Chapter 2
Learning to Program 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 Python, and how it may differ from other programming languages with which you are familiar.
- **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. It is accessible at:

`http://docs.python.org/tut/tut.html`

In the rest of this chapter, we will assume you know how to program in some language, but are new to Python. We will 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 may already know about programming; but these notes are by no means complete.

2.1 Using Python

Python is designed for easy interaction between a user and the computer. 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 entered, 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 of instructions. Here is 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 is not required. In addition, the semi-colons are used to indicate the end of a statement, independent of the locations of the line breaks in the file. So, the following code fragment has the same meaning as the previous one, although it is much harder to read and understand.

```
    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 would 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 is too long for a line, you can signal it with a backslash:

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

It is easy for Java programmers to get confused about colons and semi-colons in Python. Here is the deal: (1) Python does not 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 is entirely a matter of taste. The Python method is pretty unusual. But if you are going to use Python, you need to remember that indentation and line breaks are significant.

2.1.2 Types and declarations

Java programs are what is known as *statically and strongly typed*. Thus, 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 are used. It also means that the variables cannot be used in a way that is inconsistent with their type. So, for instance, you would declare `x` to be an integer by saying

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

But you would 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, the following is fine, in Python:

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

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 certain about what kind of data is stored in each variable, and it does not have to check it at runtime. An additional advantage is that many programming mistakes can be caught at compile

[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.

time, rather than waiting until the program is being run. Java would complain even before your program started to run that it could not evaluate

```
    3 + "hi"
```

Python would not 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 is easy to make a list or array with objects of different types stored in it. In Java, it can be done, but it is 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 will 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 will need to say

```
import utility
```

so that all those procedures become available to us and to the Python interpereter. Now, if we have a procedure in `utility.py` called `foo`, we can use it with the name `utility.foo`. You can read more about modules, and how to reference procedures defined in 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 is much easier to test the individual pieces as you go, rather than to spend hours writing a big program, and then find it does not 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, do not despair. Debugging a program does not 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 are 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. Do not change any code until you know what is wrong with what you are doing now, and therefore believe that the change you make is going to correct the problem.

Ultimately, for debugging big programs, it is 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 will learn to debug systematically using "print" statements.

One good way to use "print" statements to help in debugging is to use a variation on binary search. Find a spot roughly halfway through your code at which you can predict the values of variables, or intermediate results your computation. Put a print statement there that lists expected as well as actual values of the variables. Run your test case, and check. If the predicted values match the actual ones, it is likely that the bug occurs after this point in the code; if they do not, then you have a bug prior to this point (of course, you might have a second bug after this point, but you can find that later). Now repeat the process by finding a location halfway between the beginning of the procedure and this point, placing a print statement with expected and actual values, and continuing. In this way you can narrow down the location of the bug. Study that part of the code and see if you can see what is wrong. If not, add some more print statements near the problematic part, 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 will 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
```

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

What if we define

[4] Although you can type procedure definitions directly into the shell, you will not want to work that way, because if there is a mistake in your definition, you will 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.

```
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` did not return anything. Well...strictly speaking, it returned a special value called `None`, which the shell does not 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 did not know how to add something of type `NoneType` and something of type `int`.

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

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 does not 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 does not return a value, so there is 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 is 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', 14
x 100 x squared 10000 xiv 14
```

We have 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.

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

As you can see, indexing refers to the extraction of a particular element of a string, by using square brackets `[i]` where `i` is a number that identifies the location of the character that you wish to extract (*note that the indexing starts with 0*).

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 **section 3.3**). In that case we use `is`:

```
>>> [1, 2] == [1, 2]
True
>>> [1, 2] is [1, 2]
False
>>> a = [1, 2]
>>> b = [1, 2]
>>> c = a
>>> a == b
True
>>> a is b
False
>>> a == c
True
>>> a is c
True
```

Thus, in the examples above, we see that `==` testing can be applied to nested structures, and basically returns true if each of the individual elements is the same. However, `is` testing, especially when applied to nested structures, is more refined, and only returns `True` if the two objects point to exactly the same instance in memory.

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:
    <statementF1>
```

[5] See the Python documentation for more variations.

```
    ...
    <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. If it does yield a list or list-like structure, then the block of statements will, under normal circumstances, be executed one time for 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:

[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.

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

At the end of this execution, `result` will have the value 26, and `x` will have the value 4.

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 procedure call. 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 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. Sometimes, however, you need to do an operation several times, but you do not 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
```

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
>>> y[-1]
```

```
3
>>> y[-2]
2
>>> 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. Note that you can index elements of a list starting from the initial (or zeroth) one (by using integers), or starting from the last one (by using negative integers).

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]

[8] For any program you will 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 will 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 will have to write it less cleanly or clearly or briefly. But it is important to have a version that is correct before you worry about getting one that is fast.

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 does not 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
```

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 is 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 do not 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 is called `sum`:

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

In section ??, we will see another way to do `addList`, which many people find more beautiful than the methods shown here.

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>`. The `if <conditionExpr>` is optional; if it is present, then only those values of `<var>` for which that expression is `True` are included in the resulting computation.

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 is 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]
```

Note the use of the `if` conditional here to include only particular values of `x`.

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 is an example of using `zip` 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.

2.4 Common Errors and Messages

Here are some common Python errors and error messages to watch out for. Please let us know if you have any favorite additions for this list.

- A complaint about `NoneType` often means you forgot a return.

```
def plus1 (x):
    x + 1
>>> y = plus1(x)
>>> plus1(x) + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

- Weird results from math can result from integer division

```
>>> 3/ 9
0
```

- “Unsubscriptable object” means you are trying to get an element out of something that isn’t a dictionary, list, or tuple.

```
>>> x = 1
>>> x[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is unsubscriptable
```

- “Object is not callable” means you are trying to use something that isn’t a procedure or method as if it were.

```
>>> x = 1
>>> x(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

- “List index out of range” means you are trying to read or write an element of a list that is not present.

```
>>> a = range(5)
>>> a
[0, 1, 2, 3, 4]
>>> a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- “Maximum recursion depth exceeded” means you have a recursive procedure that is nested *very* deeply or your base case is not being reached due to a bug.

```
def fizz(x):
    return fizz(x - 1)
>>> fizz(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fizz
  File "<stdin>", line 2, in fizz
...
  File "<stdin>", line 2, in fizz
RuntimeError: maximum recursion depth exceeded
```

- “Key Error” means that you are trying to look up an element in a dictionary that is not present.

```
>>> d = {'a':7, 'b':8}
>>> d['c']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
```

- Another common error is forgetting the `self` before calling a method. This generates the same error that you would get if you tried to call a function that wasn't defined at all.

  ```
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "V2.py", line 22, in __add__
      return add(v)
  NameError: global name 'add' is not defined
  ```

2.5 Python Style

Software engineering courses often provide very rigid guidelines on the style of programming, specifying the appropriate amount of indentation, or what to capitalize, or whether to use underscores in variable names. Those things can be useful for uniformity and readability of code, especially when a lot of people are working on a project. But they are mostly arbitrary: a style is chosen for consistency and according to some person's aesthetic preferences.

There are other matters of style that seem, to us, to be more fundamental, because they directly affect the readability or efficiency of the code.

- **Avoid recalculation of the same value.**

  You should compute it once and assign it to a variable instead; otherwise, if you have a bug in the calculation (or you want to change the program), you will have to change it multiple times. It is also inefficient.
- **Avoid repetition of a pattern of computation.**

  You should use a function instead, again to avoid having to change or debug the same basic code multiple times.
- **Avoid numeric indexing.**

  You should use destructuring if possible, since it is much easier to read the code and therefore easier to get right and to modify later.
- **Avoid excessive numeric constants.**

  You should name the constants, since it is much easier to read the code and therefore easier to get right and to modify later.

Here are some examples of simple procedures that exhibit various flaws. We'll talk about what makes them problematic.

2.5.1 Normalize a vector

Let's imagine we want to normalize a vector of three values; that is to compute a new vector of three values, such that its length is 1. Here is our first attempt; it is a procedure that takes as input a list of three numbers, and returns a list of three numbers:

```
def normalize3(v):
    return [v[0]/math.sqrt(v[0]**2+v[1]**2+v[2]**2),
            v[1]/math.sqrt(v[0]**2+v[1]**2+v[2]**2),
            v[2]/math.sqrt(v[0]**2+v[1]**2+v[2]**2)]
```

This is correct, but it looks pretty complicated. Let's start by noticing that we're recalculating the denominator three times, and instead save the value in a variable.

```
def normalize3(v):
    magv = math.sqrt(v[0]**2+v[1]**2+v[2]**2)
    return [v[0]/magv,v[1]/magv,v[2]/magv]
```

Now, we can see a repeated pattern, of going through and dividing each element by `magv`. Also, we observe that the computation of the magnitude of a vector is a useful and understandable operation in its own right, and should probably be put in its own procedure. That leads to this procedure:

```
def mag(v):
    return math.sqrt(sum([vi**2 for vi in v]))

def normalize3(v):
    return [vi/mag(v) for vi in v]
```

This is especially nice, because now, in fact, it applies not just to vectors of length three. So, it's both shorter and more general than what we started with. But, one of our original problems has snuck back in: we're recomputing `mag(v)` once for each element of the vector. So, finally, here's a version that we're very happy with:[9]

```
def mag(v):
    return math.sqrt(sum([vi**2 for vi in v]))

def normalize3(v):
    magv = mag(v)
    return [vi/magv for vi in v]
```

2.5.2 Perimeter of a polygon

Now, let's consider the problem of computing the length of the perimeter of a polygon. The input is a list of vertices, encoded as a list of lists of two numbers (such as `[[1, 2], [3.4, 7.6], [-4.4, 3]]`). Here is our first attempt:

```
def perim(vertices):
    result = 0
    for i in range(len(vertices)-1):
        result = result + math.sqrt((vertices[i][0]-vertices[i+1][0])**2 + \
                                    (vertices[i][1]-vertices[i+1][1])**2)
    return result + math.sqrt((vertices[-1][0]-vertices[0][0])**2 + \
                                    (vertices[-1][1]-vertices[0][1])**2)
```

[9] Note that there is still something for someone to be unhappy with here: the use of a list comprehension means that we're creating a new list, which we are just going to sum up; that's somewhat less efficient than adding the values up in a loop. However, as we said at the outset, for almost every program, clarity matters more than efficiency. And once you have something that's clear and correct, you can selectively make the parts that are executed frequently more efficient.

Again, this works, but it ain't pretty. The main problem is that someone reading the code doesn't immediately see what all that subtraction and squaring is about. We can fix this by defining another procedure:

```
def perim(vertices):
    result = 0
    for i in range(len(vertices)-1):
        result = result + pointDist(vertices[i],vertices[i+1])
    return result + pointDist(vertices[-1],vertices[0])

def pointDist(p1,p2):
    return math.sqrt(sum([(p1[i] - p2[i])**2 for i in range(len(p1))]))
```

Now, we've defined a new procedure `pointDist`, which computes the Euclidean distance between two points. And, in fact, we've written it generally enough to work on points of any dimension (not just two). Just for fun, here's another way to compute the distance, which some people would prefer and others would not.

```
def pointDist(p1,p2):
    return math.sqrt(sum([(c1 - c2)**2 for (c1, c2) in zip(p1, p2)]))
```

For this to make sense, you have to understand `zip`. Here's an example of how it works:

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

2.5.3 Bank transfer

What if we have two values, representing bank accounts, and want to transfer an amount of money `amt` between them? Assume that a bank account is represented as a list of values, such as `['Alyssa', 8300343.03, 0.05]`, meaning that `'Alyssa'` has a bank balance of $8,300,343.03, and has to pay a 5-cent fee for every bank transaction. We might write this procedure as follows. It moves the amount from one balance to the other, and subtracts the transaction fee from each account.

```
def transfer(a1, a2, amt):
    a1[1] = a1[1] - amt - a1[2]
    a2[1] = a2[1] + amt - a2[2]
```

To understand what it's doing, you really have to read the code at a detailed level. Furthermore, it's easy to get the variable names and subscripts wrong.

Here's another version that abstracts away the common idea of a deposit (which can be positive or negative) into a procedure, and uses it twice:

```
def transfer(a1, a2, amt):
    deposit(a1, -amt)
    deposit(a2, amt)

def deposit(a, amt):
    a[1] = a[1] + amt - a[2]
```

Now, `transfer` looks pretty clear, but `deposit` could still use some work. In particular, the use of numeric indices to get the components out of the bank account definition is a bit cryptic (and easy to get wrong).[10]

```
def deposit(a, amt):
    (name, balance, fee) = a
    a[1] = balance + amt - fee
```

Here, we've used a destructuring assignment statement to give names to the components of the account. Unfortunately, when we want to change an element of the list representing the account, we still have to index it explicitly. Given that we have to use explicit indices, this approach in which we name them might be better.

```
acctName = 0
acctBalance = 1
acctFee = 2
def deposit(a, amt):
    a[acctBalance] = a[acctBalance] + amt - a[acctFee]
```

Strive, in your programming, to make your code as simple, clear, and direct as possible. Occasionally, the simple and clear approach will be too inefficient, and you'll have to do something more complicated. In such cases, you should still start with something clear and simple, and in the end, you can use it as documentation.

2.5.4 Coding examples

Following are some attempts at defining a procedure `isSubset`, which takes two arguments, `a` and `b`, and returns `True` if `a` is a subset of `b`, assuming that `a` and `b` are represented as lists of elements.

Here is one solution to the problem which is in the Pythonic functional style.

```
    def isSubset(a, b):
        return reduce(operator.and_, [x in b for x in a])
```

This is short and direct. The expression `x in b` tests to see if the item `x` is in the `list` `b`. So, `[x in b for x in a]` is a list of Booleans, one for each element in `a`, indicating whether that element is in `b`. Finally, we reduce that list using the `and` operator[11] , which will have the value `True` if *all* of the elements of the list are `True`, and `False` otherwise.

An alternative is to do it recursively:

```
def isSubset(a, b):
    if a == []:
        return True
    else:
        return a[0] in b and isSubset(a[1:], b)
```

[10] We'll see other approaches to this when we start to look at object-oriented programming. But it's important to apply basic principles of naming and clarity no matter whether you're using assembly language or Java.

[11] To get versions of basic Python operations in the form of procedures, you need to do `import operator`. Now, you can do addition with `operator.add(3, 4)`. Because `and` already has special syntactic significance in Python, they had to name the operator version of it something different, and so it is `operator.and_`.

The base case of the recursion is that a is the empty list; in that case, it's clear that a is a subset of b. Otherwise, we can define our answer in terms of isSubset, but asking a simpler question (in this case, on a list for a which is one element smaller). So, we say that a is a subset of b if the first element of a is a member of b and the set of rest of the elements of a is a subset of b.

We could go even farther in compressing the recursive solution:

```
def isSubset(a, b):
    return a == None or a[0] in b and isSubset(a[1:], b)
```

Here, we are taking advantage of the fact that in Python (and most other languages), the or operator has the "early out" property. That is, if we are evaluating e1 or e2, we start by evaluating e1, and if it is True, then we know the result of the expression has to be True, and therefore we return without evaluating e2. So, or can act as a kind of conditional operator. Some people would find this example too abstruse (shorter isn't *always* better), and some would find it very beautiful.

Here is another good solution, this time in the imperative style:

```
def isSubset(a, b):
    for x in a:
        if not x in b:
            return False
    return True
```

It works by going through the elements in a, in order. If any of those elements is not in b, then we can immediately quit and return False. Otherwise, if we have gotten through all of the elements in a, and each of them has been in b, then a really is a subset of b and we can return True.

Here is another good imperative example:

```
def isSubset(a, b):
    result = True
    for x in a:
        result = result and x in b
    return result
```

This procedure starts by initializing a result to True, which is the identity element for and (it plays the same role as 0 for add). Then it goes all the way through the list a, and ands the old result with x in b. This computation is very similar to the computation done in the functional version, but rather than creating the whole list of Booleans and then reducing, we interleave the individual membership tests with combining them into a result.

All of these examples are short and clear, and we'd be happy to see you write any of them. However, there are lots of versions of this procedure that students have written that we are not happy with. Here are some examples of procedures that are correct, in the sense that they return the right answer, but incorrect, in the sense that they are long or confused or hard to read.

Bad Example 1.

```
def isSubset(a,b):
    list1=[]
    for i in a:
        for j in b:
            if i==j:
                list1.append(i)
                break
    if len(list1)==len(a):
        return True
    else:
        return False
```

This procedure works by going through both lists and making a list of the items that are in common (basically computing the intersection). Then, it checks to see whether the intersection is of the same length as `a`. There are several problems here:

- Using the idea of computing the intersection and then seeing whether it is equal to `a` is a nice idea. But it's hard to see that's what this code is doing. Much better would be to make a procedure to compute the intersection, and then call it explicitly and compare the result to `a`.
- Comparing the length of the intersection and the length of `a` is dangerous. Because we allow repetitions of elements in the lists representing sets, two sets that are equal, in the set sense, may not have the same length.
- The `break` statement exits one level of the loop you are currently executing. It is often hard to understand and best avoided.
- If you ever find yourself writing:

  ```
  if condition:
      return True
  else:
      return False
  ```

  You could always replace that with

  ```
  return condition
  ```

  which is shorter and clearer.

Bad Example 2.

```
def isSubset(a,b):
    itera = 0
    okay = True
    while itera < len(a):
        iterb = 0
        tempokay = False
        while iterb < len(b):
            if a[itera] == b[iterb]:
                tempokay = True
            iterb = iterb + 1
        if tempokay == False:
            okay = False
        itera = itera + 1
    return okay
```

This procedure is basically iterating over both lists, which is fine, but it is using `while` rather than `for` to do so, which makes it hard to follow. We can rewrite it with `for`:

```
def isSubset(a,b):
    okay = True
    for itera in range(len(a)):
        tempokay = False
        for iterb in range(len(b)):
            if a[itera] == b[iterb]:
                tempokay = True
        if tempokay == False:
            okay = False
    return okay
```

continued

Bad Example 3. Previous bad example, being made over.
Now, the remaining lack of clarity is in the handling of the Booleans. We can replace

```
            if a[itera] == b[iterb]:
                tempokay = True
```

with

```
            tempokay = tempokay or (a[itera] == b[iterb])
```

which will set `tempokay` to True if `a[itera] == b[iterb]`, and otherwise leave it the same. And we can replace

```
        if tempokay == False:
            okay = False
```

with

```
        okay = okay and tempokay
```

It still has the effect that if `tempokay` is false, then `okay` will be false, and otherwise it will be as it was before. So, now we have:

```
def isSubset(a,b):
    okay = True
    for itera in range(len(a)):
        tempokay = False
        for iterb in range(len(b)):
            tempokay = tempokay or a[itera] == b[iterb]
        okay = okay and tempokay
    return okay
```

The logical structure is becoming clearer. It might be even better if we were to write:

```
def isSubset(a,b):
    foundAllAsSoFar = True
    for itera in range(len(a)):
        foundThisA = False
        for iterb in range(len(b)):
            foundThisA = foundThisA or a[itera] == b[iterb]
        foundAllAsSoFar = foundAllAsSoFar and foundThisA
    return okay
```

Exercise 2.4. Now, see if you can, first, explain, and then improve this example!

```
def issubset(a,b):
    i = 0
    j = 0
    while i < len(a):
        c = False
        while j < len(b):
            if a[i] == b[j]:
                c = True
                j = j+1
            if c:
                c = False
            else:
                return False
        j = 0
        i = i+1
    return True
```