ASSIGNMENT 0: LEARNING HOW TO USE PYTHON

Contents

1. Basic usage of Python

The purpose of this assignment is to familiarize you with the basics of using Python. There won't be any serious number theory, and the programs you will be asked to write will be rather trivial. There is a somewhat nontrivial written question, though.

First, go to `http://www.python.org` to install Python if you have not yet already done so. We will be using version 2.7, not 3.x. There are seriously non-trivial differences between the two versions, so make sure you get version 2.7!

Okay, so hopefully you were able to get Python 2.7 installed and up and running on your computer. Start IDLE; this should come prepackaged with the installation distributed at `http://www.python.org`.

When you start up IDLE, you should see a few lines of information about the version you have, and then the following prompt in a window titled "Python Shell":

```
>>>
```

with a blinking cursor in front of the triple arrows. This is the interactive Python interpreter; you feed it lines of code and it evaluates them after you hit enter. For example, you can use Python as a basic calculator. If you type in 2+2 at the prompt, and then hit enter, you see something like the following:

```
>>> 2 + 2
4
```

(The color of the result printed by the interpreter might be different than black, to distinguish input you wrote from input the interpreter is printing out in response to what you wrote.)

If you want to use the previous result computed by the interpreter somewhere in your next command, you can use '_' (a single underscore character) to represent that value. For example, if you type in '$\_ * 3$ immediately after the previous line of $2 + 2$, the interpreter will print out 12. Alternately, if you are using IDLE, you can push Ctrl-P repeatedly to see a history of lines you have previously typed and entered into the interpreter, and if you are using the command line, you can push the up key.

Let's now try division. First, if we try dividing, say, 6 by 3, we get the following:

```
>>> 6 / 3
2
```

So far so good. But what if we try dividing 5 by 2?

```
>>> 5/2
2
```

Hmm, that's not quite right! What's happening here is that when we divide two integers, we are not getting the exact answer, but rather the answer rounded down. This might seem a bit annoying, but this is standard across most programming languages and will actually be incredibly convenient for our purposes. (Try some other integer divisions of your own to get a better idea of what's going on.)

But let's say that you actually want the exact value of the quotient of two integers. The first important fact is that computers represent numbers using a variety of different formats. For example, integers are the most natural numbers for computers to represent, because you can encode any integer using a sequence of bits. On the other hand, in general numbers in decimal form with infinitely many digits cannot be represented exactly. Nevertheless, people still need a way to store approximations to these numbers in computers; this is most commonly done using the floating point format. For Python-specific representation information, you can consult this section of the Python documentation.

The way Python knows to do division of floating point numbers instead of integer division is by having at least one of the two numbers we are performing division on be a floating point number itself. You can do this by typing a decimal point at the end of an integer:

```
>>> 5./2
2.5
```

Ah, there we go! We will rarely ever need (maybe never?) to use floating-point arithmetic in this class. Nevertheless, this is an important point for you to be aware of should you ever use any sort of programming language for other purposes in the future. Before moving on to the next topic, can you explain the following output?

```
>>> 5./2 * 1
2.5
>>> 5/2 * 1.
2.0
```

If you want to force the integer-style division with floating point numbers, you can use // instead of /, as in the following:

```
>>> 4. // 3.
1.0
>>> -7.5 // 2.7
-3.0
```

Two operations which will be frequently of use to use are *exponentiation* and *remainder after division.* Exponentiation is exactly what you think it is, and it is denoted using '**':

```
>>> 2**3
8
>>> 4**.5
2.0
>>>3**(-3)
0.037037037037037035
```

Notice that in the last line, even though 3^{-3} can be exactly represented as 1/27, Python returns a floating point approximation to 1/27. To represent rational numbers exactly, you need to use the 'fractions' module. We won't have need for this, but you can consult this section of the Python documentation for more information.

Finally, we have another mathematical operator, denoted `%` in Python, called the modulo operation. If a, b are integers, the output of $a \% b$ is the remainder of a after dividing by b. We will be exclusively concerned with the case where $b > 0$, but Python supports the modulo operator when `b` is negative as well. When $b > 0$, the result of $a \% b$ will be an integer between 0 and $b - 1$, inclusive:

```
>>> 8 % 2
0
>>> 13 % 5
3
>>> -4 % 7
3
```

The value of $a \% b$ is the remainder after a is divided by b. For example, 2 divides 8 evenly, so 8 divided by 2 has a remainder of 0, whereas $13 = 2(5) + 3$, so 13 divided by 5 has a remainder of 3. This operation is of fundamental importance in number theory; we will spend almost the entire class using it.

2. Variables and functions

So far, we have directly used numbers and some basic mathematical operations on those numbers. However, an essential part of programming is the ability to perform operations on variables, not just literal numbers, as well as the ability to define our own functions. Let's take a look at how to do both of these things in Python.

Suppose you want to assign a value to a variable. For example, suppose we want to set the variable x to the value of 4. Then type the following:

```
>>> x = 4
>>>
```

After you hit enter, nothing appears. However, if you now type in x at the interactive interpreter, you get the following output:

```
>>> x
4
```

After the statement '$x = 4$', the interpreter now remembers that x has the value of 4 and will use that value whenever it sees x, at least as long as we don't reassign x to take on another value. So, for instance:

```
>>> x**2 + 1
17
>>> x/3
```

```
1
```

On the other hand, if you enter in a variable name which has nothing assigned to it yet, Python will complain:

```
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

The second line gives you information on where the error occurred (right now, this information isn't very helpful, but if you have a file with several hundred lines of code this will pinpoint the location where something went wrong), and the third line explains the nature of the error. In this case, the Python interpreter is telling us that the variable y has not yet been defined, because we have not assigned anything to it yet.

If you've had any experience with a language like Java or C, this variable behavior is totally different than what you might be used to; in those languages you need to explicitly declare the type of a variable before being able to use it, and the type of that variable is fixed for the remainder of that variable's existence. However, Python allows you to change the value a variable has to whatever type you want, simply by reassigning that value:

```
>>> x = 7.25
>>> x*x
52.5625
>>> x = "Hello"
>>> x
'Hello'
```

The last example illustrates a new data type, known as a string. We're not going to use strings very often in this class, but learning how to work with strings is important in many different parts of programming, especially in programs which manipulate text or deal with input/output.

If you're interested in learning a bit more about the difference in the typing mechanisms between Python and certain other programming languages, you might want to check out the Wikipedia article on typing, especially the sections on static vs dynamic typing. Python is a dynamically typed language, which means that the types of variables can change over the course of execution of a program.

If you forget the type that a variable has (or, for that matter, a literal object has), you can use the type function:

```
>>> x = 7
>>> type(x)
<type 'int'>
>>> x = "math"
>>> type(x)
<type 'str'>
>>> type(7.3)
<type 'float'>
```

We can write statements in programs which mathematically would be nonsense, at least on the surface, but which make perfect sense to the Python interpreter. For example, suppose we have already assigned x the value of 4:

```
>>> x = x+1
>>> x
5
```

What's happening here is a combination of two things: first, in virtually all programming languages, the = sign does not actually test for equality; instead, it is an assignment operator, which assigns whatever is on the left hand side of the operator to whatever's on the right. (Even this statement about variable assignment is hiding a surprising amount of complexity. When subtle details about variable assignment might appear, we will address them at that time.) Second, when executing the = operator, Python will determine whatever the value on the right hand side is first, and then assign the value to what's on the left. So, if we write something like $x = x + 1$ in a program, what we are saying is 'increment the value of x by 1, and let that be the new value of x'.

If you want to test for equality, you use two equals signs: '=='. For example, if you type the previous code sample, then you can get the following:

```
>>> x == 5
True
>>> x == x+1
False
```

The second result matches our sense of mathematical reality, where an equation like $x = x + 1$ is not true, at least when x is a real number. Keep in mind that when we write $x == x+1$ in the Python interpreter, the result is not claiming that $x == x+1$ is false for all possible values of x, it is only claiming that $x == x + 1$ is false for the particular value that x has at that moment.

There's a lot more to say about how variables work in Python (or any programming language for that matter – it's an important, non-trivial, and sometimes counterintuitive topic), but we'll come back to a few of those sorts of questions later and instead now take a look at how to define our own functions.

Suppose that we're given a variable x which contains an integer (or any real number), and we want to calculate $x^3 + x^2 + 4$. We could just type that expression in every time we needed to use it:

```
>>> x = 2
>>> x**3 + x**2 + 4
16
```

However, if we wanted to calculate x^3+x^2+4 many times in the course of a program, we might get tired of repeatedly typing 'x**3 + x**2 + 4' and just want to use a shorthand for it. We can do so by defining a function:

```
>>> def f(x):
        return x**3 + x**2 + 4
```

To input this into the interpreter, you type the first line, hit enter (notice that the interpreter does not print output or give you another >>> prompt), and then type in return and everything that follows, and then hit enter twice. If you're using IDLE, the interpreter will automatically insert a tab after the first line, but if you're using the console, the interpreter might not do so. In any case, you need indentation after the first line for the syntax to be correct. This command defines a function which accepts a variable x, and then returns the value of $x^3 + x^2 + 4$:

```
>>> f(2)
16
>>> f(1.5)
9.625
```

The indentation is Python's way of knowing that everything which is indented is code which belongs to the function you're now defining, as opposed to code lying outside that function. The use of indentation to denote block structure of programs is one of the idiosyncratic features of the Python language, and many people find it somewhat strange, but it really does a good job of enforcing code readability by requiring some sort of uniformity in the visual look of the code. It is generally agreed upon by now that each indentation should consist of four spaces. You can set up IDLE to automatically generate four spaces instead of a tab when you hit the tab key; this is usually the default setting.

Again, if you've learned languages like Java or C, you know that you need to include the types of the various variables passed into a function. In Python, there's no need to do that; the interpreter will just try to use whatever values those variables have.

We'll need to implement more interesting functions, and to do so we'll need to use basic control structures. Let's take a look at a for loop. Suppose we want to write a function which prints all the integers from 1 to n in increasing order. We can do so using the following function:

```
>>> def printints(n):
        for i in range(1, n+1):
            print(i)
        print("Done!")
>>> printints(3)
1
2
3
Done!
```

This function illustrate's Python's version of the for loop. Make sure you remember the colon at the end of the line containing the for statement. The expression 'range(1, n+1)' generates a list of integers from 1 to n. In general, 'range(a, b)' generates a list of integers from a to $b - 1$, where a, b are themselves integers. Python will complain if a or b are not integers, but it is fine if $b \leq a$; Python will just return an empty list.

If you need to loop over a very large range, it might be inefficient to use range, because range first generates the entire underlying list of integers. For example, if you wanted to loop from 1 to 10^9 using range, Python would need to first generate a list of 10^9 integers – this would either take a long time, or Python would just get tell you that you have a memory error, since Python can't allocate a billion integers into memory at once. The way around this is to use xrange instead of range: xrange only generates integers as they are necessary, instead of generating the entire list of integers at once. This might be a good time to point out that this feature is strictly a Python 2 feature; in Python 3 there is no xrange and range exhibits the behavior of xrange in Python 2.

A list is one of the fundamental data structures in Python; we will say more about lists later, but for now you can think of a list as an ordered sequence of some finite number of objects. Python denotes lists using brackets, so for instance:

```
>>> range(1, 4)
[1, 2, 3]
```

Returning to the for loop code, what happens is that the variable i in the for loop statement will iterate through the members of the list that the range expression generates, in order, and execute whatever code follows using the appropriate value of i. Notice that the additional level of indentation after the for loop indicates that all that code should be executed in each pass of the loop. The last line in the function only has one level of indentation, so it is not executed in the for loop; instead, it executes when the for loop finishes.

There are other variations on the range function. For example, range(n) is shorthand for range(0, n), and range(a, b, step) iterates from a to b, with a step size of step. Try the following basic exercises on your own (no need to turn them in):

- Write a function printodd(n) which prints out all the odd integers from 1 to n.
- Write a function endsintwo(n) which prints out all the integers from 1 to n which end in the digit 2.

Another looping control structure is the while loop. A while loop will test whether a condition is true, and if so, execute all the code which belongs to the while loop (which is again indicated by an additional level of indentation). At the end of the execution of that code, the original condition is tested again, and if it is true, the code is executed again. Once the condition which is tested is false, code execution will skip to whatever comes after the while loop.

For instance, suppose we want to print all the squares from 1 to n. We could use a for loop, but we'd have to calculate the square root of n, and maybe we don't want to do that. We can instead use a while loop:

```
>>> def squares(n):
        i = 1
        while (i*i <= n):
            print(i*i)
            i += 1
        print "Done!"
>>> squares(17):
1
4
9
16
Done!
>>> squares(4)
1
4
Done!
```

We need a statement which evaluates to true or false to determine whether we should execute the code in the while loop. In our case, we test whether or not $i^2 \leq n$. If so, we print i^2, and then increment i by 1; i+=1 is shorthand for i = i+1. Again, remember the colon at the end of the line containing the while statement.

Suppose you forgot to increment i. Then the while loop would run forever, which would be bad. You can terminate an infinite loop by sending a keyboard interrupt command to the interpreter; this is usually Ctrl-C (hold down control and push C). Experiment with your computer to determine how to send a keyboard interrupt signal to the interpreter. If you're using IDLE, another option is to select the "Restart shell" command from the "Shell" menu in the menu bar.

At this point, you might have made some errors typing in functions into the interactive interpreter. Python complains, and then you'd have to retype everything back in. That's pretty annoying, so once you get past testing small (usually one-line) snippets of code, you'll probably want to put function definitions and more complicated sequences of commands into a separate file which you can more easily edit and save.

If you're using IDLE, go to the "File" menu and select "New Window". This will open up a blank window. This is not the Python interpreter; instead, it's a text file in which you write code. You can now write function definitions or other code and not worry about having to type everything in perfectly. IDLE will automatically provide syntax highlighting and proper indentation, which is one benefit of writing Python code in IDLE as opposed to a bare-bones text editor.

Once you've written some function definitions in a file, you can save it (use the extension .py, which indicates that your file contains Python code), and then load those functions into the interactive interpreter by selecting "Run → Run Module" from the "Run" menu. On my computer, at least, you can also push F5. This will execute all the code in your file and put you at the interactive interpreter. If your file only has function definitions, then nothing seems to happen, but your functions will now be loaded into the interpreter and you can use them. Try this out with some of the functions we've written above!

If you have some additional code besides function definitions, that code is executed in sequence from top to bottom. You might want to experiment with this a bit as well. Make sure that you're using the appropriate level of indentation.

3. Basic exercises

Let's put the basic knowledge you've learned to use with some simple problems. You will need to submit code for the functions given in each problem in a file [lastname]0.py, but the remarks that come after each problem are meant for educational purposes and do not need to be answered in writing. Nevertheless, you should do them yourself and think about the results.

Problem 1 (5pts)**.** Write a function summation(n), which is given a positive integer n and returns the sum $1+2+\ldots+n$, using a loop which iterates over each individual term in that sum and adds it to a running total.

Try using this function for various values of n. For small n, this function should be nearly instantaneous, but for larger n; say on the order of 10^7 or 10^8, you should notice your computer taking a bit of time (maybe a lot of time) to return the answer. Also, notice that large integers have an 'L' appended to the end; this indicates that the integer is a long integer, which in programming means an integer which needs to be stored in more space than a typical integer, and will usually perform slower when used in arithmetic operations. Your CPU clock speed is probably around 2 gigahertz or so; to a first approximation does this accord with what you're seeing here?

Problem 2 (5pts)**.** In class, when we learned about induction, we proved the formula

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}.$$

(Recall that Gauss had determined this formula using other means at the age of 10, or so they say.)

Write a function, summation2(n), which is given a positive integer n and returns the sum of the first n positive integers using this formula.

Try using this function for various values of n. How does the performance compare to the function you wrote for the first problem? Does this match what you expected? Can summation2 compute this sum for really large n; say, n with $\sim$1000 digits? (Don't actually write down a 1000 digits and pass that number into summation2; just use an exponential like 2^{3500} or something similar). Notice how a bit of math can make a function which can only handle 'small' n instead handle very 'large' n!

3.1. **The Fibonacci sequence.** In the early 13th century, Leonardo Fibonacci defined the Fibonacci sequence, which is a sequence of positive integers F_n whose first two values are $F_1 = 1, F_2 = 1$ (some sources also include $F_0 = 0$), and whose later terms are defined using the recursive formula

$$F_{n+2} = F_{n+1} + F_n, n \geq 1.$$

(If you haven't looked at the first written assignment yet, you'll have a question on this sequence in that assignment as well.) We will examine two different ways of computing the terms of the Fibonacci sequence.

First, consider the following *recursive* function:

```
def fib(n):
    if (n <= 2):
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

The first comment about this function is that it illustrates yet another control statement, called the if-else statement. We begin by writing a line 'if (condition):', which tests whether condition is true or false. If condition is true, it evaluates whatever code belongs to the if statement (denoted by one additional level of indentation). The else which follows is optional; if it is not included then the code belonging to the if statement is skipped if condition is false, but if the else is included, then whatever code belongs to the else statement is executed if condition is false.

This type of function is known as a recursive function because it can call itself (with different parameters) during the course of its execution. Notice that there is a 'base case', where $n = 1, 2$, which returns the value 1, and for all larger n, the function calculates the values of the preceding two terms of the Fibonacci sequence and adds them together.

Implement this function on your computer and run it for, say $n = 1, 10, 20, 30, 50$. Does the function seem to be taking an inordinately long time to run when $n = 50$? Notice that this function seems to do a lot of repeated work. For example, if we want to calculate fib(4), the program will calculate fib(3) and fib(2). The function will quickly determine that fib(2) = 1, but it will calculate fib(3) using the formula fib(2) + fib(1). So, for instance, fib(2) gets called twice by this function.

Problem 3 (15pts)**.** (Turn this in on paper.) When we run the function fib(n) above, how many times is fib called with an argument of 1? More generally, how many times is fib(k) called, for all $k \leq n$? Why does your answer explain the incredibly long run time of, say, fib(50)? Remember that you need to justify your answers.

A function which can't calculate F_{50} more or less instantaneously is horrendously inefficient. After all, based on the original definition of the Fibonacci sequence, we can intuitively see that it should only take about 50 additions to determine F_{50}. The problem with the previous function is that it performed too much repetitive work.

A more efficient way to compute F_n is to compute F_k, for all $k \leq n$, starting at $k = 1, 2$, and then working our way up to $F_3, F_4, F_5, \ldots$, using the recursive formula $F_{k+2} = F_{k+1} + F_k$. For instance, to compute F_5, we calculate $F_3 = 1 + 1 = 2, F_4 = 2 + 1 = 3, F_5 = 3 + 2 = 5$.

Problem 4 (5pts)**.** Implement a function fib_iter(n), which accepts a positive integer $n \geq 1$, and returns F_n, using the method sketched above. Unlike the original fib function described earlier, your program should be able to more or less instantaneously compute F_n for n up to 10000, say.

The above program takes about n additions to calculate F_n. However, you might notice that your computer takes a while to compute F_{100000}, and for modern laptops 100,000 arithmetic operations shouldn't take that long. What's happening here is that to calculate F_n, you need to sum F_{n-1} and F_{n-2}. However, for large n, F_{n-1}, F_{n-2} are gigantic numbers, because F_n grows exponentially (this is a problem from the first written homework assignment). We'll come back to a more precise analysis of this phenomenon in the next problem set, and about halfway through the term we'll examine an even more rapid way of computing Fibonacci numbers which is not nearly as obvious as the two functions we've looked at here.

3.2. Summary of items to turn in.

- Turn in problems 1, 2, 4 in a single file named [lastname]0.py.
- Turn in problem 3 on a physical piece of paper, and remember to justify your answer!