WAT Camp CS61A[1]

I. Introductions
   a. Download Python
   b. Text editor
   c. How to start an interactive python session (python3)
   d. Hello World Program

II. Elements of Programming
   a. Python code consists of **expressions** (typically describe computations) and **statements** (typically describe actions).
   b. Assignment statement:
   c. `shakespeare = urlopen('http://composingprograms.com/shakespeare.txt')`
   d. this line associates the name Shakespeare with the value of the expression that follows =.
   e. urlopen is a **function**.
   f. Objects, you'll be working with objects throughout the class.
      i. An example of one type of object is a **set**. It supports set operations like computing intersections and membership
         1. Question: What are some operations that sets would support? Think about some sets you've encountered before, like the set of Natural numbers and Real numbers.
      ii.
   g. Expressions (in interactive session)
      i. Primitive expressions
      ii. One kind of primitive expression is a number. More precisely, the expression that you type consists of the numerals that represent the number in base 10.
      iii. `>>> 42`
      `42`

      Expressions representing numbers may be combined with mathematical operators to form a compound expression, which the interpreter will evaluate:

      ```
      >>> -1 - -1
      0
      >>> 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
      0.9921875
      ```

      iv. **Call Expressions** apply a function to some arguments.

      ```
      >>> max(7.5, 9.5)
      9.5
      >>> pow(100, 2)
      10000
      ```

III. Importing Library Functions
   a. Python defines a very large number of functions but does not make all of their names available by default. For example, the `math` module provides a variety of familiar mathematical functions:

      ```
      >>> from math import sqrt
      >>> sqrt(256)
      ```

16.0

The `operator` module provides access to functions corresponding to infix operators:

```
>>> from operator import add, sub, mul
>>> add(14, 28)
42
>>> sub(100, mul(7, add(8, 4)))
16
```

The Python 3 Library Docs list the functions defined by each module, such as the math module.

IV. Evaluating Nested Expressions

a. In evaluating nested call expressions, the interpreter is itself following a procedure. To evaluate a call expression, Python will do the following:

i. Evaluate the operator and operand subexpressions, then

ii. Apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions.

iii. Exercise:

```
>>> mul(add(2, mul(4, 6)), add(3, 5))
208
```

V. Defining New Functions

Now we will learn about *function definitions*, a much more powerful abstraction technique by which a name can be bound to compound operation, which can then be referred to as a unit.

We begin by examining how to express the idea of *squaring*. We might say, "To square something, multiply it by itself." This is expressed in Python as

```
>>> def square(x):
        return mul(x, x)
```

which defines a new function that has been given the name `square`.

How to define a function:

Function definitions consist of a `def` statement that indicates a `<name>` and a comma-separated list of named `<formal parameters>`, then a `return` statement, called the function body, that specifies the `<return expression>` of the function, which is an expression to be evaluated whenever the function is applied:

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

VI. Calling User-Defined Functions (go to site with environment diagrams)

a. Applying a user-defined function introduces a second local frame, which is only accessible to that function. To apply a user-defined function to some arguments:

i. Bind the arguments to the names of the function's formal parameters in a new local frame.

ii. Execute the body of the function in the environment that starts with this frame.

b. The environment in which the body is evaluated consists of two frames: first the local frame that contains formal parameter bindings, then the global frame that contains everything else.

VII. Designing functions: qualities of good functions all reinforce the idea that functions are abstractions.
   a. Each function should have exactly one job.
   b. *Don't repeat yourself*.
   c. Functions should be defined generally. Squaring is not in the Python Library precisely because it is a special case of the `pow` function, which raises numbers to arbitrary powers.
   d. **Documentation**: A function definition will often include documentation describing the function, called a docstring, which must be indented along with the function body. Docstrings are conventionally triple quoted. The first line describes the job of the function in one line. The following lines can describe arguments and clarify the behavior of the function:

```
>>> def pressure(v, t, n):
        """Compute the pressure in pascals of an ideal gas.

        Applies the ideal gas law:
        http://en.wikipedia.org/wiki/Ideal_gas_law

        v -- volume of gas, in cubic meters
        t -- absolute temperature in degrees kelvin
        n -- particles of gas        """

        k = 1.38e-23  # Boltzmann's constant
        return n * k * t / v
```

   When you call `help` with the name of a function as an argument, you see its docstring (type `q` to quit Python help).

VIII. Control flow
IX. Higher-order Functions
X. Environments
XI. Project 1 DUE
XII. Newton's method, objects, data abstraction
XIII. Sequences, strings
XIV. Project 2 DUE
XV. Mutable data, lists, dictionaries
XVI. MT1
XVII. OOP, classes, inheritance
XVIII. Using objects, recursion
XIX. Project 3 DUE
XX. Orders of growth, recursive data
XXI. MT2