Chapter 2
Programming in Python

Depending on your previous programming background, we recommend different paths through the available readings:

- **If you have never programmed before:** you should start with a general introduction to programming and Python. We recommend *Python for Software Design: How to Think Like a Computer Scientist*, by Allen Downey. This is a good introductory text that uses Python to present basic ideas of computer science and programming. It is available for purchase in hardcopy, or as a free download from

  `http://www.greenteapress.com/thinkpython`

  After that, you can go straight to the next chapter.
- **If you have programmed before, but not in Python:** you should read the rest of this chapter for a quick overview of what's new in Python.
- **If you have programmed in Python:** you should skip to the next chapter.

Everyone should have a bookmark in their browser for *Python Tutorial*, by Guido Van Rossum. This is the standard tutorial reference by the inventor of Python.
`http://docs.python.org/tut/tut.html`.

In the rest of this chpater, we will assume you know how to program in some language, but are new to Python. We'll use Java as an informal running comparative example. In this section we will cover what we think are the most important differences between Python and what you already know about programming; but these notes are by no means complete.

2.1 Using Python

Python is designed to be easy for a user to interact with. It comes with an interactive mode called a *listener* or *shell*. The shell gives a prompt (usually something like >>>) and waits for you to type in a Python expression or program. Then it will evaluate the expression you typed in and print out the value of the result. So, for example, an interaction with the Python shell might look like this:

```
>>> 5 + 5
10
>>> x = 6
>>> x
6
>>> x + x
12
>>> y = 'hi'
>>> y + y
'hihi'
>>>
```

So, you can use Python as a fancy calculator. And as you define your own procedures in Python, you can use the shell to test them or use them to compute useful results.

2.1.1 Indentation and line breaks

Every programming language has to have some method for indicating grouping. Here's how you write an if-then-else structure in Java:

```
    if (s == 1){
        s = s + 1;
        a = a - 10;
    } else {
        s = s + 10;
        a = a + 10;
    }
```

The braces specify what statements are executed in the `if` case. It is considered good style to indent your code to agree with the brace structure, but it isn't required. In addition, the semicolons are used to indicate the end of a statement, independent of where the line breaks in the file are. So, the following code fragment has the same meaning as the previous one.

```
    if (s == 1){
s = s
+ 1;     a = a - 10;
    } else {
                s = s + 10;
    a = a + 10;
    }
```

In Python, on the other hand, there are no braces for grouping or semicolons for termination. Indentation indicates grouping and line breaks indicate statement termination. So, in Python, we'd write the previous example as

```
    if s == 1:
        s = s + 1
        a = a - 10
    else:
        s = s + 10
        a = a + 10
```

There is no way to put more than one statement on a single line.[3] If you have a statement that's too long for a line, you can signal it with a backslash:

```
aReallyLongVariableNameThatMakesMyLinesLong = \
        aReallyLongVariableNameThatMakesMyLinesLong + 1
```

It's easy for Java programmers to get confused about colons and semi-colons in Python. Here's the deal: (1) Python doesn't use semi-colons; (2) Colons are used to start an indented block, so they appear on the first line of a procedure definition, when starting a `while` or `for` loop, and after the condition in an `if`, `elif`, or `else`.

Is one method better than the other? No. It's entirely a matter of taste. The Python method is pretty unusual. But if you're going to use Python, you need to remember about indentation and line breaks being significant.

2.1.2 Types and declarations

Java programs are what is known as *statically and strongly typed*. That means that the types of all the variables must be known at the time that the program is written. This means that variables have to be declared to have a particular type before they're used. It also means that the variables can't be used in a way that is inconsistent with their type. So, for instance, you'd declare `x` to be an integer by saying

```
int x;
x = 6 * 7;
```

But you'd get into trouble if you left out the declaration, or did

```
int x;
x = "thing";
```

because a *type checker* is run on your program to make sure that you don't try to use a variable in a way that is inconsistent with its declaration.

In Python, however, things are a lot more flexible. There are no variable declarations, and the same variable can be used at different points in your program to hold data objects of different types. So, this is fine, in Python:

```
if x == 1:
    x = 89.3
else:
    x = "thing"
```

[3] Actually, you can write something like `if a > b:  a = a + 1` all on one line, if the work you need to do inside an `if` or a `for` is only one line long.

The advantage of having type declarations and compile-time type checking, as in Java, is that a compiler can generate an executable version of your program that runs very quickly, because it can be sure what kind of data is stored in each variable, and doesn't have to check it at runtime. An additional advantage is that many programming mistakes can be caught at compile time, rather than waiting until the program is being run. Java would complain even before your program started to run that it couldn't evaluate

```
    3 + "hi"
```

Python wouldn't complain until it was running the program and got to that point.

The advantage of the Python approach is that programs are shorter and cleaner looking, and possibly easier to write. The flexibility is often useful: In Python, it's easy to make a list or array with objects of different types stored in it. In Java, it can be done, but it's trickier. The disadvantage of the Python approach is that programs tend to be slower. Also, the rigor of compile-time type checking may reduce bugs, especially in large programs.

2.1.3 Modules

As you start to write bigger programs, you'll want to keep the procedure definitions in multiple files, grouped together according to what they do. So, for example, we might package a set of utility functions together into a single file, called `utility.py`. This file is called a `module` in Python.

Now, if we want to use those procedures in another file, or from the the Python shell, we'll need to say

```
import utility
```

Now, if we have a procedure in `utility.py` called `foo`, we can use it in this program with the name `utility.foo`. You can read more about modules in the Python documentation.

2.1.4 Interaction and Debugging

We encourage you to adopt an interactive style of programming and debugging. Use the Python shell a lot. Write small pieces of code and test them. It's much easier to test the individual pieces as you go, rather than to spend hours writing a big program, and then find it doesn't work, and have to sift through all your code, trying to find the bugs.

But, if you find yourself in the (inevitable) position of having a big program with a bug in it, don't despair. Debugging a program doesn't require brilliance or creativity or much in the way of insight. What it requires is persistence and a systematic approach.

First of all, have a test case (a set of inputs to the procedure you're trying to debug) and know what the answer is supposed to be. To test a program, you might start with some special cases:

what if the argument is 0 or the empty list? Those cases might be easier to sort through first (and are also cases that can be easy to get wrong). Then try more general cases.

Now, if your program gets your test case wrong, what should you do? Resist the temptation to start changing your program around, just to see if that will fix the problem. Don't change any code until you know what's wrong with what you're doing now, and therefore believe that the change you make is going to correct the problem.

Ultimately, for debugging big programs, it's most useful to use a software development environment with a serious debugger. But these tools can sometimes have a steep learning curve, so in this class we'll learn to debug systematically using "print" statements.

Start putting print statements at all the interesting spots in your program. Print out intermediate results. Know what the answers are supposed to be, for your test case. Run your test case, and find the first place that something goes wrong. You've narrowed in on the problem. Study that part of the code and see if you can see what's wrong. If not, add some more print statements, and run it again. **Don't try to be smart....be systematic and indefatigable!**

You should learn enough of Python to be comfortable writing basic programs, and to be able to efficiently look up details of the language that you don't know or have forgotten.

2.2 Procedures

In Python, the fundamental abstraction of a computation is as a procedure (other books call them "functions" instead; we'll end up using both terms). A procedure that takes a number as an argument and returns the argument value plus 1 is defined as:

```
def f(x):
    return x + 1
```

The indentation is important here, too. All of the statements of the procedure have to be indented one level below the `def`. It is crucial to remember the `return` statement at the end, if you want your procedure to return a value. So, if you defined `f` as above, then played with it in the shell,[4] you might get something like this:

```
>>> f
<function f at 0x82570>
>>> f(4)
5
>>> f(f(f(4)))
7
```

[4] Although you can type procedure definitions directly into the shell, you won't want to work that way, because if there's a mistake in your definition, you'll have to type the whole thing in again. Instead, you should type your procedure definitions into a file, and then get Python to evaluate them. Look at the documentation for Idle or the 6.01 FAQ for an explanation of how to do that.

If we just evaluate `f`, Python tells us it's a function. Then we can apply it to 4 and get 5, or apply it multiple times, as shown.

What if we define

```
def g(x):
    x + 1
```

Now, when we play with it, we might get something like this:

```
>>> g(4)
>>> g(g(4))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in g
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

What happened!! First, when we evaluated `g(4)`, we got nothing at all, because our definition of `g` didn't return anything. Well...strictly speaking, it returned a special value called `None`, which the shell doesn't bother printing out. The value `None` has a special type, called `NoneType`. So, then, when we tried to apply `g` to the result of `g(4)`, it ended up trying to evaluate `g(None)`, which made it try to evaluate `None + 1`, which made it complain that it didn't know how to add something of type `NoneType` and something of type `int`.

Whenever you ask Python to do something it can't do, it will complain. You should learn to read the error messages, because they will give you valuable information about what's wrong with what you were asking for.

Print vs Return

Here are two different function definitions:

```
def f1(x):
    print x + 1
def f2(x):
    return x + 1
```

What happens when we call them?

```
>>> f1(3)
4
>>> f2(3)
4
```

It looks like they behave in exactly the same way. But they don't, really. Look at this example:

```
>>> print(f1(3))
4
None
>>> print(f2(3))
4
```

In the case of `f1`, the function, when evaluated, prints 4; then it returns the value `None`, which is printed by the Python shell. In the case of `f2`, it doesn't print anything, but it returns 4, which is printed by the Python shell. Finally, we can see the difference here:

```
>>> f1(3) + 1
4
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
>>> f2(3) + 1
5
```

In the first case, the function doesn't return a value, so there's nothing to add to 1, and an error is generated. In the second case, the function returns the value 4, which is added to 1, and the result, 5, is printed by the Python read-eval-print loop.

The book *Think Python*, which we recommend reading, was translated from a version for Java, and it has a lot of `print` statements in it, to illustrate programming concepts. But for just about everything we do, it will be returned values that matter, and printing will be used only for debugging, or to give information to the user.

Print is very useful for debugging. It's important to know that you can print out as many items as you want in one line:

```
>>> x = 100
>>> print 'x', x, 'x squared', x*x, 'xiv', 13
x 100 x squared 10000 xiv 13
```

We've also snuck in another data type on you: strings. A string is a sequence of characters. You can create a string using single or double quotes; and access individual elements of strings using indexing (*note that the indexing starts with 0*).

```
>>> s1 = 'hello world'
>>> s2 = "hello world"
>>> s1 == s2
True
>>> s1[3]
'l'
```

Look in the Python documentation for more about strings.

2.3 Control structures

Python has control structures that are slightly different from those in other languages.

2.3.1 Conditionals

Booleans

Before we talk about conditionals, we need to clarify the Boolean data type. It has values `True` and `False`. Typical expressions that have Boolean values are numerical comparisons:

```
>>> 7 > 8
False
>>> -6 <= 9
True
```

We can also test whether data items are equal to one another. Generally we use `==` to test for equality. It returns `True` if the two objects have equal values. Sometimes, however, we will be interested in knowing whether the two items are the exact same object (in the sense discussed in ??structuredData). In that case we use `is`:

```
>>> [1, 2] == [1, 2]
True
>>> [1, 2] is [1, 2]
False
```

In addition, we can combine Boolean values conveniently using `and`, `or`, and `not`:

```
>>> 7 > 8 or 8 > 7
True
>>> not 7 > 8
True
>>> 7 == 7 and 8 > 7
True
```

If

Basic conditional statements have the form:[5]

```
if <booleanExpr>:
    <statementT1>
    ...
    <statementTk>
else:
```

[5] See the Python documentation for more variations.

```
    <statementF1>
    ...
    <statementFn>
```

When the interpreter encounters a conditional statement, it starts by evaluating `<booleanExpr>`, getting either `True` or `False` as a result.[6] If the result is `True`, then it will evaluate `<statementT1>,...,<statementTk>`; if it is `False`, then it will evaluate `<statementF1>,...,<statementFn>`. Crucially, it always evaluates only one set of the statements.

Now, for example, we can implement a procedure that returns the absolute value of its argument.

```
def abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

We could also have written

```
def abs(x):
    if x >= 0:
        result = x
    else:
        result = -x
    return result
```

Python uses the level of indentation of the statements to decide which ones go in the groups of statements governed by the conditionals; so, in the example above, the `return result` statement is evaluated once the conditional is done, no matter which branch of the conditional is evaluated.

For and While

If we want to do some operation or set of operations several times, we can manage the process in several different ways. The most straightforward are `for` and `while` statements (often called `for` and `while` loops).

A `for` loop has the following form:

```
for <var> in <listExpr>:
    <statement1>
    ...
    <statementn>
```

The interpreter starts by evaluating `listExpr`. If it does not yield a list, tuple, or string[7], an error occurs. The block of statements will then, under normal circumstances, be executed one time for

[6] In fact, Python will let you put any expression in the place of `<booleanExpr>`, and it will treat the values `0`, `0.0`, `[]`, `''`, and `None` as if they were `False` and everything else as `True`.

[7] or, more esoterically, another object that can be iterated over.

every value in that list. At the end, the variable `<var>` will remain bound to the last element of the list (and if it had a useful value before the `for` was evaluated, that value will have been permanently overwritten).

Here is a basic `for` loop:

```
result = 0
for x in [1, 3, 4]:
    result = result + x * x
```

At the end of this execution, `result` will have the value 26.

One situation in which the body is not executed once for each value in the list is when a `return` statement is encountered. No matter whether `return` is nested in a loop or not, if it is evaluated it immediately causes a value to be returned from a loop. So, for example, we might write a procedure that tests to see if an item is a member of a list, and returns `True` if it is and `False` if it is not, as follows:

```
def member(x, items):
    for i in items:
        if x == i:
            return True
    return False
```

The procedure loops through all of the elements in `items`, and compares them to `x`. As soon as it finds an item `i` that is equal to `x`, it can quit and return the value `True` from the procedure. If it gets all the way through the loop without returning, then we know that `x` is not in the list, and we can return `False`.

Exercise 2.1. Write a procedure that takes a list of numbers, `nums`, and a limit, `limit`, and returns a list which is the shortest prefix of `nums` the sum of whose values is greater than `limit`. Use `for`. Try to avoid using explicit indexing into the list. (Hint: consider the strategy we used in `member`.)

Range

Very frequently, we will want to iterate through a list of integers, often as indices. Python provides a useful procedure, `range`, which returns lists of integers. It can be used in complex ways, but the basic usage is `range(n)`, which returns a list of integers going from 0 to up to, but not including, its argument. So `range(3)` returns `[0, 1, 2]`.

Exercise 2.2. Write a procedure that takes `n` as an argument and returns the sum of the squares of the integers from 1 to n-1. It should use `for` and `range`.

Exercise 2.3. What is wrong with this procedure, which is supposed to return `True` if the element `x` occurs in the list `items`, and `False` otherwise?

```
def member (x, items):
    for i in items:
        if x == i:
            return True
        else:
            return False
```

While

You should use `for` whenever you can, because it makes the structure of your loops clear. But sometimes you need to do an operation several times, but you don't know in advance how many times it needs to be done. In such situations, you can use a `while` statement, of the form:

```
while <booleanExpr>:
    <statement1>
    ...
    <statementn>
```

In order to evaluate a `while` statement, the interpreter evaluates `<booleanExpr>`, getting a Boolean value. If the value is `False`, it skips all the statements and evaluation moves on to the next statement in the program. If the value is `True`, then the statements are executed, and the `<booleanExpr>` is evaluated again. If it is `False`, execution of the loop is terminated, and if it is `True`, it goes around again.

It will generally be the case that you initialize a variable before the `while` statement, change that variable in the course of executing the loop, and test some property of that variable in the Boolean expression. Imagine that you wanted to write a procedure that takes an argument `n` and returns the largest power of 2 that is smaller than `n`. You might do it like this:

```
def pow2Smaller(n):
    p = 1
    while p*2 < n:
        p = p*2
    return p
```

List Comprehensions

Python has a very nice built-in facility for doing many iterative operations, called *list comprehensions*. The basic template is

```
[<resultExpr> for <var> in <listExpr> if <conditionExpr>]
```

where `<var>` is a single variable (or a tuple of variables), `<listExpr>` is an expression that evaluates to a list, tuple, or string, and `<resultExpr>` is an expression that may use the variable `<var>`.

You can view a list comprehension as a special notation for a particular, very common, class of `for` loops. It is equivalent to the following:

```
*resultVar* = []
for <var> in <listExpr>:
    if <conditionExpr>:
        *resultVar*.append(<resultExpr>)
*resultVar*
```

We used a kind of funny notation `*resultVar*` to indicate that there is some anonymous list that is getting built up during the evaluation of the list comprehension, but we have no real way of accessing it. The result is a list, which is obtained by successively binding `<var>` to elements of the result of evaluating `<listExpr>`, testing to see whether they meet a condition, and if they meet the condition, evaluating `<resultExpr>` and collecting the results into a list.

Whew. It's probably easier to understand it by example.

```
>>> [x/2.0 for x in [4, 5, 6]]
[2.0, 2.5, 3.0]
>>> [y**2 + 3 for y in [1, 10, 1000]]
[4, 103, 1000003]
>>> [a[0] for a in [['Hal', 'Abelson'],['Jacob','White'],
                    ['Leslie','Kaelbling']]]
['Hal', 'Jacob', 'Leslie']
>>> [a[0]+'!' for a in [['Hal', 'Abelson'],['Jacob','White'],
                        ['Leslie','Kaelbling']]]
['Hal!', 'Jacob!', 'Leslie!']
```

Imagine that you have a list of numbers and you want to construct a list containing just the ones that are odd. You might write

```
>>> nums = [1, 2, 5, 6, 88, 99, 101, 10000, 100, 37, 101]
>>> [x for x in nums if x%2==1]
[1, 5, 99, 101, 37, 101]
```

And, of course, you can combine this with the other abilities of list comprehensions, to, for example, return the squares of the odd numbers:

```
>>> [x*x for x in nums if x%2==1]
[1, 25, 9801, 10201, 1369, 10201]
```

You can also use structured assignments in list comprehensions

```
>>> [first for (first, last) in [['Hal', 'Abelson'],['Jacob','White'],
                    ['Leslie','Kaelbling']]]
['Hal', 'Jacob', 'Leslie']
```

```
>>> [first+last for (first, last) in [['Hal', 'Abelson'],['Jacob','White'],
                   ['Leslie','Kaelbling']]]
['HalAbelson', 'JacobWhite', 'LeslieKaelbling']
```

Another built-in function that is useful with list comprehensions is `zip`. Here are some examples of how it works:

```
> zip([1, 2, 3],[4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
> zip([1,2], [3, 4], [5, 6])
[(1, 3, 5), (2, 4, 6)]
```

Here's an example of using with a list comprehension:

```
>>> [first+last for (first, last) in zip(['Hal', 'Jacob', 'Leslie'],
                                         ['Abelson','White','Kaelbling'])]
['HalAbelson', 'JacobWhite', 'LeslieKaelbling']
```

Note that this last example is very different from this one:

```
>>> [first+last for first in ['Hal', 'Jacob', 'Leslie'] \
                for last in ['Abelson','White','Kaelbling']]
['HalAbelson', 'HalWhite', 'HalKaelbling', 'JacobAbelson', 'JacobWhite',
'JacobKaelbling', 'LeslieAbelson', 'LeslieWhite', 'LeslieKaelbling']
```

Nested list comprehensions behave like nested `for` loops, the expression in the list comprehension is evaluated for every combination of the values of the variables.

Lists

Python has a built-in list data structure that is easy to use and incredibly convenient. So, for instance, you can say

```
>>> y = [1, 2, 3]
>>> y[0]
1
>>> y[2]
3
>>> len(y)
3
>>> y + [4]
[1, 2, 3, 4]
>>> [4] + y
[4, 1, 2, 3]
>>> [4,5,6] + y
[4, 5, 6, 1, 2, 3]
>>> y
[1, 2, 3]
```

A list is written using square brackets, with entries separated by commas. You can get elements out by specifying the index of the element you want in square brackets, but note that, like for strings, the indexing starts with 0.

You can add elements to a list using `'+'`, taking advantage of Python operator overloading. Note that this operation does not change the original list, but makes a new one.

Another useful thing to know about lists is that you can make *slices* of them. A slice of a list is sublist; you can get the basic idea from examples.

```
>>> b = range(10)
>>> b
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b[1:]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> b[:7]
[0, 1, 2, 3, 4, 5, 6]
>>> b[:-1]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> b[:-2]
[0, 1, 2, 3, 4, 5, 6, 7]
```

Iteration over lists

What if you had a list of integers, and you wanted to add them up and return the sum? Here are a number of different ways of doing it.[8]

First, here is a version in a style you might have learned to write in a Java class (actually, you would have used `for`, but Python doesn't have a `for` that works like the one in C and Java).

```
def addList1(l):
    sum = 0
    listLength = len(l)
    i = 0
    while (i < listLength):
        sum = sum + l[i]
        i = i + 1
    return sum
```

[8] For any program you'll ever need to write, there will be a huge number of different ways of doing it. How should you choose among them? The most important thing is that the program you write be correct, and so you should choose the approach that will get you to a correct program in the shortest amount of time. That argues for writing it in the way that is cleanest, clearest, shortest. Another benefit of writing code that is clean, clear and short is that you will be better able to understand it when you come back to it in a week or a month or a year, and that other people will also be better able to understand it. Sometimes, you'll have to worry about writing a version of a program that runs very quickly, and it might be that in order to make that happen, you'll have to write it less cleanly or clearly or briefly. But it's important to have a version that's correct before you worry about getting one that's fast.

It increments the index `i` from 0 through the length of the list - 1, and adds the appropriate element of the list into the sum. This is perfectly correct, but pretty verbose and easy to get wrong.

Here's a version using Python's `for` loop.

```
def addList2(l):
    sum = 0
    for i in range(len(l)):
        sum = sum + l[i]
    return sum
```

A loop of the form

```
for x in l:
   something
```

will be executed once for each element in the list `l`, with the variable `x` containing each successive element in `l` on each iteration. So,

```
for x in range(3):
    print x
```

will print `0 1 2`. Back to `addList2`, we see that `i` will take on values from 0 to the length of the list minus 1, and on each iteration, it will add the appropriate element from `l` into the sum. This is more compact and easier to get right than the first version, but still not the best we can do!

This one is even more direct.

```
def addList3(l):
    sum = 0
    for v in l:
        sum = sum + v
    return sum
```

We don't ever really need to work with the indices. Here, the variable `v` takes on each successive value in `l`, and those values are accumulated into `sum`.

For the truly lazy, it turns out that the function we need is already built into Python. It's called `sum`:

```
def addList4(l):
    return sum(l)
```

In **section A.2**, we'll see another way to do `addList`, which many people find more beautiful than the methods shown here.