# Lab 0

# 6.034 requires 6.0001 (http://catalog.mit.edu/search/?P=6.0001) , or equivalent experience, as a prerequisite.

6.034 uses Python 3 for all of its labs, and over the course of the semester you will be writing significant pieces of Python code. You will be asked to understand the functioning of existing large systems, to add to or change such systems, and to interact with them through APIs.

While coding is not, in itself, a focus of this class, artificial intelligence is a subject full of subtleties. It is imperative that you are able to focus on the problems you are solving, rather than the mechanical code necessary to implement the solutions.

If Python doesn't come back to you by the end of this diagnostic, or if you are still having trouble understanding some of the concepts, we strongly suggest that you avail yourself of the multitude of available Python resources. Such resources include:

---

- Any of the many good Python 3 handbooks out there, such as:
  - Dive Into Python 3 (http://www.diveintopython3.net/) , for experienced programmers
  - O'Reilly's Learning Python (http://proquest.safaribooksonline.com/book/programming/python/9781449355722)
  - Think Python (http://greenteapress.com/wp/think-python-2e/) , for beginning programmers
- The standard Python documentation (http://docs.python.org/) (the Library Reference and the Language Reference are particularly useful, if you know what you're looking for)
- The Course 6/HKN tutoring program (https://hkn.mit.edu/tutoring)

# Installing Python 3

There are a number of versions of Python available. 6.034 uses standard Python ("CPython") (https://stackoverflow.com/questions/17130975/python-vs-cpython#17130986) from http://www.python.org/. If you are running Python on your own computer, you should download and install Python from http://www.python.org/download/ if you do not already have it. All lab code uses Python 3.x, and we recommend using at least version 3.4, if possible.

**Please note that our code will NOT work with Python 2.**

On a Unix machine, you can see which version of Python 3 you have by typing the command `python3 --version`. Note that on Unix machines (including Athena machines), to use Python 3 you should use the command `python3`, as the `python` command defaults to Python 2.

If you are on a Windows machine and downloaded Python 3, you will probably have access to IDLE. To run a python file using IDLE, just open the file in IDLE and click `Run > Run Module` or hit F5 on your keyboard.

If you are using an Athena machine, you might not have access to IDLE for Python 3 (command `idle3`). However, you can still edit Python files in a plain-text editor, and run them like this:

```
python3 filename.py
```

Note, you can use the `idle3` command with an Athena dialup connection (e.g. using SSH with the -X flag). `idle3` does not yet exist on Athena workstations.

## Not all versions are created equal

If you have Python version 3.5.0 or 3.5.1, you should upgrade to 3.5.2+ (or any other version of Python 3, really). There is a bug in the XMLRPC library (https://bugs.python.org/issue26402) for versions 3.5.0 and 3.5.1 which will prevent you from communicating with your server.

If you are running Mac OS X and have Python 3.5 or 3.6, you may get the following error when attempting to submit your code to the server:

```
ssl.SSLError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed (_ssl.c:749)
```

This may be because your version of Python on Mac OS X is not using the system's certificate store (https://stackoverflow.com/questions/41691327/ssl-sslerror-ssl-certificate-verify-failed-certificate-verify-failed-ssl-c) . Assuming you have Python 3.6, executing the following may remedy this issue:

```
sudo /Applications/Python\ 3.6/Install\ Certificates.command
```

## Windows users beware

If your computer is running Windows and you are having trouble running Python from the command line or Git Bash, make sure that Python 3 is added to your `PATH` variable. We strongly suggest reading this page (https://docs.python.org/3.3/using/windows.html) , especially sections 3.2 through 3.4.

# Getting the code

Before working on any lab, you will need to get the code:

- You can use Git to clone the files from the Athena locker at `/mit/6.034/www/labs/lab0/`
- Or, download the code as a ZIP file: http://web.mit.edu/6.034/www/labs/lab0/lab0.zip
- Or, view the individual files: http://web.mit.edu/6.034/www/labs/lab0/

See detailed instructions below.

## Method 1: Git (recommended)

On a command line, navigate to the folder in which you want to put all your 6.034 labs. Then, type this command to clone the lab0 code, replacing `username` with your Athena username:

```
git clone username@athena.dialup.mit.edu:/mit/6.034/www/labs/lab0
```

It will ask for your Athena password. After the files download, you will have a new directory called "lab0" containing the files for Lab 0.

### If you are working on Athena...

...then you can instead use this command to clone directly from the Athena locker:

```
git clone /mit/6.034/www/labs/lab0
```

You can also SSH into `athena.dialup.mit.edu` to directly work on Athena from a different computer.

### Why use Git?

Using Git has a couple advantages:

- Local version control: You can use Git to keep a local record of the changes you've made as you work on your lab, which makes it easy to revert to a previous implementation. You can learn more about using Git here (https://dev.to/jmourtada/a-quick-introduction-to-git) (good for getting started) or from the official Git documentation (https://git-scm.com/doc) . Using Git is not a requirement for 6.034, but you may find it helpful for tracking your files.
- Easy updates: If we need to release an update to the lab code, you can run a simple "`git pull`" command to get the update instead of having to manually download the updated files.

## Method 2: Download the files from a browser

Create a folder for the lab, then download this file and extract it: http://web.mit.edu/6.034/www/labs/lab0/lab0.zip

You can also view or download individual files: http://web.mit.edu/6.034/www/labs/lab0/

# Writing and Running Your Code

The main file of this lab is called `lab0.py`. This is where you will write all of your code. Open that file in IDLE or your preferred text editor.

The first items to fill in are multiple choice questions. The answers should be extremely easy. Many labs will begin with some simple multiple choice questions to make sure you're on the right track.

The remainder of the file contains several incomplete functions for you to fill in with your solutions. For each function, replace the line that says "`raise NotImplementedError`" with your own implementation for that function.

When you are ready to test some or all of your solutions, you can run the tester, explained below.

## Running the tester

Every lab comes with a file called `tester.py`. This file checks your answers to the lab. For problems that ask you to implement a function, the tester will test your function with several different inputs and ensure the outputs are correct. For multiple choice questions, the tester will tell you if your answers were right: yes, that means that you never need to submit wrong answers to multiple choice questions.

The tester has two modes: "offline" or "local" mode (the default), and "online" or "submit" mode. The offline tester runs some basic, self-contained internal tests on your code. The online tester runs more tests, some of which may be randomly generated, and uploads your code to the 6.034 server for grading.

To submit your lab to the test server, you will need to download your unique key.py file (https://ai6034.mit.edu/labs/) and put it in either your lab0 directory or its parent directory. Note that the file must be named 'key.py'.

You can, of course, run the tester as many times as you'd like until your code passes all of the tests.

### Using IDLE

If you are using IDLE, or if you do not have easy access to a command line (as on Windows), IDLE can run the tester.

Open the `tester.py` file and run it using `Run > Run Module` or hitting the F5 button on your keyboard. This will run the local tests for you.

### Using the command line

If you realize just how much emacs and/or the command line rock, then you can open your operating system's Terminal or Command Prompt, and `cd` to the directory containing the files for the diagnostic. Then, execute

```
python3 tester.py
```

to run the tester.

# Python programming

Now it's time to write some Python!

## Warm-up

Using Python's modulus operator, write a function `is_even(x)` that takes in a real number `x` and returns `True` if it's an even integer, and `False` otherwise. Be careful with negative and fractional inputs! For example, `is_even(4) -> True` and `is_even(-1) -> False`.

```
def is_even(x):
```

Write a function `decrement(x)` which takes in a real number `x` and decrements it by 1, unless the result would be less than 0, in which case it returns 0. For example, `decrement(2) -> 1` and `decrement(-1) -> 0`.

```
def decrement(x):
```

Write a function `cube(x)` that takes in a real number `x` and returns its cube. For example, `cube(3) -> 27`. You should use the built-in exponentiation operator `**`, or use the Math module's `math.pow()` function. If you want to use `math.pow()`, you will first have to import the Math module by putting `import math` at the top of your Python file.

```
def cube(x):
```

## Iteration

A real number `x` is prime if it is an integer larger than 1, and if no positive integer besides 1 and `x` evenly divides it. The first few primes are 2, 3, 5, 7, ...

Using a `for` loop, write a function that returns `True` if its input is a prime number, and otherwise returns `False`.

```
def is_prime(x):
```

Write a function that takes a real number `x` and returns an ordered list (possibly empty) of all primes up to and including `x` (hint: you may want to utilize your `is_prime` function). Take care: the list of returned primes must be in order from least to greatest.

```
def primes_up_to(x):
```

## Recursion

The nth Fibonacci number, $F_n$, is defined recursively as

$$F_n = F_{n-1} + F_{n-2}$$

starting from $F_1 = F_2 = 1$, so that the sequence starts: 1, 1, 2, 3, 5, 8, ...

Using recursion, write a function that takes in a positive integer `n` and returns the `n`th Fibonacci number:

```
def fibonacci(n):
```

We recommend writing your functions so that they raise nice clean errors instead of failing silently or messily when the input is invalid. For example, it would be nice if `fibonacci` rejected negative inputs right away; otherwise, you might loop forever. You can signal an error like this:

```
raise Exception("fibonacci: input must not be negative")
```

Or, better yet, you can signal a specific error type:

```
raise ValueError("fibonacci: input must not be negative")
```

Properly dealing with errors like this will help save you some angst in the future when you're trying to track down bugs.

## Expressions as nested lists

Let's look at a more interesting application of recursion.

One way to measure the complexity of a mathematical expression is to compute the "depth" of the expression when it is represented in prefix notation. Prefix notation (as opposed to infix notation) just means that the operator comes first, followed by all of its operands (arguments). Although we won't be using prefix notation much in 6.034, you may encounter it in more advanced AI classes that use languages such as Lisp (Scheme, Clojure, etc) and PDDL. Expressions represented in prefix notation can also be much easier to parse, which is why we will use it in this problem.

For example,

> $x^2 + y^2$ (infix notation)

is equivalent to

> `['+', ['expt', 'x', 2], ['expt', 'y', 2]]` (prefix notation, with `'expt'` representing exponentiation)

Each operator can have one or more arguments. For example, the operator `'+'` can be used to sum many numbers, as in `['+', 10, 20, 30, 40]`.

For this problem, we will represent expressions in prefix form using Python lists. We define the expression depth of an expression recursively based on the number of nested operations:

- The expression depth of a variable or number (such as `'x'` or `3`) is 0.
- The expression depth of an operation depends on the expression depths of all of its children (arguments). In particular, it is one more than the **maximum** of the expression depths of its children. If you think of the expression as a tree, the expression depth is the height of the tree.

Using recursion, write a function that finds the depth of an expression:

```
def expression_depth(expr):
```

For example:

- `expression_depth('x') -> 0`
- `expression_depth(['expt', 'x', 2]) -> 1`
- `expression_depth(['+', ['expt', 'x', 2], ['expt', 'y', 2]]) -> 2`
- `expression_depth(['/', ['expt', 'x', 5], ['expt', ['-', ['expt', 'x', 2], 1], ['+', 5, 2, 3, 'w', 4]]]) -> 4`

Note that you can use the built-in Python function `isinstance` to help determine whether a variable points to a list. `isinstance` takes two arguments: the variable to test, and the type (or tuple of types) to compare it to. For example:

```
>>> x = [1, 2, 3]
>>> y = "hi!"
```

```
>>> isinstance(x, list)
True
>>> isinstance(y, list)
False
```

# Built-in data types

Here, we're going to practice working with strings, sets, lists, and tuples, and using the Python keywords `len` and `sorted`. If you're unfamiliar with any of these, you can learn more about them using the `help` command (e.g. `help(len)`) or by searching the documentation at https://wiki.python.org/moin/.

Write a function `remove_from_string` that takes in 1) a string to copy, and 2) a string of letters to be removed. The function should return a copy of the string with all instances of the given letters removed. For example, `remove_from_string("6.034", "46") -> ".03"`.

```
def remove_from_string(string, letters):
```

Without using the keywords `for` or `while`, write a function `compute_string_properties` that takes in a string of lowercase letters and returns a tuple containing the following three elements:

0. The length of the string (using `len`),
1. A list of all the characters in the string (including duplicates, if any), sorted in REVERSE alphabetical order (using `sorted` with the `reverse` argument (https://wiki.python.org/moin/HowTo/Sorting#Ascending_and_Descending) ),
2. And the number of distinct characters in the string (using a set)

```
def compute_string_properties(string):
```

For example:

- `compute_string_properties("hello") -> (5, ['o', 'l', 'l', 'h', 'e'], 4)`
- `compute_string_properties("zebrafishes") -> (11, ['z', 's', 's', 'r', 'i', 'h', 'f', 'e', 'e', 'b', 'a'], 9)`

Next, we're going to practice using a dictionary.

Write a function `tally_letters` that takes in a string of lowercase letters and returns a dictionary mapping each letter to the number of times it occurs in the string:

```
def tally_letters(string):
```

For example:

- `tally_letters("hello") -> {'h': 1, 'e': 1, 'l': 2, 'o': 1}`

If you want a challenge, you can try to write a one-line implementation of `tally_letters` using a dictionary comprehension (https://docs.python.org/3.4/tutorial/datastructures.html#dictionaries) .

# Functions that return functions

In Python, a function can return any type of object: an int, a list, a user-defined object, ... or even another function!

By defining your function to be returned inside the main function `create_multiplier_function`, implement `create_multiplier_function` to take in a multiplier `m` (a number) and return a function that, when executed, multiples its input by `m`.

```
def create_multiplier_function(m):
```

For example, after you implement this, you should be able to do the following:

```
>>> my_multiplier_fn = create_multiplier_function(5)  # define a function my_multiplier_fn that multiplies its input by 5
>>> my_multiplier_fn(3)
15
>>> my_multiplier_fn(-10)
-50
```

Next, define a function `create_length_comparer_function` which takes in a boolean `check_equals` (either `True` or `False`), and returns a function whose behavior depends on the value of `check_equals`:

- If `check_equals` is `True`, the function returned should take in two lists and return `True` only if they are of equal length.
- If `check_equals` is `False`, the function returned should take in two lists and return `True` only if they are of different lengths.

For example, `create_length_comparer_function(True)([], [2]) -> False` whereas `create_length_comparer_function(False)([], [2]) -> True`. Note how you can chain these function calls because (for example) `create_length_comparer_function(True)` itself is a function that takes in two arguments!

```
def create_length_comparer_function(check_true):
```

# Objects and APIs

Many 6.034 labs include an API for a staff-defined object class. The API is generally a section on the wiki that briefly describes the attributes and functions in the object class. We provide the API as an abstraction so that you don't need to learn the implementation details.

Here's an example:

**Point API**

A `Point` object represents a point in the 2D Cartesian plane. It has an X coordinate and a Y coordinate, each of which you can access and modify:

- **`point.`**`getX()`: returns current X value
- **`point.`**`getY()`: returns current Y value
- **`point.`**`setX(x)`: sets the X value to `x`, then returns the point
- **`point.`**`setY(y)`: sets the Y value to `y`, then returns the point

(where **`point`** is a `Point` instance)

You can also use the following (these are defined for most 6.034 object classes):

- **`point.`**`copy()`, to return a deep copy of the point (that is, a copy that has all the same attributes as the original point)
- `==` , to check whether two points have the same coordinates
- `print`, to print out a human-readable version of a `Point`

## Warm-up

Write a function `sum_of_coordinates` that takes in a `Point` and returns the sum of the X and Y coordinates. For example, `sum_of_coordinates(Point(2, 3)) -> 5`.

```
def sum_of_coordinates(point):
```

## Copying and modifying objects

In some 6.034 labs, you'll need to make copies of an object (such as a `Point`) and modify each copy separately. Here, you'll implement a function `get_neighbors` that takes in a 2D point (represented as a `Point` object). **In the function, use the `.copy()` method on the input point** to create and return a list of the input point's four neighbors: the four points that neighbor the input point in the four coordinate directions. **Do not construct the `Point` objects manually.** By using the `.copy()` method, you can avoid modifying the original point or having to manually construct new points.

```
def get_neighbors(point):
```

For example:

- `get_neighbors(Point(7,20)) -> [Point(6, 20), Point(8, 20), Point(7, 19), Point(7, 21)]`

Note: In general, the order of elements in a Python list matters, but for this problem you may return the four neighbors in any order.

Debugging hint: If your answer looks the same as the expected answer but the tests aren't passing, make sure that

- you are using `.copy()` instead of constructing `Point`s manually
- you aren't modifying the original point

The tester checks for these!

### Using the "`key`" argument

You've used `sorted`, and you're probably already familiar with using `min` and `max` to find the smallest or largest element of a list. The `key` argument is a powerful tool for sorting objects based on a specific attribute (such as a point's Y coordinate) and finding the "smallest" or "largest" object based on a specific attribute.

The `key` argument takes a sorting function. For example, you could use `sorted` with `key` to sort a list of tuples in ascending order by their second elements:

```
>>> my_list = [(3, 100, 10000), (1, 999), (2, 50)]
>>> my_sorting_function = lambda tup: tup[1]  # [1] is the second element
>>> sorted(my_list, key=my_sorting_function)
[(2, 50), (3, 100, 10000), (1, 999)]
>>> my_list
[(3, 100, 10000), (1, 999), (2, 50)]  # unchanged
```

First, use `sorted` with the `key` argument to write a function that takes in a list of 2D points (represented as `Point` objects), then creates and returns a list of the points sorted in decreasing order based on their Y coordinates, without modifying the original list. You may assume that no two points have the same Y coordinate. (Try doing this without using the `reversed` argument.)

```
def sort_points_by_Y(list_of_points):
```

Second, use `max` with the `key` argument to write a function that takes in a list of 2D points (represented as `Point` objects) and returns the point that is furthest to the right (that is, the point with the largest X coordinate), again without modifying the original list. You may assume that no two points have the same X coordinate. This can be done without sorting the list.

```
def furthest_right_point(list_of_points):
```

# For an extra challenge...

If you want an extra challenge, try implementing each of the above functions using just one line of Python code!

# Survey

We are always working to improve the class, so most labs will have at least one survey question at the end to help us with this.

Please answer these questions at the bottom of your `lab0.py` file:

- `PROGRAMMING_EXPERIENCE`: How much programming experience do you have, in any language?

  A. No experience (never programmed before this lab)
  B. Beginner (just started learning to program, e.g. took one programming class)
  C. Intermediate (have written programs for a couple classes/projects)
  D. Proficient (have been programming for multiple years, or wrote programs for many classes/projects)
  E. Expert (could teach a class on programming, either in a specific language or in general)

- `PYTHON_EXPERIENCE`: How much experience do you have with Python?

    A. No experience (never used Python before this lab)
    B. Beginner (just started learning, e.g. took 6.0001)
    C. Intermediate (have used Python in a couple classes/projects)
    D. Proficient (have used Python for multiple years or in many classes/projects)
    E. Expert (could teach a class on Python)

- `NAME`: What is your name? (string)
- `COLLABORATORS`: Other than 6.034 staff, with whom did you work on this lab? (string, or empty string if you worked alone)
- `HOW_MANY_HOURS_THIS_LAB_TOOK`: Approximately how many hours did you spend on this lab? (number or string)
- (optional) `SUGGESTIONS`: What specific changes would you recommend, if any, to improve this lab for future years? (string)

# When you're done

Remember to run both the local and online testers! For example, to run the online tester you might run something like this: 'python3 tester.py submit.' The online tester will automatically upload your code to the 6.034 server for grading and collection.

Because the coding section of Lab 0 is purely for your benefit, the online tester will only run a few simple tests against your survey responses. In future labs, the online tests will also test the functions you write.

---

-