Lecture 4 How to Make and Use Functions

- A **function** is a re-usable piece of code that performs operations on a specified set of variables, and returns the result.
- In this week, we have worked with many built-in python and numpy **functions** like `print` and `np.sort`, etc.
- In addition, in your homework you will explore functions like mean, median, etc.

Why do I need this?

- Some pieces of code perform a particular operation which you plan to use many times.
- Rather than copying code over and over again you can use the function.
- If we define a function it becomes easy to use the same code in many programs.
- It allows you to have confidence that you have solved one piece of a larger puzzle.
- It improves testing, readability, and reliability by breaking up the program into smaller units.

In [1]:

```
import numpy as np
from numpy import random
rng = random.default_rng(seed = 1967)
```

Syntax of a Function

- These are the essential pieces to a function:

1. `def` statement naming the function and how to call it. The def statement should end with a colon :
2. The function has a name. This name will be used to call the function.
3. The function may have **arguments**. Arguments are variables passed into the function.
4. `return` statement indicating what variables are returned (almost always!).

- So, the most basic structure of a function definition is
  def function_name(arg): some code that works on arg and creates output_variable return output_variable

In [2]:

```
### Example: Square a number
def square(x):  #Here, x is called the argument.
    y = x**2    #Here I do the calculation
    return y    # here i tell it to return y
```

In [3]:

```
x = 5
y = square(x)
print(y)
```

```
25
```

In [4]:

```
### I dont really need to define x, I could just pass the number in
y = square(2)
print(y)
```

```
4
```

- There is nothing special about x and y, I could use any variable names.
- The only name that is important is the **function** name, which is `square`
- The function name is defined in the `def` statement

In [5]:

```
my_var = 3
my_var_sq = square(my_var)
print(my_var_sq)
```

```
9
```

Functions Have Independent Namespaces

- What is a Namespace?

1. The Namespace is the total set of variables, object, functions defined by you while programming.
2. You can quickly view the namespace using the Variables pane
3. The Namespace of a function is *hidden*.
4. The variables inside a function are not visible to you, unless you explicitly return them.

In [6]:

```
def cube(z):
    z_cube = z**3
    return z_cube

my_number = 4
my_cube = cube(my_number)
print(my_cube)
```

```
64
```

In [7]:

```
### z and z_cube only exist inside the function.
print(z_cube)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[7], line 2
      1 ### z and z_cube only exist inside the function.
----> 2 print(z_cube)

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

In [8]:

```
def cubeplusone(z):
    z = z+1  #notice here the value of z is being updated by adding 1.
    z_cube = z**3
    return z_cube
x = 2
y = cubeplusone(x)
print('x = ',x)  # even though x was put in the position of the argument z and 1 was added to z nothing happens to x
print('y = ',y)
```

```
x =  2
y =  27
```

In [9]:

```
z = 1
z_cube = cubeplusone(z)
print(z)
print(z_cube)
```

```
1
8
```

- even if the variable name z is inside and outside the function it does not get modified in the global name space.
- z inside the function and z outside the function are separate variables

What happens in the function, stays in the function

- The only thing you can see is what comes back in the `return` statement.

Print statements inside functions.

- One way to see what is happening inside the function is a print statement inside the function.
- Another way is a debugger but that is IMO unneccesarily complicated for this class.

In [10]:

```
def cubeplusone(z):
    z = z+1  #notice here the value of z is being updated by adding 1.
    z_cube = z**3
    print(z,z_cube)
    return z_cube
x = 2
y = cubeplusone(x)
print('x = ',x)  # even though x was put in the position of the argument z and 1 was added to z nothing happens to x
print('y = ',y)
```

```
3 27
x =  2
y =  27
```

- `print` is not the same thing as `return`

In [11]:

```
def cubeplustwo(z):
    z = z+2  #notice here the value of z is being updated by adding 1.
    z_cube = z**3
    print(z,z_cube)
testin = 2
testout = cubeplustwo(x)
print('testin = ',testin)  # even though x was put in the position of the argument z and 1 was added to z nothing happens to
print('testout = ',testout)
```

```
4 64
testin =  2
testout =  None
```

- If you want to get something back from a function you need a `return` statement

Multiple Inputs and Outputs To A Function

- A function can take multiple input and return multiple outputs.

In [12]:

```
def npower(x,n):
    y = x**n
    return y

z = 3
m = 4
y = npower(z,m)  # this computes z**m.
print(y)
```

```
81
```

In [13]:

```
def power_and_root(x,n):
    y = x**n
    z = x**(1/n)
    return y,z

z = 4
m = 2
y1,y2 = power_and_root(z,m)  # this computes z**m and z**(1/m) which is the mth root of z.
print(y1)
print(y2)
```

```
16
2.0
```

In [14]:

```
### What if you forget that there are 2 outputs?
y = power_and_root(z,m)
print(y)
```

```
(16, 2.0)
```

In [15]:

```
### Then you get them back as a list (Actually a tuple).
print(y[0])
print(y[1])
```

```
16
2.0
```

Functions Can Interact with the Global Namespace

- The first time I encountered this, it made me really uncomfortable.

In [16]:

```
def zum(a,b):  #Here, a,b the argument.
    c = a+b
    d = (a+b)/z #Notice, z does not appear in the argument list.
    return c,d
```

In [17]:

```
a = 1
b = 2
z = 3
c,d = zum(a,b)
print(c)
print(d)
```

```
3
1.0
```

- This should bother you!
- Why is it that I can use the variable z, even though it was not one of the arguments.

In [18]:

```
def zum_a1(a,b):  #Here, a,b the argument.
    a = a+1
    c = a+b
    d = (a+b)/z #Notice, z does not appear in the argument list.
    return c,d
```

In [19]:

```
a = 1
b = 2
z = 3
c,d = zum_a1(a,b)
print(c)
print(d)
print(a)
```

```
4
1.3333333333333333
1
```

- Functions have access to the global namespace!

- In this next example, I plan to manipulate a variable in the global name space.

In [20]:

```
def zum_z1(a,b):  #Here, a,b the argument.
    z = z+1 # here I add 1 to z and place it in z. z is a global namespace variable.
    c = a+b
    d = (a+b)/z
    return c,d
```

In [21]:

```
a = 1
b = 2
z = 3
c,d = zum_z1(a,b)
print(c)
print(d)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[21], line 4
      2 b = 2
      3 z = 3
----> 4 c,d = zum_z1(a,b)
      5 print(c)
      6 print(d)

Cell In[20], line 2, in zum_z1(a, b)
      1 def zum_z1(a,b):  #Here, a,b the argument.
----> 2     z = z+1 # here I add 1 to z and place it in z. z is a global namespace variable.
      3     c = a+b
      4     d = (a+b)/z

UnboundLocalError: cannot access local variable 'z' where it is not associated with a value
```

- While I can use global namespace variable z in the function, **I cannot update its value**

- Do I need any arguments at all?

In [22]:

```
def zum():  #Here, a,b the argument.
    c = a+b #Notice, a,b does not appear in the argument list.
    d = (a+b)/z  #Notice, z does not appear in the argument list.
    return c,d
```

In [23]:

```
a = 1
b = 2
z = 3
c,d = zum()
print(c)
print(d)
```

```
3
1.0
```

- This is bad practice. I think in general, you should try to use arguments.

Global versus Local Namespace

- Any function can see all the variables in the **global** namespace and use them in calculations.
- *But it cannot manipulate them or change their values*
- Any argument passed into the function can have its value changed inside the **local** namespace of the function, without changing its value in the global namespace.
- Values of variables in the **global** namespace are only changed by returning the value and **replacing** the original value.

In [24]:

```
def zum_a1(a,b):  #Here, a,b the argument.
    a = a+1
    d = (a+b)/z #Notice, z does not appear in the argument list.
    return d,a
```

In [25]:

```
a = 1
b = 2
z = 3
c,e = zum_a1(a,b)
print(c) #this is d inside the function
print(e) #this is a inside the function
print(a) #this is a in the global namespace
```

```
1.3333333333333333
2
1
```

- I could just choose to overwrite a value in the global name space

In [26]:

```
a = 1
b = 2
z = 3
print(a) # this is a before the function
c,a = zum_a1(a,b)
print(c) #this is d inside the function
print(a) # this is a after the function runs
```

```
1
1.3333333333333333
2
```

When do I use an argument?

- Almost always. As good programming practice, I like to use arguments as much as possible

When do I prefer not to use an argument?

- The main case is **data**.
- data should never be changed.
- If all your analysis is written in functions and the data remains global it cant be accidently changed.

Keyword Arguments

There are two types of inputs to a function:

- postional arguments - the variable corresponding to the argument is based on its position in the argument list.
- keyword arguments - the variable corresponding to the argument is made explicit with an equality sign.

- In the example functions I made today, I only used positional arguments.
- But actually I have made use of a keyword argument previously

```
st = [1,2,3]
st_float = np.array(lst,dtype='float')
```

- In the call to np.array, `st` is a positional argument which has the list we want to convert into an array
- `dtype = 'float'` is a **keyword** argument, where the **keyword** is `dtype` and the value is `'float'`

- You can always use keyword arguments instead of positional arguments.
- The advantages of keyword arguments are:

1. you dont have to remember the order of the arguments.
2. you can set **default** values

- Any function can always be called with keyword arguments.

In [27]:

```
def power_and_root(x,n):
    y = x**n
    z = x**(1/n)
    return y,z

y,z = power_and_root(x=9,n=2)
print(y)
print(z)
y,z = power_and_root(n=2,x=9)  #now the order DOES NOT MATTER because I am making the variables explicit.
print(y)
print(z)
```

```
81
3.0
81
3.0
```

- You can even mix together positional and keyword arguments.

In [28]:

```
y,z = power_and_root(9,n=2)
print(y)
print(z)
```

```
81
3.0
```

- As soon as you use a keyword argument you can no longer use positional arguments.

In [29]:

```
y,z = power_and_root(x=9,2)
print(y)
print(z)
```

```
  Cell In[29], line 1
    y,z = power_and_root(x=9,2)
                               ^
SyntaxError: positional argument follows keyword argument
```

- Functions can be used with

1. positional arguments alone. In this case, you need to know the correct order.
2. keyword arguments alone. In this case you can provide them in any order.
3. positional and keyword arguments. In this case, positional arguments must come first then keyword arguments.
4. As soon as you make use of a keyword argument, you can no longer use positional arguments.

Default values for arguments

- The most useful thing about keyword arguments, is that they can be used to set default values of arguments in the function `def` statement.
- This means if you use a function with a particular argument most of the time, you can just ignore those arguments and let them assume predefined values.

In [30]:

```
def npower(x,n=2):  # notice that in the definition of the function, I provided a default value of n
    y = x**n
    return y
```

- When defining a function, you can determine **default** values of the arguments.
- Then if those arguments are not provided the function will assume the default value applies.

In this example, I only provide the value of the base (x) and it automatically assumes the power n = 2*

In [31]:

```
a = 3
z = npower(a)
print(z)
```

```
9
```

- You can always override the default by passing an argument.

In [32]:

```
#I can override the default with a keyword argument.
z = npower(3,n=3)
print(z)
```

```
27
```

In [33]:

```
#I can override the default with a positional argument.
z = npower(3,4)
print(z)
```

```
81
```

In [34]:

```
#But you must provide a value for x or it fails.
z= npower(n=3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 2
      1 #But you must provide a value for x or it fails.
----> 2 z= npower(n=3)

TypeError: npower() missing 1 required positional argument: 'x'
```