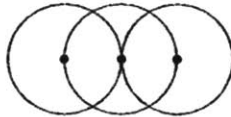
Because circles are perfectly symmetrical, we can create only simple relations between instances in compositions ratios of diameters and relations of center points (fig. 10-19). Certain simple relations of the circle and the straight line are very common in architectural and graphic compositions (fig. 10-20) diametrical, radial, chord, and tangential. The combination of repeating straight line patterns and repeating circular patterns yields many common motifs, such as the radio concentric web (fig. 10-21).



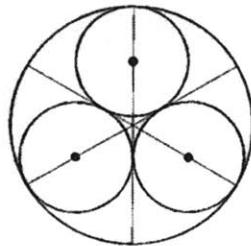
10-18. A circle specified by three points on its perimeter.



a. Simple ratios of diameters.

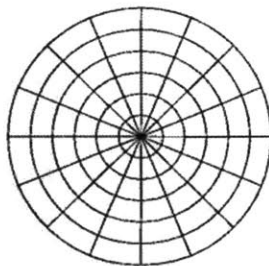


b. Simple relationships of center points.

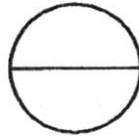


c. A more complex relationship of four circles.

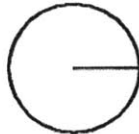
10-19. Relationships of circles in compositions.



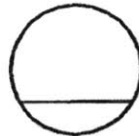
10-21. The radioconcentric web.



a. Diametrical.



b. Radial.



c. Chord.



d. Tangential.

10-20. Common simple relationships of the circle and the straight line.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def polygon(xc, yc, radius, n_sides):
    ptList = []
    radians = 0.01745
    increment = (360 / n_sides) * radians
    angle = 0
    x_coord = xc + radius
    pt0 = [x_coord, yc, 0]
    ptList.append(pt0)

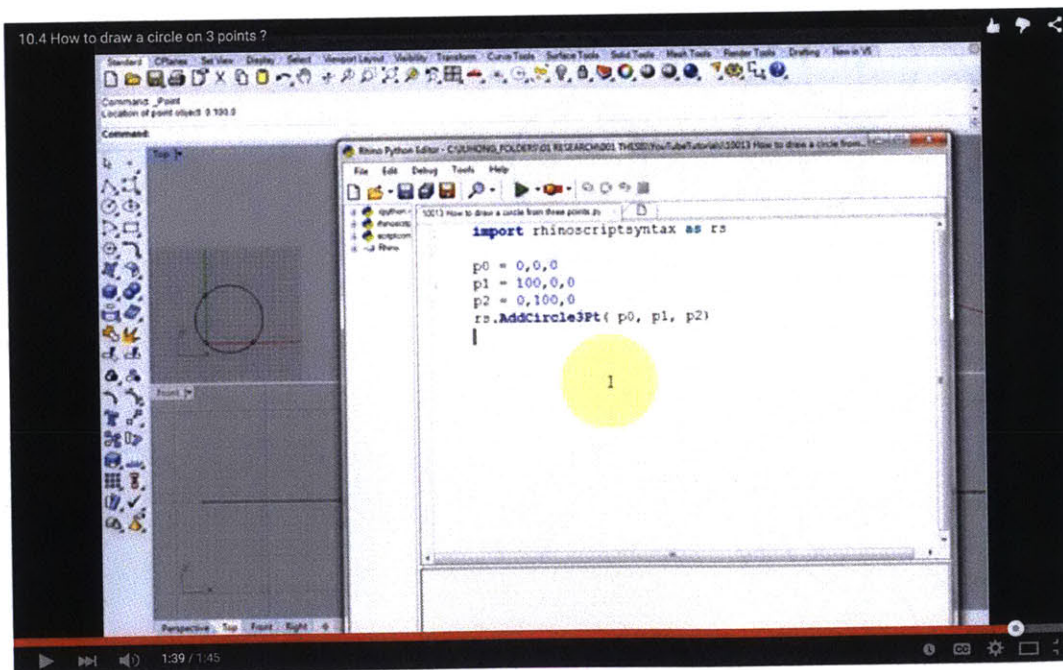
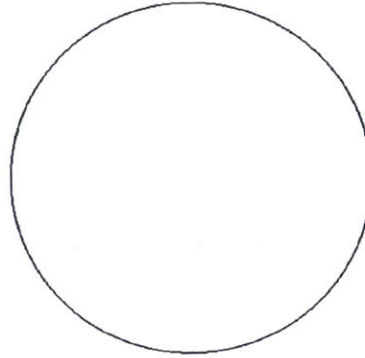
    count = range(1, n_sides+1)
    for i in count:
        angle = angle + increment
        x_coord = xc + (radius * math.cos(angle))
        y_coord = yc + (radius * math.sin(angle))
        pt = [x_coord, y_coord, 0]
        ptList.append(pt)

    rs.AddPolyline(ptList)

def circle3(x1, y1, x2, y2, x3, y3, n_sides):
    tx2 = x2 - x1
    ty2 = y2 - y1
    tx3 = x3 - x1
    ty3 = y3 - y1
    temp1 = ty3 * (tx2 + tx2 + ty2 * ty2)
    temp2 = ty2 * (tx3 + tx3 + ty3 * ty3)
    temp3 = (tx2 + tx3 - tx3 + ty2) / 2
    xc = (temp1 - temp2) / temp3
    temp1 = tx3 * (tx2 * tx2 + ty2 * ty2)
    temp2 = ty2 * (tx3 * tx3 + ty3 * ty3)
    temp3 = (ty2 * tx3 - ty3 * tx2) / 2
    yc = (temp1 - temp2) / temp3
    radius = math.sqrt(xc * xc + yc * yc)
    xc = xc + x1
    yc = yc + y1
    polygon(xc, yc, radius, n_sides)

circle3(0,0, 100,30, 10,200, 300)
```

RESULT



Module 116: 10.5 Arc

10. CURVES

10.5 ARCS

Our circle procedure can be generalized to create an arc. In figure 10-22a an arc is defined by specifying the center point and Radius, together with an angle Theta_1 degrees to specify where it starts and an angle Theta_2 to specify where it ends. To draw this arc, we can modify our circle procedure to go from Theta_1 to Theta_2 in appropriately sized increments, instead of from 0 degrees to 360 degrees, as follows

[sample codes on the right side]

Some examples of output for different values of the parameters Radius, Theta_1, Theta_2, and N_sides are shown in figure 10-22b.

Arcs have convex and concave sides and can vary in their curvature. A straight line segment may be considered a degenerate case of an arc with infinite radius and zero curvature. A common compositional principle, then, is to contrast convexity with concavity and high curvature with low. The Plaza of the Three Powers at Brasilia (fig. 10-23) is a simple relation of horizontal, vertical, concave, and convex, and Aldo van Eyck's Arnheim Pavilion plan (fig. 10-24) breaks parallel straight lines with semicircles, three-quarter circles, and complete circles of varying sizes.

Synthetic Tutor

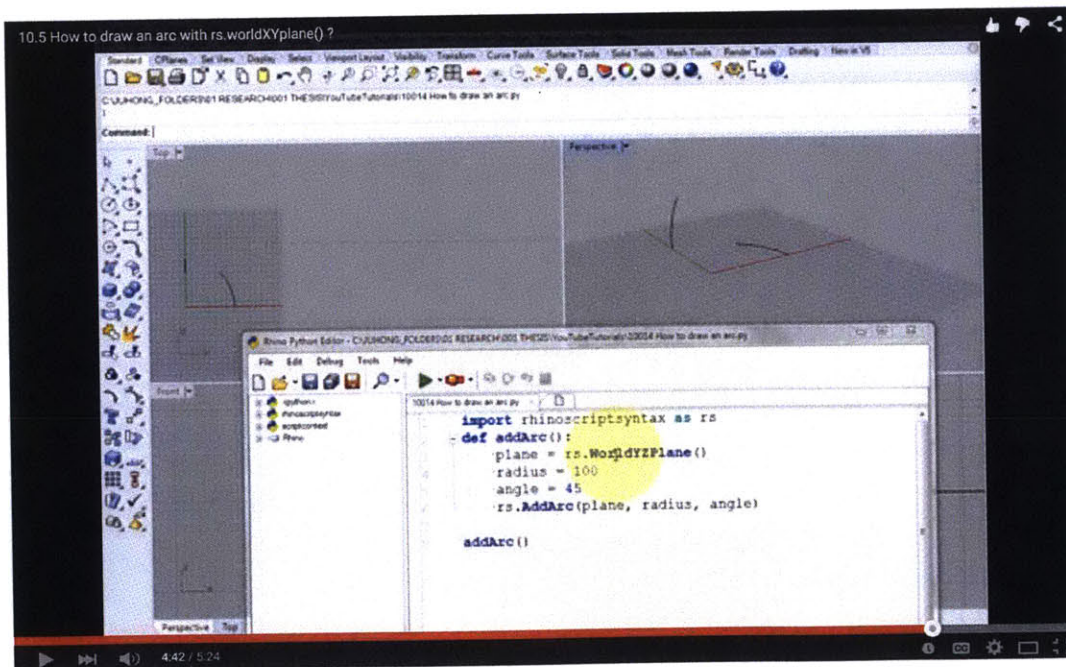
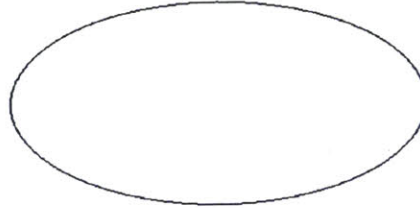
CODE

```
import rhinoscriptsyntax as rs

def addEllipse():
    plane = rs.WorldXYPlane()
    radiusX = 6
    radiusY = 3
    rs.AddEllipse(plane, radiusX, radiusY)

addEllipse()
```

RESULT



Module 117: 10.6 Ellipse

10. CURVES

10.6 ELLIPSE

Another generalization of our circle procedure is a procedure that generates an ellipse. This is analogous to the generalization of a square procedure to generate a rectangle. In both cases, we generalize by introducing more parameters.

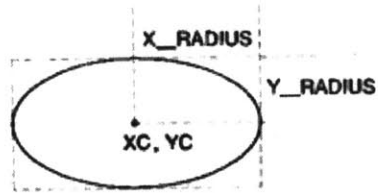
As figure 10-25 shows, we no longer have a single Radius, but an X_Radius and a Y_radius. Coordinates are now assigned by the expressions

$$\begin{aligned} X_COORD &= XC + X_RADIUS * MATH.COS(ANGLE) \\ Y_COORD &= YC + Y_RADIUS * MATH.SIN(ANGLE) \end{aligned}$$

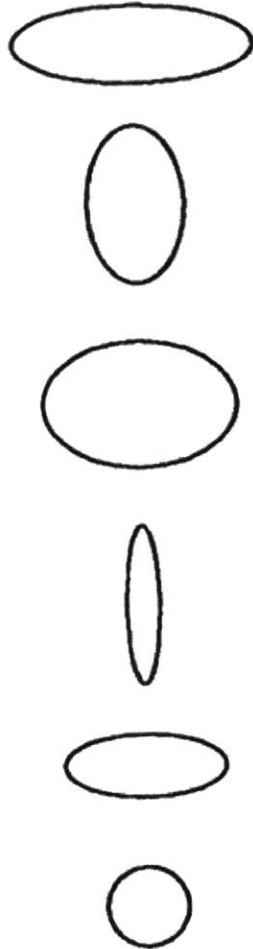
So the procedure now becomes

[sample codes on the right side]

An ellipse, unlike a circle, has a major axis, two different dimensions, and two foci rather than a single center point, so a richer variety of relations can be constructed in a composition of ellipses than in a composition of circles. Baroque architects realized and made extensive use of elliptical geometry in their plans, where their more classically inclined predecessors would have preferred the simplicity of the circle. Figure 10-26 shows a more recent exploitation of the complexities of the ellipse Charles Moore's design for the Beverly Hills Civic Center.

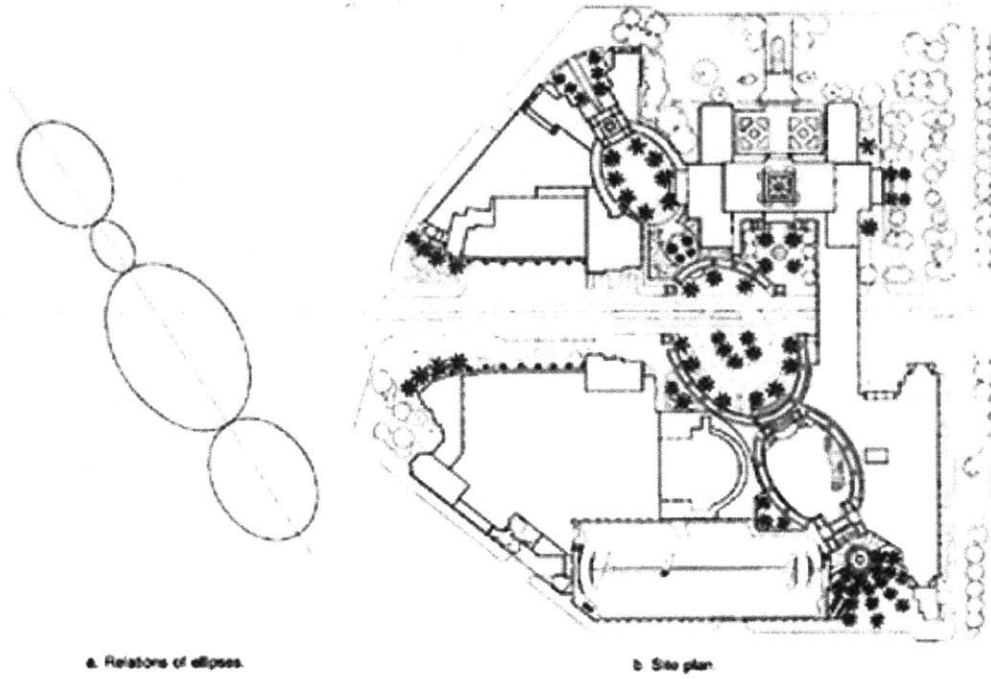


a. Type diagram.



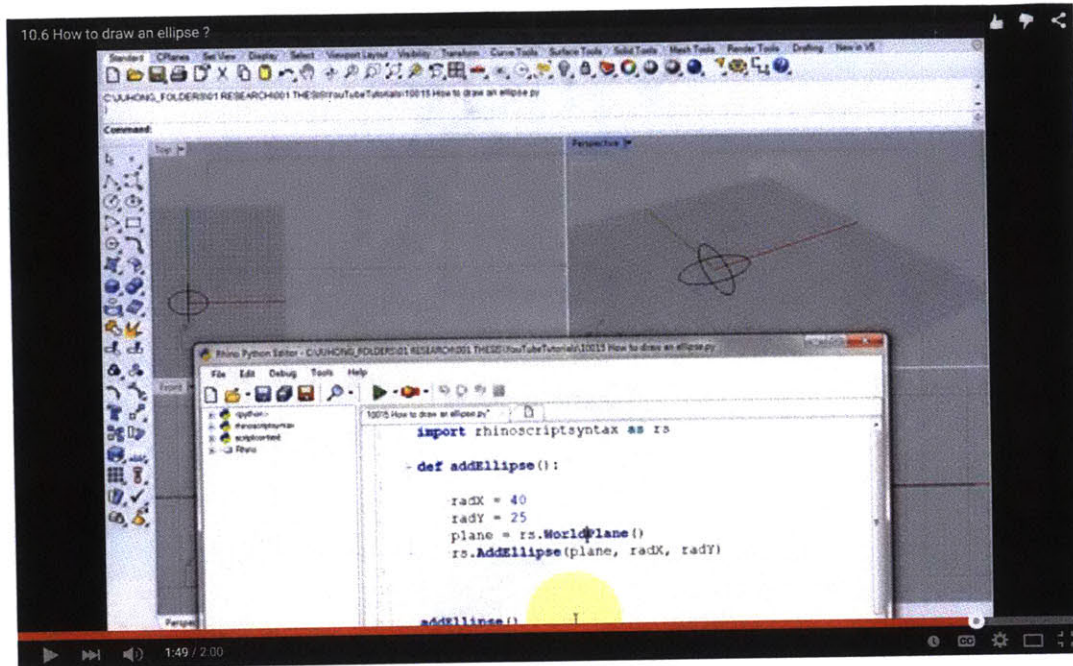
b. Some instances.

10-25. The ellipse.



10-86. Charles Moore and Urban Innovations Group. Beverly Hills Civic Center, California.

Synthetic Tutor



Module 118: 10.7 Summary

10. CURVES

10.7 SUMMARY

Curves are drawn by loops. Each iteration of a loop that draws a curve enervates one straight segment, the beginning point and the end point of which lie on the mathematically defined curve. Coordinate values are returned by functions that are invoked from within the loop. We have seen how polynomial and sine curves can be generated by functions of X or of Y in this way and how circles, arcs, ellipses, and spirals can be generated by functions of Angle.

Module 119: 11.1 Conditionals

11. CONDITIONALS

An artist or designer constantly chooses among sets of alternatives—colors from a palette, line weights provided by a set of pens, or shapes in a graphic vocabulary. Sometimes the designer makes choices such that the sizes, or shapes, or colors of elements in the composition are uniform. A regular composition results, and there is a danger that it will become boring. Conversely, the designer's choices may be entirely random or willful. The result, then, is an irregular composition, and there is a danger that it will become chaotic. Most interesting compositions, though, are neither uniform nor haphazard. Instead, the shapes, positions, and other attributes of the compositional elements vary conditionally, according to context. That is, variation is controlled by conditional rules.

Consider the three compositions of circles shown in figure 11-1. In the first, the circles are of uniform diameter and spacing, and it is easy to see that the composition can be generated by the execution of loops as follows:

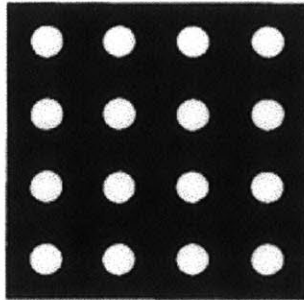
[sample codes on the right side]

In the second, the circles are of random diameter and spacing, and the coordinates and diameter of each must be explicitly specified:

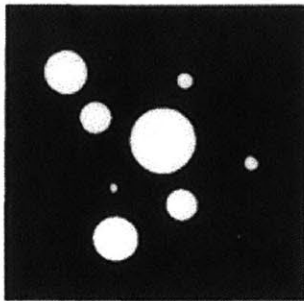
[sample codes on the right side]

There is variation from circle to circle in the third composition, too, but it is according to rule, rather than random.

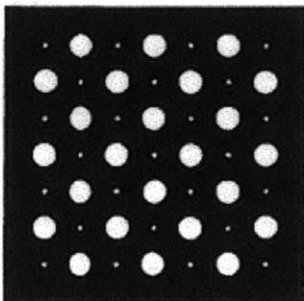
How can we write programs to generate compositions with conditional variations? We must be able to identify conditions that we want to respond to and specify what response to make when a condition is identified. In Python, Boolean expressions and Boolean functions can be evaluated to determine the existence of specific conditions of interest, and if and case statements (generally known as conditionals) can be used to relate conditions to responses. In this chapter, we shall discuss these constructs and explore their applications.



a. Uniform circles on a regular grid.



b. Random circles.



c. Variation according to a simple rule.

11-1. Unity and variety.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs

def circle(x, y, radius):
    plane = [x, y, 0]
    rs.AddCircle(plane, radius)

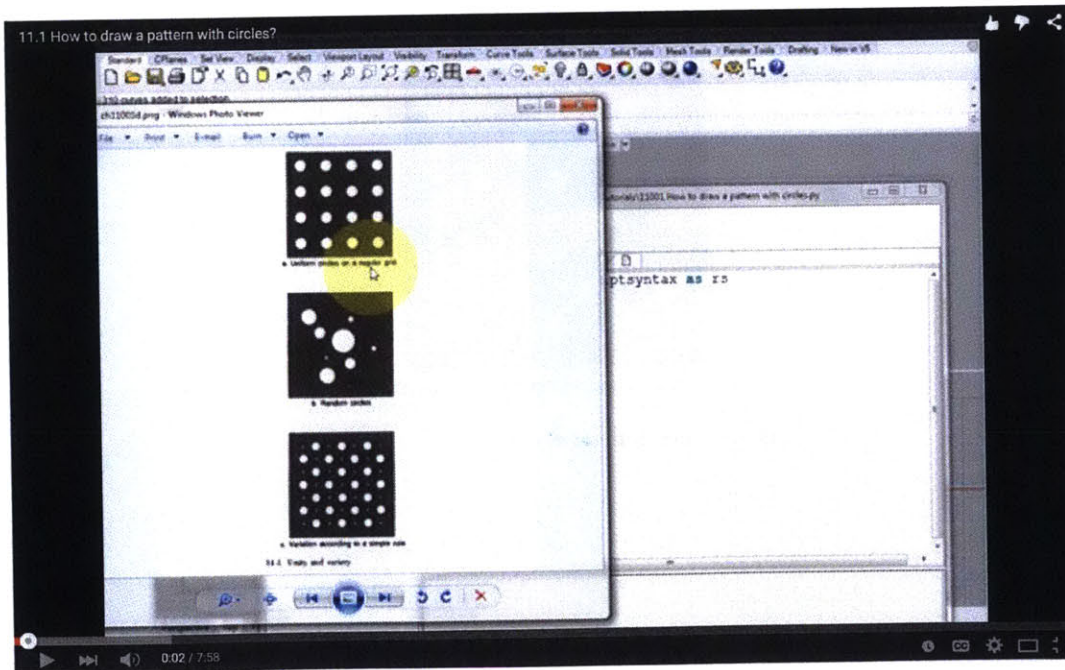
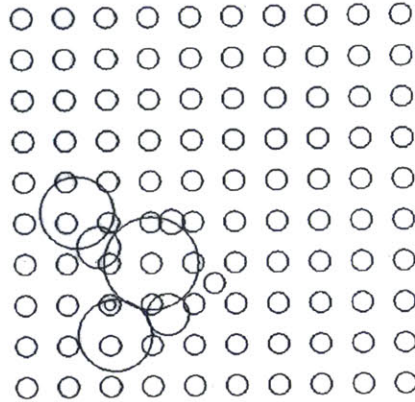
def uniform():
    x = 0
    y = 0
    increment = 200
    radius = 50
    NY = range(10)
    NX = range(10)
    for count_y in NY:
        x=0
        for count_x in NX:
            circle(x, y, radius)
            x = x + increment
            y = y + increment

uniform()

def random():
    circle(250, 850, 175)
    circle(350, 680, 100)
    circle(600, 600, 225)
    circle(425, 250, 175)
    circle(675, 350, 100)
    circle(400, 400, 25)
    circle(900, 500, 50)
    circle(700, 800, 60)

random()
```

RESULT



Module 120: Exercises 6

10.
CURVES

10.8 EXERCISES

1. On grid paper, make accurate plots of the curves generated by the following code:

a.
import rhinoscriptsyntax as rs

```
x = 10
y = 10000 / x
pt0 = [ x, y, 0 ]
```

```
while ( x < 1023 ):
    x = 1 + 10
    y = 10000 / x
    pt1 = [ x, y, 0 ]
    rs.AddLine( pt0, pt1 )
    pt0 = pt1
```

b.
import rhinoscriptsyntax as rs
import math

```
pt0 = [ 0, 0, 0 ]
for count in range( 1, 11 ):
    x = 100 * count
    y = math.sqrt( x )
    pt1 = [ x, y, 0 ]
    rs.AddLine( pt0, pt1 )
    pt0 = pt1
```

c.
import rhinoscriptsyntax as rs
import math

```
def polygon( xc, yc, radius, n_sides ):
    radians = 0.01745
    increment = (360 / n_sides) * radians
    angle = 0
    x_coord = xc + radius
    pt0 = [x_coord, yc, 0]
```

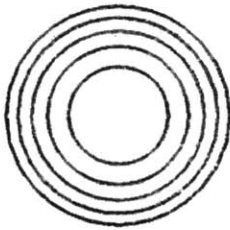
```
    for sides in range( 1, n_sides+1 ):
        angle = angle + increment
        x_coord = xc + radius * cos(angle)
        y_coord = yc + radius * sin(angle)
        pt1 = [ x_coord, y_coord, 0 ]
        rs.AddLine( pt0, pt1 )
        pt0 = pt1
```

```
radius = 300
```

Synthetic Tutor

```
while ( radius > 10 ):  
    polygon( 500, 500, radius, 4 )  
    radius = radius / math.sqrt( 2 )  
    polygon ( 500, 500, radius, 8 )
```

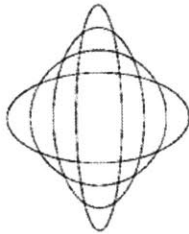
2. Write a procedure to generate Fresnel circles as shown in figure 10-29. (The area enclosed by each larger circle is twice that enclosed by its predecessor.) Write procedures as comments in a python.



10-29. Fresnel circles; each larger circle encloses twice the area of its predecessor.

Please upload your python file: No file chosen

3. Write a procedure to generate compositions of concentric ellipses (fig. 10-30). Let the X_diameter and Y_diameter vary independently. Write procedures as comments in a python.



10-30. A composition of concentric ellipses.

Please upload your python file: No file chosen

4. Write a procedure to generate curves of the type defined by the formula. Write procedures as comments in a python.

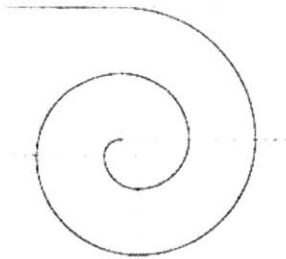
$$Y = X * \text{SIN}(X)$$

Please upload your python file: No file chosen

5. Generalizing the idea of a circular spiral, write an appropriately parameterized procedure to generate elliptical spirals. Write procedures as comments in a python.

Please upload your python file: No file chosen

6. The construction of a well-known ancient architectural motif, the Ionic volute, is illustrated in figure 10-31. Write a procedure to generate this curve. Write procedures as comments in a python.



10-31. The construction of an Ionic volute from quarter circles.

Please upload your python file: No file chosen

7. Figure 10-32 illustrates the footprint of Alvar Aalto's Baker Dormitory at Massachusetts Institute of Technology. What are the important relations between straight lines and axes in this composition? How might you generate interesting variations on on this theme? Write a parameterized procedure to do so. Write procedures as comments in a python.



10-32. The footprint of Alvar Aalto's Baker Dormitory at Massachusetts Institute of Technology.

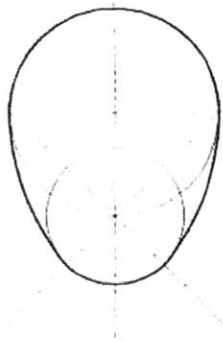
Please upload your python file: No file chosen

8. Take the procedures that you have written to generate curves and use them to write simple interactive programs that prompt for and read in parameter values. Then invoke the procedures to generate the corresponding curve instance. Experiment with the effects of inputting different values. Write procedures as comments in a python.

Please upload your python file: No file chosen

9. Figure 10-33 shows how to construct the shape of an egg from four arcs. Write an appropriately parameterized

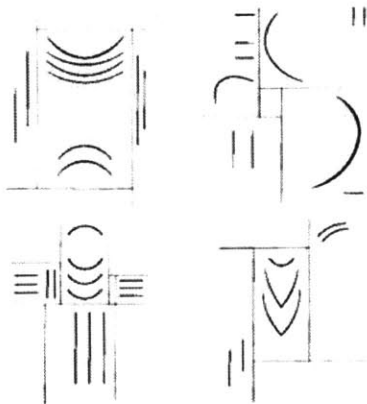
procedure to draw eggs. Write procedures as comments in a python.



10-33. The construction of an egg from four arcs.

Please upload your python file: No file chosen

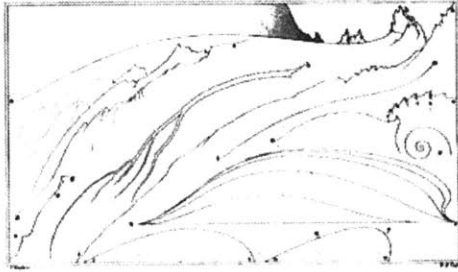
10. The schemata of four paintings by Georges Vantongerloo are shown in figure 10-34. What types of curves are used, and how are they related? Write procedures to generate the curves, and use them in programs to generate variations on Vantongerloo scheme. Write procedures as comments in a python file.



10-34. Schemata of four paintings by Georges Vantongerloo from the 1930s.

Please upload your python file: No file chosen

11. In *The Stones of Venice*, John Ruskin discussed the abstraction of different types of lines from nature (fig. 10-35). How might each of these types be generated? What are the parameters? Investigate ways to write procedures to generate them. Write procedures as comments in a python file.



10-35. Some curves sketched by John Ruskin in *The Stones of Venice*.

Please upload your python file: No file chosen

Module 121: 11.1 If Else

11. CONDITIONALS

11.1 TWO ALTERNATIVES: if, else

The simplest possible kind of design choice is between just two alternatives. Rules for making this kind of choice can conveniently be expressed in the following form:

```
if some condition is true:
    choose the first alternative
else:
    choose the second
```

The inner block must be indented. If the block statements do not indented properly, Python will cause an error.

A typographer, for example, might follow this rule:

```
if this is the first character of a sentence:
    uppercase
else:
    lowercase
```

We can represent this graphically, as illustrated in figure 11-2, as a choice between two branches. The proposition stated after if is either true or false. When it is true, the first branch is taken. When it is false, the second branch is taken.

Python provides the if... else... statement for specifying that one of two alternative actions is to be chosen, depending upon the truth value of an expression in this way. It takes the form:

```
if Boolean expression:
    Executable statement 1
else:
    Executable statement 2
```

Here is a simple example:

```
if X < 0:
    print (X IS NEGATIVE)
else:
    print (X IS POSITIVE)
```

In executing this, the computer first evaluates the Boolean expression $x < 0$. Depending on the value that has been assigned to the variable X, the result will either be true or false. If the result if true, the message

```
x IS NEGATIVE
```

will be output. If the result is false, then the message:

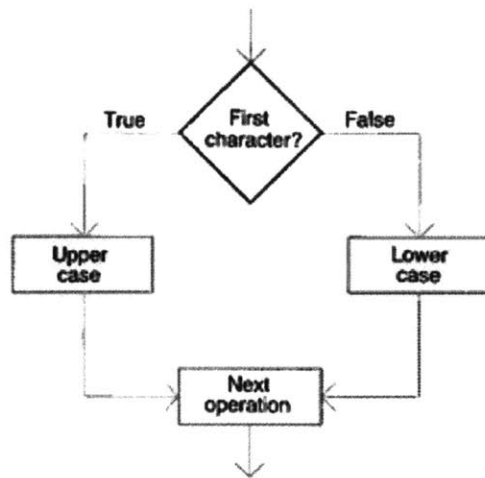
```
x IS POSITIVE
```

will be output. The computer then goes on to execute the next statement in the program in the normal way.

Note that an if... else statement always ends with : (colon).

```
if Boolean expression:  
    Executable statement 1  
    Executable statement 2  
else:  
    Executable statement 3  
    Executable statement 4
```

It is usual to indent, according to the convention shown, for clarity.



11-2. A flow diagram representing a choice between two branches in a program.

Synthetic Tutor

CODE

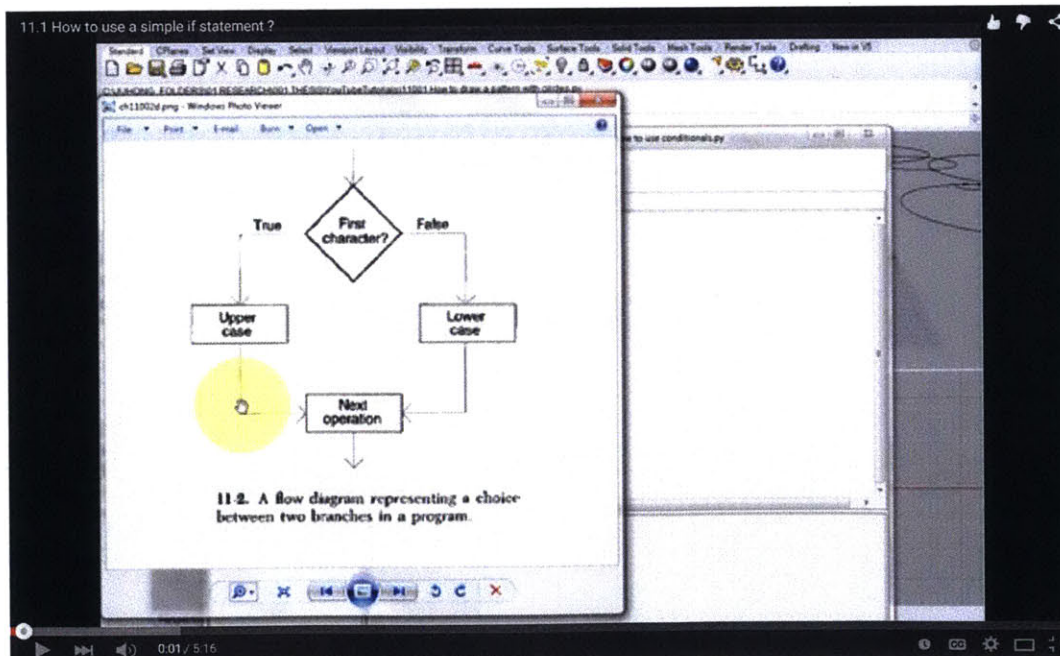
```
import rhinoscriptsyntax as rs

def simpleConditional():
    x = rs.CoefReal('type any number between -10 ~ 10')
    if x < 0 :
        print 'x is negative'
    else:
        print 'x is positive'

simpleConditional()
```

RESULT

x is positive



Module 122: 11.2 Boolean

11. CONDITIONALS

11.2 THE EVALUATION OF BOOLEAN EXPRESSIONS

As we have seen, an expression of type Boolean always appears after `if` in an `if... else...` statement. `True` and `false` are the only two possible values for such expressions. When a Boolean expression is evaluated, one of these values is substituted for the expression in the statement in which it occurs. We have encountered such expressions before, particularly in our discussions of a `while` loop, but only in its very simplest form. It will now be useful, before we go further, to look at them in more detail.

The simplest Boolean expression is a single Boolean variable. More complicated Boolean expressions consist of a constant, variable, or arithmetic or Boolean expression on the left-hand side, followed by a relational operator then a constant, variable, or arithmetic or Boolean expression on the right-hand side. Boolean expressions are evaluated by comparing the left-hand side to the right-hand side to see whether the specified relation holds (in which case the value is `true`) or does not (yielding a value `false`).

The relational operators, which are used to compare the left-hand side to right-hand side were introduced along with arithmetic and other operators in chapter 5. To refresh your memory, they are:

```
= equal
!= not equal
< less than
> greater than
<= less or equal
>= greater or equal
```

Here are some examples of their use in Boolean expressions:

```
A+B != MATH.SQRT(X)
MATH.SIN(X) < MATH.COS(Y)
TRUNCATE(X) = 0
```

The values of the left-hand side and the right-hand side must be of the same type, or the comparison would make no sense. We shall be concerned mostly with comparing integer values to integer values, but you can also compare to reals, Booleans to Booleans and characters to characters. You have to be particularly careful, incidentally, when using `and !=` to compare real values to real values, since round off errors may generate unexpected results.

Compound Boolean expressions can be formed from simple Boolean expressions by using the Boolean operators:

```
not: negation operator
and: conjunction operator
or: disjunction operator
```

If you know a little elementary logic, you will be familiar with these. If you are not, you should study the following definitions carefully. Where `P` and `Q` are Boolean variables, the meaning and effects of the Boolean operators are defined as follows:

```
Expression: ( Value )
not P: (FALSE if P is TRUE, TRUE if P is FALSE)
P and Q: (P AND Q TRUE if both P and Q are TRUE, FALSE otherwise)
P or Q: (FALSE if both P and Q are FALSE, TRUE otherwise)
```

Synthetic Tutor

Parentheses can, and sometimes must, be used to make compound Boolean expressions read unambiguously. Here are some examples of the use of Boolean operators to form compound Boolean expressions:

```
(X >= 0) and (X <= 1023)
(Y < 0) or (Y > 780)
(X < 10) and (Y=500) and P
```

Values of Boolean expressions can be assigned to Boolean variables in the same way that values of integer expressions can be assigned to integer variables, or values of real expressions to real variables:

```
value = ((X >= 0) and (X <= 1024))
        and
        ((Y >= 0) and (Y <= 1024))
```

Module 123: 11.3 Boolean

11. CONDITIONALS

11.3 BOOLEAN FUNCTIONS

Sometimes a considerable amount of code must be executed in order to determine the existence of some condition. In this case, it is usually clearest and most convenient to express that code as a Boolean function.

Consider, for example, the task of determining whether a specified integer is prime—that is, whether it is exactly divisible only by itself and 1. An obvious (though inefficient) way to do this is to attempt division by every integer from 2 to the specified number minus 1. The following Boolean function takes an integer argument `Number`, determines in this way whether `Number` is prime, and returns a value of true or false accordingly:

[sample codes on the right side]

Incidentally, there are many much cleverer ways to write this function. If you are interested, you should be able to discover some obvious improvements for yourself.

Boolean functions are employed in `if...else...` statements in the obvious way:

```
if PRIME (NUMBER):
    print ("NUMBER IS PRIME")
else:
    print ("NUMBER IS NOT PRIME")
```


Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

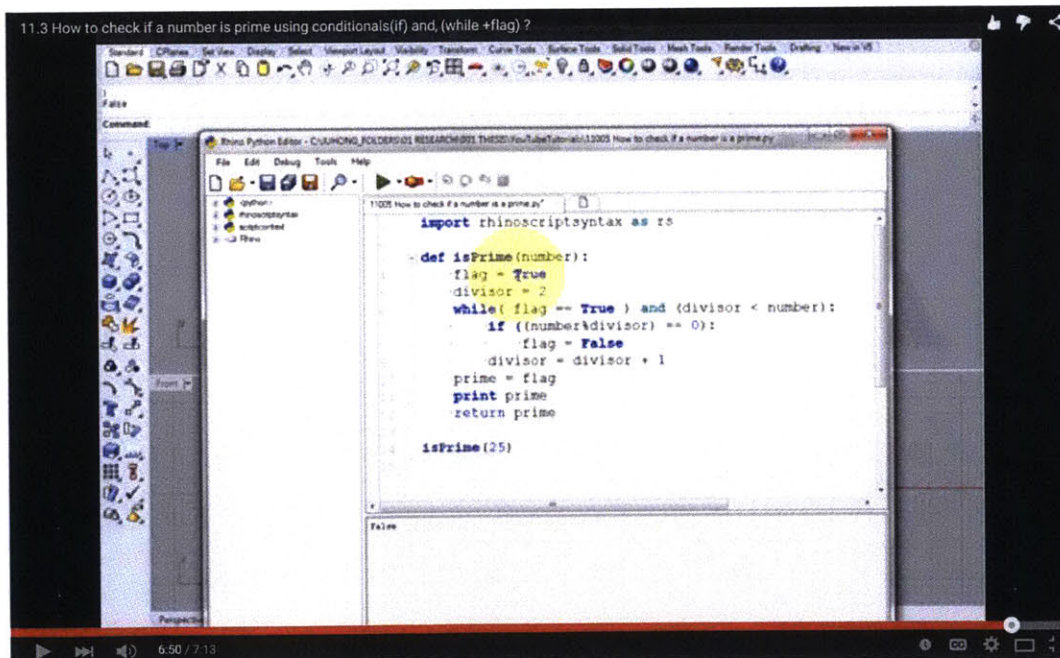
def prime(number):
    flag = True
    divisor = 2

    while (flag == True) and (divisor < number):
        if ((number % divisor) == 0):
            flag = False
            divisor = divisor + 1
    prime = flag
    print prime
    return prime

prime(23)
```

RESULT

True



Module 124: 11.4 Alternatives

11. CONDITIONALS

11.4 PAIRS OF DESIGN ALTERNATIVES

Now that we have the means to express rules of choice between two alternatives in Python, let us return to the design and graphic issues. The primary use of `if...else...` statements in graphics programs is to specify that either one graphics procedure or another is to be invoked, depending on circumstance, in order to instantiate either one or another graphic type. An architect drawing a floor plan, for example, might have a choice between square and circular columns and might express the rule of choice like this:

```
if condition:
    SQUARE (X,Y,DIAMETER)
else:
    CIRCLE (X,Y,DIAMETER)
```

An artist or designer may take a great many things into consideration when choosing vocabulary elements to instantiate in a composition, but Python is much more limited. The universe that it can consider consists of the variables that have been declared in the program and the values that have been assigned to them. Rules of choice must always be formulated in terms of these variables and must take the form of Boolean expressions constructed using the relational and Boolean operators. This seems and is very restrictive, but some surprisingly powerful and useful design rules can be expressed in this way.

In a graphics program (as we have seen in the examples considered so far) we will typically have variables describing the shapes, positions, and numbers of instances of various types of graphic elements in a composition. These variables, then, can be used in decision rules. It is often necessary, as well, to introduce additional variables specifically for the purpose of storing values that we want to use in decision rules, plus functions to assign values to these variables. Remember, Python cannot inspect a partially completed drawing in the way that you can when you are making a graphic decision it can inspect only the values currently assigned to declared variables, evaluate functions of these, and compare values.

Module 125: 11.4 Parity

11. CONDITIONALS

11.4 PAIRS OF DESIGN ALTERNATIVES

11.4.1 ODD OR EVEN: PARITY CONDITIONS

Consider the row of columns shown in figure 11-3. Beginning at the left, the odd columns are square and the even columns circular. This composition can be generated by a loop that instantiates the required number of columns and chooses either a square or circular one at each iteration, depending on whether the count is odd or even. We can build a Boolean function `odd(X)`, which takes an integer argument `X` and returns a value `true` when `X` is odd and `false` when `X` is even. Using this function, a procedure to generate the row of alternating square and circular columns can be written as follows:

[sample codes on the right side]

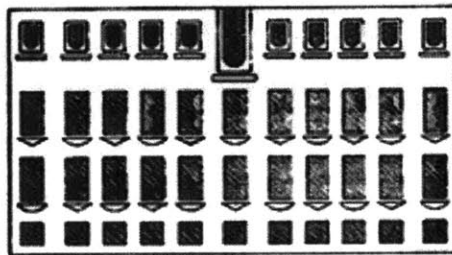
The zigzag illustrated in figure 11-4 can be generated in a very similar way. Here the decision rule is:

```
if ODD(COUNT):
    ZIG (COUNT * INCREMENT)
else:
    ZAG (COUNT * INCREMENT)
```

Figure 11-5 shows the elevation of a Renaissance palace. You will notice that the window types alternate. (This was a very common device for enlivening the rhythms of what could otherwise be fairly static compositions.) The general form of the decision rule for such rows of windows is:

```
if ODD(COUNT):
    WINDOW_1
else:
    WINDOW_2
```

11-5. Schematic elevation of a renaissance palace with alternating window types.



11-4. A zigzag.



11-3. A row of alternating square and circular columns.



Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import random

def square(x, y, diameter):
    x1 = x - diameter / 2
    y1 = y - diameter / 2
    plane = [x1, y1, 0]
    rs.AddRectangle(plane, diameter, diameter)

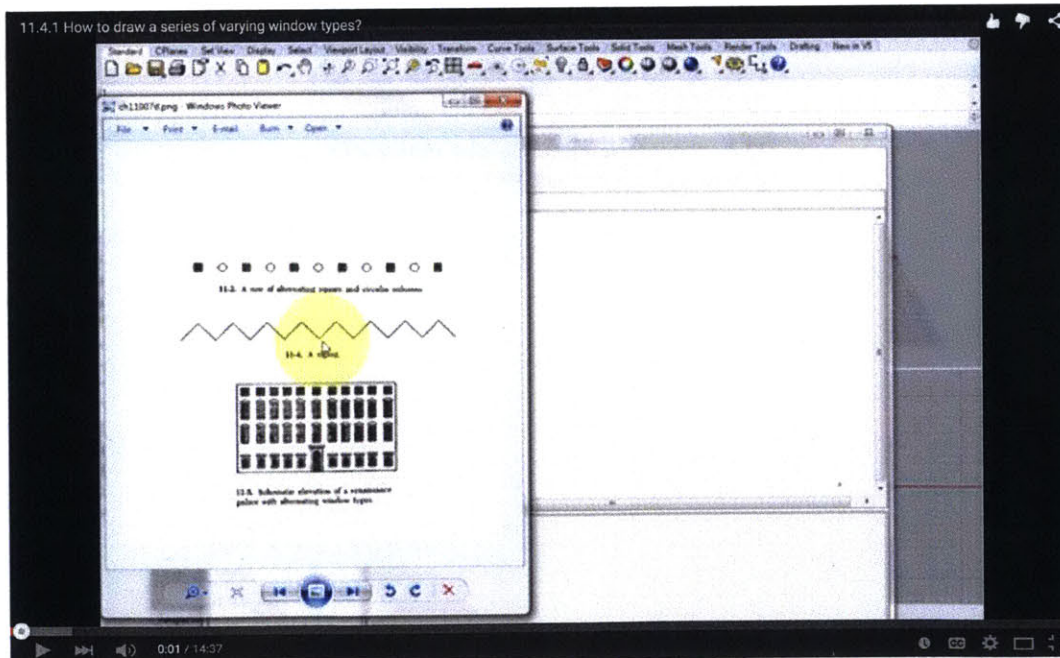
def circle(x, y, diameter):
    center = [x, y, 0]
    radius = diameter / 2
    rs.AddCircle(center, radius)

def odd(number):
    if (number % 2) != 0:
        return True

def row(initial_x, y, diameter, spacing, n_columns):
    x = initial_x
    count = range(n_columns)
    for i in count:
        if odd(i):
            square(x, y, diameter)
        else:
            circle(x, y, diameter)
        x = x + spacing

row(0, 0, 20, 100, 11)
```

RESULT



Module 126: 11.4 Ending

11. CONDITIONALS

11.4 PAIRS OF DESIGN ALTERNATIVES

11.4.2 EXTERIOR OR INTERIOR: END CONDITIONS

Figure 11-6 illustrates another row of circular and square columns. Here square columns are at the ends, and circular columns are everywhere else. The decision rule needed to generate this type of composition can be expressed:

[sample codes on the right side]

i is the control variable of a for loop from First to Last, and a Circular column at the center of an odd End_condition is the following Boolean function: row.

[sample codes on the right side]

The procedure to generate the complete composition is:

[sample codes on the right side]



11-6. A row of circular columns terminated by square columns.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def square(x, y, diameter):
    x1 = x - diameter / 2
    y1 = y - diameter / 2
    plane = [x1, y1, 0]
    rs.AddRectangle(plane, diameter, diameter)

def circle(x, y, diameter):
    center = [x, y, 0]
    radius = diameter/2
    rs.AddCircle(center, radius)

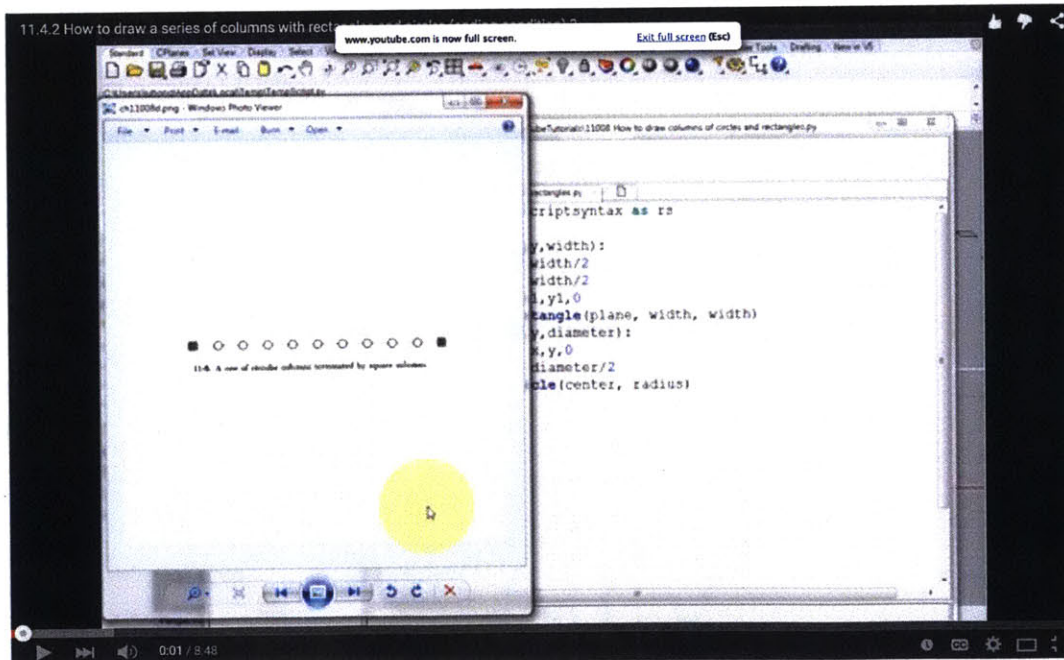
def odd(number):
    if (number % 2) != 0:
        return True

def endCondition(count, first, last):
    if count == first or count == last:
        return True
    else:
        return False

def rowWithEndCondition(initial_x, y, diameter,
                        spacing, n_columns):
    x = initial_x
    count = range(n_columns+1)
    for i in count:
        if endCondition(i, 0, n_columns):
            square(x, y, diameter)
        else:
            circle(x, y, diameter)
        x=x+spacing

rowWithEndCondition(0, 0, 25, 50, 13)
```

RESULT



Module 127: 11.4 Middle

11. CONDITIONALS

11.4 PAIRS OF DESIGN ALTERNATIVES

11.4.3 CENTER OF SLIDE: MIDDLE CONDITION

Yet another type of row of square and round columns is shown in figure 11-7a. There is an odd number of columns, and the center column is round those on either side are square. The decision rule needed here is:

[sample codes on the right side]

The Boolean function `Center_point` is as follows:

[sample codes on the right side]

Note that this function will work correctly only if the total number of columns in the row is odd. If the total is even, there is no center point.

[sample codes on the right side]

The case of an even number of columns is illustrated in figure 11-7b. Here the rule is to make the middle pair of columns circular instead of square. The decision rule is:

[sample codes on the right side]

The necessary Boolean function `Middle_pair` can be written:

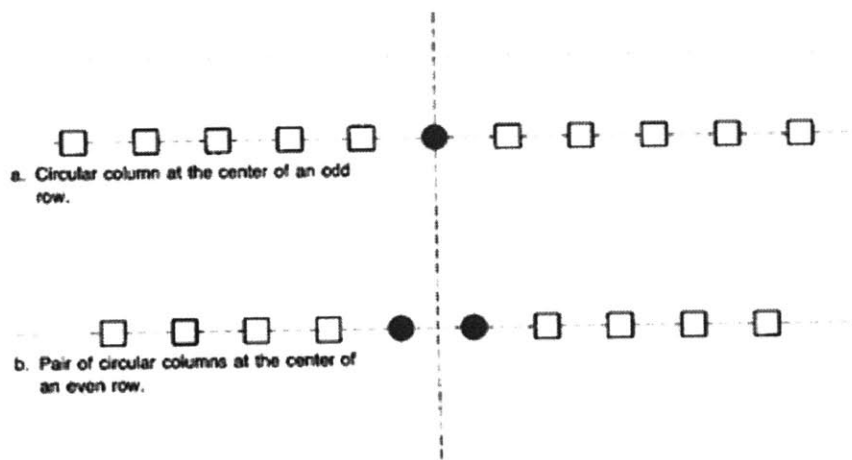
[sample codes on the right side]

If we want a general procedure that will work for either an odd or an even number of columns, we can use a decision rule like this,

[sample codes on the right side: `if...else...statements in the main() function`]

where `Oddrow` and `Evenrow` are procedures that draw odd and even rows of columns respectively.

The following is an interactive program that prompts for and accepts input specifying the starting coordinates of the row, the column spacing, the diameter, Note that this function will work correctly only if the total number of columns and the number of columns, then chooses whether to invoke `Odd row` or `Even row`:



11-7. The treatment of middle conditions.

```

CODE
import rhinoscriptsyntax as rs

def square(x, y, diameter):
    x1 = x - diameter / 2
    y1 = y - diameter / 2
    plane = [x1, y1, 0]
    rs.AddRectangle(plane, diameter, diameter)

def circle(x, y, diameter):
    center = [x, y, 0]
    radius = diameter / 2
    rs.AddCircle(center, radius)

def odd(number):
    if (number % 2) != 0:
        return True

def endCondition(count, first, last):
    if count == first or count == last:
        return True
    else:
        return False

def center_point(count, first, last):
    middle = int((last + first) / 2)
    isCenterPoint = (count == middle)
    return isCenterPoint

def middle_pair(count, first, last):
    left = int((last + first) / 2)
    right = left + 1
    is_middle_pair = ((count == left) or (count == right))
    return is_middle_pair

def oddRow(initial_x, y, diameter, spacing, n_columns):
    x = initial_x
    repeat = range(1, n_columns+1)
    for count in repeat:
        condition = center_point(count, 1, n_columns)
        if condition:
            circle(x, y, diameter)
        else:
            square(x, y, diameter)
        x = x + spacing

def evenRow(initial_x, y, diameter, spacing, n_columns):
    x = initial_x
    repeat = range(1, n_columns+1)
    for count in repeat:
        condition = middle_pair(count, 1, n_columns)
        if condition:
            circle(x, y, diameter)
        else:
            square(x, y, diameter)
        x = x + spacing

def main():
    x = rs.GetReal('enter X coordinate', 12)
    y = rs.GetReal('enter Y coordinate', 4)
    spacing = rs.GetReal('enter column spacing', 100)
    diameter = rs.GetReal('enter column diameter', 20)
    n_columns = rs.GetInteger('enter number of columns', 10)

    if odd(n_columns):
        oddRow(x, y, diameter, spacing, n_columns)
    else:
        evenRow(x, y, diameter, spacing, n_columns)

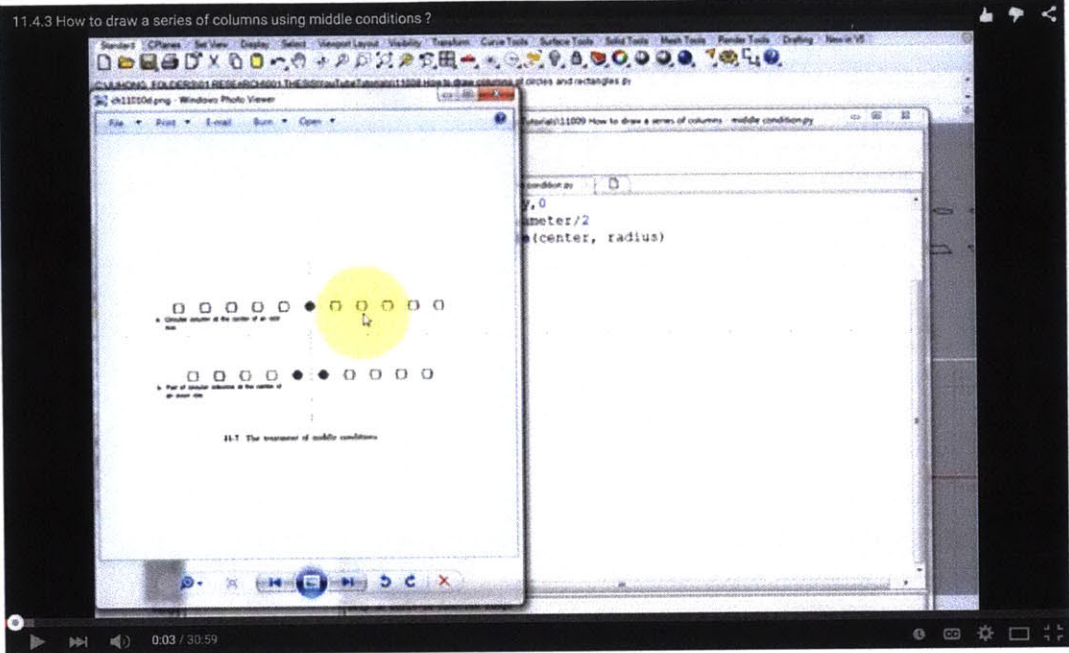
main()

```

RESULT



Synthetic Tutor



Module 128: 11.5 Arbitrary

11. CONDITIONALS

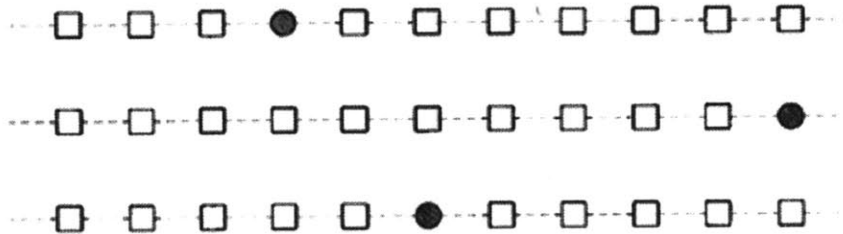
11.4 PAIRS OF DESIGN ALTERNATIVES

11.4.4 ARBITRARY EXCEPTIONS

Sometimes a designer wants to make an arbitrarily chosen element in a repetitive composition different from all the others. For example, an architect might want to make the *n*th column in a colonnade round instead of square. The following procedure takes *N* as one of its parameters and produces the required results.

[sample codes on the right side]

Some examples of output are shown in figure 11-8.



11-8. Rows with circular columns at arbitrarily chosen locations.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import random

def square(x, y, diameter):
    x1 = x - diameter / 2
    y1 = y - diameter / 2
    plane = [x1, y1, 0]
    rs.AddRectangle(plane, diameter, diameter)

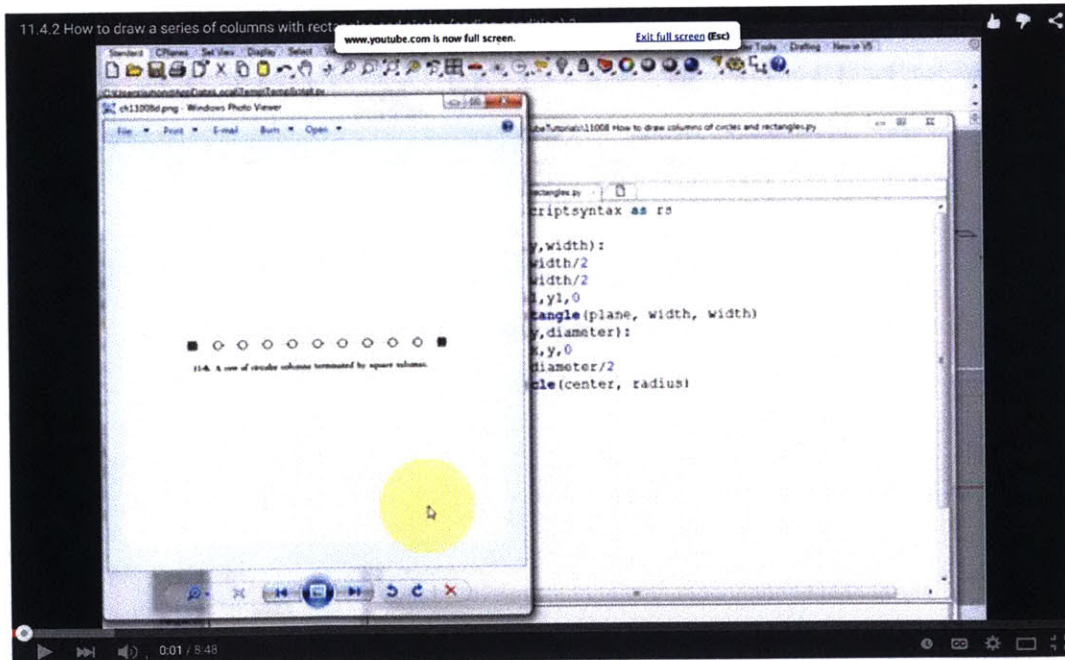
def circle(x, y, diameter):
    center = [x, y, 0]
    radius = diameter / 2
    rs.AddCircle(center, radius)

def odd(number):
    if (number % 2) != 0:
        return True

def arbitrary_row(initial_x, y, diameter, spacing, n_columns, n):
    x = initial_x
    count = range(n_columns)
    for i in count:
        if i == n:
            circle(x, y, diameter)
        else:
            square(x, y, diameter)
            x=x+spacing

rnd = random.randint(0,11)
arbitrary_row(1, 1, 30, 100, 12, rnd)
```

RESULT



Module 129: 11.6 Selecting

11. CONDITIONALS

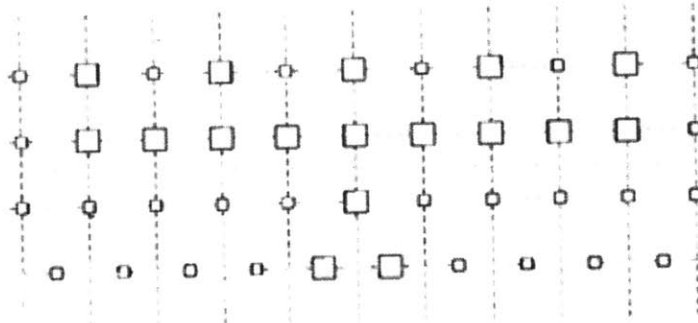
11.4 PAIRS OF DESIGN ALTERNATIVES

11.4.5 SELECTING SIZE AND SPACING

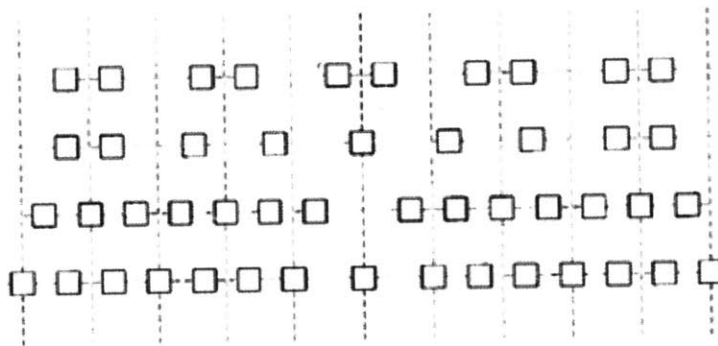
An architect might want to vary not only the type of column according to rules based on the value of the for-loop control variable, but also the diameter and spacing. For example, odd and even columns might be of different diameters, end columns might be smaller, or middle columns might be larger (fig. 11-9a). Spacing might be handled in similar ways (fig. 11-9b).

All this can be handled, straightforwardly, within the framework of if... else... statements. Here, for example, is a rule to vary the type, size, and spacing of odd and even columns (fig. 11-10):

```
if odd(count):
    circle(x, y, odd_diameter)
    x = x + odd_spacing
else:
    square(x, y, even_diameter)
    x = x + even_spacing
```

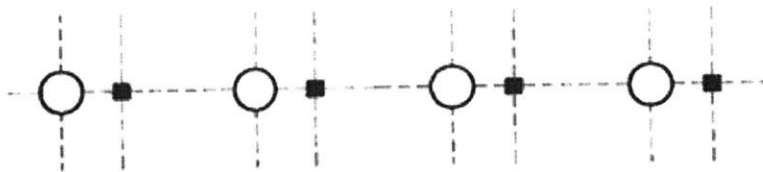


a. Size varied by parity, end, and center conditions.



b. Spacing varied by parity, end, and center conditions.

11-9. Further variations by condition.



11-10. Type, size, and spacing vary by parity.

Module 130: 11.7 If Then 1

11. CONDITIONALS

11.4 PAIRS OF DESIGN ALTERNATIVES

11.4.6 CHOOSING NO ACTION: if . . .

Sometimes a designer must choose between the alternatives of doing something and doing nothing. If walls are close together, for example, beams can economically span a space with nothing in between to support them (fig. 11-11a). But if walls are further apart, intermediate columns must be introduced (fig. 11-11b). The rule that applies here can be expressed:

[sample codes on the right side]

Span specifies the width of the space, and the value of Limit specifies when the span is sufficiently great to require columns.

This is an unnecessarily clumsy way of expressing the rule, though. When the decision to be made is whether to execute an action or not, Python allows contracted if statements of the form:

```
if Boolean expression:
    Statement
```

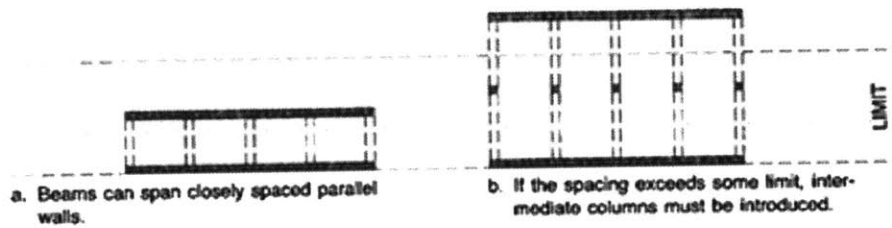
So our rule could be rewritten more clearly and cogently as:

```
if ( span < limit ):
```

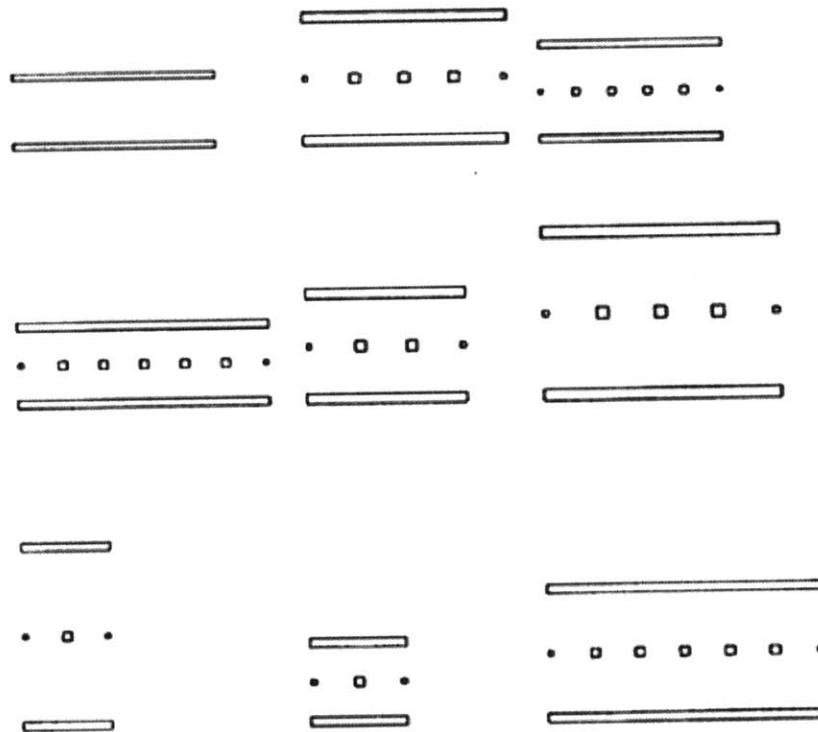
To illustrate the use of this rule, the following interactive program prompts for and reads in the length and span of a space, and the number of equally spaced beams that are to span between the walls. If the span is greater than the distance between beams, it introduces columns. There are also proportioning rules walls and interior columns are one-fifth as wide as the distance between beams, and end columns are one-tenth as wide (since they support only half as much roof area as interior columns).

[sample codes on the right side]

Figure 11-12 shows some examples of output from this program.



11-11. The genesis of the hypostyle hall, as explained in Auguste Choisy's *Histoire de l'Architecture*, 1899.



11-12. Plans of hypostyle halls generated by the program Support.

CODE

```

import rhinoscriptsyntax as rs

def end_condition(count, first, last):
    if (count == first) or (count == last):
        return True
    else:
        return False

def wall(x, y, length, width):
    x2 = x + length
    y2 = y + width

    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def square(x, y, diameter):
    x1 = x - diameter / 2
    y1 = y - diameter / 2
    plane = [x1, y1, 0]
    rs.AddRectangle(plane, diameter, diameter)

def row(initial_x, y, diameter, spacing, n_columns):
    x = initial_x
    half_diameter = diameter / 2
    count = range(1, n_columns + 1)
    for i in count:
        if end_condition(i, 1, n_columns):
            square(x, y, half_diameter)
        else:
            square(x, y, diameter)
        x = x + spacing

def main():
    length = rs.GetReal('enter length of space', 100)
    span = rs.GetReal('enter width of space', 30)
    n_beams = rs.GetInteger('enter number of beams', 20)
    spacing = length / (n_beams - 1)
    diameter = 0.2 * spacing
    x = 100
    y = 100
    y1 = y + span + diameter

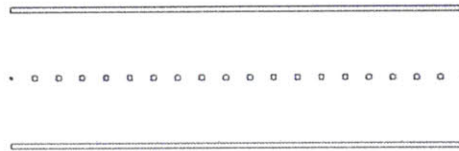
    wall(x, y, length, diameter)
    wall(x, y1, length, diameter)

    if (span > spacing):
        y = y + span / 2 + diameter
        row(x, y, diameter, spacing, n_beams)

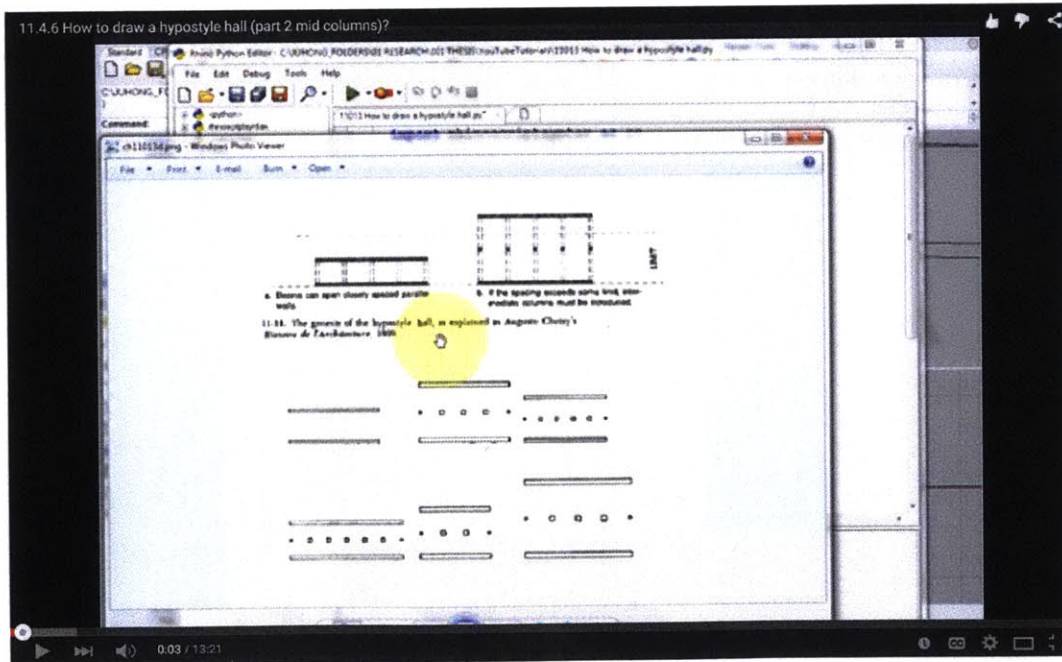
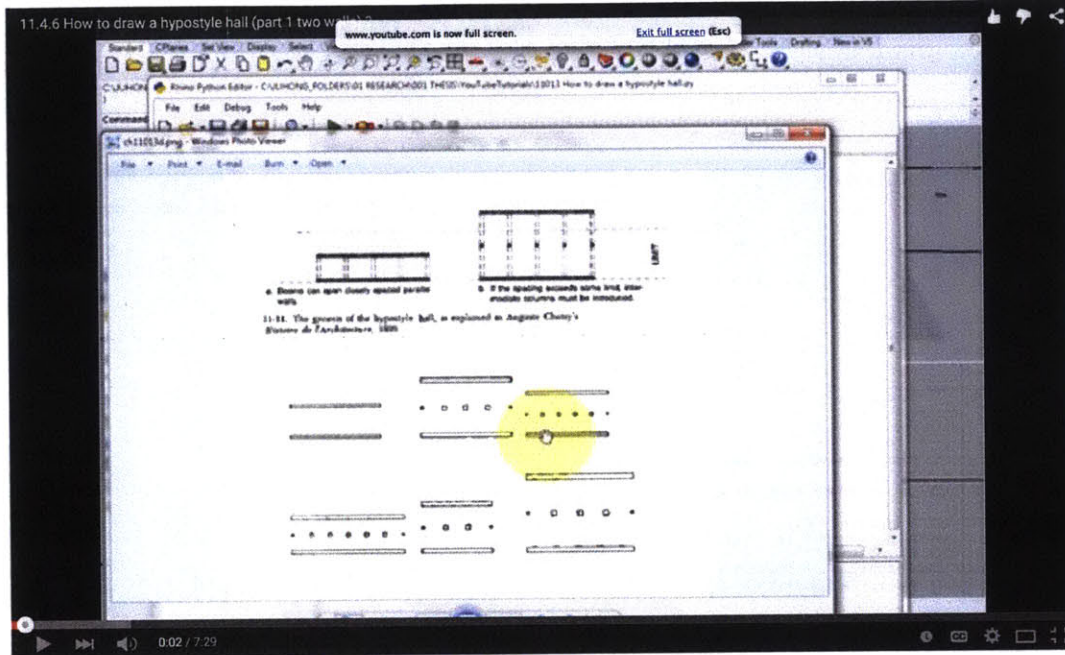
main()

```

RESULT



Synthetic Tutor



Module 131: 11.7 If Then 2

11. CONDITIONALS

11.4 PAIRS OF DESIGN ALTERNATIVES

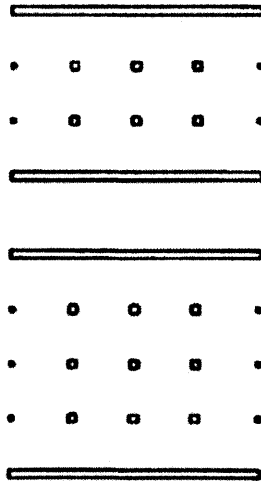
11.4.6 CHOOSING NO ACTION: if . . .

This program assumes that one central row of columns will suffice. A more general program might first determine how many rows were needed, then insert the required number. The following function takes Span and Limit as parameters and returns the required number of rows:

[sample codes on the right side]

Assuming equal spans, this next procedure now draws the required number of rows (fig. 11-13).

[sample codes on the right side]



11-13. Hypostyle hall with multiple rows of columns.

Synthetic Tutor

CODE

```

import rhinoscriptsyntax as rs

def end_condition(count, first, last):
    if (count == first) or (count == last):
        return True
    else:
        return False

def square(x, y, diameter):
    x1 = x - diameter / 2
    y1 = y - diameter / 2
    plane = [x1, y1, 0]
    rs.AddRectangle(plane, diameter, diameter)

def row(initial_x, y, diameter, spacing, n_columns):
    x = initial_x
    half_diameter = diameter / 2
    count = range(1, n_columns + 1)
    for i in count:
        if end_condition(i, 1, n_columns):
            square(x, y, half_diameter)
        else:
            square(x, y, diameter)
        x = x + spacing

def wall(x, y, length, width):
    x2 = x + length
    y2 = y + width

    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def main():
    length = rs.GetReal('enter length of space', 100)
    span = rs.GetReal('enter width of space', 30)
    n_beams = rs.GetInteger('enter number of beams', 20)
    spacing = length / (n_beams - 1)
    diameter = 0.2 * spacing
    x = 100
    y = 100
    y1 = y + span + diameter

    wall(x, y, length, diameter)
    wall(x, y1, length, diameter)

    if (span > spacing):
        y = y + span / 2 + diameter
        row(x, y, diameter, spacing, n_beams)

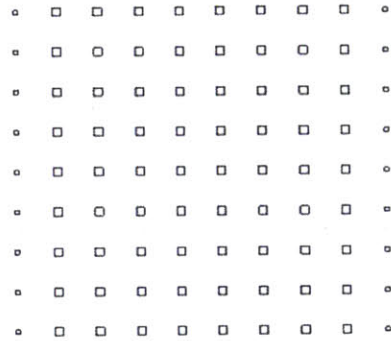
def end_condition(count, first, last):
    if (count == first) or (count == last):
        return True
    else:
        return False

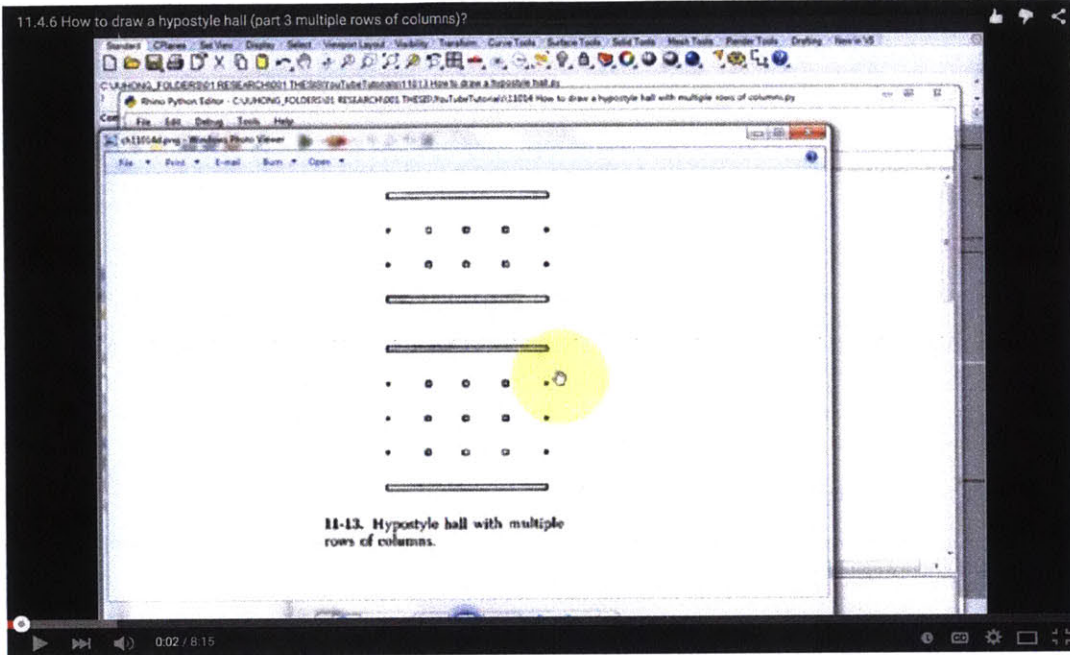
def square(x, y, diameter):
    x1 = x - diameter / 2
    y1 = y - diameter / 2
    plane = [x1, y1, 0]
    rs.AddRectangle(plane, diameter, diameter)

def n_rows(span, limit):
    result = int((span-1)/limit)
    return result

```

RESULT





Module 132: 11.7 If Then 3

11. CONDITIONALS

11.4 PAIRS OF DESIGN ALTERNATIVES

11.4.6 CHOOSING NO ACTION: if . . .

We have now produced a program to generate a well-known plan type (found, for example, in ancient Egyptian architecture) known as the hypostyle hall. Figure 11-14a illustrates some instances of another familiar and very similar plan type, that of the classical temple. Here we have a column grid surrounding a cella, rather than parallel walls surrounding a column grid. Columns in an interior rectangle are left out to provide room for the cella.

The limits of the cella can be specified by values for X_start , X_finish , Y_start , Y_finish (fig. 11-14b). The following function is true for a grid point outside the cella and false for a grid point within it:

[sample codes on the right side]

The column grid can be drawn by a pair of nested loops (see chapter 9), and space for the cella can be left out of the center of the grid by application of the rule:

[sample codes on the right side]

Here is a complete procedure to draw column grids with appropriate space left for the cella (fig. 11-15).

[sample codes on the right side]

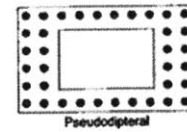
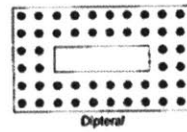
The nested loops enumerate column positions. The subset of positions at which columns are actually drawn is the union of the four subsets specified by the four conditions:

```
COUNT_X < X_START  
COUNT_X > X_FINISH  
COUNT_Y < Y_START  
COUNT_Y > Y_FINISH
```

This is illustrated in figure 11-15.

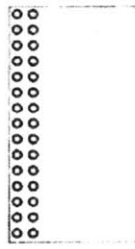


a. Type diagram for the column grid.

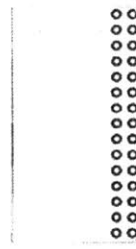


b. Some traditional subtypes.

11-14. The classical temple plan.



a. Count_X < X_start



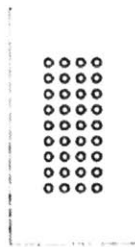
b. Count_X > X_start



c. Count_Y < Y_start



d. Count_Y > Y_finish



e. Excluded columns.

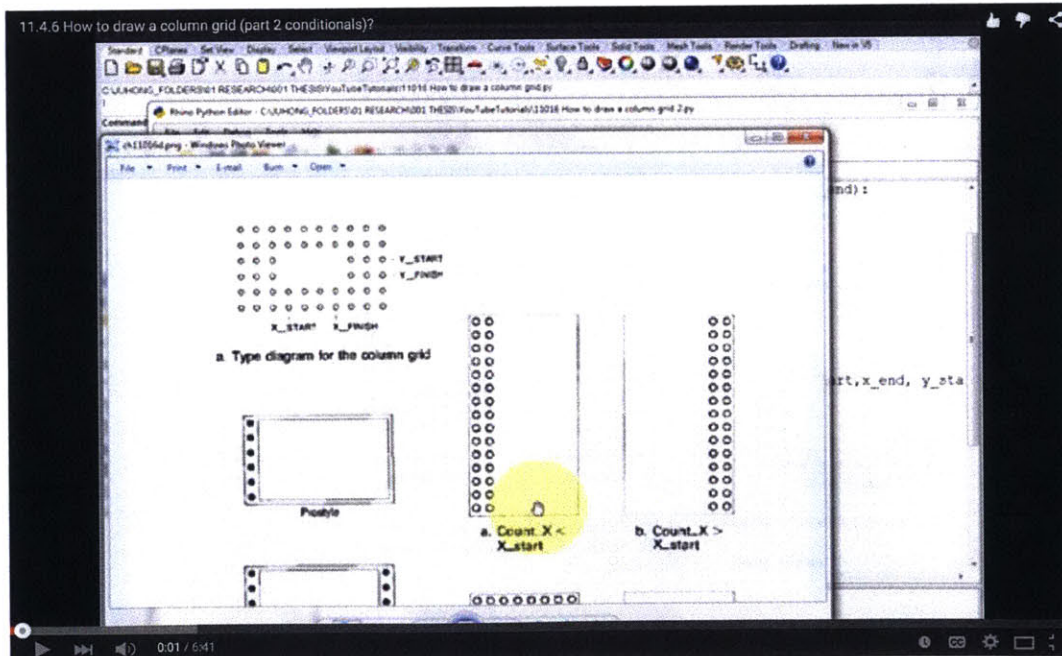


f. Included columns

11-15. Column grid of the Temple of Apollo at Didyma, drawn by the procedure Grid.

Synthetic Tutor

CODE	RESULT
<pre> import rhinoscriptsyntax as rs def column(x, y, diameter): xl = x - diameter / 2 yl = y - diameter / 2 plane = [xl, yl, 0] rs.AddRectangle(plane, diameter, diameter) def outside(count_x, count_y, x_start, x_finish, y_start, y_finish): cond1 = count_x < x_start cond2 = count_x > x_finish cond3 = count_y < y_start cond4 = count_y > y_finish outside_cond = cond1 or cond2 or cond3 or cond4 return outside_cond def grid(initial_x, initial_y, diameter, spacing, nx, ny, x_start, x_finish, y_start, y_finish): y = initial_y repeat_nx = range(1, nx+1) repeat_ny = range(1, ny+1) y = initial_y for count_y in repeat_ny: x = initial_x for count_x in repeat_nx: condition = outside(count_x, count_y, x_start, x_finish, y_start, y_finish) if condition: column(x, y, diameter) x = x + spacing y = y + spacing grid(0,0, 200,1000, 10,8, 3,8, 3,6) </pre>	



Module 133: 11.5 Conditional

11. CONDITIONALS

11.5 CONDITIONAL CHOICES AMONG MANY DESIGN ALTERNATIVES

There are often more than two design alternatives to choose from. Else if chains and the case statement are two ways to express rules for choosing from many alternatives.

11.5.1 elif (else if) CHAINS

The following type of statement chooses one alternative from n possibilities:

```
if Boolean expression 1:  
    Statement 1  
elif Boolean expression 2:  
    Statement 2  
elif Boolean expression 3:  
    Statement 3  
else:  
    Statement n
```

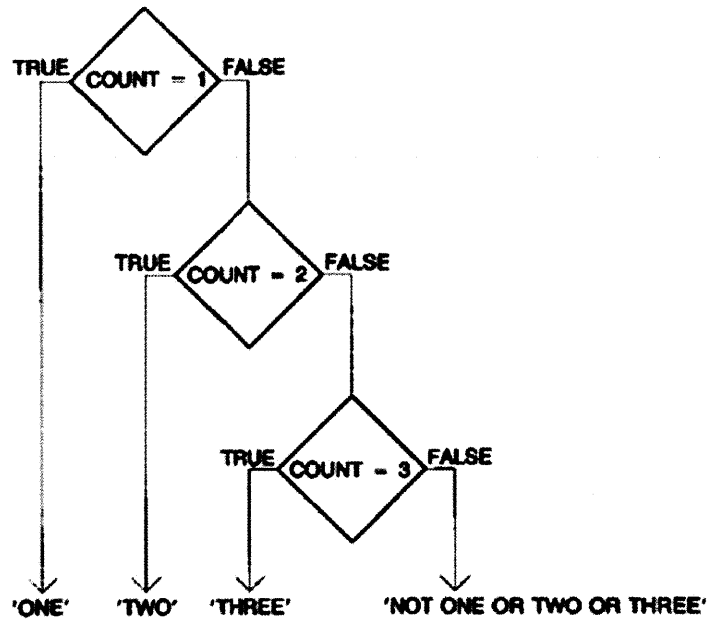
The computer evaluates each Boolean expression in order. If it finds a value of true, it executes the associated statement, then jumps to the statement following the elif chain.

When none of the Boolean expressions turns out to have a value of true the statement following the last else is executed. This last statement, then, specifies what happens in a none-of-the-above situation. The following code illustrates this:

```
if count == 1:  
    print 1  
elif count == 2:  
    print 2  
elif count == 3:  
    print 3  
else:  
    print 0
```

The structure of this code is illustrated in figure 11-16.

If you want to do nothing when none of the Boolean expressions turns out to be true, you can simply omit the last else and the associated statement:



11-16. A flow diagram of an else if chain.

Module 134: 11.5 Grid

11. CONDITIONALS

11.5 CONDITIONAL CHOICES AMONG MANY DESIGN ALTERNATIVES

11.5.2 CONDITIONS IN A COLUMN GRID

To illustrate the use of elif chains in graphics programs, let us consider the column grid shown in figure 11-17. One architect might make all the columns the same (fig. 11-17a). Another architect after more subtle effects might recognize that columns on the face support only half as much load as columns in the interior, and that columns at the corners support only a quarter as much. There are other architectural differences, too. Operating on the principle that form follows function, then, the second architect might respond to each of these conditions in a different way (fig. 11-17b). How can we write Python code to express the rules that are followed here?

First, we need Boolean functions to identify the conditions that concern us. Here, then, is a function Corner:

[sample codes on the right side]

And here is a function Face:

[sample codes on the right side]

The following procedure executes a pair of nested for-loops to generate the column grid and an elif chain to select small square columns for the corners, large square columns for the faces, and circular columns everywhere else:

[sample codes on the right side: Grid]

Some of the results generated by this procedure for different-sized grids are shown in figure 11-18.

This procedure is easily altered to associate different responses with the same conditions. The following elif chain, for example, places cross-shaped columns at the corners, circular columns along the faces, and square columns in the interior (fig. 11-19):

```

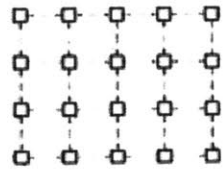
if corner(count_x, count_y, nx, ny):
    cross(x, y, diameter)
elif face(count_x, count_y, nx, ny):
    circle(x, y, diam4)
else:
    square(x, y, diameter)

```

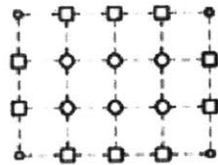
We might be interested in responding to different conditions. The following procedure, for example, uses the built-in Boolean function odd to generate a checkerboard pattern of square and circular columns (fig. 11-20):

[sample codes on the right side: Grid]

Sophisticated designers sometimes use much more complex rules to inflect column grids than the elementary ones we have explored here. Consider, for example, Alvar Aalto's plan for the editorial offices of the Turun Sanomat in Turku (fig. 11-21). It begins as a regular grid, then some rows are eliminated and others are offset. Finally, circular, square, rectangular, and bullet-shaped columns are introduced according to relations with walls and stairs and use of the adjacent floor area.

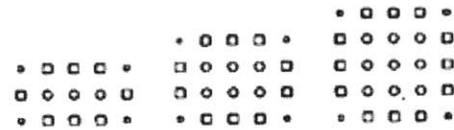
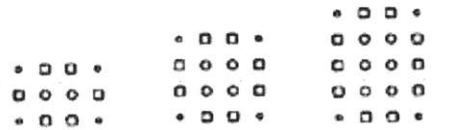
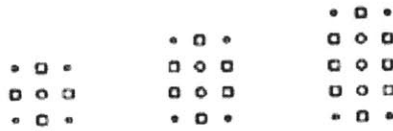


a. Uniform columns.



b. Corner, face, and interior columns take different forms.

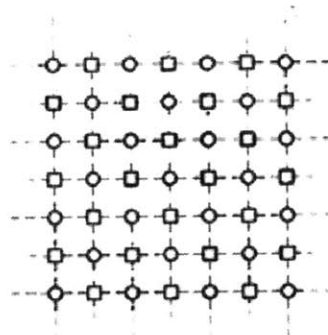
11-17. The inflection of a column grid.



11-18. Column grids of various sizes with different corner, face, and interior columns.



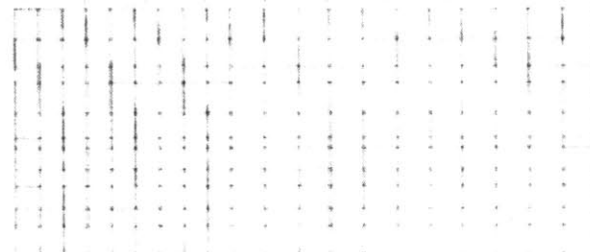
11-19. Different responses associated with the same conditions in the grid.



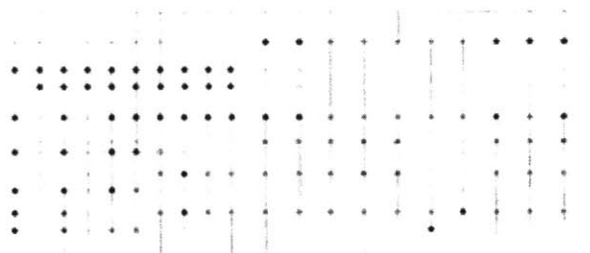
11-20. A checkerboard pattern of square and circular columns.



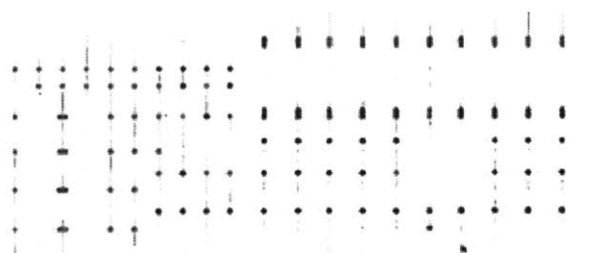
a. Two simple, regular rhythms.



b. The grid becomes more complex.



c. Columns occupy some, but not all, of the grid locations.

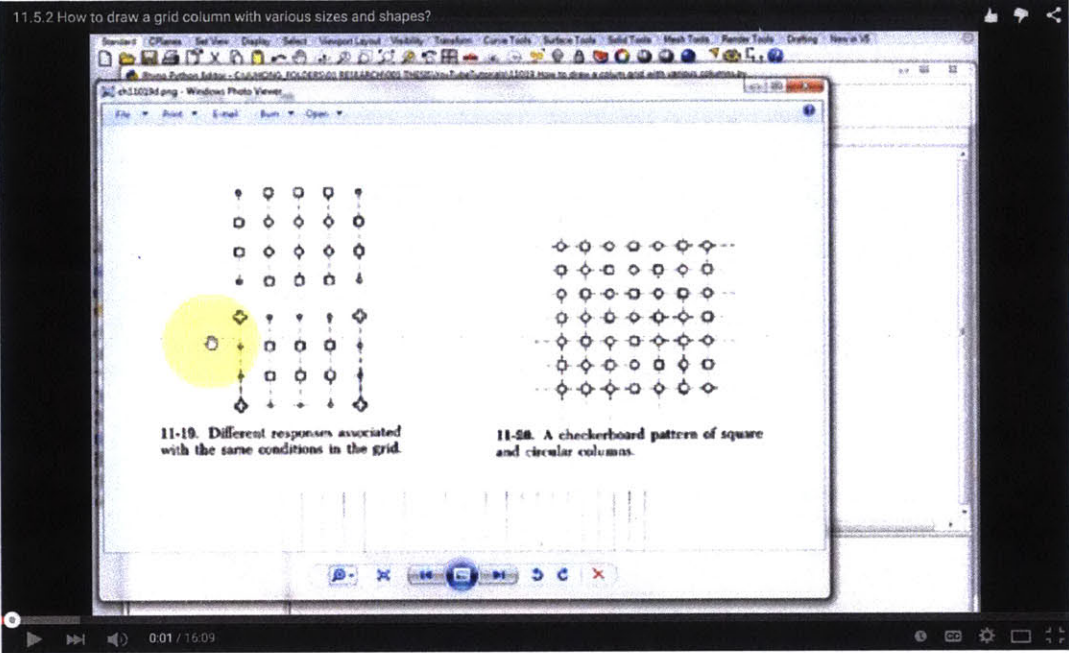


d. Column types vary.

11-21. Entrance-level plan of the editorial offices of the *Turun Sanomat* newspaper, Turku, Finland by Alvar Aalto, 1927/29.

Synthetic Tutor

CODE	RESULT
<pre> import rhinoscriptsyntax as rs def square(x, y, diameter): xl = x - diameter / 2 yl = y - diameter / 2 plane = [xl, yl, 0] rs.AddRectangle(plane, diameter, diameter) def circle(x,y,diameter): center = [x,y,0] radius=diameter rs.AddCircle(center,radius) def corner(count_x, count_y, nx, ny): cond1 = count_x == 1 and count_y == 1 cond2 = count_x == 1 and count_y == ny cond3 = count_x == nx and count_y == 1 cond4 = count_x == nx and count_y == ny is_corner = cond1 or cond2 or cond3 or cond4 return is_corner def face(count_x, count_y, nx, ny): condX = count_x == 1 or count_x==nx condY = count_y == 1 or count_y==ny is_face = condX or condY return is_face def grid(initial_x, initial_y, diameter, spacing, nx, ny): diam4 = diameter / 4 y = initial_y repeat_nx = range(1,nx+1) repeat_ny = range(1,ny+1) for count_y in repeat_ny: x = initial_x for count_x in repeat_nx: if corner(count_x, count_y, nx, ny): square(x, y, diam4) elif face(count_x, count_y, nx, ny): square(x, y, diameter) else: circle(x, y, diameter) x = x + spacing y = y + spacing grid(0,0, 100, 1000, 8,6) </pre>	



Module 135: 11.5 Rhythms 1

11. CONDITIONALS

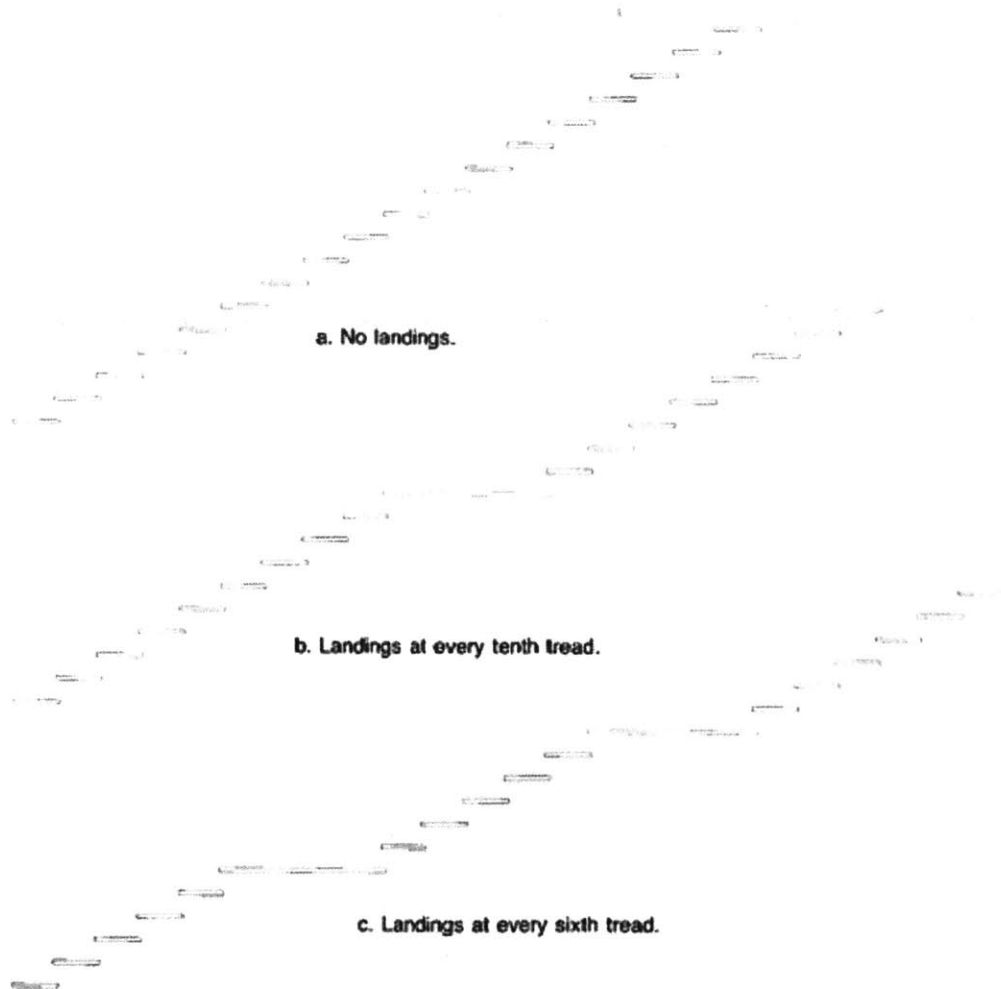
11.5 CONDITIONAL CHOICES AMONG MANY DESIGN ALTERNATIVES

11.5.3 USE OF MODULAR ARITHMETIC TO GENERATE RHYTHMS

Consider the very long stair shown in figure 11-25a. It needs landings an architect might decide that a landing is to be inserted in place of every n th tread, where n is some specified positive integer. The Python `mod (%)` function, which, as you will recall from chapter 5, computes the remainder of the division of two integer factors, provides a convenient way to accomplish this. The following procedure takes N as a parameter and computes the value of $i \bmod N$ to decide when to insert a landing:

[sample codes on the right side]

Some typical results for different values of N are shown in figure 11-25b and c.



11-25. The insertion of landings into stairs.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

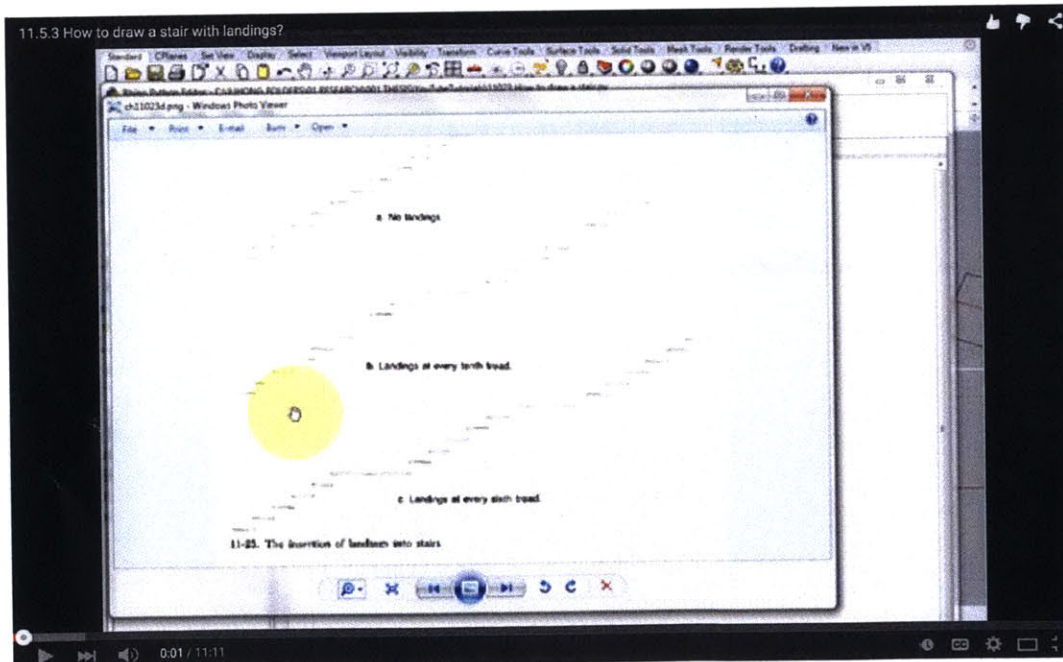
def rectangle(x1, y1, length, width):
    x2 = x1 + length
    y2 = y1 + width
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def stairs(x_initial, y_initial,
          depth, width,
          x_increment, y_increment,
          num_of_stairs, n):

    landing_depth = 2 * depth
    x = x_initial
    y = y_initial
    count = range(1, num_of_stairs+1)
    for i in count:
        if (i % n == 0):
            rectangle(x, y, landing_depth, width)
            x = x + depth
        else:
            rectangle(x, y, depth, width)
            x = x + x_increment
            y = y + y_increment

stairs(0, 0, 300, 20, 300, 200, 10, 10)
```

RESULT



Module 136: 11.5 Rhythms 2

11. CONDITIONALS

11.5 CONDITIONAL CHOICES AMONG MANY DESIGN ALTERNATIVES

11.5.3 USE OF MODULAR ARITHMETIC TO GENERATE RHYTHMS

Similar logic might be followed in fenestration. Figure 11-26, for example, shows a row of square windows in which every *n*th square has been replaced by a larger, pedimented window. This is simply a generalization of the idea of alternating window types, which was discussed earlier.

A further generalization, familiar to musicians, is to consider the possibilities for rhythms of different period. The simplest possible rhythm of period 1 has the form:

AAA ...

When the period is 2, an additional possibility emerges:

ABABAB ...

A period of 3 gives us:

ABBABBABB..
ABCABCABC..

The possibilities now begin to multiply rapidly. Going to a period of 4 adds the possibilities:

ABBBABBBABBBABBB ...
AABBAABBAABBAABB ...
ABBCABBCABBCABBC ...
ABCBABCABCABCABC ...
etc.

This enumeration can, of course, continue indefinitely.

Procedures to generate compositions that have complex rhythmic structures can be written, very simply, using the `mod (%)` function together with an `if...elif...else...` statement. The next procedure, for example, generates rows of triangular pedimented square, and rectangular windows in the following rhythm.

ACBBCAACBBCAACBBCA

Here is the code:

[sample codes on the right side]

Notice that the value of `i % (mod) 6` is computed at each iteration, and the result is then used to select a case. The case statement establishes the structure of the rhythm by associating a window type with each of the possible values (fig. 11-27).



11-26. The use of modular arithmetic to vary every *n*th window in a row.



11-27. A complex fenestration rhythm generated by using the **mod** function and a **case** statement.

CODE

```

import rhinoscriptsyntax as rs

def rectangular_window(xc, yc, length, width):
    x1 = xc - (length / 2)
    x2 = x1 + length
    y1 = yc - (width / 2)
    y2 = y1 + width * 1.3

    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def triangular_window(xc, y1, width, height):
    x1 = xc - (width / 2)
    x2 = x1 + width
    y0 = height
    y2 = y1 + height*1.3

    pt1 = [x1, y0, 0]
    pt2 = [x2, y0, 0]
    pt3 = [xc, y2, 0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt1)

    rectangular_window(xc, y1, width, height)

def square_window(x, y, diameter):
    x1 = x - diameter / 2
    y1 = y - diameter / 2
    plane = [x1, y1, 0]
    rs.AddRectangle(plane, diameter, diameter)

def row_windows(x, y, width, height, spacing, n_of_windows):
    y_square = y + ((height-width) / 2)
    count = range(1, n_of_windows + 1)

    for i in count:
        mod = i % 6
        if (mod == 0 or mod == 1):
            triangular_window(x, y, width, height)
        elif (mod == 2 or mod == 5):
            square_window(x, y_square, width)
        else:
            rectangular_window(x, y, width, height)
        x = x + spacing

```

RESULT



Module 137: 11.5 Random

11. CONDITIONALS

11.5 CONDITIONAL CHOICES AMONG MANY DESIGN ALTERNATIVES

11.5.5 RANDOM CHOICE

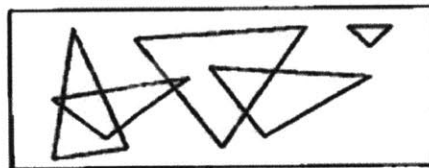
Random choice is relatively rare in architecture; however, it does occasionally occur, Figure 11-29, for example, shows a traditional motif of Chinese garden architecture. Windows are regularly spaced in a row along a walkway wall, but there are several different window types, and the choices for each location do not fall into any regular pattern,

Random number generators can be used to choose not only between different vocabulary elements, but also to choose shape and position parameter values. The following program generates compositions of a specified number of random triangles within a specified rectangle on the screen (fig 11-30).

[sample codes on the right side]



11-29. A Chinese garden-wall motif.



11-30. A composition of random triangles.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import random

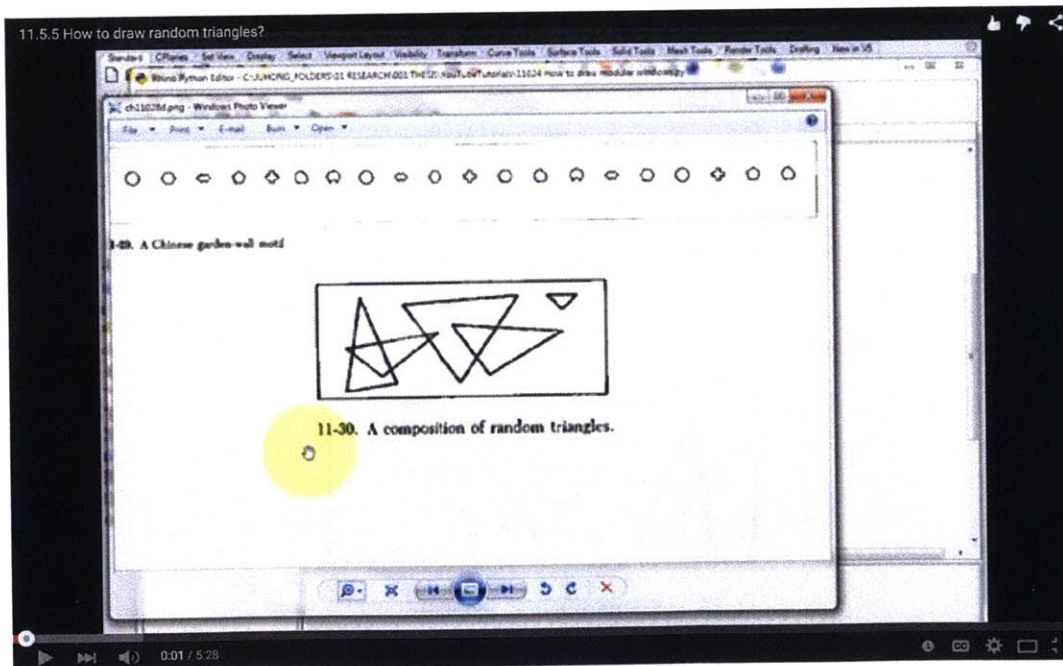
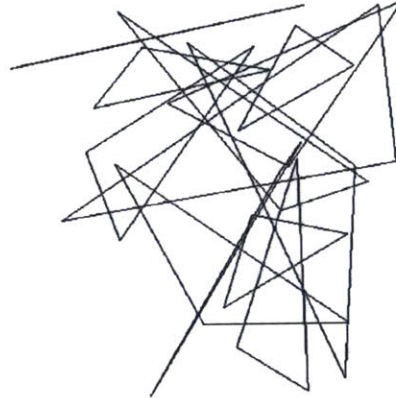
def triangle(x1, y1, x2, y2, x3, y3):
    pt1 = [x1, y1, 0]
    pt2 = [x2, y2, 0]
    pt3 = [x3, y3, 0]
    pts = [pt1, pt2, pt3, pt1]
    rs.AddPolyline(pts)

def random_triangles():
    min_x = rs.GetInteger('enter minimum x coordinate', 1)
    max_x = rs.GetInteger('enter maximum x coordinate', 100)
    min_y = rs.GetInteger('enter minimum y coordinate', 1)
    max_y = rs.GetInteger('enter maximum y coordinate', 60)
    n = rs.GetInteger('how many triangles?', 12)
    count = range(1, n+1)

    for i in count:
        x1 = random.randrange(min_x, max_x)
        x2 = random.randrange(min_x, max_x)
        x3 = random.randrange(min_x, max_x)
        y1 = random.randrange(min_x, max_x)
        y2 = random.randrange(min_x, max_x)
        y3 = random.randrange(min_x, max_x)
        triangle(x1, y1, x2, y2, x3, y3)

random_triangles()
```

RESULT



Module 138: 11.6 Other

11. CONDITIONALS

11.6 OTHER APPLICATIONS OF CONDITIONALS

So far we have seen how to use if... else... statements to choose between design alternatives. But they also have many other important uses. They can be used for error checking, in situations that arise with the construction of curves, and in programs that search for design alternatives that meet specified criteria.

11.6.1 ERROR CONDITIONS

It is often necessary to check values that are read in by a program, since you cannot control what a user of the program may enter in response to a prompt it might be some absurd value. In this case, you probably want to give the user the chance to reenter a correct value. A while...break...loop can be used for this purpose:

[sample codes on the right side]

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

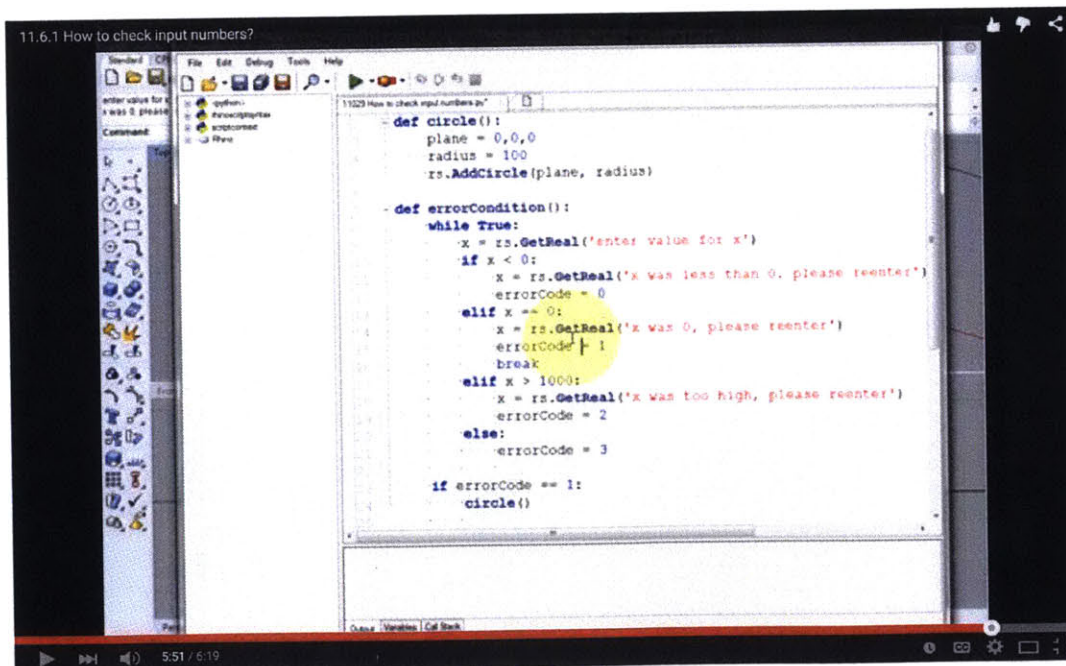
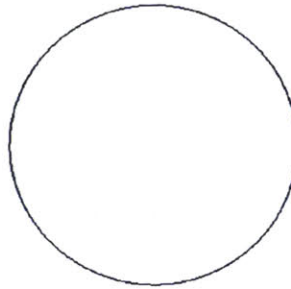
def circle(x, y, radius):
    plane=[x,y,0]
    rs.AddCircle(plane,radius)

def errorCondition():
    while True:
        error_code = 0
        x = rs.GetReal('enter value for x')
        if (x < 0):
            x = rs.GetReal('x less than 0: re-enter')
            error_code = 1
        elif (x == 0):
            break
        elif (x > 1023):
            x = rs.GetReal('x greater than 1023: re-enter')
            error_code = 2
        else:
            error_code = 3

    if (error_code == 0):
        circle(0,0,100)

errorCondition()
```

RESULT



Module 139: 11.6 Curves 111.
CONDITIONALS

11.6 OTHER APPLICATIONS OF CONDITIONALS

11.6.2 CURVES AND CONDITIONALS

Mathematical functions, like architectural compositions, often manifest special conditions (requiring special treatment) at their extremes. The following Python function, for example, computes the reciprocal of a number X

This function will work for any value of X except zero. When X is zero, the value of $1/X$ should be infinity, but an infinitely large integer cannot be represented in a finite memory location, so an attempt to execute Reciprocal when X is zero will generate an error. It is good practice, then, to test for a zero value of X before passing it into Reciprocal as follows:

[sample codes on the right side]

Synthetic Tutor

CODE

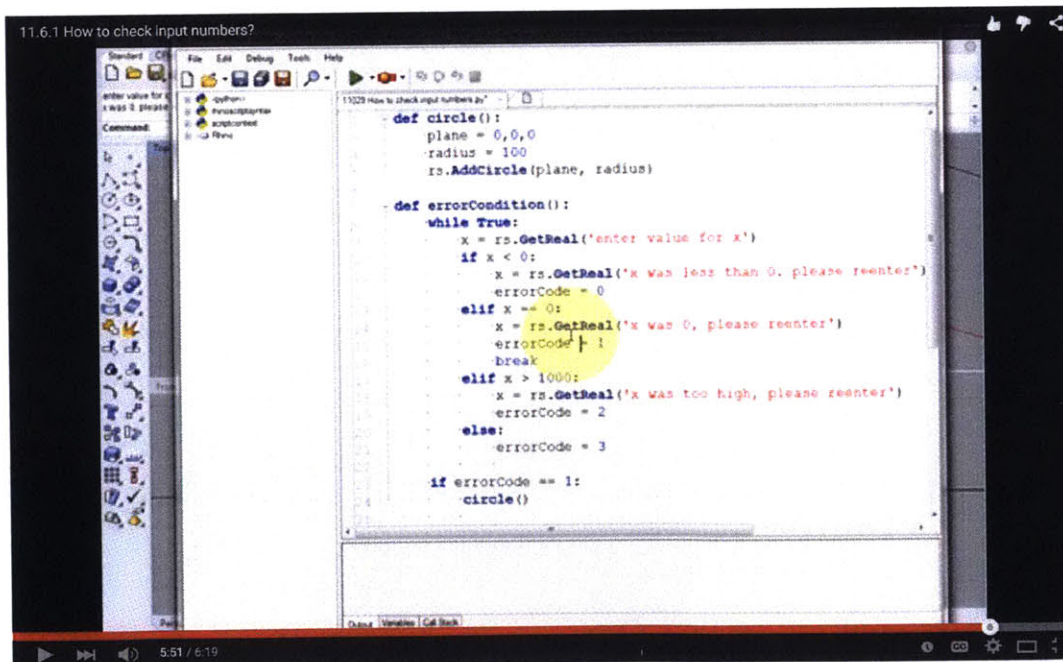
```
import rhinoscriptsyntax as rs

def reciprocal(x):
    if ( x != 0 ):
        y = 1 / x
        print y
    else:
        print 'Cannot take the reciprocal of 0'

reciprocal(10000)
reciprocal(0.00001)
reciprocal(0)
```

RESULT

```
0.0001
100000.0
Cannot take the reciprocal of 0
```



Module 140: Exercise 7

11.
CONDITIONALS

EXERCISES

1. Assume there is a procedure to draw a Square, a procedure to draw a Circle, and the Boolean functions `end_condition` and `center_point` as defined earlier. Execute the following code by hand and draw the graphic output on grid paper.

```
def end_condition( count, first, last ):
    result = (count == first) or (count == last)
    return result

def center_point( count, first, last ):
    middle = ( last + first ) / 2
    result = ( count == middle )
    return result

initial_x = 100
initial_y = 100
nx = 5
ny = 5
diameter = 25
spacing = 100

y = initial_y

for y_count in range( 1 to ny+1 ):
    x = initial_x

    for x_count in range( 1 to nx + 1 ):
        cond1 = end_condition( y_count, 1, ny )
        cond2 = center_point ( x_count, 1, nx )
        cond3 = end_condition( x_count, 1, nx )
        cond4 = center_point ( y_count, 1, ny )

        if ( cond1 and cond2 or cond3 and cond4 ):
            circle ( x, y, diameter )
        else:
            square ( x, y, diameter )

        x = x + spacing

    y = y + spacing
```

Keep the same loop structure, replace the conditional with the following, and draw the output that results.

```
cond1 = center_point( x_count, 1, nx )
cond2 = center_point( y_count, 1, ny )

if ( cond1 and cond2 ):
    circle( x, y, diameter )
else:
```

Synthetic Tutor

```
square ( x, y, diameter )
```

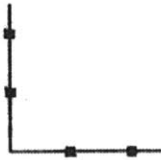
Finally, draw the output that results when the conditional is replaced with:

```
cond1 = ( x_count == y_count )  
cond2 = ( x_count == ny - ny_count + 1 )
```

```
if ( cond1 or cond2 ):  
    circle( x, y, diameter )  
else:  
    square( x, y, diameter )
```

Please upload your python file: No file chosen

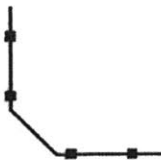
2. A common way to handle the corners in a high-rise office building is to leave out the corner columns and cantilever the floor (fig. 11-44a). This allows the outline of the floor to be shaped freely at the corners (fig. 11-44b, c, and d). Choose a way to shape the corner and write a general procedure to produce floor plans of this type.



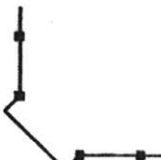
a. Cantilevered corner.



b. Rounded corner.



c. Chamfered corner.

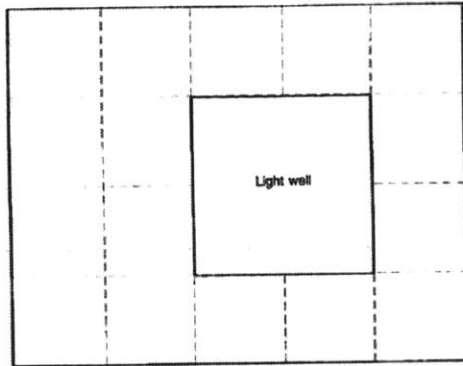


d. Projecting corner.

11-44. Corner treatments for a high-rise office building.

Please upload your python file: No file chosen

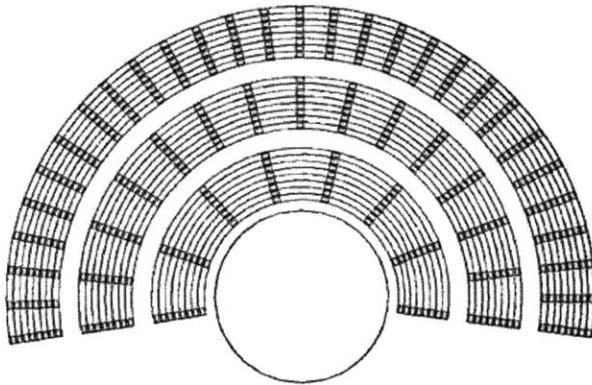
3. In the floor plan shown in figure 11-45, a rectangular light well has been introduced. The reentrant corners of the light well constitute a special condition in the column grid. What architectural response might you make to this? Using conditionals, write a general procedure that generates a plan with a rectangular light well of arbitrary size and arbitrary location within the column grid and that handles the reentrant corners appropriately.



11-45. A floor plan with a square column grid and an interior light well.

Please upload your python file: No file chosen

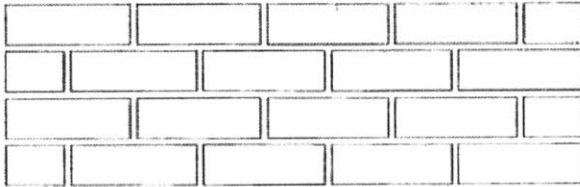
4. A scheme for the layout of an amphitheater is shown in fig. 11-46. Notice that concentric aisles for access alternate at regular intervals between the concentric rows of seating, and that a conditional rule governs placement of the radial lines of steps. Taking stage diameter, seating row width, aisle width, step width, number of rows of seating, and number of seating rows per aisle as parameters, and using conditionals, write a general procedure to draw layouts of this type.



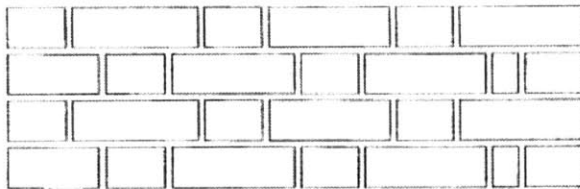
11-46. Type diagram for an amphitheater.

Please upload your python file: No file chosen

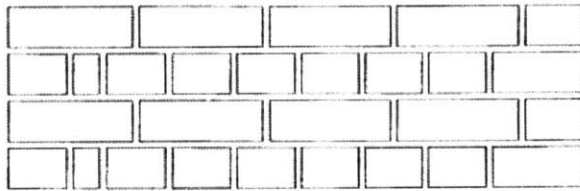
5. Figure 11-47 illustrates various traditional types of brickwork bonding. Choose one of these and consider the parameters that would be needed to control the position, size, and pattern of a rectangular panel in this bond. Using conditionals, write an appropriately parameterized procedure to generate such a panel.



a. Running.



b. Flemish.



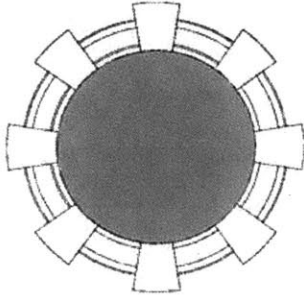
c. English.

11-47. Traditional types of brickwork bonding.

Please upload your python file: No file chosen

6. A window design by the eighteenth-century architect Batty Langley is shown in figure 11-48. There are eight identical voussoirs in this version, but the number of voussoirs might become a variable, and the shapes of voussoirs might be varied in response to special conditions. Top might differ from bottom, horizontal from vertical, odd from even, and so on. Write a procedure, incorporating conditionals, to generate variants.

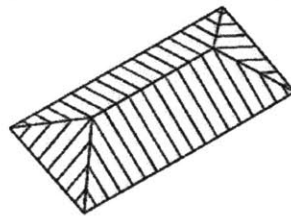
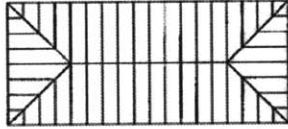
Synthetic Tutor



11-48. A circular window design from *Batty Langley's Builder's and Workman's Treasury of Designs*.

Please upload your python file: No file chosen

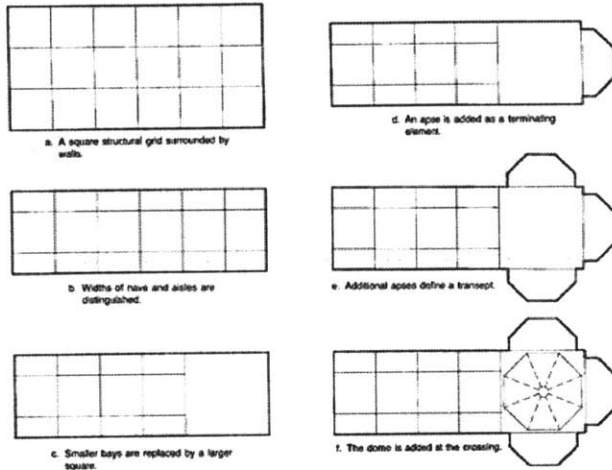
7. A standard scheme for the layout of rafters in a hipped roof over a rectangular floor plan is shown in figure 11-49. Notice that if the rafters on the long side intersect the hip, they are short. Otherwise, they are of normal length. Write an appropriately parameterized procedure to lay out rafters in this fashion and draw the plan.



11-49. The layout of rafters in a hipped roof.

Please upload your python file: No file chosen

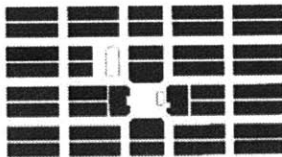
8. Figure 11-50 illustrates how the parti for a church, of the Florence Cathedral type, might be developed: this process can be understood as one of increasingly refined differentiation of an initially uniform design. Begin by writing a procedure to generate uniform designs of the type shown in figure 11-50a. Then, by successively introducing additional vocabulary elements and conditionals controlling their use, successively refine it into procedures to generate the more differentiated types.



11-50. Elaboration of the *partis* for Florence Cathedral.

Please upload your python file: No file chosen

9. Many town plans (fig. 11-51) consist of regular street grids interrupted at various points and in various ways. Examine some plans of this type. What are the conditions under which the grid is interrupted? Write a set of conditional rules that could be used to produce plans of this type, and discuss their effects.

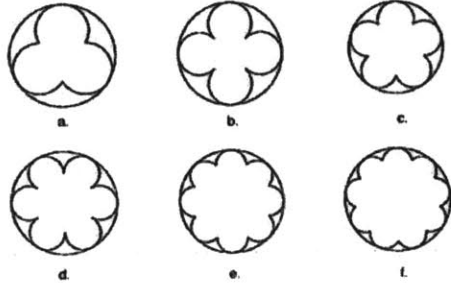


11-51. The regular street grid of the French bastide Montpazier, interrupted by major public buildings.

Please upload your python file: No file chosen

10. Figure 11-52a illustrates a common motif of Gothic architecture, composed of circles and arcs and known as the trefoil. Similarly, a quatrefoil is shown in figure 11-52b. In general, we can have multifoiils - any number of segments. What parameters are needed to control the position, size, segmentation, and shape of a multifoil? Write a general multifoil procedure.

Synthetic Tutor



11-52. Some multifoil designs.

Please upload your python file: No file chosen

Module 141: 11.6 Curves 1

11. CONDITIONALS

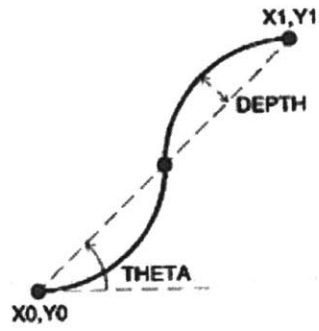
11.6 OTHER APPLICATIONS OF CONDITIONALS

11.6.2 CURVES AND CONDITIONALS

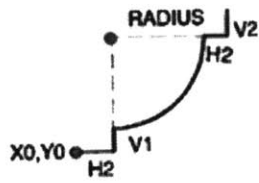
In the design of a molding, an architect manipulates shape variables to create effects of light, shading, and shadow line. Increasing the depth of an overhang, for example, creates deeper shadows. Varying the curvature of an arc alters the shading. And appropriate balances of flat and curved surfaces, and of light and shade, must be achieved. Figure 11-37a shows how an ovolo profile might be parameterized to allow this kind of design exploration. The corresponding procedure, which invokes Arc, is as follows:

[sample codes on the right side]

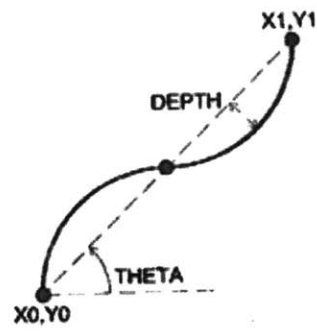
More complex types of molding profiles are formed by fitting together several arcs. The S-shaped cyma recta, for example, is constructed as shown in figure 11-40a. Its converse is the cyma reverse (fig. 11-40b). Both the cyma recta and the cyma reverse can be constructed from equal arcs, or they can be quirked by using unequal arcs (fig. 11-40c). In any case, their intersection must be tangential.



a. Cyma recta.



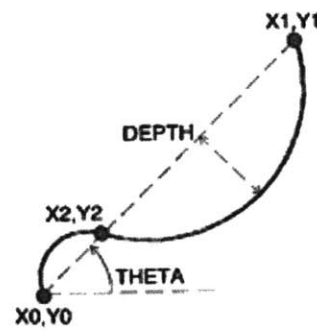
a. The type diagram.



b. Cyma reversa.



b. Some instances.



c. Quirked cyma reversa.

11-37. The parametric variation of the ovolo.

11-40. Arcs connected at tangents.

CODE

```
import rhinoscriptsyntax as rs
import math

def ovo1o(x0, y0, h1, v1, h2, v2, radius):
    xc = x0 + h1
    y1 = y0 + v1
    yc = y1 + radius
    y2 = yc + v2
    x1 = xc + radius + h2

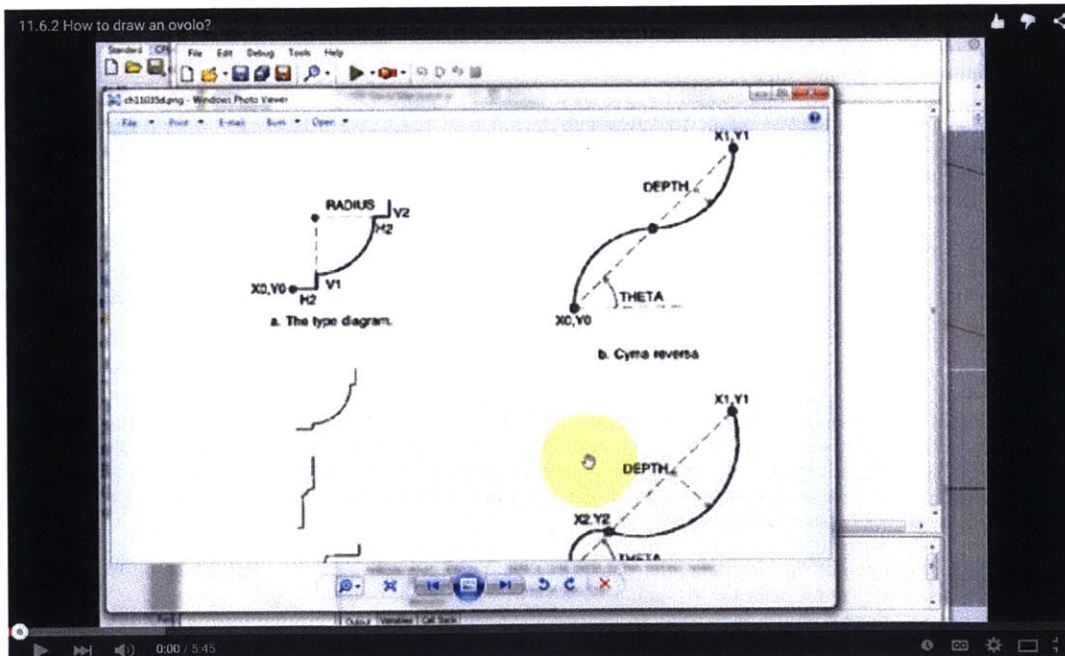
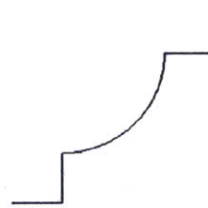
    pt0 = [x0, y0, 0]
    pt1 = [xc, y0, 0]
    pt2 = [xc, y1, 0]
    ptC = [xc, yc, 0]
    pt3 = [xc + radius, yc, 0]
    pt4 = [x1, yc, 0]
    pt5 = [x1, y2, 0]

    rs.AddLine(pt0, pt1)
    rs.AddLine(pt1, pt2)

    plane = [xc, yc, 0]
    angle = -90
    rs.AddArc(plane, radius, angle)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt5)

ovo1o(0,0, 10,10, 10,10, 20)
```

RESULT



Module 142: 11.6 Generates

11. CONDITIONALS

11.6 OTHER APPLICATIONS OF CONDITIONALS

11.6.3 GENERATE-AND-TEST PROCEDURES

Sometimes a designer knows conditions that an instance of some design element must satisfy, but does not know the parameter values that will generate a satisfactory instance. The problem, then, is to find these parameter values. A structural designer, for example, might intend to span a space with a rectangular wooden beam. The span and loading conditions are given, and the problem is to find satisfactory values for the height and width of the beam's cross section.

There are often formulas that can be evaluated to yield the required value directly. In other cases, however, there is no alternative to engaging in a trial-and-error process of generating candidate sets of parameter values for consideration and testing these for compliance with the conditions. The flow diagram in figure 11-41 illustrates the general structure of this process.

As the diagram suggests, it is not difficult to write programs to perform this kind of search. You use loops to enumerate parameter values, Boolean expressions to describe the conditions that must be met, and a conditional to terminate the search when the required values are found.

The next program illustrates this. It finds satisfactory values for Length and Width of a rectangular room. It begins by prompting for and reading in values for:

- . minimum acceptable Length
- . maximum acceptable Length
- . minimum acceptable Width
- . maximum acceptable Width
- . minimum acceptable Area
- . maximum acceptable Area
- . minimum acceptable Proportion (Length/Width)
- . maximum acceptable Proportion
- . minimum acceptable Perimeter
- . maximum acceptable Perimeter

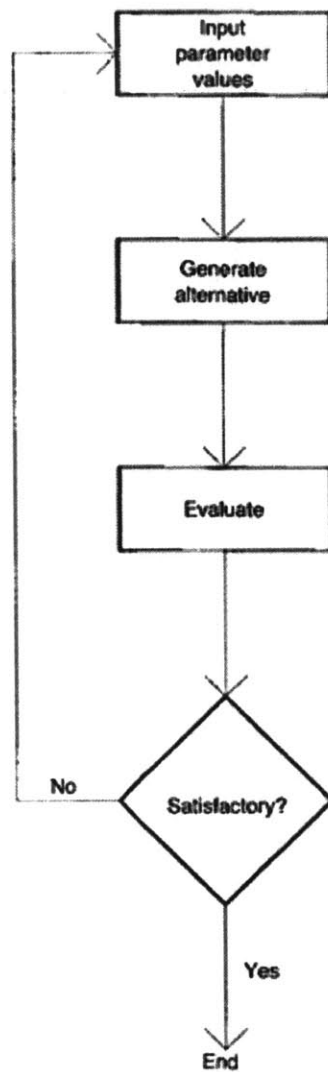
It then uses nested loops to enumerate values for Length and Width. At each iteration, it calculates values for Area, Proportion, and Perimeter. When it first finds Length and Width values that satisfy the requirements, it exits from the loop and draws the appropriately sized rectangle. If the requirements are impossible to satisfy, then it eventually exhausts all the possibilities and reports this. Here is the complete code:

[sample codes on the right side]

With the use of generate-and-test methods, you can build problem-solving intelligence into graphics programs. You should be careful, though, of the combinatorial explosion that results when the number of design variables, and the number of values possible for each, begins to grow. A computer can execute generate-and-test loops very rapidly, but each iteration does take a finite amount of time. When the number of iterations becomes very large, total execution time can easily become impractically great.

Notice that the execution of the Search program can be made more effective by adding logical conditions to the inner while loop. For example, once the Width reaches a value such that the calculated Area is greater than the maximum allowed, there is no reason to continue incrementing Width. In generate-and-test programs, it is important to take advantage of any available way to reduce the number of iterations. Where a motif is generated by a

parameterized procedure, you can always generate sets of variants by invoking the procedure from within loops that increment parameter values. And you can always turn such enumeration procedures into generate-and-test procedures by testing, within the loop, for compliance with conditions.



11-41. A flow diagram of a simple trial-and-error design process.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs

def rectangle(x, y, length, width):
    x2 = x + length
    y2 = y + width
    pt1 = [x, y, 0]
    pt2 = [x2, y, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def search():
    min_length = rs.GetInteger('enter the minimum length',100)
    max_length = rs.GetInteger('enter the maximum length',1000)
    min_width = rs.GetInteger('enter the minimum width',200)
    max_width = rs.GetInteger('enter the maximum width',2000)
    min_area = rs.GetInteger('enter the minimum area',1)
    max_area = rs.GetInteger('enter the maximum area',50000)
    min_proportion = rs.GetInteger('enter the minimum proportion',0.1)
    max_proportion = rs.GetInteger('enter the maximum proportion',10)
    min_perimeter = rs.GetInteger('enter the minimum perimeter',300)
    max_perimeter = rs.GetInteger('enter the maximum perimeter',1200)

    satisfies = False
    length = min_length - 1

    while (length < max_length) and not(satisfies) :
        length = length + 1
        width = min_width - 1

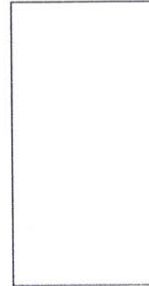
        while (width < max_width) and not(satisfies):
            width = width + 1
            area = length * width
            proportion = length / width
            perimeter = 2 * (length + width)
            cond1 = area > min_area
            cond2 = area < max_area
            cond3 = proportion > min_proportion
            cond4 = proportion < max_proportion
            cond5 = perimeter > min_perimeter
            cond6 = perimeter < max_perimeter

            if cond1 and cond2 and cond3 and cond4 and cond5 and cond6:
                satisfies = True

        if satisfies :
            rectangle(0, 0, length, width)
        else:
            print 'conditions not satisfied'

search()
```

RESULT



Module 143: 11.7 Summary

11. CONDITIONALS

11.7 SUMMARY

In the simplest Python programs, the computer executes each successive statement once and only once (fig. 11-42a):

```
def function( ):
    Statement 1
    Statement 2
    Statement 3
```

It neither repeats nor skips a statement.

In chapter 9, we explored the use of loops that cause a sequence of statements to be repeated (fig. 11-42b):

```
for i in range(10)
    function()
```

Whereas loops serve the graphic purpose of repeating instances of graphic elements in uniform patterns, conditionals allow us to break the uniformity of a pattern where appropriate, in order to adjust or inflect parts of a composition to respond to special conditions. We have illustrated this principle mostly by considering odd and even conditions, end conditions, interior conditions, and center conditions in rows and grids of architectural elements, but you should be able to think of many more ways in which parts of compositions are varied in response to existing conditions.

Uniformity is prized by some designers, and programs to generate work in their style will contain few conditionals. Much of classical architecture and of the work of Mies van der Rohe, for example, is like this (fig. 11-43a). Other designers, however, value the way that interest and meaning can be created within a composition by varying and inflecting elements in response to subtle contextual differences. Baroque architects were masters of this approach as was Alvar Aalto (fig. 11-43b). Programs to generate compositions of this sort will be rich in conditionals. Finally, some designers are interested in the startling effects of purely arbitrary variation. Programs to produce this kind of composition will incorporate random selection mechanisms. In any case, the code makes explicit answers to two basic aesthetic questions. Under what conditions should elements vary? And, under a given condition, what kind of variation should occur?

Module 144: 12.1 Structure

12. HIERARCHICAL STRUCTURE

12.1 SUBSYSTEMS

Consider a composition consisting just of a square, generated by,

```
SQUARE (X_SQUARE, Y_SQUARE, SIDE)
```

and a circle, generated by,

```
CIRCLE (X_CIRCLE, Y_CIRCLE, DIAMETER)
```

Each procedure has three parameters (fig. 12-1), so there are six design variables in all. In other words, we have established a spatial relation with six degrees of freedom between the square and the circle.

We can reduce the degrees of freedom by making one of these six variables depend upon another:

```
SQUARE (X_SQUARE, Y_SQUARE, SIDE)  
X_CIRCLE = X_SQUARE  
CIRCLE (X_CIRCLE, Y_CIRCLE, DIAMETER)
```

The circle can now move only on a vertical line through the center of the square (fig. 12-2). We have established a particular type of graphic subsystem, consisting of two vocabulary elements spatially related in a specific way.

Subtypes can be established by further reducing the degrees of freedom in the relation (fig. 12-3):

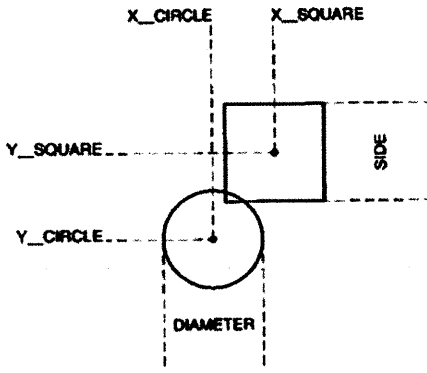
```
SQUARE (X_SQUARE, Y_SQUARE, SIDE)  
X_CIRCLE = X_SQUARE  
Y_CIRCLE = Y_SQUARE - SIDE  
DIAMETER = SIDE * 2  
CIRCLE (X_CIRCLE, Y_CIRCLE, DIAMETER)
```

A procedure to generate this subsystem can now be written:

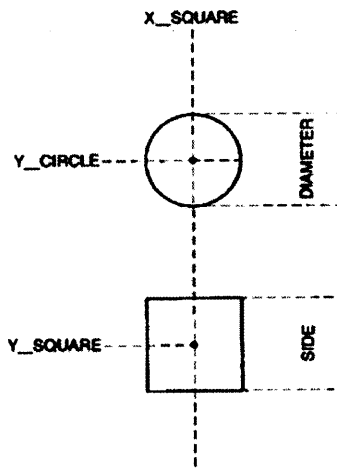
[samples code on the right side]

This gives the type of subsystem a name and specifies the relation between its constituent vocabulary elements.

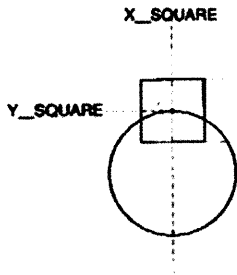
The same vocabulary elements may be related in many different ways to form different types of subsystems. Figure 12-4 suggests some of the possibilities for a circle and a square.



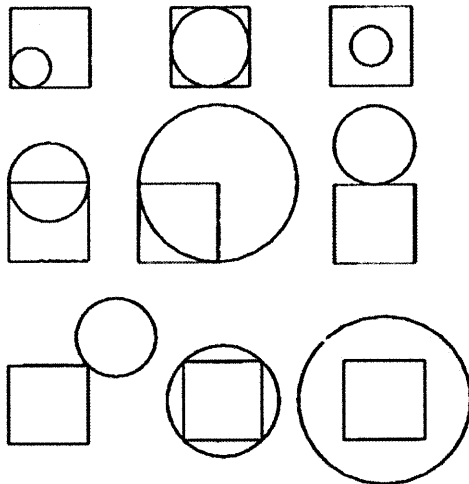
12-1. A composition of a square and circle, with six degrees of freedom.



12-2. A composition with five degrees of freedom; the circle moves on a vertical line through the center of the square.



12-3. A composition with three degrees of freedom.



12-4. Some possible relationships between a circle and square.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

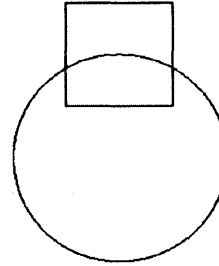
def square(xc, yc, length):
    x1 = xc - length / 2
    x2 = x1 + length
    y1 = yc - length / 2
    y2 = y1 + length
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]
    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def circle(x, y, radius):
    center = [x, y, 0]
    rs.AddCircle(center, radius)

def pantheon(x_square, y_square, side):
    square(x_square, y_square, side)
    x_circle = x_square
    y_circle = y_square - side
    circle(x_circle, y_circle, side)

pantheon(10,10,20)
```

RESULT



Module 145: 12.1 Subsystems

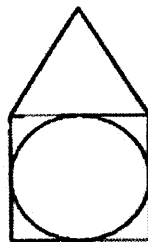
12. HIERARCHICAL STRUCTURE

12.1.1 SUBSYSTEMS OF MANY ELEMENTS

A subsystem that is built up from available vocabulary elements placed in some specific spatial relationship need not consist of just two such elements. There may be any number. The following procedure, for example, interrelates a square, a circle, and a triangle:

[samples code on the right side]

Some output is shown in figure 12-5.



12-5. A subsystem consisting of a circle, a square, and a triangle.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def square(xc, yc, length):
    x1 = xc - length / 2
    x2 = x1 + length
    y1 = yc - length / 2
    y2 = y1 + length

    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def circle(x, y, radius):
    center = [x, y, 0]
    rs.AddCircle(center, radius)

def triangle(x, y, length):
    pt1 = [x, y, 0]
    y2 = y + length * 1.7
    x2 = x + length
    pt2 = [x2, y2, 0]
    x3 = x + length * 2
    pt3 = [x3, y, 0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt1)

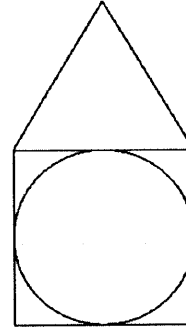
def clockFace(x, y, diameter):
    square(x, y, diameter)

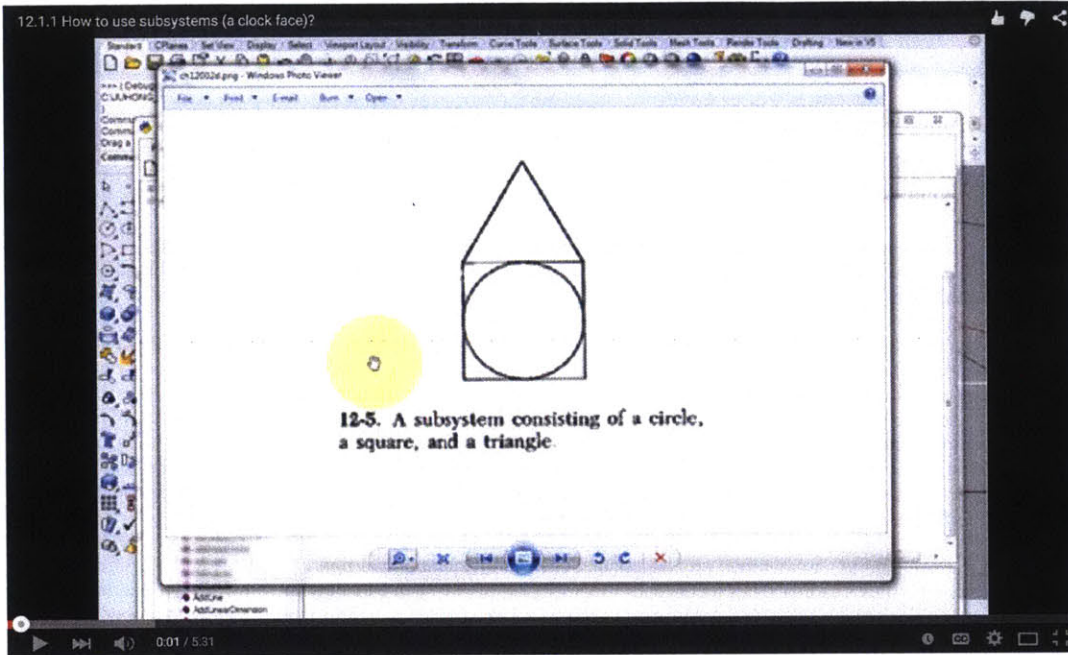
    radius = diameter / 2
    circle(x, y, radius)

    yt = y + radius
    xt = x - radius
    triangle(xt, yt, radius)

clockFace(0,0,100)
```

RESULT





Module 146: 12.1 Relations

12. HIERARCHICAL STRUCTURE

12.1.2 KINDS OF SPATIAL RELATIONS

Vocabulary elements can be related to form subsystems in an infinite number of ways, but certain kinds of subsystems have always been of particular interest to architects and graphic designers. These can be characterized by their properties of symmetry.

Many vocabulary elements (circles, regular polygons, and the like) have symmetry about a point, and these are often related by making the symmetry points coincident. Figure 12-6, for example, shows some cases of a square and circle related in this way. In figure 12-6a they have the same diameter. This relationship characterizes Roman, Byzantine, and Ottoman building plans, where hemispherical domes are often placed over square spaces. In figure 12-6b the circle is much smaller than the square. We find this in the plan of Palladio's Villa Rotonda, for example, where a cylindrical central space is placed within an outer cubic mass. In figure 12-6c and d the diameters are similar but not the same, so that an interesting residual space is created between the inner and outer enclosures. Louis Kahn often used this relationship in his compositions.

It is usually convenient to parameterize procedures that draw point symmetrical figures by the X and Y coordinates of the center point, plus a radius. Thus a concentric relationship of two such elements is established by making their center point coordinates the same, and perhaps by specifying a ratio of the two radii as well. Our next procedure, for example, draws a concentrically related pair of regular polygons. Parameters are X and Y, the numbers of sides of each polygon, and the radius of each polygon:

[samples code on the right side]

Where vocabulary elements have symmetry about one or more axes, they are frequently related by making the axes coincident. Figure 12-8 shows a plan composed of squares, circles, and ovals related in this way. Architects have often extended this principle by stringing many bilaterally symmetrical spaces along a circulation axis. Classical and neoclassical architects tended to keep the constituent spaces disjoint (fig 12-9a). Baroque architects, on the other hand, often let them intersect in complex ways (fig. 12-9b).

Procedures that draw bilaterally symmetrical figures often take the X coordinate of the axis of symmetry as a parameter. Thus bilaterally symmetrical figures can be coaxially related by invoking their procedures with the same x value. The following procedure, for example, draws an isosceles triangle:

[samples code on the right side]

The next procedure invokes Isosceles three times to draw three coaxially related isosceles triangles (fig. 12-10a):

[samples code on the right side]

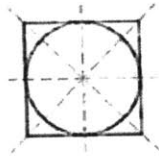
This relationship of triangles was explored by Palladio in his church elevations (figs. 12-10b, c).

Another common way to relate elements is to align their edges. Consider the relationship of a large and a small square window in elevation. As shown in figure 12-11, the edges of the large window establish four axes, which divide the surrounding wall plane into eight quadrants. The small window might intersect any one of these axes on either side to yield eight possible relations. Or the small window might be contained wholly within any one of the quadrants to yield eight more possibilities. Within any one of the quadrants, the top, bottom, or side of the small window might be on an axis another sixteen possibilities emerge from this. If we allow the windows to touch at their edges, we obtain twelve more possibilities. And, if we allow them to touch just at their corners, we obtain another four. Finally, we might make the diagonal axes coincident.

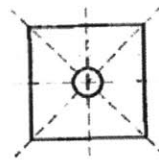
Instead of aligning edges or axes, we can choose to intersect them at an angle. Figure 12-12, for example, shows a common path for the plan of a mosque. The large rectangle, forming the body of the mosque, has its principal axis pointing toward Mecca. The small rectangle is an entrance space, aligned with the street. The axes of the two rectangles intersect at the center of a circular transition space. The dimensions of the three elements and the parameter Theta can be adjusted in order to fit the plan to any given context.

Finally, we can place one element in a -floating- relationship to another by carefully avoiding coincidence of important points or alignment of axes and edges. A square, for instance, has four edges and four axes of symmetry, dividing the plane into sixteen regions outside the square and eight within it (fig. 12-13a). We can float a circle in relation to this square by placing it so that its center point falls within any one of these regions and not on one of their boundaries, and also so that its circumference is not tangent to any one of these boundaries (fig. 12-13b). We might go further and require that the circumference does not pass through any of the intersections of axes and edge lines.

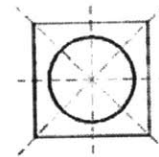
Sometimes designers make the relationships of vocabulary elements in a composition explicit by showing axes, grids, circles, and other construction lines on their drawings. Le Corbusier called these -traces regulators, - and they show up on many of his early projects (fig. 12-14).



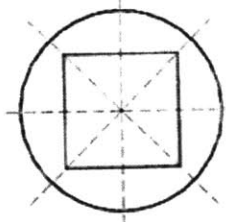
a. A dome over a square space.



b. A circular room at the center of a square building.

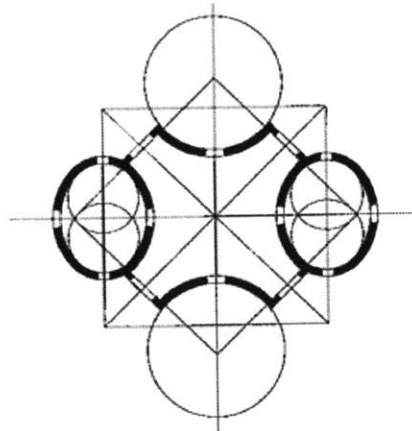


c. Residual space outside the circle.

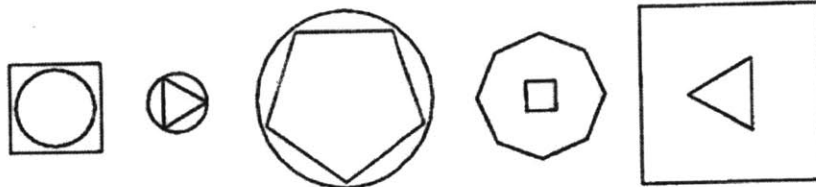


d. Residual space outside the square.

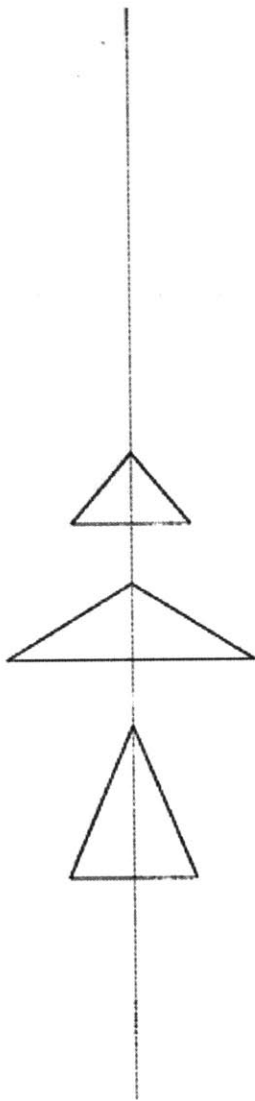
12-6. Coincident symmetry points.



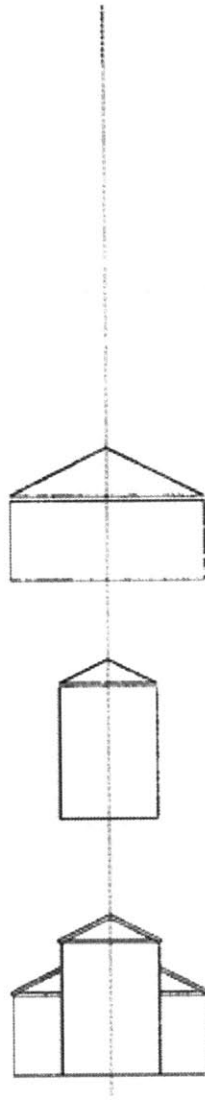
12-8. Axial relations of circles and squares in a sketch for a Lustgartengebäude in Vienna by J.B. Fischer von Erlach.



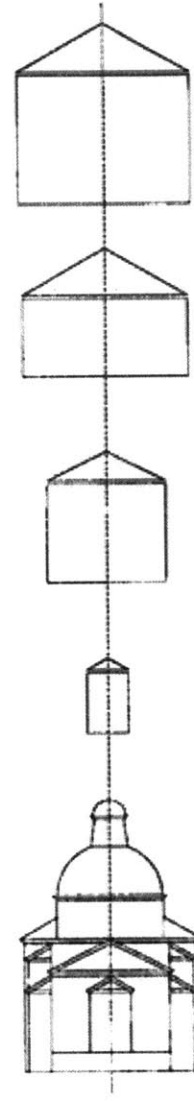
12-7. Pairs of concentric regular polygons.



a. The principle of coaxial composition.

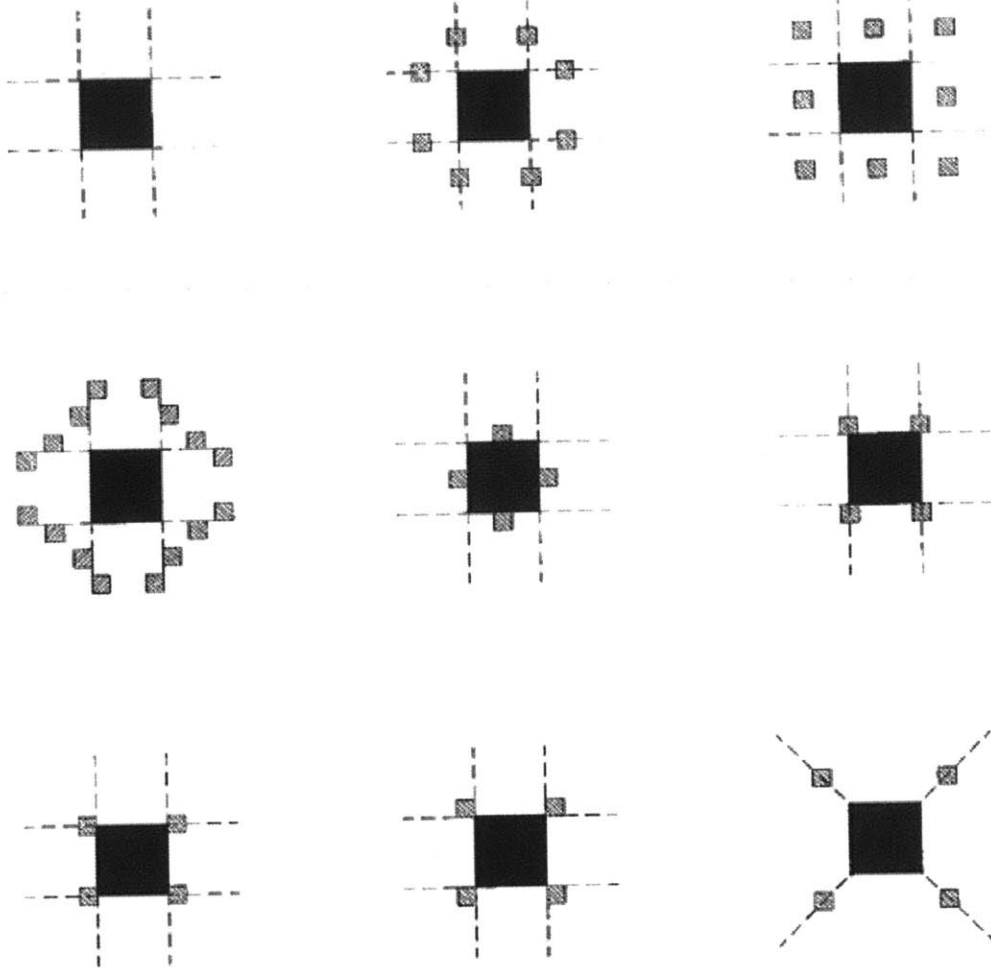


b. S. Giorgio Maggiore, Venice.

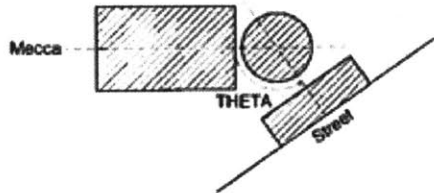


c. Il Redentore, Venice.

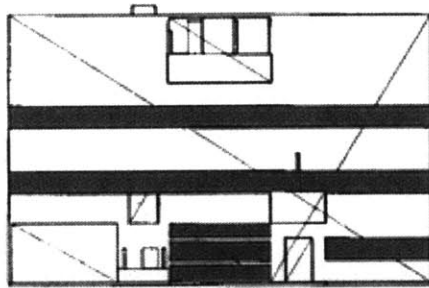
12-10. Coaxial isosceles triangles in two church elevations by Palladio.



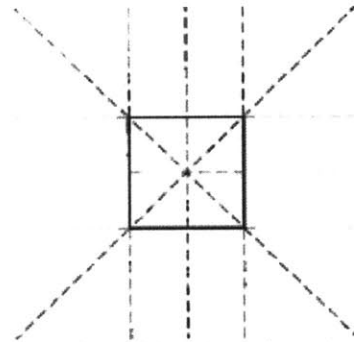
12-11. Relating a small to a large square window.



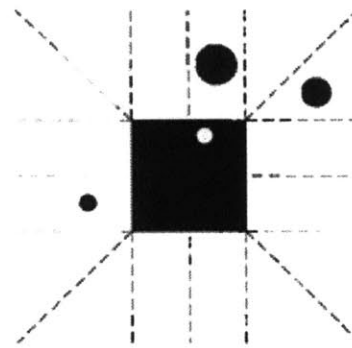
12-12. The arrangement of a mosque in relation to Mecca and to the street.



12-14. Regulating lines of Le Corbusier's Villa Garches (north elevation).



a. Axes of symmetry and edge lines.



b. Circles fall between the lines.

12-13. Circles in floating relationship to a square.

Synthetic Tutor

CODE

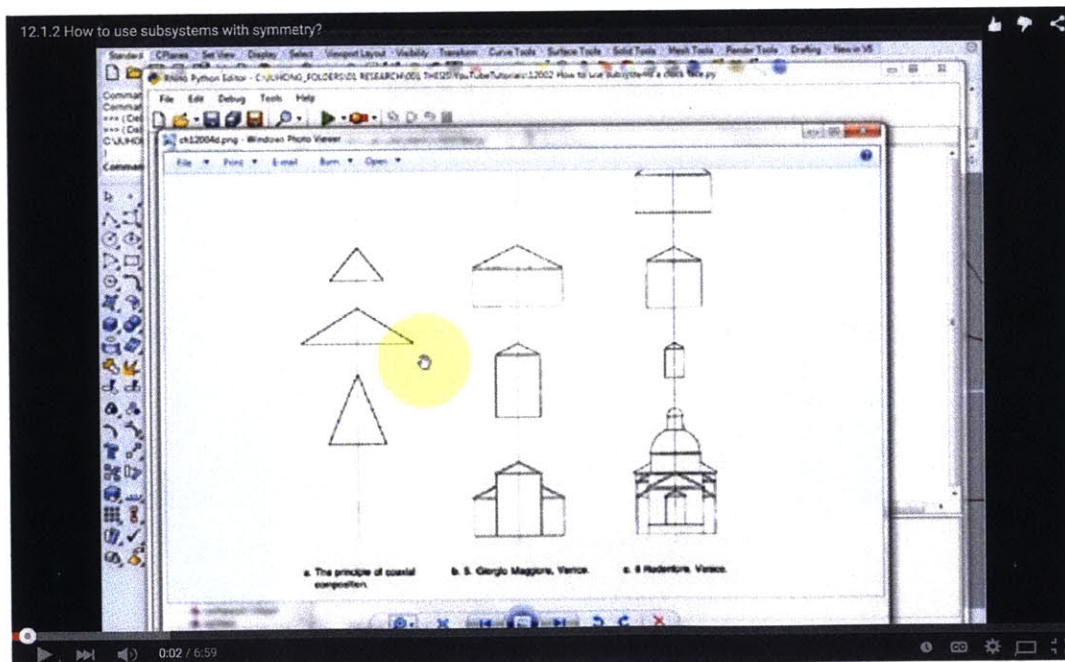
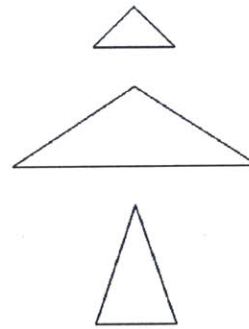
```
import rhinoscriptsyntax as rs

def isosceles(x,y,base,height):
    x1=x-(base/2)
    x2=x1+base
    y1=y+height
    p0=[x1,y,0]
    p1=[x2,y,0]
    p2=[x,y1,0]
    p3=[x1,y,0]
    rs.AddLine(p0, p1)
    rs.AddLine(p1, p2)
    rs.AddLine(p2, p3)

def three_triangles(x,y1,y2,y3,base_1,base_2,base_3,
                    height_1,height_2,height_3):
    isosceles(x,y1,base_1,height_1)
    isosceles(x,y2,base_2,height_2)
    isosceles(x,y3,base_3,height_3)

three_triangles(0, 0, 200, 350, 100,300,100, 150,100,50)
```

RESULT



Module 147: 12.2 Interface

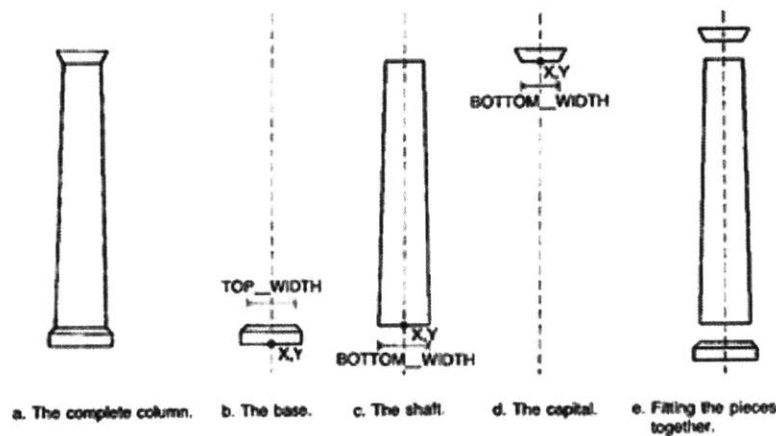
12. HIERARCHICAL STRUCTURE

12.2 INTERFACES

Now consider the column shown in figure 12-15a. It consists of a base, a shaft, and a capital (fig. 12-15b, c, d). Procedures to draw these elements might be written as follows:

[samples code on the right side]

In the complete column, the base, the shaft, and the capital are in a particular spatial relation (fig. 12-15e). They are stacked on top of each other, and their axes of symmetry are coincident. What information is needed to draw the three elements in the correct relation? To draw the base, it is necessary first to establish position and size. Then, to draw the shaft, it is also necessary to know the height and width of the base. Finally, to draw the capital, it is necessary to know the height of the shaft and (since it tapers) its width at the top. This presents a problem, since the height of the base is calculated within the procedure Base and is therefore inaccessible outside that procedure. Similarly, the height and top width of the shaft are calculated inside the procedure Shaft, and are inaccessible outside that procedure.



12-15. A column consisting of base, shaft, and capital.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def base(x, y, top_width):
    bottom_width = top_width * 1.5
    height = top_width * 0.3

    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width
    y1 = y + height * 0.8
    y_top = y + height

    p0 = [x1, y, 0]
    p1 = [x2, y, 0]
    p2 = [x2, y1, 0]
    p3 = [x4, y_top, 0]
    p4 = [x3, y_top, 0]
    p5 = [x1, y1, 0]
    p6 = [x1, y, 0]
    p7 = [x1, y1, 0]
    p8 = [x2, y1, 0]
    pts = [p0, p1, p2, p3, p4, p5, p6, p7, p8]
    rs.AddPolyline(pts)

def shaft(x, y, bottom_width):
    top_width = bottom_width * 0.6
    height = bottom_width * 6
    y_top = y + height
    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width

    p0 = [x1, y, 0]
    p1 = [x3, y_top, 0]
    p2 = [x4, y_top, 0]
    p3 = [x2, y, 0]
    pts = [p0, p1, p2, p3]
    rs.AddPolyline(pts)

def capital(x, y, bottom_width):
    top_width = bottom_width * 2.0
    height = bottom_width * 0.7

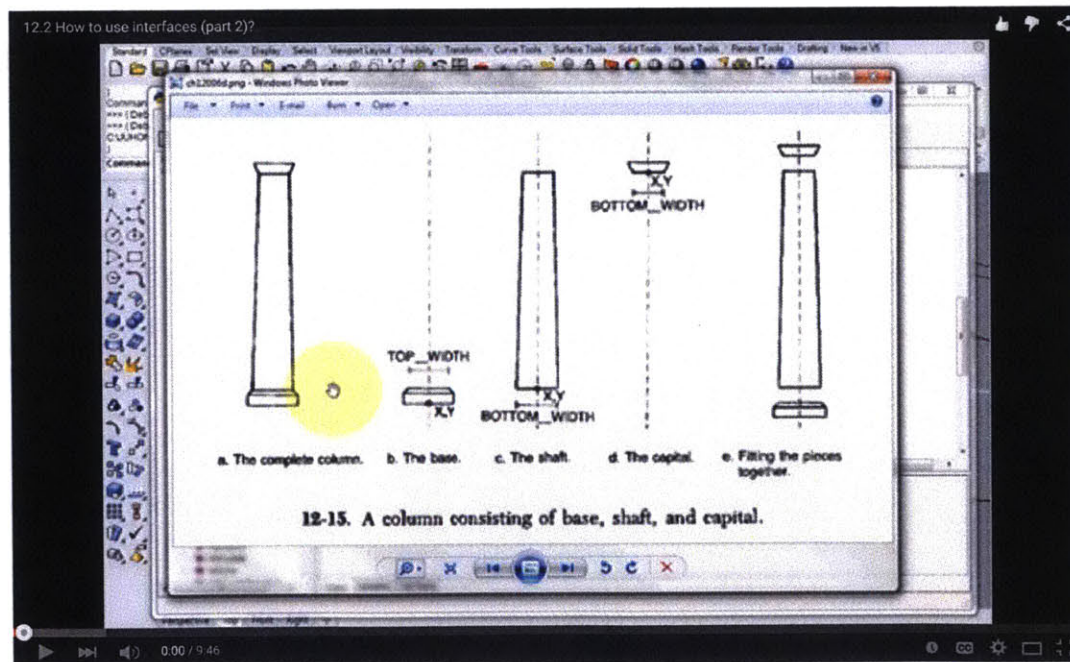
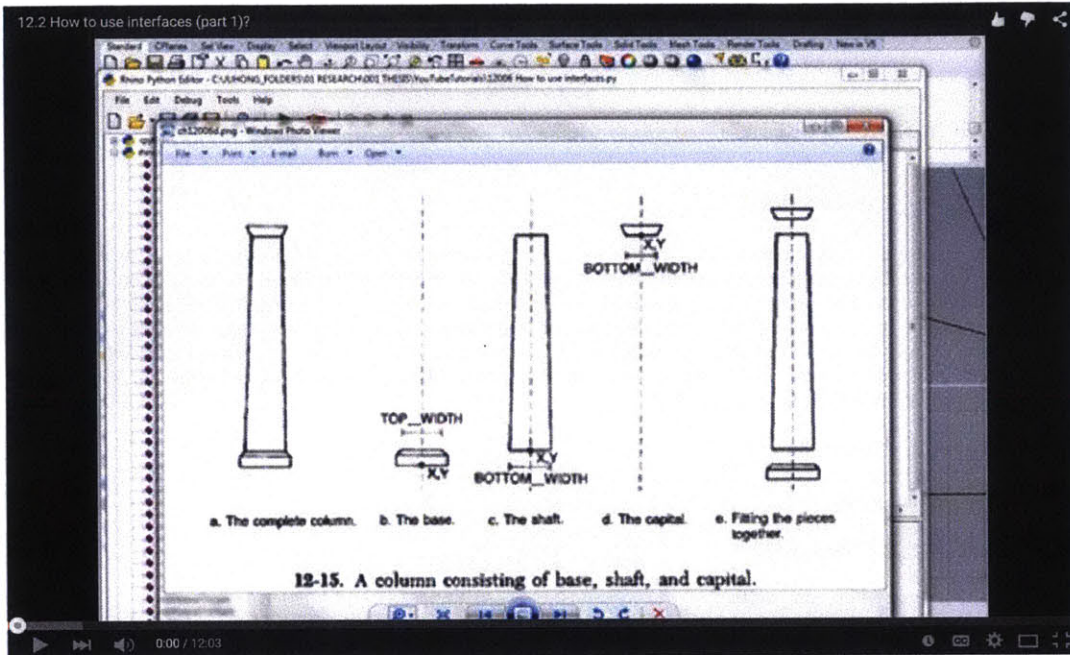
    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width
    y_top = y + height

    p0 = [x1, y, 0]
    p1 = [x2, y, 0]
    p2 = [x4, y_top, 0]
    p3 = [x3, y_top, 0]
    p4 = [x1, y, 0]
    pts = [p0, p1, p2, p3, p4]
    rs.AddPolyline(pts)

base(0, 0, 100)
shaft(0, 30, 100)
capital(0, 630, 60)
```

RESULT





Module 148: 12.3 Variable

12. HIERARCHICAL STRUCTURE

12.2.1 VARIABLE PARAMETERS

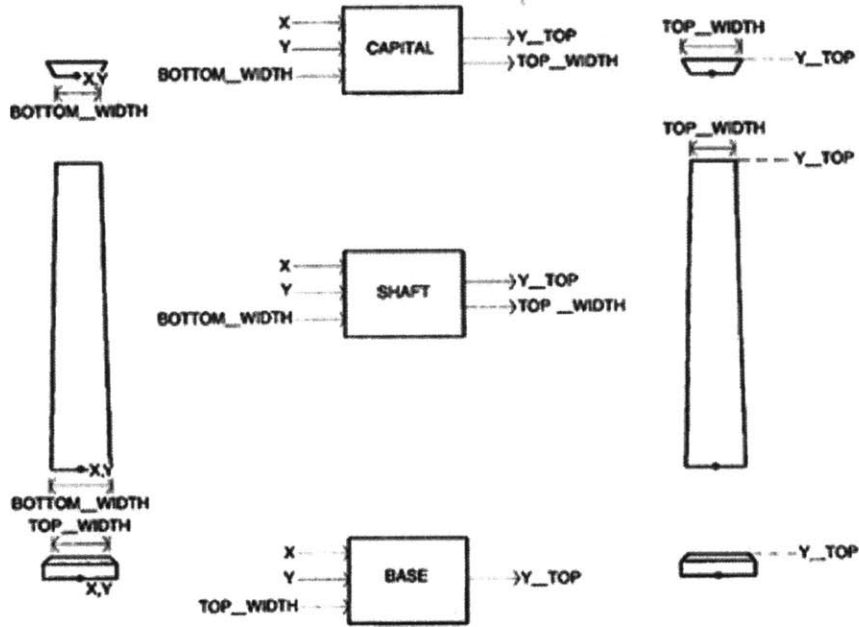
The need here is for a means to pass values out of a procedure the converse of using the parameter list to pass values in. Python provides the construct of a variable parameter for passing the result of a computation out of a procedure. The rules for use of variable parameters are simple. First, the actual parameter must be a variable it cannot be a number, a constant, or any other kind of expression. Second, the corresponding formal parameter must be preceded by the word `var`. Any operation within the procedure on this formal parameter will then be performed directly upon the actual parameter. Here, now, are the headings of procedures `Base`, `Shaft`, and `Capital`, modified by the incorporation of variable parameters as required:

```
def base (x, y, top_width):
    ...
    return y_top

def shaft (x, y, bottom_width):
    ...
    return y_top, top_width

def capital (x, y, bottom_width):
    ...
    return y_top, top_width
```

The inputs and outputs of these three procedures now are as shown in figure 12-16



12-16. Inputs and outputs of procedures Base, Shaft, and Capital.

Module 149: 12.3 Syntax

12. HIERARCHICAL STRUCTURE

12.2.2 THE SYNTAX OF A COMPOSITION

In general, the formal parameters of a procedure to draw a graphic element or a subsystem define the interface between that element or subsystem and the rest of the composition. The spatial relations of instances of this element or subsystem with other parts of a composition can be defined only in terms of these parameters. Values passed in to the procedure, to specify an instance, establish the spatial relations of that instance to existing parts of the composition, and values passed out provide information that will be needed, later, to add instances of other elements or subsystems in the required spatial relations. Arithmetic expressions and assignments relating the values passed into procedures establish spatial relations (for example, of relative size, of spacing, of alignment, or of coincidence) between the resulting instances. The syntax of a composition is defined in this way. Here, now, is a complete program to read in values for the design variables and draw a column (fig. 12-17):

[samples code on the right side]

Notice how the Python principle of declaring something before it can be used is followed here. Specifically, procedures must be declared before they are invoked. Where procedures are invoked from within other procedures, this means that the invoked procedure must appear, in the text of the program, before the procedure that does the invoking.

Notice also that procedure Column has three variable parameters: the position variables (x, y) and the radius of a column. These are necessary if the column is to be placed in a more complex composition.



12-17. A column drawn by the procedure Column.

```

CODE
import rhinoscriptsyntax as rs

def base(x, y, top_width):
    bottom_width = top_width * 1.5
    height = top_width * 0.3

    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width
    y1 = y + height * 0.8
    y_top = y + height

    p0 = [x1, y, 0]
    p1 = [x2, y, 0]
    p2 = [x2, y1, 0]
    p3 = [x4, y_top, 0]
    p4 = [x3, y_top, 0]
    p5 = [x1, y1, 0]
    p6 = [x1, y, 0]
    p7 = [x1, y1, 0]
    p8 = [x2, y1, 0]
    pts = [p0, p1, p2, p3, p4, p5, p6, p7, p8]
    rs.AddPolyline(pts)

    return y_top

def shaft(x, y, bottom_width):
    top_width = bottom_width * 0.6
    height = bottom_width * 6
    y_top = y + height
    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width

    p0 = [x1, y, 0]
    p1 = [x3, y_top, 0]
    p2 = [x4, y_top, 0]
    p3 = [x2, y, 0]
    pts = [p0, p1, p2, p3]
    rs.AddPolyline(pts)

    return y_top, top_width

def capital(x, y, bottom_width):
    top_width = bottom_width * 2.0
    height = bottom_width * 0.7

    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width
    y_top = y + height

    p0 = [x1, y, 0]
    p1 = [x2, y, 0]
    p2 = [x4, y_top, 0]
    p3 = [x3, y_top, 0]
    p4 = [x1, y, 0]
    pts = [p0, p1, p2, p3, p4]
    rs.AddPolyline(pts)

def column(x, y, diameter):
    top_shaft = 100
    y_base = base(x, y, diameter)
    y_shaft, top_shaft = shaft(x, y_base, diameter)
    capital(x, y_shaft, top_shaft)

def column(x, y, diameter):
    top_shaft = 100
    y_base = base(x, y, diameter)
    y_shaft, top_shaft = shaft(x, y_base, diameter)
    capital(x, y_shaft, top_shaft)

def main():
    x = rs.GetReal('enter x coordinates of base', 0)
    y = rs.GetReal('enter y coordinates of base', 0)
    diameter = rs.GetReal('enter column diameter', 100)
    column(x, y, diameter)

main()

```

RESULT



Module 150: 12.2 Procedure

12. HIERARCHICAL STRUCTURE

12.2.3 DECLARATION OF PROCEDURES WITHIN PROCEDURES

Here is another way to structure our Column procedure:

```
def base():  
    ...  
  
def shaft():  
    ...  
  
def capital():  
    ...  
  
def column():  
    base()  
    shaft()  
    capital()
```

The difference here is that the procedures Base, Shaft, and Capital are declared within the procedure Column. The effect, as when a variable is declared within a procedure, is to make them local to that procedure.

When should you declare graphic procedures locally, and when should you declare them globally? It depends on the specialization of the graphic element. A shaft, for example, might never appear in a composition except as part of a column. It is more expressive of the nature of this element, then, to declare procedure Shaft within procedure Column. But an arc might appear in a composition in many different contexts as part of several different types of arches in an elevation, for example. So it makes sense to declare procedure Arc globally, so it is accessible from the different procedures that draw the various types of arches.

Module 151: 12.3 hierarchy

12. HIERARCHICAL STRUCTURE

12.3 HIERARCHIES OF SUBSYSTEMS

Figure 12-18a illustrates a composition of two columns supporting a lintel. Each column can be thought of as a subsystem of the composition, and these subsystems have a particular spatial relationship to each other. They are of the same dimensions, and their bases are aligned along the same horizontal line. We might reasonably take the distance between the columns to be a parameter of the relationship. The next procedure, then, invokes procedure `Column` twice to generate a pair of columns in the appropriate relationship (fig. 12-18b):

[samples code on the right side]

We can now go a step further and think of the whole composition as two subsystems, a pair of columns and a lintel related in a particular way. The lintel sits on top of the columns. Let us take the span of the lintel as a parameter and assume that depth is always one-eighth of the span (fig. 12. 18c). A procedure to draw lintels of this type may be written:

[samples code on the right side]

Assuming the overhang of the lintel at either end is half the width of a column top, the complete composition can be generated by a procedure that invokes `Column_pair` and `Lintel` as follows:

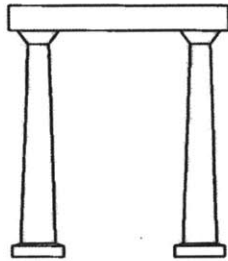
[samples code on the right side]

Notice that the heights of the columns and the widths of the tops, which are required to position the lintel correctly, are communicated from the `Column` procedure through the `Column_pair` procedure to `Frame`, which can then pass the values of these variables on to the `Lintel` procedure.

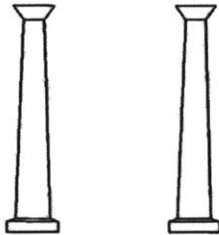
This procedure expresses rules for composing frames, using a particular vocabulary of architectural elements and a particular syntax. A designer using `Frame` has only to choose values for two shape variables, `Diameter` and `Spacing`, and the rules take care of everything else. The procedure `Base` produces a correctly proportioned base. `Shaft` fits a shaft correctly on top and chooses a proper height. `Capital` adds a correctly proportioned capital. `Column_pair` produces identical left and right columns, on a common baseline, the specified distance apart. `Lintel` generates a lintel of the required length and proportion. Finally, `Frame` places the lintel symmetrically on top of the columns.

A frame of this kind, governed by specific proportioning rules, is traditionally known as an architectural order. Figure 12-19 illustrates the proportions (as specified by Vignola) of the Tuscan, Doric, Ionic, and Corinthian orders. The procedure `Frame` can be modified to produce any one of these simply by altering the arithmetic expressions that define proportions. We could also substitute procedures to draw more specialized types of capitals, shafts, and so on.

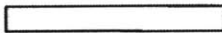
The hierarchy of elements and subsystems that we have now developed is illustrated by the tree diagram in figure 12-20. Each vertex corresponds to a procedure that defines a spatial relationship between lower-level subsystems or elements.



a. The complete system Frame.

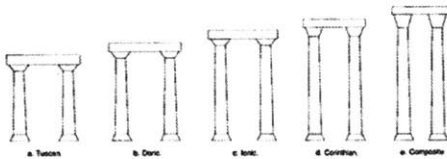


b. The Column_pair subsystem.

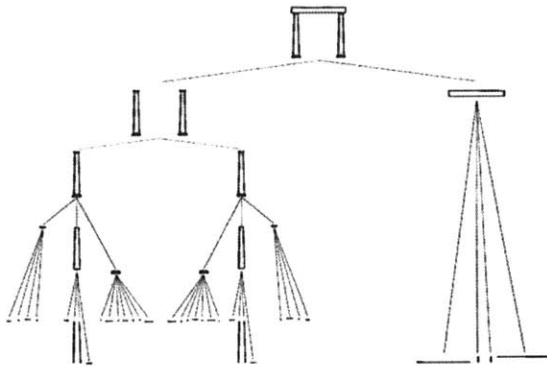


c. The Lintel subsystem.

12-18. A system of two columns supporting a lintel.



12-19. Examples of frames with a variety of proportions as specified by Vitruvius.



12-20. A hierarchy of elements and subsystems.

CODE

```

import rhinoscriptsyntax as rs

def base(x, y, top_width):
    bottom_width = top_width * 1.5
    height = top_width * 0.3

    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width
    y1 = y + height * 0.8
    y_top = y + height

    p0 = [x1, y, 0]
    p1 = [x2, y, 0]
    p2 = [x2, y1, 0]
    p3 = [x4, y_top, 0]
    p4 = [x3, y_top, 0]
    p5 = [x1, y1, 0]
    p6 = [x1, y, 0]
    p7 = [x1, y1, 0]
    p8 = [x2, y1, 0]
    pts = [p0, p1, p2, p3, p4, p5, p6, p7, p8]
    rs.AddPolyline(pts)

    return y_top

def shaft(x, y, bottom_width):
    top_width = bottom_width * 0.6
    height = bottom_width * 6
    y_top = y + height
    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width

    p0 = [x1, y, 0]
    p1 = [x3, y_top, 0]
    p2 = [x4, y_top, 0]
    p3 = [x2, y, 0]
    pts = [p0, p1, p2, p3]
    rs.AddPolyline(pts)

    return y_top, top_width

def capital(x, y, bottom_width):
    top_width = bottom_width * 2.0
    height = bottom_width * 0.7

    x1 = x - (bottom_width / 2)
    x2 = x1 + bottom_width
    x3 = x - (top_width / 2)
    x4 = x3 + top_width
    y_top = y + height

    p0 = [x1, y, 0]
    p1 = [x2, y, 0]
    p2 = [x4, y_top, 0]
    p3 = [x3, y_top, 0]
    p4 = [x1, y, 0]
    pts = [p0, p1, p2, p3, p4]
    rs.AddPolyline(pts)

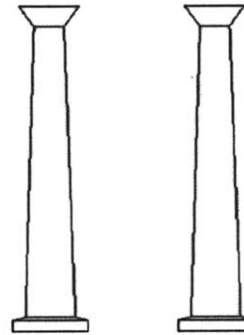
def column(x, y, diameter):
    y_base = base(x, y, diameter)
    y_shaft, top_shaft = shaft(x, y_base, diameter)
    capital(x, y_shaft, top_shaft)

def column_pair(x, y, diameter, spacing):
    bottom_x = x - (spacing / 2)
    bottom_y = y
    column(bottom_x, bottom_y, diameter)
    bottom_x = bottom_x + spacing
    column(bottom_x, bottom_y, diameter)

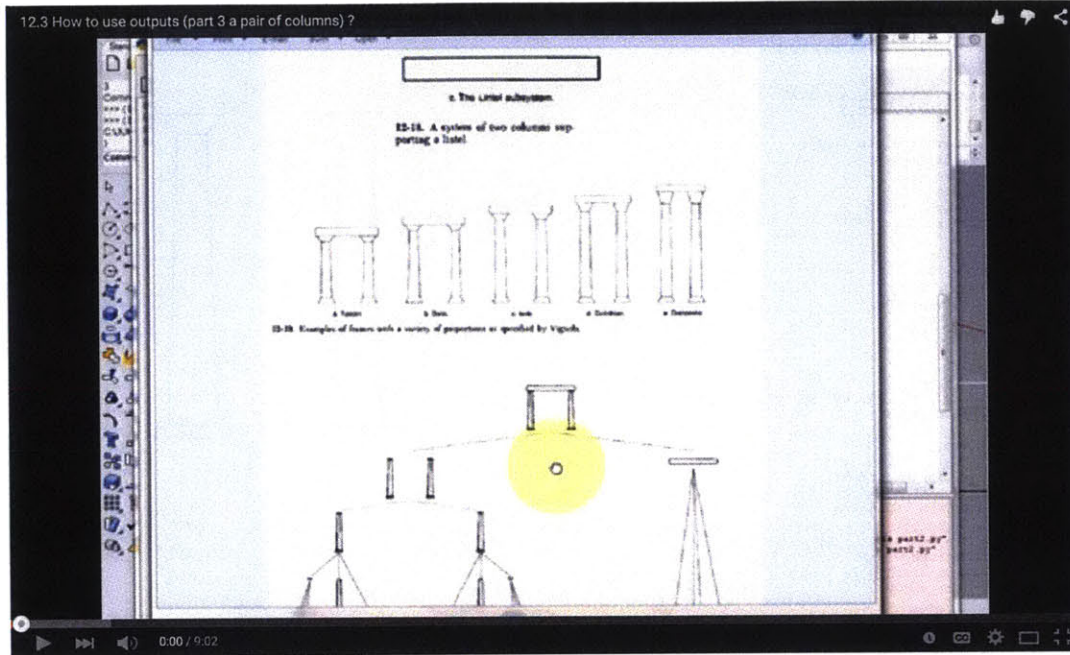
column_pair(0, 0, 30, 100)

```

RESULT



Synthetic Tutor



Module 152: 12.3 Structure

12. HIERARCHICAL STRUCTURE

12.3.1 STRUCTURE AND MEANING

The way that we parse a graphic composition into elements and subsystems determines the meaning of the composition just as the way that we parse a sentence determines its meaning. Individual vectors carry little specific meaning. When vectors are spatially related in particular ways, though, we can interpret them as components of bases, shafts, capitals, and lintels, and we express those interpretations in the names given to procedures: Base, Shaft, Capital, Lintel. When base, shaft, and capital are stacked coaxially, we can interpret the result as a column. When columns are paired, we can interpret the result as a subsystem capable of supporting a lintel. When the lintel is added in the correct relation, we recognize that we have a frame. In other words, the hierarchy of elements and subsystems determines the way that the meaning of the complete composition is built up from the meanings of its parts, just as the hierarchy of phrases determines the way that the meaning of a sentence is built up from the meanings of words. Procedures define the vocabulary of a graphic language, the hierarchy and interfacing of procedures defines its syntax, and the names given to procedures begin to define its semantics.

Module 153: 12.3 Block

12. HIERARCHICAL STRUCTURE

12.3.2 BLOCK STRUCTURE

The principles of program organization that we have introduced in this chapter may now be summarized and generalized by describing the general rules of block structure in Python. A Python program is a collection of segments called blocks. The program as a whole is a block. Each procedure is a block, and each function is a block. A block has a heading, so it can be identified by name. Within each block (following the heading) constants and types may be defined then variables, procedures, and functions may be declared, and finally, program actions may be specified.

Since functions and procedures may be declared within any block, it follows that blocks may be nested within blocks. That is, procedures may be declared within procedures, functions within functions, procedures within functions, and functions within procedures.

A definition or declaration is local to the block in which it is declared. In other words, the names of constants, types, variables, procedures, and functions have significance only within the blocks in which they are declared. The scope of significance of a definition or declaration is from its first appearance to the end of the block.

Communication between blocks is strictly disciplined. The parameter list of a block defines the connections between a block (procedure or function) and its environment. Different kinds of parameters establish different kinds of connections. Value parameters allow values to be passed in, and variable parameters allow results to be passed out.

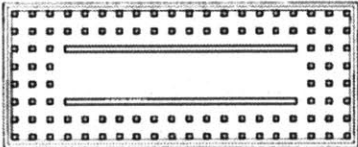
Where a variable is declared in a block before the function and procedure declarations in that block, it is global relative to those functions and procedures. This means that its value is accessible to these functions and procedures, and that execution of one of these functions or procedures may modify its value. Such a modification is called a side effect of execution. Side effects generally make a program difficult to understand, can propagate errors, and tend to make debugging difficult. It is therefore best to avoid the global declaration of variables except where absolutely necessary. Communication via parameter lists, on the other hand, keeps interfaces well defined.

To illustrate these principles, let us consider a program to draw classical temple plans of the type shown in figure 12-21a. The element and subsystem hierarchy that we shall adopt is shown in figure 12-21b. The basic vocabulary elements are Circle and Rectangle. A Column is composed of a Circle and a Rectangle. A Column_grid is composed of Columns. A Plinth is composed of Rectangles. The Celia consists of two Rectangles. The Temple consists of a Plinth, a Column_grid, and a Celia, all concentrically related.

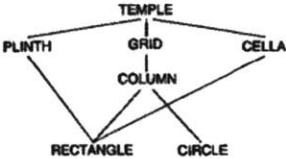
Here is the complete code of the program:

[samples code on the right side]

Some examples of output from this program for different values of the design variables are illustrated in figure 12-22

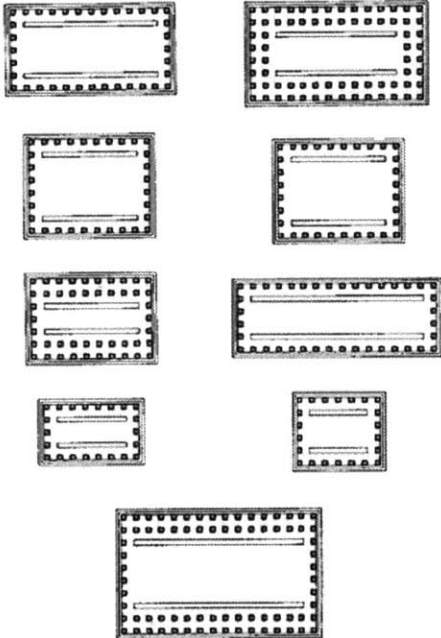


a. Type diagram.



b. Hierarchy of elements and subsystems.

12-21. A classical temple plan.



12-22. Instances of classical temple plans.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def rectangle(x, y, length, width):
    x1 = x - (length / 2)
    y1 = y - (width / 2)
    x2 = x1 + length
    y2 = y1 + width

    p0 = [x1, y1, 0]
    p1 = [x2, y1, 0]
    p2 = [x2, y2, 0]
    p3 = [x1, y2, 0]
    p4 = [x1, y1, 0]
    pts = [p0, p1, p2, p3, p4]
    rs.AddPolyline(pts)

def circle(x, y, radius):
    center = [x, y, 0]
    rs.AddCircle(center, radius)

def column(x, y, diameter):
    rectangle(x, y, diameter, diameter)
    circle(x, y, diameter/2)

def plinth(x, y, length, width, steps, step_size):
    repeat = range(steps)
    for count in repeat:
        rectangle(x, y, length, width)
        length = length + step_size
        width = width + step_size

def cella(x, y, length, diameter, width):
    rectangle(x, y, length, diameter)
    y = y + width
    rectangle(x, y, length, diameter)

def outside(count_x, count_y,
            x_start, x_finish,
            y_start, y_finish):

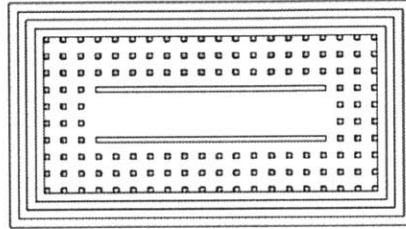
    cond1 = count_x < x_start
    cond2 = count_x > x_finish
    cond3 = count_y < y_start
    cond4 = count_y > y_finish
    outside = cond1 or cond2 or cond3 or cond4
    return outside

def grid(x_initial, y_initial, spacing, diameter,
        nx, ny, x_start, x_finish, y_start, y_finish,
        steps, step_size):

    repeat_ny = range(ny)
    repeat_nx = range(nx)
    y = y_initial
    for count_y in repeat_ny:
        x = x_initial
        for count_x in repeat_nx:
            isOutside = outside(count_x, count_y,
                               x_start, x_finish,
                               y_start, y_finish)

            if isOutside:
                column(x,y,diameter)
            x = x + spacing
        y = y + spacing
```

RESULT



```
def draw_temple():
    x = rs.GetReal('enter x coordinate', 0)
    y = rs.GetReal('enter y coordinate', 0)
    diameter = rs.GetReal('enter column diameter', 300)
    spacing = rs.GetReal('enter column spacing', 1000)
    nx = rs.GetInteger('enter number of columns in x.', 20)
    ny = rs.GetInteger('enter number of columns in y.', 10)
    x_start = rs.GetInteger('enter x start number', 3)
    x_finish = rs.GetInteger('enter x finish number', 16)
    y_start = rs.GetInteger('enter y start number', 3)
    y_finish = rs.GetInteger('enter y finish number', 6)
    steps = rs.GetInteger('enter number of steps', 5)
    step_size = rs.GetInteger('enter width of step', 1000)

    temple(x, y, spacing, diameter,
           nx, ny,
           x_start, x_finish, y_start, y_finish,
           steps, step_size)

draw_temple()
```

Module 154: 12.4 Recursion

12. HIERARCHICAL STRUCTURE

12.4 RECURSION

There is an important special case of invocation of procedures from within procedures that we have not yet considered. In Python, a procedure may invoke itself. This is called recursion.

Let us begin by considering a very simple example. The following program draws patterns of concentric squares (fig. 12-23a).

[samples code on the right side]

Notice how the procedure `Nested_square` invokes itself.

A recursive graphic procedure, such as `Nested_square`, is best understood in terms of its initial shape, its construction rule, and its limit. In our example, the initial shape is a square centered at 512,512, of side length 512, as specified by the actual parameters passed into `Nested_square` (12-23b). The construction rule is to place a square of half its predecessor's size concentrically within its predecessor (fig. 12-23c). This spatial relationship is expressed, within `Nested_square`, by the statement:

```
SIDE = SIDE / 2
```

The limit tells when to stop applying the construction rule. In this case the limit is specified in the code of `Nested_square` by the Boolean expression:

```
SIDE > 4
```

What actually happens when the recursive procedure `Nested_square` executes? Let us trace the execution step by step. When `Nested_square` is first invoked, `Side` has the initial value of 512, and the initial shape is drawn.

The construction rule is then applied, so the value of `Side` becomes 256. Since this is greater than 4 (the limit), `Nested_square` invokes itself. The second invocation of `Nested_square` draws a square with `Side` of 256, then applies the construction rule to reduce `Side` to 128, tests against the limit, and invokes `Nested_square`. This process continues, as follows:

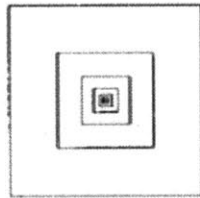
Invocation Number	From	Value of Side
1	Nest	512
2	<code>Nested_square</code> 1	256
3	<code>Nested_square</code> 2	128
4	<code>Nested_square</code> 3	64
5	<code>Nested_square</code> 4	32
6	<code>Nested_square</code> 5	16
7	<code>Nested_square</code> 6	8

Finally, `Nested_square` is invoked with a value of 8 for `Side`. Application of the construction rule then reduces the value of `Side` to 4, the limit is reached, and this seventh invocation ends. Control then passes back to the sixth invocation, which ends, and so on, all the way back to the first invocation. This ends, and control passes back to `Nest`.

Another non recursive way to generate the same composition of nested squares is to use a while loop:

```
def nested_square(x, y, side):  
    while (side > 4):  
        square(x, y, side)  
        side = side / 2
```

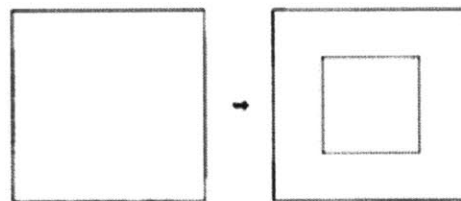
The program using recursion and the program using the while loop are both clear and concise. There are, however, many situations in which use of recursion provides the clearest, shortest, and most natural way to express the logic of a graphic composition.



a. The complete pattern.



b. The initial shape.



c. The construction rule.

12-23. A recursively constructed pattern of concentric squares.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def square(x, y, length):
    x1 = x - (length / 2)
    y1 = y - (length / 2)
    x2 = x1 + length
    y2 = y1 + length

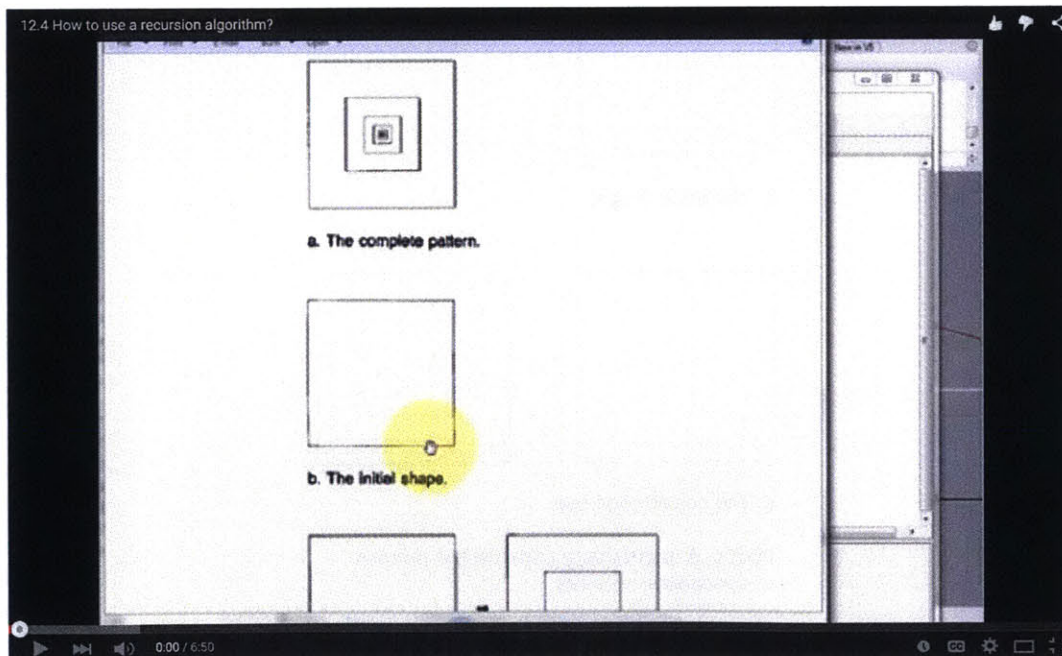
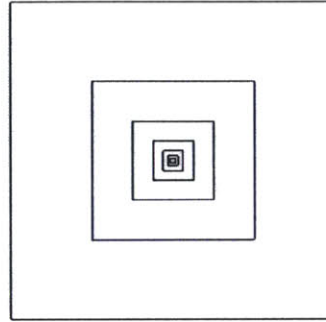
    pt1 = [x1, y1, 0]
    pt2 = [x2, y1, 0]
    pt3 = [x2, y2, 0]
    pt4 = [x1, y2, 0]

    rs.AddLine(pt1, pt2)
    rs.AddLine(pt2, pt3)
    rs.AddLine(pt3, pt4)
    rs.AddLine(pt4, pt1)

def nested_square(x, y, side):
    square(x, y, side)
    side = side / 2
    if (side > 4):
        nested_square(x, y, side)

nested_square(512, 512, 512)
```

RESULT



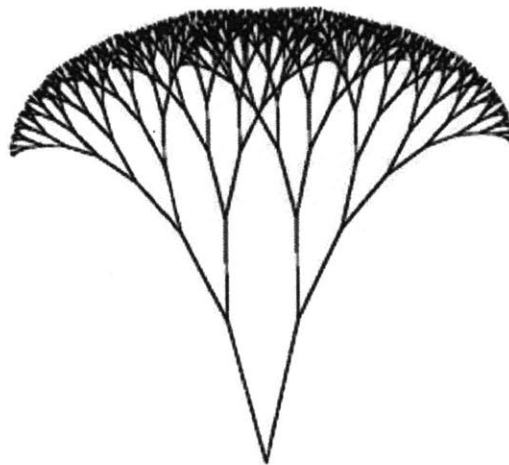
Module 155: 12.4 Tree 1

12. HIERARCHICAL STRUCTURE

12.4.1 TREES

Generally, you should consider use of recursion when you can see that a complex composition can be generated by the application of a simple construction rule to some initial shape. Look at the branching tree shown in figure 12-24, for example. The initial shape is a V-shaped pair of branches. The construction rule is simply to add a V-shaped pair of branches to each preceding branch. Thus branches divide and grow at their tips, much as real trees grow. Here is a simple recursive procedure to generate trees in this way:

[samples code on the right side]



12-24. A symmetrical recursively constructed tree.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def branch(x, y, length, start_angle, ang, min, ratio):
    radians = 0.01745

    theta1 = start_angle - ang
    x1 = x + length * math.cos(theta1 * radians)
    y1 = y + length * math.sin(theta1 * radians)

    theta2 = start_angle + ang
    x2 = x + length * math.cos(theta2 * radians)
    y2 = y + length * math.sin(theta2 * radians)

    p0 = [x1, y1, 0]
    p1 = [x, y, 0]
    p2 = [x2, y2, 0]

    rs.AddLine(p1, p0)
    rs.AddLine(p1, p2)

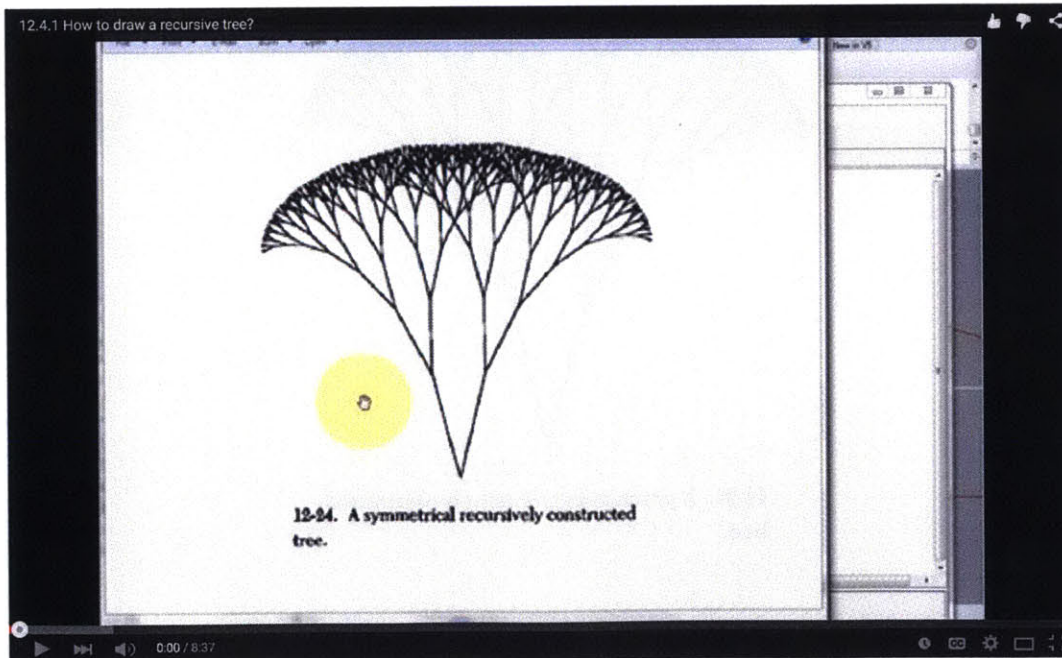
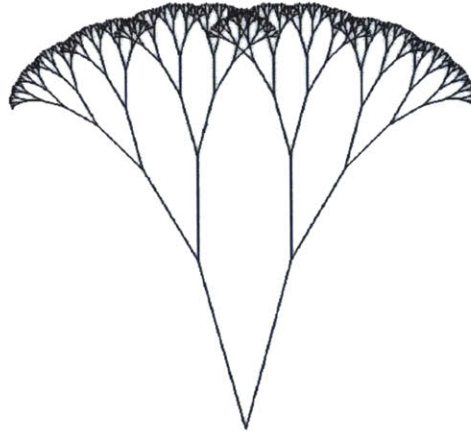
    length = length * ratio

    if (length > min):
        start_left = start_angle - ang
        branch(x1, y1, length, start_left, ang, min, ratio)

        start_right = start_angle + ang
        branch(x2, y2, length, start_right, ang, min, ratio)

branch(0,0, 100, 90, 15, 1, 0.6)
```

RESULT



Module 156: 12.4 Tree 2

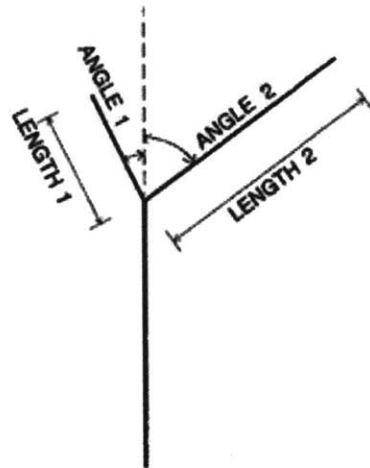
12. HIERARCHICAL STRUCTURE

12.4.1 TREES

The design variables here are Length of the initial branches, Angle of the V, Ratio of a branch's length to its predecessor's length, and Minimum size for a branch (to establish the limit). By entering different values for these design variables, you can generate a very wide variety of trees. The trees generated by this procedure are too regular to look entirely natural. This can be remedied to some extent by allowing the lengths and angles of the two sides of a V to vary independently (fig. 12-25). Here is the corresponding modified procedure:

[samples code on the right side]

Figure 12-26 shows some typical results for different values of the parameters.



12-25. The independent variation of branch lengths and angles.



12-26. Instances of recursively constructed asymmetrical trees.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def branch(x, y, leng1, leng2, start_angle, ang, min, rat):
    radians = 0.01745

    thetal = start_angle - ang
    x1 = x + leng1 * math.cos(thetal * radians)
    y1 = y + leng1 * math.sin(thetal * radians)
    theta2 = start_angle + ang
    x2 = x + leng2 * math.cos(theta2 * radians)
    y2 = y + leng2 * math.sin(theta2 * radians)

    p0 = [x1, y1, 0]
    p1 = [x, y, 0]
    p2 = [x2, y2, 0]

    rs.AddLine(p1, p0)
    rs.AddLine(p1, p2)

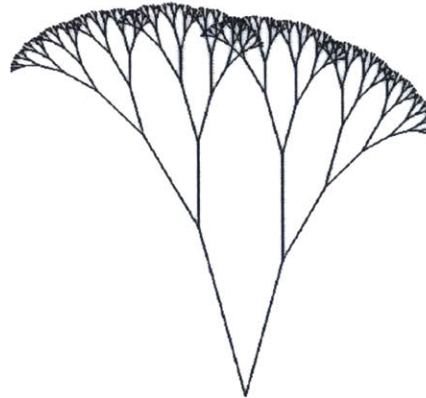
    leng1 = leng1 * rat
    leng2 = leng2 * rat

    if (leng1 > min) and (leng2 > min):
        start_l = start_angle - ang
        branch(x1, y1, leng1, leng2, start_l, ang, min, rat)

        start_r = start_angle + ang
        branch(x2, y2, leng1, leng2, start_r, ang, min, rat)

branch(0,0, 80, 100, 90, 15, 1, 0.6)
```

RESULT



12.4.1 How to draw a recursive tree (part 2)

www.youtube.com is now full screen. Exit full screen (Esc)

12-25. The independent variation of branch lengths and angles.

12-26. Instances of recursively constructed asymmetrical trees.

0:00 / 4:57

Module 157: 12.4 Tree 312.
HIERARCHICAL STRUCTURE

12.4.1 TREES

A further improvement can be made by allowing random choice (within limits) of lengths and angles for branches. We can use the random number generator that was introduced in chapter 11:

[samples code on the right side]

Each time that Branch is invoked, Length1, Angle1, Length2, and Angle2 have different, randomly chosen, values. Pascal automatically keeps track of these values. In a non-recursive version you would have to declare variables to record them. Branching terminates when a value for Length1 or for Length2 that is less than or equal to the specified minimum is generated. Figure 12-27 shows some results.



12-27. Trees generated by a recursive procedure incorporating a random number generator.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math
import random

def branch(x, y, min_length, max_length,
          start_angle, min_angle, max_angle,
          minimum, ratio):

    radians = 0.01745
    min_length = int(min_length)
    max_length = int(max_length)

    length1 = random.randrange(min_length, max_length)
    length2 = random.randrange(min_length, max_length)

    min_angle = int(min_angle)
    max_angle = int(max_angle)

    angle1 = random.randrange(min_angle, max_angle)
    angle2 = random.randrange(min_angle, max_angle)

    theta1 = start_angle - angle1
    x1 = x + length1 * math.cos(theta1 * radians)
    y1 = y + length1 * math.sin(theta1 * radians)

    theta2 = start_angle + angle2
    x2 = x + length2 * math.cos(theta2 * radians)
    y2 = y + length2 * math.sin(theta2 * radians)

    p0 = [x1, y1, 0]
    p1 = [x, y, 0]
    p2 = [x2, y2, 0]
    rs.AddLine(p1, p0)
    rs.AddLine(p1, p2)

    min_length = min_length * ratio
    max_length = max_length * ratio

    if (min_length > minimum) :
        start_left = start_angle - angle1
        branch(x1, y1, min_length, max_length,
              start_angle, min_angle, max_angle,
              minimum, ratio)

        start_right = start_angle + angle2
        branch(x2, y2, min_length, max_length,
              start_angle, min_angle, max_angle,
              minimum, ratio)

branch(0,0, 60, 100, 95, 5, 40, 1, 0.7)
```

RESULT



Module 158: 12.4 Subdivision

12. HIERARCHICAL STRUCTURE

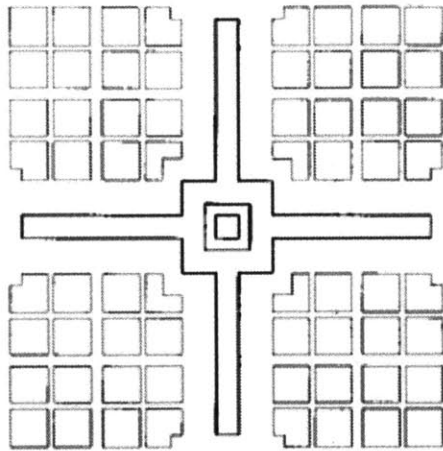
12.4.2 RECURSIVE SUBDIVISION

Figure 12-28 introduces another type of composition that can be generated by means of a recursive procedure. It is a slightly simplified plan of the famous garden of the Taj Mahal (as it existed originally). This type of plan is produced by recursively subdividing a square into four squares by paths in the form of a +. Here is a procedure to generate such plans:

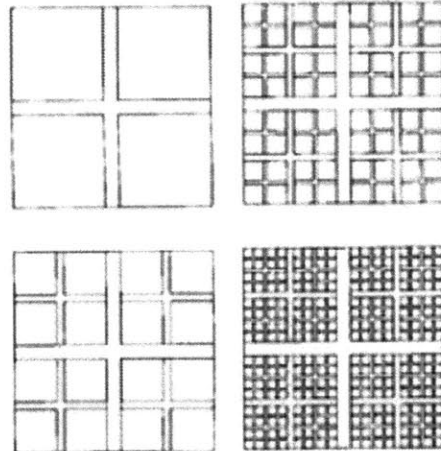
[samples code on the right side]

Here the design variables are Diameter of the outer square, Width of the initial path, Ratio of a path's width to that of its predecessor, and Minimum size for a square. Figure 12-29 shows a few of the possible outputs.

Notice the very close similarity in organization of the procedures that generate the tree and Taj Mahal garden. Although the types of objects that they generate look very different, the underlying constructive logic is much the same.



12-28. A schematic plan of the garden of the Taj Mahal, Agra.



12-29. Recursively generated garden plans of the Taj Mahal type.

CODE

```

import rhinoscriptsyntax as rs

def divide(x_center, y_center, diameter, width, min, ratio):

    half_diameter = diameter / 2
    half_width = width / 2

    x1 = x_center - half_diameter
    x2 = x_center - half_width
    x3 = x_center + half_width
    x4 = x_center + half_diameter

    y1 = y_center - half_diameter
    y2 = y_center - half_width
    y3 = y_center + half_width
    y4 = y_center + half_diameter

    p0 = [x1, y3, 0]
    p1 = [x2, y3, 0]
    p2 = [x2, y4, 0]
    p3 = [x3, y4, 0]
    p4 = [x3, y3, 0]
    p5 = [x4, y3, 0]
    p6 = [x4, y2, 0]
    p7 = [x3, y2, 0]
    p8 = [x3, y1, 0]
    p9 = [x2, y1, 0]
    p10 = [x2, y2, 0]
    p11 = [x1, y2, 0]
    p12 = [x1, y3, 0]
    pts = [p0,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12]
    rs.AddPolyline(pts)

    if (half_diameter > min):
        new_diameter = half_diameter - half_width

        d4 = (half_diameter - half_width) / 2

        x1 = x_center - d4 - half_width
        x2 = x_center + d4 + half_width
        y1 = y_center + d4 + half_width
        y2 = y_center - d4 - half_width

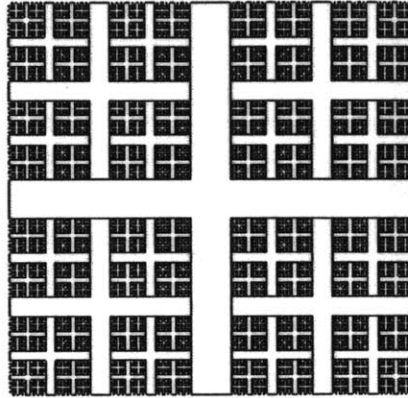
        width = width * ratio

        divide(x1, y1, new_diameter, width, min, ratio)
        divide(x2, y1, new_diameter, width, min, ratio)
        divide(x1, y2, new_diameter, width, min, ratio)
        divide(x2, y2, new_diameter, width, min, ratio)

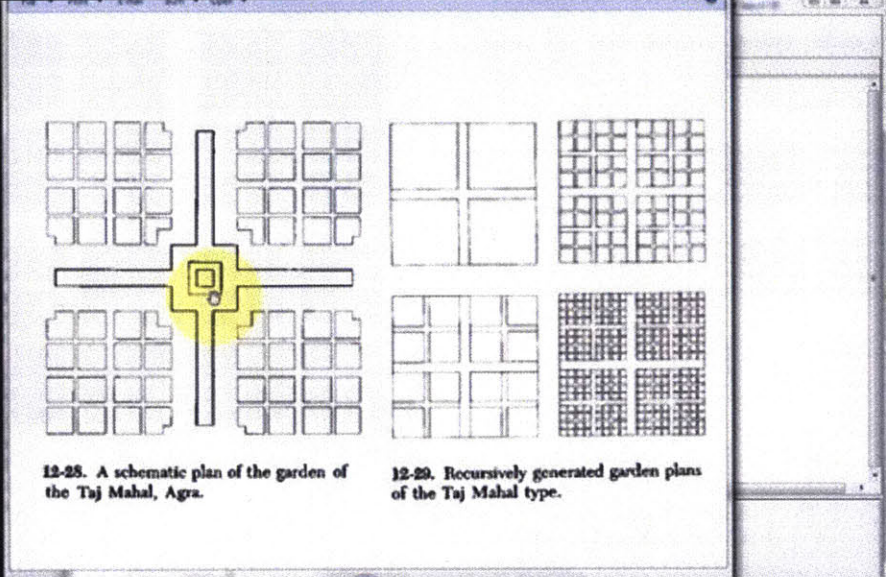
divide(0,0, 100,10,1,0.5)

```

RESULT



12.4.2 How to draw a recursive garden (Taj Mahal like)?



12-25. A schematic plan of the garden of the Taj Mahal, Agra.

12-20. Recursively generated garden plans of the Taj Mahal type.

0:01 / 14:46

Module 159: 12.5 Alternative

12. HIERARCHICAL STRUCTURE

12.5 ALTERNATIVE STRUCTURES

So far in our examples we have chosen obvious and natural ways to break drawings down into hierarchies of elements and subsystems. But it is important to remember that there are always alternative ways to define this hierarchy, and that different spatial relations of elements and subsystems may be taken to be the essential ones. Consider the plan party shown in figure 12-30. We might reasonably interpret the central part as - a circle nested within a square - there are two elements, and the essential relationship between them is of concentricity and equality of diameter. The design variables become the X and Y coordinates of the center point and Diameter.

Another way to parse this composition is shown in figure 12-31. Now there are two different subsystems, each consisting of two elements, and the design variables are X, Y, Top_width, Top_height, Bottom_width, Bottom_height, Arc_diameter, and Arc_spacing we can generate variants such as those illustrated in figure 12-32.

Some English sentences may also be parsed in alternative ways, and each alternative yields a different interpretation. A famous example is:

Time flies like an arrow.

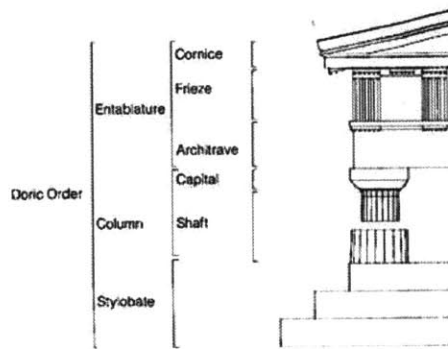
It seems most natural to take "flies" as the verb, but you can also take "like" as the verb, to yield a sentence describing the dietary preferences of "time flies", or you can take "time" as the verb, to yield an imperative. Similarly, in graphic compositions, parsing in different ways yields different interpretations. Since even simple graphic compositions can be parsed in alternative ways, the semantic properties of graphic languages are rich and complex. By committing ourselves to a particular hierarchy of elements and subsystems, expressed as a hierarchy of named procedures, we radically simplify and clarify the semantics.

Module 160: Exercise 8

12. HIERARCHICAL STRUCTURE

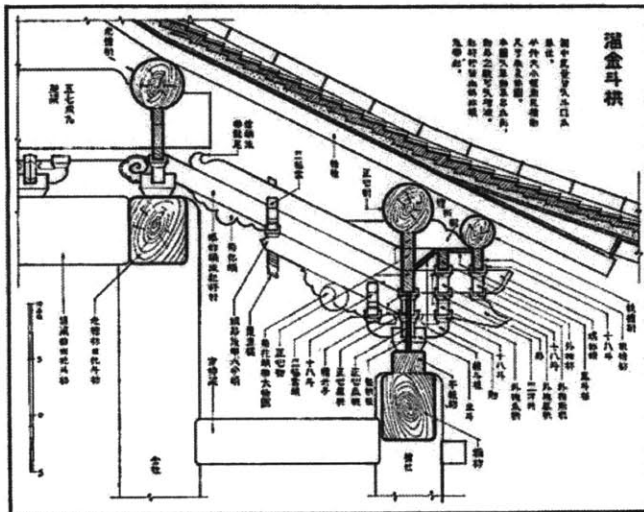
EXERCISES

1. All of the elements and subsystems of the Doric order have names (fig. 12-33). Draw a tree diagram that depicts this hierarchy. Then write a program, structured in the same way, that generates the order. (Simplify the details where necessary to reduce the task to manageable proportions.)



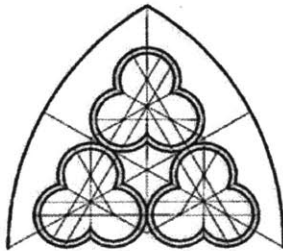
12-33. Well-defined hierarchy of elements and subsystems—all with traditional names—in the Doric order.

2. The elements and subsystems of traditional Chinese timber construction also have standard names (fig. 12-34). Draw the corresponding tree diagram write a program that expresses this structure to generate a drawing of the section of such a building. Take advantage of the recursive pattern of the bracket system. (Once again, simplify the details as necessary.)



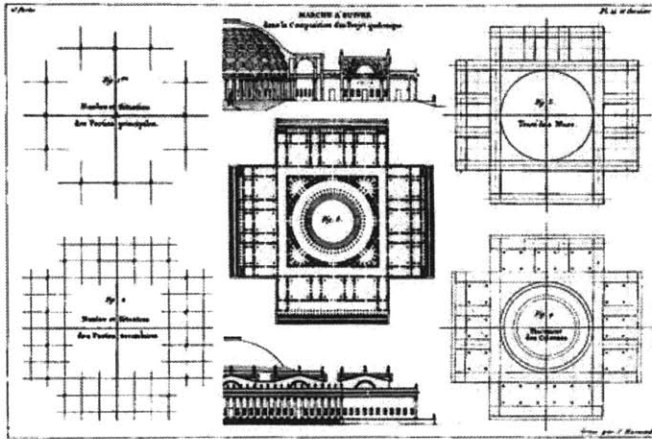
12-34. Named elements and subsystems in traditional Chinese timber construction.

3. An example of gothic tracery, as drawn by Eugene Viollet-le-Duc, is illustrated in figure 12-35. Work out a reasonable way to break this design down into a hierarchy of elements and subsystems, and draw the tree diagram. Write a program, structured in this way, that generates a single-line diagram of the design. Then, if you want to go further, elaborate it to show the thickness and detailing of the tracery members.



12-35. The analysis of gothic tracery by Viollet-le-Duc.

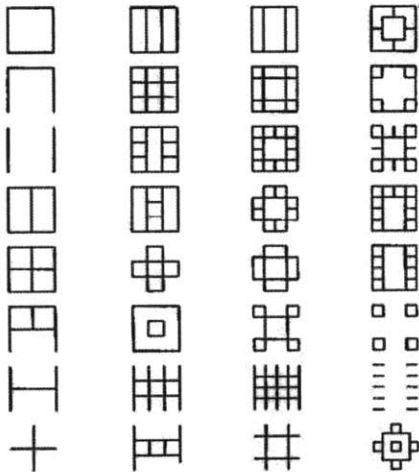
4. The French architectural theorist Jean Nicholas-Louis Durand produced many beautiful plates demonstrating how architectural compositions could be understood as -combinations- built up from lower-level vocabulary elements. Figure 12-36 shows an example. Select one of Durand's combinations for careful analysis. What are the parts and subparts? What are the essential spatial relations? What are the design variables? On the basis of your analysis, write a program that generates an interesting series of variants on this architectural theme. Use a top-down programming strategy that parallels the sequence of refinement steps by Durand.



12-36. The combination of vocabulary elements, as illustrated by Davrand.

Please upload your python file: No file chosen

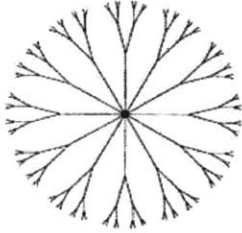
5. Another of Davrand's plates is shown in figure 12-37. This one shows a set of plan paths -skeletons- of axes and simple geometric figures that are used to establish the essential spatial relations in a plan. Select one of these paths and, using a simple vocabulary of wall and column elements, write a program that generates variant plans based on it.



12-37. Plan partis, as illustrated by Davrand.

Please upload your python file: No file chosen

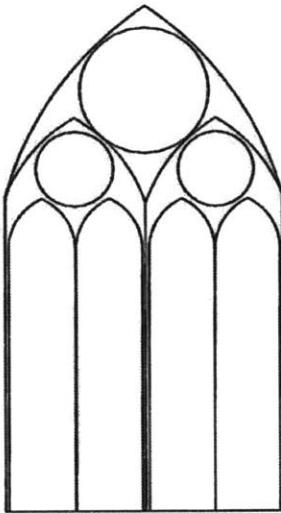
6. Figure 12-38 illustrates a tree drawn in plan. Write a recursive procedure that generates trees of this type. Make the depth of the recursion (the number of levels of branching) one of the design variables.



12-38. A tree in plan.

Please upload your python file: No file chosen

7. Gothic tracery is often recursive (fig. 12-39). That is, a large pointed arch is subdivided into two smaller pointed arches, each of which is further subdivided in the same way, and so on. Write a recursive procedure to generate such tracery designs.



12-39. Gothic window tracery.

Please upload your python file: No file chosen

8. We have seen how principles of architectural and graphic composition can be expressed in terms of the control constructs of Python: block structure, conditionals, repetition, and recursion. An alternative approach is to express rules of composition as productions organized in a production system (Stiny 1980). Compare the advantages and

Synthetic Tutor

disadvantages of the two approaches.

Please upload your python file: No file chosen

Module 161: 12.5 Top Down

12.

HIERARCHICAL STRUCTURE

12.5.1 TOP-DOWN PROGRAMMING

When you write a program to generate a complex composition of elements and subsystems, it is usually best to work top-down. Begin with a procedure that puts together the whole thing from two or more major subsystems. Within this top-level procedure, define the names of the major subsystems, their parameters, and their spatial relations. Write very schematic, - dummy - versions of the procedures to generate these major subsystems (draw them just as rectangles, for example), and make sure that this version of your program runs correctly.

Next, write fully developed versions of the procedures to generate the major subsystems. Within these, define the names and parameters of the lower-level subsystems from which they are composed, and establish the spatial relations of the lower-level subsystems. Using - dummy - versions of the procedures for the lower-level subsystems, check that this elaborated version of your program works correctly. Now repeat the process for the lower-level subsystems, and so on through a sequence of increasingly refined versions of your program, until you reach the procedures to draw the lowest level vocabulary elements.

This parallels the common design strategy of working down to the details from an initial, rough sketch of the whole. It has several important advantages. You always have a running program, and you can always see the whole composition. Design variables are introduced one by one in a systematic way. And you can, at any point, choose not to refine any further.

Module 162: 12.6 Summary

12. HIERARCHICAL STRUCTURE

12.6 SUMMARY

In this chapter we have explored the ways in which graphic compositions can usefully be broken down into hierarchies of elements and subsystems. We have seen how such hierarchies are expressed directly in the block structures established by procedure declarations in graphics programs. The value parameters of procedures allow position and shape information to be passed in to blocks, and the variable parameters allow position and shape information to be passed out. This information is used to relate elements and subsystems correctly to each other the spatial relations of the elements and subsystems of a composition are specified by means of arithmetic expressions and assignments relating the values of parameters.

Module 163: 14.1 Transform

14. TRANSFORMATIONS.

A transformation carries an object from one state to another. For example, you can transform a polygon from the unshaded to the shaded state (fig. 14- 1a). (We discussed a procedure to do this in the last chapter.) Or you can transform its edges from solid to dashed (fig. 14-1b). You might transform its position by translating or rotating it (fig. 14-1c), or its size by scaling it (fig. 14-1d). Yet again, you might transform its shape by distorting it in some way (fig. 14-1e).

In this chapter we shall see how to write Python programs that transform drawings by translating, rotating, reflecting, and scaling them within coordinate systems. Such programs work by transforming a graphic data structure from an initial state that represents the initial state of the drawing to a new state that represents the new state of the drawing.

14.1 GEOMETRIC TRANSFORMATIONS OF A POINT

Let us begin by considering a single point with coordinates (X, Y) in some coordinate system (fig. 14-2a). Translation of this point through a distance Tx parallel to the X axis (fig. 14-2b) is represented by the assignment `X = X + Tx`. Similarly, translation through a distance Ty parallel to the Y axis (fig. 14-2c) is represented by

$$Y = Y + Ty$$

Since any translation can be resolved into a component Tx and a component Ty, a translation in general (fig. 14-2d) is represented by the pair of assignments

$$Y = Y + Ty$$

The variables involved here (X, Y, Tx, and Ty) might either be integer or real. In general, it is most convenient to perform transformation calculations upon real variables, then to round the results back to integer values for display

$$\begin{aligned} X &= X + Tx \\ Y &= Y + Ty \end{aligned}$$

in a screen coordinate system. This will become increasingly evident as we go along.

Rotations can be represented in a similar way. A rotation of angle Theta clockwise about the origin of the coordinate system (fig. 14-2e) is represented

$$\begin{aligned} X &= X * \text{math.cos}(\text{theta}) + Y * \text{math.sin}(\text{theta}) \\ Y &= - X * \text{math.sin}(\text{theta}) + Y * \text{math.cos}(\text{theta}) \end{aligned}$$

Scaling is represented by multiplication by a scale factor Scale_F (fig. 14-2f) as follows

$$\begin{aligned} X &= X * \text{SCALE_F} \\ Y &= Y * \text{SCALE_F} \end{aligned}$$

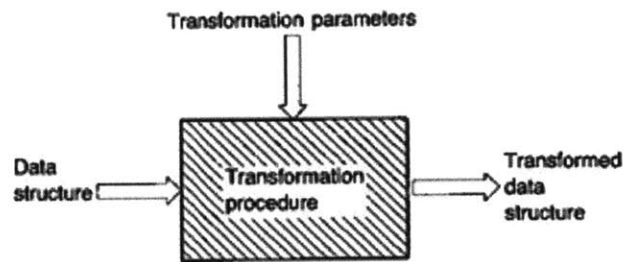
If Scale_F is an integer variable, you can enlarge by any integer factor. If Scale_F is a real variable, you can enlarge by some real factor, or you can reduce by using a factor between 0 and 1. If Scale_F has a value of 0, the point is shifted to the origin of the coordinate system.

Unequal scaling (stretch) is represented by using two different scale factors as follows.

$$X = X * \text{SCALE_X}$$

$$Y = Y * SCALE_Y$$

If Scale_F is an integer variable, you can enlarge by any integer factor. If Scale_F is a real variable you can enlarge by some real factor or you can.



14-5. A transformation procedure.

```
CODE
import rhinoscriptsyntax as rs

point = [10,0,0]
pt_id = rs.AddPoint(point)

def translationX(obj,tx):
    translation = [tx,0,0]
    obj = rs.MoveObject(obj,translation)
    return obj

translationX(pt_id,10)

def translationY(obj,ty):
    translation = [0,ty,0]
    obj = rs.MoveObject(obj,translation)
    return obj

translationY(pt_id,10)

def translation(obj,tx,ty):
    translation = [tx,ty,0]
    obj = rs.MoveObject(obj,translation)
    return obj

translation(pt_id,10,10)

def rotation(obj,angle):
    center=[0,0,0]
    rs.RotateObject(obj,center,angle)

rotation(pt_id,60)

def scaling(obj,scale_factor):
    origin=[0,0,0]
    scale = [scale_factor,scale_factor,1]
    rs.ScaleObject(obj,origin,scale)

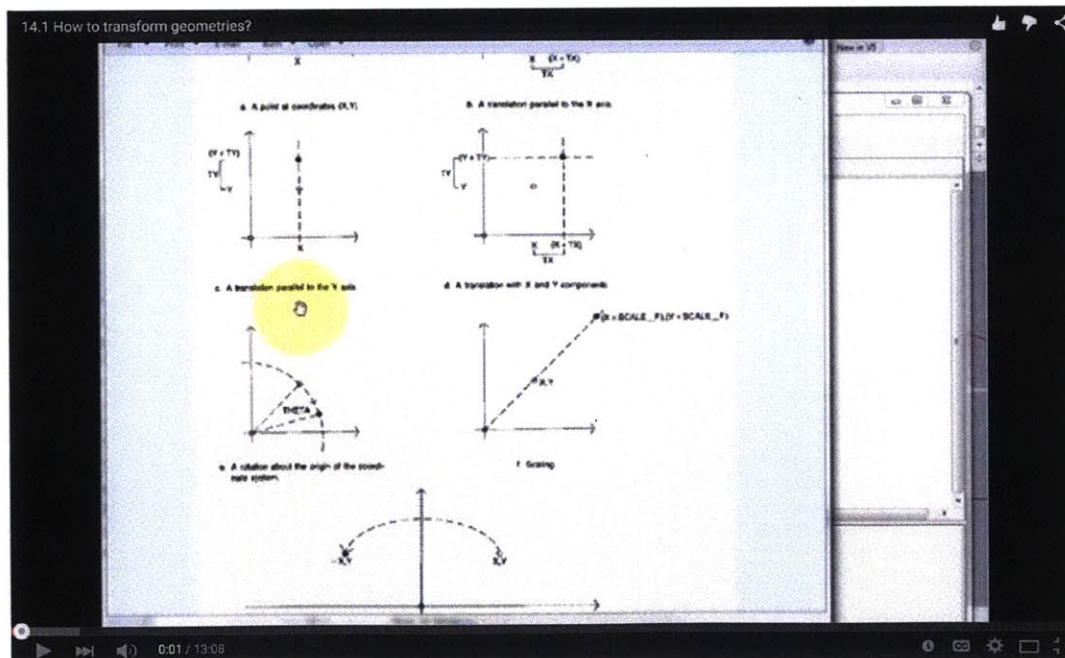
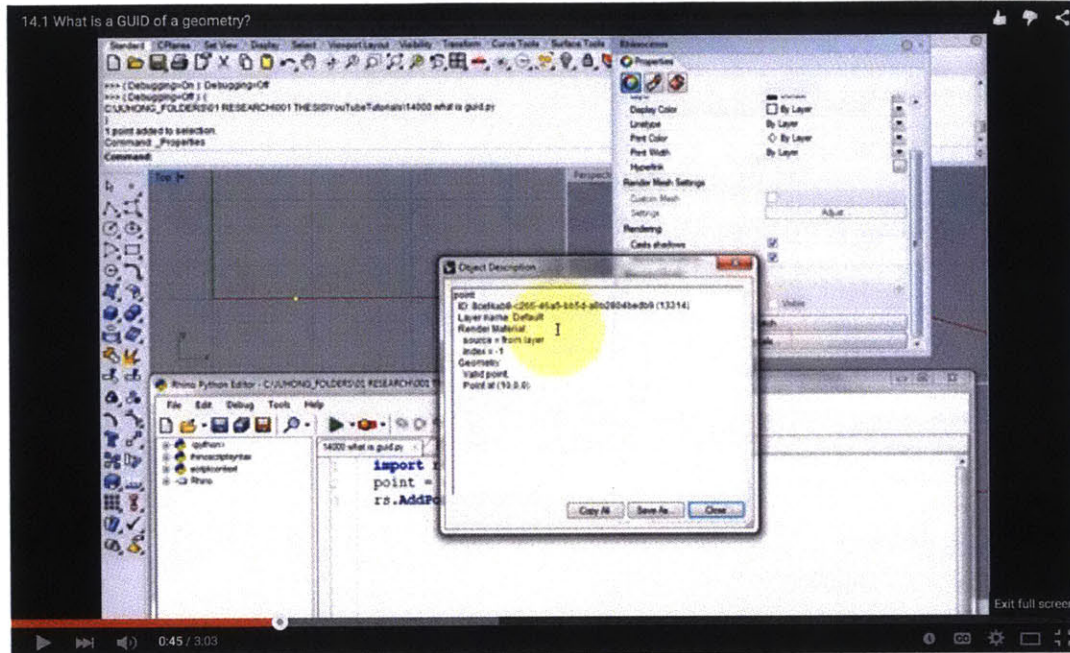
scaling(pt_id,0.5)

def unequal_Scaling(obj,scale_x,scale_y):
    origin=[0,0,0]
    scale = [scale_x,scale_y,1]
    rs.ScaleObject(obj,origin,scale)

unequal_Scaling(pt_id,0.4,0.2)
```

RESULT

Synthetic Tutor



Module 164: 14.2 Data Structure

14. TRANSFORMATIONS.

14.2 TRANSFORMATIONS ON GRAPHIC DATA STRUCTURES

A geometric transformation applied to a drawing transforms every point(X, Y) in the original drawing to some new point (New_X, New_Y) in the transformed drawing. So it might seem that we would have to operate on every point in our screen coordinate system in order to transform a drawing. This, fortunately, is not the case. First, translation, rotation, reflection, and scaling all transform straight lines into straight lines. Second, we represent straight lines by their endpoints. So we have only to apply transformations to the endpoints of each line in a drawing, then draw lines between the transformed endpoints in the usual way (fig. 14-3).

Consider, for example, the simple drawing shown in figure 14-4a. Using the technique that we discussed in the last chapter, we might store it in a XY one-dimensional array.

The first field specifies whether a line is to be drawn, or whether an invisible movement is to be made this remains unaffected by a transformation. The second and third fields specify an endpoint coordinate. Our purpose here, though, is not to store a completed drawing in screen coordinates, but a state of a drawing that is to be transformed to another state.

Since every endpoint must be operated on, we need a loop to step through the data structure. The following code, for example, translates the drawing a distance of 500 in the X direction and 600 in the Y direction

[sample codes on the right side]

Figure 14-4c shows the transformed state of the drawing, and figure 14-4d shows the transformed state of the data structure.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs

def rectangle(x, y, length, width):
    x1 = x - (length / 2)
    y1 = y - (width / 2)
    x2 = x1 + length
    y2 = y1 + width

    p0 = [x1, y1, 0]
    p1 = [x2, y1, 0]
    p2 = [x2, y2, 0]
    p3 = [x1, y2, 0]
    p4 = [x1, y1, 0]
    pts = [p0, p1, p2, p3, p4]
    rs.AddPolyline(pts)

def circle(x, y, radius):
    center = [x, y, 0]
    rs.AddCircle(center, radius)

def column(x, y, diameter):
    rectangle(x, y, diameter, diameter)
    circle(x, y, diameter/2)

def plinth(x, y, length, width, steps, step_size):
    repeat = range(steps)
    for count in repeat:
        rectangle(x, y, length, width)
        length = length + step_size
        width = width + step_size

def cella(x, y, length, diameter, width):
    rectangle(x, y, length, diameter)
    y = y + width
    rectangle(x, y, length, diameter)

def outside(count_x, count_y,
            x_start, x_finish,
            y_start, y_finish):
    cond1 = count_x < x_start
    cond2 = count_x > x_finish
    cond3 = count_y < y_start
    cond4 = count_y > y_finish
    outside = cond1 or cond2 or cond3 or cond4
    return outside

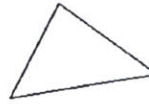
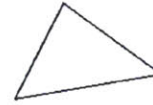
def grid(x_initial, y_initial, spacing, diameter,
        nx, ny, x_start, x_finish, y_start, y_finish,
        steps, step_size):
    repeat_ny = range(ny)
    repeat_nx = range(nx)
    y = y_initial
    for count_y in repeat_ny:
        x = x_initial
        for count_x in repeat_nx:
            isOutside = outside(count_x, count_y,
                               x_start, x_finish,
                               y_start, y_finish)
            if isOutside:
                column(x,y,diameter)
            x = x + spacing
        y = y + spacing

def draw_temple():
    x = rs.GetReal('enter x coordinate', 0)
    y = rs.GetReal('enter y coordinate', 0)
    diameter = rs.GetReal('enter column diameter', 300)
    spacing = rs.GetReal('enter column spacing', 1000)
    nx = rs.GetInteger('enter number of columns in x-direction', 20)
    ny = rs.GetInteger('enter number of columns in y-direction', 10)
    x_start = rs.GetInteger('enter x start number', 3)
    x_finish = rs.GetInteger('enter x finish number', 16)
    y_start = rs.GetInteger('enter y start number', 3)
    y_finish = rs.GetInteger('enter y finish number', 6)
    steps = rs.GetInteger('enter number of steps', 5)
    step_size = rs.GetInteger('enter width of step', 1000)

    temple(x, y, spacing, diameter,
          nx, ny,
          x_start, x_finish, y_start, y_finish,
          steps, step_size)

draw_temple()
```

RESULT



Module 165: 14.3 Procedure 114.
TRANSFORMATIONS.

14.3 TRANSFORMATION PROCEDURES

Just as we have used functions to name and perform certain kinds of coordinate calculations and procedures to name and generate graphic vocabulary elements, it is a useful abstraction to use procedures to name and perform geometric transformations. In general, a transformation procedure accepts as input a data structure representing a drawing (such as Points: pts in the sample code), together with transformation parameters (such as Tx ,Ty ,Theta, and Scale_F), and produces as output the transformed data structure. This process is diagramed in figure 14-5.

An invocation of a procedure to translate a drawing might look like this

```
TRANSLATE (POINTS, Tx, Ty)
```

Synthetic Tutor

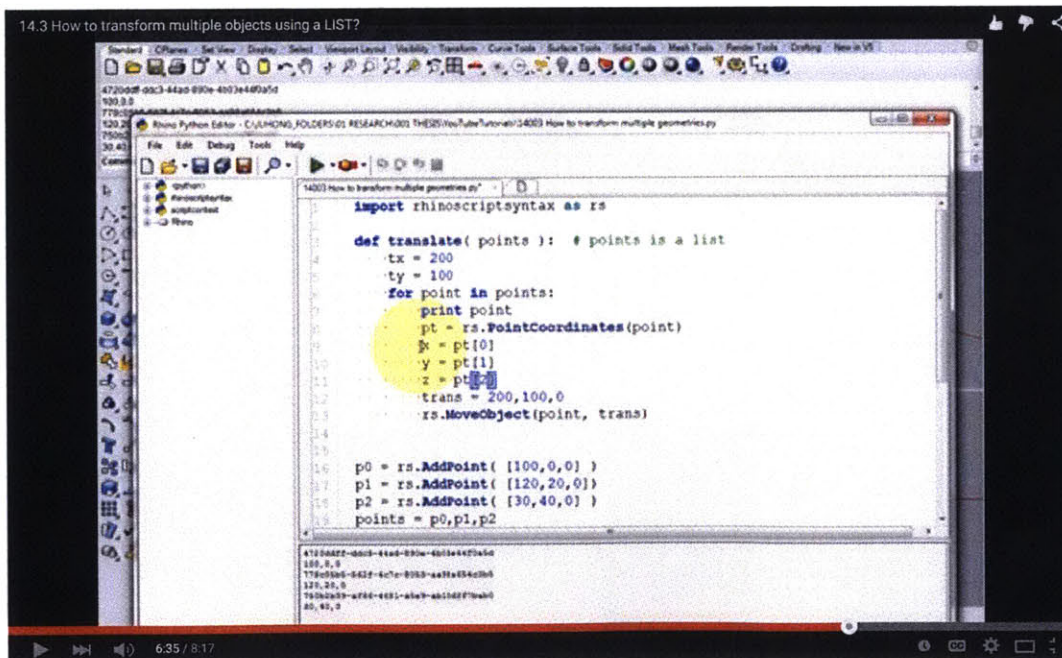
CODE

```
import rhinoscriptsyntax as rs

def translate(pts):
    tx = 200
    ty = 100
    for point in pts:
        pt = rs.PointCoordinates(point)
        x = pt[0] + tx
        y = pt[1] + ty
        rs.AddPoint([x,y,0])

p0 = rs.AddPoint([100,0,0])
p1 = rs.AddPoint([120,20,0])
p2 = rs.AddPoint([30,60,0])
pts = [p0,p1,p2]
translate(pts)
```

RESULT



Module 166: 14.3 Procedure 2

14.
TRANSFORMATIONS.

14.3 TRANSFORMATION PROCEDURES

If the drawing is stored in a one-dimensional list (pts in the sample code), the procedure Translate will look something like this

[sample codes on the right side]

Notice that Points(pts in the sample code) is declared as a variable parameter of Translate, so that the original drawing can be passed into the procedure, and the transformed drawing can be passed back out. It is also efficient to pass lists as variable parameters if they are passed as value parameters, a copy must be created by the compiler when the procedure is invoked.

Synthetic Tutor

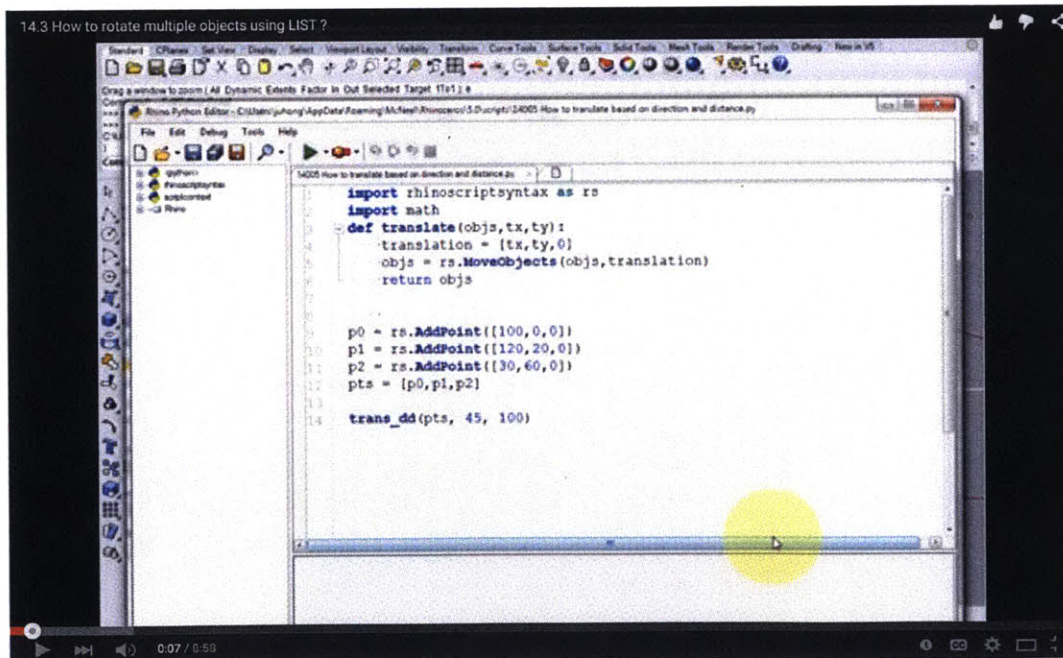
CODE

```
import rhinoscriptsyntax as rs

def translate(obj,tx,ty):
    translation = [tx,ty,0]
    obj = rs.MoveObject(obj,translation)
    return obj

p0 = rs.AddPoint([100,0,0])
p1 = rs.AddPoint([120,20,0])
p2 = rs.AddPoint([30,60,0])
pts = [p0, p1, p2]
tx = 200
ty = 100
translate(pts,tx,ty)
```

RESULT



Module 167: 14.3 Procedure 3

14.
TRANSFORMATIONS.

14.3 TRANSFORMATION PROCEDURES

Another useful way to parameterize a translation procedure is by direction and distance (fig. 14-6). The invocation looks like this:

```
trans_dd (points, direction, distance):
```

The procedure `trans_dd` can easily be build using `translate`

[sample codes on the right side]

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def translate(obj,tx,ty):
    translation = [tx,ty,0]
    obj = rs.MoveObject(obj,translation)
    return obj

def trans_dd(obj, direction, distance):
    radiance = 0.01745
    tx = distance * math.cos(direction * radiance)
    ty = distance * math.sin(direction * radiance)
    newPoints = translate(obj, tx, ty)
    return newPoints

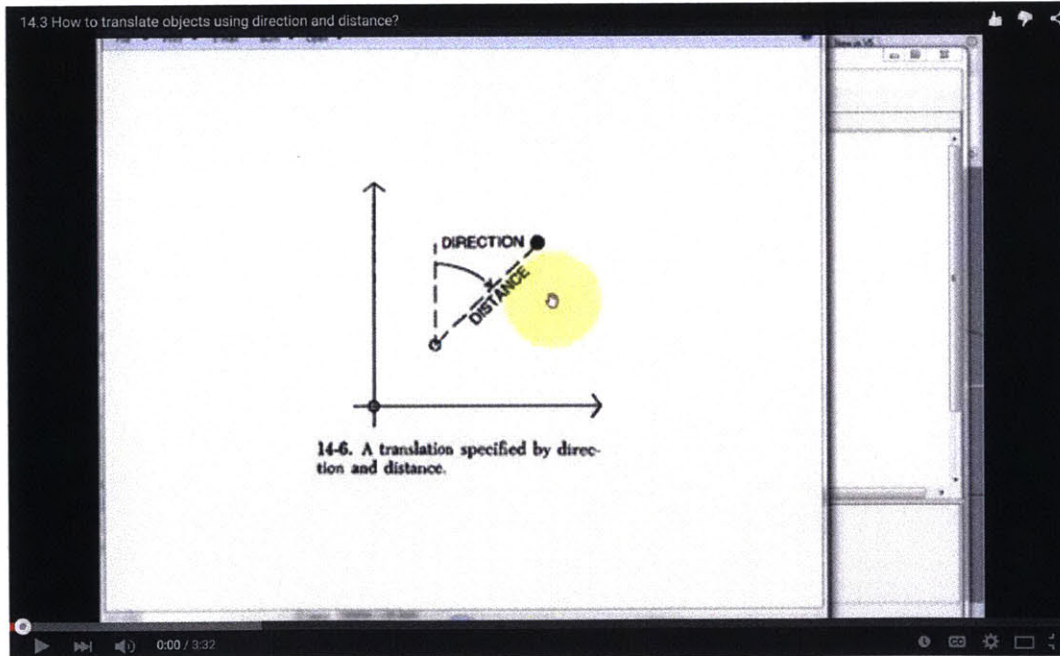
p0 = rs.AddPoint([100,0,0])
p1 = rs.AddPoint([120,20,0])
p2 = rs.AddPoint([30,60,0])
pts = [p0,p1,p2]
direction = 45
distance = 100
trans_dd(pts, direction, distance)
```

RESULT

□

□

□



Module 168: 14.3 Procedure 4

14.
TRANSFORMATIONS.

14.3 TRANSFORMATION PROCEDURES

Since this gives us the greatest flexibility, we shall use such list data structures in our examples from now on, unless otherwise indicated. Here is a procedure *Rotate*, to rotate about the origin of the coordinate.

[sample codes on the right side]

Note its close similarity to *Translate*.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def rotate_original(points, angle):
    newPoints = []
    for point in points:
        center = [0,0,0]
        newPoint = rs.RotateObject(point, center, angle)
        newPoints.append(newPoint)
    return newPoints

p0 = rs.AddPoint([100,0,0])
p1 = rs.AddPoint([120,20,0])
p2 = rs.AddPoint([30,60,0])
pts = [p0,p1,p2]
angle = 30
rotate_original(pts, angle)
```

RESULT

Module 169: 14.3 Procedure 5

14.
TRANSFORMATIONS.

14.3 TRANSFORMATION PROCEDURES

Finally, here is a procedure Scale, based on the same principles as Translate and Rotate.

[sample codes on the right side]

Synthetic Tutor

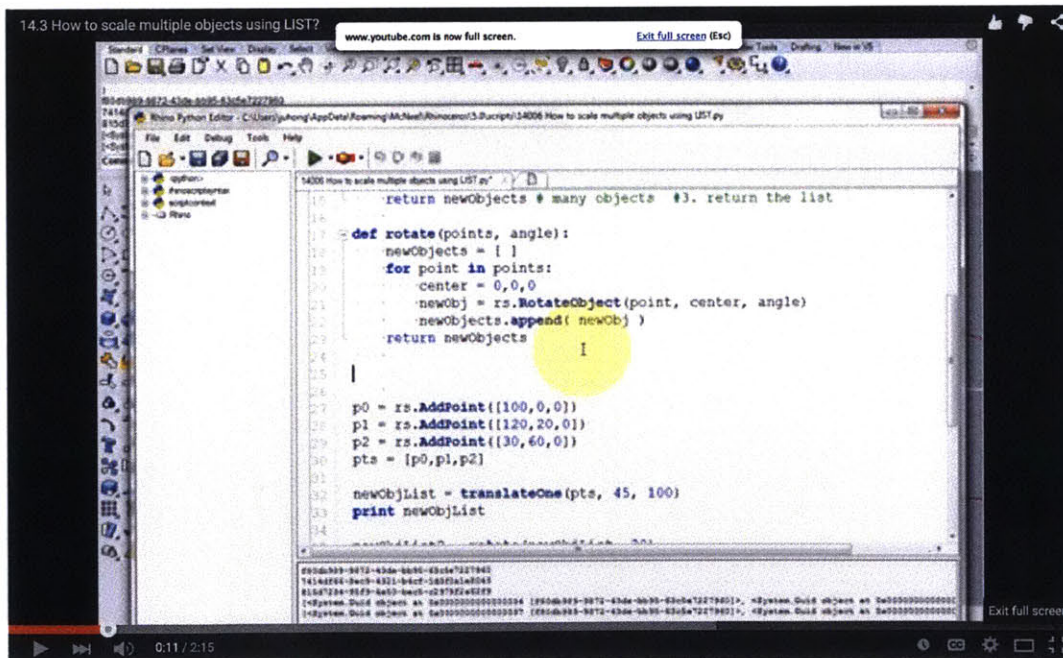
CODE

```
import rhinoscriptsyntax as rs
import math

def scaling(points, scale_factor):
    origin = [0, 0, 0]
    for point in points:
        rs.ScaleObject(point, origin, scale_factor)

p0 = rs.AddPoint([100, 0, 0])
p1 = rs.AddPoint([120, 20, 0])
p2 = rs.AddPoint([30, 60, 0])
pts = [p0, p1, p2]
scale_factor = [2, 3, 1]
scaling(pts, scale_factor)
```

RESULT



Module 170: 14.3 Concatenation 1

14. TRANSFORMATIONS.

14.3.1 CONCATENATION OF TRANSFORMATIONS

How could you rotate the drawing about its center point, rather than (as performed by Rotate) about the origin of the coordinate system (fig. 14-7)? A sequence of transformations is required to accomplish this. First translate the drawing's center point to the origin of the coordinate system (fig. 14-8a). Then rotate it about the origin by the required amount (fig. 14-8b). Finally, translate the drawing back to the original position (fig. 14-8c). More generally, how could you rotate the drawing about any arbitrary point (Cx ,Cy)? The required sequence of transformations is illustrated in figure 14-9. It can be executed by the following sequence of invocations of transformation procedures

```
TRANSLATE(POINTS, -CX, -CY)
ROTATE(POINTS, THETA)
TRANSLATE(POINTS, CX, CY)
```

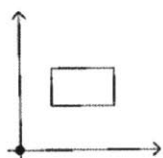
Rotation about an arbitrary point (cx, cy) is a very common operation, so it would be convenient to have a procedure to perform it. This would be invoked as follows:

```
ROTATE_XY(POINTS, CX, CY, THETA)
```

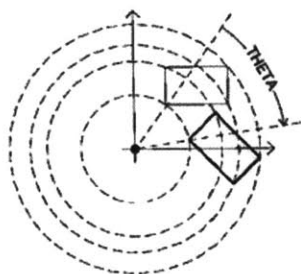
The procedure is easy to write using translate and rotate thus:

[sample codes on the right side]

This simple example illustrates an important theoretical point. Any sequence of translation, rotation, reflection, and scaling transformations can be regarded as a single transformation that carries an object from an initial state to the final state, which results from the last transformation in the sequence. The combination of transformations, in this way, is known as concatenation. The concatenated transformation may be given a name, such as Rotate_XY, and can be executed by a procedure, in the same way that we named the elementary transformations (for example, Translate) and wrote procedures to perform them.

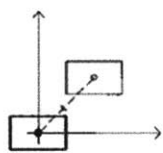


a. The original figure.

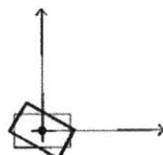


b. Each vertex rotates around the origin.

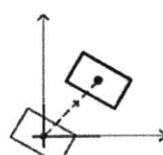
14-7. A figure rotated about the origin of the coordinate system.



a. Translate to the origin.

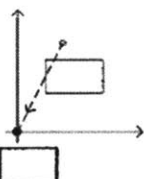


b. Rotate about the origin.

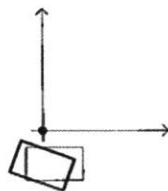


c. Translate back.

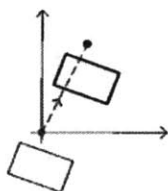
14-8. Steps in rotation of a figure about its center point.



a. Translate the center of rotation to the origin.



b. Rotate about origin.



c. Translate back.

14-9. Steps in rotation of a figure about an arbitrary point.

CODE

```

import rhinoscriptsyntax as rs
import math

def translate(objs,tx,ty):
    translation = [tx, ty, 0]
    objs = rs.MoveObjects(objs, translation)
    return objs

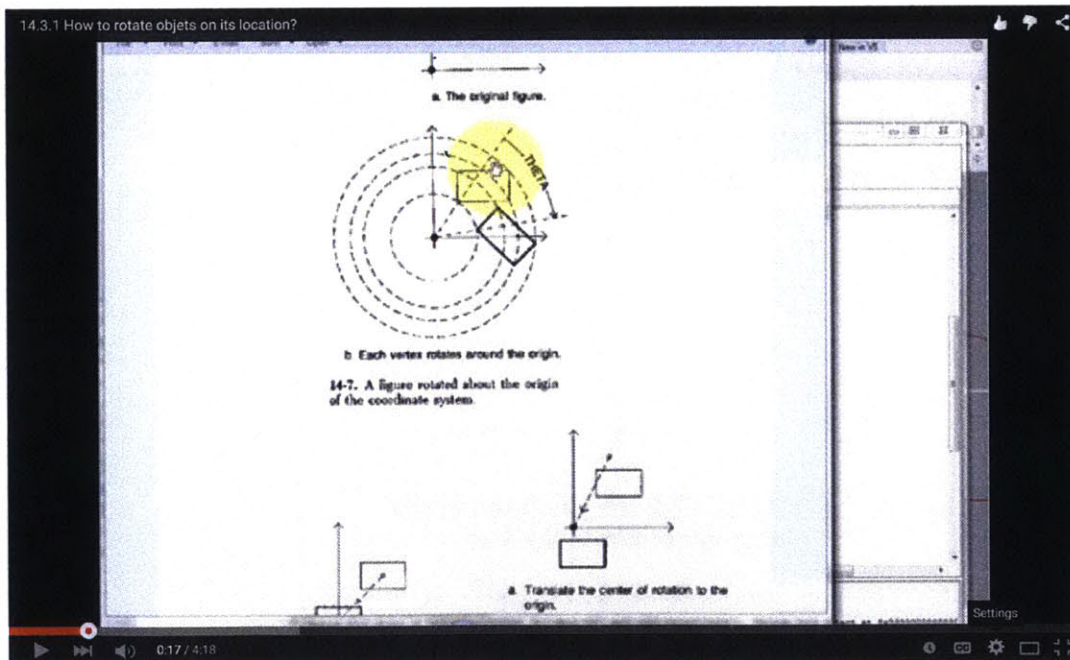
def rotate(objs,angle):
    center = [0, 0, 0]
    objs = rs.RotateObjects(objs, center, angle)
    return objs

def rotate_xy(objs, cx, cy, theta):
    objs = translate(objs, -cx, -cy)
    objs = rotate(objs, theta)
    objs = translate(objs, cx, cy)
    return objs

p0 = rs.AddPoint([100, 0, 0])
p1 = rs.AddPoint([120, 20, 0])
p2 = rs.AddPoint([30, 60, 0])
pts = [p0, p1, p2]
rotate_xy(pts, 0, 0, 60)

```

RESULT



Module 171: 14.3 Concatenation 2

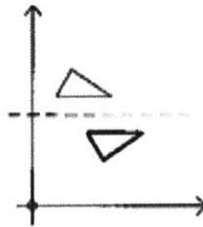
14. TRANSFORMATIONS.

14.3.1 CONCATENATION OF TRANSFORMATIONS

Here is another, more complicated example of concatenation. A glide reflection across an arbitrary axis parallel to the X axis is illustrated in figure 14-10. It is a concatenation of reflection and translation and is specified by the Y coordinate of the axis and the X translation. A glide reflection procedure, then, is invoked like this

[sample codes on the right side]

There are much more efficient ways to perform concatenated transformations than to perform component transformations in sequence, as for example, in our procedures Rotate_XY and Glide. In particular, you can use concatenated transformation matrix. We need not concern ourselves with this technique here, but if you are interested, it is explained in detail in several of the computer graphics texts listed in the Bibliography.



14-10. Glide reflection across an arbitrary axis parallel to the X axis.

CODE

```
import rhinoscriptsyntax as rs

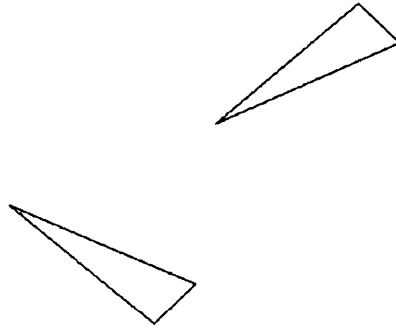
def translate(objs,tx,ty):
    translation = [tx, ty, 0]
    objs = rs.MoveObject(objs, translation)
    return objs

def scale(objs, scale_x, scale_y):
    origin = [0, 0, 0]
    scale = [scale_x, scale_y, 1]
    objs = rs.ScaleObject(objs, origin, scale)
    return objs

def glide(objs, axis_x, axis_y):
    objs = rs.CopyObject(objs)
    objs = translate(objs, 0, -axis_y)
    objs = scale(objs, 1, -1)
    objs = translate(objs, axis_x, axis_y)

p0 = [100, 0, 0]
p1 = [120, 20, 0]
p2 = [30, 60, 0]
pts = [p0, p1, p2, p0]
objID = rs.AddPolyline(pts)
glide(objID, 100, 80)
```

RESULT



Module 172: 14.3 Order

14.
TRANSFORMATIONS.

14.3.2 THE ORDER OF TRANSFORMATIONS

The result of performing two translations like

```
TRANSLATE (POINTS, 100, -100)
TRANSLATE (POINTS, -200, -50)
```

is the same as that of performing them in reverse order (fig. 14-11)

```
TRANSLATE (POINTS, - 200, - 50)
TRANSLATE (POINTS, 100, - 100)
```

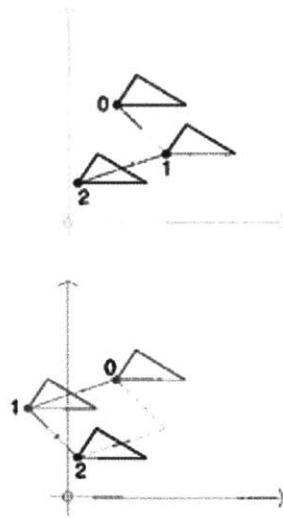
However, the result of this sequence of transformations

```
TRANSLATE (POINTS, 0, - 150)
ROTATE (POINTS, 90)
```

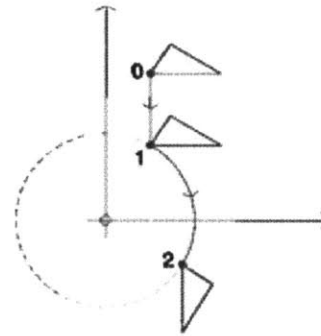
is not the same as that of its reverse:

```
ROTATE (POINTS, 90)
TRANSLATE (POINTS, 0, - 150)
```

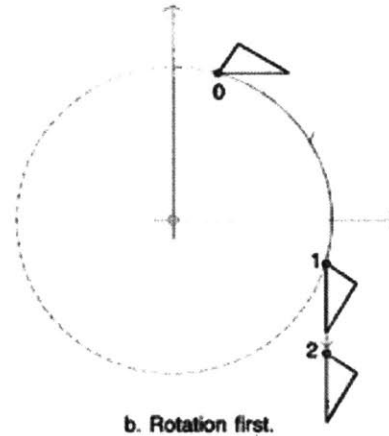
Figure 14-12 compares the two results. In general, then, the order of transformations matters. To achieve the results that you want, you must apply transformations in the appropriate sequence.



14-11. The same translations performed in different orders produce the same result.



a. Translation first.



b. Rotation first.

14-12. The same translations and rotations performed in different orders produce different results.

Module 173: 14.3 Operators

14. TRANSFORMATIONS.

14.3.3 TYPES OF GEOMETRIC OPERATORS

The transformation procedures that we have considered may be regarded as geometric operators, just as + , - , and sort are arithmetic operators. The name of the transformation procedure specifies the type of geometric operation that it performs (translation, glide reflection, and so on). The first parameter (pts, in our examples) specifies the object to be operated upon. Then come the parameters that specify the specific instance of this type of transformation that is to be applied-for example, a translation specified by Tx and Ty.

These procedures, you will notice, are similar to those that generate graphic vocabulary types. Just as you can build up a vocabulary out of which to construct graphic compositions, you can build up a -toolbox- of geometric operators for manipulating compositions. In both cases, you must abstract to determine the types that you will need, name these types, define their essences in the code of procedures, and define the parameters that specify instances.

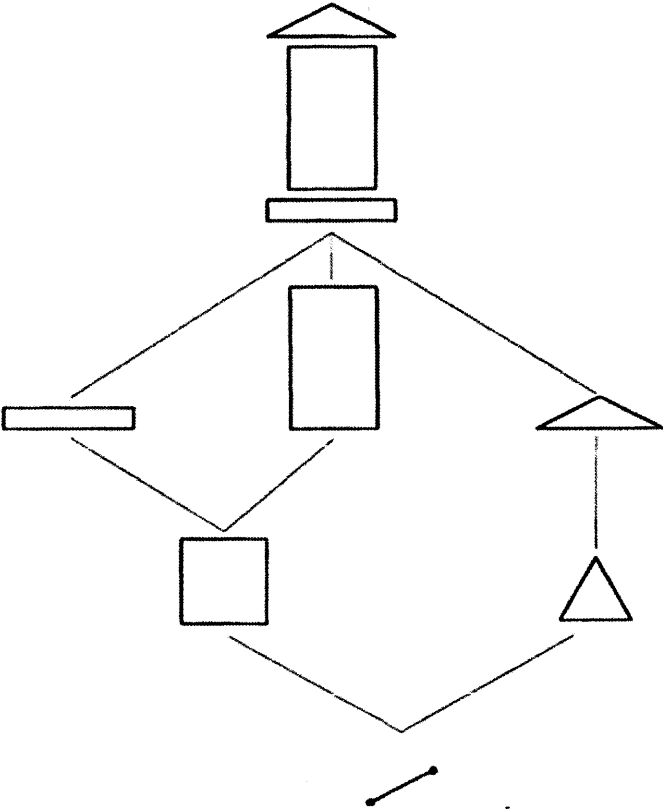
Module 174: 14.4 Hierarchy

14.
TRANSFORMATIONS.

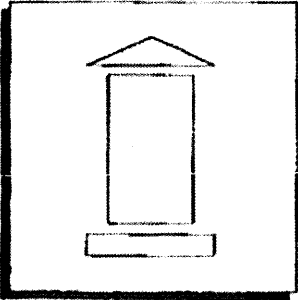
14.4 TRANSFORMATIONS AND HIERARCHICAL STRUCTURE

The hierarchical structure of the resulting drawing, as defined by the nesting of start and finish pairs, is illustrated in figure 14-25. And figure 14-26 illustrates what finally appears on the screen.

In chapter 12 we saw how to express the hierarchical structure of a graphic composition in the block structure of a program each part (element or subsystem) was given a name, and a procedure of that name specified the lower-level components and how these were spatially related. Then, in chapter 13, we saw how the same structure might be reflected in the hierarchical organization of a data structure. Now we can also describe structure in terms of the nesting of geometric transformations. A part is now defined as a set of lower-level



14-25. Hierarchical structure of the drawing generated by Window.



14-26. Screen display produced by Window.

Module 175: 14.4 Graphic

14. TRANSFORMATIONS.

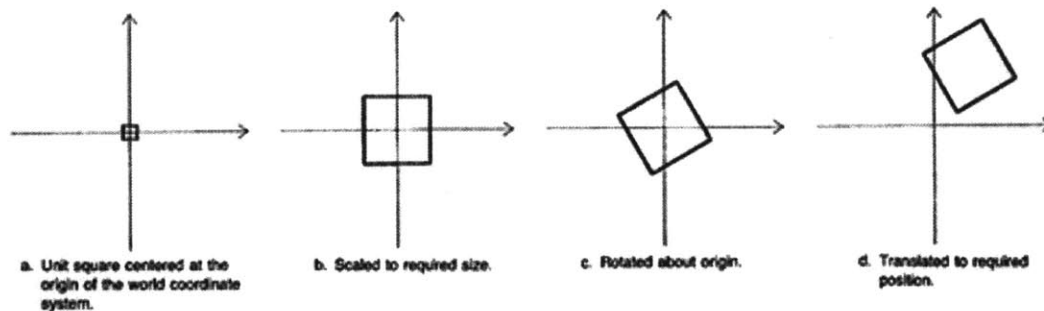
14.4.1 GRAPHIC VOCABULARIES REVISITED

The use of a world coordinate system and transformation operators requires modification of the approach to parameterization of graphic procedures that was introduced in chapter 7. We no longer need position parameters, since translations, rotations, and reflections are used to position instances in the composition. Neither do we need size parameters, since elements can be scaled up and down. We will, however, still frequently need parameters that control shape and repetition.

It will generally be most convenient to define graphic vocabulary elements with their center at the origin of the world coordinate system and with their most characteristic dimension set at unity. This makes it easy to scale an instance to the correct size, rotate it to the correct orientation, then translate it to the required position. Here, for example, is a procedure to generate a square of unit length, centered at the origin of the world coordinate system.

[sample codes on the right side]

These stages are illustrated in figure 14-27.



14-27. Stages in sizing and positioning a square.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def square():
    p1 = [-0.5, -0.5, 0]
    p2 = [0.5, -0.5, 0]
    p3 = [0.5, 0.5, 0]
    p4 = [-0.5, 0.5, 0]
    pts = [p1, p2, p3, p4, p1]
    square = rs.AddPolyline(pts)
    return square

def scale(obj,x,y):
    origin = [0, 0, 0]
    scale = [x, y, 1]
    result = rs.ScaleObject(obj, origin, scale, True)
    return result

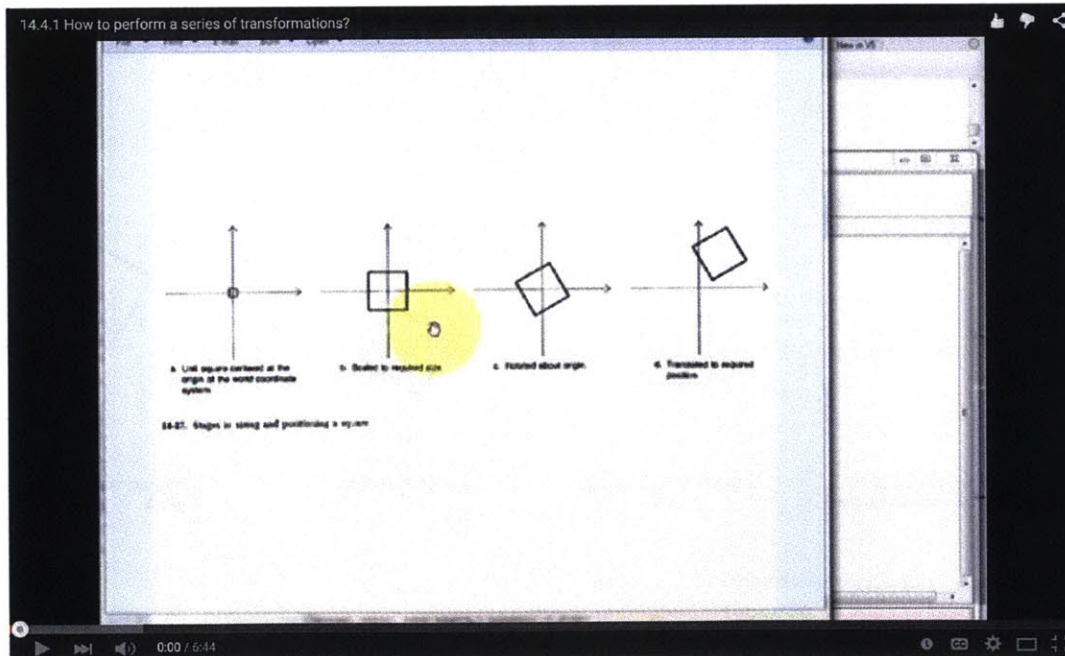
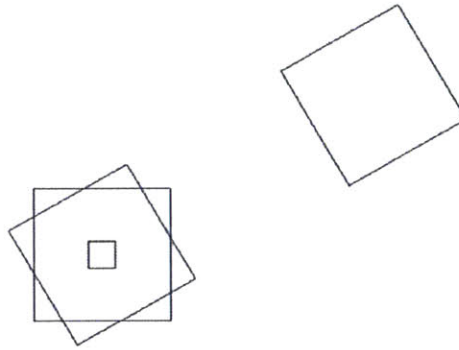
def rotate(obj,angle):
    center = [0, 0, 0]
    result = rs.RotateObject(obj, center, angle, None, True)
    return result

def translate(obj, x, y):
    translation = [x, y, 0]
    rs.CopyObject(obj, translation)

def transform():
    rec = square()
    rec = scale(rec, 5, 5)
    rec = rotate(rec,30)
    rec = translate(rec,10,6)

transform()
```

RESULT



Module 176: 14.5 Symmetry

14. TRANSFORMATIONS.

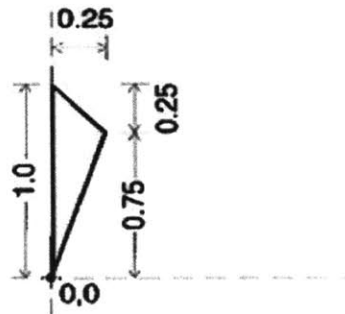
14.5 TRANSFORMATIONS, LOOPS, AND SYMMETRY

When transformation operators are applied repeatedly within loops, symmetrical patterns result. The graphic element to which they are applied becomes the repeating element of the pattern, and different kinds of symmetry result from using different combinations of transformation operators. To investigate this, let us define an asymmetrical vocabulary element, as follows

[sample codes on the right side]

This is illustrated in figure 14-28.

There are four kinds of plane symmetry rotational, dihedral, frieze, and wallpaper. We shall consider these in turn.



14-28. An asymmetrical vocabulary element.

Synthetic Tutor

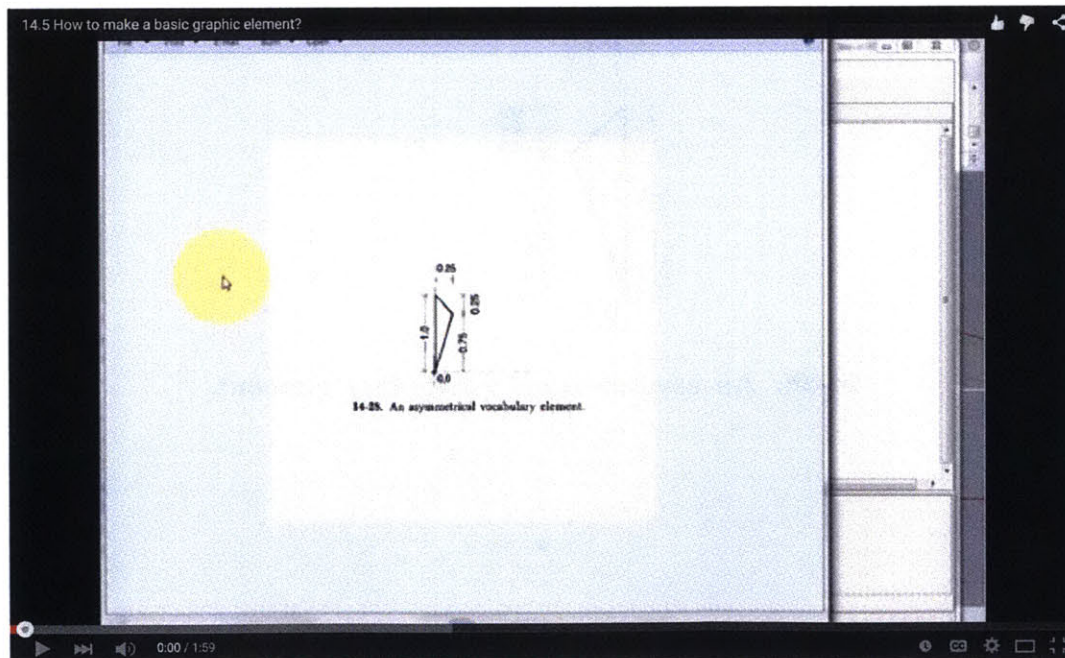
CODE

```
import rhinoscriptsyntax as rs

def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

triangle()
```

RESULT



Module 177: 14.5 Rotational

14.
TRANSFORMATIONS.

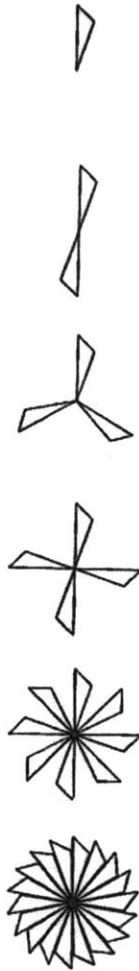
15.5.1 ROTATIONAL SYMMETRY

The following procedure has only one parameter, the Number of repetitions, and generates patterns with rotational symmetry

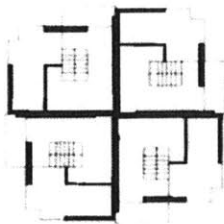
[sample codes on the right side]

Some typical output is illustrated in figure 14-29. Notice the effects of varying values for Number.

By substituting different procedures for Triangle in this code, we can produce different figures with rotational symmetry. Figure 14-30, for example, shows a floor plan that was generated this way.



14-29. Some patterns with rotational symmetry generated by Cyclic.



14-30. A floor plan with rotational symmetry of the Suntop Homes, by Frank Lloyd Wright, generated by substituting another procedure for Triangle in the code of Cyclic.

CODE

```
import rhinoscriptsyntax as rs

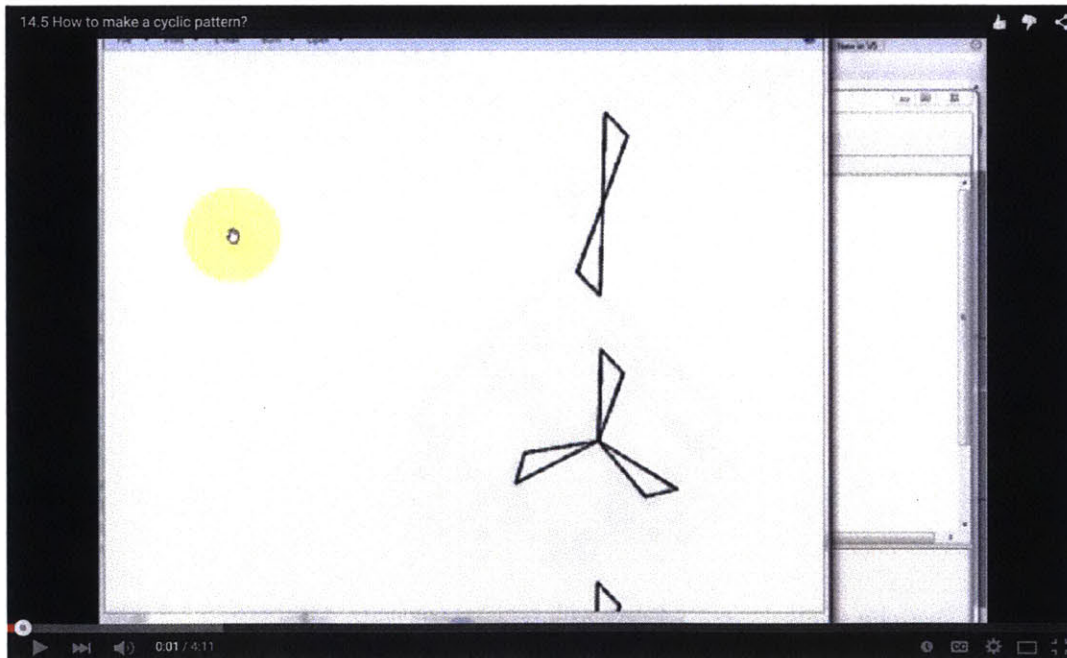
def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

def rotate(obj, angle):
    center = [0,0,0]
    result = rs.RotateObject(obj, center, angle)
    return result

def cyclic(number):
    increment = 360 / number
    angle = 0
    count = range(number)
    for i in count:
        tri = triangle()
        tri = rotate(tri, angle)
        angle = angle + increment

cyclic(290)
```

RESULT



Module 178: 14.5 Bilateral

14. TRANSFORMATIONS.

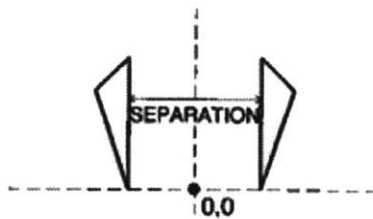
14.5.2 BILATERAL SYMMETRY

Bilateral symmetry results from the reflection of a motif across an axis (fig. 14-31). The human body has (approximate) bilateral symmetry, as do many buildings and architectural elements. The following code generates patterns with bilateral symmetry

[sample codes on the right side]

Note the introduction of a parameter Separation, controlling the distance of the motif from the axis.

By substituting different procedures for Triangle, we can produce a wide variety of compositions with bilateral symmetry. Figure 14-32 shows an architectural example.



14-31. Bilateral symmetry.



14-32. A floor plan of Montmorency Palace with bilateral symmetry, generated by substituting another procedure for Triangle in the code of Bilateral.

```

CODE
import rhinoscriptsyntax as rs

def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

def translate(obj, x, y):
    translation = [x, y, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObject(obj, center, angle)
    return result

def reflect_y(obj):
    start = [0, 1, 0]
    end = [0, -1, 0]
    ref = rs.MirrorObject(obj, start, end)
    return ref

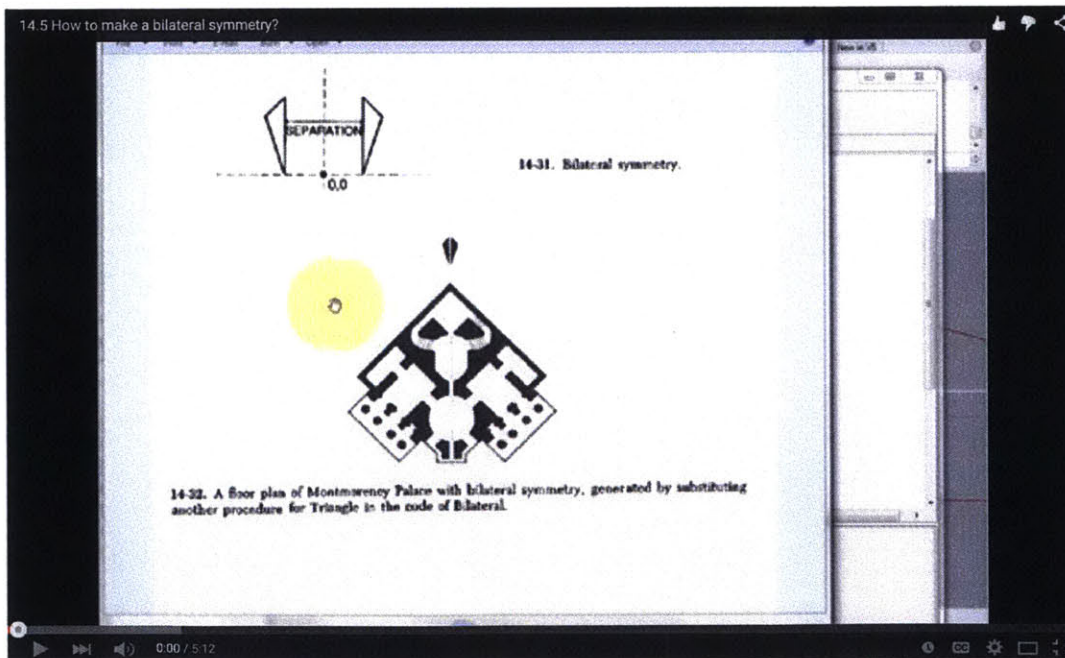
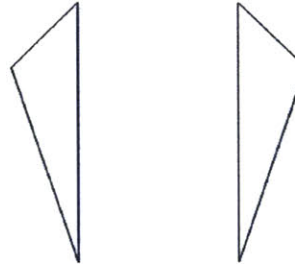
def bilateral(separation):
    tri_right = triangle()
    tri_right = translate(tri_right, separation, 0)

    tri_left = triangle()
    tri_left = translate(tri_left, separation, 0)
    tri_left = reflect_y(tri_left)

bilateral(0.3)

```

RESULT



Module 179: 14.5 Dihedral

14.
TRANSFORMATIONS.

15.5.3 DIHEDRAL SYMMETRY

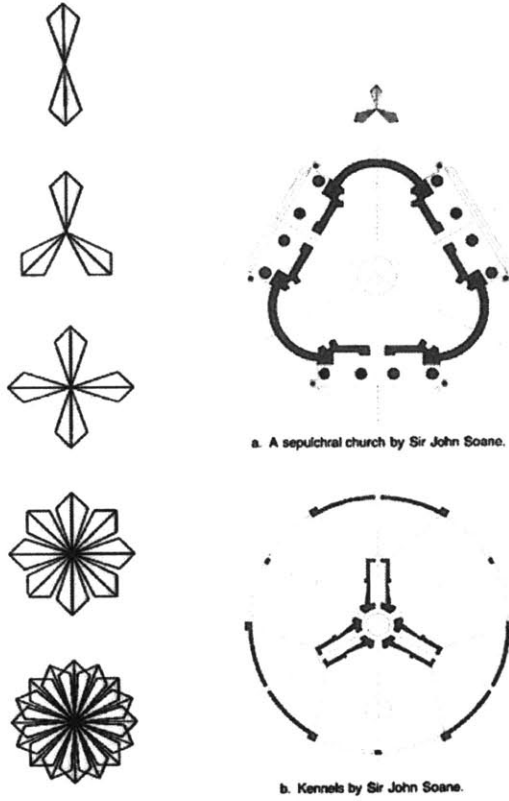
If we rotate a motif with bilateral symmetry, we can produce figures with dihedral symmetry. So a procedure to generate patterns with dihedral symmetry can be produced by modifying our procedure Cyclic to invoke Bilateral instead of Triangle

[sample codes on the right side]

Figure 14-33 shows some examples of output. Note that when the parameter symmetry.

Number is set to 1, a figure with bilateral symmetry results. This illustrates that bilateral symmetry is properly regarded as a limiting special case of dihedral symmetry.

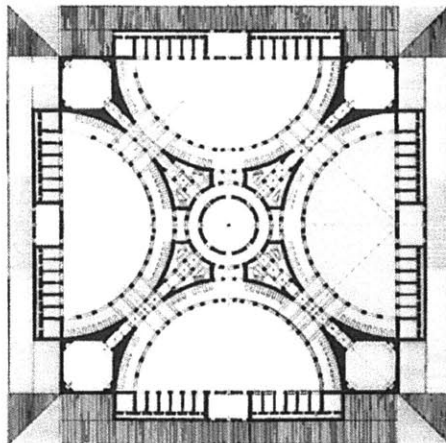
A standard method of plan composition in architecture is to begin with axes passing through a point (fig. 14-34), then to construct a plan with dihedral symmetry over this skeleton. Such plans can be produced by substituting procedures that generate appropriate motifs for Triangle in our dihedral symmetry procedure. Figure 14-35 illustrates some examples of plans produced this way.

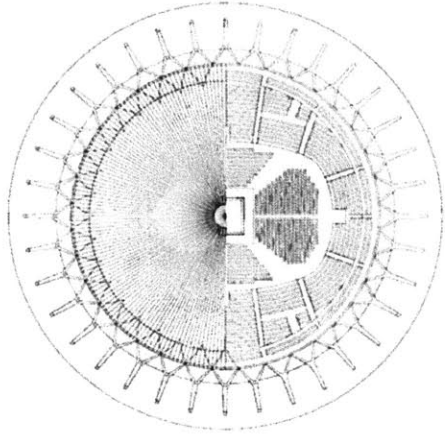


14-33. Some patterns with dihedral symmetry.

14-34. Floor plans with dihedral symmetry, generated by substituting other procedures for Triangle in the code of Dihedral.

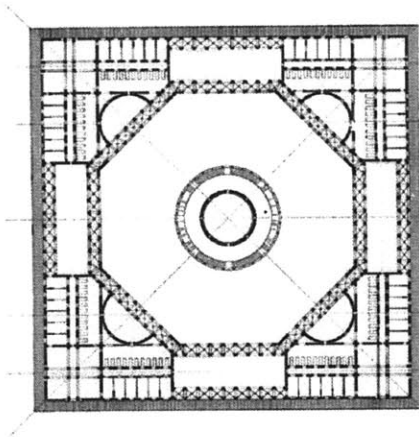
a. Project by Ledoux. (Image by Carlos Dell'Acqua.)



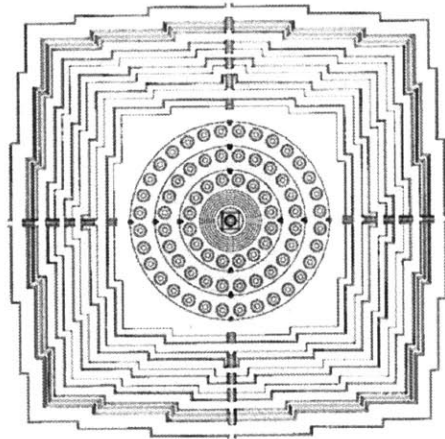


c. A sports stadium by Pier Luigi Nervi. (Image by Ehud Rapoport.)

14-35. Plan construction over axes passing through a point.



b. Project by Ledoux. (Image by Carlos Dell'Acqua.)



d. The Borobudur stupa, Java. (Image by Barbara Russell.)

CODE

```

import rhinoscriptsyntax as rs

def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

def translate(obj, x, y):
    translation = [x, y, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObjects(obj, center, angle)
    return result

def reflect_y(obj):
    start = [0, -1, 0]
    end = [0, 1, 0]
    ref = rs.MirrorObject(obj, start, end)
    return ref

def bilateral(separation):
    tri_right = triangle()
    tri_right = translate(tri_right, separation, 0)

    tri_left = triangle()
    tri_left = translate(tri_left, separation, 0)
    tri_left = reflect_y(tri_left)
    return tri_right, tri_left

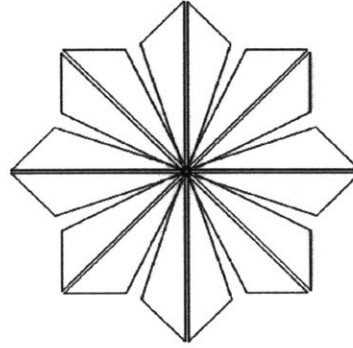
def dihedral(separation, number):
    increment = 360 / number
    angle = 0
    count = range(number)

    for i in count:
        bil = bilateral(separation)
        bil = rotate(bil, angle)
        angle = angle + increment

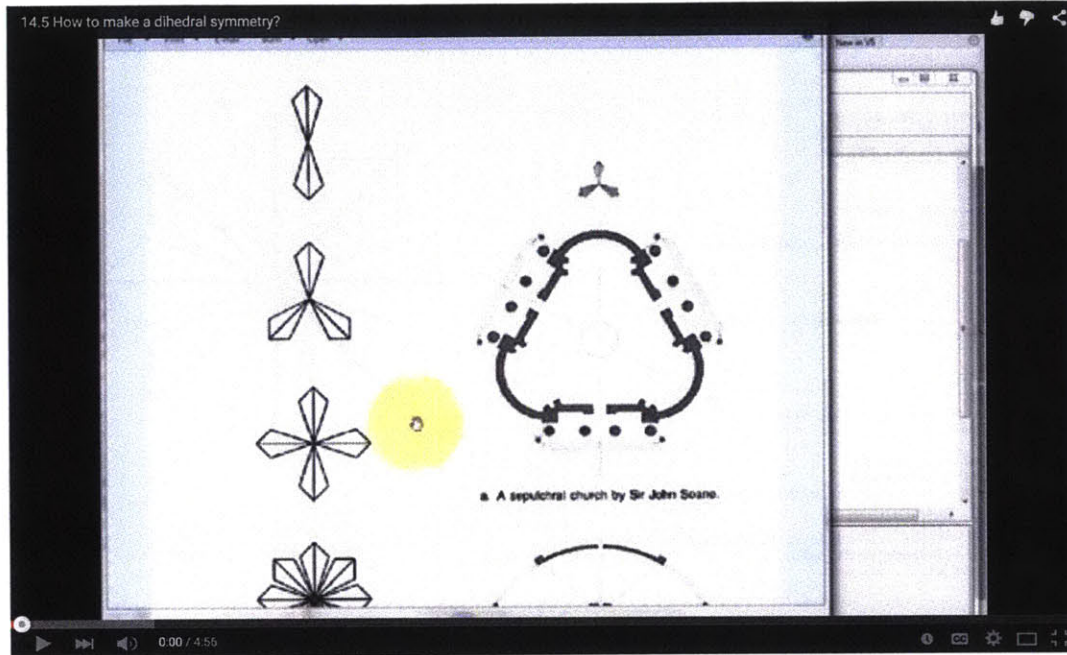
dihedral(0.01, 8)

```

RESULT



Synthetic Tutor



Module 180: Exercise 9

14.
TRANSFORMATIONS.

EXERCISES

1. The procedure Square generates a square of unit diameter, centered at the origin of the world coordinate system, with sides parallel to the coordinate axes. Draw on graph paper, in the world coordinate system, the result produced by the following sequence of code.

```
a.
def shift_sqaure():
    sqr = sqaure()
    sqr_t = translate( 0.6, 0.6, sqr )
    return sqr_t

def two_square():
    sqr = shift_square()
    sqr_r = rotate( 180, sqr)

    return sqr + sqr_r

def four_squares():
    sqr_2 = two_squares()
    sqr_rf = reflect_y( sqr_2 )
    return sqr_2 + sqr_rf

def rotate_sqaures():
    sqr_4 = four_squares()
    rotate( 45, srq_4 )

rotate_squares()

b.
def diamond( spacing, sy ):
    sqr = square()
    ty = math.sqrt( 2 ) + spacing
    sqr_s = scale( sy, sy, sqr )
    sqr_r = rotate( 45, sqr_s )
    sqr_t = translate( 0.0, ty, sqr_r )
    return sqr_t

def cyclic( spacing, sy, n ):
    increment = 360 / n
    angle = 0

    for count in range( 1, n+1 ):
        obj = diamond( spacing, sy )
        rotate( angle, obj )
        angle = angle + increment

cyclic( 0.1, 1.0, 4 )

c.
```

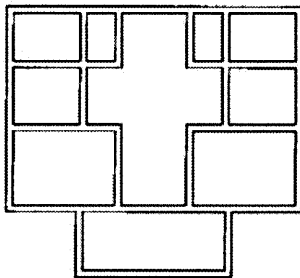

Synthetic Tutor

cyclic(0.0, 2.0, 2)

2. Write a procedure to generate a regular polygon of unit diameter, centered on the origin of the world coordinate system, with the number of sides as a parameter. Using translation, rotation, and reflection to position instances, unequal scaling to vary size and proportion, and variation in the number of sides, explore the kinds of compositions that can be produced using this motif.

Please upload your python file: No file chosen

3. Figure 14-74 shows the schema of a villa plan from Andrea Palladio's Four Books of Architecture. It is composed of just a few different types of polygons. Using procedures that generate such polygons, write a program to generate this plan.



14-74. Schema of a villa plan from Andrea Palladio's *Four Books of Architecture*.

Please upload your python file: No file chosen

4. Numerous examples of rotational, dihedral, frieze, and wallpaper patterns are shown in

- Dye, Daniel Sheets. *Chinese Lattice Designs*. New York: Dover, 1974. (Reprint of the original 1937 edition)
- Grünbaum, Branko, and G. C. Shepherd. *Tilings and Patterns*. New York: W.H. Freeman, 1987.
- Jones, Owen. *The Grammar of ornament*. New York: Van Nostrand Reinhold, 1982. (Reprint of the original 1856 edition)

Write procedures to generate some of those.

Please upload your python file: No file chosen

5. Choose a simple geometric motif write a procedure to generate it and use it to produce patterns with rotational, dihedral, frieze, and wallpaper symmetry. Experiment with effects that result from scaling the motif, varying the spacing parameters, and overlapping. Using procedural parameter, experiment with substituting different motifs.

Please upload your python file: No file chosen

6. The plan and elevation compositions of the modern architectural masters Le Corbusier, Frank Lloyd Wright, and Alvar Aalto rarely display rigid axial symmetry in the classical manner. But, on careful inspection, they can usually be found to display less obvious symmetries and carefully broken symmetries. Take a composition that interests you, and see if you can discover the underlying principles of symmetry, and the exceptions and distortions that are used to break symmetry. Using the insights that you gain from this analysis, write a concise, expressive program to generate the composition.

Please upload your python file: No file chosen

Module 181: 14.5 Frieze 1

14.
TRANSFORMATIONS.

14.5.4 FRIEZE SYMMETRY

There are just seven different kinds of frieze symmetrical patterns. They can be generated by repeated applications of translation along one axis, rotation, reflection, and glide reflection transformations. This next procedure generates the simplest of them

[sample codes on the right side]

The remaining six frieze symmetries can be generated by substituting different procedures for Triangle. We get the result shown in figure 14-37a if, for example, we substitute and we get the result shown in figure 14-37b if we substitute:

CYCLIC(2)

And we get the result shown in figure 14-37b if we substitute:

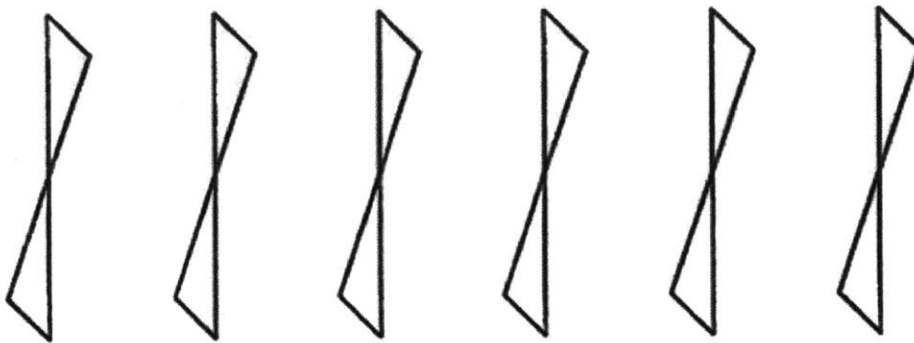
BILATERAL (0)

The repetition of an asymmetrical elevation motif along an axis to produce a continuous facade is a very common architectural motif. Compositions of this type can be produced by substituting for Triangle a procedure that generates the required motif. Figure 14-38 is an example of an elevation produced this way.

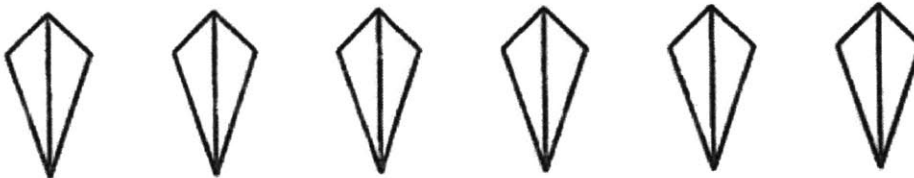
The repetition of bilaterally symmetrical elevation motifs to produce a different kind of frieze symmetry is also common. Figure 14-39 illustrates some examples of elevations produced using our Bilateral and Frieze procedures.



14-36. A simple frieze pattern produced by Frieze.

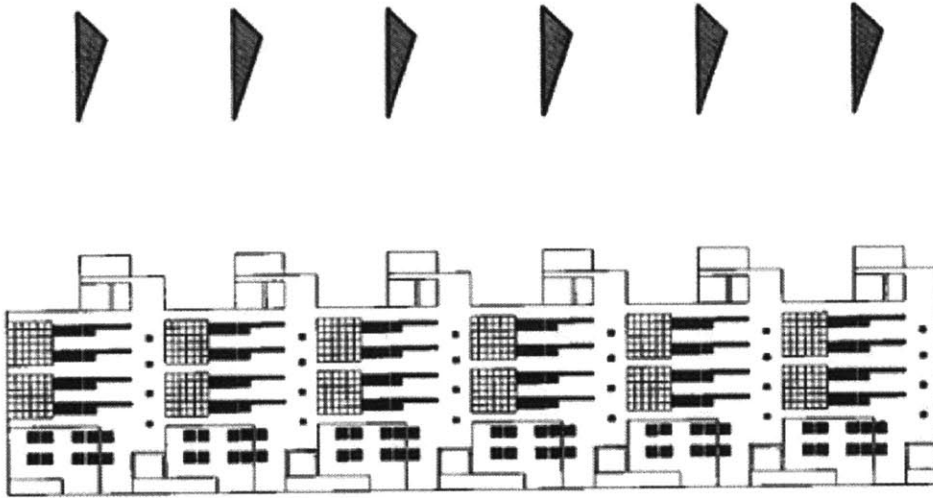


a. With a cyclically symmetrical motif.

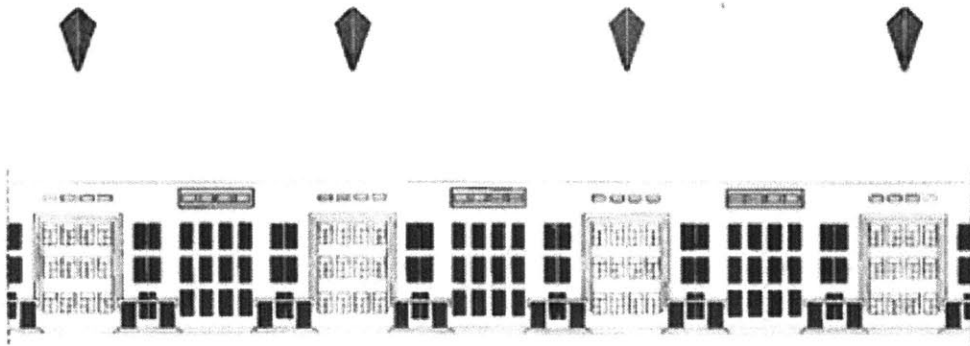


b. With a bilaterally symmetrical motif.

14-37. More complex frieze patterns.



14-38. A facade of asymmetrical elements produced by Frieze. (Richard Meier's Am Karlsbad Housing, Berlin.)



a. A housing scheme by J.P. Oud.



b. An elevation of the Allgemeine Bauerschule, Berlin, by Karl Friedrich Schinkel
(Note how end conditions are handled, so that frieze symmetry is combined with bilateral symmetry.
Conditionals are introduced into Frieze to produce the effect.)

14-39. Facades of bilaterally symmetrical elements produced by Frieze.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

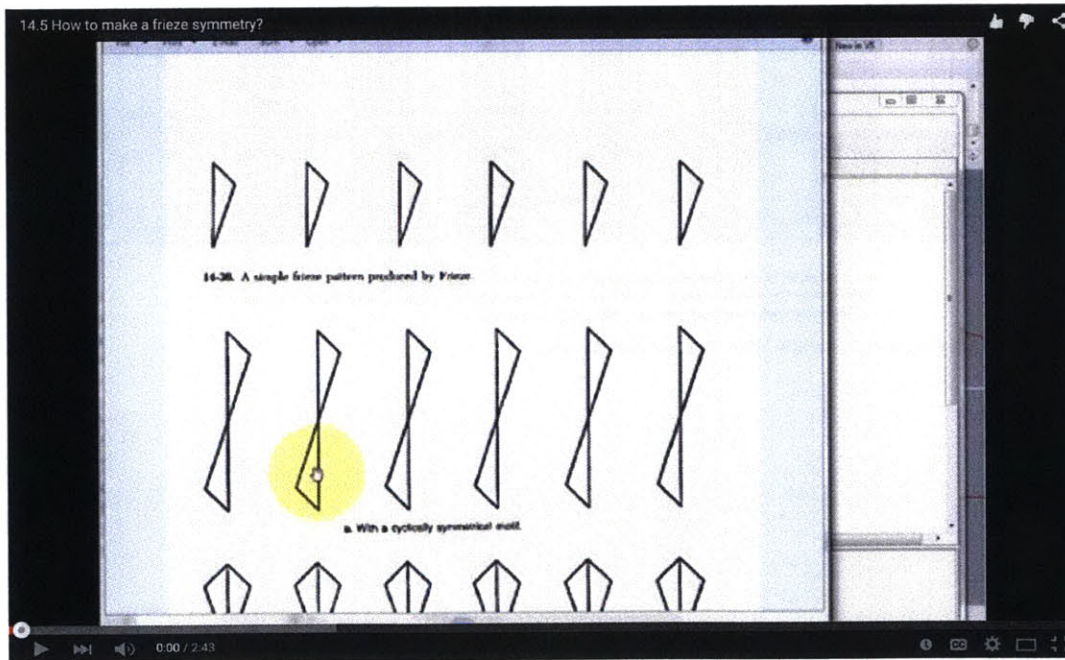
def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

def translate(obj, x, y):
    translation = [x, y, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def simpleFrieze(spacing, number):
    tx = 0
    count = range(number)
    for i in count:
        tri = triangle()
        tri = translate(tri, tx, 0)
        tx = tx + spacing

simpleFrieze(1,6)
```

RESULT



Module 182: 14.5 Frieze 2

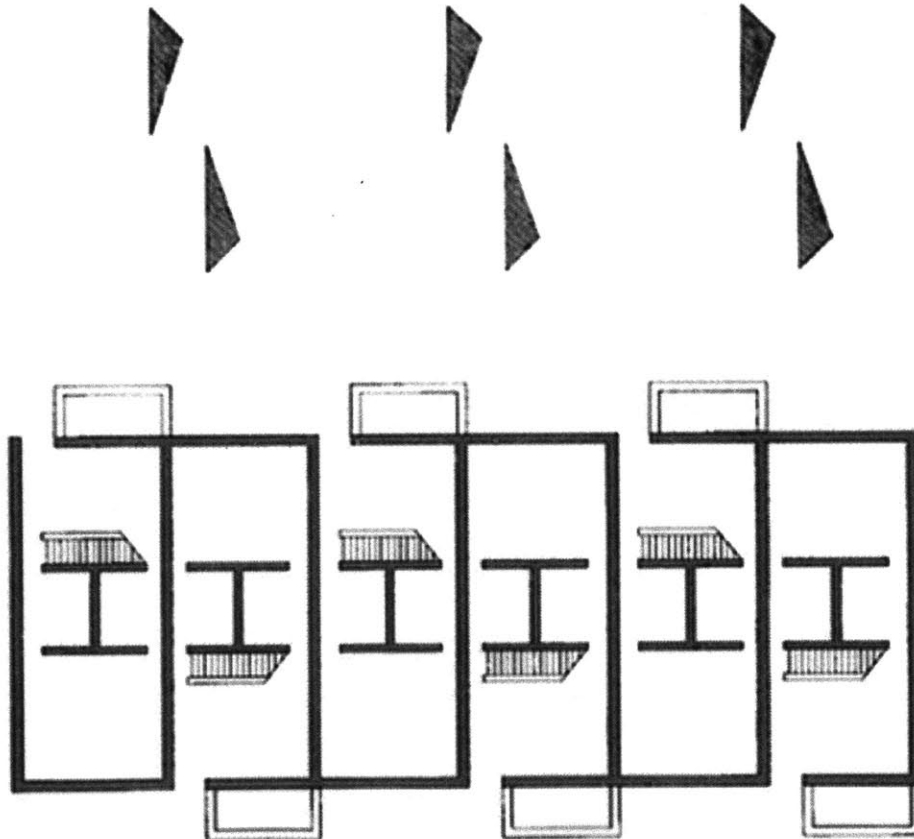
14. TRANSFORMATIONS.

14.5.4 FRIEZE SYMMETRY

In the layout of housing units, frieze patterns with glide reflection are sometimes used. The following code generates layouts of this type:

[sample codes on the right side]

The three parameters that control the spacing of the units in such layouts become critically important design variables. A layout produced by Frieze is shown in figure 14-40.



14-40. A housing layout in a frieze pattern with glide reflection. (Le Corbusier's terrace housing at Pessac).

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

def translate(obj, x, y):
    translation = [x, y, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def reflect_y(obj):
    start = [0, -1, 0]
    end = [0, 1, 0]
    ref = rs.MirrorObject(obj, start, end)
    return ref

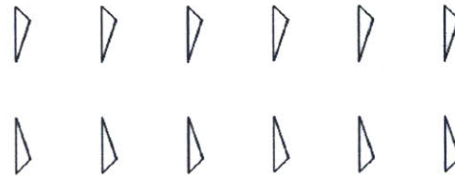
def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObject(obj, center, angle)
    return result

def frieze(x_separation, y_separation, spacing, number):
    tx=0
    count = range(number)
    for i in count:
        tri = triangle()
        tri = reflect_y(tri)
        tri = rotate(tri,180)
        tx = tx + x_separation
        ty = -y_separation
        tri = translate(tri, tx, ty)

        tri = triangle()
        tri = translate(tri, tx, 0)
        tx = tx + spacing

    frieze(1, 1, 0.5, 6)
```

RESULT



Module 183: 14.5 Wallpaper 1

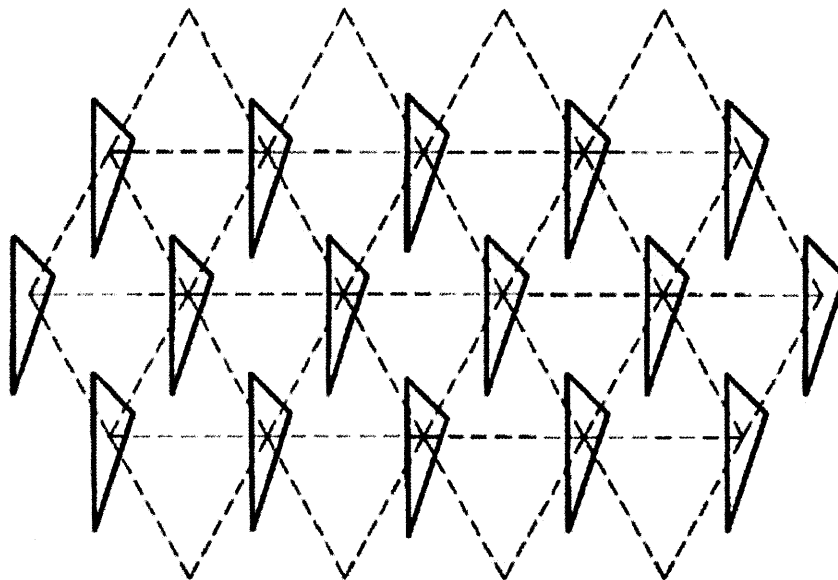
14.
TRANSFORMATIONS.

14.5.5 WALLPAPER SYMMETRY

There are just seventeen different kinds of wallpaper symmetrical patterns generated by translations in two directions, rotations, reflections, and glide reflections. Here is a procedure to generate one of them

[sample codes on the right side]

The effect is to lay out triangles in a sequence grid (fig 14-41)



14-42. Pattern generated by placement in an equilateral triangular grid.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs

def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

def translate(obj, x, y):
    translation = [x, y, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

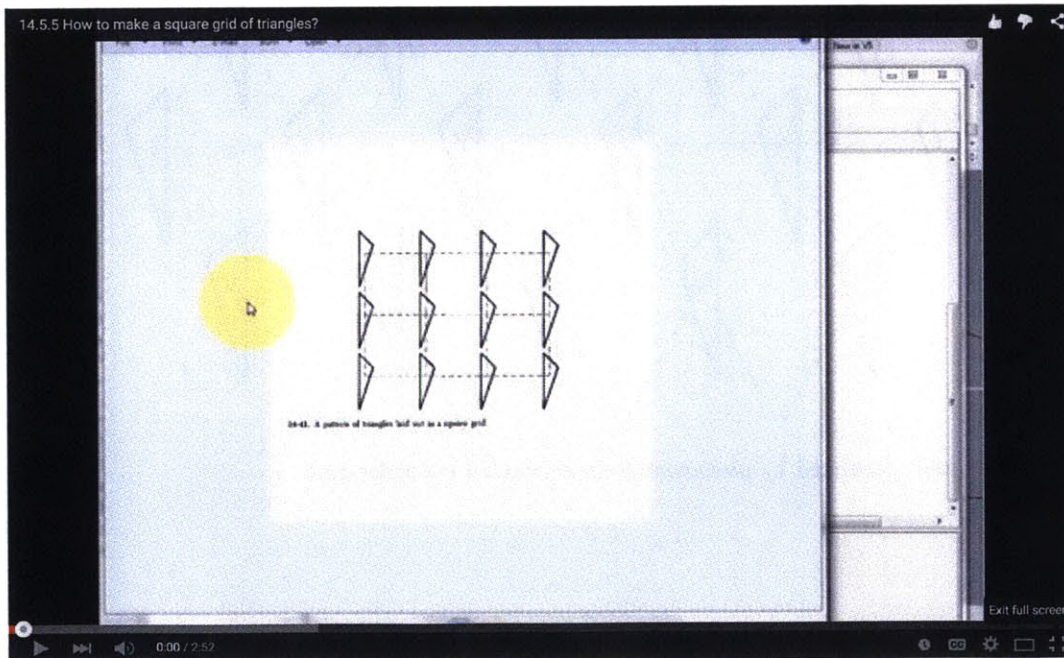
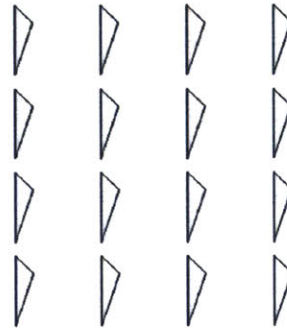
def reflect_y(obj):
    start = [0, -1, 0]
    end = [0, 1, 0]
    ref = rs.MirrorObject(obj, start, end)
    return ref

def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObject(obj, center, angle)
    return result

def square_grid(spacing, h_number, v_number):
    ty = 0
    v_count = range(v_number)
    h_count = range(h_number)
    for v in v_count:
        tx = 0
        for h in h_count:
            tri = triangle()
            tri = translate(tri, tx, ty)
            tx = tx + spacing
        ty = ty + spacing

square_grid(1.2, 4, 4)
```

RESULT



Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs
import math

def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

def translate(obj, x, y):
    translation = [x, y, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def reflect_y(obj):
    start = [0, -1, 0]
    end = [0, 1, 0]
    ref = rs.MirrorObject(obj, start, end)
    return ref

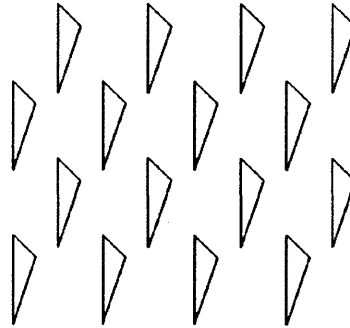
def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObject(obj, center, angle)
    return result

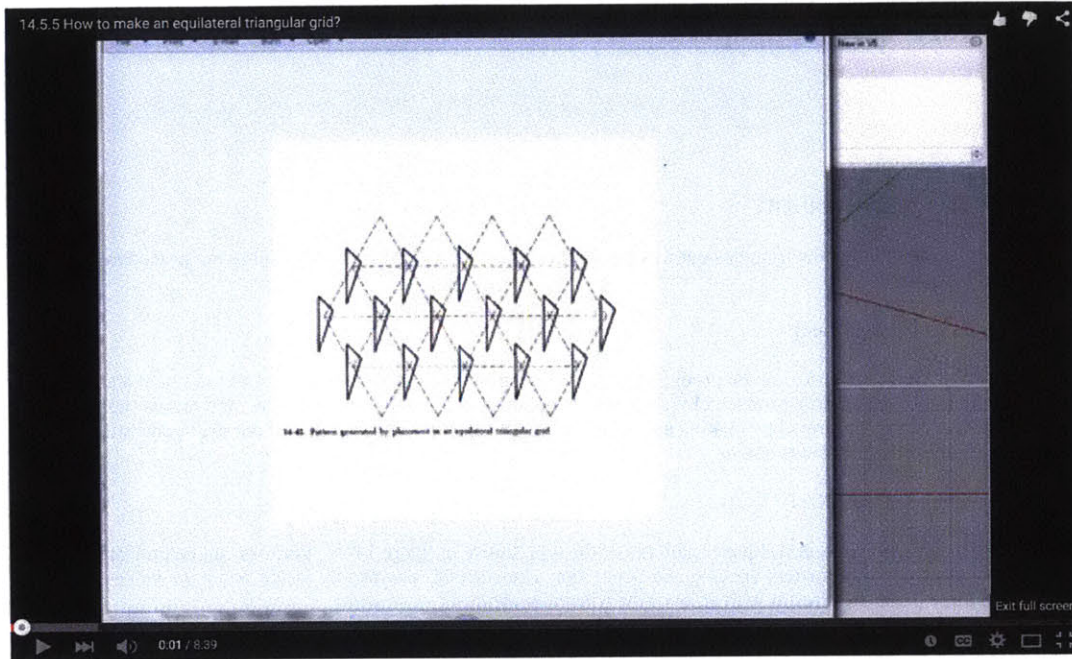
def tri_grid(spacing, h_number, v_number):
    ty = 0
    sqr_space = (spacing + spacing)
    sqr_space_half = ((spacing/2) * (spacing/2))
    h = math.sqrt(sqr_space - sqr_space_half)

    v_count = range(v_number)
    h_count = range(h_number)
    result = []
    for vc in v_count:
        tx = 0
        if vc % 2 == 1:
            tx = tx + (spacing / 2)
        for hc in h_count:
            tri = triangle()
            tri = translate(tri, tx, ty)
            result.append(tri)
            tx = tx + spacing
        ty = ty + h
    return result

tri_grid(1, 4, 4)
```

RESULT





Module 185: 14.5 Wallpaper 3

14. TRANSFORMATIONS.

14.5.5 WALLPAPER SYMMETRY

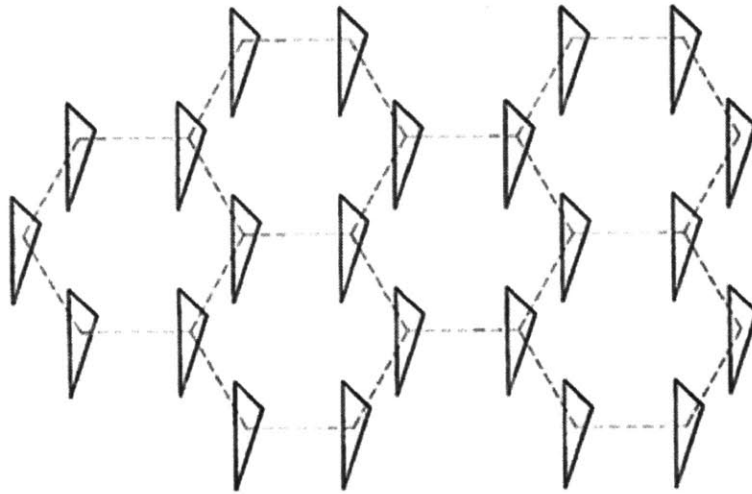
Another possibility is to lay the triangles out in a regular hexagonal grid (fig 14-43). The following procedure accomplishes this:

[sample codes on the right side]

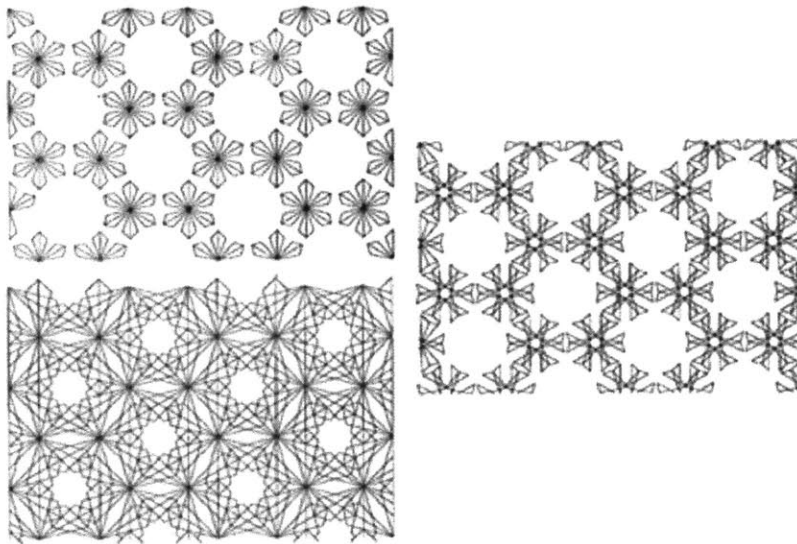
All of the wallpaper symmetries can be produced by laying out motifs with different degrees of rotational and reflective symmetry, and motifs produced by glide reflection, in square, equilateral triangular, and regular hexagonal grids. Thus they can be generated by making substitutions for Triangle in one or another of our grid-generating procedures. For example, substitution of

DIHEDRAL (SEPARATION,6)

in Hex_grid yields symmetrical wallpaper patterns of the type shown in figure 14-44. Variants can be produced by changing values for the parameters Spacing and Separation. Considerable complexity results when the values assigned to these parameters result in overlap of the triangular elements.



14-43. Pattern generated by placement in a hexagonal grid.



14-44. Variants of a symmetrical pattern with the same elementary motif and the same symmetry.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2]
    tri = rs.AddPolyline(pts)
    return tri

def translate(obj, x, y):
    translation = [x, y, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def reflect_y(obj):
    start = [0, -1, 0]
    end = [0, 1, 0]
    ref = rs.MirrorObject(obj, start, end)
    return ref

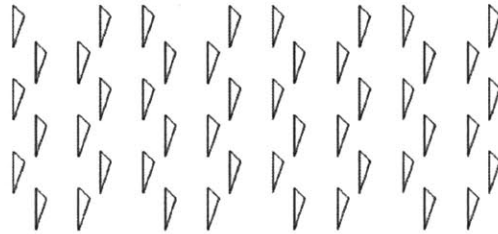
def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObject(obj, center, angle)
    return result

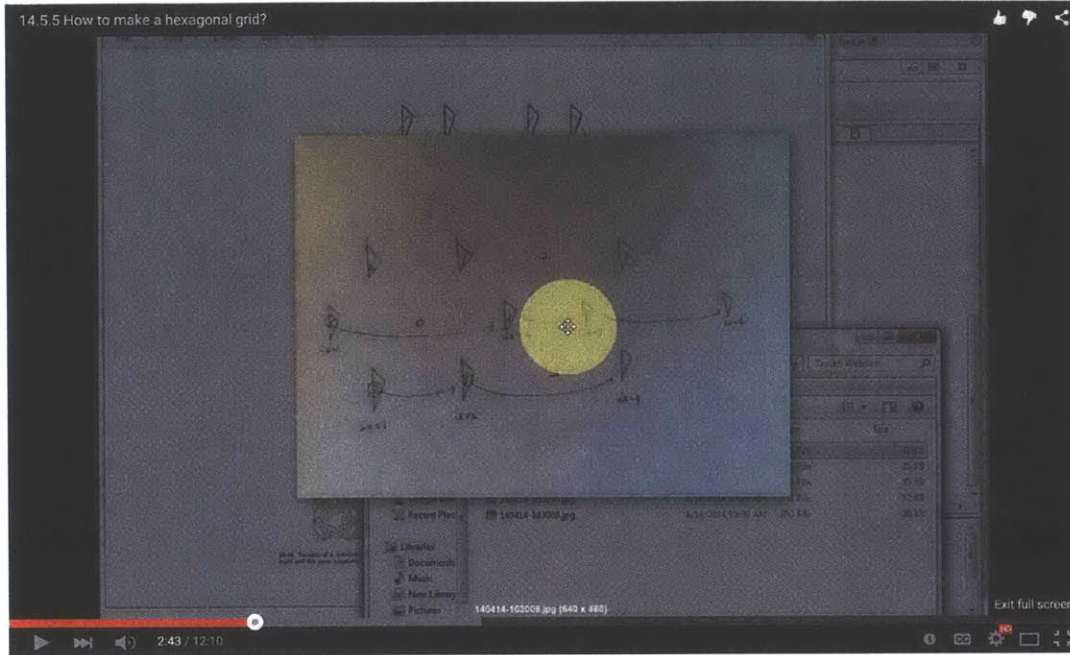
def hex_grid(spacing, h_number, v_number):
    ty = 0
    sqr_space = (spacing * spacing)
    sqr_space_half = ((spacing/2) * (spacing/2))
    h = math.sqrt(sqr_space - sqr_space_half)

    v_count = range(1, v_number+1)
    h_count = range(1, h_number+1)
    tri_list = []
    for vc in v_count:
        tx = 0
        if ((vc % 2) == 1):
            tx = tx + (spacing / 2)
        for hc in h_count:
            tri = triangle()
            tri = translate(tri, tx, ty)
            tri_list.append(tri)
            sum = (vc % 2) + (hc % 2)
            if (sum == 0 or sum == 2):
                tx = tx + spacing
            elif (sum == 1):
                tx = tx + (2 * spacing)
            else: pass
        ty = ty + h
    return tri_list

hex_grid(1, 8, 6)
```

RESULT





Module 186: 14.6 Parameters

14. TRANSFORMATIONS.

14.6 PROCEDURAL AND FUNCTIONAL PARAMETERS

Let us summarize a general approach to the generation of symmetrical patterns. You begin with a library of procedures to generate elementary motifs. You can then generate patterns with rotational symmetry by invoking such procedures from within Cyclic, patterns with bilateral symmetry by invoking them from within Bilateral, and patterns with dihedral symmetry by invoking them from within Dihedral. Regular frieze patterns can be produced by invoking procedures to generate motifs with no symmetry, rotational symmetry, and dihedral symmetry from within Frieze. In addition, you can use motifs that have glide symmetries across the X and Y axes. Finally, you can produce patterns with wallpaper symmetry by invoking procedures to generate repeating motifs from within Square_grid, Tri_grid, and Hex_grid. The repeating motifs may have no symmetry, or they may, themselves, be generated by the rotation, reflection, and glide reflection of more elementary motifs.

By manipulating the scale of the repeating motif in a pattern, and the various parameters that control spacing of instances of the motif, you can vary proportions and figure/ground relationships. Very complex effects can be produced by scaling motifs and setting spacing parameters so that overlaps occur.

A symmetry group is a type of spatial organization that, as we have seen, can be generated by an appropriate procedure. Different instances can be produced not only by varying the parameters, but also by using the procedure to arrange different elementary motifs in the same type of spatial organization.

It is cumbersome, however, to change the code of a symmetry-generating procedure whenever we want it to repeat a different elementary motif. Python allows use of procedural parameters and functional parameters (in addition to value and variable parameters with which we are already familiar), and the use of procedural parameters resolves this difficulty.

Procedures and functions can be passed as parameters to other procedures and functions. In the formal parameter list, procedure and function parameters have the same syntax as procedure and function headings. Whenever a formal procedure or function is referenced, the corresponding actual parameter is activated. In the following symmetry-generating program, for example, Motif is a procedural parameter. This allows procedures that draw different figures to be passed in, so that symmetrical patterns of different figures can be drawn.

[sample codes on the right side]

Symmetrical patterns illustrate the use of procedural parameters particularly clearly, but any graphic procedure may be parameterized in this way. We might, for example, pass different line types into a procedure to draw a polygon, different window types into a procedure to fenestrate an elevation, and so on. This generalizes the idea of type of graphic motif that we have used until now the essence of the type is simply a spatial organization. Instances might vary not only in shape and position, but also in the nature of their components.

Some Python systems do not support procedural and functional parameters. You should check the documentation of the particular system that you are using.

CODE

```

import rhinoscriptsyntax as rs
import math

def triangle():
    p0 = [0, 0, 0]
    p1 = [0.25, 0.75, 0]
    p2 = [0, 1, 0]
    pts = [p0, p1, p2, p0]
    tri = rs.AddPolyline(pts)
    return tri

def translate(obj, x, y):
    translation = [x, y, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def reflect_y(obj):
    start = [0, -1, 0]
    end = [0, 1, 0]
    ref = rs.MirrorObject(obj, start, end)
    return ref

def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObject(obj, center, angle)
    return result

def tri_grid(spacing, h_number, v_number):
    ty = 0
    sqr_space = (spacing * spacing)
    sqr_space_half = ((spacing/2) * (spacing/2))
    h = math.sqrt(sqr_space - sqr_space_half)

    v_count = range(v_number)
    h_count = range(h_number)
    result = []
    for vc in v_count:
        tx = 0
        if vc % 2 == 1:
            tx = tx + (spacing / 2)
        for hc in h_count:
            tri = triangle()
            tri = translate(tri, tx, ty)
            result.append(tri)
            tx = tx + spacing
        ty = ty + h
    return result

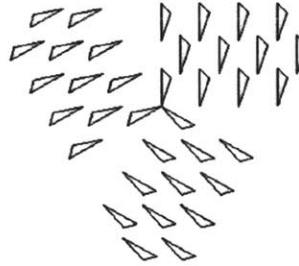
def rotate(obj,angle):
    center = [0,0,0]
    result = rs.RotateObjects(obj, center, angle)
    return result

def cyclic(number, motif):
    increment = 360 / number
    angle = 0
    repeat = range(1, number+1)
    for count in repeat:
        mot = motif(1,4,3)
        mot = rotate(mot,angle)
        angle = angle+increment

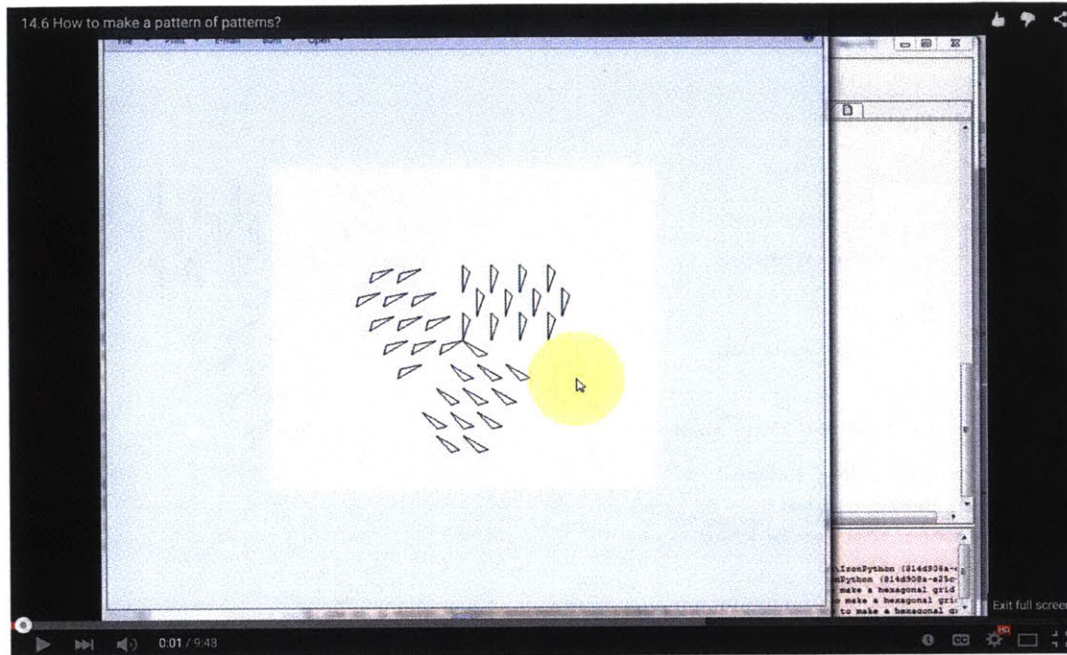
cyclic(3, tri_grid)

```

RESULT



Synthetic Tutor



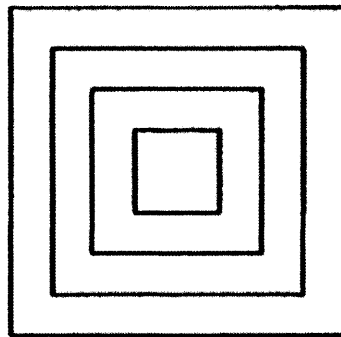
Module 187: 14.7 Repetition 1

14.
TRANSFORMATIONS.

14.7 REPETITION OF SCALE TRANSFORMATIONS

So far in this chapter, we have considered the repetition of rotations, reflections, and translations to generate regular figures. We can also use our scale operator this way. This next procedure, for example, generates the pattern of nested squares shown in figure 14-45

[sample codes on the right side]



14-45. A pattern of nested squares.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs
import math

def scale(obj,x,y):
    origin = [0,0,0]
    scale = [x,y,1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

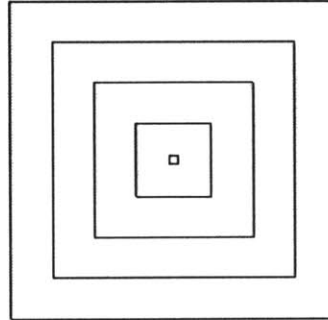
def rotate(obj,angle):
    center = [0,0,0]
    result=rs.RotateObjects(obj, center, angle)
    return result

def square():
    p1 = [-0.5, -0.5, 0]
    p2 = [0.5, -0.5, 0]
    p3 = [0.5, 0.5, 0]
    p4 = [-0.5, 0.5, 0]
    pts = [p1,p2,p3,p4,p1]
    square = rs.AddPolyline(pts)
    return square

def nest_sqaures(increment, number):
    side = 1
    repeat = range(1,number+1)
    for count in repeat:
        sqr = square()
        sqr = scale(sqr, side, side)
        side = side - increment

nest_sqaures(10, 5)
```

RESULT



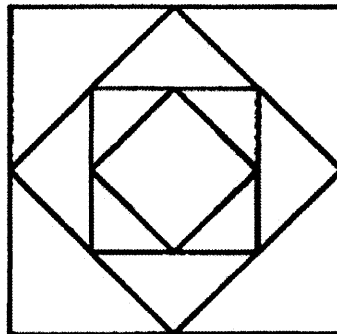
Module 188: 14.7 Repetition 2

14.
TRANSFORMATIONS.

14.7 REPETITION OF SCALE TRANSFORMATIONS

You will recall from earlier chapters the pattern of rotating and diminishing squares shown in figure 14-46. Now that we have scale and rotation operators at our disposal, we can express the procedure to generate it thus

[sample codes on the right side]



14-46. A pattern of rotating and diminishing squares.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def scale(obj,x,y):
    origin = [0,0,0]
    scale = [x,y,1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

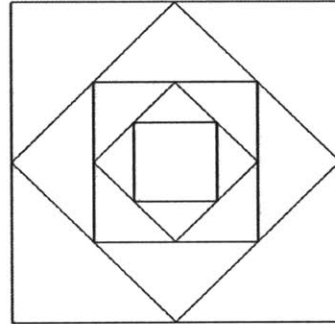
def rotate(obj,angle):
    center = [0,0,0]
    result=rs.RotateObjects(obj, center, angle)
    return result

def square():
    p1 = [-0.5, -0.5, 0]
    p2 = [0.5, -0.5, 0]
    p3 = [0.5, 0.5, 0]
    p4 = [-0.5, 0.5, 0]
    pts = [p1,p2,p3,p4,p1]
    square = rs.AddPolyline(pts)
    return square

def rotating_nest_sqaures(number):
    side = 1
    angle = 0
    repeat = range(1,number+1)
    for count in repeat:
        sqr = square()
        sqr = scale(sqr, side, side)
        sqr = rotate(sqr, angle)
        side = math.sqrt(side * side / 2)
        angle = angle + 90

rotating_nest_sqaures(5)
```

RESULT



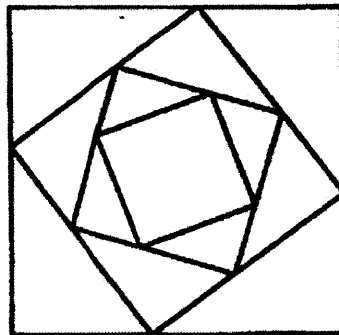
Module 189: 14.7 Repetition 3

14.
TRANSFORMATIONS.

14.7 REPETITION OF SCALE TRANSFORMATIONS

A variant, in which 3, 4, 5 rational triangles instead of root two triangles are generated, is shown in figure 14-47.
The procedure to generate this is

[sample codes on the right side]



14-47. A variant pattern of rotating and diminishing squares, in which 3, 4, 5 rational triangles appear.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def scale(obj,x,y):
    origin = [0,0,0]
    scale = [x,y,1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

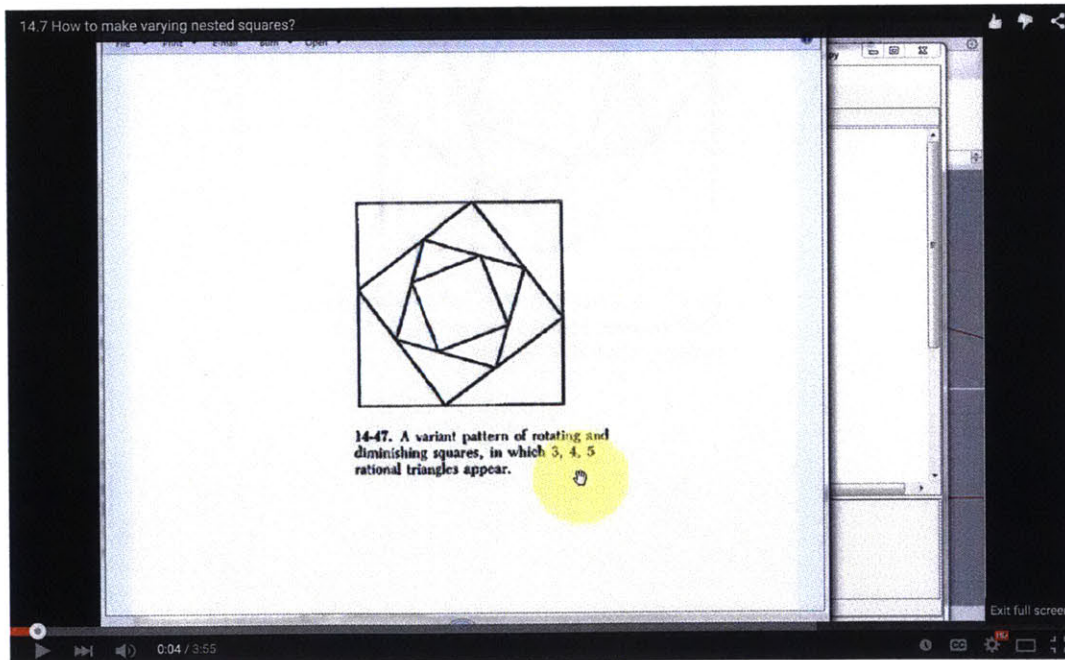
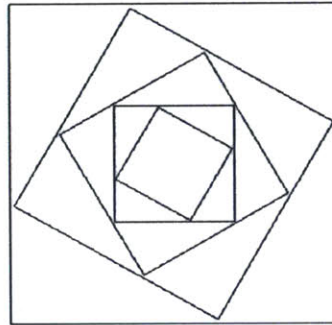
def rotate(obj,angle):
    center = [0,0,0]
    result=rs.RotateObjects(obj, center, angle)
    return result

def square():
    p1 = [-0.5, -0.5, 0]
    p2 = [0.5, -0.5, 0]
    p3 = [0.5, 0.5, 0]
    p4 = [-0.5, 0.5, 0]
    pts = [p1, p2, p3, p4, p1]
    square = rs.AddPolyline(pts)
    return square

def variant_rotating_nest_squares(number):
    side = 1
    angle = 0
    repeat = range(1, number+1)
    for count in repeat:
        sqr = square()
        sqr = scale(sqr, side, side)
        sqr = rotate(sqr, angle)
        side = side * 5 / 7.0
        angle = angle + 60

variant_rotating_nest_squares(5)
```

RESULT



Module 190: 14.7 Repetition 4

14. TRANSFORMATIONS.

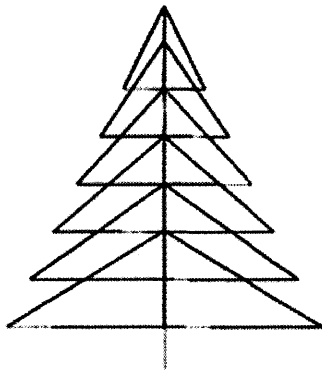
14.7 REPETITION OF SCALE TRANSFORMATIONS

The -tree- composed of triangles (fig. 14-48) is a motif taken from the notebooks of Paul Klee. The repeating element is an equilateral triangle to which translation and unequal scale transformations are applied. The procedure for this is

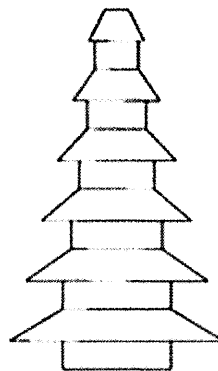
[sample codes on the right side]

Different combinations of values for the parameters controlling the translation increment and the two scale increments produce variants with different characters.

Some architectural compositions are constructed according to a similar logic. The pagoda in figure 14-49 was generated this way. Both the tree and the pagoda can, in fact, be produced by the same procedure if we introduce a procedural parameter. Much architectural form can be understood, in the same way, as the result of abstracting a spatial organization from nature and substituting a new motif within that organization.



14-48. A tree motif (from the *Notebooks of Paul Klee*) generated by translation and unequal scaling of an equilateral triangle.



14-49. A pagoda motif.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs

def scale(obj,x,y):
    origin = [0,0,0]
    scale = [x,y,1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

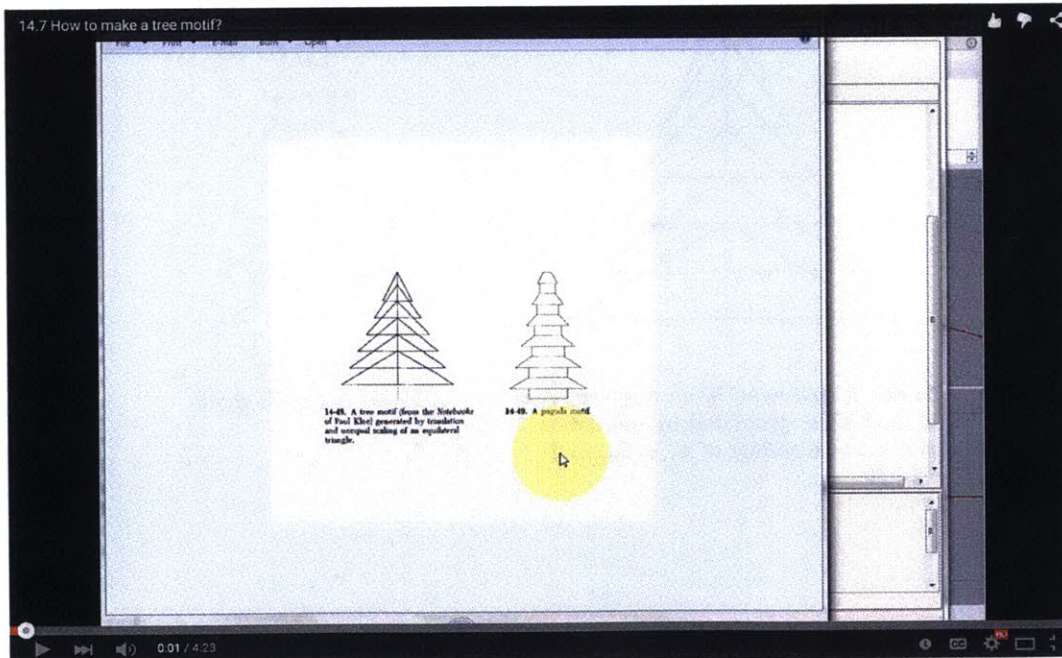
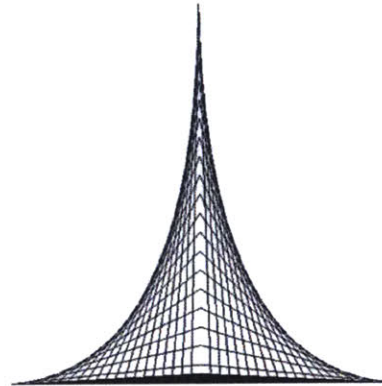
def triangle():
    p0 = [-100,0,0]
    p1 = [100,0,0]
    p2 = [0,20,0]
    pts=[p0,p1,p2,p0]
    trg = rs.AddPolyline(pts)
    return trg

def translate(obj, tx, ty):
    translation = [tx, ty, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def tree(spacing, sx, sy, sx_inc, sy_inc, number):
    ty = 0
    count = range(1, number+1)
    for i in count:
        sqr = triangle()
        sqr = scale(sqr, sx, sy)
        sqr = translate(sqr, 0, ty)
        sx = sx - sx_inc
        sy = sy + sy_inc
        ty = ty + spacing

tree(1, 4,2, 0.2,2, 20)
```

RESULT



Module 191: 14.8 Arcs 1

14.
TRANSFORMATIONS.

14.8 MOTIFS CONSTRUCTED FROM ARCS

Many traditional architectural and decorative motifs are constructed by using compasses to strike arcs of specified radius about a specified center. In computer graphics, it is often convenient to strike the arc about the origin of the coordinate system, then to use transformations to locate the motif in the composition.

An arch can be constructed from arcs in this way. If only one arc is used in the construction, then there are three basic types (fig. 14-50) the flat arch, in which the radius is greater than half the span the round arch, in which the radius is exactly half the span and the bulging arch, in which the center point is higher than the springing point.

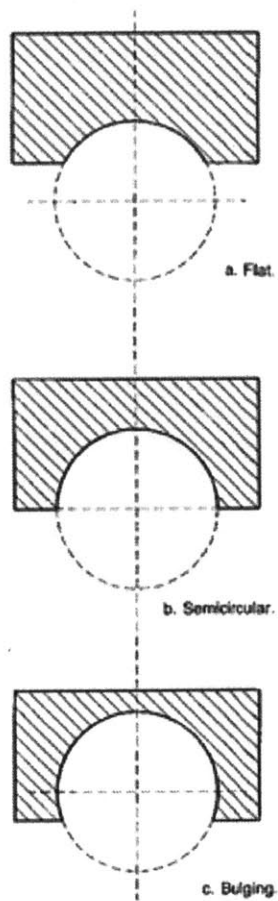
There are many reasonable ways to parameterize arches, depending on how an architect might want to fit them into a composition. With a semicircular arch, we know that the bounding rectangle will have a proportion of 2:1. Thus we probably would choose to specify the width of the bounding rectangle (that is, the span of the arch) as the single parameter (fig. 14-51).

Using our transformation system, it is convenient to center the arch at the origin and set the span equal to one world unit. It can then be scaled to the appropriate size and translated to the desired position with the scale and translate transformation procedures.

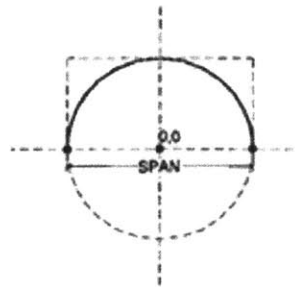
If we assume that the arch can either be flat, semicircular, or bulging, then span is disconnected from height to yield an additional parameter (fig. 14-52).

Since the structural logic of a uniformly loaded arch dictates bilateral symmetry of form, we need only draw half an arch, then reflect it to get the other half. Here, then, is a generalized arch procedure based on this principle, with a single parameter, Height, that represents the ratio of height to span. (The additional parameter $N_segments$ is required by the Arc procedure, but does not concern us here.)

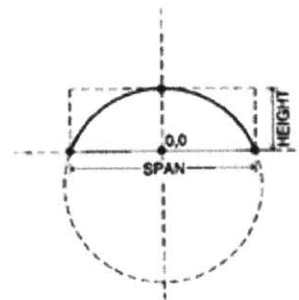
[sample codes on the right side]



14-50. Basic types of circular arches.



14-51. The parameterization of a semicircular arch.



14-52. The parameterization of an arch with variable height and span.

```

CODE
import rhinoscriptsyntax as rs
import math

def translate(obj, tx, ty):
    translation = [tx, ty, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def arc(xc, yc, radius, start_angle, stop_angle,
       n_segments):
    radians = 0.01745
    degrees = (stop_angle - start_angle) / n_segments * radians
    theta = start_angle * radians
    repeat = range(1, n_segments+2)
    result = []
    for sides in repeat:
        y = yc + radius * math.sin(theta)
        x = xc + radius * math.cos(theta)
        if (sides == 1):
            p0 = [x, y, 0]
        else:
            p1 = [x, y, 0]
            ln = rs.AddLine(p0, p1)
            result.append(ln)
            p0 = p1
        theta = theta + degrees
    return result

def half_arch(height, n_segments):
    half_span = 0.5
    radians = 0.01745
    half_span_square = half_span * half_span
    height_saure = height * height
    radius = (half_span_square + height_saure) / (2 * height)
    xc = 0
    yc = height - radius
    stop_angle = 90
    if (height == radius):
        start_angle = 0
    else:
        start_radians = math.atan((radius - height) / half_span)
        start_angle = math.degrees(start_radians)
    result = arc(xc, yc, radius, start_angle, stop_angle, n_segments)
    return result

def reflect(obj):
    start = [0, -1, 0]
    end = [0, 1, 0]
    ref = rs.MirrorObject(obj, start, end, True)
    return ref

def arch(height, n_segments):
    h_arch = half_arch(height, n_segments)
    reflect(h_arch)

arch(0.15, 10)

```

RESULT



Module 192: 14.8 Arcs 2

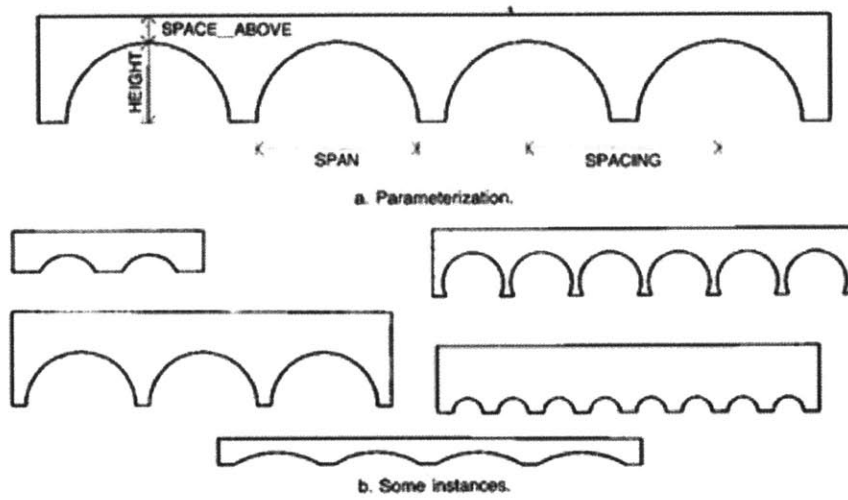
14.
TRANSFORMATIONS.

14.8 MOTIFS CONSTRUCTED FROM ARCS

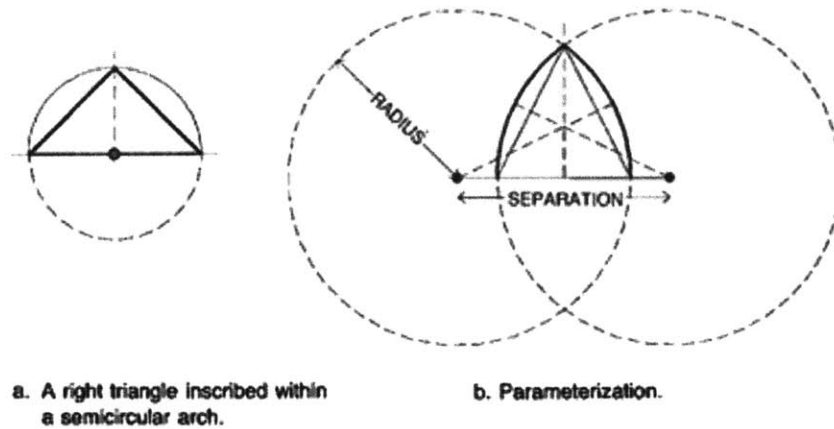
A closely related architectural motif is the arcade. Here the architect has three additional choices to make the number of arches the spacing between arches and the space above the arch (fig. 14-53a). Figure 14-53b shows some variants of the type. They were generated by the following program:

[sample codes on the right side]

Now, a right triangle may be inscribed within a semicircular arch (fig. 14-54a). This suggests another way to generalize and introduce additional degrees of design freedom we can let the height of the triangle become greater than half the base, and we can let the radii of the two half arcs be offset (fig. 14-54b), so that we obtain five degrees of design freedom. This new type is the pointed arch, used in gothic architecture.



14-53. An arcade motif.



14-54. The pointed arch.

Synthetic Tutor

```
CODE
import rhinoscriptsyntax as rs
import math

def translate(obj, tx, ty):
    translation = [tx, ty, 0]
    trans = rs.MoveObjects(obj, translation)
    return trans

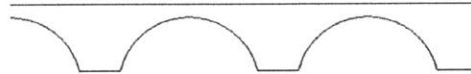
def arc(xc, yc, radius, start_angle, stop_angle, n_segments):
    radians = 0.01745
    degrees = (stop_angle - start_angle) / n_segments * radians
    theta = start_angle * radians
    repeat = range(1, n_segments+2)
    result = []
    for sides in repeat:
        y = yc + radius * math.sin(theta)
        x = xc + radius * math.cos(theta)
        if (sides == 1):
            p0 = [x, y, 0]
        else:
            p1 = [x, y, 0]
            ln = rs.AddLine(p0, p1)
            result.append(ln)
            p0 = p1
        theta = theta + degrees
    return result

def half_arch(height, n_segments):
    half_span = 0.5
    radians = 0.01745
    half_span_square = half_span * half_span
    height_saure = height * height
    radius = (half_span_square + height_saure) / (2 * height)
    xc = 0
    yc = height - radius
    stop_angle = 90
    if (height == radius):
        start_angle = 0
    else:
        start_radians = math.atan((radius - height) / half_span)
        start_angle = math.degrees(start_radians)
    result = arc(xc, yc, radius, start_angle, stop_angle, n_segments)
    return result

def reflect(obj):
    start = [0, -1, 0]
    end = [0, 1, 0]
    ref = rs.MirrorObjects(obj, start, end, True)
    result = obj + ref
    return result

def arch(height, n_segments):
    h_arch = half_arch(height, n_segments)
    result = reflect(h_arch)
    return result
```

RESULT



```

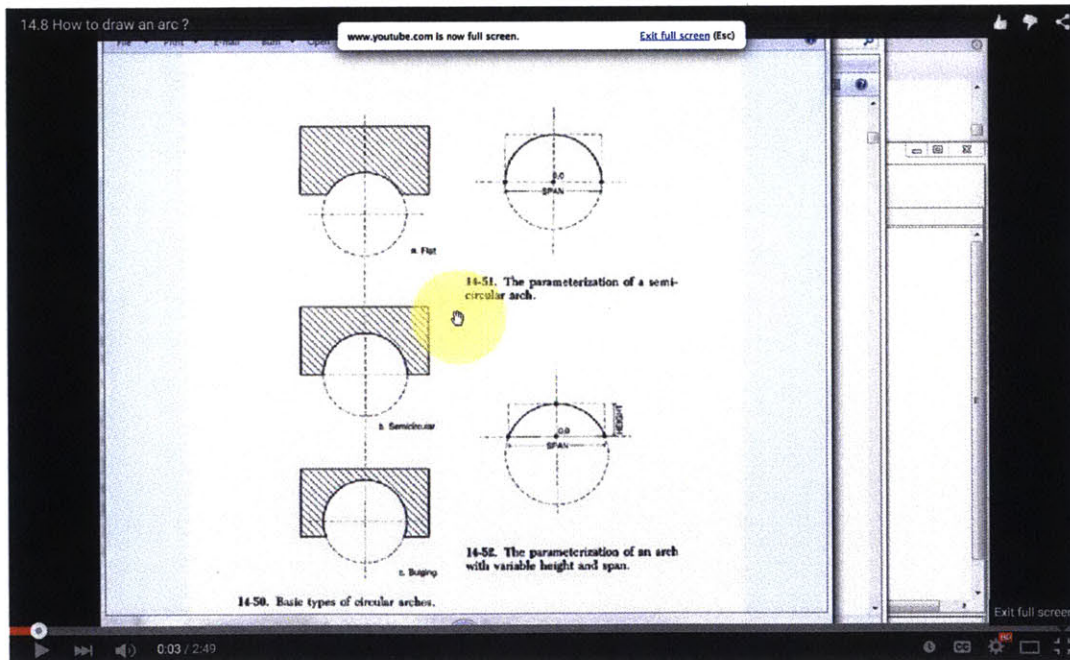
def half_arcade(height, spacing, space_above, n_arches, n_segments):
    span = 1
    half_span = span / 2
    if ((n_arches % 2) == 1):
        half_arch(height, n_segments)
        p0 = [half_span, 0, 0]
        x = spacing + half_span
        p1 = [x, 0, 0]
        rs.AddLine(p0, p1)
        tx = span + spacing
    else:
        p0 = [0, 0, 0]
        x = spacing / 2
        p1 = [x, 0, 0]
        tx = (spacing / 2) + half_span

    n = int(n_arches/2)
    count = range(1, n+1)
    for i in count:
        archID = arch(height, n_segments)
        translate(archID, tx, 0)
        x = tx + half_span
        p0 = [x, 0, 0]
        x = x + spacing
        p1 = [x, 0, 0]
        rs.AddLine(p0, p1)
        tx = tx + span + spacing

    y = height + space_above
    p0 = [0, y, 0]
    p1 = [x, y, 0]
    p2 = [x, 0, 0]
    pts = [p0, p1, p2]
    rs.AddPolyline(pts)

half_arcade(0.4, 0.3, 0.1, 5, 10)

```



Module 193: 14.8 Arcs 3

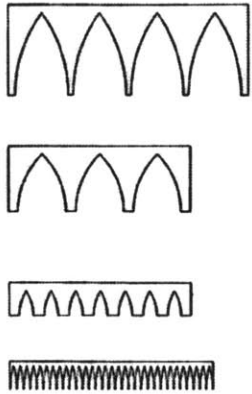
14.
TRANSFORMATIONS.

14.8 MOTIFS CONSTRUCTED FROM ARCS

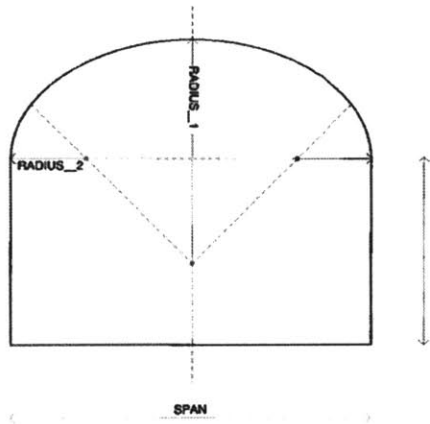
Figure 14-55 illustrates some arcades of pointed arches, such as we might find in the nave of a gothic cathedral. Notice how the parameterization of the pointed arch allows achievement of the soaring verticality that is characteristic of gothic architecture. Compare this with the horizontal rhythms of semicircular arcades, which we considered earlier. Another possibility is to base the arch on the form of the oval. Such arches are called ogees. The oval is visually almost indistinguishable from the ellipse, but is constructed from circular arcs of different radii. There are two standard subtypes one is drawn from three centers (fig. 14-56) and the other is drawn from five centers (fig. 14-57). A three-centered ogee is generated.

[sample codes on the right side]

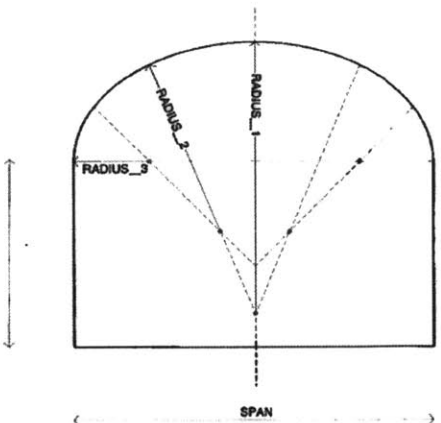
Once an arch has been constructed, another classic architectural problem often arises. How can we subdivide it by window panes? More generally, what principles can we adopt for subdividing the circle?



14-55. Arcades of pointed arches.



14-56. The three-centered ogee.



14-57. The five-centered ogee.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def arc(xc, yc, radius, start_angle, stop_angle, n_segments):
    radians = 0.01745
    degrees = (stop_angle - start_angle) / n_segments * radians
    theta = start_angle * radians
    repeat = range(1, n_segments+2)
    result = []
    for sides in repeat:
        y = yc + radius * math.sin(theta)
        x = xc + radius * math.cos(theta)
        if (sides == 1):
            p0 = [x, y, 0]
        else:
            p1 = [x, y, 0]
            ln = rs.AddLine(p0, p1)
            result.append(ln)
            p0 = p1
        theta = theta + degrees
    return result

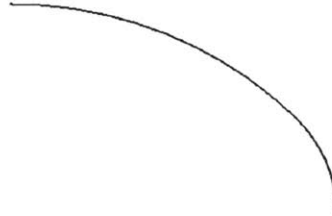
def oval(a, b, n_segments):
    # a is radius of major axis
    # b is radius of minor axis
    d = math.sqrt(2 * (a-b) * (a-b)) / 2

    # construct the first arc
    xc1 = a - b + d
    yc1 = 0
    radius2 = b - d
    start_angle = 0
    stop_angle = 45
    arc(xc1, yc1, radius2, start_angle, stop_angle, n_segments)

    # construct the second arc
    xc2 = 0
    yc2 = -xc1
    radius1 = radius2 + math.sqrt(2 * xc1 * xc1)
    start_angle = 45
    stop_angle = 90
    arc(xc2, yc2, radius1, start_angle, stop_angle, n_segments)

oval(0.6, 0.4, 10)
```

RESULT



Module 194: 14.8 Arcs 4

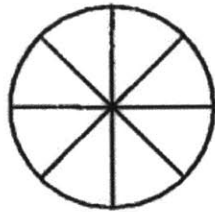
14.
TRANSFORMATIONS.

14.8 MOTIFS CONSTRUCTED FROM ARCS

One commonly used principle is subdivision by regularly spaced radial lines (fig. 14-58a). Another is subdivision by concentric circles (fig. 14-58b). These can be combined to yield a radio concentric pattern (fig. 14-58c). Yet another possibility is subdivision by parallel straight lines (fig. 14-58d).

One parameter for any of these types of constructions will obviously be the number of subdivisions. The following procedure, for example, radically divides a specified circle into a specified number of equal parts

[sample codes on the right side]



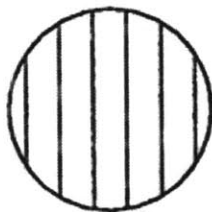
a. Radial.



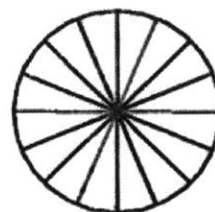
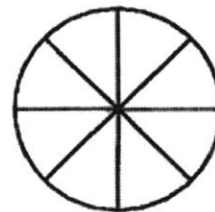
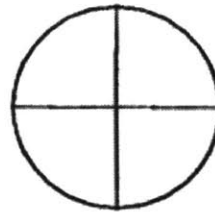
b. Concentric.



c. Radiocentric.



d. Parallel.



14-58. Methods for subdividing circles.

14-59. Radial division of a circle into equal parts.

CODE

```
import rhinoscriptsyntax as rs

def circle():
    center = [0, 0, 0]
    r = 10
    obj = rs.AddCircle(center, r)
    return obj

def line():
    start = [0, 0, 0]
    end = [1, 0, 0]
    obj = rs.AddLine(start, end)
    return obj

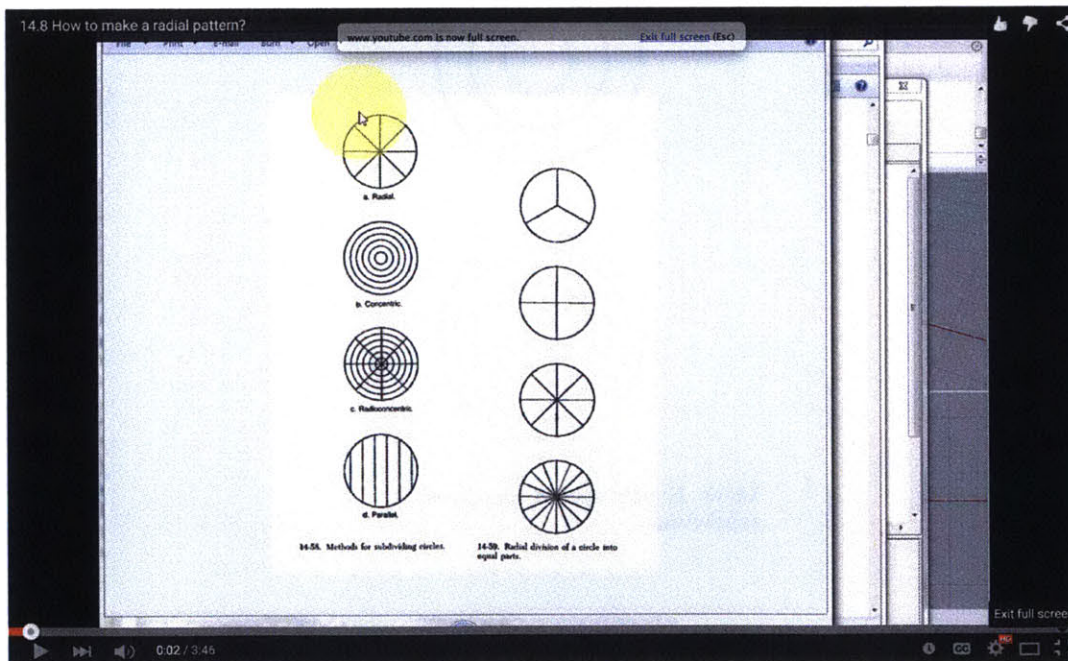
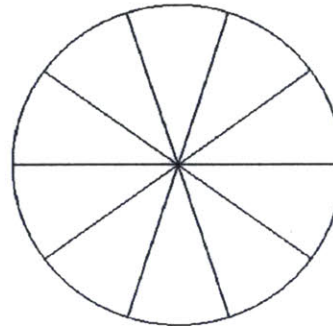
def scale(obj, x, y):
    origin = [0, 0, 0]
    scale = [x, y, 1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObjects(obj, center, angle)
    return result

def radial(n_parts):
    circle()
    increment = 360 / n_parts
    angle = 0
    count = range(1, n_parts+1)
    for i in count:
        lineID = line()
        lineID = scale(lineID, 10, 1)
        lineID = rotate(lineID, angle)
        angle = angle + increment

radial(10)
```

RESULT



Module 195: 14.8 Arcs 5

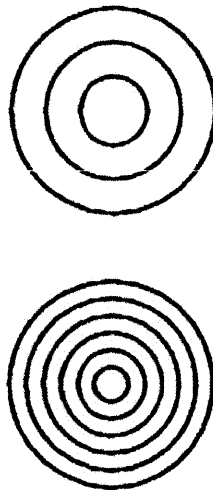
14.
TRANSFORMATIONS.

14.8 MOTIFS CONSTRUCTED FROM ARCS

In the case of concentric subdivision, we must consider how to space the circles. The simplest approach is to subdivide the radius of the outer circle into a specified number of equal parts. This is accomplished with the following procedure

[sample codes on the right side]

Some examples of output are shown in figure 14-60.



14-60. Evenly spaced concentric subdivision.

CODE

```
import rhinoscriptsyntax as rs

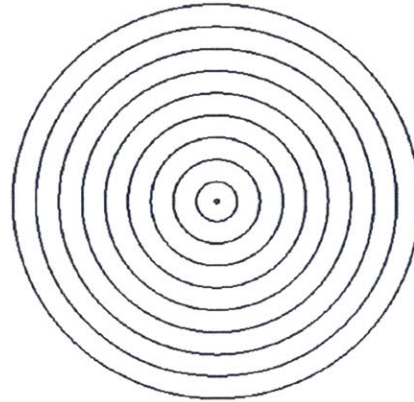
def circle():
    center = [0, 0, 0]
    r = 10
    obj = rs.AddCircle(center, r)
    return obj

def scale(obj, x, y):
    origin = [0, 0, 0]
    scale = [x, y, 1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

def concentric(n_parts):
    diameter = 1
    spacing = diameter * n_parts
    repeat = range(n_parts)
    for part in repeat:
        cir = circle()
        cir = scale(cir, diameter, diameter)
        diameter = diameter - spacing

concentric(10)
```

RESULT



Module 196: 14.8 Arcs 6

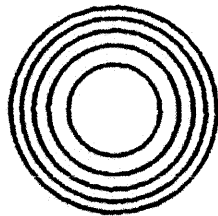
14. TRANSFORMATIONS.

14.8 MOTIFS CONSTRUCTED FROM ARCS

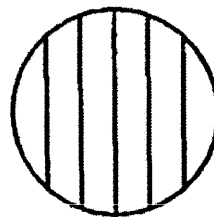
Another possibility is a so-called Fresnel subdivision into annular regions of equal area. This requires a rather more complex procedure.

[sample codes on the right side]

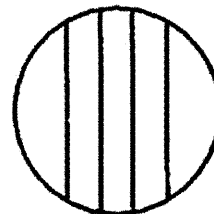
As the example of output in figure 14-61 illustrates, Fresnel circles become more closely spaced as they become larger. Similar approaches can be taken to subdivision by parallel lines. The lines may be equally spaced, or they may divide the circle into equal areas (fig. 14-62).



14-61. Fresnel subdivision.



a. Equal spacing.



b. Equal areas.

14-62. Subdivision by parallel lines.

CODE

```
import rhinoscriptsyntax as rs
import math

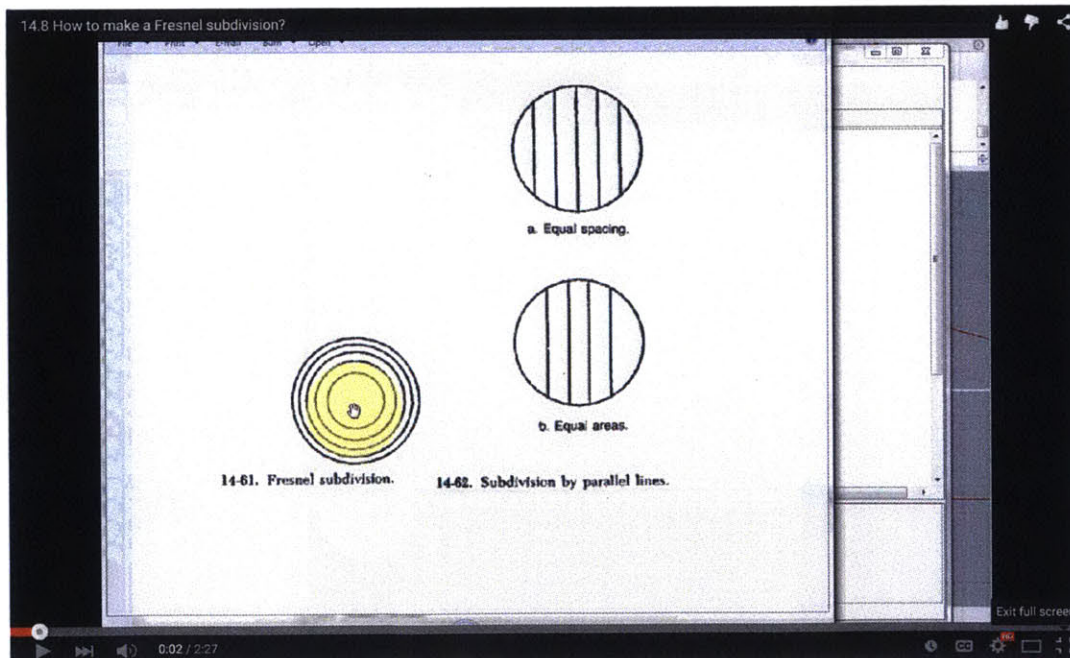
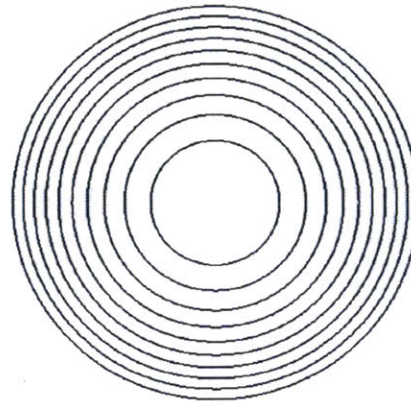
def circle():
    center = [0, 0, 0]
    r = 10
    obj = rs.AddCircle(center, r)
    return obj

def scale(obj, x, y):
    origin = [0, 0, 0]
    scale = [x, y, 1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

def fresnel(n_parts):
    s = 1
    spacing = s / n_parts
    repeat = range(n_parts)
    for part in repeat:
        diameter = math.sqrt(s)
        cir = circle()
        cir = scale(cir, diameter, diameter)
        s = s - spacing

fresnel(10)
```

RESULT



Module 197: 14.8 Arcs 7

14.
TRANSFORMATIONS.

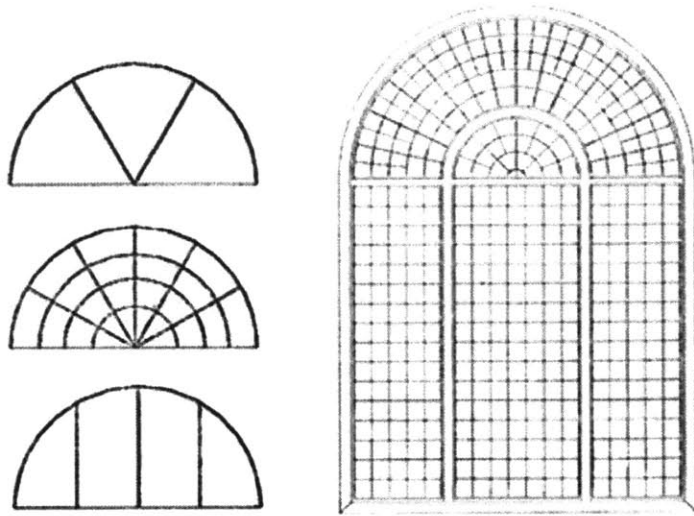
14.8 MOTIFS CONSTRUCTED FROM ARCS

The subdivided semicircular window, or fanlight, is a closely related architectural type. The subdivision by mullions and transoms may be radial, or radio concentric, or divided by parallel vertical lines (fig. 14-63).

The construction shown in figure 14-64 is another variant on the principle of radio concentric subdivision. It might be used for the layout of wedge shaped rooms in a plan, or the layout of the voussoirs of an arch. A reasonable parameterization is by the inner radius, outer radius, start angle, finish angle, and number of divisions. The following procedure produces patterns of this type

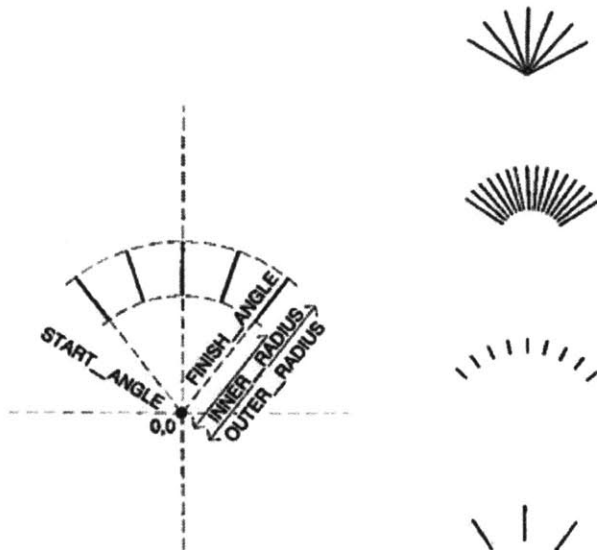
[sample codes on the right side]

Various examples of output are shown in figure 14-65.



a. Methods of subdivision. b. An elaborate arched window by Sir Christopher Wren.

14-63. Fanlight motifs.



14-64. The voussoir motif.

14-65. Some output from Radioconcentric.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs

def line(x1, x2):
    start = [x1, 0, 0]
    end = [x2, 0, 0]
    obj = rs.AddLine(start, end)
    return obj

def scale(obj, x, y):
    origin = [0, 0, 0]
    scale = [x, y, 1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObjects(obj, center, angle)
    return result

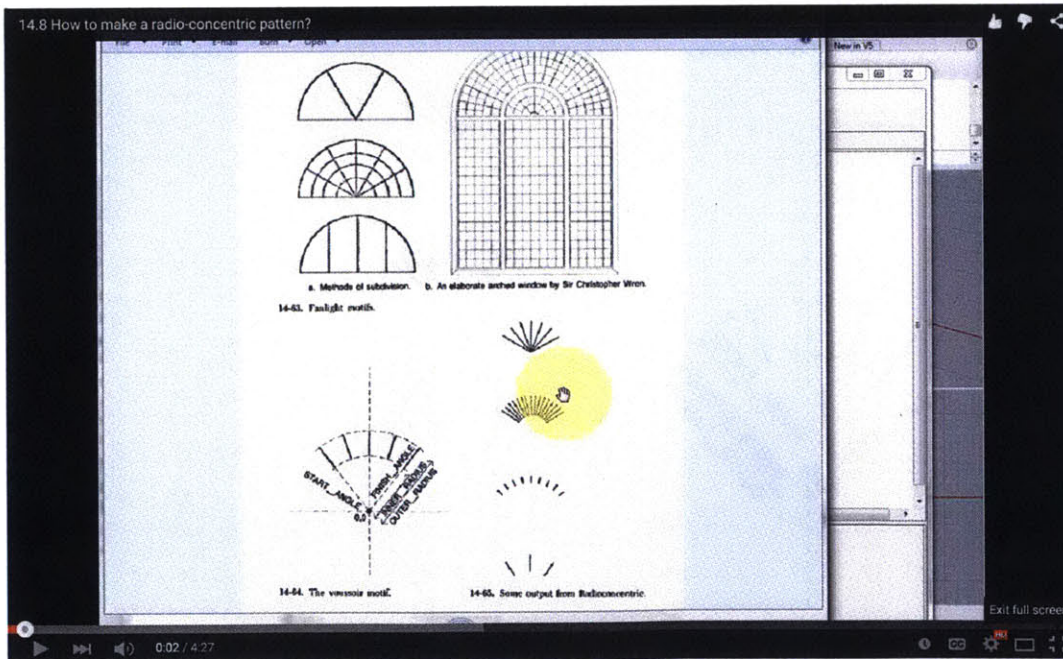
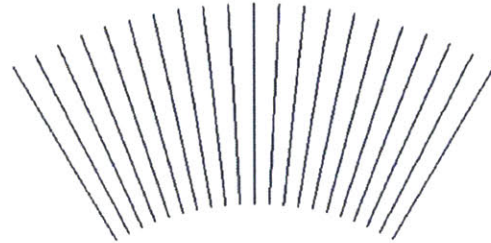
def translate(obj, tx, ty):
    translation = [tx, ty, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def radioconcentric(inner_radius, outer_radius,
                    start_angle, finish_angle,
                    n_parts):

    line_length = outer_radius - inner_radius
    increment = (finish_angle - start_angle) / n_parts
    angle = start_angle
    repeat = range(1, n_parts + 2)
    for part in repeat:
        lineID = line(inner_radius, outer_radius)
        lineID = scale(lineID, line_length, 1)
        lineID = translate(lineID, inner_radius, 0)
        lineID = rotate(lineID, angle)
        angle = angle + increment

radioconcentric(3, 6, 60, 120, 20)
```

RESULT



Module 198: 14.8 Arcs 8

14.
TRANSFORMATIONS.

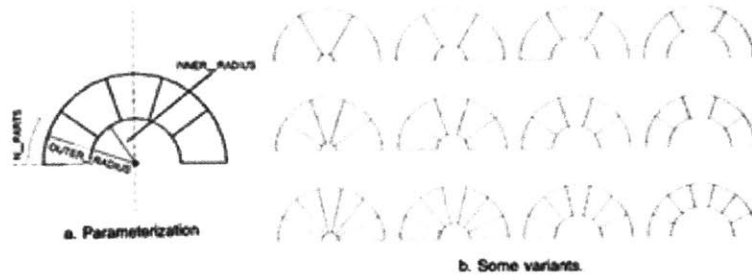
14.8 MOTIFS CONSTRUCTED FROM ARCS

Using the Arc and Radio concentric procedures to generate an intrados (inner curve of an arch), radial subdivisions, and an extrados (outer curve), we can develop a general procedure for construction of arches of any thickness, with any specified number of voussoirs. The parameterization is as shown in figure 14-66a, and the code runs as follows:

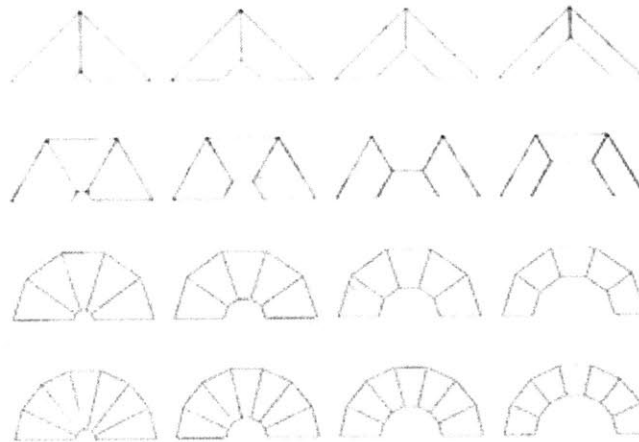
[sample codes on the right side]

An array of variant arches that were generated by this procedure is illustrated in figure 14-66b.

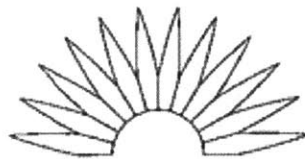
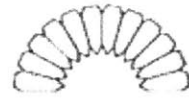
If the intrados and extrados are created with straight line segments, rather than arcs, a subtly different array of instances emerges (fig. 14-67). Further varieties can be produced by introducing procedural parameters to allow variation of intrados and extrados profiles (fig. 14-68).



14-66. An arch with voussoirs.



14-67. Arches with intrados and extrados formed by straight line segments.



14-68. More arches, produced by allowing variation of intrados and extrados profiles.

Synthetic Tutor

CODE

```
import rhinoscriptsyntax as rs
import math

def line(x1, x2):
    start = [x1, 0, 0]
    end = [x2, 0, 0]
    obj = rs.AddLine(start, end)
    return obj

def scale(obj, x, y):
    origin = [0, 0, 0]
    scale = [x, y, 1]
    result = rs.ScaleObject(obj, origin, scale)
    return result

def rotate(obj, angle):
    center = [0, 0, 0]
    result = rs.RotateObjects(obj, center, angle)
    return result

def translate(obj, tx, ty):
    translation = [tx, ty, 0]
    trans = rs.MoveObject(obj, translation)
    return trans

def radioconcentric(inner_radius, outer_radius,
                    start_angle, finish_angle,
                    n_parts):

    line_length = outer_radius - inner_radius
    increment = (finish_angle - start_angle) / n_parts
    angle = start_angle
    repeat = range(1, n_parts + 2)
    for part in repeat:
        lineID = line(inner_radius, outer_radius)
        lineID = rotate(lineID, angle)
        angle = angle + increment

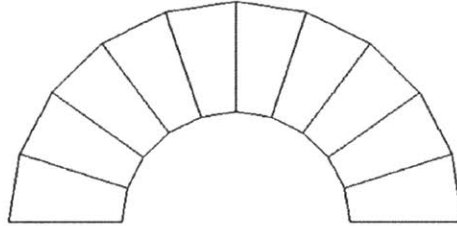
def arc(xc, yc, radius, start_angle, stop_angle,
        n_segments):

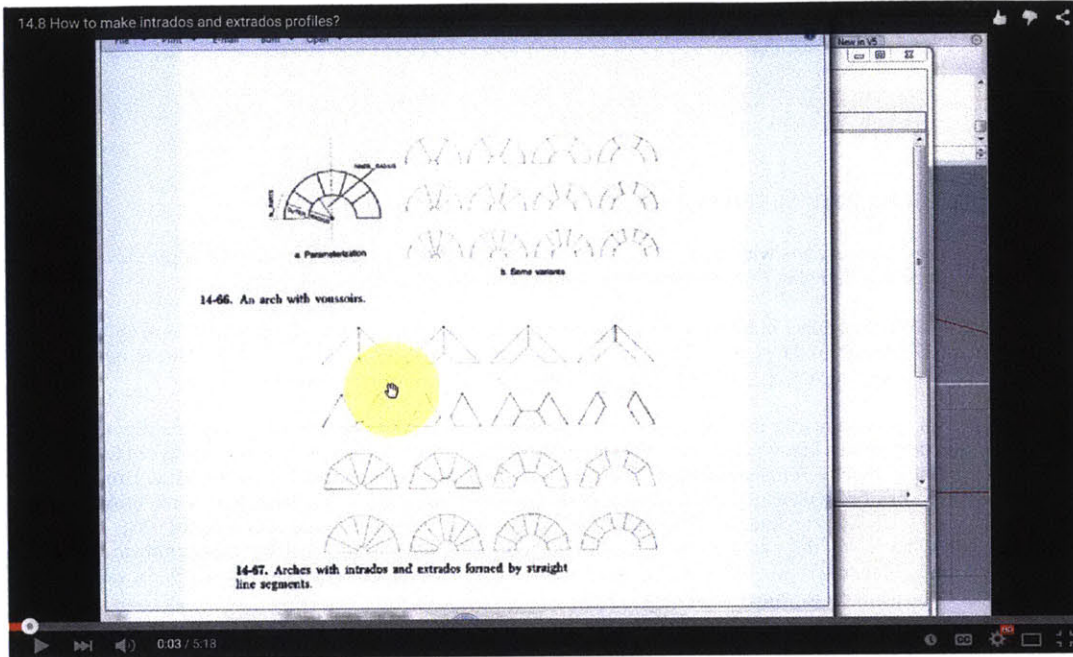
    rad = 0.01745
    degrees = (stop_angle - start_angle) / n_segments * rad
    theta = start_angle * rad
    repeat = range(1, n_segments+2)
    result = []
    for sides in repeat:
        y = yc + radius * math.sin(theta)
        x = xc + radius * math.cos(theta)
        if (sides == 1):
            p0 = [x, y, 0]
        else:
            p1 = [x, y, 0]
            ln = rs.AddLine(p0, p1)
            result.append(ln)
            p0 = p1
        theta = theta + degrees
    return result

def arch_modified(inner_radius, outer_radius, n_parts):
    arc(0,0,inner_radius,0,180,n_parts)
    arc(0,0,outer_radius,0,180,n_parts)
    radioconcentric(inner_radius,outer_radius,0,180,n_parts)

arch_modified(4,8,10)
```

RESULT





Module 199: 14.9 Composition

14.
TRANSFORMATIONS.

14.9 COMPOSITIONS WITH MIXED SYMMETRY

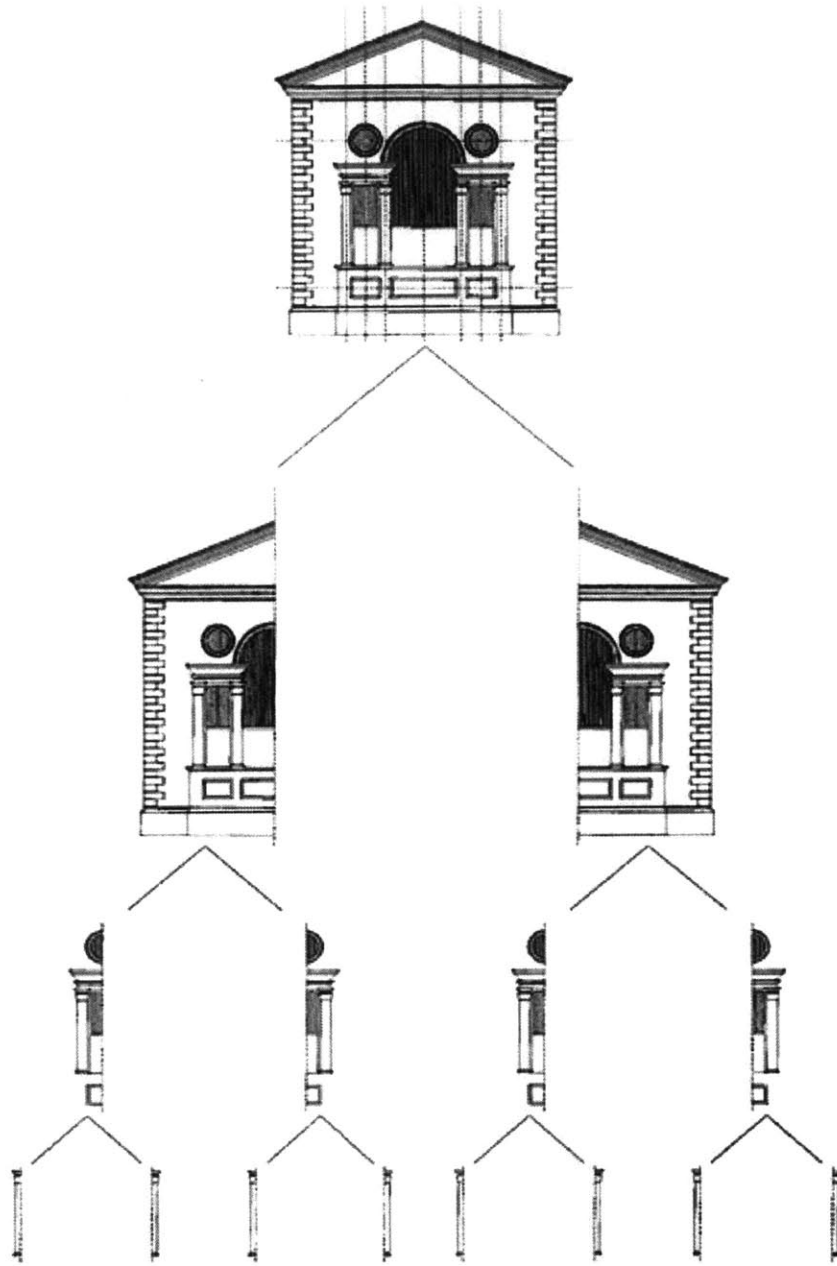
Many compositions contain parts with different symmetries. Consider, for example, the classical temple elevation in figure 14-69. The whole thing has bilateral symmetry, and the colonnade has frieze symmetry.

The plan (fig. 14-69) has a rather more complex combination of symmetries. The overall rectangle has symmetry about two axes. The details of the plan, however, distinguish between front and back and so reduce this to bilateral symmetry. The column grid sets up wallpaper symmetry, but this is broken by insertion of the cellar.

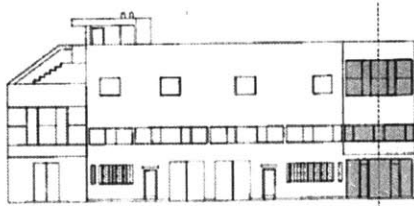
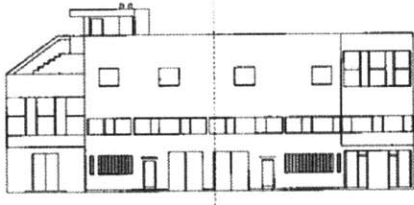
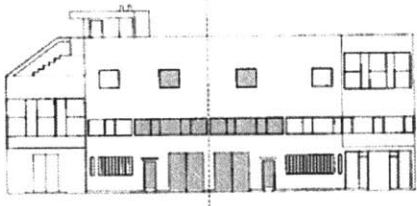
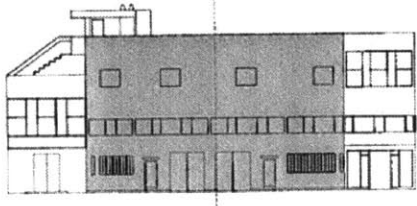
In general, the various symmetries that can be discovered within a composition suggest an appropriate way to break the composition down into a hierarchy of elements and subsystems. The elementary motifs can then be put together by a corresponding hierarchy of transformations that expresses the various symmetries that are involved. Broken symmetries can be handled with conditionals. Figure 14-70 shows an elevation by Sir Christopher Wren broken down in this way. In classical architecture, generally, there is symmetry of each element, symmetry of each subsystem, and symmetry of the whole. Early modern architects, on the other hand, often arranged symmetrical elements to form asymmetrical overall compositions (fig. 14-71). In Japanese stone gardens (Ryoan-ji, for example), asymmetrical elements form asymmetrical wholes.

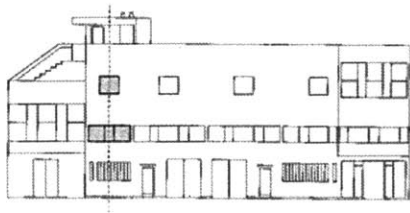
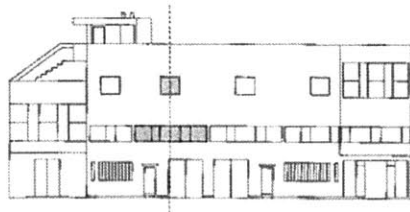
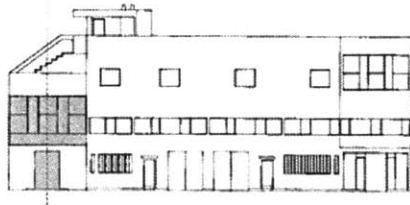
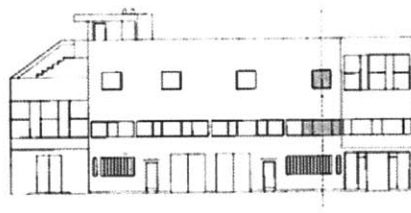
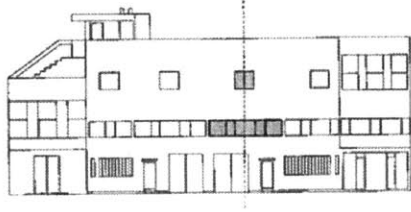


14-69. The symmetries of a classical temple in elevation and plan.



14-70. Hierarchy of symmetries in an elevation by Sir Christopher Wren (St. Olave's, Jewry).





14-71. Early modern architectural composition; symmetrical elements form an asymmetrical whole. (The Cooks' Villa, Coonara, south elevation)

Module 200: Final Test

[Back to Contents](#)

FINAL EXERCISE¹¹¹

DIRECTIONS

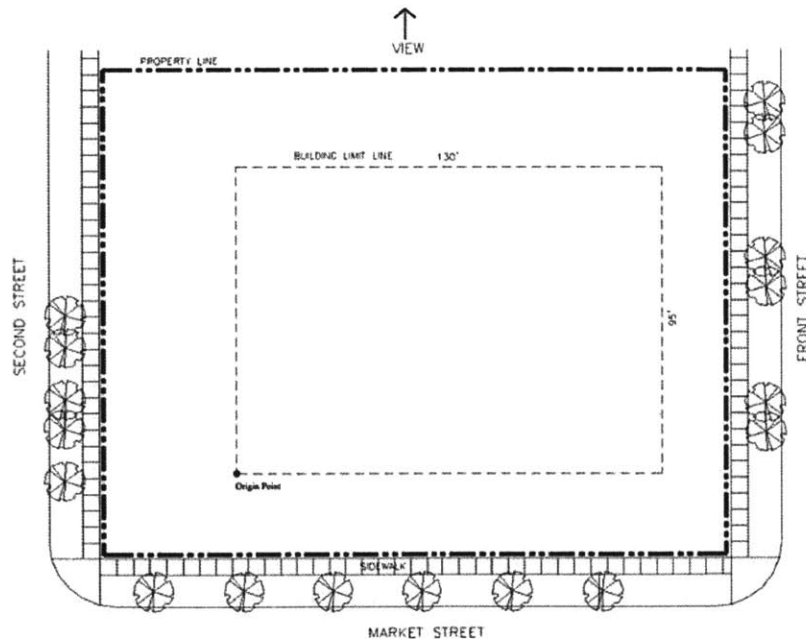
Using Python programming, develop floor plans for the two-story building described in the program. Your floor plans should be responsive to given program and code requirements and should reflect principles of sound design logic. Adequate and code-compliant circulation should be provided, and the orientation of the building should be responsive to site influences. Develop your floor plans by sizing and locating all required spaces and any necessary corridors on the site plan provided on the work screen. Indicate all windows, doors, and wall openings. Before beginning your solution, you should review the program and code information that can be accessed and familiarize yourself with the site plan on the work screen. Please draw your plan inside the building limit line. Consider the low-left corner as an origin point.

Please upload your python file: No file chosen

Please upload your first floor JPG file: No file chosen

Please upload your second floor JPG file: No file chosen

SITE PLAN



PROGRAM

The family life center will provide recreational and fellowship facilities for a community church.

1. The site is located on Market Street adjacent to a community church. Parking is available off the site.
2. The major view is to the north.

Synthetic Tutor

3. The receptionist is to have visual control of the entry to each of the following spaces: the lobby, the game room, and the children's room.
4. The main entrance door shall face west.
5. All spaces shall have a 9 ft ceiling height except the multi-purpose room, which shall have an 18 ft ceiling height.
6. The area of each space shall be within 10 percent of the required program area.
7. The total corridor area shall not exceed 25 percent of the total program area.
8. The second floor envelope must be congruent with or wholly contained within the first floor envelope with the exception that doors to the exterior may be recessed for weather protection.

PROGRAM- SPACES

Tag	Name	Area (ft ²)	Requirements
ST	Stair	800	2 per floor @ 200 ft ² per stair
E	Elevator Shaft	200	1 per floor @ 100 ft ² each; Minimum dimension = 7 ft
EE	Elevator Equipment Room	100	
EM	Electrical/Mechanical Room	500	
AO	Assistant Director's Office	200	Exterior window required; Direct access to Secretarial Office
CR	Children's Room	750	Exterior window required; Near Multi-purpose Room
DO	Director's Office	350	Exterior window required; Direct access to Secretarial Office
GR	Game Room	1,350	View—exterior window required
L	Lobby	700	Main Entrance
LM	Large Meeting Room	1,000	Exterior window required
LR	Locker Rooms	200	2 @ 100 ft ² each; Exterior windows prohibited; Direct access to Multi-purpose Room
MP	Multi-purpose Room	2,600	View—exterior window required; 18 ft ceiling; 2 exits; First floor
R	Receptionist	400	Exterior window required; Near Lobby
SM	Small Meeting Room	750	Exterior window required; Near Large Meeting Room
SO	Secretarial Office	500	Exterior window required; Near Large Meeting Room; Second Floor
SW	Social Worker	500	Exterior window required
TR	Toilet Rooms	800	2 per floor @ 200 ft ² each
TS	Table/chair Storage	300	Near Multi-purpose Room
TOTAL PROGRAM AREA		12,000	ft ²

CODE

Comply with the following code requirements. These are the ONLY code-related criteria you are required to use.

DEFINITIONS

1. Means of egress: A continuous and unobstructed path of travel from any point in a building to a public way. A means of egress comprises the vertical and horizontal means of travel to an exit and includes intervening doors, interior wall openings, corridors, circulation areas, and stairs.
2. Circulation area: A lobby or a space designated as an "area."
3. Exit: That portion of a means of egress that provides a protected route of travel to the exit discharge. Exits include both exterior exit doors and exit stairways.

EXITING REQUIREMENTS

1. Provide a minimum of two exits from each floor separated by a travel distance equal to not less than 1/2 of the length of the maximum overall diagonal dimension of the floor to be served.
2. Every room shall connect directly to a corridor or circulation area.
Exception: elevator equipment rooms and rooms with an area of 50 ft² or less may connect to a corridor or circulation area through an intervening space, but not directly to a stair.
3. In rooms required to have two exit doors, separate the two exit doors by a distance equal to not less than 1/2 of the length of the maximum overall diagonal dimension of that room.
Exit doors may discharge directly to the exterior of the building at grade.
4. Required exit doors shall swing in the direction of egress travel.
5. Door swings shall not reduce the minimum clear exit path to less than 3 ft.

CORRIDORS

1. Discharge corridors directly to the exterior at grade or through stairs or circulation areas.
2. Do not interrupt corridors with intervening rooms - circulation areas are not considered to be intervening spaces.
3. Maximum length of dead-end corridors: 20 ft.
4. Minimum clear width of corridors: 6 ft.

STAIRS

1. Discharge stairs directly to the exterior at grade.
2. Connect stairs directly to a corridor or circulation area at each floor with exit access doors.
3. Minimum width of stairs: 4 ft.

PROCEDURAL TIPS

- Read the Program and Program Spaces listing carefully.
- You may want to draw each space so that it has approximately the required area, assemble the spaces into a finished floor plan, then adjust the areas if necessary. Note that the dimensions that are given as you draw a space are from wall centerline to wall centerline. It is especially important to take this into consideration when you are drawing corridors to code-required widths, which are measured from one edge of the corridor to the other.
- Check for overlaps while you are drawing.
- You are not required to show doors or wall openings in elevator wall.

⚡ This problem comes from ARE 4.0 EXAM GUIDE Schematic Design, NCARB 2012

Back to Contents How do you evaluate this content?: Useless 1 2 3 4 5 Highly Useful

Synthetic Tutor

APPENDIX C.

Computer Vision Tutor Survey Questions.

1. Are you currently in an undergraduate or a graduate program?

1. Undergraduate
2. Graduate

2. What year are you in?

1. More than 5
2. 5
3. 4
4. 3
5. 2
6. 1

3. What is your major?

1. Architecture
2. Urban Design
3. Real Estate

4. How many architectural design studios did you have previously (including this semester)? Please include any studios in your undergraduate school(s), if you are a graduate student.

5. Overall, how do you evaluate the feedback the Machine Tutor provided on your sketch?

1. Highly Satisfied
2. Somewhat Satisfied
3. Neither Satisfied nor Unsatisfied
4. Somewhat Unsatisfied
5. Highly Unsatisfied

6. How much do you think the feedback is related to your design concept?

1. Highly Related
2. Somewhat Related
3. Neither Related nor Unrelated
4. Somewhat Unrelated
5. Highly Unrelated

7. How much do you think the feedback is useful for your design improvement?

Synthetic Tutor

1. Highly Useful
2. Somewhat Useful
3. Neither Useful nor Useless
4. Somewhat Useless
5. Highly Useless

8. Comparing to your previous (human) studio instructors, how do you evaluate the feedback the Machine Tutor provided?

1. The Machine Tutor's feedback is so much better than instructors' feedback
2. The Machine Tutor's feedback is somewhat better than instructors' feedback
3. Both are equally useful
4. Instructors' feedback is somewhat better than the Machine Tutor's feedback
5. Instructors' feedback is so much better than the Machine Tutor's feedback

9. How many times did you try the feedback of the Machine Tutor?

10. Do you have any suggestions or questions? Please feel free to write here.

11. Do you feel or see any potential benefits of the Machine Tutor?

12. Do you feel or see any potential drawbacks of the Machine Tutor?

APPENDIX D.

List of Architects and Their Projects that the Computer Vision Tutor learned.

1. Frank Gehry
 - a. Sirmai Peterson House
 - b. Winton Guest House
2. Frank Lloyd Wright
 - a. Dr. George Ablin House
 - b. Lockridge Medical Clinic
 - c. Massaro House
 - d. Norman Lykes House
 - e. Paul Olfelt House
3. Le Corbusier
 - a. Curutchet House
 - b. Ternisien House
 - c. Villa La Roche
 - d. Villa Le Lac
 - e. Villa Savoye
 - f. Villa Shodan
 - g. Villa Stein
 - h. Weessenhofsiedlung
4. Luis Sullivan
 - a. George Harvey House
5. Mies Van der Rohe
 - a. 1300 Lake Shore Drive Apartment
 - b. Brookfarm Apartment
 - c. Esplanade Apartment Building
 - d. Farnsworth House
 - e. Highfield House Apartments
 - f. Joseph Cantor House
 - g. Leon J. Caine House
 - h. Tugendhat House
 - i. Ulrich Lange House
6. Phillip Johnson
 - a. Johnson House at Cambridge
 - b. Rockefeller Guest House
 - c. The Glass House
7. Richard Meier

Synthetic Tutor

- a. Bodrum House
 - b. Douglas House
 - c. Rachofsky House
 - d. Smith House
 - e. Vitrum Apartments
8. Robert Venturi
- a. The Lieb House
 - b. Trubek and Wislocki House
 - c. Vanna Venturi House
9. Tadao Ando
- a. 4 x 4 House
 - b. Benesse House
 - c. Row House in Sumiyoshi Azuma House

APPENDIX E.

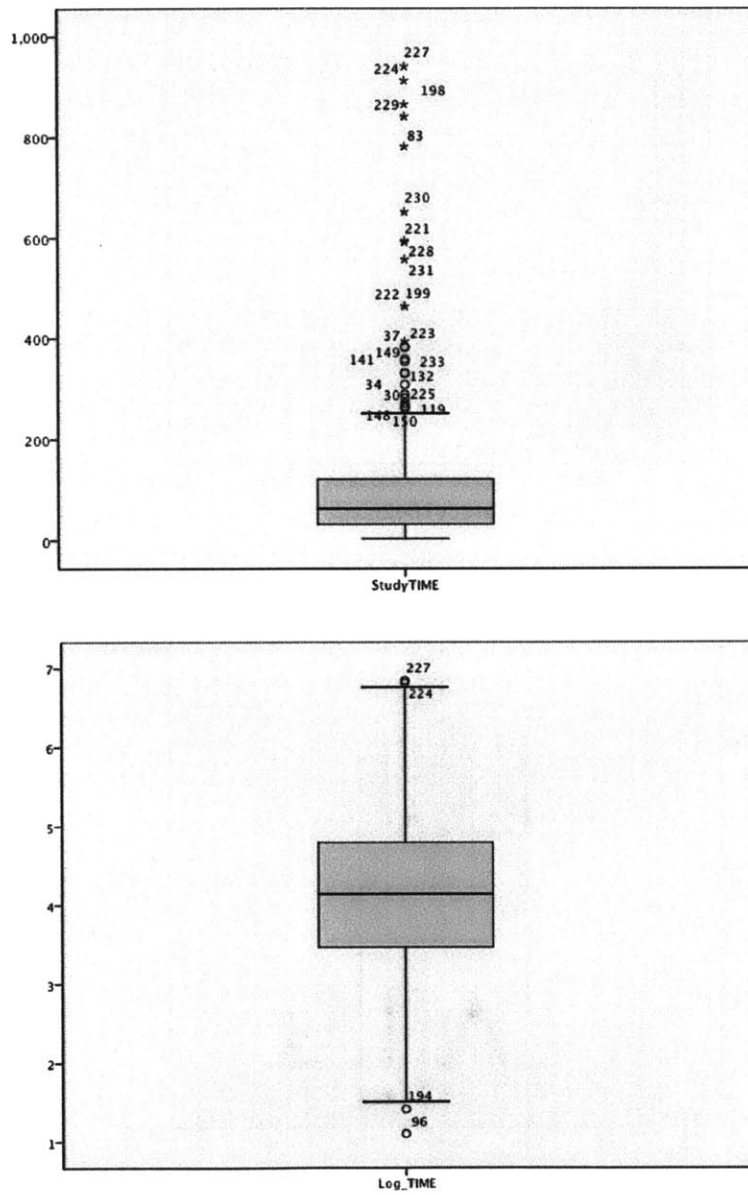


Figure 27. Boxplots of participants' total study time in its original scale (top) and in natural logarithm scale (bottom).

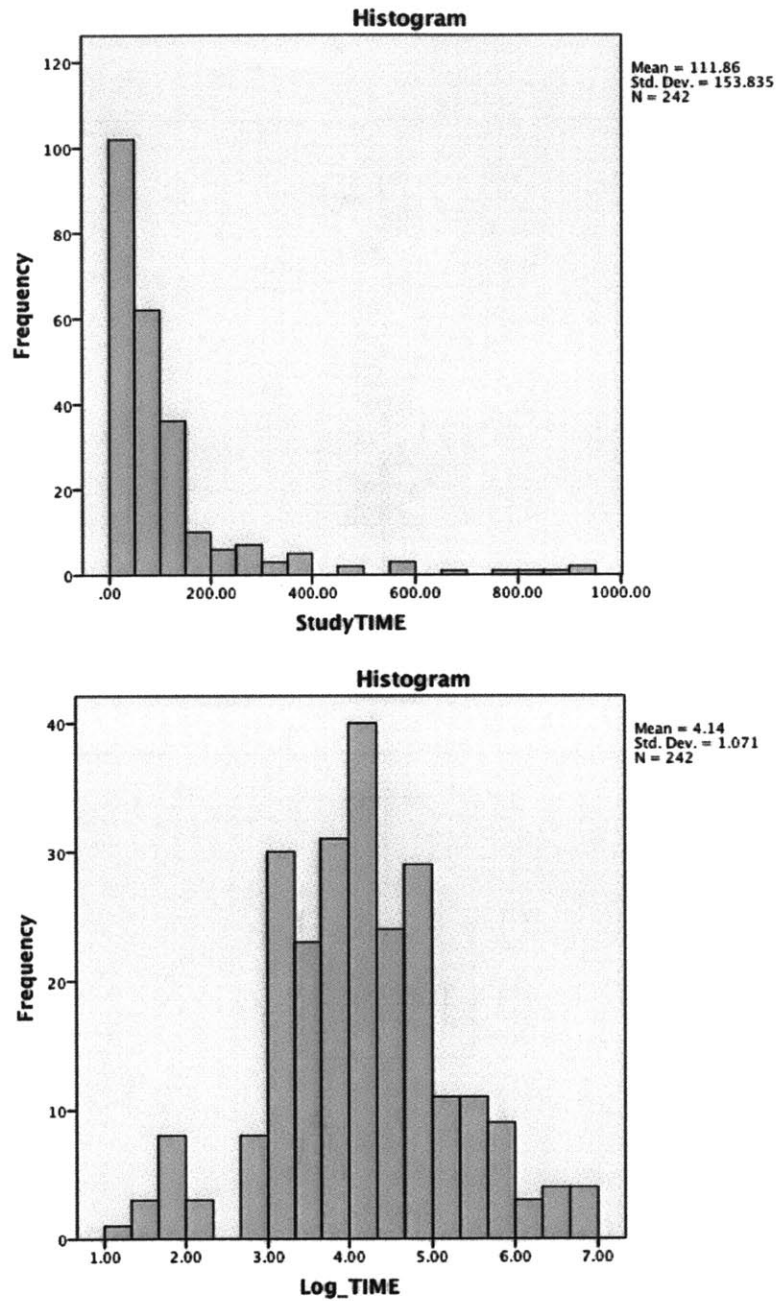


Figure 28. The histogram of participant's total study time in its original scale (n = 242, top) and in natural logarithm scale (n = 242, bottom).

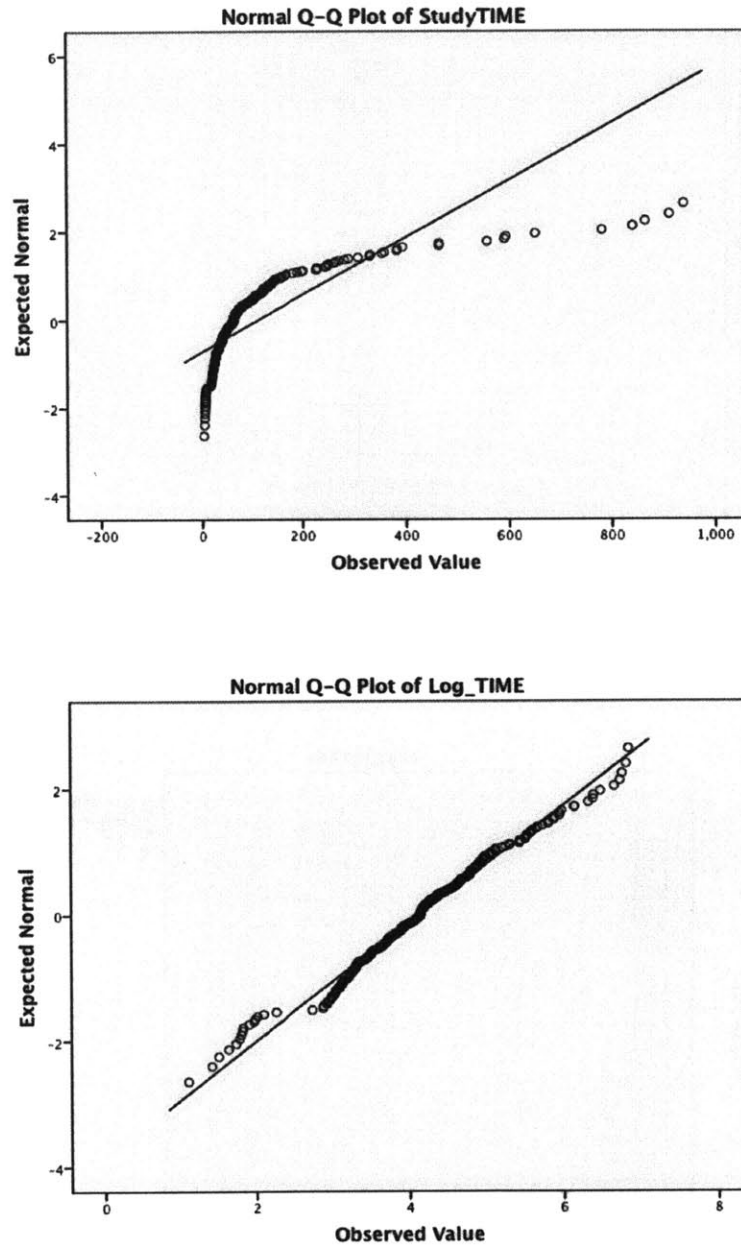


Figure 29. The normality test plots of participants' total study in its original scale (top) and in natural logarithm scale (bottom). As seen on the right Q-Q plot, the normality is improved compared the left Q-Q plot. The log-transformed samples lie closer to the samples in its original scale.

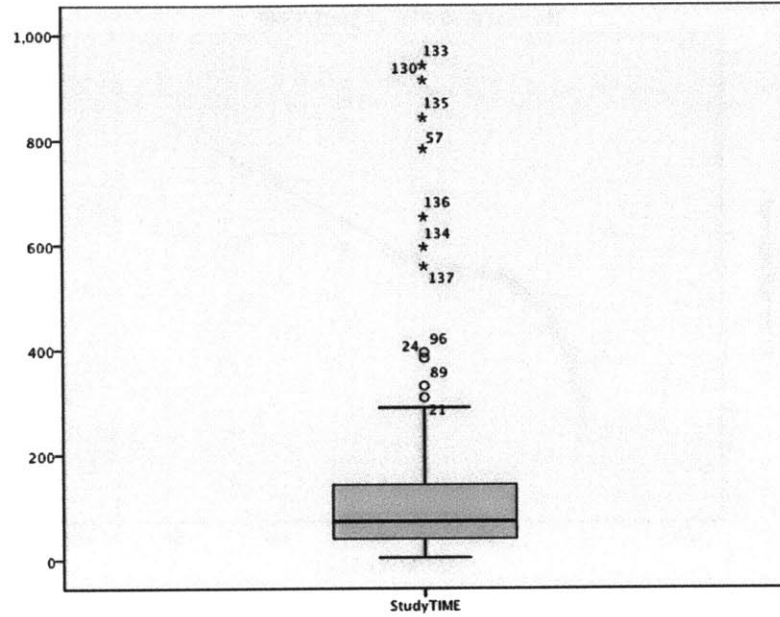


Figure (a) The boxplot of the ML participants' daily study time in minute.

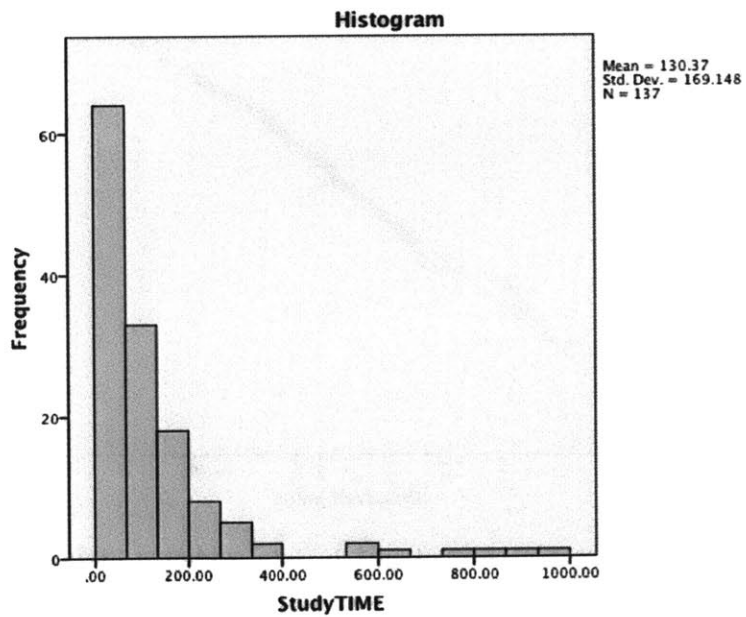


Figure (b) The histogram of the ML participants' daily study time in minute.

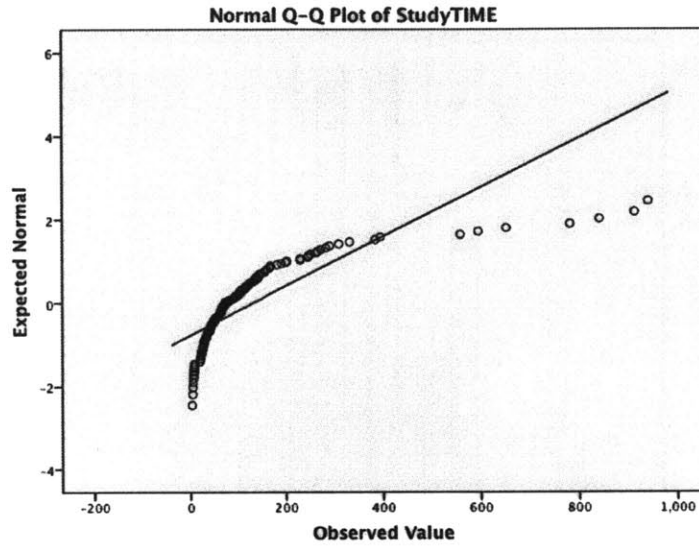


Figure (c) The Q-Q plot of the ML participants' daily study time in minute.

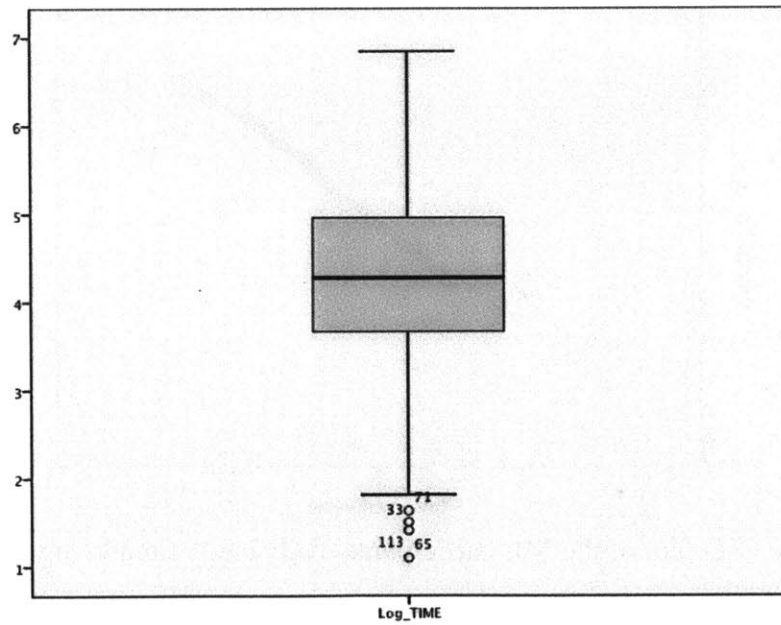


Figure (d) The boxplot of the ML participants' daily study time in log-scale.

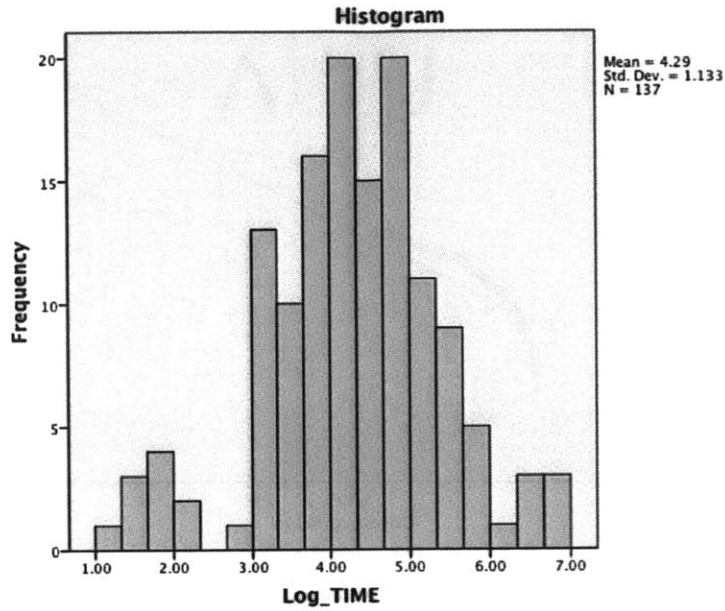


Figure (e) The histogram of the ML participants' daily study time in log-scale.

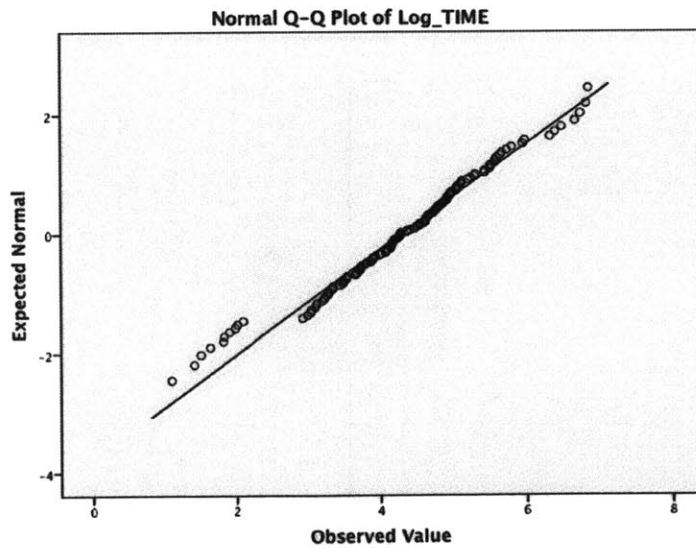


Figure (f) The Q-Q plot of the ML participants' daily study time in log-scale.

Figure 32. Descriptive statics for the Daily Study Time (a, b, and c) and the Daily Log_e Study Time (d, e, and f) of the ML participants (n = 137)

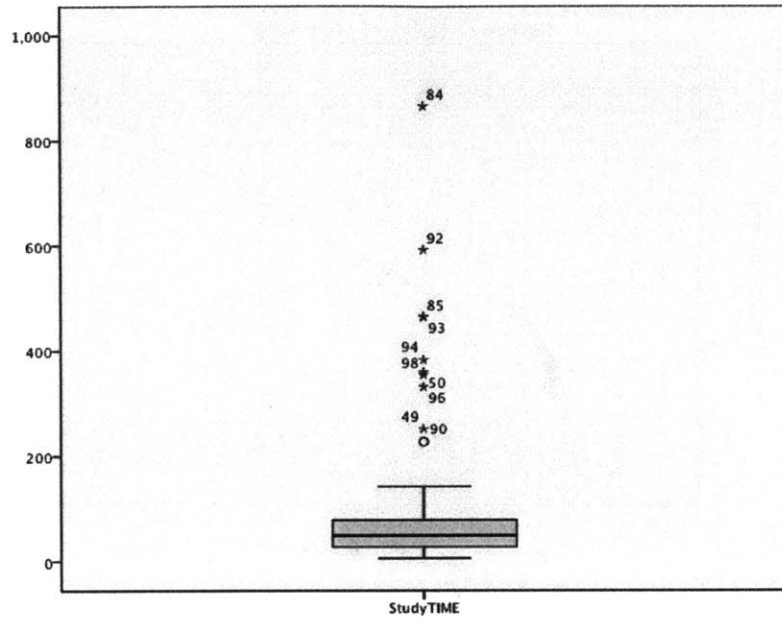


Figure (a) The boxplot of the non-ML participants' daily study time in minute.

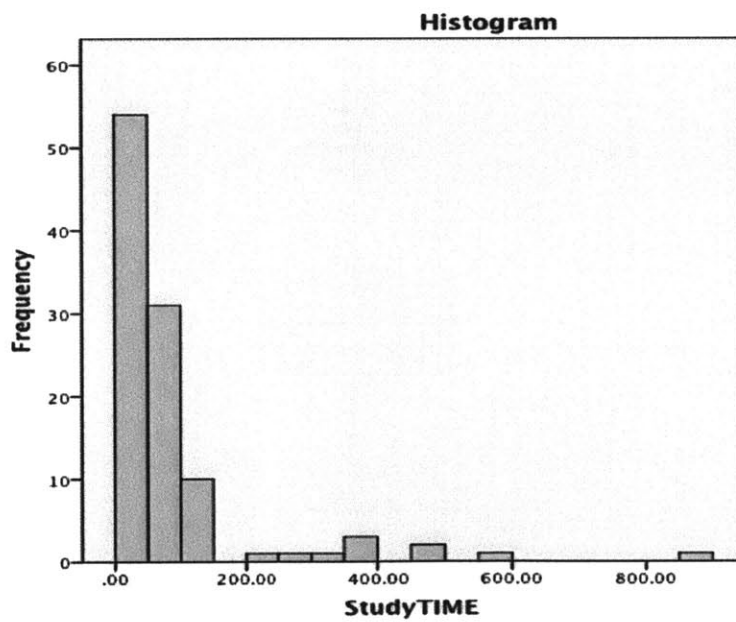


Figure (b) The histogram of the non-ML participants' daily study time in minute.

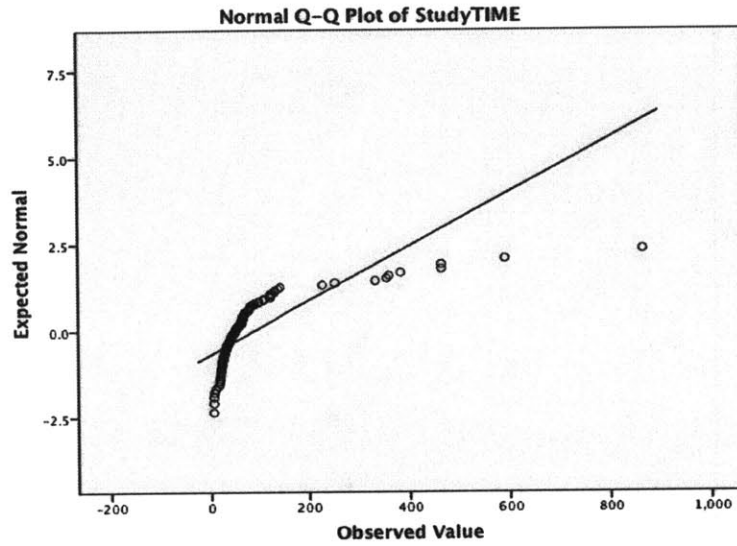


Figure (c) The Q-Q plot of the non-ML participants' daily study time in minute.

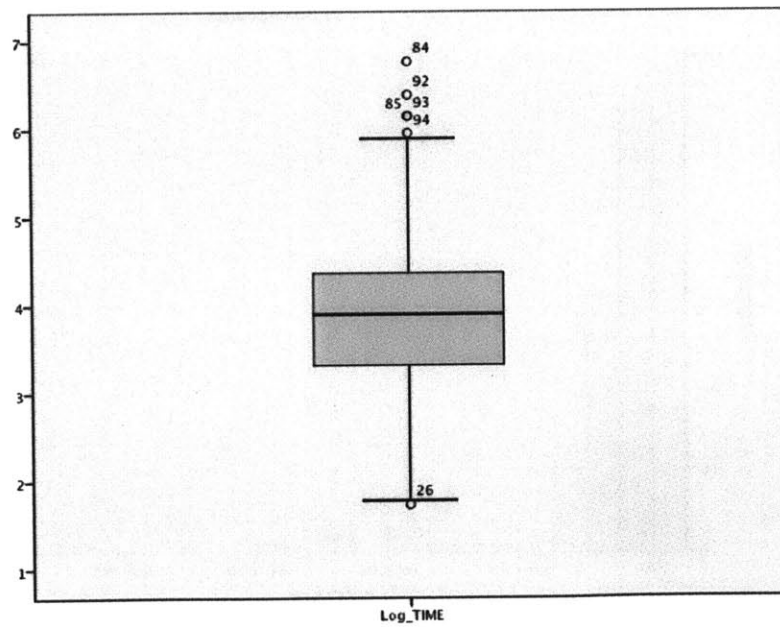


Figure (d) The boxplot of the non-ML participants' daily study time in log-scale.

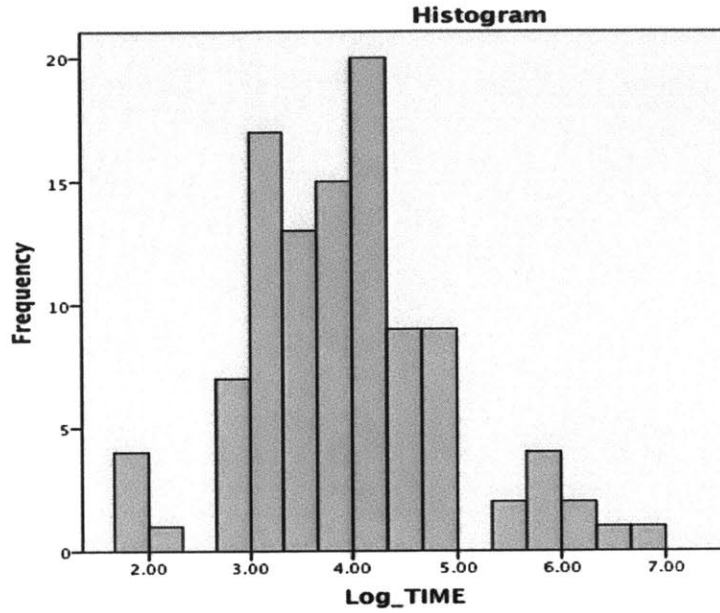


Figure (e) The histogram of the ML participants' daily study time in log-scale.

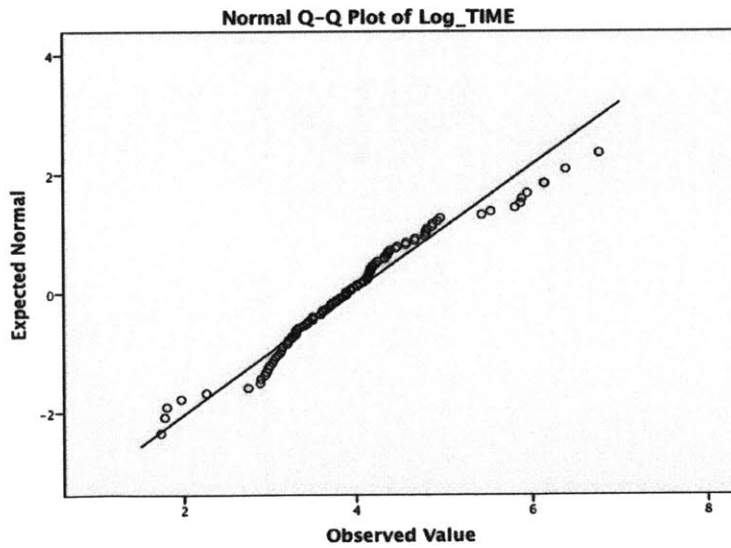


Figure (f) The Q-Q plot of the ML participants' daily study time in log-scale.

Figure 33. Descriptive statics for the Daily Study Time (a, b, and c) and the Daily Log_e Study Time (d, e, and f) of the non-ML participants (n = 105)