Mark Miyashita

negativetwelve

Student (/resume/#education), Software Engineer (/resume/#work), Teacher (/cs61a), Aspiring Entrepreneur (/blog)

Looking for more CS classes? Check out: **CS61A (/cs61a)** - CS61B (/cs61b) - CS61C (/cs61c) - CS70 (/cs70)

Getting Started with Python

Welcome!

Welcome to CS61A! For the most part, this course is not about learning Python, the language, but instead, we focus on big computer science ideas. However, to implement those ideas in code…we need Python to do so. So we're going to spend the first 1-2 weeks going over simple ideas in Python that will allow us to work on implementing the bigger ideas of the course.

Downloading and Installing Python 3

In this class, we're going to be using Python 3 as opposed to Python 2.x. There are a few major changes, enough to make sure that you are always using version 3 instead of any version of 2. If you have any problems installing Python on your computer, please leave a comment below with specific problems you are having.

We can open up our Python *interpreter* by typing `python3` into our Terminal. If you did it correctly, you'll see the following prompt.

```
>>>
```

Variables

Let's start with a few fundamentals. First, we have variables. Variables are like labels, they allow us to store values and to use them later. In Python, it does not matter what type of data we store in the variable, it can be a string, integer, function, etc. We'll learn all about what those are later. For now, let's define some variables and see how that work!

```
>>> a = 5
>>> b = 6
>>> a
5
>>> b
6
```

Here, we defined to variables, `a` and `b` and their values are now 5 and 6, respectively.

Strings are like words or sentences and we represent them as being surrounded by either single quotes, `'` or double quotes, `"`. We can store strings as variables too.

```
>>> c = "hello world"
>>> c
'hello world'
```

Notice how we defined our string with double quotes, but when we recalled the value, Python represented the same string with single quotes. Either way works and they mean the same thing, just remember that Python uses single quotes when it represents strings.

The reason why we use double quotes sometimes is so that we can represent a string that has a single quote in the middle of it. For example:

```
>>> d = 'don't do this'
File "<stdin>", line 1
  d = 'don't do this'
             ^
SyntaxError: invalid syntax
>>>
```

Here, we encounter a syntax error because Python thinks the string ends at `'don'` and then doesn't know what to do with the rest. The proper way to represent this string is to do the following:

```
>>> d = "don't (not) do this"
>>> d
"don't (not) do this"
```

Now it works!

Expressions

Besides strings, we can work with a variety of other values in Python. Numbers (or integers) are fundamental and they, themselves, are what we call a primitive expression. They are expressions that *evaluate* to themselves.

```
>>> 12
12
```

We can also create expressions that represent mathematical equations. For example:

```
>>> 1 + 2
3
>>> 4 - 5
-1
```

One thing that you will notice, there are two different types of division. There is normal decimal division and integer division (which rounds down). We call these `div` and `floordiv` and they are represented by `/` and `//` respectively.

```
>>> 4 / 5
0.8
>>> 4 // 5
0
>>> 5 // 4
1
```

You can think of `floordiv` as meaning to compute the normal division and then round the result down.

Functions and Calling Functions

In Python, we have some built-in functions that compute common mathematical functions and values. These functions are different from the *infix* notation that we saw above. When we call functions, we have to evaluate both the *operator* and all of the *operands*. In the following example, the function `max` is our *operator* and the two values `3` and `5` are our operands.

```
>>> max(3, 5)
5
```

The way we go about evaluating this function is that we first evaluate the *operator*, `max`. We check the name in our current *environment* and find that it is the built-in function `max`. Next, we evaluate each of the *operands* going from left to right order. In this case, both of the *operands* are primitive expressions so we just have to evaluate the number for its value. However, we can imagine a scenario in which the *operands* are themselves expressions. We call this *nesting expressions*. Take a look at the following example:

```
>>> max(5, min(6, 7))
6
```

Following the rules that we defined above, what do we do first? Well, we first evaluate the *operator* which in this case is `max`. Then, we start evaluating the *operands* from left to right. We evaluate `5` to the value `5`, and then we encounter our next *operand* which happens to be another expression. Since it's a whole new expression, we must start the process over. If you are familiar with the term **recursion**, this is exactly what we are doing. If you're not familiar with the term, recursion is something we will spend a lot of time with this semester (so don't worry!). Recursion is basically repeating the same process until we hit a base case (which you can think of as being the most basic input). We are recursively evaluating the expressions in order, while evaluating any other expressions that might come up along the way. The full order of evaluating this function is this:

1. Evaluate `max`, the *operator* and find that it's the built-in function.
2. Evaluate `5`, the first *operand*.
3. Evaluate the second *operand*, `min(6, 7)`.
4. To evaulate the second *operand*, we start over by evaulating the new operator, `min`.
5. Then, we evaulate the *operands* in order, first `6`, then `7`.

6. Then, we *apply* the *operator* that we evaluated earlier, in this case, applying the *operator* `min` with the *operands* `6` and `7`, evaulates to `6`.
7. Now that we have both of our *operands* for the original *operator*, we can now *apply* `max` with the *operands* `5` and `6`.
8. After applying the function `max`, we get the final value `6`.

Advice for the Semester

One of the best things you can do, for your own learning, is to try things out on your own. There are many things that Python (and other languages) can do that we won't have time to teach you in this class. Although you won't be tested on that material, it might help you in learning and understanding the material that we *will* be testing you on. You also might find random intricacies in Python that might help you on your homeworks. You're free to use them on your homework unless we explicitly forbid you from using them. It's up to you to learn as much as you can (but only if you want to)!

I don't claim to be perfect so if you find an error on this page, please send me an email (/contact) preferably with a link to this page so that I know what I need to fix!