CS61A Lab 1: Functions

Week 1, Spring 2012

Try to get as much done as possible, but don't panic if you don't finish.

Exercise 1: Unix/Emacs Tutorial

Do the Unix/Emacs Tutorial which can be found here: Unix/Emacs Tutorial

Exercise 2: Command-line - Python Expressions

At your terminal, type in python as follows:

```
star[200] ~ # python
```

Type each of the following expressions into the Python prompt ">>>", ending the line with the Enter (carriage return) key. Think about what the results will be before you type them! Try to understand how Python interprets what you type. Some of these expressions will offend Python and cause it to spit out confusing error messages. In those cases, just try to get the gist of what caused the error.

```
# this is a comment
3
2 + 3
5 + 6 + 7
-16 - -16
3 * 4 + 1
```

```
# do this column second
from math import sqrt, exp
exp(1)
sqrt(144)
pi
from math import pi
```

```
3 * (4 + 1)
```

```
from operator import mul, add
```

```
3 * 4
```

```
mul(3, 4)
```

```
mul(3, add(4, 1))
```

```
2 ** 3
```

```
pow(2, 3)
```

```
pow(pow(2, 3), abs(-2))
```

```
pi
```

```
pi * 3
```

```
print(pi)
```

```
print(4)
```

```
print(add(9, 1))
```

```
print(print(2))
```

Exercise 3: Defining functions at the command-line

Recall the structure of defining a function:

```
def <name>(<formal parameters>):
    return <return expression>
```

At your Python prompt ">>>" type the following:

```
>>>def cube(n):
...     return n * n * n
...
>>>
```

Be sure to indent the return statement correctly. Then call the function `cube`, with some numerical argument.

Exercise 4: Writing functions in Emacs

Now we will use Emacs, our code editor. Create a new file `sumofsquares.py` and type in the program below.

Don't copy and paste the following code directly. Retype it! Copying and pasting can introduce weird characters or unusual quotes, and they tend to confuse Python. (Poor Python!) Note the indentation.

```
# put your name here
# put your TA name here

def square(x):
    return x * x

def sum_of_squares(a, b):
    return square(a) + square(b)
```

Now load this program into the Python interpreter and test your function by calling `sum_of_squares` with some numerical argument at the prompt.

Exercise 5: Defining your own function

Finally, we will have you write your own function! In the emacs editor, create a `distance.py` file that contains the following program:

```
# put your name here
# put your TA name here

from math import sqrt, pow

def distance(x1, y1, x2, y2):
    # write your code for distance here
```

`distance` should take in two sets of x-y coordinates `(x1, y1)` and `(x2, y2)` [parameters shown above] and should compute the Euclidean distance between the two points. Use the following formula:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Load this program into the Python interpreter, and test the function by calling `distance` at the python prompt ">>>" with the appropriate arguments.

```
>>> distance(1, 1, 1, 2)
1.0
>>> distance(1, 3, 1, 1)
2.0
>>> distance(1, 2, 3, 4)
2.8284271247461903
```

Now let's edit this program to get the distance between two 3-dimensional coordinates. Your distance function should now take in 6 arguments, and compute the following:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Reload the program and again test your function does the right thing!

```
>>> distance(1, 1, 1, 1, 2, 1)
1.0
>>> distance(2, 3, 5, 5, 8, 3)
6.164414002968976
```

Exercise 6 - ucb.py Features

For this course, there are a couple features that end up being very useful in writing code - the staff have provided these in a file called ucb.py (which will be provided with every project). You will need to import the features of the ucb.py file into your Python files though - to do so, you will want to include a statement at the top of each file importing the features you will want to use. For now, we are going to go over the "main" feature, which allows you to easily test your functions:

```
from ucb import main
```

Section I: main

An entry point of a program is the place where execution starts happening. It's usually very convenient to be able to demarcate an entry point in a Python file for testing purposes. Say we have the following file `cube.py`:

```
def cube(x):
    return x * x * x

print("Should be 1:", cube(1))
print("Should be 8:", cube(2))
print("Should be 27:", cube(3))
```

```
star [123] ~ # python cube.py
Should be 1: 1
Should be 8: 8
Should be 27: 27
```

One problem with this file is that the tests aren't cleanly arranged - it'd be much better if we had a test function that performed these tests:

```
def cube(x):
    return x * x * x

def run_tests():
    print("Should be 1:", cube(1))
    print("Should be 8:", cube(2))
    print("Should be 27:", cube(3))
```

However, now, if I run the file, nothing happens:

```
star [123] ~ # python cube.py
star [124] ~ #
```

This is because, to Python, all we've done is define two functions: `cube`, and `run_tests`. We want Python to actually do something when we type in '`python`

`cube.py`', however. So, we specify an entry point with the @main annotation:

```
def cube(x):
    return x * x * x

def run_tests():
    print("Should be 1:", cube(1))
    print("Should be 8:", cube(2))
    print("Should be 27:", cube(3))

@main
def main():
    print("Starting.")
    run_tests()
    print("Ending.")

star [123] ~ # python cube.py
Starting.
Should be 1: 1
Should be 8: 8
Should be 27: 27
Ending.
```