# Introduction to Python

*Due: May 24th - 26th (hand in by your next section)*

## Overview

Welcome to the Python lab! This is due one week after you start the lab in section, to be handed in by the next section (this means each person's due date depends on when they have section).

## 1 What is Python?

Python is a programming language! It bears some resemblance to Java, but in general it is cleaner and easier to read. A few more important differences from Java:

- While variables in Python do have types, you don't need to declare them when you first use variables! You write `x = 1` rather than `int x = 1`. In fact, you can write `x = 1` and then `x = "abcd"` in the same program. After the first assignment, x is an integer; after the second, it's a string. (N.B. In general this is a very bad idea, so avoid it!)
- The Python environment lets you type bits of code and see what happens without an intermediate compiling step. This makes experimentation and testing very simple.
- Look how easy Python is to read! Python uses English keywords and natural-sounding syntax instead of a lot of punctuation and symbols, making it look less scary than code in most programming languages. For example, this is a declaration and an if-condition in Python:

```
x = 1
if x > 0:
    print("x is positive")
    print("What beautiful syntax!")
    print("Do you love Python yet?")
```

- You'll notice that instead of using curly braces to delimit code blocks, Python uses whitespace indentation. That means that correctly nesting your code now has semantic meaning, instead of just making it easier to read. More on syntax later!

Like Java, Python also handles your memory management, meaning it allocates memory and has a garbage collector to free up that memory once it is no longer needed. This makes your life a lot easier!

## 2 Writing your first program in Python

It's tradition when learning a new programming language that your first program is a "Hello World" program, so let's start out by writing a simple one-line program that prints "Hello World!"

### 2.1 Setting up

Clone the Python Lab stencil code from GitHub into a directory of your choosing on your computer. Click **here** to get the stencil code. Check out the **Github Guide** for more details about how to clone a repo!

If you have a Windows computer, make sure you're using Git Bash (which comes with your installation of Git) or some other Linux terminal to run the commands described.

Before we begin programming, we need to configure the editor you will use to write your Python code. While you are free to use any editor of your choice, we recommend you use VSCode.

### 2.2 Setting up your editor for Python

First we will make the `.py` file you will write your program in.

1. Create a new file: File > New File
2. Save this file, File > Save, naming it `helloWorld.py`. The `.py` is very important!! Make sure the file is saved in your pythonIntro directory.

We now need to configure VSCode to work best with Python:

1. On the bottom-left side of the VSCode, click the `Settings` logo. This should open up a new tab in the editor. This is where you can configure different preferences for your VSCode. Take a look at some of the options and feel free to play around with them.
2. In the `Settings` search bar, type `tab size`. Change the `Editor: Tab Size` to be `4` and change `Editor: Insert Spaces` is ticked to ensure tabs are inserted as spaces.
3. Close this tab and you're ready to go!

### 2.3 Let's get to coding!

If you took CS15, you are probably familiar with using these text editors to write Java (`.java`) code. We'll be using them to write Python (`.py`) files in CS16.

It's important you have configured your editor as specified above because Python uses whitespace indentation to delimit your code (more on this later). For the sake of convenience, we insist that you use 4 spaces to indent your code. It will make your code look consistent across machines and prevent inconsistencies between spaces and hard tabs.

Now, let's begin! Type:

```
print("Hello world!")
```

and save your file. Now go back to your terminal (again, use Git Bash if you're on Windows), make sure you are in the `pythonIntro` directory and type `python3 helloWorld.py` to run the program. It will print `Hello world!` to your terminal.

Hold on, do you really have to type `python3 yourProgramName.py` every time you want to run a program? Heck no! At the top of your `helloWorld.py`file, type:

```
#! /usr/bin/python3
```

Or if you're on Windows, type:

```
#!C:/Users/_YOURUSER_/AppData/Local/Programs/Python/Python37/python.exe
```

This tells your machine to use Python to interpret the file when executed. Then save the file, go back to your terminal, and type `chmod +x helloWorld.py` to make the file an executable. (`chmod` is a terminal command used to change file permissions, in this case to make your Python file executable. The `+x` argument adds executability for the owner of the file, you!) Now if you type `./helloWorld.py` into your terminal your program prints `Hello world!` to the terminal. From now on, all of your Python files should start with `#! /usr/bin/python3` (or the corresponding path for Windows).

## 3 Python Syntax

Let's say that instead of wanting to write a program that just prints "Hello world!" and then ends, you want to write a program with a function that takes in a string with your name as the parameter, and prints "Hello <name>!" If we were to write this method in Java, it would look something like this:

```
public void sayHello(String name) {
    System.out.println("Hello " + name + "!");
}
```

Following the CS16 Python coding conventions, the Python function would look like this:

```
def say_hello(name):
    """say_hello: string -> nothing
    Purpose: prints a greeting of the form "Hello <name>!"
    Example: say_hello("Doug") -> "Hello Doug!"
    """
    print("Hello " + name + "!") # this is the function body
```

When you define a function in Python, you simply write `def` (short for define), followed by the name of the function, with all words lowercase and separated by underscores, then the parameters in parentheses, and lastly a colon. You do not need to specify the type of your parameters in Python!

Next, document your function with a block comment! Triple quotes (`"""`) create block comments much like `/*` do in Java. `#` creates an in-line comment, like `//` in Java. The block comment should include a description of the parameters and return type, the purpose of the method, and an example of the method in use. This type of block comment is called a `docstring`. It is crucial to writing readable code that is easy to understand later.

There is a detailed handout on coding conventions on the course website that you can read for more information on writing good Python.

The actual body of this function is simple:

- First off, it is indented four spaces from the function declaration. This is **crucial**; incorrectly indented code will not work. In Python, whitespace indentation is used to nest blocks of code, rather than curly braces. Each subordinating code block must be indented four spaces relative to the code on which it depends. As you get used to programming in Python, this will become second nature.
- The code here `print`s the concatenated string of `"Hello" + str + "!"` to the shell.

To test out this function, type it into your text editor, and put this code at the end:

```
if __name__ == "__main__":
    say_hello("Doug")                # substitute your name
```

It's very important this code comes **after** the function definition, because unlike Java, Python functions must be defined before they can be called.

This bit of code will allow you to run it as a standalone program. The main line here is similar to Java's `public static void main(String args[])`. It contains the code that will run when you execute the program. Save your file (make sure it ends in `.py`) and then run it using one of the two techniques we discussed earlier. The terminal will now greet you as if your name is Doug; substitute your name into the parameters and it will greet you!

```
gemini ~/course/cs0160 $ python3 sayHi.py
Hello Doug!
```

Let's look at something a little more complicated. Say you wrote out pseudocode for a function that prints out the numbers 1 to $n$ for $n \geq 1$. It might look something like this:

```
Algorithm printOneToN(n):
     This algorithm prints out the numbers from 1 to n for n $\geq$ 1.
     If n is less than 1, it prints an error message to alert the user.
     Input: an integer n
     Output: none

if n < 1 then
     print "Invalid input: integer value cannot be less than 1"
     return
for i from 1 to n
     print i
```

In Python, following the CS16 Python coding conventions, it would look like this:

```
def print_one_to_n(n):
    """print_one_to_n: int -> nothing
    Purpose: this function prints out the numbers from 1 to n for n >= 1.
             If n is less than 1, it prints an error message to alert the user.
    """
    if n < 1:
        print("Invalid input: integer value cannot be less than 1")
        return
    for i in range(1, n + 1):
        print(i)
```

Notice that there aren't many differences between the pseudocode and Python. That's one of the reasons Python is so wonderful! Let's go over some of the new Python syntax.

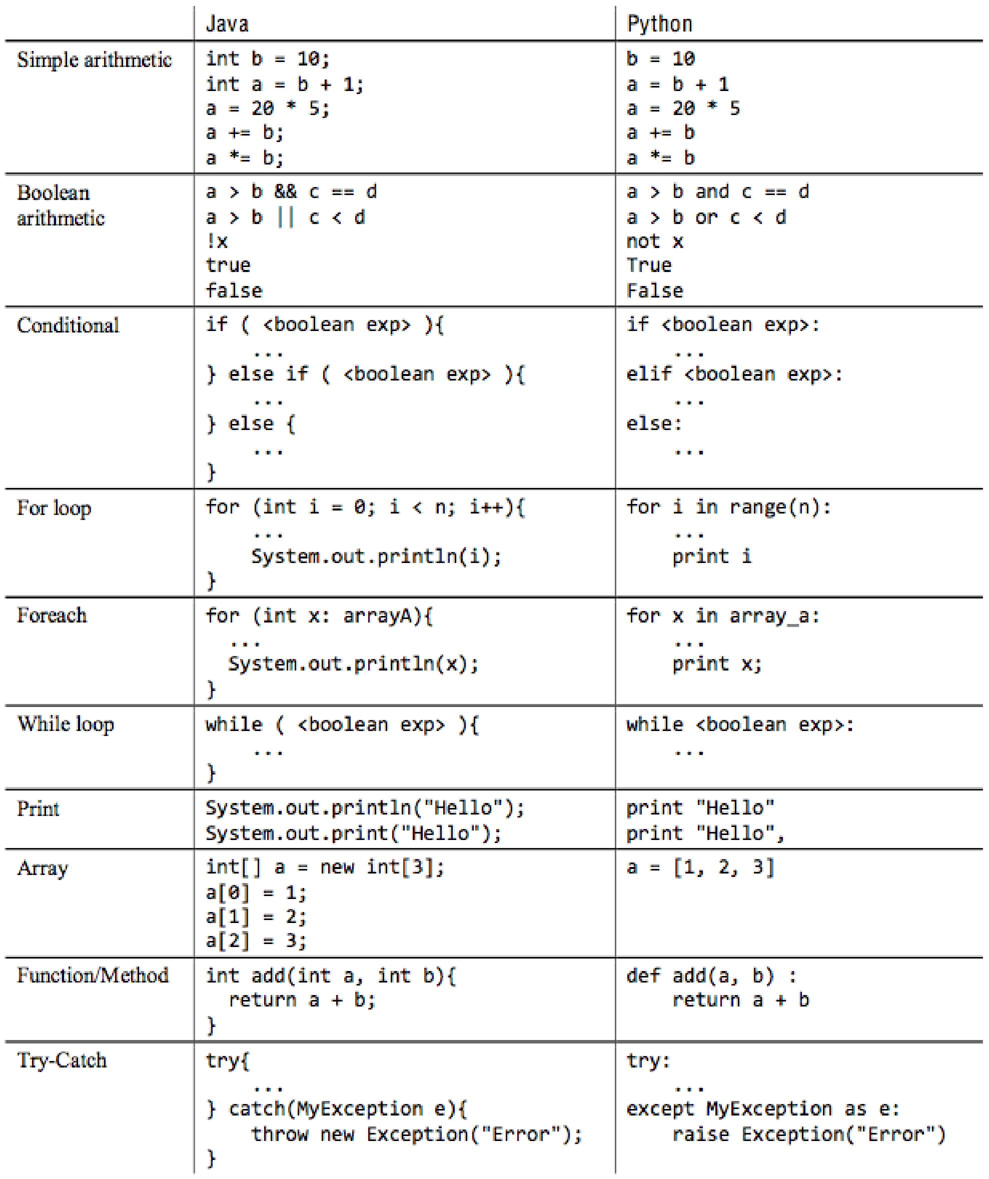
- An if-condition starts with `if` followed by the condition and a colon, no parentheses needed. Even though there are multiple lines of code in the if-block, there are no curly braces because everything is indented four spaces. We could also write the if-statement as the equivalent statement:

  ```
  if not n > 0:
  ```

- Python favors English keywords in the place of punctuation, so you will see `not` used in Python in place of `!` in Java, `and` instead of `&&`, and `or` for `||`.

- The function also has a for-loop to print the numbers from 1 to $n$. So why does it say `range(1, n + 1)`? The built-in function `range()` generates arithmetic progressions based on the optional start parameter, and required stop parameter that you feed it. The range function's start parameter is inclusive, but the stop parameter is non-inclusive. So, if you type `range(1, 10)`, it includes the numbers from 1 to 9. So if we want the function to print out the sequence, including n, we have to write `range(1, n + 1)`.

There is a lot more to Python syntax, but these are some basics to get you started. Here is a super nifty Java to Python conversion table, which will be a boon to you in the coming weeks (note that some of these, like `print`, were still written in Python2):



| | Java | Python |
|---|---|---|
| Simple arithmetic | `int b = 10;`<br>`int a = b + 1;`<br>`a = 20 * 5;`<br>`a += b;`<br>`a *= b;` | `b = 10`<br>`a = b + 1`<br>`a = 20 * 5`<br>`a += b`<br>`a *= b` |
| Boolean arithmetic | `a > b && c == d`<br>`a > b \|\| c < d`<br>`!x`<br>`true`<br>`false` | `a > b and c == d`<br>`a > b or c < d`<br>`not x`<br>`True`<br>`False` |
| Conditional | `if ( <boolean exp> ){`<br>`    ...`<br>`} else if ( <boolean exp> ){`<br>`    ...`<br>`} else {`<br>`    ...`<br>`}` | `if <boolean exp>:`<br>`    ...`<br>`elif <boolean exp>:`<br>`    ...`<br>`else:`<br>`    ...` |
| For loop | `for (int i = 0; i < n; i++){`<br>`    ...`<br>`    System.out.println(i);`<br>`}` | `for i in range(n):`<br>`    ...`<br>`    print i` |
| Foreach | `for (int x: arrayA){`<br>`  ...`<br>`  System.out.println(x);`<br>`}` | `for x in array_a:`<br>`    ...`<br>`    print x;` |
| While loop | `while ( <boolean exp> ){`<br>`    ...`<br>`}` | `while <boolean exp>:`<br>`    ...` |
| Print | `System.out.println("Hello");`<br>`System.out.print("Hello");` | `print "Hello"`<br>`print "Hello",` |
| Array | `int[] a = new int[3];`<br>`a[0] = 1;`<br>`a[1] = 2;`<br>`a[2] = 3;` | `a = [1, 2, 3]` |
| Function/Method | `int add(int a, int b){`<br>`  return a + b;`<br>`}` | `def add(a, b) :`<br>`    return a + b` |
| Try-Catch | `try{`<br>`    ...`<br>`} catch(MyException e){`<br>`    throw new Exception("Error");`<br>`}` | `try:`<br>`    ...`<br>`except MyException as e:`<br>`    raise Exception("Error")` |

## 4 Testing

In future Python assignments, we'll be expecting you to write thorough test cases to exercise your code and ensure it is correct. Testing is a very important part of software engineering. When new features are added to a software package, or when code is refactored to improve design/readability, developers need a way to make sure none of the old functionality breaks. Tests will allow you to make sure your functions do precisely what they say they do. They can also help in debugging — if you know from your tests which functions are working, you won't waste time looking for bugs in the wrong place. Let's take a look at `assert`.

```
def add(a, b):
    return a + b

if __name__ == "__main__":
    assert add(2, 3) == 5, "Arithmetic failure"
```

`assert` will check that both sides of the equality are equal. If the evaluation of the first argument does not equal the second argument, an `AssertionError` exception will be thrown, printing the (optional) message. More generally, if the statement following `assert` evaluates to `False`, then the exception will be thrown and print out the (optional) message.

We will be putting a lot of emphasis on testing throughout this course, in homeworks and projects. Testing should not be considered an additional aspect of a problem or project, but rather a core part of it. Given this, we want you to write your tests **before** you write your code. This practice is known as Test-Driven Development (TDD for short). We'll be walking you through the steps of TDD that will help you write code for `is_prime`.

To test `factorial` (more on this method in the next section), print the result of the `factorial` method with a number you pass in. If your code returns the correct value, you've probably implemented it correctly.

## 4.1 Design Recipe

1. Write some **examples** of the data your function will process. For instance:
   `Input: 4`
   `Output: 24`
   Think of all edge cases your function may encounter, and write your own examples.

2. Outline the **method signature** using header comments to describe the Input/Output and define what the function does. As in the stencil in `sectionOne.py`, we have given you:

```
def factorial(n):
      """factorial: int [n] → int [n!]
      Purpose: Returns the factorial of the argument
      Example: factorial(4) -> 24
      """
```

3. Use the method signature and your examples to write **test cases**. You should write another function called `test_factorial` in the file and add in all of your test cases in that function. For example:
   `assert factorial(3) == 6, "Test failed: Factorial of 3 is 6"`
   Add in your other examples as assertions.

4. **Implement** the method `factorial` now! (more hints in the next section)
5. **Run** your test cases by calling `test_factorial` in the `__main__` call and then executing the Python file.

### 4.2 Raising Exceptions

When you reach the Python problem in `sectionOne.py`, you'll see that there is some code at the beginning of the function body of `factorial` that looks like this:

```
if n < 0:
   raise InvalidInputException("input must be greater than or equal to 0")
```

This is called raising (or throwing) an exception. You may be familiar with some exceptions (e.g. the dreaded `NullPointerException` or `ArrayIndexOutOfBoundsException`) from your work in CS15 last semester. Exceptions are used to detect various errors during program execution. Now, you will learn how to raise your own exceptions! Woohoo!!

Open `sectionOne.py` and examine the code above. When computing the factorial of a number, that number must be an integer greater than or equal to 0. But what if the user of your program doesn't know that, and they try to run `factorial(-3)`? The above `if` statement will be entered, and an `InvalidInputException` will be raised. If this `if` statement was not present in the code, there would be some pretty big issues, as your base case would never be reached, and the method would infinitely recur.

When you write methods, your job is to ensure that the input given is valid before continuing on with your program. Using if statements with similar structure to the one above, you have to check for valid input - if the input is invalid, you have to raise an InvalidInputException that prints out an informative message regarding why the exception was raised. Don't worry about writing your own InvalidInputException class, and, for now, don't worry about determining whether a number is an integer or decimal.

## 5 Your First Python Problem

Now it is time to write a short Python program that prints out the first 100 Fibonacci numbers. Remember, each number in the Fibonacci series is the sum of the two numbers before it, and the first and second numbers of the series are both one.

Once you complete this assignment, create a second method which recursively calculates and returns `n!` where `n` is the input value. Make sure to follow the testing design recipe for this problem.

**Methods**

You need to implement the following methods in the `sectionOne.py` file that was installed earlier.

- `fibonacci`: This method takes no arguments and returns nothing. It prints out the first one hundred Fibonacci numbers, with one number per line.
- `factorial`: This method takes a non-negative integer `n` as an argument and it returns the `n!`. This method should be implemented in a recursive fashion. Remember that 0! is 1!

**Raising Exceptions**

You may notice that there is already some code written in the definition for `factorial`. This was explained in the previous section of the lab.

## 6 Python lists

You can follow along in this section using `python3` command in terminal. This will produce the >>> symbol, indicating you are in a Python environment and can run Python commands. Python lists will be used frequently in CS16. Lists are Python's version of Java arrays. To create a list:

```
 >>> myList = [10, 2, 15, 30]
```

And index into them in much the same way as in Java:

```
 >>> myVal = myList[2] #myVal will now = 15
```

Unlike Java arrays, Python lists also have methods you can call, which you can read about in the documentation. Here's an example that builds a list of the first ten perfect squares:

```
>>> perfectSquares = []
>>> for i in range(1, 11):
...     perfectSquares.append(i*i)
        #append adds the specified parameter to the end of the list
```

If you need to instantiate a list of a certain length (i.e. you want to use it as an array), the following syntax will be very helpful:

```
>>> pythonArray = [0]*5
>>> pythonArray
[0, 0, 0, 0, 0]
```

"But what if I need a multi-dimensional list/array?" you ask. Great question! Python handles this data structure as a list of lists:

```
>>> grid = [[1,2,3], [4,5,6], [7,8,9]]
>>> grid
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> grid[0][1]
2
>>> grid[2]
[7,8,9] #notice that row 2 is itself a list within the 2D grid list
```

This may seem strange at first, but as long as you remember "lists within lists," you'll be fine. You can read about additional Python data structures, (like dictionaries!) at:
`https://docs.python.org/3.7/tutorial/datastructures.html`

### 6.1 List comprehensions

Python supports a concept called "list comprehensions" which can help you create lists very easily. Here are a few examples:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [[x,x**2] for x in vec if x > 3] #x**2 in Python is x*x
[[4, 16], [6, 36]]
>>> [[0]*3 for i in range(3)]
[[0,0,0],[0,0,0],[0,0,0]]
```

## 7 Object-oriented programming in Python

Python supports object-oriented programming as you learned in CS15 with Java. To do so, you declare a class with properties (instance variables) and capabilities (methods) as follows:

```
class Student:
    """The student class models a student with a name, ID number,
    a graduation year, and a concentration.
    """

    def __init__(self, name, idNumber, concentration, graduationYear):
        self._name = name
        self._idNumber = idNumber
        self._graduationYear = graduationYear
        self._concentration = concentration

    def set_concentration(self, concentration):
        self._concentration = concentration

    # Other accessors/mutators...

    def print_student_info(self):
        print("Student named " + self._name + " has ID number " + \
              str(self._idNumber) + ", is graduating in " + \
              str(self._graduationYear) + " and is studying " + \
              self._concentration + ".")
```

To see how this Python class may be compared to a Java implementation, check out the file `Student.java`.To actually create an instance of this class, and call methods on it, we do the following:

```
if __name__ == "__main__" :
    dara = Student("Dara", 1002354, "Physics", 2018)
    dara.set_concentration("Computer Science")
    dara.print_student_info()
```

Create a file called `student.py` and test this out for yourself. What does this example show us?

1. A class constructor is defined using a special method named `__init__`, which can take any number of parameters. To create an instance of the class, we call the constructor and pass in the appropriate parameters. Unlike Java, we don't use a "new" operator.

2. Unlike Java, instance variables belonging to a class do not need to be explicitly defined at the top of the file. Like local variables, we don't need to specify their types, and they spring into existence when they're first assigned a value. If we try to use a variable that has not yet been given a value, we'll get an AttributeError that looks like this: `AttributeError: Student instance has no attribute '_idNumber'`. This is kind of like a Java null pointer in that initializing your variables is important.

3. We access/mutate the value of an instance variable with `self._variableName`, **NOT** by using `_variableName`.

4. The methods belonging to a class should take in `self` as their first parameter. When you call `dara.print_student_info()`, it calls the `print_student_info` method of the `Student` class, and passes in `dara` as the first parameter. So when you define the method, it must take in a reference to the object it's modifying, and when you call the method, you don't supply that first parameter. For this reason, method definitions

will always have one more formal parameter (in the method definition) than actual parameters (when we invoke the method).

5. We couldn't fit the entire `print` statement of `print_student_info` on one line. We could have broken up the method into multiple calls to `print`. Instead we used a `\` to write a multiline Python statement. Remember Python uses whitespace to delimit code, not braces and semicolons like Java, so we need a way to tell Python the next line is still a continuation of the current command.

Now, try changing the method signature to:

```
def __init__(self, name, idNumber, concentration, graduationYear=2020):
```

And the instantiation of `Student` in main to:

```
dara = Student("Dara", 1002354, "Physics")
```

This tells Python to set the default value of `graduationYear` to 2020 if no parameter is supplied. Often you have a function that uses lots of default values, but you rarely want to override the defaults. Default argument values provide an easy way to do this, without having to define lots of functions for the rare exceptions. Also, Python does not support overloaded methods/functions and default arguments are an easy way of "faking" the overloading behavior. (Note: once you use a default parameter, all subsequent parameters must have default values as well.)

## 7.1 Inheritance

The syntax for subclassing is as follows:

```
class SubclassName(SuperclassName):
```

The class `SuperclassName` must have already been defined. Inheritance works approximately the same as it does in Java. One major difference is the way to call methods of your parent class. Instead of using `super.methodName(arguments)`, just call `SuperclassName.methodName(self, arguments)`.

## 7.2 Private variables

Python does have private instance variables available to classes. By declaring an attribute name with 2 underscores as a prefix (e.g. `__myAttribute`), that attribute can now only be accessed from inside an object. However, Python is known for its flexibility and openness, so some programmers follow a different convention: a name prefixed with an underscore (e.g. `_myObject`) should be treated as private and not used outside of the class in which it is contained (even though it technically is still available).

# 8 File I/O

## 8.1 Reading a .txt file

File I/O is very simple in Python, and is one of the tools we'll expect you to become familiar with this semester. You can manipulate files from JPEGS to spreadsheets, but we'll start off working with a simple text file. Open up the `demo.txt` file, to see what you'll be working with. Now, open up Python in your terminal by running `python3`, and try out the following commands:

```
>>> myfile = open("demo.txt")
>>> colors = []
>>> for line in myfile:
...     colors.append(line) #be sure to tab on this line
...
>>> myfile.close()
>>> print(colors)
['red\n', 'orange\n', 'yellow\n', 'green\n', 'blue\n', 'indigo\n', 'violet\n']
>>> for hue in colors:
...     print(hue.strip())
...
red
orange
yellow
green
blue
indigo
violet
```

The `\n` character is called a "newline" and represents a carriage return in a file. We use the `strip()` method to remove unneeded whitespace from our strings. Always make sure to close the files you open to avoid eating up system resources.

## 8.2 Writing to a .txt file

Try the following code in the Python interactive interpreter:

```
>>> file = open('my_file.txt', 'w') #the 'w' arg allows us to write to the file
>>> file.write('This is the first line\n')
>>> for i in range(11):
...    file.write(str(i) + '\n')
...
>>> file.close()
```

Verify that this code created a file named `my_file.txt` in your current directory that contains “This is the first line” and the numbers 0-10. The built-in function `str()` converts objects, such as ints, to their string representations.

## 9 Writing your second program in Python!

Yay! Now it’s time to play with `objects` in Python! Also with pies! Because pies are amazing and delicious. In your `pythonIntro` folder, you should see two files called `pieCount.txt` and `pieCounter.py`. `pieCount.txt` is a text file (woohoo! you know how to read from and write to these) with each line of the form `<name of a TA>, <number of pies>`. Throughout the month, TAs have been recording how many pies they eat and `<number of pies>` is the number of pies that a given TA ate at a given sitting.

The file `pieCounter.py` in your stencil folder is an almost empty `.py` file that you’ll fill up with code! YAY!

### 9.1 Your code

You should create one object that takes in a filename (a Python `string`) as a parameter to its constructor. You can call your object whatever you want, but I’ll refer to it as `PieCounter`. `PieCounter` should have one method (which I’ll call `count_pies`) that takes in the name of a specific TA, counts up how many pies that the TA has eaten this month, and prints that number. Once you’ve written your class, instantiate your class in the `__name__ == "__main__"` block at the bottom of the `.py` file and call your method, passing in a few different TA names. Some examples are `Lisa Yang`, `Doug Woos`, and `Lucy Qu`.

### 9.2 Testing

As usual, we expect you to use `asserts` to test your functions. Refer to the `Student` class for help on the syntax to call class methods. Write method signatures for methods you might need for the class you are about to write and follow the testing design recipe from day 1 to write test cases. Read the hints section that follows for ideas on how to factor code into test-able methods.

Do not yet worry about testing for invalid file names given to the `open()` method. You will learn how to test exceptions on a future homework. We do expect that you will test the functionality of the string parsing you do on the contents of the file.

### 9.3 Hints!

1. Characters of text files are strings in Python! To cast a string to an integer, use the `int()` function. `int(‘5’)` returns `5`.

2. Opening and closing files is expensive! Do you want to open and close the file every time a user calls `count_pies`? Or should you read in the file's contents in the constructor and never have to open the file again?

3. If you don't need a counter when iterating through a list, you shouldn't use `range`. Use the following syntax instead:

```
>>> wordlist = ['wooo', 'I', 'love', 'pie!!!']
>>> for word in wordlist:
...     print word
...
wooo
I
love
pie!!!
```

4. You can get all the lines from a file in one list by calling `myfile.readlines()`.

5. `split` is a super cool method of Python `strings`! Here's an example use: `"Samuel - L. - Jackson - loves - pretzels!!!".split('-')` returns `["Samuel ", " L. ", " Jackson ", " loves ", " pretzels!!!"]`. That is: given some delimiter as a parameter, it breaks a string up into chunks, separated in the original by that delimiter. By default, this delimiter is a space. Check out Python's documentation for more details like how to change the delimiter.

## HANDING IN

You're done with the Python Intro! To hand in your code for this lab ensure your repo on GitHub is up-to-date with your final version. Make sure your files are well-tested and commented!

## 10 Notes about Python

### 10.1 Python version

**Please do NOT use** Python 2.7 because Python 2.7 is not compatible with Python 3, which we'll be using for the duration of the course (and is now the industry standard).

### 10.2 The use of "if main"

For reasons explained here: `https://developers.google.com/edu/python/introduction` (under section *Python Module*) you should always put all code you wish to execute directly (i.e. code *not* part of a specific class or function definition) under a big if statement. The Python syntax for the condition is: `if __name__ == "__main__"`. Here's an example:

```
def function():
    # Function definition goes here.

if __name__ == "__main__":
    # Code to be executed when you run this program
```

All your Python handins are required to follow this convention. We will deduct points from your grade if your code does not include this, so don't forget!

## 10.3 Coding Conventions

We require you to follow the CS16 coding conventions for Python assignments throughout the semester. We expect you to read and follow it. You can find the coding conventions here: **CS16 Python Convention**

## 10.4 Additional reading

This introduction to Python only scratches the surface of what Python is and what it can do. You can find the official tutorial here: `https://docs.python.org/3/tutorial/`. The sections that will be most useful for CS16 are: 1, 3, 4.1-4.6, 4.7.6, 5, 6, 7.1, 8.1-8.5 and 9.3.

Take a look at `this blog post` about common anti-patterns and how to avoid them. Anti-patterns are, essentially, coding solutions many people use that are inefficient, confusing, or even incorrect. Following some of the tips in this post can make your Python programs more efficient and more readable.