Background Reading #2

Table of Contents

1. Command and Shortcut Summary

Command/Symbol	**Description**
def function_name([input_variables]):	Used to define the start of function, where function_name is the chosen name of your function, the round brackets contain the input parameters, and the colon signifies that everything following is inside of the space or scope of that function
return	Signals the end of a function, where the object(s) (e.g., a variable(s)) that you wish to be the output of the function follow the return command
None	A special word that is often used as a placeholder or default assignment or output. Can be used in the return line to show that a function is closed and nothing is returned, e.g., return None
If/elif/else	Commands that precede branching options in the code.
import	Tells your editor that you would like to import either an entire package from a library you have downloaded to the IDE (Eclipse, or other) or one or more specific functions from that library.
Useful Operators for if Statements	**Description**
a > b	a is greater than b
a >= b	a is greater than or equal to b
a ==b	a is equal to b
a != b	a is not equal to b
a < b	a is less than b
a =< b	a is less than or equal to b

2. Branching

Sometimes you want to execute code differently based on the value of a certain variable. This is most commonly done with “if-else” statements. If-else branching allows the code to execute different branches based on the value of the expressions. The following mock code will execute the block of code [some code] if expression1 evaluates to true, execute [some other code] if expression1 evaluates to false and expression2 evaluates to true, or will execute [something else] if both expression1 and expression2 evaluate to false.

```
if expression1:
        [some code]

elif expression2:
        [some other code]

else:
        [something else]
```

There could also be more than one elif condition, or the else and elif statements could not be there at all. That is, it is possible to have an if without an elif or an else.

We note that after using the colon, the next line is automatically indented by Eclipse. This is because Python groups everything that must be done if the condition before the colon is met into a “code block”. We also note that all commands in Eclipse (if/else, print, etc.) are lowercase.

The following section of code shows an example of how to handle a list index that could potentially be the wrong value based on the length of a certain list by using if-else statements.

```
elementslist = ['uranium', 'plutonium', 'mercury']

# the elements at each index are 0= uranium, 1=plutonium, and 2=mercury
# if we want to access one of these elements, we have to use an index
# of 0, 1, or 2. Other index values will give an error

# we can get around this with some if-else statements
# let's define the max possible value of the list index (the length - 1)
# this is because python starts numbering at 0, so while the length of
# elementslist is 3, elementslist[3] doesn't exist
max_user_value = len(elementslist) - 1

# this is our fake user input
desired_list_element = rd.randint(0, 10)
print ('You chose list index number: ', desired_list_element)

if desired_list_element > max_user_value:
    # possible solution could be picking a random number in the
    # correct range and reassigning desired_list_element to this new value
    desired_list_element = rd.randint(0, max_user_value)

    # line break, \, added when code was too long (generally > column 80)
    print ('Your chosen value was too high, a value of', desired_list_element,\
           'was chosen.')
    # another way to "wrap" the text on the screen (so it doesn't run on
    # horizontally), is to put round brackets around portions of your code.
    # the practice is to hit enter immediately after the start of the brackets,
    # then once again before the closing bracket, visually separating the
    # bracketed text with a bracket above and a bracket below, as follows:
    print (
        'This is the element at the chosen index #: ',
        elementslist[desired_list_element]
        )
```

Console Output:

```
You chose list index number:  9
Your chosen value was too high, a value of 1 was chosen.
This is the element at the chosen index #:  plutonium
```

Note: if we want to confirm whether two things are equal via if statements, we use the == operator (see the Command and Shortcut Summary table at the start of this guide).

```
A = 5
if A == 5:
    print ('double equals sign used to check if A is equal to 5')
```

None is a special “null” keyword in Python that tells the code nothing needs to be done. None can also be assigned to a variable as a default (more on this in the Functions Section). It is often used as a placeholder or a way to neatly wrap up an if statement for completeness.

It is sometimes not necessary to use, such as in the example above, where if A is not equal to 5, the computer knows not to do anything if there was no else statement.

In a longer if statement with many branches, using else: None can be a clear way to show that an if statement is closed.

```python
A = 5
B = 33
if A == 5:
    print ('double equals sign used to check if A is equal to 5')
    if B > 35:
        print ('I do not like this value of B')
    elif B < 35:
        print ('This value is better')
else:
    None
```

Console Output:

```
double equals sign used to check if A is equal to 5
33 : This value of B is okay
```

3. FUNCTIONS

Defining functions will make our code more robust, less prone to errors, and more usable. When we define a function what we want to do is to create a concrete set of steps that we want to execute in our code. We will be able to run the same lines of code repeatedly without typing them over and over. Functions are very flexible in both the inputs they can take, and the outputs they can return. You can define a function using the syntax:

```
def function_name(input_variables):
        code
        return output_variables
```

In this definition, the name of the function is function_name and the input variables to the function are separated by commas in the place of input_variables. At the end of the function, we have a return statement. This tells Python what the function is returning to the caller. The function above returns the variable, or variables, listed in the place of output_variables. We can create a function to calculate and return the y coordinates of points on of various lines given the input of the x coordinate and the y-intercept. We will then print the output of the function.

```python
def line_solver(x, y_int):
    m = 4
    y = m * x + y_int
    return y

print ('the value of y is:', line_solver(2, 4))
```

Console Output:

```
the value of y is: 12
```

In the above example, we called our function **line_solver** by listing out the arguments in the order that it expects them: slope, y_int. Nevertheless, Python allows you to call them in any order, as long as you are explicit about what goes where. In the next snippet of code, we reverse the order of x and y_int from their location in the function definition.

```
print ('with explicit definition of variables, the value of y is still:', \
       line_solver(y_int=4, x=2))
```

Console Output:

```
with explicit definition of variables, the value of y is still: 12
```

If our function has multiple returns (or outputs), we can grab the first entry in the list returned by the function using square brackets on the end of the function call. We can also assign the two entries in the list to two variables in a single line.

```
def line_solver_2(x, y_int):
    m = 4
    y = m * x + y_int
    return y, m

y_val, slope = line_solver_2(x=2, y_int=4)
y_val = line_solver_2(x=2, y_int=4)[0]
```

It is often a good idea to call a function explicitly, that way if you mess up the order of the arguments, it does not matter. Simply calling a function without printing or assigning it (where would return/output to a variable) doesn’t result in anything happening outside of the function. We learn more about this in the next subsection on scope. E.g., in the example above, **y_val** and **slope** are assigned to the outputs of **line_solver_2**. In the example below, we call the **line_solver_2**, but without its assignment to anything, these output(s)/return(s) have no way to be stored outside of the function:

```
line_solver_2(x=2, y_int=4)[0]
```

3.1 Function Scope

The variables that we define in our code store information in the computer’s memory. Scoping rules are, in essence, a way for the program to separate information in memory and control access to that information. Functions have their own “scope” in memory that is different than the memory used by other parts of a code.

When we start experimenting with functions, we quickly realize that most types of objects that we define inside of a function (e.g., variable assignments, created lists, etc.) only exist while we are in that function’s space or scope. Things that are in the return line and assigned to variables outside of the function and used later in the code will live on outside of the function. Otherwise, things defined inside the function only exist while the function is running and do not have a “global” scope.

Below, the variables **x** and **y** are defined outside the function **testfunction**. The variable **y** is then passed through the function (e.g., the input). Even though the function redefines **x** and **y**, printing them after calling the function reveals that the original values of these variables, as defined outside the function, have not changed. This is because calling **testfunction** creates its own memory space and any variable created in there is different than in the rest of the program, even if the variables have the same name! Scope is one of the more important things to understand when using functions. There are a few exceptions to this rule, one of which is lists or arrays, discussed below in the Lists and Functions section.

```
x = 4
y = [1, 3, 4]
z = []

constant = 4

def testfunction(input1):
    x = 10 * constant
    y = [i * 2 for i in input1]

    for i in range(y[2]):
        q = i + 1
        z.append(q)
    z[0] = 10
    return x, y
# calling function does not update previously defined x and y (outside function)
# Even though there are variables with same names inside testfunction,
# their scope is only inside that function

testfunction(y)

print ('x and y are equal to:', x, y, ': x and y are not updated.')
```

Console Output:

```
x and y are equal to: 4 [1, 3, 4] : x and y are not updated.
```

```
# if x and y are redefined outside of the function, in the scope of the module,
# they are updated (here, based on function output)

x, y = testfunction(y)

print ('x is equal to:', x, y, ''': x and y are now equal to the function return
that they were reassigned to.''')

print ('''z was a list defined outside of the function and added to
inside the function. z was updated twice because the function was called twice:
''', z)
```

Console Output:

```
x is equal to: 40 [2, 6, 8] : x and y are now equal to the function returns
that they were reassigned to.
```

```
z was a list defined outside of the function and was added to
inside the function. z was updated twice because the function was called twice:
 [1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8]
```

3.2 Mutable Objects

In the example above, while the variables **x** and **y** are not updated outside of the function simply because they are redefined inside of it, the list **z** is updated outside of the function memory due to operations that take place inside **testfunction**.

This changing of memory outside the function is called a “side effect” because it is a way that a function interacts with the rest of the code not through the mechanism of returning information from a function. The occurrence of side effects is specific to data types such as lists, which are known as “mutable.”

A mutable object is an object whose value can change once created. Mutable objects are often objects that can store a collection of data, such as lists. For example, you can make a list of items you’ll need from the grocery store; as you decide what you need, you add them to the list. As you buy things, you remove them from the list. Notice that you’re using the same list and making modifications to it (crossing out or adding to the end) as opposed to having many lists in which you copy over items every time you want to make a change.

“Mutable” data types such as lists are passed by what is known as “by reference” to functions. This means that a function can change the values of a mutable data type when it is the input of the function. Any data type that can have its size modified is called a mutable type, and these are passed by reference to functions. The addition of a modification to the input1 argument, a list, which is mutable, inside the function will update the variable **y** outside the function.

```
def testfunction(input1):
    x = 10 * constant
    # this variable y is not the same as y outside the function
    # the scope and memory inside the function does not carry over
    # unless a "mutable" variable is being changed, e.g., a list
    y = [i * 2 for i in input1]

    # this changes the first entry of a list input
    # which carries over to the memory outside the function as lists
    # are mutable or changeable data types
    input1[0] = 10
    for i in range(y[2]):
        q = i + 1
        z.append(q)
    z[0] = 10
    return x, y

testfunction(y)
print (y)
```

Console Output:

```
[10, 3, 4]
```

3.3 Function Docstrings

Docstrings are long comments at the beginning of the body of a function that tells the user what the function needs as input parameters and what the function returns, in addition to any side effects.

Side effects of functions can include printing something to the screen, writing to file, modifying a list that was passed to the function, or making a plot. While some programmers will argue that the correct (and most clear) way to use functions is to only use the return line to make changes to your overall code. Meaning, you use the return of a function to assign or reassign some variable, list, etc. However, as long as functions are written clearly and even more importantly, documented properly, side effects are okay to have. The user of the code simply needs to be aware if there are other changes with global, or outside of just the function's memory, scope.

Shown below is the **testfunction**, but now with included docstring documentation at the start (indicated by the triple quotation marks at the start and end of the green block of text).

```
def testfunction(input1):
    # below is the function docstring
    '''
    Calculates the values of x and y.

    Args:
    input1: list input

    Return: returns two objects, x and y.
    x is an integer or float
    y is a list

    Side effects:
    the first element of the input list is modified after y is calculated
    the list z is appended to using on a for-loop with a range
    dictated by the second element of the list, y
    '''

    x = 10 * constant
    # this variable y is not the same as y outside the function
    # the scope and memory inside the function does not carry over
    # unless a "mutable" variable is being changed, e.g., a list
    y = [i * 2 for i in input1]

    # this changes the first entry of a list input
    # which carries over to the memory outside the function as lists
    # are mutable or changeable data types
    input1[0] = 10
    for i in range(y[2]):
        q = i + 1
        z.append(q)
    z[0] = 10
    return x, y
```

4 Python and External Packages

4.1 Pediodictable Pacakge

Instructions on how to install packages into Eclipse using both the GUI and the shell are in the **Coding Exercises > Coding Exercise 2** folder on D2L in a separate .pdf document called “Package Import Instructions_CodEx2”.

For any package, we can find the documentation with a quick Google search. A straightforward guide that goes through the full features (with examples) of the periodictable package can be found here: https://periodictable.readthedocs.io/en/latest/guide/using.html

We will go through a few brief examples here. Elements from the periodic table can be accessed using their symbols. Python uses dot notation to access the attributes (e.g., element or isotope properties) and methods (e.g., functions that can calculate stuff related to the element/isotope) of objects (more on this later when we discuss classes in the next coding exercise).

```python
import periodictable as pdt

# if we only want to access particular elements by name:
from pdt import hydrogen
print (hydrogen.mass)

# we could also access particular elements as symbols
print (pdt.Li.mass, pdt.H.mass, pdt.Ca.mass)
```

Console Output:

```
1.00794
6.941 1.00794 40.078
```

We can also assign our own variables to the elements and access information that has been built-in by the creators who made this package.

```python
# accessing mass_units attribute
print (pdt.H.mass_units)
# if we want, we can assign this element to our own variable
hydr = pdt.H
print (hydr.mass_units)
# assigning mass of element to our own variable
mass_hydr = hydr.mass
# other way of assigning mass of element to our own variable
mass_hydr = pdt.H.mass
print (mass_hydr)

# lists atomic mass numbers of all isotopes of an element
print (pdt.Au.isotopes)
```

Lastly, we can access isotopes and their information using square brackets and the atomic mass number. You can create a unique variable for an individual isotope.

```python
# assigning our own variable (H_3) to the tritium isotope of hydrogen
H_3 = pdt.H[3]
print (H_3.number, 'this is the number of protons or atomic #, Z')
print (H_3.isotope, 'this is the atomic mass number (A = Z + N)')
```

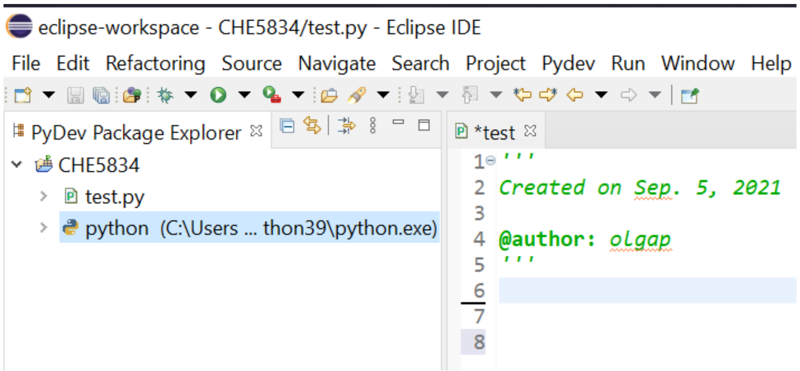
5 Extra Useful Stuff

5.1 Code Wrapping

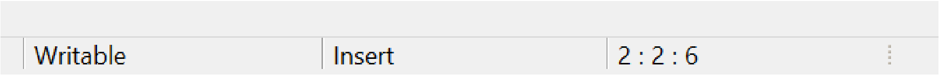
The bottom right bar of the Eclipse window will have three numbers, separated by colons. These indicate the line number (also shown in front of each line in the text editor when turned on, as above), the column number at each line, and the character count (going to a new line counts as two characters).

E.g., placing the cursor after the letter ‘C’ in the string ‘Created’ in the screen shot below has the coordinates 2 : 2 : 6 (second line, second column, overall 6th character from the start of document).

We will use the column number as a way to manually wrap our code **at roughly column 80.**


eclipse-workspace - CHE5834/test.py - Eclipse IDE
File Edit Refactoring Source Navigate Search Project Pydev Run Window Help
PyDev Package Explorer
CHE5834
test.py
python (C:\Users ... thon39\python.exe)
*test

```
'''
Created on Sep. 5, 2021

@author: olgap
'''
```



Writable | Insert | 2 : 2 : 6

This is done by keeping an eye on the middle value and typically adding a \ character to break the line and move onto the next line without interruption to your code.

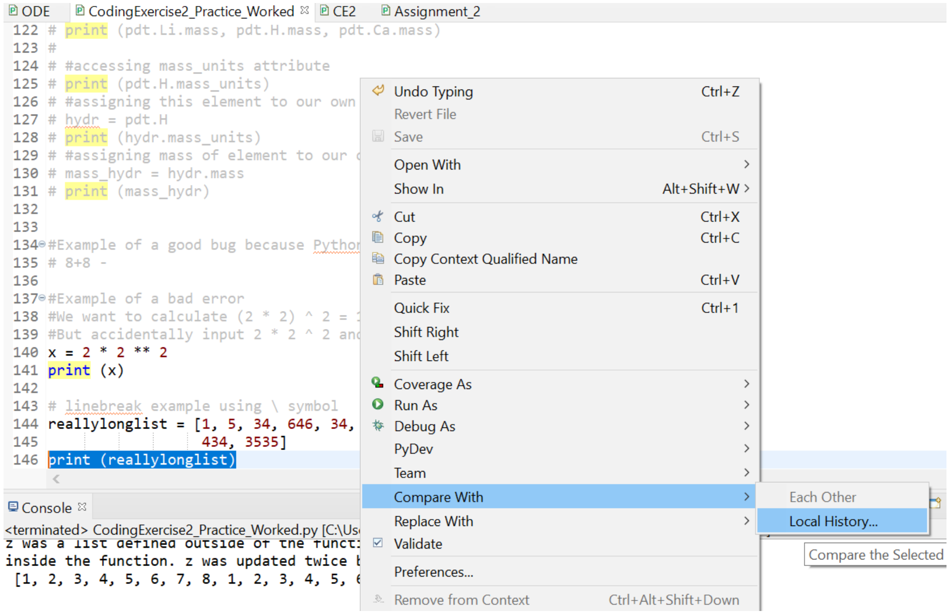
```
reallylonglist = [1, 5, 34, 646, 34, 534346, 4642, 23423, 343, 3434, 1, 43, \
                  434, 3535]

print (reallylonglist)
```

Console Output:

```
[1, 5, 34, 646, 34, 534346, 4642, 23423, 343, 3434, 1, 43, 434, 3535]
```

5.2 Checking Revision History

We choose 80 because this works well with non-wide screen computer monitors if you need to look at two code editor windows side-by-side. Most codes are backed up on physical or virtual servers by using “repositories”. Since this is outside the scope of our course activities, if you are working on a complicated code and would like to compare to a previous version, we can access this in Eclispe by right clicking anywhere in the editor window, selecting **Compare with** and then **Local history**

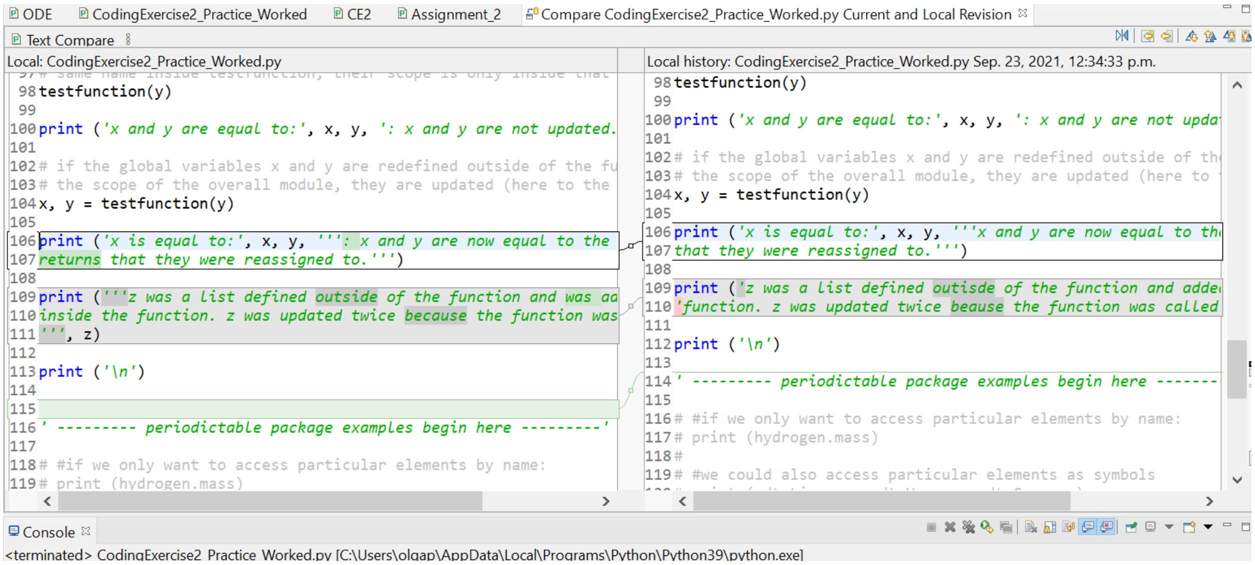


```
# print (pdt.Li.mass, pdt.H.mass, pdt.Ca.mass)
#
# #accessing mass_units attribute
# print (pdt.H.mass_units)
# #assigning this element to our own
# hydr = pdt.H
# print (hydr.mass_units)
# #assigning mass of element to our
# mass_hydr = hydr.mass
# print (mass_hydr)


#Example of a good bug because Pythor
# 8+8 -

#Example of a bad error
#We want to calculate (2 * 2) ^ 2 = 1
#But accidentally input 2 * 2 ^ 2 and
x = 2 * 2 ** 2
print (x)

# linebreak example using \ symbol
reallylonglist = [1, 5, 34, 646, 34,
                  434, 3535]
print (reallylonglist)
```

```
<terminated> CodingExercise2_Practice_Worked.py [C:\Us
z was a list defined outside of the funct
inside the function. z was updated twice b
 [1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6
```

This will bring up your revision save history over the last two days (Ctrl + S shortcut for saving works in Eclipse, but a module is also saved by default each time the code in it is run). This will likely show up on the right-hand side of your Eclipse workspace in a tab beside the Outline.

The local history is configured in **Preferences** > **General** > **Workspace > Local History**. You can increase the number of days history is kept, but this only starts working from the time you set it. Selecting one of the revision time stamps will bring up the two windows side-by-side. This opens in an entirely new module tab. You can scroll through these to compare your changes and potentially rescue parts of your code to something that worked earlier!



ODE
CodingExercise2_Practice_Worked
CE2
Assignment_2
Compare CodingExercise2_Practice_Worked.py Current and Local Revision
Text Compare
Console

Local: CodingExercise2_Practice_Worked.py

```
# same name inside testfunction, their scope is only inside that
testfunction(y)

print ('x and y are equal to:', x, y, ': x and y are not updated.
 
# if the global variables x and y are redefined outside of the fu
# the scope of the overall module, they are updated (here to the
x, y = testfunction(y)

print ('x is equal to:', x, y, ''': x and y are now equal to the
returns that they were reassigned to.''')

print ('''z was a list defined outside of the function and was aa
inside the function. z was updated twice because the function was
''', z)

print ('\n')


' --------- periodictable package examples begin here ---------'

# #if we only want to access particular elements by name:
# print (hydrogen.mass)
```

Local history: CodingExercise2_Practice_Worked.py Sep. 23, 2021, 12:34:33 p.m.

```
testfunction(y)

print ('x and y are equal to:', x, y, ': x and y are not upda

# if the global variables x and y are redefined outside of th
# the scope of the overall module, they are updated (here to
x, y = testfunction(y)

print ('x is equal to:', x, y, '''x and y are now equal to th
that they were reassigned to.''')

print ('z was a list defined outisde of the function and adde
'function. z was updated twice beause the function was called

print ('\n')

' --------- periodictable package examples begin here -------

# #if we only want to access particular elements by name:
# print (hydrogen.mass)
#
# #we could also access particular elements as symbols
```

```
<terminated> CodingExercise2_Practice_Worked.py [C:\Users\olgap\AppData\Local\Programs\Python\Python39\python.exe]
```

6 REFERENCES:

"Computational Nuclear Engineering and Radiological Science Using Python," Ryan G. McClarren, Elsevier, 2017.