Download (right-click, save target as ...) this page as a jupyterlab notebook from: Lab6

Laboratory 6: FUN with functions

Medrano, Giovanni

R11521018

ENGR 1330 Laboratory 6 - In-Lab

In [1]:
```
# Preamble script block to identify host, user, and kernel
import sys
! hostname
! whoami
print(sys.executable)
print(sys.version)
print(sys.version_info)
```

```
DESKTOP-6HAS1BN
desktop-6has1bn\medra
C:\Users\medra\anaconda3\python.exe
3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
sys.version_info(major=3, minor=8, micro=5, releaselevel='final', serial=0)
```

What is a function in Python?

Functions are simply pre-written code fragments that perform a certain task.

How do I implement (call) the function?

We call a function simply by typing the name of the function or by using the dot notation. Whether we can use the dot notation or not depends on how the function is written, whether it is part of a class, and how it is imported into a program.

Some functions expect us to pass data to them to perform their tasks. These data are known as parameters(older terminology is arguments, or argument list) and we pass them to the function by enclosing their values in parenthesis ( ) separated by commas.

For instance, the `print()` function for displaying text on the screen is \called" by typing `print('Hello World')` where print is the name of the function and the literal (a string) 'Hello World' is the argument.

How does a function fit within the program flow ?

A function, whether built-in, or added **must be defined before it is called**, otherwise the script will fail.
Certain built-in functions "self define" upon start (such as `print()` and `type()` and we need not worry about those funtions).

Example(s)

The example below illustrate some features of functions. The forst code block should generate an error because the **math** package is not yet imported (if you have it from another notebook, just restart the kernel to clear everything). Then fix the indicated line (remove the leading "#" in the import math ... line) and rerun, should get a functioning script.

The second block of code uses an alias (remaming) technique to import a specific function contained within an external module.

In [4]:
```
# reset the notebook using a magic function in JupyterLab
%reset -f
import math

# An example, run once as is then activate indicated line, run again - what happens?
x= 4.0
sqrt_by_arithmetic = x**0.5
print('Using arithmetic square root of ', x, ' is ',sqrt_by_arithmetic )
#import math # import the math package ## activate and rerun
sqrt_by_math = math.sqrt(x)  # note the dot notation
print('Using math package square root of ',  x,' is ',sqrt_by_arithmetic)
```

```
Using arithmetic square root of  4.0  is  2.0
Using math package square root of  4.0  is  2.0
```

In [5]:
```
# Here is an alternative way: We just load the function that we want:
# reset the notebook using a magic function in JupyterLab
%reset -f
# An example, run once as is then activate indicated line, run again - what happens?
x= 4.
sqrt_by_arithmetic = x**0.5
print('Using arithmetic square root of ', x, ' is ',sqrt_by_arithmetic )
from math import sqrt # import sqrt from the math package ## activate and rerun
sqrt_by_math = sqrt(x)  # note the notation
print('Using math package square root of ',  x,' is ',sqrt_by_arithmetic)
```

```
Using arithmetic square root of  4.0  is  2.0
Using math package square root of  4.0  is  2.0
```

Added-In using External Packages/Modules and Libaries (e.g. math)

Python is also distributed with a large number of external functions. These functions are saved in files known as modules or libraries or packages. To use the built-in codes in Python modules, we

have to import the modules themselves into our programs first. We do that by using the `import` keyword. There are three ways to import:

1. Import the entire module by writing import moduleName; For instance, to import the random module, we write import random. To use the randrange() function in the random module, we write random.randrange(1, 10);28
2. Import and rename the module by writing import random as r (where r is any name of your choice). Now to use the randrange() function, you simply write r.randrange(1, 10); and
3. Import specific functions from the module by writing from moduleName import name1[,name2[, ... nameN]]. For instance, to import the randrange() function from the random module, we write from random import randrange. To import multiple functions, we separate them with a comma. To import the randrange() and randint() functions, we write from random import randrange, randint. To use the function now, we do not have to use the dot notation anymore. Just write randrange(1, 10).

In [6]:
```
# Example 1 of import
%reset -f
import random
low = 1 ; high = 10
random.randrange(low,high) #generate random number in range low to high
```

Out[6]: 3

In [7]:
```
# Example 2 of import
%reset -f
import random as r
low = 1 ; high = 10
r.randrange(low,high)
```

Out[7]: 8

In [8]:
```
# Example 3 of import
%reset -f
from random import randrange
low = 1 ; high = 10
randrange(low,high)
```

Out[8]: 3

The modules that come with Python are extensive and listed at https://docs.python.org/3/py-modindex.html. There are also other modules that can be downloaded, installed into your python system, then imported into your codes. (just like user defined modules below). In lab we are building fairly primitive codes to learn how to code and how to create algorithms.

For many practical cases you will want to load a well-tested package to accomplish the tasks, and only enough of your own scripting to implement the package.

How do I make my own functions?

We can define our own functions in Python and reuse them throughout the program. The syntax for defining a function is:

```
def functionName( argument ):
    code detailing what the function should do
    note the colon above and indentation
    ...
    ...
    return [expression]
```

The keyword `def` tells the program that the indented code from the next line onwards is part of the function. The keyword `return` tells the program to return an answer from the function. There can be multiple return statements in a function. Once the function executes a return statement, the program exits the function and continues with *its* next executable statement. If the function does not need to return any value, you can omit the return statement, although my preference is to always have a return, even if null.

Functions can be pretty elaborate; they can search for things in a list, determine variable types, open and close files, read and write to files.

To get started we will build a few really simple mathematical functions; we will need this skill in the future anyway, especially in scientific programming contexts.

User-built within a Code Block

For our first function we will code $$f(x) = x\sqrt{1 + x}$$ into a function named `dusty()` .

When you run the next cell, all it does is prototype the function (defines it), nothing happens until we use the function.

In [9]:
```
def dusty(x) :
    temp = x * ((1.0+x)**(0.5)) # don't need the math package
    return temp
# the function should make the evaluation
# store in the local variable temp
# return contents of temp
```

In [11]:
```
# wrapper to run the dusty function
yes = 0
while yes == 0:
    xvalue = input('enter a numeric value')
    try:
        xvalue = float(xvalue)
        yes = 1
    except:
        print('enter a bloody number! Try again \n')
# call the function, get value , write output
yvalue = dusty(xvalue)
print('f(',xvalue,') = ',yvalue) # and we are done
```

```
f( 4.0 ) =  8.94427190999916
```

Example: The Average Function

Create the AVERAGE function for three values and test it for these values:

- 3,4,5
- 10,100,1000
- -5,15,5

In [12]:
```
def AVERAGE3(x,y,z) : #define the function "AVERAGE3"
    avg = (x+y+z)/3  #computes the average
    return avg
```

In [13]:
```
print(AVERAGE3(3,4,5))
print(AVERAGE3(10,100,1000))
print(AVERAGE3(-5,15,5))
```

```
4.0
370.0
5.0
```

Example: The KATANA Function

Create the Katana function for rounding off to the nearest hundredths (to 2 decimal places) and test it for these values:

- 25.33694
- 15.753951
- 3.14159265359

In [14]:
```
def Katana(x) : #define the function "Katana"
    newX = round(x, 2)
    return newX
```

In [15]:
```
print(Katana(25.33694))
print(Katana(15.753951))
print(Katana(3.14159265359))
```

```
25.34
15.75
3.14
```

Exercise: The Wakizashi function

Create the Wakizashi function for rounding off to the nearest thousands (to 3 decimal places) and test it for these values:

- 25.33694
- 15.753951

- 3.14159265359

In [22]:
```
def Wakizashi(x): #define the function "Katana"
    retval = round(x,3)
    return retval
```

In [23]:
```
print(Wakizashi(25.33694))
print(Wakizashi(15.753951))
print(Wakizashi(3.14159265359))
```

```
25.337
15.754
3.142
```

Example: Variable Scope

An important concept when defining a function is the concept of variable scope. Variables defined inside a function are treated differently from variables defined outside.

We can leverage this feature to protect the calling program and not clobber variables as below

In [24]:
```
def a():
    x=25
    print('\n Variable x inside function a is',x)
    x = x +1
    print(' Variable x inside function a is',x,'\n')
    return x
```

In [25]:
```
# now a silly script
x = 13
print('Variable x before executing function a is',x)
y = a()
print('Variable x after executing function a is',x)
print('Variable y after executing function a is',y)
```

```
Variable x before executing function a is 13

 Variable x inside function a is 25
 Variable x inside function a is 26

Variable x after executing function a is 13
Variable y after executing function a is 26
```

Now we change **x** into a global variable within the function and get:

In [26]:
```
def a():
    global x
    x=25
    print('\n Variable x inside function a is',x)
    x = x +1
    print(' Variable x inside function a is',x,'\n')
    return x
```

In [27]:
```
# now a silly script
x = 13
print('Variable x before executing function a is',x)
```

```
y = a()
print('Variable x after executing function a is',x)
print('Variable y after executing function a is',y)
```

```
Variable x before executing function a is 13

 Variable x inside function a is 25
 Variable x inside function a is 26

Variable x after executing function a is 26
Variable y after executing function a is 26
```

Example: Saving a User Function as a Separate Module/File

In this section we will invent the `neko()` function, export it to a file, so we can reuse it in later notebooks without having to retype or cut-and-paste. The `neko()` function evaluates:

$$f(x) = x\sqrt{|(1 + x)|}$$

Its the same as the dusty() function, except operates on the absolute value in the wadical.

1. Create a text file named "mylibrary.txt"
2. Copy the neko() function script below into that file.

```
def neko(input_argument) :
    import math #ok to import into a function
    local_variable = input_argument * math.sqrt(abs(1.0+input_argument))
    return local_variable
```

1. rename mylibrary.txt to mylibrary.py (you may need to do this in the file explorer)
2. modify the wrapper script to use the neko function as an external module

bash commands to make the file and populate it directly on a linux (this is here as a backup) notice its a raw cell ! echo 'def neko(input_argument):' > line1.txt ! echo ' import math #ok to import into a function' > line2.txt ! echo ' local_variable = input_argument * math.sqrt(abs(1.0+input_argument))' > line3.txt ! echo ' return local_variable' > line4.txt ! cat line1.txt line2.txt line3.txt line4.txt > mylibrary.txt ! rm line*.txt ! mv mylibrary.txt mylibrary.py

In [28]:
```
# wrapper to run the neko function
import mylibrary
yes = 0
while yes == 0:
    xvalue = input('enter a numeric value')
    try:
        xvalue = float(xvalue)
        yes = 1
    except:
        print('enter a bloody number! Try again \n')
# call the function, get value , write output
yvalue = mylibrary.neko(xvalue)
print('f(',xvalue,') = ',yvalue) # and we are done
```

```
f( 6.0 ) =  15.874507866387544
```

In JupyterHub environments, you may discover that sometimes changes you make to your external python file are not reflected when you re-run your script; you need to restart the kernel to get the changes to actually update.

Exercise

Make a function that squares its input:

$$ f(x) = x^2 $$
and test it for the following values of x:

- -1
- 0.0
- 1.0
- 2.0
- 3.0

In [30]:
```
# define the function here
def xsquared(x):
    return x ** 2
```

In [31]:
```
print(xsquared(-1))
print(xsquared(0.0))
print(xsquared(1.0))
print(xsquared(2.0))
print(xsquared(3.0))
```

```
1
0.0
1.0
4.0
9.0
```

Readings

Here are some great reads on this topic:

- **"Functions in Python"** available at *https://www.geeksforgeeks.org/functions-in-python/
- **"Defining Your Own Python Function"** by **John Sturtz** available at *https://realpython.com/defining-your-own-python-function/
- **"Graph Plotting in Python | Set 1"** available at *https://www.geeksforgeeks.org/graph-plotting-in-python-set-1/
- **"Python Plotting With Matplotlib (Guide)"** by **Brad Solomon** available at *https://realpython.com/python-matplotlib-guide/

Here are some great videos on these topics:

- **"How To Use Functions In Python (Python Tutorial #3)"** by **CS Dojo** available at *https://www.youtube.com/watch?v=NSbOtYzIQI0
- **"Python Tutorial for Beginners 8: Functions"** by **Corey Schafer** available at *https://www.youtube.com/watch?v=9Os0o3wzS_I
- **"Python 3 Programming Tutorial - Functions"** by **sentdex** available at *https://www.youtube.com/watch?v=owgINL1KQf0