CSci 1133, Spring 2015
Lab Exercise 6

Procedural Abstraction

In a previous Lab Exercise, you learned about useful *functions* that have been created by other people. A *function* was described as an *abstraction*; a named set of statements that perform some computation and return the results. In other words, a *function* is a named, self-contained computational *procedure* that we simply treat as a "black box". We provide some input to it and obtain a result without knowing (or caring!) what it does internally.

As discussed in lecture, *procedural abstraction* is a fundamental computer science concept that provides a powerful means to decompose complex problems into more manageable pieces. This approach to problem solving involves breaking larger problems into simpler ones that can be implemented and tested as individual Python functions. Once we've written and validated the operation of a particular function, we can largely ignore its detailed (step-by-step) operation and focus on its abstract behavior to solve larger problem(s).

Most of our programming from now on will involve writing our own Python functions and using them in our programs. In this lab we expand our understanding of *procedural abstraction*.

Warm-up

Defining Functions
Perhaps the most powerful construct in any high-level computer language is the ability to create higher-order abstractions from more primitive ones. Among other benefits, this allows us to avoid "re-creating the wheel" each time we need to solve some problem. In Python, it's pretty simple:

```
def function_name ( formal parameter list ):
        statement(s)
```

function_name is the name we will subsequently use to *invoke*, or *call* the function, the *formal parameter list* consists of zero or more variable names separated by commas, and *statement(s)* are one or more Python statements that will be executed when we invoke the function (the function *body*).

It is very important to understand that this construct is *defining* what the process will do *when it is called*. It is a *static description* of a process. When the Python interpreter encounters this definition statement, it does *not* execute the statements in the function body. In order for the statements in the function body to be executed, the function must be *called* from somewhere else in the program. When the function is called, the variables listed in the *formal parameter list* will dynamically take on the values that are supplied as arguments in the function call in order of argument position.

Let's create a simple program that includes a function definition. First, if you haven't done so already, start the IDLE development environment. Now enter the following into a new source file:

```
radius = float(input("enter a radius: "))
area = computeArea(radius)
print("The area of a circle with radius ", radius, "is: ",area)

def computeArea(x):
```

```
    return 3.14159 * x ** 2
```

Verify that you've entered it exactly as shown and then save and run the module.

What happened? You should have caused a Python runtime error:

```
NameError: name 'computeArea' is not defined
```

Python is telling you that at the time it tries to call `computeArea()`, it has no idea what `computeArea` is! Recall the idea of algorithm *sequence*. At runtime, the interpreter is executing one line at a time *in-order* from first to last. When it encounters the function *call*, it has not yet processed the *definition* of the function! This is easy to fix. You simply place all function definitions ahead of any *runtime* references to the function in the source file. Rearrange your simple program so it looks like this (moving the function definition to the top:

```
def computeArea(x):
    return 3.14159 * x ** 2

radius = float(input("enter a radius: "))
area = computeArea(radius)
print("The area of a circle with radius ", radius, "is: ",area)
```

Now save and run the module again. Confirm your calculation using a radius value of 1.

Let's explore how this works… The function definition includes a variable `x` that is clearly used by the statements *inside* the function body to compute the area of a circle. Note that `x` is *local* to the function *definition*. It is not "visible" to any statements other than those within the function body. When the function is *called*, `x` will be assigned the value that the *caller* of the function provides as the *argument* of the function call. This assignment operation will occur *before* the first line of the function body executes.

The `return` statement is the mechanism that returns a *value* as output of the computation. After the value is returned, it is used to complete the caller's expression evaluation.

You are encouraged to explore further on your own. You may want to look at the `radians` and `degrees` functions in the `math` module and understand their use with respect to the trigonometric functions (sin, cos, tan, etc.).

Stretch

In Lab1, you were introduced to the Python Turtle Graphics module in interactive mode. For these stretch exercises, you will use the `turtle` module to write functions and programs. You should review appendix C3 in your textbook and note the following:

1. You must first import the `turtle` module and then construct a turtle object using the `turtle` constructor:

```
import turtle
myturtle = turtle.Turtle()  #note the capital 'T'
```

2. The "turtle" has a "pen" that will draw a line if the state of the pen is *down*. To draw a line, you simply *move* the turtle. e.g.,

```
myturtle.pendown()
myturtle.forward(100)
```

   The default state of the turtle pen is *down*. If you want to move the turtle without drawing a line, you need to change the state of the pen to *up*:

```
myturtle.penup()
myturtle.forward(100)
```

   Don't forget to put the pen back down before you begin drawing again!

3. You can turn the turtle left or right before moving…

```
myturtle.left(90)
myturtle.right(45)
```

4. You can change the drawing speed by using the `.speed()` method. The default speed is set to a slower value so that you can observe the drawing process. In order to draw at the maximum speed, you should set the speed to 0:

```
myturtle.speed(0)
```

There is much more that you can do using Turtle Graphics. Appendix C3 describes a number of useful turtle methods

1) **Abstract Squares**

Write the definition for a function named `drawsquare` that will use Turtle Graphics to draw any sized square starting at the current turtle location and heading. The `drawsquare` function should accept two arguments: a turtle object and the length of each side, and draw a square whose sides are specified by the second input argument. Note that if you do not pass the turtle object as an argument to the function, it won't work. (can you explain why?) Test your function using the Idle command line interface.

2) **Lots of Squares**

Now, write a Python program that will repeatedly call the `drawsquare` function from problem 1) to draw a series of identical squares in different orientations around a single point. Your program should use a loop to draw 10 squares with sides of length 100, rotating the turtle by 36 degrees following each iteration.

3) **Abstractions of Abstractions**

Finally, turn the program you just completed (and tested!) in 2) into a *function* named `squares`. Your function should accept two arguments: a turtle object and the number of squares to draw. The `squares` function should then draw the specified number of squares, rotating the turtle by the proper amount each time to rotate completely through 360 degrees. Write a Python program that will input a value for a number of squares and call the `squares` function to draw the specific number of squares. Show your results to one of your TAs.

Workout

1) **Shortest Distance**

The distance between 2 points (x_1,y_1), (x_2,y_2) on a Cartesian plane is given by Pythagoras:

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

Write a function named `shortestDist` that will take a (nested) *list* of points as its only argument and return the shortest distance between any two points in the list. Each point in the list is represented by a list of two elements:

$$[\ [x_1,y_1], [x_2,y_2], \ldots , [x_n,y_n]\ ]$$

Note that this will require an "all-pairs" comparison. Avoid comparing a point with itself!

Write a Python program that calls the `shortestDist` function and displays the result for the following list:

```
[[45, -99], [24, 83], [-48, -68], [-97, 99], \
 [-8, -77], [-2, 50], [44, 41], [-48, -58],  \
 [-1, 53], [14, 86], [31, 94], [12, -91],    \
 [33, 50], [82, 72], [83, -90], [10, 78],    \
 [7, -22], [90, -88], [-21, 5], [6, 23]]
```

(Note, the shortest distance in this list is 3.1623)

2) **Newton's Root Finding Method**

We've explored the remarkably effective Babylonian "guess and check" method for finding square roots. The problem of computing the square root of any number is a specific case of the more general *root-finding* problem. Recall that the *roots* (or *zeros*) of any function are the parametric value(s) for which the function produces a zero result:

$$x : f(x) = 0$$

By definition, the *square root* of some number K is the value x such that $x^2 = K$. This is equivalent to finding the principal *zero* of

$$f(x) = x^2 - K$$

We could just as easily compute the *cube* root of K in the same fashion by solving:

$$x^3 - K = 0$$

and so on. Generalizing, we can find the n^{th} root of any value K by solving:

$$x^n - K = 0$$

Issac Newton used calculus to generalize the Babylonian square root finding algorithm The Newton-Raphson algorithm that bears his name is an efficient and ingenious method for finding the principal root of any continuously differentiable function. The method is ascribed to Sir Isaac Newton, although Joseph Raphson

was the first to publish it in a more refined form. The method is a generalized version of the Babylonian "guess and check" approach that you explored in previous lab exercises.

Newton's method generalizes the Babylonian approach by employing the *derivative* of the function. Given a *guess*, x_i, the subsequent ("better") guess, x_{i+1} is computed by:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Note that the Babylonian Algorithm is simply the specific case of Newton-Raphson for $f(x) = x^2 - K$

For the generalized *n*th root problem, $f(x)$ is:

$$x^n - K$$

and the derivative, $f'(x)$ is:

$$nx^{n-1}$$

substituting these formulas, the Newton-Raphson update rule for computing the n^{th} root of any value K is:

$$\begin{aligned} x_{i+1} &= x_i - \frac{x_i^n - K}{nx_i^{n-1}} \\ &= x_i - \frac{x_i^n}{nx_i^{n-1}} + \frac{K}{nx_i^{n-1}} \\ &= x_i - \frac{x_i}{n} + \frac{K}{nx_i^{n-1}} \\ &= \frac{1}{n}\left[(n-1)x_i + \frac{K}{x_i^{n-1}}\right] \end{aligned}$$

Write a program that includes a *user-defined function* named `rootN` that will calculate and return the *n*th root of any positive value using the Newton-Raphson method described above:

```
def rootN(n,root):
```

The function takes two arguments: a floating-point value > 0 and an integer root > 1. This function should work exactly as the Babylonian square root, but using the more generalized update rule described above. **Do not use any math module functions in your solution**.

Write a Python program that will input positive values for the value and root, and then call the `rootN` function to determine and display the n^{th} root of the input value. Include a continuation loop that will continue this process as long as the user wishes.

Example:

```
enter value and root: 9 2
the root is: 3.00002
continue? (y/n): y
```

```
enter value and root: 9 3
the root is: 2.08008
continue? (y/n): y

enter value and root: 9 4
the root is: 1.73207
continue? (y/n): n
```

Challenge

Try this challenge problem if you have extra time or would like additional practice outside of lab.

1) **Sudoku**

Sudoku is a popular number puzzle that appears in many newspapers on a daily basis. The objective of the game (from Wikipedia) is:

> "to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "boxes", "blocks", "regions", or "sub-squares") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which typically has a unique solution.
>
> Completed puzzles are always a type of Latin square with an additional constraint on the contents of individual regions. For example, the same single integer may not appear twice in the same 9×9 playing board row or column or in any of the nine 3×3 subregions of the 9×9 playing board"

Here's an example puzzle and its solution:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Write a function named `checkSudoku` that will check a proposed Sudoku puzzle solution and return True if its correct, False otherwise. Represent the puzzle grid as a two-dimensional list with 9 rows and 9 columns, and pass it as a single argument to the `checkSudoku` function.

There are two basic approaches to checking the solution:

1). Verify that every row, column and sub-square contains *all* the digits 1-9

or…

2). Check every cell in the grid and verify that is unique among all the elements in its row, column and sub-square.

[Hint: you can obtain the index [a][b] of the upper-left corner a 3x3 sub-square for any grid index [i][j] using the following: a = i//3*3, b = j//3*3

Write a Python program that will call `checkSudoku` and display either "Solution" or "Not a solution" for each of the following (at a minimum):

```
[ [5,3,4,6,7,8,9,1,2], \
  [6,7,2,1,9,5,3,4,8], \
  [1,9,8,3,4,2,5,6,7], \
  [8,5,9,7,6,1,4,2,3], \
  [4,2,6,8,5,3,7,9,1], \
  [7,1,3,9,2,4,8,5,6], \
  [9,6,1,5,3,7,2,8,4], \
  [2,8,7,4,1,9,6,3,5], \
  [3,4,5,2,8,6,1,7,9]  ]

[ [5,3,4,6,7,8,9,1,2], \
  [6,7,2,1,9,5,3,4,8], \
  [1,9,8,3,4,2,4,6,7], \
  [8,5,9,7,6,1,5,2,3], \
  [4,2,6,8,5,3,7,9,1], \
  [7,1,3,9,2,4,8,5,6], \
  [9,6,1,5,3,7,2,8,4], \
  [2,8,7,4,1,9,6,3,5], \
  [3,4,5,2,8,6,1,7,9]  ]
```