CAT-SOOP: A Tool for Automatic Collection and Assessment of Homework Exercises

by

Adam J. Hartz
B.Sc. Computer Science and Engineering, M.I.T. 2011

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

June 2012

Author: ....
Department of Electrical Engineering and Computer Science
May 21, 2012

Certified By: ....
Tomás Lozano-Pérez
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted By: ....
Professor Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

CAT-SOOP: A Tool for Automatic Collection and Assessment of Homework Exercises

by

Adam J. Hartz

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

CAT-SOOP is a tool which allows for automatic collection and assessment of various types of homework exercises. CAT-SOOP is capable of assessing a variety of exercises, including symbolic math and computer programs written in the Python programming language. This thesis describes the design and implementation of the CAT-SOOP system, as well as the methods by which it assesses these various types of exercises. In addition, the implementation of an add-on tool for providing novel forms of feedback about student-submitted computer programs is discussed.

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank all of the people —students and staff alike —with whom I have worked as part of MIT's *6.01: Introduction to EECS I.* Were it not for the experience I gained through working with 6.01, I would not be the person I am today; at the very least, I'd be submitting a different thesis!

In particular, I would like to thank Professors Leslie Kaelbling, Tomás Lozano-Pérez, and Dennis Freeman, who have been fantastic mentors throughout my time at MIT.

I would be remiss if I did not also thank all of the people whose code I borrowed, or from whose code I drew inspiration, while working on this project. In particular, I have to thank Armin Ronacher (whom I have never met, but whose Python AST module was invaluable) and Phil Guo, whose Online Python Tutor made my work on the Detective possible.

Additional thanks go to Chris Buenrostro, Seleeke Flingai, Nora Mallory, Abiy Tasissa, and Adnan Zolj for their willingness to help, listen, and discuss, and for putting up with my rants about programming and education over the years.

Last, but not least, I would like to thank my family for their continued support in all of my endeavors.

Contents

List of Figures

Chapter 1

Introduction

CAT-SOOP is a tool designed to automate the collection and assessment of homework exercises for a variety of disciplines. This thesis focuses on the design and implementation of CAT-SOOP, and on the methods by which it evaluates and provides feedback on submissions to different types of questions. Significant attention is also given to the Detective, an add-on to CAT-SOOP designed to provide novel types of feedback in response to student submissions to programming exercises.

Throughout, design decisions are considered in the context of other automatic tutors, principles of software engineering, and educational research.

1.1 Background

The history of systems like CAT-SOOP[1] dates back to 1926, when Pressey[20], noting the simplicity of many types of drilling exercises, presented a mechanical device capable of posing multiple choice questions to users, as well as collecting and scoring their submissions to said exercises.

Naturally, as technology has progressed since then, newer and more advanced systems have been developed to accomplish this same task, but more efficiently and for

[1] I will refer to these systems, which comprise components of Intelligent Tutoring Systems and Learning Management Systems, as "automatic tutors" throughout this document. Because this is something of an umbrella term, encompassing numerous projects with differing goals and features, I strive, when possible, to make clear the specific goals and features of the automatic tutor in question.

a broader range of problems. Checking of various types of problems is built into some Learning Management systems (e.g., Moodle[5] and LON-CAPA[11]), which often, in addition, take on the role of managing course materials, calendars, discussions, grades, etc.

Modern technologies have also allowed automatic tutoring systems to move beyond simple assessment of correctness, toward providing meaningful, conceptual feedback in response to students' submissions in a variety of contexts.

Bloom[2] has long since shown that one-on-one tutoring has dramatic benefits over traditional classroom instruction. Many automatic tutors thus attempt to recreate the feeling of interaction with a human tutor. It is certainly worth noting that such a system (i.e., an automatic tutor which accurately mimics a human tutor) has tremendous potential to help both students and staff alike, even if it works only for relatively simple concepts.

Since then, a wide variety of promising techniques have been attempted to improve the feedback generated by these systems. Among these are:

- Measuring clues about the user's affect (emotional state) and using that information to adjust the feedback presented[4]
- Using machine learning techniques to automatically generate hints for programming exercises[9]
- Recording a "trace" of submitted code as it is executed, and using this information to provide additional feedback[21]
- Attempting to create a conversational dialogue with the student[6]
- Creating an internal model of a student's understanding so as to individualize feedback[1]

While automatic tutors are not a replacement for in-person instruction, they can serve as an approximation thereof in a pinch, which can be invaluable to students. Particularly in introductory computer programming courses (but also in other fields, as well), students often begin with little-to-no relevant experience. A direct consequence of this is that students spend a lot of time on assignments, getting stuck and attempting to debug their solutions, but often with poor technique; many require a lot of help in one-on-one or small group scenarios to get over these hurdles. Because of this, most introductory courses (at least in post-secondary education) hold "office hours," where professors or teaching assistants are available to help with homework exercises or conceptual review. In most cases, students find these hours quite helpful, but there are certainly limitations:

- Many problems that novices face are simple to diagnose and fix, but require a nontrivial amount of time to explain. While these problems are certainly still important to the students who face them, the teaching assistants' time may be better spent helping to solve more complex problems, particularly if the diagnosing and explanation of these errors can be automated.
- There are a limited number of hours in the day, and teaching assistants cannot spend all of their time holding office hours, or even making themselves available via e-mail. Frequently, students working late at night miss out on the benefit of office hours.
- Not all teaching assistants are equal, and no single teaching assistant has seen every problem that students will encounter. An automatic tutor that can provide feedback for a variety of common problems can help to create some sense of uniformity with respect to the feedback students receive on their work.

Because of these reasons and more, automatic tutors have the potential to have a really positive impact on students' learning experience, particularly for novices, whose common errors tend to be easier to diagnose and fix.

What's more, students enjoy working with automatic tutors, and find them beneficial[2]. Figure 1-1 shows the results from an end-of-term survey in MIT's *6.01 Introduction to EECS I*, which shows that students, in general, found the assignments

[2]Buy-in on the part of the students should not be understated as a contributing factor to the overall success of these systems, or of any pedagogical experiment.

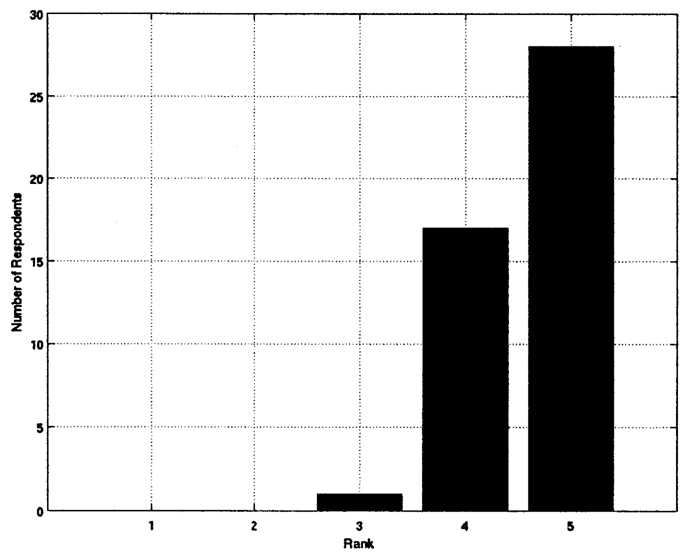


Figure 1-1: Students' responses to end-of term survey question relating to tutor2 for *6.01*, fall term 2011. Users were asked to rank their degree of agreement with the statement, "The on-line tutor helped me learn the 6.01 material," on a scale from 1 (total disagreement) to 5 (total agreement), with 3 as a neutral point. A total of 46 data points were collected.

delivered through the automatic tutor in *6.01* to be helpful. Similar results from an end-of-term survey in *6.003 Signals and Systems* (discussed in chapter 6) show that students also enjoy working with these types of software.

These results, along with the history, and the wide variety of available software in this area, have informed CAT-SOOP's design philosophy, as well as its implementation. Before discussing the specifics of its design, however, it is important to place CAT-SOOP in the context of the systems on which it is based, as well as to specify its purpose and design goals.

1.2 xTutor, tutor2, and CAT-SOOP

CAT-SOOP is the sibling of tutor2, an automatic tutor currently used in *6.01.* Both were developed in parallel[3], but completely ignorantly of one another; as time has gone on, however, certain parts of CAT-SOOP have found their way into tutor2, and vice versa[4].

In a sense, CAT-SOOP and tutor2 are both spiritual descendents of xTutor[5], an automatic tutor widely used at MIT throughout the 2000's, in a number of courses including *6.01*, *6.042 Discrete Math for Computer Science*, *6.034 Artificial Intelligence*, and the now-defunct *6.001 Structure and Interpretation of Computer Programs.* Both tutor2 and CAT-SOOP were designed as successors to xTutor in *6.01*; however, where tutor2 is essentially a port of xTutor to Python/Django, CAT-SOOP was started from a clean slate.

xTutor and tutor2 differ from CAT-SOOP in a number of respects. Firstly, CAT-SOOP is based on a design philosophy of simplicitly and minimalism. Thus, the focus of CAT-SOOP is extremely limited. CAT-SOOP's goal is to automate the collection and assessment of online homework exercises; intrinsically, this means that

[3]CAT-SOOP was originally designed for use in *6.01*; in fact, its name comes from the fact that CAT-SOOP was designed as an **A**utomatic **T**utor for **S**ix-**O**h-**O**ne **P**roblems.

[4]In particular, CAT-SOOP's symbolic math checking, which is described in chapter 3, was ported into tutor2, and tutor2 and CAT-SOOP both currently use a scheme for checking Python code (decribed in chapter 4) which is an amalgamation of the schemes originally used by the two.

[5]`http://icampus.mit.edu/xTutor/`

tasks such as managing a course calendar, or managing final grading and weighting of various assignments, are not included in—and are not designed to be handled by—CAT-SOOP[6]. While tutor2 and xTutor don't go the way of full-fledged Learning Management Systems, both do include features beyond the assessment of student submissions.

xTutor (at least the version used in *6.01* most recently) and tutor2 also have a number of *6.01*-specific details built directly into their core systems. While this doesn't hinder the use of these tutors by other courses, it does mean that other courses have to ignore these parts of the systems if they intend to use tutor2 of xTutor. One major goal in CAT-SOOP's design was modularity, based on the belief that the core system should be as minimal as possible, and any course-specific content should make its way into the system via plug-ins or extensions. Teaching *6.01* using CAT-SOOP, for example, would still involve writing a good deal of course-specific material, but this material would live outside the core system. Because this course-specific material still needs to be written, another design goal was to make the creation of new content as easy as possible.

One additional point worth noting is that, while xTutor and tutor2 allow only one course per instance (and thus require the installation of a new instance for each course[7]), CAT-SOOP allows multiple courses to coexist in the same instance, in the hopes of providing a centralized location for students to submit online homeworks for multiple courses.

When considering the various components of CAT-SOOP in relation to other automatic tutors, this thesis will primarily make reference and comparisons to tutor2, and occasionally to xTutor (particularly in areas where xTutor and tutor2 differ significantly).

[6]Currently, CAT-SOOP does per-problem scoring, but does not have any notion of how scores from multiple problems should be combined to generate a final score. In an ideal system, the grading scheme is something that should be easy to change, and thus not something that is hard-coded into the core system.

[7]This has the additional downside that users need to create accounts on each instance separately.

1.3 Outline

The remainder of this thesis is structured as follows:

Chapter 2 discusses the design and implementation of the CAT-SOOP base system.

Chapters 3 and 4 discuss the means by which CAT-SOOP assesses submissions to symbolic math exercises and computer programming exercises, respectively.

Chapter 5 discusses the Detective, an add-on designed to provide a unique type of additional feedback on students' submissions to computer programming exercises. The design and implementation of the system, as well as the types of feedback it generates and the means by which it does so, are all discussed in this chapter.

Finally, chapter 6 provides concluding remarks, as well as suggestions for future research.

In addition, Appendix A contains complete source-code listings for select modules from CAT-SOOP and the Detective.

Chapter 2

Design

2.1 Typical Interactions with CAT-SOOP

CAT-SOOP is designed with two separate groups in mind: students and instructors. Thus, in designing the system, it was important to consider the ways in which each of these groups would potentially want to interact with the system. The list of instructors' desired features was gathered directly from instructors, but the list of students' desired features was speculative.

Students were expected to interact with the system primarily by logging in, navigating to a specific assignment, submitting answers, and viewing the resulting feedback, as well as viewing the solutions when they are made available. In addition, it was anticipated that students would want to be able to view a concise summary of their performance on a given problem or assignment[1].

Instructors were expected to want to be able to navigate, view, and complete assignments just as students (for testing purposes), but without the restrictions of completing the assignments within a certain range of dates. From an administrative standpoint, instructors also wanted to be able to view a student's scores, or his entire submission history for a problem; to update or modify scores; to make submissions

[1]While easy from a technical perspective, this presented an interesting issue, primarily because of CAT-SOOP's philosophy on grading. It is easy for a student to get an incorrect impression that the score being displayed to him is his actual score in the course; to minimize this possibility, scores are explicitly reported as "Raw Scores," and no assignment averages are displayed.

for a student; and to edit problems, assignments, and course announcements.

2.2 Choice of Languages and Libraries

When beginning any new project, consideration must also be given to the tools on which that project is built, and how they relate to that project's goals.

For CAT-SOOP, one of the main factors driving the choice of implementation was ease of access and ease of use for students. The easiest way to ensure easy access to CAT-SOOP for all students was to make it a web-based tool, so that any student with a computer and an Internet connection can access the system without having to install any additional software on his machine.

Beyond this, one hope was that executing and checking code written in the Python programming language would be straightforward, and that the system would be easily extensible. For these reasons, CAT-SOOP is written in the Python programming language[2] (it is compatible with versions 2.6 and 2.7).

For reasons of familiarity, CAT-SOOP is built on the cherrypy web framework[3], and interacts with a MySQL database using the SQLAlchemy Python module[4].

Because it is designed for use primarily in technical subjects, the ability to display mathematical formulae in the web browser is a crucial feature. Near its inception, CAT-SOOP used a homebrew SVG-based system for rendering mathematical formulae; currently, however, the MathJax JavaScript library[5] is used to render math, for reasons of browser compatability and aesthetics.

The Detective add-on, described in detail in chapter 5, was written in PHP, JavaScript (with jQuery), and Python, primarily because it was built as an extension to a piece of software built on these technologies.

---

[2] `http://python.org`
[3] `http://www.cherrypy.org/`
[4] `http://www.sqlalchemy.org/`
[5] `http://www.mathjax.org/`

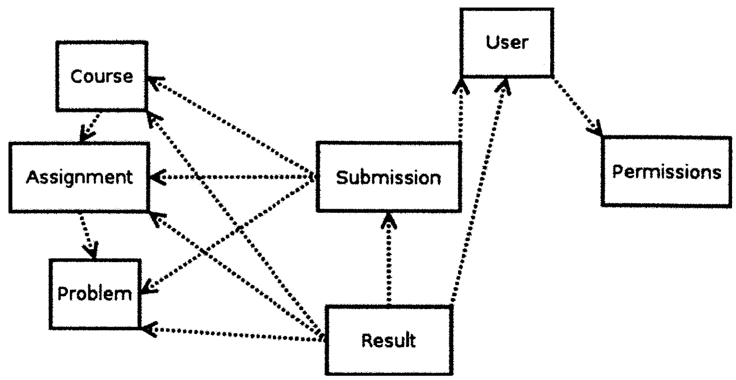


Figure 2-1: Graphical summary of the relationship between data structures in CAT-SOOP. Each line represents a "has-a" relationship.

2.3 Data Structures

This section describes the data structures used within CAT-SOOP. At times, the language in this section may shift back and forth between talking about objects in Python, and talking about entries in a MySQL database; it is worth noting here that each Python class described below (with the exception of the Question class) has an exact analog in CAT-SOOP's MySQL database, and so the concerns in each of these two realms will be lagely considered simultaneously.

2.3.1 Questions

Questions are central to the functionality of CAT-SOOP, as they represent requests for user input. Questions in the system each belong to a certain question type. These types are implemented as Python classes which inherit from a base Question class, and live in a specific location in the server's filesystem. Questions are never instantiated except as part of a Problem (see the following section regarding Problems), but CAT-SOOP keeps track of which Question Types are available in the system at all times.

Example question types which have been implemented for CAT-SOOP include: True/False, Multiple Choice, Short Answer, Numerical Answer, Symbolic Math (see chapter 3), Python Programming (see chapter 4), and PDF Upload.

Creating a new Question type amounts to making a new Python class which inherits from the base Question class, and has the following attributes and methods:

- attributes `name`, `author`, `email`, `version`, and `date`, which contain the problem's metadata, represented as strings.

- a method `get_html_template`, which returns a template for displaying the problem to the user, and can display blank problems, as well as displaying a previous submission back to a student.

- a method `checker`, which takes as input a solution and a submission, and returns a tuple of four elements: the fraction of this problem's points the supplied submission earned, feedback to be given back to the user, a header for the feedback, and a submission that should be referenced as the previous solution the next time this problem is loaded[6])

2.3.2 Problems

In the CAT-SOOP terminology, "Problems" are collections of Questions, accompanied (potentially) by blocks of descriptive text, figures, formulae, or other resources.

Each student is allotted a certain number of submissions per problem, as specified in the problem's description. He may continue submitting new answers (and receiving feedback on them) until he runs out of submissions, but may stop at any time before reaching that point. A student's score on his most recent submission to a given problem will be taken as his score for that problem (see section 2.3.5 for details about how this information is stored).

2.3.2.1 Specification Language

Problems are specified using an XML markup language which is designed to be easy to use. For the most part, this language is plain HTML, but with a few additional tags added:

[6]This usually ends up being the submission currently being handled, but was necessary to prevent some undesireable behavior in PDF upload problems. In future versions, this will be cleaned up, and a nicer way to handle such situations will be found.

- The entire problem description must be surrounded by `<problem></problem>` tags.

- Inline mathematical formulae are specified through the use of `<math></math>` tags.

- "Display" mathematical formulae are specified through the use of `<dmath></dmath>` tags.

- Questions to be asked as part of a given problem are specified through the use of `<question></question>` tags.

Figure 2-2 shows an example of a problem description specified in this markup language. Note that options in the outer `problem` tag specify how many submits each student is allotted for a given problem, and that options in the `question` tag specify the number of points that a given question is worth, as well as a valid solution.

Problems can be edited within the browser[7] by individuals with proper permissions (see section 2.3.4).

2.3.3 Assignments and Courses

Problems are further grouped into Assignments. Each Assignment contains a number of problems, and has three dates associated with it, which control access to the problems contained therein:

- A release date, after which problems in the assignment can be viewed and submitted.

- A due date, after which time problems are marked as late.

- A solution date, after which time students can view solutions.

[7]Currently, the only way to edit problems is through the browser; however, multiple instructors have expressed interest in editing problems in their own favorite text editors. Thus, in future versions, Problems may be removed from the database and instead live in the filesystem as plain-text files, so as to allow for easy editing.

```
<problem title="Mystery Feedback" maxsubmits="5">

Consider the following feedback system where <math>F</math> is the
system functional for a system composed of just adders, gains, and
delay elements:

<br /> <br />

<center>
<img src="static/figures/mystery_system.png" />
</center>

<br /> <br />

If <math>\alpha=10</math> then the closed-loop
system functional is known to be:
<dmath>\left.{Y\over X}\right|_{\alpha =10}=~~{1+R\over 2+R}</dmath>

Determine the closed-loop system functional when <math>\alpha=20</math>.

<br /> <br />

<math>\left.{Y\over X}\right| _{\alpha =20} =</math> 

<question type="expression" points="4">
<solution>(2+2R)/(3+2R)</solution>
</question>

</problem>
```

Figure 2-2: Example problem specification, including graphics, math, and a single question.

Assignments are further grouped into courses. At its core, a course in CAT-SOOP is little more than a collection of Assignments, just as an Assignment is a collection of Problems. However, courses also have associated with them a set of ranks, which define the actions that certain individuals associated with that course are allowed to take, as well as a field containing announcements, which are displayed on a course's main page within CAT-SOOP.

2.3.4 Permissions

User permissions are controlled on a per-course basis. Each course has its own set of permissions levels ("ranks" in the CAT-SOOP terminology), and a user's rank in one course in no way affects his rank (and, thus, his permissions) in another course. For example, a student might be participating in one course as a TA, but in another as a student; it is crucial that he is allowed to take certain actions in one course, but not in another.

The CAT-SOOP system contains 8 different permissions bits, each of which can be enabled or disabled independently of the others:

1. "View" allows a user to view course materials as they are released.

2. "View Always" allows a user to view all course materials, regardless of release date. If a user's "view always" bit is set, his "view" bit is ignored.

3. "Submit" allows a user to submit solutions to problems, subject to release dates, due dates, and submission limits.

4. "Submit Always" allows a user to submit solutions to problems, regardless of time or submission limits. If a user's "submit always" bit is set, his "submit" bit is ignored.

5. "Grade" allows a user to edit other users' scores, and impersonation of other users (as described in section 2.4).

6. "Edit" allows a user to edit course materials, including release and due dates.

7. "Enroll" allows a user to add new users to a course, regardless of whether the course registration is open.

8. "Admin" allows a user to edit other users' permissions within the course, and open or close the course or registration.

Finally, each user has a single permissions bit (called the "in charge" bit) which, if set, allows him to modify global system settings.

2.3.5 Submissions and Results

CAT-SOOP's main goal is to facilitate the automatic collection and assessment of homework exercises. As such, it is important that the system keep a record of students' submissions to problems. In CAT-SOOP, this is handled by means of the Submission class.

Whenever a student makes a submission, a new instance of the Submission class is created, which contains the student's entire submission. Thus, every answer he ever submitted exists in the database in its entirety, along with the score he received on it. This information is useful for reviewing a student's peformance on a problem over time (for, e.g., assigning partial credit to a problem, or verifying a student complaint about faulty checking[8]).

Each student may have multiple Submissions for each problem he opens. With so many Submissions in the database, however, a need quickly arises for a sort of summary of a student's performance on a given problem, to avoid searching through numerous Submission objects to find the proper one, for scoring or for display of a problem; this is where the Result object comes in.

Each user has one Result object per problem. This object contains a reference to his most recent submission, as well as information about his current score. When he

[8] In systems where information about students' previous submissions is not stored, this can be a real pain. Firstly, there is no way to verify whether a student is telling the truth, and secondly, it can be very difficult to re-create (and subsequently fix) a checking error without knowing what exactly was submitted.

opens a problem, this Result object is loaded, and his previous responses and score (as gathered by loading his most recent submission, if any) are shown.

2.4 Grading and Impersonation

When an instructor views a student's submissions, he has the option of requesting only the student's most recent submission for that problem, or the student's entire history of submissions. He also has the ability to modify a student's score while viewing that student's submissions. When he does so, the student's original score remains in the database, but is augmented with information about the updated score, as well as the user who assigned him that score. Thus, when a problem is loaded for which a student has been specifically assigned a score by staff, that score will appear; for problems for which he has not been assigned a specific score by staff, CAT-SOOP's automatically-generated score will be displayed instead.

Staff may also want the ability to "impersonate" students. Impersonation is handled very differently in CAT-SOOP than in xTutor and tutor2. Both xTutor and tutor2 allow persistent impersonation in the sense that a user can impersonate a student for some duration of time, during which the system will behave as though he is the student he is impersonating. In xTutor, when one impersonates a student, a complete copy of that student's data is created and used as the impersonator's data until he is done impersonating the student. This gives the impersonator the freedom to do as he pleases while masquerading as the student, with no possibility of impacting the student's actual state in the system. In tutor2, when one impersonates a student, the system simply treats all actions he takes as though they had been taken by the student he is impersonating. This means that the impersonator can modify a student's state in the system if he so desires (or by accident, if he is not careful). Both of these schemes have positives and negatives associated with them, and neither is a clear-cut "better" solution.

CAT-SOOP does not allow persistent impersonation. Instead, a staff member may make submissions as a user if he needs to (or wants to). The staff member does not "become" the student in the system's eyes, but any submission he makes in this fashion will be treated as though it were made by the student (although the submission is stored with additional information about who actually made it[9]).

[9]Another design goal of CAT-SOOP worth mentioning is that all important actions should be logged. Every submission, entry of grades, modification of problems, etc, should result in something being logged to the database. Having this information makes retrospection (in the event of a complaint, or a system failure) possible. xTutor keeps an even more detailed log, including every page load. tutor2 does the same, but misses some important information when logging students' submissions to problems.

Chapter 3

Evaluating Symbolic Math

CAT-SOOP underwent a pilot test in MIT's *6.003 Signals and Systems* in fall term 2011, where it was used almost exclusively to assess students' responses to mathematical questions. One easy way to approach this problem would have been to force the instructors to phrase all of the questions they wanted to ask in forms already allowed in the base system (e.g., instead of asking for a symbolic expression, ask for a numerical answer corresponding to that expression evaluated with certain values for each variable).

However, this seemed particularly restrictive, and so CAT-SOOP's symbolic math checking routines came to be. Currently, the system is capable of checking two main types of symbolic math: symbolic expressions, and numerical ranges, which are discussed in detail in the following sections. An example of CAT-SOOP's display during the solving of these types of problems can be seen in figure 3-1.

3.1 Mathematical Expressions

Appendix A (section A.1.1) contains the full source-code listing for `expressions_ast.py`[1], which is responsible for handling symbolic expressions in CAT-SOOP.

[1]This style of checking is used in both CAT-SOOP and tutor2, so it exists as a stand-alone module.

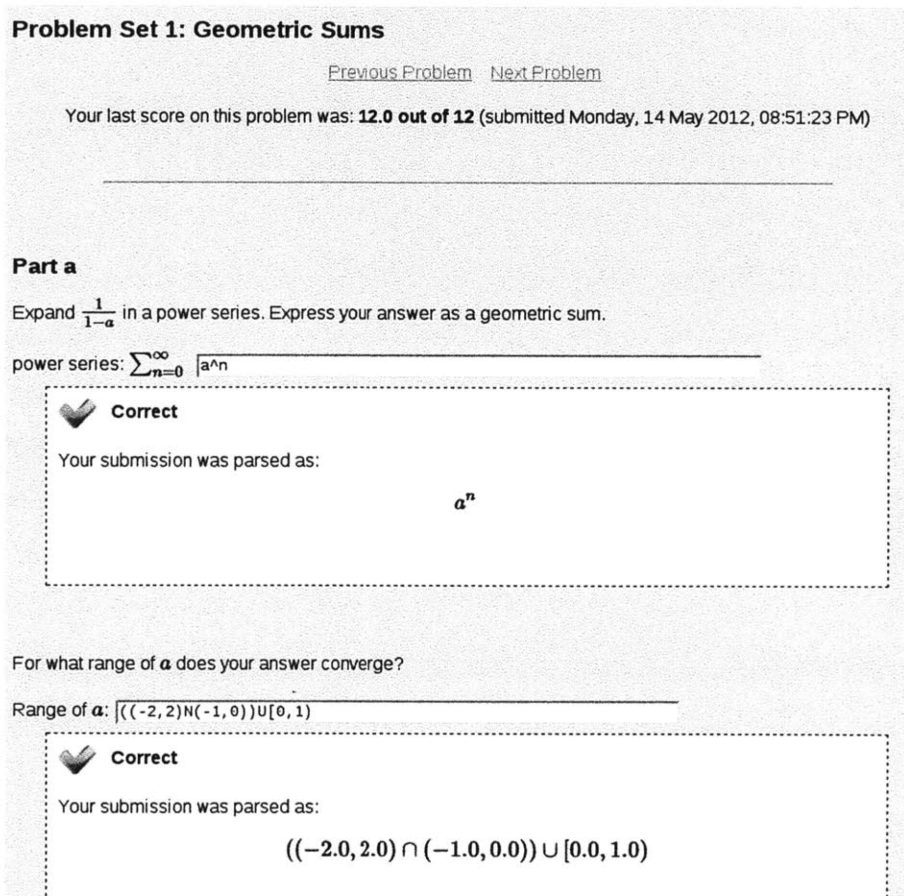


Figure 3-1: Screenshot showing CAT-SOOP's display of a simple symbolic math problem involving multiple parts.

3.1.1 Testing

The procedure for testing correctness of symbolic expressions has gone through several iterations. At first, CAT-SOOP made use of a symbolic math library for correctness checking. However, this approach was found to be lacking, particularly when checking complicated expressions. For example, checks involving complex exponentials or trigonometric functions tended to eat up a lot of CPU time (and could possible enter infinite recursions, forcing a restart of the server), and were not always accurate[2].

Because of these limitations, and the general difficulty of symbolic equivalence checking, CAT-SOOP currently does all its correctness checking numerically, which has proven in practice to be very efficient and accurate when compared against the symbolic approaches used before. The checking process unfolds as follows:

1. The given submission and solution are both parsed down into Python AST's[3].
2. Each variable that appears in at least one of the two expressions is assigned a numerical value (a random complex number within a certain range)[4].
3. Each AST is evaluated in the Python environment containing the variable bindings created in step 2.
4. These numbers are compared to one another; if they are within a certain threshold of one another, they are assumed to be equivalent expressions.

3.1.1.1 Errors in Checking

This method is not guaranteed to produce correct assessments, and both false positives (marking incorrect submissions as correct) and false negatives (marking correct solutions as incorrect) are possible.

[2]These flaws were responsible for some student frustration early on in *6.003*, when this checking scheme was still in use.

[3]It is worth noting here that, while this step relies on expressions being specified using Pythonic syntax, it is certainly possible to allow input languages other than Python, through the use of pre-processors which translate from the desired input language into Python.

[4]Currently, four variable names are reserved, and assumed to have special meaning: `j`, `e`, `abs`, and `sqrt`. If these variables appear within an expression, they are not assigned random values, but are interpreted as the imaginary unit, the base of the natural logarithm, the absolute value function, and the square root function, respectively.

Of the two types of errors, false positives are more likely, and could occur in the case where the randomly-generated numbers happen to cause the evaluation of the incorrect submission to be close enough to the evaluation of the correct solution. In practice, this rarely occurs with a sufficiently wide distribution over values which variables can take, even with threshold values as forgiving as 10^{-4}, and can be guarded against by running the above procedure n times, and only marking solutions as correct which pass all n tests (the false positive rate decays exponentially with n).

False negatives are also technically possible, but are extremely unlikely (even compared to false positives), to the extent that they can be largely ignored. Since the checker uses the same initial values for each variable, the only apparent way that a correct submission's evaluated value can diverge from that of the solution is through rounding error. While it is technically possible for this type of divergence to happen (particularly with a small enough threshold value), it is not a practical concern[5].

3.1.2 Feedback

Currently, the symbolic math system provides very limited feedback. The only type of feedback currently offered is a LaTeX representation of the user's input (see figure 3-1), which is useful for catching entry errors, but not terribly useful for catching conceptual errors.

3.1.3 Looking Forward

One idea for improving the feedback generated about students' submissions to symbolic math questions is to use solution-specific feedback, wherein common incorrect solutions to a problem are collected, and solution-specific canned responses are displayed to students whose answer takes one of those forms. The CyberTutor[17], an automatic tutor for introductory physics, uses this idea of feedback, and also offers feedback if the student's solution contains a variable not present in the solution, or

[5]In fact, tests involving exponentiation, as well as repeated multiplication and division, to try to introduce rounding error were never able to introduce enough error to create a false negative (with a threshold of 10^{-9}) without first running into limitations in Python's parser, or overflow errors.

vice versa (e.g., "the solution does not depend on x").

The CyberTutor also makes use of a type of proactive feedback through hints. Students are presented with a variety of hints, which are basically steps leading up to the solution. The student may ignore the hints, but if he gets stuck, he may open a hint, which could potentially push him in the right direction. An internal report by Warnakulasooriya and Pritchard[22] suggests that these hints are beneficial.

Another idea would be to systematically apply deformations to the AST which results from parsing down a submitted expression, to see if the solution can be reached; trees could be deformed, for example, by replacing nodes representing trigonometric functions with other trigonometric functions, or by negating nodes representing numbers or variables. If any combination of these deformations (and, potentially, other, more complex deformations) results in a tree that is equivalent to the solution, then targeted feedback can be given (e.g., "check your signs" if a negation caused the submission to become correct).

3.2 Ranges

In addition to checking symbolic expressions, CAT-SOOP is able to check numerical ranges. These questions are often follow-ups to symbolic expression questions, as can be seen in figure 3-1.

Appendix A (section A.1.1) contains the full source-code listing for `Range.py`, which is responsible for handling ranges in CAT-SOOP.

3.2.1 Testing

As with the symbolic expression checker, the range checker has gone through a number of changes since it was first used. Initially, input was given as a Pythonic boolean expression (for example, $|x| = 2$ could be specified as `(abs(x) == 2)`, or as `(x == -2 or x == 2)`, among other possibilities). This syntax proved tedious, however, for people with little or no programming background, to whom it felt like an unnatural way to represent ranges.

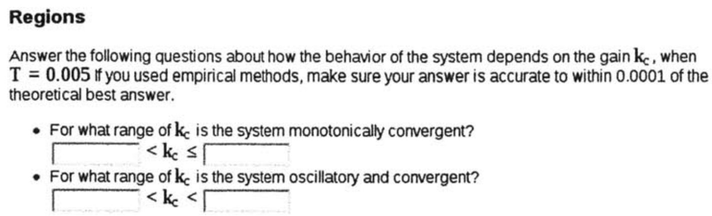


Figure 3-2: Screenshot showing range checking in tutor2, which is similar to a previous version of CAT-SOOP's range checking. Answers are given as two numbers: a lower bound, and an upper bound.

In this original scheme, checking was accomplished by randomly sampling a large number of points over some specified range, and checking whether each of those values of the variable in question caused the solution and the submission to resolve to the same answer (either `True` or `False`). If all of the points resulting in the submission and the solution resolving the same answer, then the submission was marked as correct. If they did not match, then the submission was marked as incorrect.

Obviously, this approach is not perfect; as with the method described for checking expressions, it has the potential to generate false positives (in the sense that it may mark incorrect submissions as correct), but will not mark any correct submissions as incorrect. Despite its inelegance, this approach has proven to do an adequate job of assessing student submissions in practice, and increasing the number of sampled points are tested would increase the accuracy of the checker in general.

The next iteration of the range checker required two numerical inputs per range: one for a lower bound, and one for an upper bound; a similar method is used in tutor2, as can be seen in figure 3-2. The benefit with this method was that checking was straightforward. However, phrasing questions in this manner limited the types of ranges which could be specified and the freedom of the instructors to write arbitrary problems.

Currently, the range checker uses the same testing methodology as the original Pythonic range specification, but also checks the boundaries of each region specified in either the solution or the submission. What has changed is the language used to specify ranges. Currently, the checker accepts input in a simple language designed for the sole purpose of representing regions of the number line. A single region is represented in a typical fashion: as an ordered pair delimited by brackets, where a round bracket implies that a boundary is exclusive, and a square bracket implies that a boundary is inclusive; for example, `(0,3]` includes all positive real numbers x such that $0 < x \leq 3$. Positive and negative infinity are specified as `INF` and `-INF`, respectively.

These regions can be combined through the use of two operators: `N`, which represents an intersection ($\cap$), and `U`, which represents a union ($\cup$).

This last method is CAT-SOOP's current method of choice, though from examining these three schemes, it should be apparent that each has its own strengths and weaknesses. Depending on the context and the specific question being asked, any of these three options might be favorable.

3.2.2 Feedback

Similarly to symbolic expressions, the only feedback CAT-SOOP currently gives about a student's submission, aside from whether it is correct, is a LaTeX representation (see figure 3-1) of the submission. Once again, while this is useful for detecting entry errors, it offers little in the way of conceptual feedback.

The representation into which ranges are parsed is not as rich as an AST, and so, unfortunately, many of the interesting ways to improve feedback for expressions simply do not translate to ranges.

Chapter 4

Evaluating Computer Programs

One of CAT-SOOP's primary objectives is to automate the assessment of student-submitted computer programs. Because CAT-SOOP was designed for use at MIT, and Python is the language of choice in MIT's undergraduate curriculum, CAT-SOOP is currently only capable of assessing programs written in the Python programming language; despite this, the methods described in this chapter and the next will hopefully prove, at least to some extent, generally applicable, and extensible to other programming languages.

4.1 Subset of Python

CAT-SOOP's current means of assessing and providing feedback on students' submissions to programming exercises consists of a number of components, each of which places some constraints on the subset of the Python language which can be successfully and completely assessed.

The core testing system, which is built into the CAT-SOOP system, allows for almost the complete Python 2.7 language, with the exception of certain blacklisted statements (see section 4.2.1). However, the myriad components of the Detective add-on (described in the following chapter) create additional, more severe constraints[1].

Explicitly allowed in the subset are:

- Booleans, Integers, Longs, Floats, and Complex Numbers
- Lists and Tuples
- Dictionaries
- For and While Loops
- Conditional Statements
- User-Defined Functions

Explicitly disallowed in the subset are:

- Multiple Assignment
- File Handling
- Yield Statements and Generators
- Imports
- Sets
- try/except/finally
- In-line conditional statements
- Slicing

Because the system really does consist of several disjoint pieces, the effects of using some of the above statements may be more benign than others.

[1]The aim here is to create a rich subset of the Python programming language, while still keeping it simple enough that meaningful feedback can be generated. Ideally, CAT-SOOP and the Detective will eventually be able to allow a more complete subset of Python. If the additional feedback afforded by the Detective is not a concern, the core system can still be used, which is capable of checking a much more complete subset of the language; in this case, the allow/deny lists above may be ignored.

4.2 Testing

Checking arbitrary programs for correctness in an absolute sense is an extremely difficult task, and so CAT-SOOP falls back on a method commonly used in automatic programming tutors: test cases. In particular, the code checking in CAT-SOOP is largely based off of similar systems used in the xTutor and tutor2 automatic tutors. Although details are ommitted here, appendix A (section A.1.2) contains the complete source-code listing for `pysandbox_subprocess.py`, which houses most of the code described in this section.

When a student's submission is checked for accuracy, it is run through a number of test cases, and the results of these executions are compared against the results of running a solution through the same test cases. Assuming an adequate battery of tests and a correct solution, then any submission which passes all the same test cases as the solution can be considered a correct submission.

Each programming question specifies a list of test cases, as well as (optionally) a block of code to be executed before running the submitted code (e.g., to define functions or variables which can be used in the student's submission). Each test case consists of an arbitrary number of statements, which ultimately set a variable **ans**, which is the end result of the test case. Once the student's code and the test case have been run, a string representation of **ans** is stored in a specific location. This process is repeated for each test case, and for the solution code.

Once all test cases have been run on both the student's code and the solution, the results of each test case are compared against one another. By default, the strings are compared against one another verbatim, but an arbitrary Python function may be used to compare the two (e.g., by converting each to a Python object, and then comparing those objects), which increases the variety and complexity of the checks which CAT-SOOP can perform.

4.2.1 Security

Allowing arbitrary pieces of code to run on a public web server is a dangerous prospect. CAT-SOOP's approach to avoiding executing dangerous code involves simply checking whether the submitted code contains any of a number of "blacklisted" statements, which are deemed dangerous either to the state of CAT-SOOP system, or of the machine on which it is running. This check is performed after stripping away all comments and whitespace (as well as the line continuation character \), so that formatting tricks cannot allow these statements to pass through.

Any code which contains any of these statements is not executed, and causes an e-mail to be sent to any user whose "Admin" bit (see section 2.3.4 for a discussion of permissions within CAT-SOOP) is set for the course in question; this e-mail contains the raw code submitted to the system, as well as the username of the individual who submitted the code.

To guard against infinite loops, Python's `resource` module is used to limit each test's running time to two seconds. Any code running for longer than two seconds is assumed to have entered an infinite loop.

While these measures certainly do not constitute a perfect means of sandboxing user-submitted code, they should provide a reasonable level of security nonetheless.

4.3 Feedback

The core system provides very simple feedback, letting the user know whether his code passed each of the test cases. However, Michael[14] suggests that students learning to solve problems benefit from feedback beyond a simple assessment of the correctness of their answer. Automatically generating meaningful feedback for arbitrary programs submitted by students is, in general, a very difficult problem, but one which CAT-SOOP seeks to address through the means of an add-on called the Detective.

The following chapter describes this system, which is aimed toward increasing students' understanding of how the state of a program evolves during a single execution, in detail.

Chapter 5

The Detective

CAT-SOOP focuses mainly on providing feedback about a single execution of a student's program. To this end, the Detective was developed. The Detective is a piece of software designed to provide detailed information about how the state of the execution environment changes as a program runs, as well as to provide insight into why and when errors occur during execution.

The use of run-time tracing in automatic tutors has been investigated by Striewe and Goedicke[21], who suggest that tracing in automatic tutors can be beneficial (in particular because it allows for easily generating certain valuable types of feedback which would be very difficult to generate without tracing), but also that there is much room for improvement in this regard. The goal of the Detective is to use run-time trace data, as well as syntactic information, to generate meaningful, concrete feedback about students' submissions to introductory programming exercises, and thereby increase students' power to solve programming exercises autonomously.

5.1 Tracing and Visualization

At the Detective's core is a visualization of the evolution of a program's environment as it is executed. This visualization is based on (and uses much of the original code for) Philip Guo's Online Python Tutor[1]. Guo's Tutor contains a tracer (`pg_logger`

[1] `http://people.csail.mit.edu/pgbovine/python/`

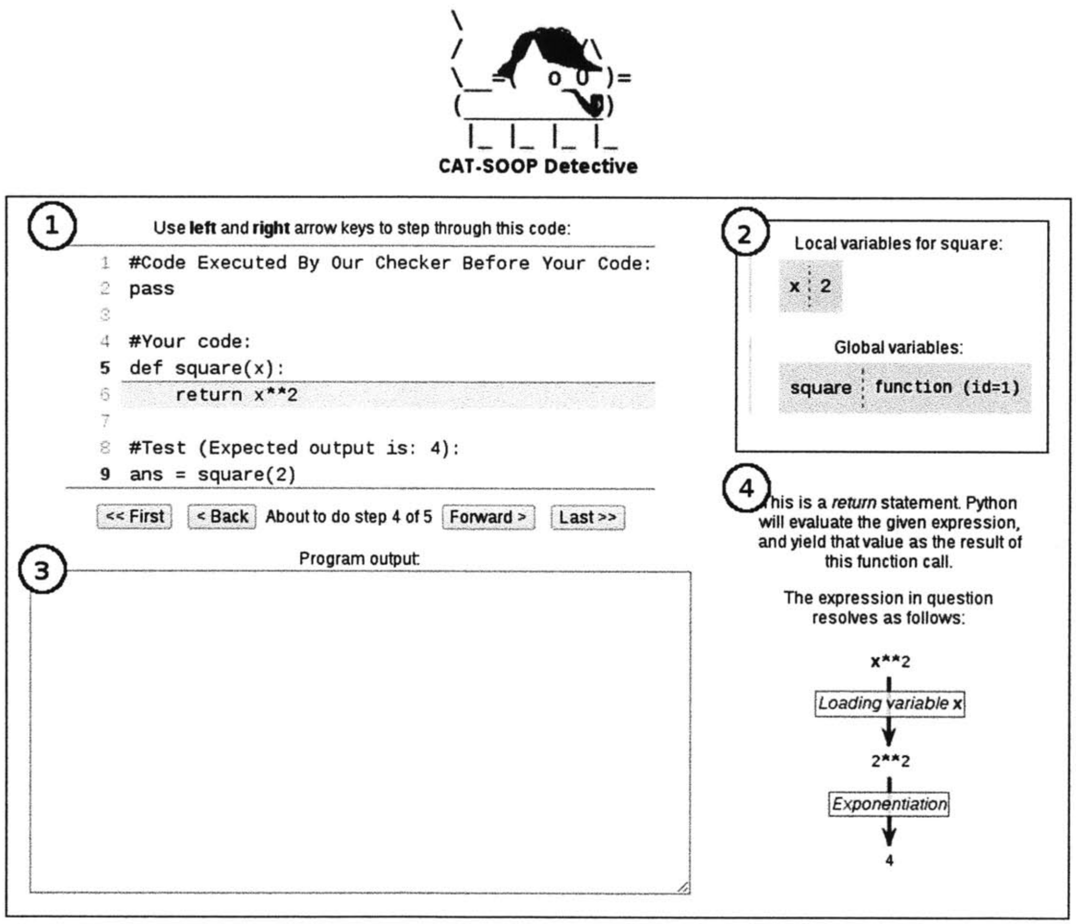


Figure 5-1: The user interface to the Detective, showing **(1)** the submitted code, **(2)** the current local and global variables, **(3)** the output from the program so far, **(4)** and an explanation of the current line's purpose.

Type of Error	Example Explanation
Name not defined	This message means that the program is trying to access a variable called `foo`. However, there is no such variable in the current scope. If this is the correct variable name, make sure it has been initialized first. If not, did you mean to use one of the following variables? `Foo`, `fo0`
Object unsubscriptable	Grabbing a single element from a collection using square brackets (`[]`) is referred to as *subscripting*. This message means that the program is trying to subscript something that can't be subscripted (a **`function`**). If you intended to call this function, you should use parentheses instead of square brackets.
Object not callable	Executing the code stored within a function using round brackets (parentheses) is referred to as *calling* that function. This message means that the program is trying to call something that can't be called (a **`list`**). If you intended to index into this **`list`**, you should use square brackets (`[]`) instead of parentheses.
Operation not supported	This message means that the program is trying to combine two objects using an operator, but doesn't know how to do so. Specifically, this line is trying to combine an **`int`** and a **`str`** using the + operator, which is not supported.

Figure 5-2: The Detective's explanations of various types of run-time errors.

by name), which logs information about the evolution of local and global variables, as well as information relating to Python exceptions, over the course of a single execution of a program.

Guo's Tutor allows users to "step" through the program's execution line-by-line and observe how the program's internal state evolves.

The Detective uses a slightly-modified version of Guo's tracer (dubbed `hz_logger`), which includes syntactic information in the form of partial AST's, to augment this visualization with interpretations of error messages (as described in section 5.2), as well as expanded explanations of program behavior (section 5.4) and more finely-grained resolution information (section 5.4.1).

5.2 Error Analysis

While valuable to the expert programmer who has learned to interpret them, error messages present a challenge to the novice programmer. Most error messages are

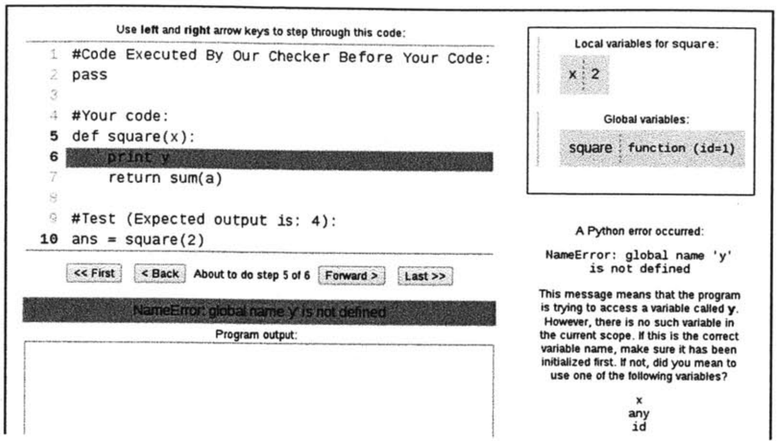


Figure 5-3: A screenshot of the Detective displaying an error message, along with an interpretation of that error message.

strangely worded, and even the more straightforward error messages are often buried in a pile of red text which can be intimidating, particularly to those just beginning with programming.

Many students have trouble interpreting these error messages, and thus require explanation as to what an error message means before they are able to go about trying to fix it.

The error analyzer tries to alleviate this problem by providing simple explanations of common error messages in plain English. The original error message generated by Python is still displayed, but is augmented by a simple explanation of what the error message means, in the hopes that students will begin to connect the simple explanation with the error message that Python generates, so that they will be better able to interpret such error messages when they are no longer working within the Detective.

The method by which these responses are generated is rather simplistic, but still provides meaningful, relevant interpretations of error messages; these messages are generated by considering the error message generated by Python, as well as the state of the local and global variables when the error occurred. Using this information, the Detective fills in an explanation template specific to the type of error encountered. Sample explanations for a few common types of errors can be seen in figure 5-2.

What follows is a description of several common errors students make, as well as the ways in which the Detective identifies and explains them. Some of these items are the Python equivalents of common Java mistakes enumerated by Hristova, et al[7] and Lang[12]; others on this list came from personal experience interacting with novice programmers, and from several semesters worth of xTutor's logfiles.

The complete source-code listing of `errors.py`, which contains the code for interpreting error messages, can be found in Appendix A, section A.2.1.

5.2.1 Common Run-time Errors

1. **Misspelled Variable Names** — Misspelling variable names is one common error. Even for an experienced programmer, a slip of the finger can result in a Python `NameError` stemming from a typographical error. For a novice, these errors are likely to be harder to understand, and to diagnose (for example, the idea that `Foo` and `foo` are different names in Python takes a little getting used to). When the Detective encounters a "name not defined" error, it displays a canned response explaining that the variable in question is not defined in the current scope. In addition, the system searches in the current scope (including Python's built-in variables and functions) for names that closely resemble the name the user typed in. These variable names are found by iterating through the current scope (+ built-ins), and computing the Damerau-Levenshtein distance[3] between the specified variable name, and each variable actually defined in the current scope. A list of those variables whose Damerau-Levenshtein distance to the specified variable name is less than or equal to two is displayed back to the user, as can be seen in figure 5-3.

2. **Incorrect Choice of Braces** — Novices will often confuse square brackets with parentheses, attempting to call a function with the syntax `foo[x]` or to index into a list with the syntax `foo(ix)`. The detective catches these types of errors by investigating certain `TypeErrors` (specifically those which are accompanied by an error message stating that a certain object is not subscriptable, or is not

callable). If a user tries, for example, to subscript a `function` object using square brackets, the Detective offers a suggestion to use parentheses instead of square brackets. Similarly, attempting to call a `list`, `tuple`, or `dict` object using parentheses will result in the Detective suggesting to use square brackets instead.

3. **Unsupported Operations** — Another common error is confusing types. This usually manifests itself when the user tried to perform some operation on an object, which its type forbids. One common error of this kind is attempting to add together two objects of differing types (e.g., `24 + '2.0'`). This error can manifest itself as an "unsupported operand types" error message[2]. In this case, the Detective gives a canned response, with some information injected about this specific instance of the error message.

4. **Index Out of Range** — When just starting with programming, most people are used to counting from one, and so Python's zero-indexing of lists and tuples can be a stumbling point, even if it is not a conceptually difficult concept. The Detective responds to "index out of range" errors with a simple canned response, a reminder about counting from zero and valid indices.

5.2.2 Pitfalls

The Detective's error checking goes beyond reporting actual exceptions to warn users about common mistakes in Python which don't necessarily cause exceptions, but might lead to unexpected behavior. Because they don't necessarily cause Python exceptions to occur, these cases are handled separately from other error reporting. Python has a few of these "pitfalls" (to borrow terminology from Lang), some of which are enumerated below:

[2]These errors can also manifest themselves in other ways, with a wide variety of error messages, depending on which of the operands is given first. Additionally, `AttributeErrors` might arise from misunderstanding types. As a proof-of-concept, the Detective currently only explains those errors of this kind which give rise to this specific error message; however, it could easily be extended to account for those other cases.

1. **Exponentiation Syntax** — Novices with backgrounds in mathematics, as well as experiences programmers who are new to Python's syntax, tend to want to use a caret (^) to denote exponentiation, when in Python this represents bit-wise exclusive or (XOR). Since students are more likely to be called to use exponentiation than XOR in introductory programming exercises, the Detective gives a warning whenever this operator is used. An example of such a warning is:

   This line contains a caret (^), which represents a bitwise XOR operation. If you intended to use exponentiation, use two asterisks (**) instead.

2. **Overwriting or Hiding Built-in with Variable** — One subtle pitfall is the possibility of overwriting or hiding built-in objects in Python through assignment statements. Many built-ins have names which are desirable for variable names; in particular, the type names (among them `list`, `str`, `dict`), as well as `max` and `min`, tend to be overwritten frequently, and this is a common occurrance for other built-in variables as well. Any time the Detective encounters an assignment statement which gives a warning whenever an assignment overwrites or hides a built-in variable. An example of such a warning is:

   This line contains an assignment to a variable named **int**. *However,* **int** *is also the name of an object built in to Python. This assignment will "hide" the built-in object, so that it will not be accessible from within this function.*

Additional pitfalls were considered, including leading zeros on integers (which are interpreted as octal numbers in Python), and using `&` and `|` instead of `and` and `or` in boolean expressions. However, both of these concepts are difficult to explain concisely without assuming a background in mathematics or computer science, and so are not considered in the current version of the Detective.

5.3 Syntax Errors

Syntax errors in Python are particularly hard to diagnose and fix. Novices tend to make a lot of mistakes when programming, which cause Python to be unable to execute their code. Many novice errors are greeted with a familiar (and really unhelpful) message: `SyntaxError: invalid syntax`. Because of this, novices tend to spend a lot of time staring at code that will not run, trying to figure out where their errors lie.

Thus, an ideal automatic tutor would be able to provide insight into why syntax errors, in addition to run-time errors, occur. However, the problem of identifying the causes of syntax errors is intrinsically more difficult than analyzing run-time errors, if for no other reason than that syntax errors disallow the possibility of investigating Abstract Syntax Trees, forcing consideration instead back to the level of textual source code.

As it currently stands, the Detective does not make any attempt to analyze or explain syntax errors, although such analysis is certainly a goal for future versions, as the potential gains are substantial.

5.4 Statement Explanation

In addition to the providing interpretations of error messages, the Detective also incorporates a system which attempts to explain what each line of a student's program is doing as it executes. This system, hereafter referred to as the explainer, is very simplistic, but may provide some clarity (or at least a useful reminder) as to what a given line will actually do when executed; this information is likely most useful for people just getting started with programming.

The explainer basically maps AST node types to canned explanations, with some small variation depending on the structure of the AST rooted at the node in question. For example, a `return` statement with no return value specified will generate an explanation similar—but not identical—to a `return` statement with a return value specified. Announcements are also made when entering (via a function call) or exiting (via a `return` statement or reaching the end of a function's definition) a function. This scheme is admittedly simplistic, but should at least serve as a proof-of-concept for future systems. Table 5-4 shows examples of generated explanations for several types of Python statements.

When appropriate, these simple explanations are augmented by more finely-grained information about how a given expression resolves; these messages, and the method by which they are generated, are described in detail in the following section.

5.4.1 (Pseudo-) Instruction-Level Resolution

When a student's program begins producing unexpected results, he is often pointed to a specific line of code where the error occurred, but from there, he is left on his own to figure out where, specifically, his error lies. Often, a line of code consists of several instructions; because of this, it can be difficult to determine when during that line's execution the program started to deviate from what the programmer intended. This is particularly true in cases when a program runs successfully (in the sense that it runs through to completion without generating a Python error) but nonetheless produces incorrect results.

For this reason, the Detective seeks to provide finely-grained information about how a given expression resolves. Other program visualizations (such as jEliot[16]) accomplish similar goals by investigating a program's bytecode. However, a quick inspection of Python's compiler showed that it makes some optimizations at compile time that could prevent the Detective from giving a complete picture of how a line resolves[3].

As an alternative, the Detective uses a system which resolves Abstract Syntax Trees step-by-step. This method I call (pseudo-) Instruction-Level Resolution (hereafter pILR). The underlying idea is that by resolving an AST step-by-step in a sys-

[3]While the only optimization I directly observed involved pre-computing additions (e.g., 2+3 compiled to `LOAD_CONST (5)`), seeing this early on made me wary of using bytecode, which might make use of other optimizations that could potentially impede the Detective's ability to show every step of a resolution.

Type of AST Node	Example Explanation
`Assign`	This is an *assignment* statement. Python will evaluate the expression on the right-hand side of the equals sign, and will store the resulting value in variable `x`.
`Break`	This is a *break* statement. If it is given inside of a loop, this statement will cause Python to jump outside the loop, skipping the rest of the code block for this iteration and all subsequent iterations. If given outside of a loop, this statement will cause an error.
`Continue`	This is a *continue* statement. If it is given inside of a loop, this statement will cause Python to jump to the top of the loop, skipping the rest of the code block for this iteration. If given outside of a loop, this statement will cause an error.
`For`	This is a *for* loop. Python will run the given code block once for each element in `foo`, each time setting a variable `i` equal to the next element in `foo`.
`FunctionDef`	This is a *function definition* statement. Python will store this function in variable `foo` so that it may be called later.
`If`	This is an *if* statement. Python will evaluate the given expression. If it evaluates to `True`, Python will jump to line x; if it evaluates to `False`, Python will jump instead to line y
`Pass`	This is a *pass* statement, which tells Python to do nothing.
`Print`	This is a *print* statement. Python will evaluate the given expression, and display it to the console.
`Return`	This is a *return* statement. Since no expression was given, Python will yield `None` as the result of this function call.
`While`	This is a *while* loop. Python will evaluate the given expression. If it evaluates to `True`, Python will jump to line x, execute the code in that block, and return here to check the expression again. If it instead evaluates to `False`, Python will skip this code block altogether.

Figure 5-4: The Detective's explanations of supported types of Python statements.

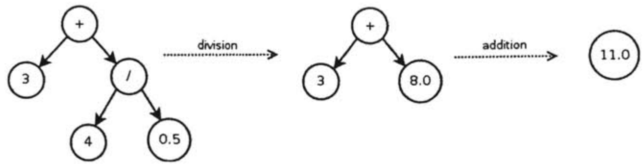


Figure 5-5: pILR trace of `3 + 4 / 0.5`, depicted as partial AST's

tematic manner, one can mimic the process by which Python would evaluate an expression, and explore the evolution of that expression as it resolves. Figure 5-5 shows an example of a simple pILR trace.

Each type of AST node[4] resolves in a specific way, and provides a specific message stating what is being done as it resolves (for example, a `Name` node, which represents loading a variable, is accompanied by a message "Loading variable x."). The specifics of each type's resolution, which are naturally motivated by the ways in which Python evaluates different types of expressions, will not be discussed here in detail, but Appendix A (section A.2.5) contains a complete source-code listing for `resolution.py`, which contains the pILR code.

As mentioned before, the main motivation in developing the pILR system was to provide information to students about when, specifically, errors occur during the resolution of a line of code. Thus, the pILR scheme must have a means of dealing with Python errors which occur mid-line, and still be able to provide a partial trace when these types of errors occur. To this end, the pILR system makes use of a special `Error` node during resolution. In the case where an error occurs when resolving a sub-tree, the error node replaces whatever node would have resulted in the case of a successful resolution. Different types of AST nodes check for errors in subtree resolution at different times, but the ultimate end result is that the `Error` node propagates up the tree; this may preclude the resolution of sibling nodes, but will not interfere with those resolutions which were completed successfully before the error occurred. Figure 5-6 shows an example of this behavior in a simple context.

Not only is pILR capable of creating finely-grained traces of the resolution of a

[4]Currently supported are `BinOp`, `BoolOp`, `Compare`, `Dict`, `List`, `Name`, `Num`, `Str`, `Subscript`, `Tuple`, and `UnaryOp`.

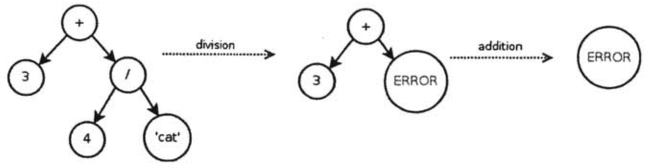


Figure 5-6: pILR trace of `3 + 4 / 'cat'`, depicted as partial AST's, and demonstrating the propagation of an `ERROR` node.

number of different Python expressions, but is seems to have an additional benefit over creating these traces from compiled Python bytecode: pILR maintains, at all times, an explicit representation of the current state of the resolution, in the form of a Python AST. This representation is currently used to create the Detective's visualization of pILR traces; the Detective walks these partially-resolved AST's to create Python code which, when parsed, would generate the AST in question; this Python code is then used in the Detective's visualization.

The Detective uses the jsPlumb JavaScript library[5] to connect the partially-resolved AST's, and to give brief descriptions of what each step in the trace is doing; figure 5-7 shows the resolution of a more complicated example as it appears within the Detective, from a student's (correct) submission to a question asking for a program to compute the roots of a quadratic expression.

5.5 Connecting with CAT-SOOP

Because the Detective exists as a stand-alone web application, some care had to be given to connecting it with CAT-SOOP in a reasonable way.

The connection is made through a modified version of the Python Code question type[6], called `PythonCodeViz`. When a `PythonCodeViz` question is submitted, the submission is checked for correctness in the usual manner, as described in section 4.2. In addition, several versions of the code (one for each test case) are sent via HTTP POST request to a CGI front-end to `hz_logger`, which generates a JSON

[5]`http://jsplumb.org/jquery/demo.html`

[6]see chapter 4 for a discussion of this question type, and section 2.3.1 for a general discussion of question types within CAT-SOOP

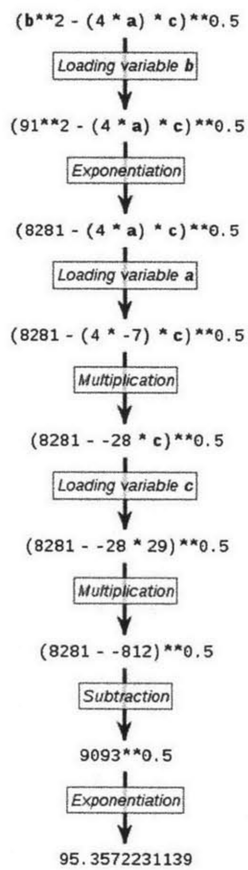


Figure 5-7: The pILR trace of calculating a determinant, as visualized in the Detective.

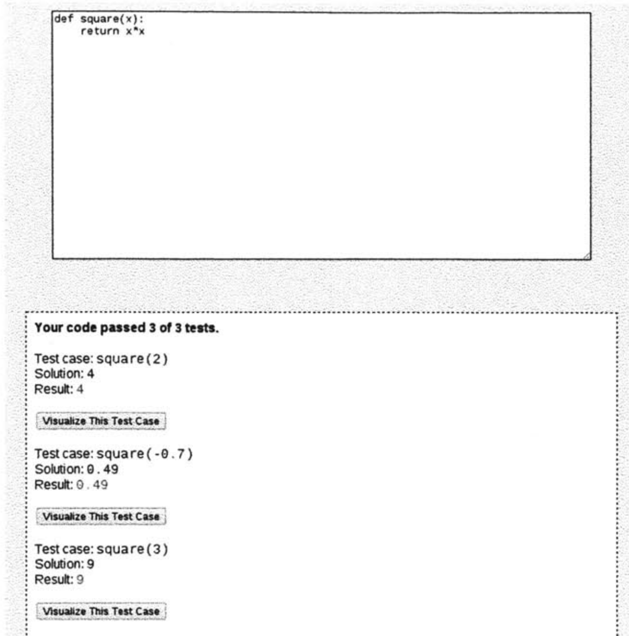


Figure 5-8: Screenshot showing a student's response to a question and the associated feedback, including buttons which open instances of the Detective.

object representing each program execution's trace. These JSON representations are hidden in the HTML source of the page that displays the results of the checking.

In addition to the normal feedback he receives about his program's feedback (which test cases his code passes, as well as any solution-specific feedback as described in section 4.3), the student is presented with buttons which offer him the ability to visualize any of the given test cases using the Detective. When one of these buttons is pressed, the corresponding test case's trace is pushed into a hidden form, which is submitted to open a new instance of the Detective for visualizing that test case's execution. An example of this interface is shown in figure 5-8.

5.6 Looking Ahead

In its current form, the Detective plays the role of a disseminator of knowledge, and as an interpreter of Python's internal state as well as the messages the Python interpreter generates. Missing, however, from this setup is a sense of interactivity. As it currently stands, a student's interaction with the Detective is limited to passively absorbing the explanations and interpretations the Detective provides. Looking toward the future, there is potential to improve the interactivity of students' use of the Detective.

Hundhausen, et al[8] suggest that the type and quality of a user's interaction with a software visualization is more important than the content of the visualization itself. This supports the principle of active learning, whose techniques have proven effective[15] across disciplines and degrees of mastery. The ideas that follow are centered around actively engaging the user through the detective, based on the fact that such engagement has proven effective over the years.

Inspired by Ko and Myers[10], one idea is to incorporate questions and answers into the Detective, allowing users to ask questions about different elements in the visualization and receive automatically-generated answers in response. In this same vein, Myller[18] suggests that incorporating "prediction"-type questions into a software visualization can increase the benefit students receive from interacting with that visualization, and that this task can be automated.

Certain types of questions (e.g., "what does this line do?", "how does this expression resolve?", and "what does this error message mean?") would be relatively easy to incorporate into the Detective in its current form, as the answers to these questions are already generated by the explainer, the pILR system, and the error analyzer, respectively. Answering additional types of questions, such as "why did variable x have value y at this time?" seems feasible, by searching backward in time through execution trace.

The inclusion of both predictive and summative questions has the potential to greatly increase the feeling of interactivity elicited from the Detective; this is desirable in that these questions could force the student to think about the issues with his program (thus potentially realizing them on his own) before being presented with information about it.

It is also worth noting that, in its current form, the Detective has no knowledge whatsoever of the problem the student is trying to solve, nor of the instructor's solution to that problem. If this extra knowledge were to be incorporated into the Detective, it is easy to imagine comparing students' submissions to instructors' solutions to provide additional information about relative complexity or style. For example, the cyclomatic complexity[13] or running time of the student's code might be compared against the solution to give students an idea not only of whether the submitted code is correct, but also about how efficient it is.

Beyond even this, one can imagine tailoring the Detective's responses to individuals, based on an estimate of each student's level of understanding of various programming practices and syntactic structures. In its current form, the Detective generates feedback that is almost exclusively geared toward novices, but the argument could be made that an ideal automatic tutor would be able to cater to a broader audience.

It is well-established that novices and experts in a given domain view problems in that domain differently; at the very least, experts tend to notice more patterns and abstractions not noticed by novices, and have a deeper understanding of how these patterns and abstractions relate to the problem being solved[19]. Thus, it makes sense that an ideal automatic tutor would (much like a human tutor) use different language and examples to explain concepts to students with various levels of understanding and ability.

Implicitly, the Detective assumes that its users are very new to programming as a discipline, using text to describe how statements are interpreted at a very low level, but not providing insight above that level. It is feasible that the templates the Detective uses to generate explanations of error messages and statements could be modified based on an estimate of an individual's understanding of various concepts. Beck, et al[1] describe a method for gathering such an estimate from students' responses to various exercises in an intelligent tutoring system for middle-school-level mathematics, which could potentially be extended to the domain of computer programming.

Chapter 6

Conclusions and Future Work

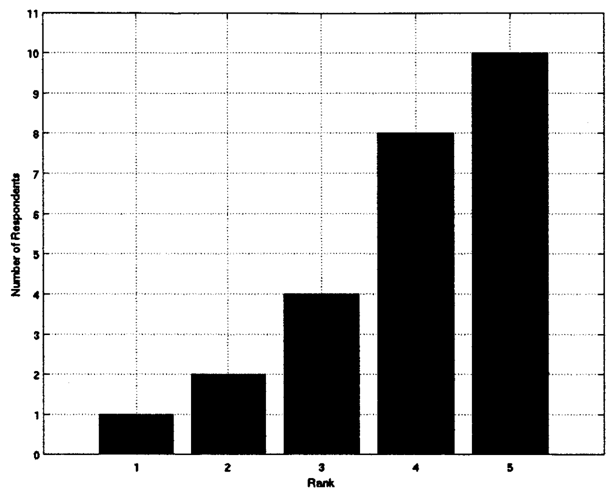
The CAT-SOOP system has proven to be a success in its initial pilot test, and early surveys have provided insight into valuable areas of future work.

Results from *6.003*'s end-of-term survey for fall 2011 suggest that, in general, students enjoyed using CAT-SOOP to submit their homework assignments, and informal qualitative feedback corroborates with this. Figure 6-1 contains a graph of the raw data collected from this survey.

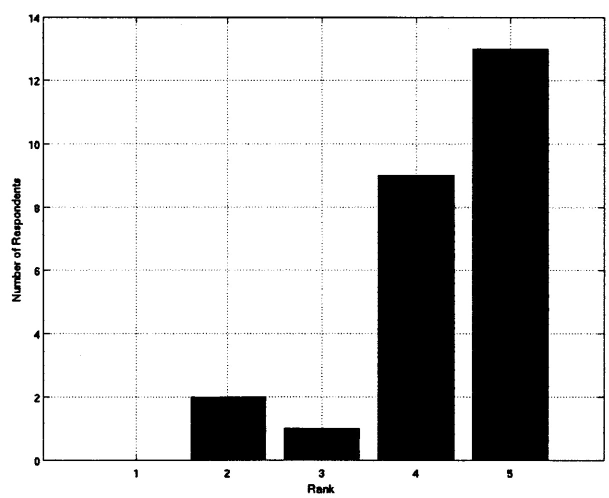
Despite the fact that feedback was generally positive, some of the most interesting feedback received took the form of negative comments. Quoting from the survey results:

- "The tutor encourages obsession over the correct answer. Due to lack of feedback about <u>why</u> an answer was wrong, you don't learn anything better than from just handing in paper."
- "The tutor should give more feedback, such as ... being off by a constant."
- "Try looking into using it differently, though, so students don't use it as a crutch."

The comments suggest that, for these types of systems to provide maximal benefit over paper assignments, the feedback they provide must not only be immediate, but most go beyond assessment of a submission's correctness. In addition, the comments



(a) "I liked the 6.003 tutor."



(b) "The most important feature of the 6.003 tutor is that it provides immediate feedback."

Figure 6-1: Students' responses to end-of term survey questions relating to CAT-SOOP for *6.003*, fall term 2011. Users were asked to rank their degree of agreement with the above statements on a scale from 1 (total disagreement) to 5 (total agreement), with 3 as a neutral point. A total of 25 data points were collected for each question.

also seem to suggest that this limited feedback may result in the students themselves focusing more on correctness than on conceptual understanding[1]. Thus, it certainly seems that a valuable line of future research in automatic tutoring lies in investigating additional forms of conceptual feedback, as well as the means by which they may be automatically generated from student submissions (chapter 3 discusses a few such possibilities, and many more certainly exist).

Unfortunately, the Detective has not been rigorously tested[2]; however, its unique type of feedback (objective data about the program's exeuction, augmented by interpretations of common Python statements and error messages, garnered and interepreted through relatively simple means) provides an interesting alternative to other methods of feedback currently being investigated. Thus, future plans include thorough testing of the Detective, as well as incorporating some of the additional feedback measures discussed in chapter 5.

While it still remains to be seen whether, and to what extent, CAT-SOOP and the Detective will prove beneficial to students in the future, early results show that students in *6.003* saw it as helpful, and suggest that this benefit could be carried over to *6.01*, or other courses, with relative ease. In addition, although it has not been thoroughly tested, the Detective provides a proof-of-concept for an interesting integration between run-time traces and automatic tutors, and suggests that more research along these lines may yield positive results.

[1]Indeed, through my experience with 6.01, I have noticed (in some students) a tendency to focus on attaining full marks on online problems, with little regard for the underlying concepts. Often this limited thinking manifests itself as an inability on the part of the student to explain the process by which he solved the problem, and an inability to abstract important concepts away from a particular problem and apply them in another context. Whether this is simply natural behavior on a student's part, or whether automatic tutors (and the immediate feedback they provide) contribute to this attitude, remains to be seen.

[2]Nor has the checking of Python code within CAT-SOOP, but since tutor2 and CAT-SOOP share essentially the same checking code for programming exercises, it is likely just fine.

Appendix A

Source Code Listings

A.1 CAT-SOOP

A.1.1 expressions_ast.py

```
# expressions_ast.py
# new module for checking symbolic expressions in CAT-SOOP/tutor2
#
# 2 march 2012, adam j hartz <hartz@alum.mit.edu>

import ast # Python's parser
import math,cmath
import random

def parse_expr(string):
    """
    Parse down an expression into a Python AST
    """
    node = ast.parse(string)
    return node.body[0].value # ast parser gives us a 'module'; first object in it is the expression

def compile_ast(tree):
    """
    Compile an AST tree into a Python code object to be run
    """
    expr = ast.Expression(tree)
    expr.lineno = 1
    expr.col_offset = 0
    return compile(expr,"<CAT-SOOP>", "eval")

def get_all_names(tree):
    """
    Given an AST, return a list of all variable names contained within it.
    For now, ignores attributes, slices, etc.
    """
    if isinstance(tree,ast.Name):
        return [tree.id]
    out = []
```

```
    for child in ast.iter_child_nodes(tree):
        out.extend(get_all_names(child))
    return out

def get_var_values(names):
    """
    Assign random values to each variable name the list passed in.

    Uses complex type for all numbers.  Always give the following values:
        'j' is complex(0,1)
        'e' is math.e
        'sqrt' is cmath.sqrt function
        'abs' is built-in absolute value function
    """
    out = dict([(name,complex(random.uniform(-20,20))) for name in names])
    out.update({'j':complex(0,1),'e':math.e,'sqrt':cmath.sqrt,'abs':abs}) # reserved names
    return out

def get_numerical_value(tree,varcache):
    #varcache is a dictionary mapping variable names to numerical value
    t = compile_ast(tree)
    return eval(t,varcache)

def check(submission, solution, threshold=1e-4):
    """
    Compare a student's submission to a solution by parsing down into an
    AST, generating numerical values for each variable, and evaluating the
    AST

    returns a dictionary with two keys:
         'ok' maps to a Boolean, whether the two submissions match
         'msg' maps to a message to be displayed back to the user
    """
    try:
        p = parse_expr(submission)
    except:
        return {'ok':False,'msg':'This expression contains a syntax error'}
    pa = parse_expr(solution)
    vars = {}
    vars.update(get_var_values(get_all_names(p)))
    vars.update(get_var_values(get_all_names(pa)))
    l = get_latex(p)
    v = get_numerical_value(p,vars)
    va = get_numerical_value(pa,vars)
    ok = abs(v-va) < threshold
    msg = "Your expression was parsed as:<br><dmath>%s</dmath>" % l
    return {'ok':ok,'msg':msg}

def check_n(n,submission,solution,threshold=1e-4):
    tests = [check(submission,solution,threshold,verbose) for i in xrange(n)]
    return {'ok': all([i['ok'] for i in tests]), 'msg':tests[0]['msg']}



####
#AST-to-LaTeX
# Most of this code is by Geoff Reedy (http://stackoverflow.com/users/166955/geoff-reedy)
# Found at http://stackoverflow.com/questions/3867028/converting-a-python-numeric-expression-to-latex
####

import ast
```

```python

#Greek letters: input-to-output mapping

GREEK_LETTERS = ['alpha', 'beta', 'gamma', 'delta', 'epsilon', 'zeta', 'eta', 'theta', 'iota',
                 'kappa', 'lambda', 'mu', 'nu', 'xi', 'omicron', 'pi', 'rho', 'sigma', 'tau',
                 'upsilon', 'phi', 'chi', 'psi', 'omega']
GREEK_DICT = {}
for i in GREEK_LETTERS:
    GREEK_DICT[i] = "\\%s" % i
    GREEK_DICT[i.upper()] = "\\%s" % i.title()


class LatexVisitor(ast.NodeVisitor):

    def prec(self, n):
        return getattr(self, 'prec_'+n.__class__.__name__, getattr(self, 'generic_prec'))(n)

    def visit_Call(self, n):
        func = self.visit(n.func)
        args = ', '.join(map(self.visit, n.args))
        if func == 'sqrt':
            return r'\sqrt{%s}' % args
        elif func == 'abs':
            return r'\left| %s \right|'
        else:
            return r'\operatorname{%s}\left(%s\right)' % (func, args)

    def prec_Call(self, n):
        return 1000

    def visit_Name(self, n):
        i = n.id
        s = i.split("_")
        if len(s) > 2:
            return "".join(s)
        elif len(s) == 2:
            if len(s[1]) > 1 or len(s[1]) == 0:
                return "".join(s)
            else:
                return "%s_%s" % (GREEK_DICT.get(s[0],s[0]),s[1])
        else:
            return GREEK_DICT.get(i,i)

    def prec_Name(self, n):
        return 1000

    def visit_UnaryOp(self, n):
        if self.prec(n.op) > self.prec(n.operand):
            return r'%s \left(%s\right)' % (self.visit(n.op), self.visit(n.operand))
        else:
            return r'%s %s' % (self.visit(n.op), self.visit(n.operand))

    def prec_UnaryOp(self, n):
        return self.prec(n.op)

    def visit_BinOp(self, n):
        if self.prec(n.op) > self.prec(n.left):
            left = r'\left(%s\right)' % self.visit(n.left)
        else:
            left = self.visit(n.left)
        if self.prec(n.op) > self.prec(n.right):
```

```
            right = r'\left(%s\right)' % self.visit(n.right)
        else:
            right = self.visit(n.right)
        if isinstance(n.op, ast.Div):
            try:
                l = get_numerical_value(n.left,{})
                r = get_numerical_value(n.right,{})
            except: #this branch means there's a variable involved
                return r'\frac{%s}{%s}' % (self.visit(n.left), self.visit(n.right))
            if isinstance(l,int) and isinstance(r,int): # if both ints, explicitly show floor division
                return r'\left\lfloor\frac{%s}{%s}\right\rfloor' % (self.visit(n.left), self.visit(n.right))
            else:
                return r'\frac{%s}{%s}' % (self.visit(n.left), self.visit(n.right))
        elif isinstance(n.op, ast.FloorDiv):
            return r'\left\lfloor\frac{%s}{%s}\right\rfloor' % (self.visit(n.left), self.visit(n.right))
        elif isinstance(n.op, ast.Pow):
            return r'%s^{%s}' % (left, self.visit(n.right))
        else:
            return r'%s %s %s' % (left, self.visit(n.op), right)

    def prec_BinOp(self, n):
        return self.prec(n.op)

    def visit_Sub(self, n):
        return '-'

    def prec_Sub(self, n):
        return 300

    def visit_Add(self, n):
        return '+'

    def prec_Add(self, n):
        return 300

    def visit_Mult(self, n):
        return '\\cdot'

    def prec_Mult(self, n):
        return 400

    def visit_Mod(self, n):
        return '\\bmod'

    def prec_Mod(self, n):
        return 500

    def prec_Pow(self, n):
        return 700

    def prec_Div(self, n):
        return 400

    def prec_FloorDiv(self, n):
        return 400

    def visit_LShift(self, n):
        return '\\operatorname{shiftLeft}'

    def visit_RShift(self, n):
        return '\\operatorname{shiftRight}'
```

```python

    def visit_BitOr(self, n):
        return '\\operatorname{or}'

    def visit_BitXor(self, n):
        return '\\operatorname{xor}'

    def visit_BitAnd(self, n):
        return '\\operatorname{and}'

    def visit_Invert(self, n):
        return '\\operatorname{invert}'

    def prec_Invert(self, n):
        return 800

    def visit_Not(self, n):
        return '\\neg'

    def prec_Not(self, n):
        return 800

    def visit_UAdd(self, n):
        return '+'

    def prec_UAdd(self, n):
        return 800

    def visit_USub(self, n):
        return '-'

    def prec_USub(self, n):
        return 800
    def visit_Num(self, n):
        return str(n.n)

    def prec_Num(self, n):
        return 1000

    def generic_visit(self, n):
        if isinstance(n, ast.AST):
            return r'' % (n.__class__.__name__, ', '.join(map(self.visit, [getattr(n, f) for f in n._fields])))
        else:
            return str(n)

    def generic_prec(self, n):
        return 0

def get_latex(tree):
    return LatexVisitor().visit(tree)
```

A.1.2 pysandbox_subprocess.py

```
#!/usr/bin/python
#
# File:    pysandbox_subprocess.py
# Date:    30-Aug-11
# Author: Adam Hartz <hartz@alum.mit.edu>


#-----------------------------------------------------------------------------
# run code in sandbox and return strings

import subprocess
import re
import resource
import os

DANGEROUS_CODES = ["mysqldb","_mysql","sqlalchemy","importos","fromosimport",\
                   "importsys","fromsysimport","open(","file.__init__",\
                   "code.__init__",".__subclasses__","subprocess","fork(","multiprocessing",\
                   "threading","builtins"]

def remove_comments(code):
    """
    Remove all comments from a piece of code
    """
    lines = code.splitlines()
    for lineno in xrange(len(lines)):
        line = lines[lineno]
        ix = line.find("#")
        if ix >= 0:
            lines[lineno] = line[:ix]
    return "\n".join([line for line in lines if line.strip()!=''])

def is_safe(code):
    """
    Rudimentary means of checking whether submitted code is an attempt to muck with the system
    """
    code = remove_comments(code).replace(" ","").replace("\t","")\
           .replace("\\","").replace("\n","")
    for c in DANGEROUS_CODES:
        if code.find(c) >= 0:
            return False
    return True

def mangle_code(code,argv):

    #if code contains blacklisted statement, don't run it
    if not is_safe(code):
        return code, False

    #otherwise, prepare code for execution

    # mangle code to change os.getenv(foo) to ENV[foo]
    code = re.sub('os\.getenv\(([a-z0-9\'\"]+)\)','ENV[\\1]',code)
    code = re.sub("os\.fdopen\(3,'w'\)",'log_output',code)

    # remove import os
    code = code.replace('import os','')

    # remove f.close()
```

```
    code = code.replace('f.close()','')

    # remove sys.exit(0)
    code = code.replace('sys.exit(0)','')

    # clean up CR's
    code = code.replace('\r','')

    head = "import sys\noldpath = sys.path\nsys.path = ['/usr/lib/python2.6','/home/tutor2/tutor/python_lib/lib601','/home/tutor2/tutor/python_lib']\n\n"
    head += "from cStringIO import StringIO\nlog_output = StringIO()\n\n"
    head += "ENV = %s\n\n" % repr(argv)

    footer = "\n\nprint \"!LOGOUTPUT\"\n" # our magic keyword
    footer += "print log_output.getvalue()\n" # values to compare
    code = head + code + footer
    return code, True

def setlimits():
    """
    Helper to set CPU time limit for check_code, so that infinite loops
    in submitted code get caught instead of actually running forever.
    """
    resource.setrlimit(resource.RLIMIT_CPU, (2, 2))

def sandbox_run_code(code,argv):
    """
    Run code, returning stdout, stderr, and output_log.

    argv should be a dict, giving the initial virtual environment.  We use it for
    passing argument valies, ie argv1, argv2, ... to the code being run

    """

    (code, code_ok) = mangle_code(code,argv)

    if not code_ok:
        return('','BAD CODE - this will be logged','')


    python = subprocess.Popen(["python"],stdin = subprocess.PIPE,\
                                         stdout = subprocess.PIPE,\
                                         stderr = subprocess.PIPE,\
                                         preexec_fn = setlimits)
    output = python.communicate(code)

    out,err = output

    n = out.split("!LOGOUTPUT") # separate output from variables we want to compare

    if len(n) == 2: #should be this
        out,log = n
    elif len(n) == 1: #code didn't run to completion
        if err.strip() == "":
            err = "Your code did not run to completion, but no error message was returned."
            err += "\nThis normally means that your code contains an infinite loop or otherwise took too long to run."
        log = ''
    else: #someone is trying to game the system?
        out = ''
        log = ''
```

```
      err = "BAD CODE - this will be logged"
if len(out) >= 500: #truncate long code output
    out = out[:500]+"\n\n...OUTPUT TRUNCATED..."

return out,err,log
```

A.1.3 Range.py

```
# range.py
# hartz 2011

from __future__ import division
import re
import sys
import random
from Question import Question

class Range(Question):
    name = "Range"
    author = "Adam Hartz"
    email = "hartz@mit.edu"
    version = "2.1"
    date = "29 December 2011"

    def checker(self,submit,solution,user,last_submit):
        try:
            sub = parse(submit)
            sol = parse(solution)
            msg = "Your submission was parsed as:<br />\[%s\]" % str(sub)
        except:
            return (0.0, ("Your submission could not be parsed:<br /><tt>%s</tt>" % submit, ), "Error", submit)
        ok = random_check_range(sub,sol) and check_key_nums(sub,sol)
        if ok == True:
            bigmsg = "Correct"
        else:
            bigmsg = "Incorrect"
        return (1.0*ok, (msg,), bigmsg, submit)

    def get_html_template(self):
        return """%%if LAST_SUBMIT != None:
<input type='text' size='60' name='%s' value='${LAST_SUBMIT}' />
%%else:
<input type='text' size='60' name='%s' value='%s' />
%%endif\n""" % (self.name,self.name,self.default)

def random_check_range(r1,r2,lo=-10000,hi=10000,num=int(1e5)):
    for i in xrange(num):
        check = random.uniform(lo,hi)
        if r1.contains(check) != r2.contains(check):
            return False
    return True

def check_key_nums(sub,sol):
    for check in get_interesting_points(sol).union(get_interesting_points(sub)):
        if sub.contains(check) != sol.contains(check):
            return False
    return True

def str_to_range(s):
    m = list(Interval.matcher.finditer(s.strip()))
    if m is None or len(m) == 0:
        return None
    g = m[0].groups()
    if g[1].strip() == "INF":
        left = float('inf')
    elif g[1].strip() == "-INF":
        left = float('-inf')
```

```
    else:
        l = ("1.0*%s" % g[1])
        left = eval(l)
    if g[2].strip() == "INF":
        right = float('inf')
    elif g[2].strip() == "-INF":
        right = float('-inf')
    else:
        r = ("1.0*%s" % g[2])
        right = eval(r)


    il = g[0].strip() == '['
    ir = g[3].strip() == ']'
    return Interval(left,right,il,ir)

class Interval(object):
    matcher = re.compile(r"([\[\(])(?!\()(.*?)\s*,\s*(.*?)([\]\)])")

    def __init__(self,left,right,incl,incr):
        assert right >= left
        self.left = left
        self.right = right
        self.incl = incl
        self.incr = incr

    def __str__(self):
        return ("[" if self.incl else "(") + \
                str(self.left)+" , "+str(self.right) + \
                ("]" if self.incr else ")")

    def __repr__(self):
        return self.__str__()

    def contains(self,num):
        return (self.left < num < self.right) or \
                (self.left == num and self.incl) or \
                (self.right == num and self.incr)

class Intersection:
    def __init__(self,one,two):
        self.one = one
        self.two = two

    def contains(self,num):
        return self.one.contains(num) and self.two.contains(num)

    def __str__(self):
        l = str(self.one) if isinstance(self.one,Interval) else ("(%s)" % str(self.one))
        r = str(self.two) if isinstance(self.two,Interval) else ("(%s)" % str(self.two))
        return "%s \\cap %s" % (l,r)

    def __repr__(self):
        return self.__str__()


class Union:
    def __init__(self,one,two):
        self.one = one
        self.two = two

```

```
    def contains(self,num):
        return self.one.contains(num) or self.two.contains(num)

    def __str__(self):
        l = str(self.one) if isinstance(self.one,Interval) else ("(%s)" % str(self.one))
        r = str(self.two) if isinstance(self.two,Interval) else ("(%s)" % str(self.two))
        return "%s \\cup %s" % (l,r)

    def __repr__(self):
        return self.__str__()

def find_matching_paren(string,dir=1):
    print string
    match = ')' if dir == 1 else '('
    this = '(' if dir == 1 else ')'
    tally = 0
    ix = 0
    while ix < (len(string)):
        m = re.match(Interval.matcher,string[ix:])
        if m:
            ix += m.end()
            continue
        if tally == 0 and string[ix] == match:
            return ix
        elif string[ix] == this:
            tally -= 1
        elif string[ix] == match:
            tally += 1
        ix += 1
    return None

def get_interesting_points(thing):
    if isinstance(thing,Interval):
        return set([thing.left,thing.right,sys.maxint,-sys.maxint - 1,0])
    else:
        return get_interesting_points(thing.one).union(get_interesting_points(thing.two))

classmap = {'U':Union,'N':Intersection}

def parse_single(string):
    m = re.match(Interval.matcher,string)
    if m is not None:
        return str_to_range(string),string[m.end():]
    elif string.startswith("("):
        next = find_matching_paren(string[1:])
        return parse_helper(string[1:1+next])[0],string[2+next:]
    else:
        raise Exception(string)

def parse_helper(string):
    res1,new1 = parse_single(string)
    if new1 == "":
        return res1,""
    op = new1[0]
    res2,new2 = parse_single(new1[1:])
    return classmap[op](res1,res2),new2


def parse(string):
    return parse_helper(string)[0]
```

A.2 Detective

A.2.1 errors.py

```
# ERRORS.PY
# Simple interpretation of error messages
# hartz 2012

# This file is a part of CAT-SOOP Detective
# CAT-SOOP Detective is copyright (C) 2012 Adam Hartz.
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

import re
import ast
from trees import downward_search

## BEGIN Pitfall Analysis
def node_specific_search(astnode,test_func):
    if isinstance(astnode,ast.If) or isinstance(astnode,ast.While):
        return downward_search(astnode.test,test_func) is not None
    if isinstance(astnode,ast.Assign):
        return downward_search(astnode.value,test_func) is not None
    if isinstance(astnode,ast.Print):
        n = [downward_search(i,test_func) for i in astnode.values]
        return len([i for i in n if i is not None]) > 0
    if isinstance(astnode,ast.Return):
        return astnode.value is not None and downward_search(astnode.value,test_func) is not None

def pitfalls(astnode,code_lines,fname):
    """
    Explanation of Python programming pitfalls.  Future versions will consider more pitfalls.
    """
    # ^ vs **
    if node_specific_search(astnode, lambda n: isinstance(n,ast.BinOp) and isinstance(n.op,ast.BitXor)):
        out = "This line contains a caret (<tt>^</tt>), which is the syntax for a bitwise XOR operation."
        out += " If you want exponentiation, use two asterisks (<tt>**</tt>) instead."
        return out
    # overwriting or hiding built-in variable
    if isinstance(astnode,ast.Assign) and (isinstance(astnode.targets[0],ast.Name) and astnode.targets[0].id in
        __builtins__):
        name = astnode.targets[0].id
        if fname == '<module>':
            #in the global scope...overwritten
            out = "This line contains an assignment to a variable named <tt><b>%s</b></tt>.  " % name
            out += "However, <tt><b>%s</b></tt> is also the name of an object built in to Python.  " % name
            out += "This line will overwrite the built-in object so it can no longer be accessed."
        else:
```

```
            #inside of a function, so just hidden
            out = "This line contains an assignment to a variable named <tt><b>%s</b></tt>.  " % name
            out += "However, <tt><b>%s</b></tt> is also the name of an object built in to Python.  " % name
            out += "This assignment will ''hide'' the built-in object, so that it will not be accessible"
            out += " from within this function."
        return out
## END Pitfall Analysis

## BEGIN Run-time Error Analysis
def explain(error_message,locals,globals):
    for i in d:
        l = list(d[i][0].finditer(error_message))
        if len(l) == 0:
            continue
        m = l[0]
        return {'msg':"A Python error occurred:<p>" +
                "<tt>%s</tt>" % ":".join(error_message.split(":")[1:]) +
                "<p>" + d[i][1](m,locals,globals)}


#functions to generate interpretations of specific error messages.

def namenotdefined_message(match,locals,globals):
    varname = match.groups()[0]
    msg = "This message means that the program is trying to access a variable called <b><tt>%s</tt></b>. " % varname
    msg += "However, there is no such variable in the current scope.  If this is the correct "
    msg += " variable name, make sure it has been initialized first."
    current_scope = {}
    current_scope.update(globals)
    if len(locals) > 0:
        current_scope.update(locals)
    current_scope.update(__builtins__) #we want to look at built-in names as well.
    dist = sorted([(edit_distance(i,varname),i) for i in current_scope])
    close = [j[1] for j in dist if j[0] <= 2]
    if len(close) > 1:
        msg += "  If not, did you mean to use one of the following variables? "
        msg += "<p> %s" % "<br />\n".join(["<tt>%s</tt>" % i for i in close])
    elif len(close) == 1:
        msg += "  If not, did you mean to use the name <tt>%s</tt>?" % close[0]
    else:
        #if no variable names are close enough, pick those that are closest.
        #this will probably do a solid job for long-enough variable names
        nearest = [j[1] for j in dist if j[0] == min([k[0] for j in dist])]
        if len(nearest) > 1:
            msg += "  If not, did you mean to use one of the following variables? "
            msg += "<p> %s" % "<br />\n".join(["<tt>%s</tt>" % i for i in nearest])
        else:
            msg += "  If not, did you mean to use the name <tt>%s</tt>?" % nearest[0]
    return msg

def invalidoperation_message(match,locals,globals):
    op,type1,type2 = match.groups()
    if type1 == type2:
        plural_thing = ("two <tt>%s</tt>s" % type1)
    else:
        plural_thing = "%s and %s" % (indefinite_article(type1),indefinite_article(type2))

    msg = "This message means that the program is trying to combine "
    msg += "two objects using an operator, but doesn't know how to do so."
    msg += "<p>Specifically, this line is trying to combine %s using the <tt>%s</tt> operator, which " % (
```

```
            plural_thing,op)
    msg += "is not supported."
    return msg

def notsubscriptable_message(match,locals,globals):
    typ = match.groups()[0]
    msg = "Grabbing a single element from a collection using square brackets (<tt>[]</tt>)"
    msg += " is referred to as <i>subscripting</i>.  This message means that the program is trying to subscript "
    msg += "something that can't be subscripted (%s)" % indefinite_article(typ)

    if typ == 'function':
        msg += "<p>If you intended to call this function, you should use parentheses"
        msg += " instead of square brackets."
    return msg

def notcallable_message(match,locals,globals):
    typ = match.groups()[0]
    msg = "Executing the code stored within a function using round brackets (parentheses)"
    msg += " is referred to as <i>calling</i> that function.  This message means that the program is trying to call "
    msg += "something that can't be called (%s)" % indefinite_article(typ)

    if typ in ('list','tuple','dict'):
        msg += "<p>If you intended to index into this %s, you should use" % typ
        msg += " square brackets (<tt>[]</tt>) instead of parentheses."
    return msg

def notiterable_message(match,locals,globals):
    typ = match.groups()[0]
    msg = "Looping over the elements within a collection"
    msg += " is referred to as <i>iterating over</i> that collection.  This message means that the "
    msg += "program is trying to iterate over something "
    msg += "something that can't be iterated over (%s)" % indefinite_article(typ)
    return msg

# UTILITY METHODS USED ABOVE

def indefinite_article(string):
    """
    Prepend an appropriate indefinite article to the start of a string.
    """
    article = "an" if string.strip().lower()[0] in ('a','e','i','o','u') else "a"
    return "%s <tt>%s</tt>" % (article, string.strip())

def edit_distance(seq1, seq2):
    """
    Find the Damerau-Levenshtein distance between two strings.

    This code is written by Michael Homer, discovered at
    http://mwh.geek.nz/2009/04/26/python-damerau-levenshtein-distance/
    """
    oneago = None
    thisrow = range(1, len(seq2) + 1) + [0]
    for x in xrange(len(seq1)):
        twoago, oneago, thisrow = oneago, thisrow, [0] * len(seq2) + [x + 1]
        for y in xrange(len(seq2)):
            delcost = oneago[y] + 1
            addcost = thisrow[y - 1] + 1
            subcost = oneago[y - 1] + (seq1[x] != seq2[y])
            thisrow[y] = min(delcost, addcost, subcost)
            # This block deals with transpositions
```

```
            if (x > 0 and y > 0 and seq1[x] == seq2[y - 1]
                and seq1[x-1] == seq2[y] and seq1[x] != seq2[y]):
                thisrow[y] = min(thisrow[y], twoago[y - 2] + 1)
    return thisrow[len(seq2) - 1]


d = {
        'namenotdefined': (re.compile(r"NameError:(?: global)? name '(.*?)' is not defined"),
                          namenotdefined_message),

        'zerodivision': (re.compile(r"ZeroDivisionError: (.*)"),
                         lambda m,l,g: "This message means that the program is trying to divide by zero, which
                              would yield an undefined result.  Look carefully for places in this vicinity where
                              you are using division (<tt>/</tt>) or modulo (<tt>%</tt>); the second argument to
                              these operators cannot be zero."),

        'invalidoperation': (re.compile(r"TypeError: unsupported operand type\(s\) for (.*?): '(.*?)' and '(.*?)'
            "),
                            invalidoperation_message),

        'notsubscriptable': (re.compile(r"TypeError: '(.*?)' object is not subscriptable"),
                            notsubscriptable_message),

        'notcallable': (re.compile(r"TypeError: '(.*?)' object is not callable"),
                            notcallable_message),

        'indexoutofrange': (re.compile(r"list index out of range"),
                            lambda m,l,g: "This message means that the program is trying to grab the element at
                                index <i>n</i> in a sequence, but there is no such item.  Remember that valid
                                indices range from <tt>0</tt> to <tt>len(s)-1</tt> (or from <tt>-1</tt> to <tt>-
                                len(s)</tt>) inclusive, where <tt>s</tt> is the sequence in question.")
        }
## END Run-time Error Analysis
```

A.2.2 explainer.py

```python
# EXPLAINER.PY
# Simple explanation of lines of Python code
# hartz 2012

# This file is a part of CAT-SOOP Detective
# CAT-SOOP Detective is copyright (C) 2012 Adam Hartz.
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

from resolution import resolve
from trees import HTMLVisitor
import json
import ast
import traceback

def explain(node,locals,globals,fcache=None,event_type='',fname=''):
    fcache = {} if fcache is None else fcache
    a = HTMLVisitor()
    out = {}

    if event_type == 'return' and fname != "<module>":
        out['msg'] = "The function <tt><b>%s</b></tt> is about to return." % fname

    elif isinstance(node,ast.Assign):
        #only support single assignment for now
        i = node.targets[0]
        out['msg'] = "This is an <i>assignment</i> statement.  Python will evaluate the expression"
        out['msg'] += " on the right-hand side of the equals sign, and will "
        if isinstance(i,ast.Name):
            out['msg'] += "store the resulting value in variable %s." % HTMLVisitor().visit(i)
        elif isinstance(i,ast.Subscript):
            d = HTMLVisitor().visit(i.value)
            x = HTMLVisitor().visit(i.slice.value) # assume slice is a single Index
            out['msg'] += "attempt to store the resulting value in variable %s at index %s" % (d,x)
        else:
            out['msg'] += "attempt to store the resulting value."

        try:
            r = resolve(node.value,locals,globals,fcache)
            out['res'] = [((a.visit(x) if x != 'ERROR' else "<font color='red'><tt>ERROR!</tt></font>"),y) for (x
                ,y) in r]
        except:
            out['res'] = None

        if out['res'] is not None:
            out['msg'] += "<p>The expression in question resolves as follows:"


```

```
    elif isinstance(node, ast.FunctionDef):
        i = node.name
        if event_type != 'call':
            out['msg'] = "This is a <i>function definition</i> statement.  Python will store this function "
            out['msg'] += "in variable <b><tt>%s</tt></b> so that it may be called later." % i
        else:
            out['msg'] = "The function <b><tt>%s</tt></b>, which was defined earlier, is now being called." % i
            out['msg'] += " Execution will now jump to line %d" % node.body[0].lineno

    elif isinstance(node, ast.Return):
        v = node.value
        if v is not None:
            out['msg'] = "This is a <i>return</i> statement.  Python will evaluate the given expression, and "
            out['msg'] += "yield that value as the result of this function call."
            try:
                r = resolve(node.value,locals,globals,fcache)
                out['res'] = [((a.visit(x) if x != 'ERROR' else "<font color='red'><tt>ERROR!</tt></font>"),y)
                     for (x,y) in r]
            except:
                out['res'] = None
            if out['res'] is not None:
                out['msg'] += "<p>The expression in question resolves as follows:"
        else:
            out['msg'] = "This is a <i>return</i> statement.  Since no expression was given, Python will "
            out['msg'] += "yield <tt>None</tt> as the result of this function call."
            out['res'] = None

    elif isinstance(node,ast.Delete):
        i = node.name
        out['msg'] = "This is a <i>deletion</i> statement."

    elif isinstance(node,ast.Print):
        out['msg'] = "This is a <i>print</i> statement."
        v = node.values
        if len(v) == 0:
            out['msg'] += "  Since no value was given, to be printed this will display a blank line."
        if len(v) == 1:
            out['msg'] += "  Python will evaluate the given expression, and display it to the console."
            try:
                r = resolve(v[0],locals,globals,fcache)
                out['res'] = [((a.visit(x) if x != 'ERROR' else "<font color='red'><tt>ERROR!</tt></font>"),y)
                     for (x,y) in r]
            except:
                out['res'] = None
            if out['res'] is not None:
                out['msg'] += "<p>The expression in question resolves as follows:"
        else:
            out['msg'] += "  Python will evaluate the given expressions, and display them to the console,
                 separated by a space."
            try:
                r = resolve(v,locals,globals,fcache)
                out['res'] = [((a.visit(x) if x != 'ERROR' else "<font color='red'><tt>ERROR!</tt></font>"),y)
                     for (x,y) in r]
            except:
                out['res'] = None
            if out['res'] is not None:
                out['msg'] += "<p>The values in question resolve as follows:"


    elif isinstance(node,ast.If):
        t = node.body[0].lineno
```

```
        try:
            f = node.orelse[0].lineno
        except:
            f = None
        out['msg'] = "This is an <i>if</i> statement.  Python will evaluate the given expression."
        out['msg'] += "  If it evaluates to <tt>True</tt>, Python will jump to line %d.  "  % t
        if f is not None:
            out['msg'] += "If it evaluates to <tt>False</tt>, Python will jump instead to line %d." % f

        try:
            r = resolve(node.test,locals,globals,fcache)
            out['res'] = [((a.visit(x) if x != 'ERROR' else "<font color='red'><tt>ERROR!</tt></font>"),y) for (x,y) in r]
        except:
            out['res'] = None
        if out['res'] is not None:
            out['msg'] += "<p>The expression in question resolves as follows:"

    elif isinstance(node,ast.For):
        iterable = HTMLVisitor().visit(node.iter)
        target = HTMLVisitor().visit(node.target)
        out['msg'] = "This is a <i>for</i> loop.  Python will run the given code block once for each element in "
        out['msg'] += "<tt><b>%s</b></tt>, each time setting a variable <tt><b>%s</b></tt> equal to the next element in <tt><b>%s</b></tt>." % (iterable,target,iterable)

    elif isinstance(node,ast.While):
        t = node.body[0].lineno
        out['msg'] = "This is a <i>while</i> loop.  Python will evaluate the given expression."
        out['msg'] += "  If it evaluates to <tt>True</tt>, Python will jump to line %d, execute the "  % t
        out['msg'] += " code in that block, and return here to check again."
        out['msg'] += " If it instead evaluates to <tt>False</tt>, Python will skip this code block altogether."

        try:
            r = resolve(node.test,locals,globals,fcache)
            out['res'] = [((a.visit(x) if x != 'ERROR' else "<font color='red'><tt>ERROR!</tt></font>"),y) for (x,y) in r]
        except:
            out['res'] = None
        if out['res'] is not None:
            out['msg'] += "<p>The expression in question resolves as follows:"

    elif isinstance(node,ast.Break):
        out['msg'] = "This is a <i>break</i> statement. If it is given inside of a loop, this statement will cause Python to jump outside the loop, skipping the rest of the code block for this iteration and all subsequent iterations. If given outside of a loop, this statement will cause an error."

    elif isinstance(node,ast.Continue):
        out['msg'] = "This is a <i>continue</i> statement. If it is given inside of a loop, this statement will cause Python to jump to the top of the loop, skipping the rest of the code block for this iteration. If given outside of a loop, this statement will cause an error."

    elif isinstance(node,ast.Pass):
        out['msg'] = "This is a <i>pass</i> statement, which tells Python to do nothing."


    else:
        out['msg'] = ''

    return out
```

A.2.3 hz_encoder.py

```
# HZ_ENCODER.PY
# encode/decode output from hz_logger, etc
# hartz 2012

# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# Most of the code in this file is taken directly from pg_encoder:
# Online Python Tutor
# Copyright (C) 2010-2011 Philip J. Guo (philip@pgbovine.net)
# https://github.com/pgbovine/OnlinePythonTutor/
# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

# This file is a part of CAT-SOOP Detective
# CAT-SOOP Detective is copyright (C) 2012 Adam Hartz.
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.


# Given an arbitrary piece of Python data, encode it in such a manner
# that it can be later encoded into JSON.
#   http://json.org/
#
# We use this function to encode run-time traces of data structures
# to send to the front-end.
#
# Format:
#   * None, int, long, float, str, bool - unchanged
#     (json.dumps encodes these fine verbatim)
#   * list     - ['LIST', unique_id, elt1, elt2, elt3, ..., eltN]
#   * tuple    - ['TUPLE', unique_id, elt1, elt2, elt3, ..., eltN]
#   * set      - ['SET', unique_id, elt1, elt2, elt3, ..., eltN]
#   * dict     - ['DICT', unique_id, [key1, value1], [key2, value2], ..., [keyN, valueN]]
#   * instance - ['INSTANCE', class name, unique_id, [attr1, value1], [attr2, value2], ..., [attrN, valueN]]
#   * class    - ['CLASS', class name, unique_id, [list of superclass names], [attr1, value1], [attr2, value2],
    ..., [attrN, valueN]]
#   * circular reference - ['CIRCULAR_REF', unique_id]
#   * other    - [<type name>, unique_id, string representation of object]
#
# the unique_id is derived from id(), which allows us to explicitly
# capture aliasing of compound values

# Key: real ID from id()
# Value: a small integer for greater readability, set by cur_small_id
real_to_small_IDs = {}
cur_small_id = 1

import re, types,ast
typeRE = re.compile("<type '(.*)'>")
classRE = re.compile("<class '(.*)'>")
```

```
def encode(dat, ignore_id=False):
  def encode_helper(dat, compound_obj_ids):
    # primitive type
    if dat is None or \
       type(dat) in (int, long, float, str, bool):
      return dat
    # compound type
    else:
      my_id = id(dat)

      global cur_small_id
      if my_id not in real_to_small_IDs:
        if ignore_id:
          real_to_small_IDs[my_id] = 99999
        else:
          real_to_small_IDs[my_id] = cur_small_id
        cur_small_id += 1

      if my_id in compound_obj_ids:
        return ['CIRCULAR_REF', real_to_small_IDs[my_id]]

      new_compound_obj_ids = compound_obj_ids.union([my_id])

      typ = type(dat)

      my_small_id = real_to_small_IDs[my_id]

      if typ == list:
        ret = ['LIST', my_small_id]
        for e in dat: ret.append(encode_helper(e, new_compound_obj_ids))
      elif typ == tuple:
        ret = ['TUPLE', my_small_id]
        for e in dat: ret.append(encode_helper(e, new_compound_obj_ids))
      elif typ == set:
        ret = ['SET', my_small_id]
        for e in dat: ret.append(encode_helper(e, new_compound_obj_ids))
      elif typ == dict:
        ret = ['DICT', my_small_id]
        for (k,v) in dat.iteritems():
          # don't display some built-in locals ...
          if k not in ('__module__', '__return__'):
            ret.append([encode_helper(k, new_compound_obj_ids), encode_helper(v, new_compound_obj_ids)])
      elif typ in (types.InstanceType, types.ClassType, types.TypeType) or \
           classRE.match(str(typ)):
        # ugh, classRE match is a bit of a hack :(
        if typ == types.InstanceType or classRE.match(str(typ)):
          ret = ['INSTANCE', dat.__class__.__name__, my_small_id]
        else:
          superclass_names = [e.__name__ for e in dat.__bases__]
          ret = ['CLASS', dat.__name__, my_small_id, superclass_names]

        # traverse inside of its __dict__ to grab attributes
        # (filter out useless-seeming ones):
        user_attrs = sorted([e for e in dat.__dict__.keys()
                             if e not in ('__doc__', '__module__', '__return__')])

        for attr in user_attrs:
          ret.append([encode_helper(attr, new_compound_obj_ids), encode_helper(dat.__dict__[attr],
               new_compound_obj_ids)])
      else:
```

```
        typeStr = str(typ)
        m = typeRE.match(typeStr)
        assert m, typ
        ret = [m.group(1), my_small_id, str(dat)]

      return ret

  return encode_helper(dat, set())


#hartz 2012
def decode(encoded): #will not work for user-defined classes, but we're okay with that for now...
    out = None
    if type(encoded) != list:
        out = encoded #encoded is just a python literal
    else:
        typ = encoded[0]
        if typ=='LIST':
            out = [decode(i) for i in encoded[2:]]
        elif typ=='TUPLE':
            out = tuple(decode(i) for i in encoded[2:])
        elif typ=='SET':
            out = set([decode(i) for i in encoded[2:]])
        elif typ=='DICT':
            out = dict([(decode(k),decode(v)) for (k,v) in encoded[2:]])
        elif typ=='complex':
            out = eval(encoded[-1])
    return out

#hartz 2012
def encode_ast(p):
    if type(p) in (int,long,float,complex):
        out = ast.Num()
        out.n = p
    elif type(p) == str:
        out = ast.Str()
        out.s = p
    elif type(p) == list:
        out = ast.List()
        out.elts = [encode_ast(i) for i in p]
    elif type(p) == tuple:
        out = ast.Tuple()
        out.elts = [encode_ast(i) for i in p]
    elif type(p) == dict:
        out = ast.Dict()
        keys = p.keys()
        values = [p[k] for k in keys]
        out.keys = [encode_ast(i) for i in keys]
        out.values = [encode_ast(i) for i in values]
    elif type(p) == set:
        out = ast.Call()
        out.func = ast.Name()
        out.func.id = 'set'
        out.args = [encode_ast(list(p))]
    elif type(p) == bool:
        out = ast.Name()
        out.id = str(p)
    else:
        return None
    out.ctx = ast.Load()
    return out
```

A.2.4 hz_logger.py

```
# HZLOGGER.PY
# trace an execution of a Python script
# hartz 2012

# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# Most of the code in this file is taken directly from pg_logger:
# Online Python Tutor
# Copyright (C) 2010-2011 Philip J. Guo (philip@pgbovine.net)
# https://github.com/pgbovine/OnlinePythonTutor/
# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

# This file is a part of CAT-SOOP Detective
# CAT-SOOP Detective is copyright (C) 2012 Adam Hartz.
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

import sys
import bdb # the KEY import here!
import os
import re
import traceback

import cStringIO
import trees
import hz_encoder
import errors
import resolution
import explainer
import ast
import pickle

# upper-bound on the number of executed lines, in order to guard against
# infinite loops
MAX_EXECUTED_LINES = 200

def set_max_executed_lines(m):
  global MAX_EXECUTED_LINES
  MAX_EXECUTED_LINES = m

IGNORE_VARS = set(('__stdout__', '__builtins__', '__name__', '__exception__'))

def get_user_stdout(frame):
  return frame.f_globals['__stdout__'].getvalue()

def get_user_globals(frame):
  d = filter_var_dict(frame.f_globals)
  # also filter out __return__ for globals only, but NOT for locals
  if '__return__' in d:
```

```
    del d['__return__']
  return d

def get_user_locals(frame):
  return filter_var_dict(frame.f_locals)

def filter_var_dict(d):
  ret = {}
  for (k,v) in d.iteritems():
    if k not in IGNORE_VARS:
      ret[k] = v
  return ret


class HZLogger(bdb.Bdb):

    def __init__(self, finalizer_func, ignore_id=False):
        bdb.Bdb.__init__(self)
        self.mainpyfile = ''
        self._wait_for_mainpyfile = 0

        # a function that takes the output trace as a parameter and
        # processes it
        self.finalizer_func = finalizer_func

        # each entry contains a dict with the information for a single
        # executed line
        self.trace = []

        # don't print out a custom ID for each object
        # (for regression testing)
        self.ignore_id = ignore_id


    def reset(self):
        bdb.Bdb.reset(self)
        self.forget()

    def forget(self):
        self.lineno = None
        self.stack = []
        self.curindex = 0
        self.curframe = None

    def setup(self, f, t):
        self.forget()
        self.stack, self.curindex = self.get_stack(f, t)
        self.curframe = self.stack[self.curindex][0]


    # Override Bdb methods

    def user_call(self, frame, argument_list):
        """This method is called when there is the remote possibility
        that we ever need to stop in this function."""
        if self._wait_for_mainpyfile:
            return
        if self.stop_here(frame):
            self.interaction(frame, None, 'call')

    def user_line(self, frame):
```

```
        """This function is called when we stop or break at this line."""
        if self._wait_for_mainpyfile:
            if (self.canonic(frame.f_code.co_filename) != "<string>" or
                frame.f_lineno <= 0):
                return
            self._wait_for_mainpyfile = 0
        self.interaction(frame, None, 'step_line')

    def user_return(self, frame, return_value):
        """This function is called when a return trap is set here."""
        frame.f_locals['__return__'] = return_value
        self.interaction(frame, None, 'return')

    def user_exception(self, frame, exc_info):
        exc_type, exc_value, exc_traceback = exc_info
        """This function is called if an exception occurs,
        but only if we are to stop at or just below this level."""
        frame.f_locals['__exception__'] = exc_type, exc_value
        if type(exc_type) == type(''):
            exc_type_name = exc_type
        else: exc_type_name = exc_type.__name__
        self.interaction(frame, exc_traceback, 'exception')


    # General interaction function

    def interaction(self, frame, traceback, event_type):
        self.setup(frame, traceback)
        tos = self.stack[self.curindex]
        lineno = tos[1]

        # each element is a pair of (function name, ENCODED locals dict)
        encoded_stack_locals = []

        encoded_locals = None
        encoded_globals = None

        # climb up until you find '<module>', which is (hopefully) the global scope
        i = self.curindex
        while True:
          cur_frame = self.stack[i][0]
          cur_name = cur_frame.f_code.co_name
          if cur_name == '<module>':
            break

          # special case for lambdas - grab their line numbers too
          if cur_name == '<lambda>':
            cur_name = 'lambda on line ' + str(cur_frame.f_code.co_firstlineno)
          elif cur_name == '':
            cur_name = 'unnamed function'

          # encode in a JSON-friendly format now, in order to prevent ill
          # effects of aliasing later down the line ...
          encoded_locals = {}
          for (k, v) in get_user_locals(cur_frame).iteritems():
            # don't display some built-in locals ...
            if k != '__module__':
              encoded_locals[k] = hz_encoder.encode(v, self.ignore_id)

          encoded_stack_locals.append((cur_name, encoded_locals))
          i -= 1
```

```
        # encode in a JSON-friendly format now, in order to prevent ill
        # effects of aliasing later down the line ...
        encoded_globals = {}
        for (k, v) in get_user_globals(tos[0]).iteritems():
          encoded_globals[k] = hz_encoder.encode(v, self.ignore_id)


        #this seems a little convoluted, but i think i like it better than just making a copy
        #hartz 2012
        real_locals = dict([(k,hz_encoder.decode(v)) for (k,v) in (encoded_locals or {}).iteritems()])
        real_globals = dict([(k,hz_encoder.decode(v)) for (k,v) in (encoded_globals or {}).iteritems()])
        cur_node = trees.downward_search(self.tree,lambda n: n.lineno == lineno)

        trace_entry = dict(line=lineno,
                           event=event_type,
                           func_name=tos[0].f_code.co_name,
                           globals=encoded_globals,
                           stack_locals=encoded_stack_locals,
                           stdout=get_user_stdout(tos[0]))

        # if there's an exception, then record its info:
        if event_type == 'exception':
          # always check in f_locals
          exc = frame.f_locals['__exception__']
          trace_entry['exception_msg'] = exc[0].__name__ + ': ' + str(exc[1])
          trace_entry['explanation'] = errors.explain(trace_entry['exception_msg'],real_locals,real_globals) #hz
        else:
          trace_entry['explanation'] = explainer.explain(cur_node,real_locals,real_globals,event_type=event_type,
               fname=trace_entry['func_name']) #hz

        # hz 2012
        try:
          trace_entry['warnings'] = errors.pitfalls(cur_node,self.script_str,trace_entry['func_name'])
        except:
          trace_entry['warnings'] = None
        # /hz 2012

        self.trace.append(trace_entry)

        if len(self.trace) >= MAX_EXECUTED_LINES:
          self.trace.append(dict(event='instruction_limit_reached', exception_msg='(stopped after ' + str(
               MAX_EXECUTED_LINES) + ' steps to prevent possible infinite loop)'))
          self.force_terminate()

        self.forget()

    def _runscript(self, script_str):
        # When bdb sets tracing, a number of call and line events happens
        # BEFORE debugger even reaches user's code (and the exact sequence of
        # events depends on python version). So we take special measures to
        # avoid stopping before we reach the main script (see user_line and
        # user_call for details).
        self._wait_for_mainpyfile = 1

        script_str = script_str.replace("\r","")

        # ok, let's try to sorta 'sandbox' the user script by not
        # allowing certain potentially dangerous operations:
        user_builtins = {}
        for (k,v) in __builtins__.iteritems():
```

```
        if k in ('reload', 'input', 'apply', 'open', 'compile',
                 'file', 'eval', 'execfile', '__import__',
                 'exit', 'quit', 'raw_input',
                 'dir', 'globals', 'locals', 'vars',
                 'compile'):
          continue
        user_builtins[k] = v

      # redirect stdout of the user program to a memory buffer
      user_stdout = cStringIO.StringIO()
      sys.stdout = user_stdout

      user_globals = {"__name__"    : "__main__",
                      "__builtins__" : user_builtins,
                      "__stdout__" : user_stdout}

      # BEGIN hartz 2012
      # store this as an instance variable so we can inspect it later...
      self.script_str = script_str.splitlines()

      # parse the input script down into an AST; we'll use this later when
      # generating explanations, etc.
      self.tree = ast.parse(script_str)

      # END hartz 2012

      try:
        self.run(script_str, user_globals, user_globals)
      # sys.exit ...
      except SystemExit:
        sys.exit(0)
      except:
        traceback.print_exc() # uncomment this to see the REAL exception msg

        trace_entry = dict(event='uncaught_exception')

        exc = sys.exc_info()[1]
        if hasattr(exc, 'lineno'):
          trace_entry['line'] = exc.lineno
        if hasattr(exc, 'offset'):
          trace_entry['offset'] = exc.offset

        if hasattr(exc, 'msg') or hasattr(exc, 'message'): # hartz 2012 ('message' would be nice, too)
          try:
              m = exc.msg
          except:
              m = exc.message
          trace_entry['exception_msg'] = "Error: " + (m)
        else:
          trace_entry['exception_msg'] = "Unknown error"

        self.trace.append(trace_entry)
        self.finalize()
        sys.exit(0) # need to forceably STOP execution

  def force_terminate(self):
    self.finalize()
    sys.exit(0) # need to forceably STOP execution


  def finalize(self):
```

```
      sys.stdout = sys.__stdout__
      assert len(self.trace) <= (MAX_EXECUTED_LINES + 1)

      # filter all entries after 'return' from '<module>', since they
      # seem extraneous:
      res = []
      for e in self.trace:
        res.append(e)
        if e['event'] == 'return' and e['func_name'] == '<module>':
          break

      # another hack: if the SECOND to last entry is an 'exception'
      # and the last entry is return from <module>, then axe the last
      # entry, for aesthetic reasons :)
      if len(res) >= 2 and \
         res[-2]['event'] == 'exception' and \
         res[-1]['event'] == 'return' and res[-1]['func_name'] == '<module>':
        res.pop()

      self.trace = res

      #for e in self.trace: print e

      self.finalizer_func(self.trace)


# the MAIN meaty function!!!
def exec_script_str(script_str, finalizer_func, ignore_id=False):
  logger = HZLogger(finalizer_func, ignore_id)
  logger._runscript(script_str)
  logger.finalize()


def exec_file_and_pretty_print(mainpyfile):
  import pprint

  if not os.path.exists(mainpyfile):
    print 'Error:', mainpyfile, 'does not exist'
    sys.exit(1)

  def pretty_print(output_lst):
    for e in output_lst:
      pprint.pprint(e)

  output_lst = exec_script_str(open(mainpyfile).read(), pretty_print)


if __name__ == '__main__':
  # need this round-about import to get __builtins__ to work :0
  import hz_logger
  hz_logger.exec_file_and_pretty_print(sys.argv[1])
```

A.2.5 resolution.py

```python
# RESOLUTION.PY
# (pseudo-) instruction-level resolution of Python programs
# hartz 2012

# This file is a part of CAT-SOOP Detective
# CAT-SOOP Detective is copyright (C) 2012 Adam Hartz.
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.


import ast
import traceback
from hz_encoder import decode,encode_ast
import sys

def set_lineno(tree,recursive=True):
    tree.lineno = 1
    tree.col_offset = 1
    if recursive:
        for i in ast.iter_child_nodes(tree):
            set_lineno(i)

def evaluate_tree(tree,locals,globals):
    """
    Given an AST node and a set of variables, return the value to which
    the given node would resolve in the specified environment.
    """

    set_lineno(tree,recursive=True)
    e = ast.Expression(tree)
    code = compile(e,"<submitted code>","eval")
    out = eval(code,globals,locals)
    return out

#AST Class -> English mapping
operators = {ast.Add:'Addition', ast.Sub:'Subtraction', ast.Mult:'Multiplication', ast.Div:'Division',
             ast.Mod:'Modulo', ast.Pow:'Exponentiation', ast.LShift:'Left-shift',
             ast.RShift:'Right-shift', ast.BitOr:'Bit-wise OR', ast.BitXor:'Bit-wise XOR',
             ast.BitAnd:'Bit-wise AND',ast.FloorDiv:'Floor (integer) division'}

unary_operators = {ast.Invert:"Bit-wise Inversion", ast.Not:"Logical NOT",
                   ast.USub:"Unary Subtraction (Negation)", ast.UAdd:"Unary Addition (Identity)"}

comp_operators = {ast.Eq:"'Equal' Comparison", ast.NotEq:"'Not-equal' Comparison",
                  ast.Lt:"'Less-than' Comparison", ast.LtE:"'Less-than-or-equal-to' Comparison",
                  ast.Gt:"'Greater-than' Comparison", ast.GtE:"'Greater-than-or-equal-to' Comparison",
                  ast.Is:"'Is' Comparison (Object Identity)",
                  ast.IsNot:"'Is-not' Comparison (Object Identity)",
```

```
                   ast.In:"'In' Comparison", ast.NotIn:"'Not-in' Comparison"}

DEBUG = True

def resolve(node,locals,globals,function_cache=None): #assume globals,locals have been decoded by here...
    """
    Return a list of tuples (state,action), where state is the current state (an AST Node)
    is the state of the resolution, and action (a string) is a description of the action
    taken to reach that state from the previous one.
    """
    if function_cache is None:
        function_cache = {}

    out = None

    if isinstance(node,ast.Num) or isinstance(node,ast.Str):
        out = [(node,None)]

    elif isinstance(node,ast.Name):
        i = node.id
        msg = "Loading variable <tt><b>%s</b></tt>" % i
        try:
            internal = evaluate_tree(node,locals,globals) #might throw exception, be ready to catch.
        except:
            return [(node,None),('ERROR',msg)]
        out = [(node,None),(encode_ast(internal),msg)]

    elif isinstance(node,ast.List):
        out = [(node,None)]
        #resolve each element in turn
        for ix in xrange(len(node.elts)):
            if out[-1][0] == 'ERROR':
                break
            res = resolve(node.elts[ix],locals,globals)
            last = out[-1][0].elts
            front = last[:ix]
            back = last[ix+1:]
            for i in res[1:]:
                a,msg = i
                if a == 'ERROR':
                    out.append(i)
                    break
                l = ast.List()
                l.ctx = ast.Load()
                l.elts = front + [a] + back
                out.append((l,msg))

    elif isinstance(node,ast.Tuple):
        #as with lists, resolve each element in turn
        out = [(node,None)]
        for ix in xrange(len(node.elts)):
            if out[-1][0] == 'ERROR':
                break
            res = resolve(node.elts[ix],locals,globals)
            last = out[-1][0].elts
            front = last[:ix]
            back = last[ix+1:]
            for i in res[1:]:
                a,msg = i
                if a == 'ERROR':
                    out.append(i)
```

```
                    break
                l = ast.Tuple()
                l.ctx = ast.Load()
                l.elts = front + [a] + back
                out.append((l,msg))

    elif isinstance(node,ast.Dict):
        # really tedious, but...resolve each key->val pair
        out = [(node,None)]
        for ix in xrange(len(node.keys)):
            if out[-1][0] == 'ERROR':
                break
            last = out[-1][0]
            front_keys = last.keys[:ix]
            back_keys = last.keys[ix+1:]
            front_vals = last.values[:ix]
            back_vals = last.values[ix+1:]
            #resolve the key step-by-step
            reskey = resolve(node.keys[ix], locals, globals)
            resval = resolve(node.values[ix], locals, globals)
            for jx in xrange(1,len(reskey)):
                a,msg = reskey[jx]
                if a == 'ERROR':
                    out.append(reskey[jx])
                    break
                d = ast.Dict()
                d.ctx = ast.Load()
                d.keys = front_keys+[a]+back_keys
                d.values = last.values[:]
                out.append((d,msg))
            #once we've resolved the key, resolve the associated value
            last = out[-1][0]
            if out[-1][0] == 'ERROR':
                break
            for i in resval[1:]:
                a,msg = i
                if a == 'ERROR':
                    out.append(i)
                    break
                d = ast.Dict()
                d.ctx = ast.Load()
                d.keys = last.keys[:]
                d.values = front_vals + [a] + back_vals
                out.append((d,msg))

    elif isinstance(node,ast.BinOp):
        # resolve left side of tree
        out = [(node,None)]
        for i in resolve(node.left,locals,globals)[1:]:
            if i[0] == 'ERROR':
                out.append(i)
                break
            new = ast.BinOp()
            new.op = out[-1][0].op
            new.ctx = ast.Load()
            new.left = i[0]
            new.right = out[-1][0].right
            out.append((new,i[1]))
        if out[-1][0] == 'ERROR':
            return out
        #resolve right side of tree
```

```
        for i in resolve(node.right,locals,globals)[1:]:
            if i[0] == 'ERROR':
                out.append(i)
                break
            new = ast.BinOp()
            new.op = out[-1][0].op
            new.ctx = ast.Load()
            new.left = out[-1][0].left
            new.right = i[0]
            out.append((new,i[1]))
        #if we've made it this far, resolve the operation itself
        msg = operators[out[0][0].op.__class__]
        try:
            pythonic = evaluate_tree(out[-1][0],{},{})
        except:
            out.append(('ERROR',msg))
            return out
        out.append((encode_ast(pythonic),msg))

    elif isinstance(node,ast.BoolOp):
        out = [(node,None)]
        #Need to be careful here...

        #first consider the AND operator
        if isinstance(node.op,ast.And):
            for ix in xrange(len(node.values)):
                #resolve each value in turn
                if out[-1][0] == 'ERROR':
                    break
                res = resolve(node.values[ix],locals,globals)
                last = out[-1][0].values
                front = last[:ix]
                back = last[ix+1:]
                for i in res[1:]:
                    a,msg = i
                    if a == 'ERROR':
                        out.append(i)
                        break
                    l = ast.BoolOp()
                    l.op = out[0][0].op
                    l.ctx = ast.Load()
                    l.values = front + [a] + back
                    out.append((l,msg))
                pythonic = evaluate_tree(a,{},{})
                bval = bool(pythonic)
                #if one of the fully-resolved values isn't a boolean,
                #cast it to one for clarity
                if not isinstance(pythonic,bool):
                    new = bool(pythonic)
                    a = encode_ast(new)
                    l = ast.BoolOp()
                    l.op = out[0][0].op
                    l.ctx = ast.Load()
                    l.values = front + [a] + back
                    out.append((l,"Casting to <tt>bool</tt> type"))
                #if we hit a False, the whole BoolOp is going to resolve to False,
                #WITHOUT CHECKING THE OTHER VALUES
                if bval == False:
                    break
            if not bval:
                out.append((encode_ast(False),"'And' operator"))
```

```
            else:
                out.append((encode_ast(True),"'And' operator"))

        #OR is exactly analogous
        elif isinstance(node.op,ast.Or):
            for ix in xrange(len(node.values)):
                #resolve each value in turn
                if out[-1][0] == 'ERROR':
                    break
                res = resolve(node.values[ix],locals,globals)
                last = out[-1][0].values
                front = last[:ix]
                back = last[ix+1:]
                for i in res[1:]:
                    a,msg = i
                    if a == 'ERROR':
                        out.append(i)
                        break
                    l = ast.BoolOp()
                    l.op = out[0][0].op
                    l.ctx = ast.Load()
                    l.values = front + [a] + back
                    out.append((l,msg))
                pythonic = evaluate_tree(a,{},{})
                bval = bool(pythonic)
                #if one of the fully-resolved values isn't a boolean,
                #cast it to one for clarity
                if not isinstance(pythonic,bool):
                    new = bool(pythonic)
                    a = encode_ast(new)
                    l = ast.BoolOp()
                    l.op = out[0][0].op
                    l.ctx = ast.Load()
                    l.values = front + [a] + back
                    out.append((l,"Casting to <tt>bool</tt> type"))
                #if we hit a True, the whole BoolOp is going to resolve to True,
                #WITHOUT CHECKING THE OTHER VALUES
                if bval == True:
                    break
            if not bval:
                out.append((encode_ast(False),"'Or' operator"))
            else:
                out.append((encode_ast(True),"'Or' operator"))

    elif isinstance(node,ast.Compare):
        out = [(node,None)]
        for i in resolve(node.left,locals,globals)[1:]:
            if i[0] == 'ERROR':
                out.append(i)
                break
            new = ast.Compare()
            new.ops = out[-1][0].ops
            new.ctx = ast.Load()
            new.left = i[0]
            new.comparators = out[-1][0].comparators
            out.append((new,i[1]))
        if out[-1][0] == 'ERROR':
            return out
        for ix in xrange(len(out[-1][0].comparators)):
            if out[-1][0] == 'ERROR':
                break
```

```
            front = out[-1][0].comparators[:ix]
            back = out[-1][0].comparators[ix+1:]
            for i in resolve(node.comparators[ix],locals,globals)[1:]:
                if i[0] == 'ERROR':
                    out.append(i)
                    break
                new = ast.Compare()
                new.ops = out[-1][0].ops
                new.ctx = ast.Load()
                new.left = out[-1][0].left
                new.comparators = front + [i[0]] + back
                out.append((new,i[1]))
        msg = comp_operators[out[0][0].ops[0].__class__] if len(out[0][0].ops) == 1 else "Multiple Comparisons"
        try:
            pythonic = evaluate_tree(out[-1][0],{},{})
        except:
            out.append(('ERROR',msg))
            return out
        out.append((encode_ast(pythonic),msg))

    elif isinstance(node,ast.UnaryOp):
        out = [(node,None)]
        for i in resolve(node.operand,locals,globals)[1:]:
            if i[0] == 'ERROR':
                out.append(i)
                break
            new = ast.UnaryOp()
            new.op = out[-1][0].op
            new.ctx = ast.Load()
            new.operand = i[0]
            out.append((new,i[1]))
        msg = unary_operators[out[0][0].op.__class__]
        try:
            pythonic = evaluate_tree(out[-1][0],{},{})
        except:
            out.append(('ERROR',msg))
            return out
        out.append((encode_ast(pythonic),msg))

    elif isinstance(node,ast.Subscript): #subscript, but assume only single Index
        out = [(node,None)]
        #not sure how best to deal with this.  show whole collection?  for now i'll
        #avoid that.
        for i in resolve(node.slice.value,locals,globals)[1:]:
            if i[0] == 'ERROR':
                out.append(i)
                break
            new = ast.Subscript()
            new.ctx = ast.Load()
            new.value = out[0][0].value
            new.slice = ast.Index()
            new.slice.ctx = ast.Load()
            new.slice.value = i[0]
            out.append((new,i[1]))
        try:
            pythonic = evaluate_tree(out[-1][0],locals,globals) #need vars here!
        except:
            out.append(('ERROR',msg))
            return out
        msg = 'Subscripting'
        out.append((encode_ast(pythonic),msg))
```

```

    return out
```

A.2.6 trees.py

```python
# TREES.PY
# utilities for dealing with trees
# hartz 2012

# This file is a part of CAT-SOOP Detective
# CAT-SOOP Detective is copyright (C) 2012 Adam Hartz.
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

import ast

def downward_search(tree,test_func):
    """
    Search down an AST for a node n such that test_func(n) is True.
    Return that node, or None if no such node exists in the tree.
    """
    try:
        if test_func(tree):
            return tree
    except:
        pass
    for child in ast.iter_child_nodes(tree):
        n = downward_search(child,test_func)
        if n:
            return n
    return None

class HTMLVisitor(ast.NodeVisitor):
    """
    Visitor which walks an AST and returns an HTML Representation of it.

    Borrowed structure from Geoff Reedy (http://stackoverflow.com/users/166955/geoff-reedy)
    Found at http://stackoverflow.com/questions/3867028/converting-a-python-numeric-expression-to-latex

    Precedence based on 5.15 at http://docs.python.org/reference/expressions.html
    """

    def prec(self, n):
        return getattr(self, 'prec_'+n.__class__.__name__, getattr(self, 'generic_prec'))(n)

    #Prepare yourself for an obnoxious enumeration of Classes...

    def visit_Call(self, n):
        func = self.visit(n.func)
        args = ', '.join(map(self.visit, n.args))
        return r'%s<tt>(</tt>%s<tt>)</tt>' % (func, args)

    def prec_Call(self, n):
```

```
            return 1000

    def visit_Name(self, n):
        return "<tt><b>%s</b></tt>" % n.id if n.id not in ('True','False') else "<tt>%s</tt>" % n.id

    def prec_Name(self, n):
        return 1000

    def visit_UnaryOp(self, n):
        if self.prec(n.op) > self.prec(n.operand):
            return r'%s <tt>(</tt>%s<tt>)</tt>' % (self.visit(n.op), self.visit(n.operand))
        else:
            return r'%s %s' % (self.visit(n.op), self.visit(n.operand))

    def prec_UnaryOp(self, n):
        return self.prec(n.op)

    def visit_BinOp(self, n):
        if self.prec(n.op) >= self.prec(n.left):
            left = r'<tt>(</tt>%s<tt>)</tt>' % self.visit(n.left)
        else:
            left = self.visit(n.left)
        if self.prec(n.op) >= self.prec(n.right):
            right = r'<tt>(</tt>%s<tt>)</tt>' % self.visit(n.right)
        else:
            right = self.visit(n.right)
        if isinstance(n.op, ast.Div):
            return r'%s <tt>/</tt> %s' % (left, right)
        elif isinstance(n.op, ast.FloorDiv):
            return r'%s <tt>//</tt> %s' % (left, right)
        elif isinstance(n.op, ast.Pow):
            return r'%s<tt>**</tt>%s' % (left, right)
        else:
            return r'%s %s %s' % (left, self.visit(n.op), right)

    def visit_BoolOp(self,n):
        opstr = self.visit(n.op)
        vals = []
        for i in n.values:
            if self.prec(i) < self.prec(n.op):
                thingy = r'<tt>(</tt>%s<tt>)</tt>' % self.visit(i)
            else:
                thingy = self.visit(i)
            vals.append(thingy)
        return (' '+opstr+' ').join(vals)

    def prec_BinOp(self, n):
        return self.prec(n.op)

    def visit_Sub(self, n):
        return '<tt>-</tt>'

    def prec_Sub(self, n):
        return 8

    def visit_Add(self, n):
        return '<tt>+</tt>'

    def prec_Add(self, n):
        return 8
```

```
    def visit_Mult(self, n):
        return '<tt>*</tt>'

    def prec_Mult(self, n):
        return 9

    def visit_Mod(self, n):
        return '<tt>%%</tt>'

    def prec_Mod(self, n):
        return 9

    def prec_Pow(self, n):
        return 11

    def prec_Div(self, n):
        return 9

    def prec_FloorDiv(self, n):
        return 9

    def visit_LShift(self, n):
        return '<tt>%lt;%lt;</tt>'

    def prec_LShift(self,n):
        return 7

    def visit_RShift(self, n):
        return '<tt>&gt;&gt;</tt>'

    def prec_RShift(self,n):
        return 7

    def visit_BitOr(self, n):
        return '<tt>|</tt>'

    def prec_BitOr(self,n):
        return 5

    def visit_BitXor(self, n):
        return '<tt>^</tt>'

    def prec_BitXor(self,n):
        return 6

    def visit_BitAnd(self, n):
        return '<tt>&</tt>'

    def prec_BitAnd(self,n):
        return 6.5

    def visit_Invert(self, n):
        return '<tt>~</tt>'

    def prec_Invert(self, n):
        return 10

    def visit_And(self, n):
        return '<tt>and</tt>'

    def prec_And(self, n):
```

```python
        return 2

    def visit_Or(self, n):
        return '<tt>or</tt>'

    def prec_Or(self, n):
        return 1

    def visit_Not(self, n):
        return '<tt>not</tt>'

    def prec_Not(self, n):
        return 3

    def visit_UAdd(self, n):
        return ''

    def prec_UAdd(self, n):
        return 10

    def visit_USub(self, n):
        return '<tt>-</tt>'

    def prec_USub(self, n):
        return 10

    def visit_Num(self, n):
        return "<tt>%s</tt>" % str(n.n)

    def prec_Num(self, n):
        return 1000

    def visit_List(self,l):
        return '<tt>[</tt>' + "<tt>, </tt>".join([self.visit(i) for i in l.elts])  +  '<tt>]</tt>'

    def prec_List(self,l):
        return 1000

    def visit_Tuple(self,l):
        if len(l.elts) == 0:
            return "tuple()"
        return '<tt>(</tt>' + "<tt>, </tt>".join([self.visit(i) for i in l.elts]) + "<tt>, </tt>" if len(l.elts) > 1 else ""  +  '<tt>)</tt>'

    def prec_Tuple(self,l):
        return 1000

    def visit_Dict(self,d):
        return '<tt>{</tt>' + "<tt>, </tt>".join(["%s<tt>: </tt>%s" % (self.visit(k),self.visit(v)) for (k,v) in zip(d.keys,d.values)]) + '<tt>}</tt>'

    def prec_Dict(self,l):
        return 1000

    def visit_Compare(self,c):
        return self.visit(c.left) + " " + " ".join(["%s %s" % (self.visit(a),self.visit(b)) for (a,b) in zip(c.ops,c.comparators)])

    def prec_Compare(self,n):
        return 4
```

```
    prec_Lt = prec_LtE = prec_Gt = prec_GtE = prec_Eq = prec_NotEq = prec_Is = prec_IsNot = prec_In = prec_NotIn = prec_Compare

    def visit_Lt(self,n):
        return "<tt>&lt;</tt>"

    def visit_LtE(self,n):
        return "<tt>&lt;=</tt>"

    def visit_Gt(self,n):
        return "<tt>&gt;</tt>"

    def visit_GtE(self,n):
        return "<tt>&gt;=</tt>"

    def visit_Eq(self,n):
        return "<tt>==</tt>"

    def visit_NotEq(self,n):
        return "<tt>!=</tt>"

    def visit_Is(self,n):
        return "<tt>is</tt>"

    def visit_IsNot(self,n):
        return "<tt>is not</tt>"

    def visit_In(self,n):
        return "<tt>in</tt>"

    def visit_NotIn(self,n):
        return "<tt>not in</tt>"

    def visit_Str(self,s):
        return "<tt>%s</tt>" % repr(s.s)

    def visit_Subscript(self,s):
        return "%s<tt>[</tt>%s<tt>]</tt>" % (self.visit(s.value),self.visit(s.slice.value))

    def prec_Subscript(self,n):
        return 1000

    def prec_Str(self,n):
        return 1000

    def generic_visit(self, n):
        if isinstance(n, ast.AST):
            return r'%s<tt>(</tt>%s<tt>)</tt>' % (n.__class__.__name__, '<tt>, </tt>'.join(map(self.visit, [getattr(n, f) for f in n._fields])))
        else:
            return str(n)

    def generic_prec(self, n):
        return 0
```

Bibliography

[1] Joseph Beck, Mia Stern, and Beverly Park Woolf. Using the student model to control problem difficulty. In *In Proceedings of the Seventh International Conference on User Modeling*, pages 277–288. Springer, 1997.

[2] Benjamin S Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Leadership*, 41(8):4–17, 1984.

[3] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, March 1964.

[4] Sidney D'Mello, Rosalind W. Picard, and Arthur Graesser. Toward an affect-sensitive autotutor. *IEEE Intelligent Systems*, 22(4):53–61, July 2007.

[5] Martin Dougiamas and Peter Taylor. Moodle: Using learning communities to create an open source course management system. In David Lassner and Carmel McNaught, editors, *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2003*, pages 171–178, Honolulu, Hawaii, USA, 2003. AACE.

[6] Arthur C. Graesser, Kurt Vanlehn, Carolyn P. Ros, Pamela W. Jordan, and Derek Harter. Intelligent tutoring systems with conversational dialogue. *AI Magazine*, 22:39–51, 2001.

[7] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE '03, pages 153–156, New York, NY, USA, 2003. ACM.

[8] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13(3):259 – 290, 2002.

[9] Wei Jin, Lorrie Lehmann, Matthew Johnson, Michael Eagle, Behrooz Mostafavi, Tiffany Barnes, and John C Stamper. Towards automatic hint generation for a data-driven novice programming tutor. In *17th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2011.

[10] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301 – 310, New York, NY, USA, 2008. ACM.

[11] Gerd Kortemeyer, Guy Albertelli, Wolfgang Bauer, Felicia Berryman, Bowers Matthew Hall, William F. Punch, Er Sakharuk, Cheryl Speier, and Gerd Kortemeyer. The LearningOnline Network with Computer-Assisted Personalized Approach (LON-CAPA), 2003.

[12] Bob Lang. Teaching new programmers: a Java tool set as a student teaching aid. In Wizard V. Oz and Mihalis Yannakakis, editors, *Proceedings of the Inaugural Conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate Representation Engineering for Virtual Machines 2002*, pages 95 – 100. National University of Ireland, 2002.

[13] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[14] J A Michael. Students' misconceptions about perceived physiological responses. *Advances in Physiology Education*, 274(6):S90–8, 1998.

[15] Joel Michael. Where's the evidence that active learning works? *Advances in Physiology Education*, 30(4):159–167, December 2006.

[16] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '04, pages 373–376, New York, NY, USA, 2004. ACM.

[17] Elsa-Sofia Morote, David Kokorowski, and David Pritchard. Cybertutor, a socratic web-based homework tutor. In Margaret Driscoll and Thomas C. Reeves, editors, *Proceedings of World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2002*, pages 2711–2712, Montreal, Canada, 2002. AACE.

[18] Niko Myller. Automatic generation of prediction questions during program visualization. *Electronic Notes in Theoretical Computer Science*, 178(0):43 – 49, 2007.

[19] Committee on Developments in the Science of Learning with additional material from the Committee on Learning Research and National Research Council Educational Practice. *How People Learn: Brain, Mind, Experience, and School: Expanded Edition*, chapter 2. The National Academies Press, 2000.

[20] S.L. Pressey. A simple device which gives tests and scores – and teaches. *School and Society*, 23:373–376, 1926.

[21] Michael Striewe and Michael Goedicke. Using run time traces in automated programming tutoring. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 303–307, New York, NY, USA, 2011. ACM.

[22] Rasil Warnakulasooriya and David E. Pritchard. Hints really help! Available at `http://relate.mit.edu/wp-content/uploads/2012/02/hints.pdf`.