CS311 - Assignment 1
Introduction to Python

Due: Tuesday February 19, at the beginning of class

In this course we will use Python for our programming assignments. The purpose of this assignment is to introduce you to Python and help you get comfortable programming in Python. Most importantly, though, the assignment will get you started using the Python Documentation (found at `http://www.python.org/doc/`). We will give you some useful information here, but I want you to get comfortable with looking at the documentation to figure things out.

For this assignment you should work on it on your own.

Getting Started

Python is installed on all of the lab machines in MBH 632. If you're planning on working in the lab all the time, you can skip to the next section.

If you plan on working on your own computer, you'll need to make sure Python is installed. Most macs and Linux computers should have it installed already, but make sure you have the right version (type `python --version` to get the version). You should have some 2.7 version. If you don't have it installed (or you have the wrong version) go to `www.python.org/download/` to obtain it.

In addition to Python, you'll need a good editor. Some options include (in no particular order):

- Sublime - `http://www.sublimetext.com/`
- Emacs
  - Mac - `http://aquamacs.org/`
  - Windows - `http://www.us.xemacs.org/`
- gEdit - `http://projects.gnome.org/gedit/`
- IDEs
  - IDLE (comes with Python)
  - WindIDE - `http://wingware.com/downloads/wingide-101` (101 version is free, though has less features)

Whichever text editor you decide to use, make sure you setup the preferences so that tabs are NOT put into the document (often called something like auto-tab expand). The editor should only use spaces (when you tab it will add multiple spaces). You can test it out by creating a new python document and inserting a tab. If it puts in spaces you're set. This will avoid a lot of headaches down the road.

If you have any trouble with any installation issues, please come talk to me sooner than later.

Python basics

Python is an interpreted language, which means you type commands directly into the Python command-line prompt and it does not require compiling. You can start Python by opening a command/terminal window and typing "`python`" at the prompt. If you installed it yourself, you may need to add Python to your path.

To get started, try the following commands, pressing enter after each one. Notice that you do NOT need semi-colons at the end of each line, nor do you need to declare variables before you use them (Python has implicit typing).

```
>>> 10**4

>>> a = 'ai is cool'

>>> len(a)

>>> a.split()

>>> myL = [1, 2, 3, 4]

>>> myL[1:3]

>>> myL[1:]
```

Make sure you understand what each of the above statements is doing. Look at the documentation online for more information. Play around some more with strings and lists and make sure you're comfortable with them.

In addition to interacting with Python via the interpreter, you can also save your programs to files and load them into the Python interpreter.

Open your editor of choice and type in the following code that defines a simple list search function and save it as "`search.py`":

```
def list_search(somelist, x):
   """ Searches through a list somelist for the element x"""
   for item in somelist:
      if item == x:
         return True    # We found it, so return True

   return False         # Item not found
```

A few things to note about this function:

- Python uses colons and indentation to indicate blocks. Indentation is *critical* in Python. The colon at the end of a line indicates the start of a block of code, all of which must be indented to the *same* degree. For example the "`return False`" line is outside the for-loop because it is indented at the level of the function body, not the for-loop body.

  Be careful! Don't mix spaces and tabs. It won't work and you'll get an error from the interpreter. If you setup your editor correctly, this shouldn't be a problem.

- Variables are not declared in Python. The `somelist` is understood to be a list and the `x` an element in the list, but it's up to the user to enforce this. Return values are also not specified. If a function includes a return statement, that's what it returns. If it does not, then it returns nothing (specifically `None`).
- The for-loop in Python is a little different then the traditional for-loop and resembles the "for-each" loop in Java. The loop specifies that the variable "`item`" should take on each value in the list `somelist`. The "`range`" function is very useful in getting for-loops in Python to behave like "normal" for-loops. (See the documentation on for-loops).
- We see two types of comments. The line in triple quotes at the top of the function is a special comment called the "docstring". It is displayed when the user asks for documentation about this function by typing "`help(listSearch)`" at the Python prompt. (Try this below). The parts of the line starting with the symbol # are regular comments. You should include docstrings for ALL of our functions written.

To run this function, you first need to load the code into the interpreter. From the normal terminal/command prompt, navigate to the directory containing the file you just created and run Python (by typing "`python`" and pressing `<enter>`). At the interpreter (the `>>>`) type:

```
>>> from search import *
```

Now that the program is loaded you can run any functions defined in the file (in this case just one). Try it out, e.g.:

```
>>> list_search([2, 5, 1, 6, 10], 1)
True
```

To get the documentation for your method, try typing `help(list_search)`. This opens the documentation in your default editor. When you're done, quit the editor and you'll be back at the interpreter.

One thing to be careful of with Python is that if you now make changes to the `search.py` file they will NOT be seen in your current session even if you import search again. There are at least three solutions to this:

- Quit python each time.
- You can "reload" the module by typing:

  ```
  >>> import search; reload(search); from search import *
  ```

  (of course changing "search" to whatever your file/module name is)
- You can use an IDE that does this for you automatically.

When you're all done, type either "`exit()`" or "`quit()`" to exit the Python interpreter.

Now you try: Programming in Python

Once you're comfortable with the basics of Python and the documentation, complete each of the problems below.

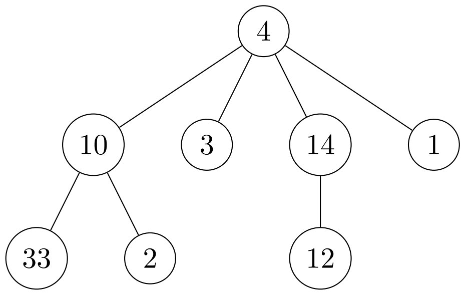
IMPORTANT For all of the assignments:

1. Include a docstrings and comments for every function *and* class you write.
2. Include some comments in your code to help make it clearer.
3. Include your name and other pertinent information at the top of each file.
4. You must name your functions, classes and filenames *exactly* as specified, with the same number of parameters, in the same order as defined.
5. It should go without saying, but your code MUST compile.
6. Do NOT include any additional print statements, etc. (for example, for debugging) when you submit your code. This makes it more difficult for us to grade.

Simple functions

Complete the following problems in a file called "`pa1pr1.py`":

1. Write a function called `twoToTheN(n)` that calculates 2^n for $n >= 0$ in $log(n)$ time.
2. Write two functions, one called `mean(somelist)` and the other called `median(somelist)` that return the average and the median of a list of numbers, respectively. You may implement this functions recursively or iteratively. For a list with an even number of elements, let the median be the average of the two central items. For those of you that took CS150, I realize this is a repeat, but try and do it from scratch again anyway :)
3. Assume a tree is represented as a nested list of lists. E.g., the tree



would be represented as the list

```
[4, [10, [33], [2]], [3], [14, [12]], [1]]
```

Notice that this tree is neither binary nor ordered in any way and that each of the leaves are all lists of length 1.

Write two functions: `bfs(tree, elem)` and `dfs(tree, elem)` that perform a breadth first search and depth first search, respectively, of the tree and return whether or not '`elem`' is in '`tree`'. You will probably want to read about how lists can be used as Stacks and Queues in the Python documentation (or it may be more intuitive to you to just create a Stack and a Queue class).

Objects

Complete this problem in a file called "`pa1pr2.py`":

Develop an object for managing a game of Tic Tac Toe. While we covered the basics of creating objects in Python in class, you will probably want to refer to Python's documentation on creating classes. Create a class called `TTTBoard` that defines the following functions within that class:

- `__init__(self)`: Initialize a 3 x 3 tic tac toe board. You may assume that the board is always 3 x 3.
- `__str__(self)`: Returns a string representation of the board. The left and right edges of the board should be delimited with '`|`'. Empty spaces with an underscore '_' and spaces in between each position. For example, the empty board would be:

  ```
  |_ _ _|
  |_ _ _|
  |_ _ _|
  ```

  and after a few moves:

  ```
  |_ _ _|
  |_ X _|
  |_ _ O|
  ```

- `makeMove(self, player, row, col)`: Places a move for player (specified as a string "X" or "O") in the position `row, col` (where the board squares are numbered starting at 0, so

  ```
   >>> b.makeMove("X", 1, 1)
   >>> b.makeMove("O", 2, 2)
  ```

  would make the above board). You must check to make sure that the move is valid. The function returns `True` if the move was made and `False` if not (because the spot was full, or outside the boundaries of the board).
- `hasWon(self,player)`: Returns `True` if the player has won the game, and `False` if not.
- `gameOver(self)`: Returns `True` if someone has won or if the board is full, `False` otherwise. You may assume only 'X' and 'O' have been placed.
- `clear(self)`: Clears the board to reset the game.

You may represent the board however you like. You may define other functions as well. You may wish also to define a function that allows two players to play the game to make sure your above functions are working correctly.

Commenting

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name and the assignment number should be at the top of each file
- Each class and method should have a short "docstring"
- If anything is complicated, put a short note in there to help the graders out if there are any issues.

Submitting

Please follow the directions below *very carefully*. It will make my life much easier (and you want to make me happy :).

When you're all done, create a directory with your name followed by the assignment number. For example, my folder would be called `davidkauchak1` (*but use your name!*).

Put all your .py files inside this directory and the create a .zip file from this directory. On a Mac, right-click on the directory and select "Compress...". If you're working on Windows, right-click on the file and select "Send to" then select "Compressed (zipped) Folder" (or if you don't have this option, use Winzip (`http://www.winzip.com/`). You will then see a file with a .zip extension created.

Submit this .zip file via the course submission mechanism online.

Grading

Part	points
`twoToTheN`	5
`mean`	5
`median`	5
`BFS`	10
`DFS`	10
`__init__`	3
`__str__`	3
`makeMove`	5
`hasWon`	5
`gameOver`	3
`clear`	3
commenting	3
total	50