**Discussion Assignment U-1**

>>> print 'Hello, World!'

File "<input>", line 1

    print 'Hello, World!'

                        ^

SyntaxError: Missing parentheses in call to 'print'. Did you mean print('Hello, World!')?

In this example we are calling on the print function in python. In this case to “print” the words hello world, But we get a syntax error.  Meaning the python cannot interpret our input. We will need to add parentheses to indicate that we are calling on the print function.

>>>print('hello world!')

hello world!

In python 2 we do not need to use the parentheses to print. Meaning that print is not a function in python 2, whereas in python 3 we do need to use the parentheses. Indicating that in python 3 print is a function.

>>> 1/2

0.5

In this example we have used python like a calculator by performing simple division.

>>> type(1/2)

<class 'float'>

Numbers are grouped into “types” or classes for example we get. Integers and floats. In this case we have asked python what type 1/2 is, python responds with “float” indicating to us that this a decimal number.

>>> print(01)

File "<input>", line 1

    print(01)

           ^

SyntaxError: invalid token

When putting a 0 in front of a number in python we are saying that the number after the 0 should be interpreted as an octal digit. We should instead write 0o1 python will now be able to interpret this as an octal number

print(0o1)

1

>>> 1/(2/3)

1.5

Once again we are using python to solve a simple math. Giving us the correct answer of 1.5

References:

Wikipedia. Ocal (2019) Retrieved from

[**https://en.wikipedia.org/wiki/Octal**](https://en.wikipedia.org/wiki/Octal)

Allen Downey (2015) Think Python, How to Think Like a Computer Scientist. Needham, Massachusetts. Green Tea Press

**Discussion Assignment U-2**

Example 1

**def your\_age(X):  
    print(X)  
  
your\_age(33)**

**Result> 33**

In this code the (X) is the parameter. The parameters are placed between the brackets after the definition name. we can have multiple parameters separated by commas. When we call the function we enter the argument for the parameters in the case 33.

Example 2

**def your\_age(X):  
    print(X)  
  
your\_age(55)**

**Result> 55**

55 would be the value of the argument in this case

**def your\_age(X):  
    print(X)  
  
my\_age=55  
  
your\_age(my\_age)**

**Result> 55**

In this example I have created a variable “my\_age” and assigned a value of 55 to it. I then called on the function “your\_age” using the variable “my\_age” as the argument.

**def your\_age(X):  
    print(X)  
  
your\_age(2019-1964)**

**Result> 55**

This example uses the expression 2019-1964 as the argument when calling on the function ‘your\_age’

Example 3

**def your\_age(birth\_year,current\_year):  
    X=current\_year-birth\_year  
    print(X)       #calling on variable X inside the function  
  
your\_age(1964,2019)**

**print(x)           #calling on variable X outside the function**

**Result>**

**55**

**Traceback (most recent call last):**

**line 10, in <module>**

**print(x)**

**NameError: name 'x' is not defined**

I have created a variable X that uses current year and birth year to calculate your age. Inside the function your\_age

I used the variable X both inside and outside the function to print X

As we can see by the result the variable X printed while in the function, but did not print X when we called on it outside of the function. This shows us that variables are localto the function that they were created in.

Example 4

**def about\_me(f\_name,country):  
    X=f\_name+country  
    print(X)  
  
print(country)**

**Result>**

**Traceback (most recent call last):**

**line 6, in <module>**

**print(country)**

**NameError: name 'country' is not defined**

In this example I have used the print function, to call on the parameter named ‘country’ outside of the function and the result is an error this once again demonstrates that the name assigned to the parameter is local to the function.

Example 5

**def sum ():  
    x=55  
    print(x)  
  
x=10  
  
print(x)  
sum()**

**Result>**

**100**

**550**

Here we have assigned a value to X both inside and outside of the function. And printed X in each case. As we can see neither of the variables are affect by the inside or outside variable.

However I would like to point out that we can call on a variable that has been defined outside of a function within a function.

def sum ():  
    y=x\*10         #here I have called on the variable x outside of the function  
  
    print(y)  
  
x=10  
  
sum()

print(x)

**Result>**

100

10

This shows us that the variable X will not be changed if called on in the function but the value can be used in a function

**Discussion Assignment U-3**

Hi, I will explain what is the difference between chained conditional and nested conditional.

I will use this code that I wrote as an example:

#example1  
answer = input('You want some coffee?<Yes/No>:').lower()  
if answer == "yes":  
    print("here it is.")  
elif answer == "no":  
    answer1 = input("Are you sure?I will make it more cheap for you.<Alright /Ok /No>:").lower()  
    if answer1 =='alright':  
        print('This coffe is the best coffee in this town!')  
    elif answer1 == 'ok':  
        print('You will love it!')  
    else:  
        print('Okay.')  
else:  
    print("Sorry, I don't understand it.")  
print('Thank you , and have a nice day!')

***"Sometimes there are more than two possibilities and we need more than two branches."(Downey A, 2015, p.41).***

When you want more than 2 options you can use the chained conditional, In the #exemple1 I'm using the chained conditional asking the user if  ('You want some coffee?') with a prompt as an argument, then the user will answer,**if** he wants some coffee he will write something like the word "yes" and the program will return printing the string"('here it is.)", **elif** he doesn't want he will write something like the word "no" and the program will return printing the string "("Are you sure? i will make it more cheap for you")" but if he writes something **else** the programs will print the string ("Sorry, I don't understand it."), here the word "elif" is same as "else if".

Basically the nested conditional is a conditional inside another conditional, in the #exemple1 on the elif statement  when you answer "no" he will run another set of branches :

elif answer == "no":  
    answer1 = input("Are you sure?I will make it more cheap for you.<Alright /Ok /No:").lower()  
    if answer1 =='alright':  
        print('This coffe is the best coffee in this town!')  
    elif answer1 == 'ok':  
        print('You will love it!')  
    else:  
        print('Okay.')

***"Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. It is a good idea to avoid them when you can."(Downey A, 2015, p.42).***

In this next example, I will explain how to avoid a nested condition using this code example:

#example2  
user\_input = input("Put how much you have to buy a coffee:")  
coffee = float(user\_input)  
price = float(5.00)  
user\_input2 = input('Coffe temperature:')  
temperature = float(user\_input2)  
coffee\_temperature = float(50)  
if coffee >= price and temperature >= coffee\_temperature:  
    print("Give me one , please ")  
else:  
    if coffee < price :  
        print('No,thank you.')  
    elif temperature < coffee\_temperature:  
        print('No,thank you.')  
    else:  
        print('No,thank you.')

The **nested  condition** is on first **else** branch to avoid  it you can write a branch using the "**and**" , "**or**" and "**not**" logical operator:

#example3  
user\_input = input("Put how much you have to buy a coffee:")  
coffee = float(user\_input)  
price = float(5.00)  
user\_input2 = input('Coffe temperature:')  
temperature = float(user\_input2)  
coffee\_temperature = float(50)  
if coffee >= price and temperature >= coffee\_temperature:  
    print("Give me one , please ")  
else:  
    if coffee < price or temperature < coffee\_temperature:  
        print('No,thank you.')

On **example3** I'm using the "**or"** logical operator to make the **nested conditional** into a  **single conditional.**

**References**

**Downey, A. (2015). Think Python. Needham, Massachusetts: Green Tea Press**

**Discussion Assignment U-4**

For a function to receive the information, it is necessary to pass the desired values by argument. However, there are several types of values, and for the function to compile properly and accomplish its purpose, the value type has to be compatible with the function. From what I understand from the book, a precondition is the kind of value expected by the function to perform it; for example, if a function expects an integer to perform an expression but receives a string, an error occurs. Similarly, in a postcondition is expected that, after receiving a value, the function executes and maintains the type of the handled value if the function receives a value and changes it wrong from int to float; for example, the expected result will be incorrect. Finally, if the function is returning the wrong variable, for example, the results used will be false.

There is something wrong with the arguments the function is getting; a precondition is violated:

def delta(a,b,c):  
    d = b\*\*2 - 4\*a\*c  
      
    return d  
a = input('value of a\n')  
b = input('value of b\n')  
c = input('value of c\n')  
delta(a,b,c)

Output:

value of a  
2  
value of b  
-16  
value of c  
-18  
Traceback (most recent call last):  
  File "C:/Users/Rodrigo/Desktop/programas/delta.py", line 10, in <module>  
    delta(a,b,c)  
  File "C:/Users/Rodrigo/Desktop/programas/delta.py", line 2, in delta  
    d = b\*\*2 - 4\*a\*c  
TypeError: unsupported operand type(s) for \*\* or pow(): 'str' and 'int'

Resolution:

def delta(a,b,c):  
    d = b\*\*2 - 4\*a\*c  
    print(d)  
    return d  
a = int(input('value of a\n'))  
b = int(input('value of b\n'))  
c = int(input('value of c\n'))  
delta(a,b,c)

The problem was the argument passing as strings; the expression could not be calculated. Fixed forcing str to int.

There is something wrong with the function; a postcondition is violated.

def bhaskara(a,b,c):  
    delta = b\*\*2 - 4\*a\*c  
    x1 = (-(b) + math.sqrt(delta))/2\*a  
    x2 = (-(b) - math.sqrt(delta))/2\*a  
    print('delta: ', delta)  
    print(x1,x2)  
    return x1,x2  
import math  
a = input('value of a\n')  
a = float(a)  
b = input('value of b\n')  
b = float(b)  
c = input('value of c\n')  
c = float(c)  
bhaskara(a,b,c)

Output:

value of a  
2  
value of b  
-16  
value of c  
-18  
delta:  400.0  
36.0 -4.0

Expected output was 9.0 and -1.0

This error occurs in the mathematical expression

x1 = (-(b) + math.sqrt(delta))/2\*a  
 x2 = (-(b) - math.sqrt(delta))/2\*a

and can be easily fixed by using parenthesis in the (2\*a)

There is something wrong with the return value or the way it is being used.

def delta(a,b,c):  
    d = b\*\*2 - 4\*a\*c  
    return delta  
a = int(input('value of a\n'))  
b = int(input('value of b\n'))  
c = int(input('value of c\n'))  
delta(a,b,c)

In this example the function is returning a undefined variable.

**Discussion Assignment U-5**

# 1

def any\_lowercase1(s):  
     for c in s:  
          if c.islower():  
               return True  
          else:  
               return False

This code is not correct because it checks only the first letter then answers. If the first letter is uppercase it returns falls if it's lowercase it returns true.

>>> print(any\_lowercase1('tEsT'))

True

>>> print(any\_lowercase1('TeSt'))

False

>>>

# 2

def any\_lowercase2(s):  
     for c in s:  
          if 'c'.islower():  
               return 'True'  
          else:  
               return 'False'

This function is not correct as it checks only the string "c" and return always true no matter what the arguments are

>>> print(any\_lowercase2('tEsT'))

True

>>> print(any\_lowercase2('TEsT'))

True

>>> print(any\_lowercase2('TEST'))

True

>>> print(any\_lowercase2('test'))

True

>>>

# 3

def any\_lowercase3(s):  
     for c in s:  
          flag = c.islower()  
     return flag

This function is not correct because it only looks at the last letter in the argument and returns true if it's a lowercase and returns falls if it's an uppercase.

>>> print(any\_lowercase3('TeSt'))

True

>>> print(any\_lowercase3('tESt'))

True

>>> print(any\_lowercase3('tEsT'))

False

>>> print(any\_lowercase3('tesT'))

False

>>> print(any\_lowercase3('TESt'))

True

>>>

# 4

def any\_lowercase4(s):  
     flag = False  
     for c in s:  
          flag = flag or c.islower()  
     return flag

This function works correctly because it looks for any lowercase within the parameter and returns the result based on if it contains any lowercase letters regardless of where they're located.

>>> print(any\_lowercase4('test'))

True

>>> print(any\_lowercase4('Test'))

True

>>> print(any\_lowercase4('TEST'))

False

>>>

# 5

def any\_lowercase5(s):  
     for c in s:  
          if not c.islower():  
               return False  
     return True

This function is not correct because it returns falls all the time except if the entire argument is lowercase.

>>> print(any\_lowercase5('TESt'))

False

>>> print(any\_lowercase5('tESt'))

False

>>> print(any\_lowercase5('teST'))

False

>>> print(any\_lowercase5('test'))

True

>>>

**Discussion Assignment U-6**

Instructions are in **blue**. Outputs are in **green**. My explanations are in **black**.

Describe the difference between **objects** and **values** using the terms “**equivalent**” and “**identical**”. Illustrate the difference using your own examples with Python lists and the “**is**” operator.

An object is "something a variable can refer to. An object has a type and a value." (Downey, 2015). All data in Python is represent by objects. A value is what the object is holding; it's the thing the object represents. Two lists can be equivalent, as in they have the same values, but not identical, as in they don't refer to the same object. If both lists are identical, that means they refer to the same object, and that also means their values are equivalent.

First example:

l1 = [5, 10]

l2 = [5, 10]

print(l1 is l2)

Output:

False

It's false because l1 and l2 do not refer to the same object. They are **equivalent**in values but not **identical**as objects. l1 and l2 are two separate objects with different memory addresses.

Second example:

l1 = l2 = [5, 10]

print(l1 is l2)

Output:

True

Here the program prints True because l1 and l2 refer to the same object. In the second line l2 is set to the same object as l1. l1 and l2 are just the variable names. l1 and l2 are **identical**because they refer to the same object. They are also **equivalent**because they have the same value.

Describe the relationship between **objects**, **references**, and **aliasing**. Again, create your own examples with Python lists.

l3 = ["hello", "there"]

In this example, l3 is a variable name that references the object list (["hello", "there"]). So right now, the list ["hello", "there"] has one reference. If it has more than one reference, then the object is aliased.

For example:

l3 = ["hello", "there", "mate"]

l4 = l3

l5 = l3

Now the list ["hello", "there"] has three references: l3, l4, and l5. That means the object is aliased.

Finally, create your own example of a function that modifies a list passed in as an argument. Describe what your function does in terms of **arguments**, **parameters**, **objects**, and **references**.

def convert\_to\_floats(int\_lst):

for i in range(len(int\_lst)):

int\_lst[i] = float(int\_lst[i])

l1 = [4, 23, 100]

print("Original l1:", l1)

print("After function convert\_to\_floats:")

convert\_to\_floats(l1)

print(l1)

Output:

Original l1: [4, 23, 100]

After function convert\_to\_floats:

[4.0, 23.0, 100.0]

This function convert\_to\_floats takes a list of integers as an argument and converts all values inside the list to float type. For example, the first value of the list was 4. After the function, it became 4.0. The parameter of the function is int\_lst, which is expecting a list of integers.

In this line:

l1 = [4, 23, 100]

It means the variable l1 references to the object [4, 23, 100]. Lists are mutable so the function will work even if the argument is a reference to l1. For example:

def convert\_to\_floats(int\_lst):

for i in range(len(int\_lst)):

int\_lst[i] = float(int\_lst[i])

l1 = [4, 23, 100]

l2 = l1

print("Original list l1:", l1)

print("After function convert\_to\_floats(l2), l1 is:")

convert\_to\_floats(l2) # The function's parameter is changed to l2

print(l1)

Output:

Original list l1: [4, 23, 100]

After function convert\_to\_floats(l2), l1 is:

[4.0, 23.0, 100.0]

Here l2 references to l1 which references to [4, 23, 100]. After the function is done on l2, l1 gets changed as well because l1 and l2 reference the same object.

Reference:

Downey, A. (2015). Think Python. Needham, Massachusetts: Green Tea Press

**Discussion Assignment U-7**

The tuple is similar to list but the major difference between the two is that the list is mutable, meaning the object inside the list can be modified. The tuple is immutable, we cannot add or remove an element in the tuple. Another difference is that tuple is of fixed size in nature while the list is dynamic.

**Example of Tuple in a loop:**

I created a tuple with different strings inside the content and iterate using for loops and printing the result.

tuple\_sample = ('College','University', 'Study', 'Graduation')  
for a in tuple\_sample:     print("printing the word ",a)

#---Output

printing the word  College

printing the word  University

printing the word  Study

printing the word  Graduation

If we want to modify the tuple, in this case adding a string, it cannot be done because it is immutable. We will get an error as a result.

tuple\_sample.append('Computer Science')  
print(tuple\_sample)

#---Output

AttributeError: 'tuple' object has no attribute 'append'

**Example of List in a loop:**

I created a list with different strings as content and iterate using for loops and printing the result.

list\_sample = ['College','University', 'Study', 'Graduation']

for a in list\_sample:

    print("printing the word ",a)

#---Output

printing the word College  
printing the word University  
printing the word Study  
printing the word Graduation

We can successfully modify this list because it is mutable.

list\_sample.append('Computer Science')  
print(list\_sample)

#---Output  
  
['College', 'University', 'Study', 'Graduation', 'Computer Science']

If there is any advantage of tuple over the list, we can say a tuple can be used in dictionaries as a dictionary key

tuple\_dictionary = {('one','two', 'three', ) : 4, ('five', 'six', 'seven'):8}  
print(tuple\_dictionary)

#---Output  
{('one', 'two', 'three'): 4, ('five', 'six', 'seven'): 8}

But not list, we get an error from the example below.

list\_dictionary = {['one','two', 'three',] : 4, ['five', 'six', 'seven']:8}  
print(list\_dictionary)

#---Output  
  
Traceback (most recent call last):  
 File "C:/Users/Ruth/AppData/Local/Programs/Python/Python37-32/bucket5.py", line 1, in <module>  
 list\_dictionary = {['one','two', 'three', ] : 4, ['five', 'six', 'seven']:8}  
TypeError: unhashable type: 'list'

**Zip function:**

In my example, the zip() the function returns the zip objects I created under the tuple\_sample1 & tuple\_sample2. On each iteration, the objects were paired together.

tuple\_sample1=('now', 'People', 'hard', 'soon')  
tuple\_sample2=('College','University', 'Study', 'Graduation')  
  
Tuple\_combine = zip(tuple\_sample2,tuple\_sample1)  
print(tuple(Tuple\_combine))

#---Output  
  
(('College', 'now'), ('University', 'People'), ('Study', 'hard'), ('Graduation', 'soon'))  
Process finished with exit code 0

**Enumerate function:**

On this example, the enumerate() function add a counter to an iterated items from my zip result and returns in the form of enumerated objects starting at zero.

tuple\_sample1=('now', 'People', 'hard', 'soon')  
tuple\_sample2=('College','University', 'Study', 'Graduation')  
  
for i,(tuple\_sample2,tuple\_sample1) in enumerate (zip(tuple\_sample2,tuple\_sample1)):  
 print (i,tuple\_sample2,tuple\_sample1)

#---Output  
  
0 College now  
1 University People  
2 Study hard  
3 Graduation soon  
  
Process finished with exit code 0

**Item function:**

My last example shows items() function applied to the tuple\_dictionary  that I created from my previous example.  It basically returns a view object that displays a list of a given dictionary's (key, value) tuple pair.

tuple\_dictionary = {('one','two', 'three',) : 4, ('five', 'six', 'seven'):8}  
items = tuple\_dictionary.items()  
print ('This are the items inside my tuple\_dictionary: ', items)

#---Output  
  
These are the items inside my tuple\_dictionary: dict\_items([(('one', 'two', 'three'), 4), (('five', 'six', 'seven'), 8)])  
  
Process finished with exit code 0

**Discussion Assignment U-8**

Catching exceptions is useful when the file has any type of problems such as invalid argument, empty argument, or a file with no permission to read it. It's important to catch exceptions just in case the user types in the wrong code related to the file, and the exception catch will display a message telling the user what exactly was wrong, so the user can go back and change it.

**1st example:**

try:

fin = open('test.txt', 'z')

except:

print("ValueError. Second argument must be 'r', 'w', 'a', or 'U'")

**Output:**

ValueError. Second argument must be 'r', 'w', 'a', or 'U'

Description:

In a large production program, I will have a prompt that asks the user to input the filename and a letter. The letter can only be 'r', 'w', 'a', or 'U'. I will have a list explain each function of a letter (e.g. 'r' is read; 'w' is write, etc.). If no input, it will automatically set to 'r', read. This 'except' statement will catch the error for when the user enters an invalid letter.

**2nd example:**

try:

fin = open()

except:

print("TypeError. Please type filename in argument of open()")

**Output:**

TypeError. Please type filename in argument of open()

Description:

In a program, I will first display a list of all available files that can be opened. The user has to choose one file from the list. They cannot leave it empty. This will ensure there will be no errors. If the user still manages to leave it blank, this will catch the error.

**3rd example:**

try:

fin = open('file.txt', 'r')

except:

print("IOError. Permission denied. File cannot be read.")

**Output:**

IOError. Permission denied. File cannot be read.

Description:

In a program, I will list all available files and their respective permissions. Files that have permission that can be read will be placed on top, and the user can choose one of them. If the user chooses a file that cannot be read, then this will catch that error.

# Example 1 (FileNotFoundError)  
try:  
    file = open('test2.txt','r')  
    file.read()  
    file.close()  
      
except Exception as e1:  
    print('The file is not in the working directory or it has been renamed',e1)  
Output:

The file is not in the working directory or it has been renamed [Errno 2] No such file or directory: 'test2.txt'

# Example 2 (UnsupportedOperation: not writable)  
try:       
    file = open('test.txt','r')  
    file.write('Hello World')  
    file.close()  
except Exception as e2:  
    print('Cannont edit file as it may be in use or you dont have the appropriate rights,',e2)

Output:

Cannont edit file as it may be in use or you dont have the appropriate rights, not writable

#Example 3 (FileExistsError)  
import os   
try:  
    a = input('Enter the name to create the file : ')  
    os.mkdir(a)  
except FileExistsError as e3:  
    print('The file already exist with the same name please input another name',e3)

Output:

Enter the name to create the file : test.txt

The file already exist with the same name please input another name [WinError 183] Cannot create a file when that file already exists: 'test.txt'

Catching Exception:

Exception handling can be very significant in debugging and also providing the user with the exact error that have occurred due to an exception. The try: and except: block can help the developer design customize error messages that can help the user to understand the exact cause of the error and the user can easily fix the error by adopting the mentioned implementation in the customize error message. For his own ease the developer can rewrite the error statement for each of the exception, because the builtin exceptional messages sometimes does not clarify the exact cause of the error and it becomes very difficult to remove and debug these errors. Through the try: and except: block I can easily phrase all the exception in my own context so that i will be able to handle each of them effectively and can remove error even for a very complex program.