Introduction to Python - 2019

Due: February 14th, 11:59pm

Overview

Welcome to the Python lab! The lab is in two parts. Both parts are due on February 14th at 11:59, but we strongly recommend you complete Part 1 before working on Homework 2, and complete Part 2 before working on Homework 3.

PART 1

1 What is Python?

A snake! A snake! A scaaary snake? Yes, but Python is also a programming language that you'll be learning this semester! Python bears some resemblance to Java, but in general it is cleaner and easier to read. There are a few important differences from Java that you should note:

First, you don't need to declare the types of variables. Thus, you write `x = 1` rather than `int x = 1`. Variables do have types, but you don't need to announce the type when you first use a variable. 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!)

Second, the environment in which you work with Python lets you type bits of code and see what happens without an intermediate compiling step. This makes experimentation and testing very simple.

Third, a feature of Python that you will grow to love and cherish is how easy it is to read. Instead of using a lot of punctuation and other symbols, which are what makes code in most programming languages look daunting and scary, Python uses English keywords and natural-sounding syntax. 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 for you (and your TAs) to read. We'll learn more about syntax later, so for now, just marvel at its beauty.

Python is similar to Java in that it also handles your memory management, meaning it allocates memory and has a garbage collectorf to free up that memory once it is no longer needed, making your life a whole lot easier. If you're not already convinced that Python rules...

2 Why should you learn Python?

We'll be using Python throughout the semester to code up some of the algorithms you'll be learning. As a computer scientist you should get used to learning new languages readily, not only because it's an important part of the subject, but also because it's useful (and fun!). Python is especially important to learn because it is used *very* widely as a scripting language – Pixar uses it for all their tools, for instance – so knowing it could one day help you get a job making a movie like Toy Story 3.

3 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!"

3.1 Setting up

Run `cs0160_install pythonIntro` from the command line. This will install a folder `pythonIntro` in your `cs0160` directory. It should contain two files `primePrinter.py` and `sectionOne.py`.

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 Atom. If you are working from a computer in the Sunlab or MSlab, complete Section 3.2, which will tell you how to configure Atom universally for Python code, and skip over Section 3.3. If you have Atom locally on your personal computer, you can follow the same instructions to configure it, although some of the instructions may be a little different depending on how you work locally. If you are working remotely, jump right to section 3.3 to learn how to set up Gedit, which performs better over SSH. Either way, be sure to begin again at Section 3.4, where you will write your first Python program!

3.2 Working from the Sunlab or MSlab

Open the text editor Atom, by typing `atom &` in your terminal. 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 `~/course/cs0160/pythonIntro` directory.

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

1. From the menu bar, select Edit > Preferences. This should open up a new tab in the editor. Scroll down to the `Editor Settings` section. This is where you can configure different preferences for your Atom. Take a look at some of the options and feel free to play around with them.
2. Change the `Tab Length` to be 4 and make sure the `Tab Type` is set to `soft`.
3. Close this tab and you're ready to go!

3.3 Working romotely over SSH

Gedit performs much better over SSH, so you should use this program to work on the lab if you are not on a CS department computer.

Type `gedit &` into a terminal and press Enter to open Gedit.

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

1. Save the current (blank) new file: File > Save as...
2. Name the file `helloWorld.py`. The `.py` is very important!! Make sure the file is saved in your `~/course/cs0160/pythonIntro` directory.

Next, we have to configure Gedit to work well with Python.

1. Go to `Edit->Preferences`
2. Click on the `Editor` tab
3. Ensure that `Tab width` is set to 4
4. Check the box that says `Insert spaces instead of tabs`

Close out of the preferences window. You're all set!

3.4 Let's get to coding!

From 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, make sure you are in the `pythonIntro` directory and type `python helloWorld.py` to run the program. It will print `Hello world!` to your terminal.

Hold on, do you really have to type `python yourProgramName.py` every time you want to run a program? (Or for the especially lazy, scroll through your commands until you find the last time you typed it?) Heck no! Go back to your editor and type:

```
#! /usr/bin/python
```

at the top of your `helloWorld.py` file. 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. (If you haven't used `chmod` before, it's 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/python`.

4 Python Syntax

Let's say that instead of wanting to write a program that just prints "Hello world!" and then ends, you wanted to write a program with a function that takes in a string with your name as the parameter, and prints "Hello <name>!" Following the CS16 Python coding conventions, the 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("Seny") -> "Hello Seny!"
    """
    print "Hello " + name + "!" #this is the function body
```

When you define a function in Python, you simply write `def` (which is 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. Note that you do not need to specify the type of your parameters in Python! Next, document your function with a block comment! Use triple quotes (`"""` to create block comments much like `/*` would in Java. For an in-line comment, use `#`, instead of the `//` from Java. This 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` and 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**. If you do not indent your code correctly, it will not work. Whitespace indentation is used in Python 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. Note that the `print` statement doesn't require that you enclose the string in parentheses since it is what is known as a statement, not a function. Functions in Python do require parentheses.

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

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

It's very important this code comes **after** the function definition, because functions must be defined before they can be called. (Note that this is different than Java).

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 Seny.

Substitute your name into the parameters and watch your terminal try to befriend you. Unfortunately if you try to say hi back, you will see this:

```
gemini ~/course/cs0160 $ python sayHi.py
Hello Seny!
gemini ~/course/cs0160 $ Hello Terminal!
bash: Hello: command not found
```

So much for that :(

Let's look at something a little more complicated. Say you had written out some 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
```

You'll notice that there aren't many differences between the pseudocode and Python. That is 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. So fresh and so clean. We could also write the if-statement as the equivalent statement:

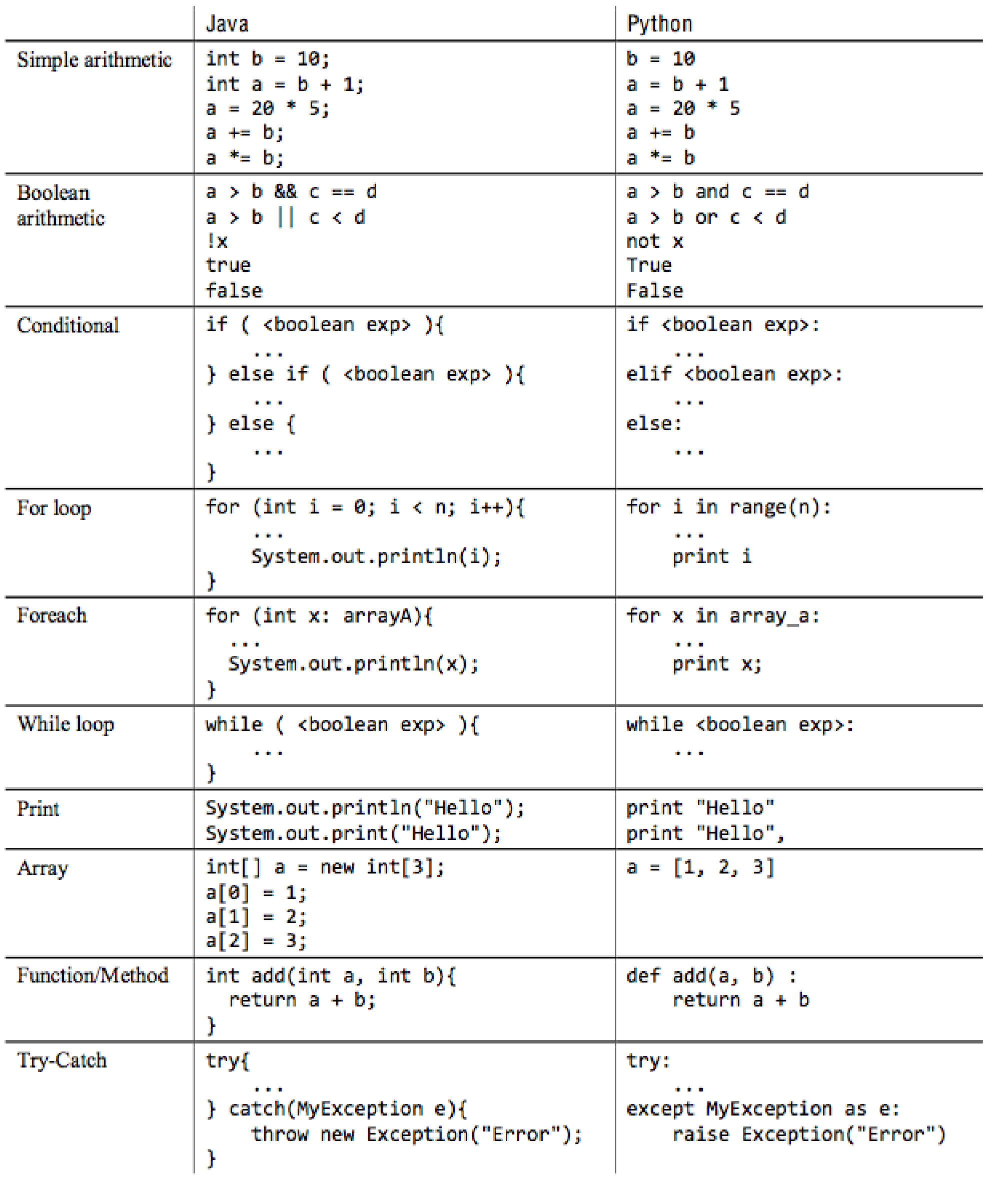
```
    if not n > 0:
```

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. Let's understand this a bit better by just running `python` from your terminal (just type 'python' into your terminal). Python provides its own shell, where you can now run Python commands to try things things out.

Type `range(10)`. Your required stop parameter is 10 and the start parameter defaults to 0, so the integers from 0 to 9 print out, since it "stops" before 10. Now type range(1, 10). Since you specified the start of the progression, it prints 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)`. When you're finished trying things out in the interactive shell, type `exit()` to quit, or use ctrl+D.

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:



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

5 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`.

5.1 Design Recipe

1. Write some **examples** of the data your function will process. For instance:
   `Input: 5`
   `Output: True`
   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 `primePrinter.py`, we have given you:

```
def is_prime(n):
      """is_prime: int → boolean
      Purpose: Test whether the given number is prime or not
      Example: is_prime(3) -> True
      """
```

3. Use the method signature and your examples to write **test cases**. You should write another function called `test_is_prime` in the file and add in all of your test cases in that function. For example:
   `assert is_prime(5) == True, "Test failed: 5 is prime"`
   Add in your other examples as assertions.

4. **Implement** the method `is_prime` now! (more hints in the next section)

5. **Run** your test cases by calling `test_is_prime` in the `__main__` call and then executing the Python file.

To test `get_today`, print the result of the `get_today` method. If your code returns today's date, you've probably implemented it correctly. Also try testing your `is_prime` at least on all the days of any month. Suggestion: write a `for` loop to loop through all of those numbers and print out whether they are prime.

6 Your first Python problem

Now that you've written tests for `is_prime`, you are ready to implement the methods in `primePrinter.py` You will write a short Python program that prints out whether today's date (the day of month) is a prime number or not. It should print out one line which states the day of the month and whether it is prime. For example:

```
It is day 23 of the month, which is prime.
```

or

```
It is day 25 of the month, which is not prime.
```

Methods

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

- `is_prime`: This method must take one non-negative integer argument and return a boolean stating whether that number is a prime or not. (This method should work for all non-negative integers, not just those between 1 and 31.) To test whether a number is prime, simply check whether it has any divisors. Hint: Remember the mod operator (`%`) from Java? It works the same in Python. This method should also raise an exception if invalid input is given (see "Raising Exceptions" below for more details).
- `get_today`: This method takes no arguments and it returns an integer in the range 1-31 representing the day of the month. This should *not* be a hard-coded value (if you invoke the same method tomorrow, it should return tomorrow's day-of-the-month!).
- `is_today_prime`: This method takes no arguments and prints out a sentence that says the day of the month and whether or not it is prime, just like in the example above.

Raising Exceptions

When you get to the next Python problem to be done 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.

As you write your `is_prime` method, your job is to ensure that the input given is valid before continuing on with your program. What sort of input is your `is_prime` method expecting to receive? Using an `if` statement with a similar structure to the one above, check for valid input, and if the input is invalid, raise an `InvalidInputException` that prints out an informative message regarding why the exception was raised. You don't have to worry about writing your own `InvalidInputException` class, and, for now, don't worry about determining whether a number is an integer or decimal.

Hints

- You'll notice at the top of the file there's a line that reads:

  `from datetime import date`.

  This imports the `datetime` module, which has already been written by someone else. Modules in Python are similar to libraries you used in Java (e.g. javaFX). But it doesn't import the entire module, just the `date` component, which you can read about more here: `http://docs.python.org/library/datetime.html#date-objects`. Using the `date` object should help you implement `get_today`.
- If you want to print out an integer value concatenated with a string, you can use the built-in Python function `str()`. `str(myInt)` will convert the integer value to a string.
- If you want to concatenate two strings together, you can simply use the `+` operator.

7 Another 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.

8 Finishing Part 1

You're done with Part 1 of the Python intro! Make sure `primePrinter.py` and `sectionOne.py`are well-tested and commented. You will be handing in Part 1 and Part 2 together, so you don't need to hand anything in until you are done with both parts.

PART 2

Welcome to Part 2 of the python lab! We strongly recommend you complete this part of the lab before working on Homework 3!

8.1 Setting up

Do not reinstall the pythonIntro. The files needed for Part 2 have already been installed. In this part, we will be working with `demo.txt`, `pieCount.txt` and `pieCounter.py`.

9 Python lists

Python lists will be used frequently in CS16. Lists are Python's version of Java arrays. You can create them as follows (try them out in the interactive interpreter using the `python` command):

```
 >>> 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:

```
>>> erfectSquares = []
>>> 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]
>>> [x[1] for x in grid]
[2, 5, 8]
#notice that x represents a sub-list in the 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 addition Python data structures, (like dictionaries!) at:
`http://docs.python.org/tutorial/datastructures.html`

9.1 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]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec] #x**2 in Python is x*x
[[2, 4], [4, 16], [6, 36]]
```

10 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 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. The constructor for a class 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. Note we do not use a "new" operator like we did in Java.

2. The instance variables belonging to a class do not need to be explicitly defined at the top of the file like they do in Java. 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'`. Try this for yourself by eliminating the assignment, `self._idNumber = idNumber` from the constructor, and re-running the program. When we re-run the program, the `print_student_info()` method attempts to print out `_idNumber`, but `_idNumber` has not been created because it has never been given a value. (This is kind of like a Java null pointer in that initializing your variables is important).

3. When accessing/mutating the value of an instance variable, we always do so with `self._variableName`, **NOT** by using `_variableName` like we do in Java.

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=2018):
```

And the instantiation of `Student` in main to:

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

This tells Python to set the default value of `graduationYear` to 2018 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.)

10.1 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)`.

10.2 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).

11 File I/O

11.1 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, 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.

11.2 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.

12 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!

12.1 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 `Xinyang Zhou`, `Seny Kamara`, and `Brian Lee`.

12.2 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.

12.3 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.
6. Consider how you might want to factor your code in order to make it easier to write tests. As an example, consider the task of creating a list of the colours of the rainbow, reversing it and testing the order of the reversed list.

```
def create_rainbow():
    rainbow = ["violet","indigo","blue","green","yellow","orange","red"]
    return rainbow


def reverse_rainbow(colours):
    reversed_rainbow = []

    # the syntax for iterating through a list
    # in backwards order is range(upper_bound,lower_bound,step_size)
    # where, a step_size of -1 indicates subtracting 1
    # from the index counter 'i' each time.

    for i in range(len(colours),-1,-1):
        reversed_rainbow.append(colours[i])
    return reversed_rainbow

    # check out the extra reading on common Python anti-patterns
    # (and how to avoid them) for a more Pythonic way to iterate
    # over this list (and many other types!)
    # hint: a Pythonic way of solving this problem uses the
    # reversed() method

if __name__ == "__main__":
    """If you were only wanting to test that
    your reverse method will work with any
    given list, you can do that because it is
    factored out from the actual creation of
    the rainbow
    """
    test_list = [1,2,3,4,5]
```

```
            reversed_list = reverse_rainbow(test_list)
            assert reversed_list[0] == 5
            assert reversed_list[4] == 1
```

Think about how this example relates to the pie counting problem and how you might want to deal with the list you get from reading in the pieCount file and make sure you are testing it properly using `asserts`

HANDING IN

To hand in your code for this lab (Parts 1 and 2) run `cs016_handin pythonIntro` from your `pythonIntro` directory. Make sure your files from both parts are well-tested and commented.

13 Notes about Python

13.1 Python version

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

13.2 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!

13.3 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: `http://http://cs.brown.edu/courses/cs016/static/files/docs/PythonStandards.pdf`

13.4 Lambdas

Python supports the ability to define one-line miniature functions on the fly. A traditional function definition looks like this:

```
def f(x):
    return x*2

f(3) # evaluates to 6
```

Here's a lambda function that accomplishes the same thing:

```
 g = lambda x: x*2
 g(3) # evaluates to 6
```

Note the abbreviated syntax. There are no parentheses around the parameter list. There's also no return statement. The `return` keyword is implied, since the entire function can only be one expression. The function itself has no name. That's why lambda functions are sometimes called anonymous functions. We can call the function through the variable it has been assigned to (in this case, `g`).

Here's one more example:

```
 (lambda x: x*2)(3) # evaluates to 6
```

In this example, we use the function without even assigning it to a variable. We will learn more about these lambda expressions in our unit on functional programming!

13.5 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: `http://docs.python.org/release/2.5.2/tut/tut.html`. 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.

14 Installing on your computer (Optional)

Python should already be installed on Mac OSX and Linux. To test it out, type `python` into your terminal to start the interactive interpreter. In the event that it isn't or you're using Windows, please see the Resources tab on the website for instructions on how to install `python` on your computer.