PYTHON PROGRAMMING FOR ENGINEERS

1ST EDITION

AUTHHORED BY
KHALID MOHAMED

[illegible]

Contents

1. Python data types, values and Identifiers

Variables in python are reserved spaces in memory that can hold some value, so whenever a variable is created some space is reserved in memory to hold the value of that variable. Every variable in python is a specific data type which tells the python interpreter how much memory should be reserved for the variable and what type of values the variable can hold.

In python variables don't need to be declared to assign values to them, so values can be assigned to variable names without explicit declaration of that variable beforehand,the declaration happens when you assign values to them.

1.1 Assigning values to variables

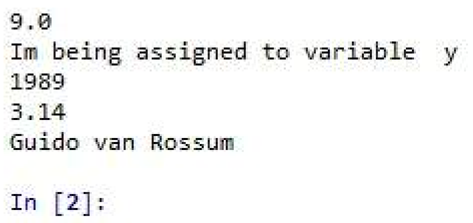
Assignments in python creates references not copies.Variables hold references to objects and do not hold the object itself. Names don't have intrinsic types objects have types, the type of reference is determined by the type of object assigned to the name. Remember the words "Name", "Variable" and "Identifier" refer to the same thing.

A name is created the first time it appears on the left hand side of an assignment. The code below shows some examples.

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 20 15:42:18 2019

author: khalid
"""
x = 3 + 12 / 2
y = "I'm being assigned to the variable y"

```

```python
a = 1989
b = 3.14
c = "Guido van Rossum"

print(x)
print(y)

print(a)
print(b)
print(c)

```

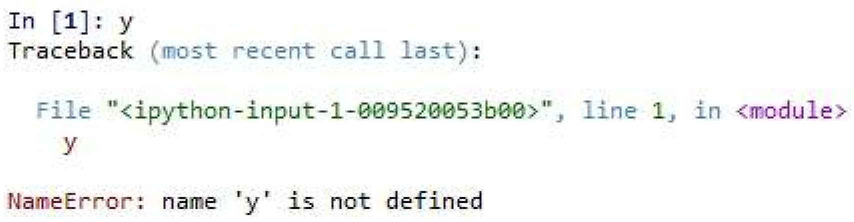
This python code when executed gives the following result.



```
9.0
Im being assigned to variable  y
1989
3.14
Guido van Rossum

In [2]:
```

The garbage collector deletes   reference after any names that are bound to the reference have passed out of scope. When the reference is deleted the object being referenced is deleted.

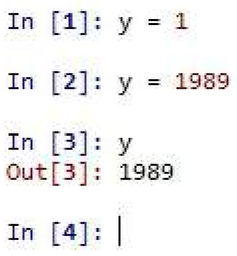
In the case where a non existent name(a variable that doesn’t exist yet) is accessed an error is raised. Take a look at the example.



```
In [1]: y
Traceback (most recent call last):

  File "<ipython-input-1-009520053b00>", line 1, in <module>
    y

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

If a name is accessed after assignment it returns the value the name was last assigned.

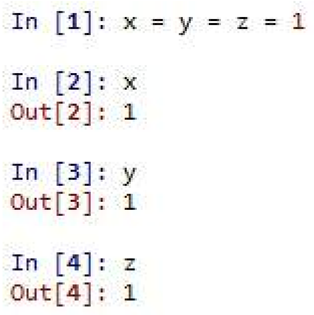
Take a look at the example given below.



```
In [1]: y = 1

In [2]: y = 1989

In [3]: y
Out[3]: 1989

In [4]: |
```

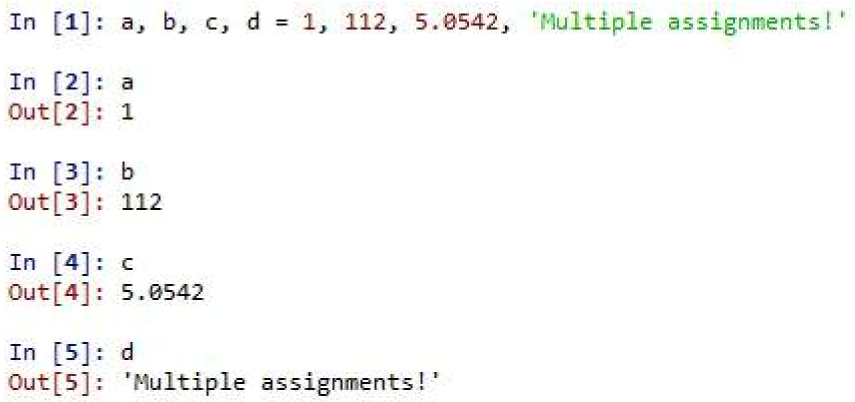
1.2 Multiple assignment

Python allows assigning a single value to several names(variables) simultaneously. For example:



```
In [1]: x = y = z = 1

In [2]: x
Out[2]: 1

In [3]: y
Out[3]: 1

In [4]: z
Out[4]: 1
```

Memory is reserved for the integer object with the value 1, and all three variables refer to the same object. Python also allows assigning multiple objects to multiple variables like the example below.



```
In [1]: a, b, c, d = 1, 112, 5.0542, 'Multiple assignments!'

In [2]: a
Out[2]: 1

In [3]: b
Out[3]: 112

In [4]: c
Out[4]: 5.0542

In [5]: d
Out[5]: 'Multiple assignments!'
```

Here two integer objects are assigned to the names ‘a’ and ‘b’ , a float object is assigned to the name ‘c’ and a string object is assigned to the name ‘d’.

1.3 Naming rules

Names are case sensitive they can contain letters, numbers and underscores but they cant start with a number.

Examples of valid name instances:

```
Var, VAR, Var1223, V123Bob, c020b
```

Examples of invalid instances:

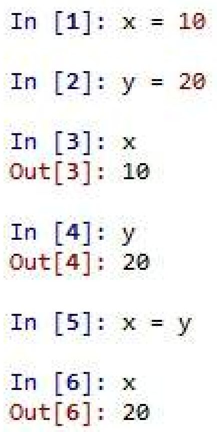
```
1Var,  2a00
```

Reserved words cant be used as a variable name, so the following cant be used as names.

```
and, assert, break, class, continue, def, del, elif, else, except,
finally, for, from, global, if, import, in, is, lambda, not, or,
pass, raise, return, try, while
```

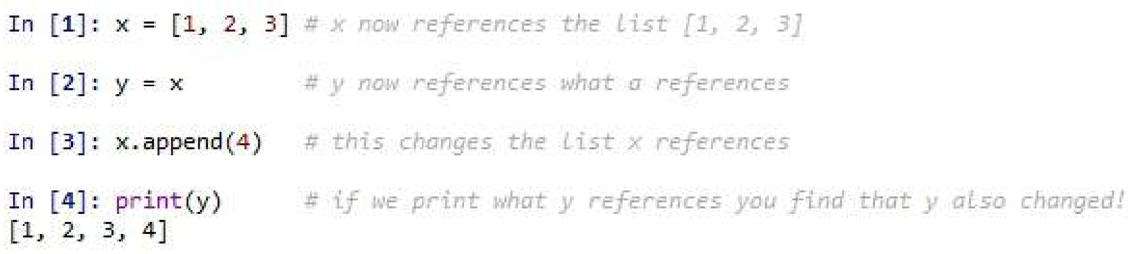
1.4 Understanding reference semantics

This may be different from the language that you used to work with but in python an assignment operation changes references instead of making a new copy of the object. This can be better understood with an example.



```
In [1]: x = 10

In [2]: y = 20

In [3]: x
Out[3]: 10

In [4]: y
Out[4]: 20

In [5]: x = y

In [6]: x
Out[6]: 20
```

Here `x = y` doesn't make a copy of the object `y` references, instead it makes `x` reference the object(integer 20) that y refers to.This can prove to be very useful but we have to be careful for example take a loo at the code below.



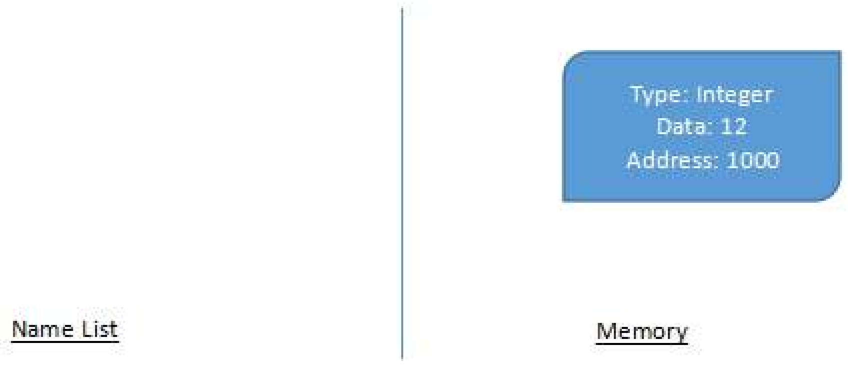
```
In [1]: x = [1, 2, 3] # x now references the list [1, 2, 3]

In [2]: y = x         # y now references what a references

In [3]: x.append(4)   # this changes the list x references

In [4]: print(y)      # if we print what y references you find that y also changed!
[1, 2, 3, 4]
```

If this is confusing it's important to first understand what happens when an assignment operation(like x = 3) is performed.

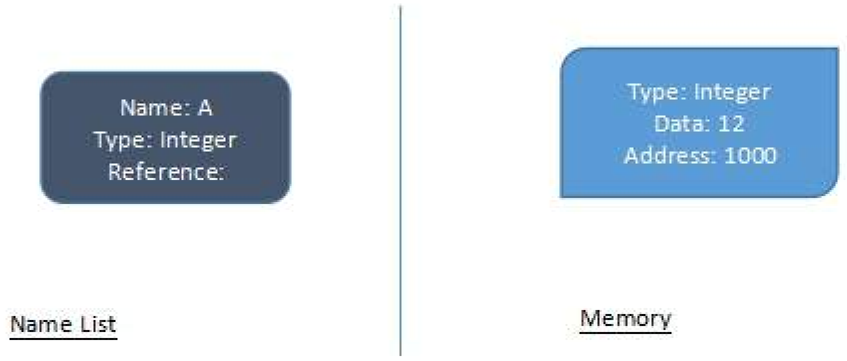
There is a lot that happens in an assignment. For example this is what happens when the below assignment expression is executed

```
A = 12
```

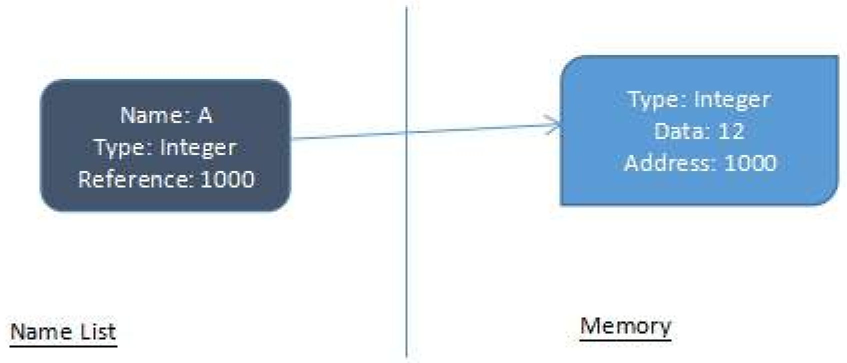
1. First an integer object 12 is created and is stored in memory.



2. A name ‘A’ is created.



3. A reference to the memory location where the integer object 12 is stored at is now assigned to the name ‘A’

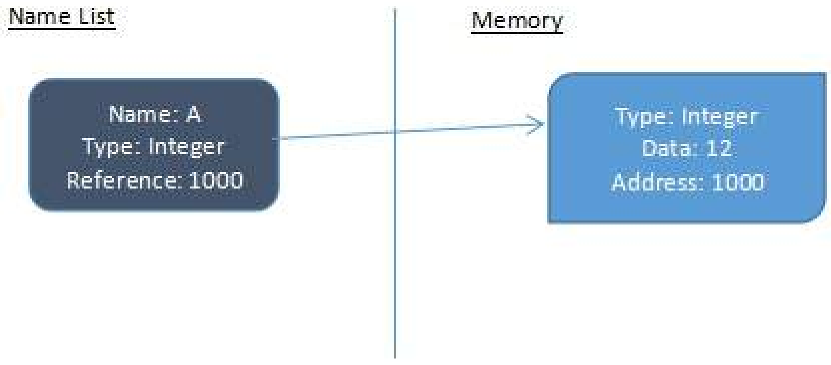


The data 12 that was just created is of type integer. In python the data types integer, float, string and tuple are immutable(which means that object cant be changed). This does not mean that we cant change the value of the variable ‘A’, it can still be reassigned to another integer object i.e what ‘A’ refers to can be changed. For example we could still increment ‘A’ even though its data type is immutable.

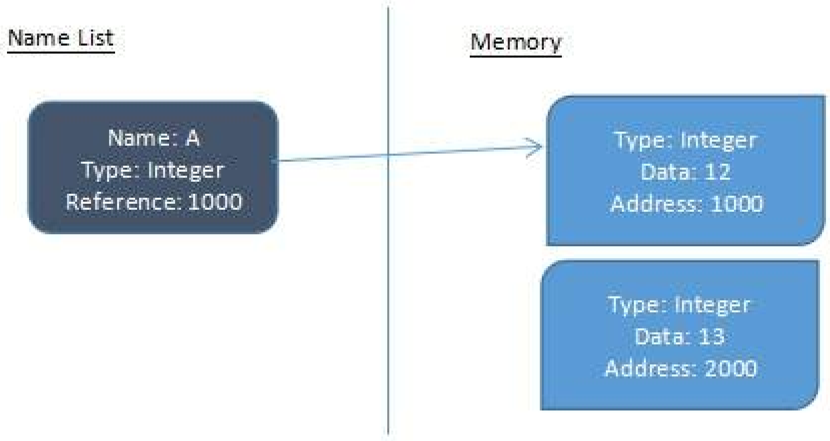
```
In [1]: A = 12

In [2]: A = A + 1

In [3]: A
Out[3]: 13
```

Lets look at what’s really happening when ‘A’ is incremented(`A = A + 1`).

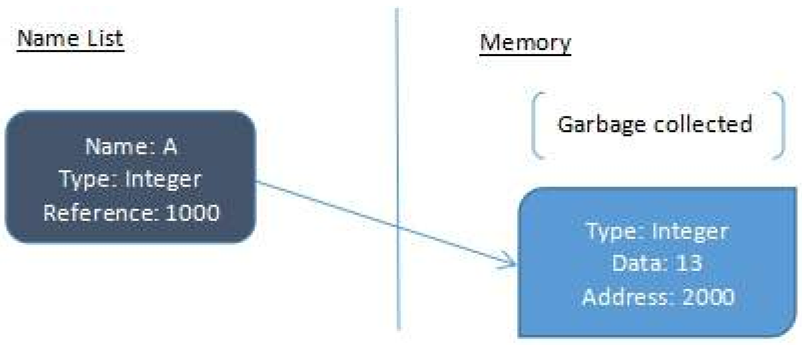
1. The reference of ‘A’ is looked up.
2. The value of the integer object at that reference is retrieved.



3. The 12 + 1 increment operation is performed and the result integer object 13 is stored in a new memory location



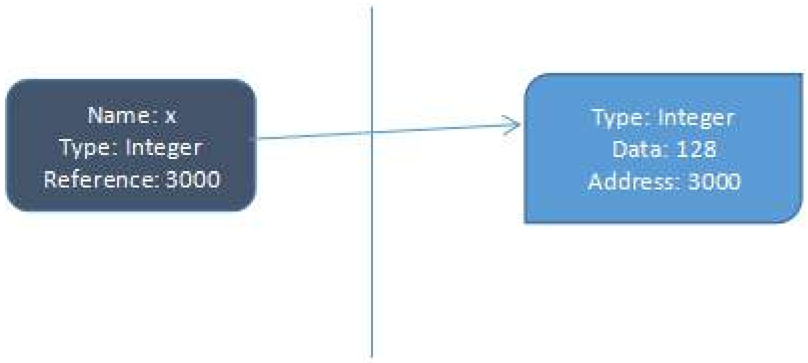
4. The variable ‘A’ is changed to point to this new integer object.while the previuos integer object ‘12’ is deleted by the garbage collector as there are no variables referring to it.


Name List
Memory
Garbage collected
Name: A
Type: Integer
Reference: 1000
Type: Integer
Data: 13
Address: 2000

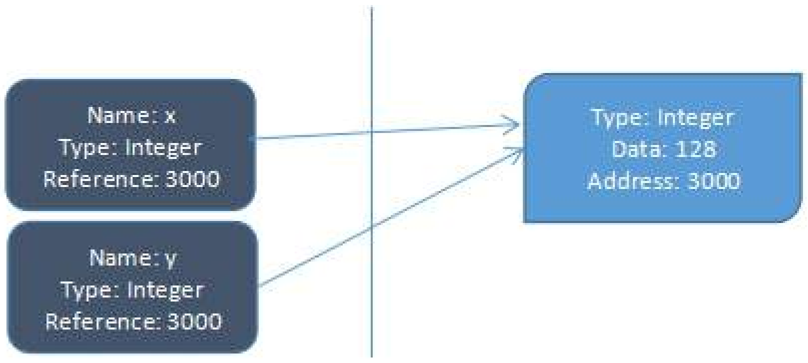
So for pythons basic data types assignments behaves just as you expect them to.

Take another example.

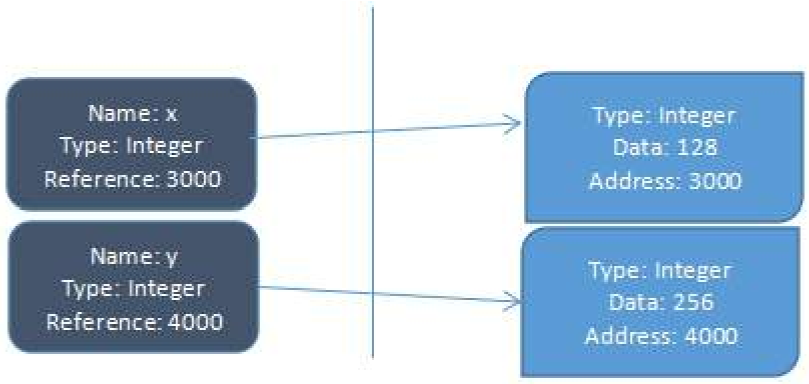
```
In [1]: x = 128
```



```
In [2]: y = x
```



```
In [3]: y = 256
```



```
In [4]: print(y)
256

In [5]: print(x)
128
```

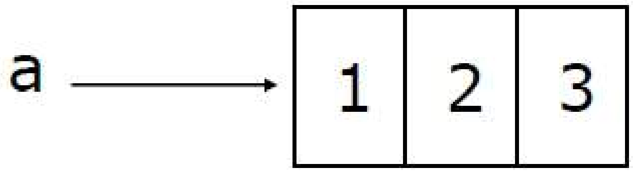
These data types that we've worked with till now are immutable(the values of the objects cant be changed). Assignments in python works differently when its done with mutable objects.

Data types like lists, dictionaries and user defined types are mutable. When mutable data is changed the change happens in place. This means they are not copied to a new memory location every time they are changed. If you type "`y = x`", this changes the value of '`y`' with the value of '`x`'. Remember a new copy is not created to make a change, the change is performed on the same object(unlike for immutable objects) so all variables referring to this object has its value changed as well. We'll go through an example on this just to make it clear.

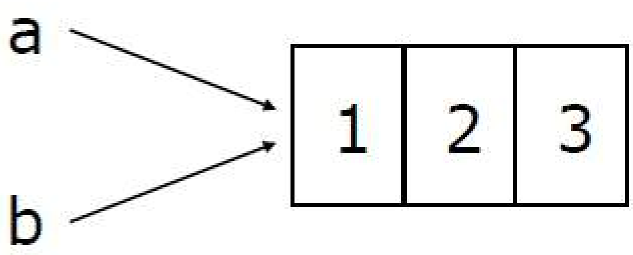
```
In [1]: a = [1,2,3]

In [2]: b = a

In [3]: b.append(4)

In [4]: a
Out[4]: [1, 2, 3, 4]
```

This is what happens when the above code is executed.

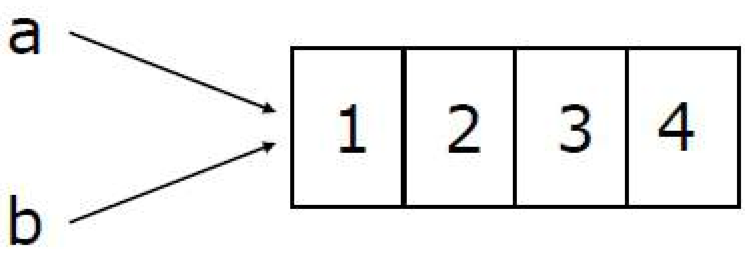
```
In [1]: a = [1,2,3]
```



```
In [2]: b = a
```



```
In [3]: b.append(4)
```



Lets look at each of python standard data types briefly one by one.

1.5 Python's standard data types

If you have experience in other programming languages you would already know that the value that can be stored in variables are of different types. Having different data types is of course a requirement of programmers in general. Each data type defines the different operations possible and it's storage mechanism.

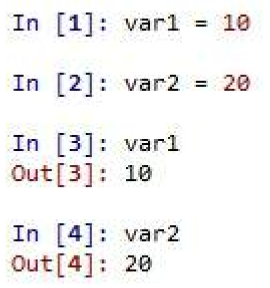
Python has five standard data types:

- Integers
- Strings
- List
- Tuple
- Dictionary

1.5.1 Python Integer(Number)

Integer objects in python are immutable,so the value of the objects can't be changed so a new object is created every time a change is made to the variable

Number objects are created when a value is assigned to them.

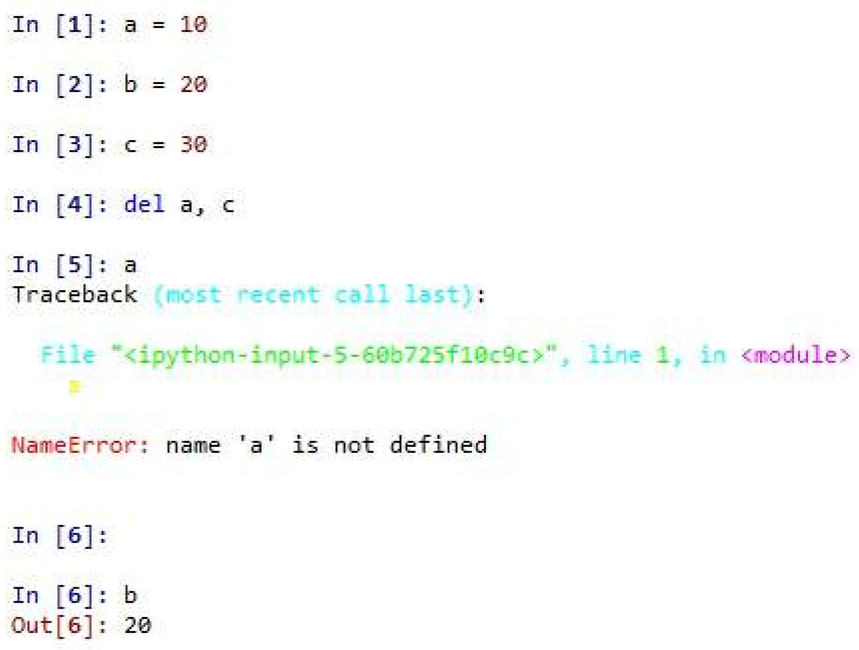


```
In [1]: var1 = 10

In [2]: var2 = 20

In [3]: var1
Out[3]: 10

In [4]: var2
Out[4]: 20
```

These objects can be deleted using the '`del`' keyword. The syntax for the '`del`' command is as follows.

```
del var1[,var2[,var3[....,varN]]]]
```

With '`del`' objects can be deleted one at a time or many at a time. In the following code three variables(a, b, c) are created then '`a`' and '`c`' are deleted using '`del`' command



```
In [1]: a = 10

In [2]: b = 20

In [3]: c = 30

In [4]: del a, c

In [5]: a
Traceback (most recent call last):

  File "<ipython-input-5-60b725f10c9c>", line 1, in <module>
    a

NameError: name 'a' is not defined


In [6]:

In [6]: b
Out[6]: 20
```

There are four different numerical types in Python:

1. int : These are standard signed integers. There is no upper limit on the number of bits that an int object can take up so they can take as much as the memory can offer. This can be tested out by running the following script

```
# A python script to show that we can store

# large numbers in python


a = 20000000000000000000000000000000000001

a = a + 11

print(a)

```

Output:

```
20000000000000000000000000000000000011
```

In python 3 there is only one int type but in python 2.7 there are two int types, one is ‘int’ which is of 32 bit length and the other is ‘long int’ which is the same as the Python 3.x int type(it can store arbitrarily large numbers).

When the following code is run on python 3.x and python 2.7 outputs in each would be as follows

```python
# This program shows that there are two types in
# Python 2.7: int and long int
# and in python 3 there is only one type: int

a = 200
print(type(a))

b = 2000000000000000000000000000000000000000000
print(type(b))

```

Output in python 3.x :

```
<class 'int'>
<class 'int'>
```

Output in python 2.7 :

```
<type 'int'>
<type 'long'>
```

2. float (floating point real values): floats can be used to represent real numbers with a whole number part and a fractional part.
3. complex (complex numbers): represented by the formula a + bJ, where a and b are real numbers(float), and J is the square root of -1 (the result of which is an imaginary number). Complex numbers aren't that often used by python programmers.

Examples of each numeric type:

int	Long int	float	complex
1242	5193488661L	0.0	3.14j
-0009743	-0x19323L	-21.9	45.j
-0x3344	0162L	42.3+e18	4e+26J
1	-05236456727123L	80.2-E12	-.6985+0J

0x958	-47223598762L	-92.	4.76e-7j

Remember ‘long int’ is only in python 2.7 which is the same as ‘int’ in python 3.

1.5.2 Sequence types(Lists, Tuples and Strings)

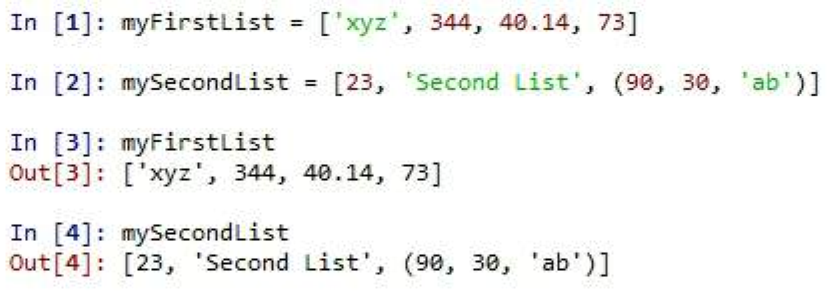
Among the standard types in python there are three sequence types Lists, Tuples and Strings.

List:

A List is a simple ordered sequence of items, these items can be of different types and can also include other sequence types. Lists are mutable so to the value of list objects can be changed in place with out having to create a copy of the object that has the changed value.

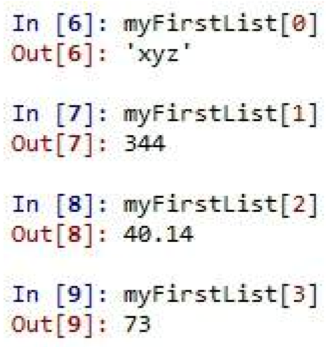
List is the most versatile of the collection types in python. They contain a sequence of items separated by commas(,) and enclosed within square brackets([ ]) This might seem similar to arrays in c, but here items of the list can be of different types.

A list variable in python is created when a list object is assigned to a name as follows.



```
In [1]: myFirstList = ['xyz', 344, 40.14, 73]

In [2]: mySecondList = [23, 'Second List', (90, 30, 'ab')]

In [3]: myFirstList
Out[3]: ['xyz', 344, 40.14, 73]

In [4]: mySecondList
Out[4]: [23, 'Second List', (90, 30, 'ab')]
```

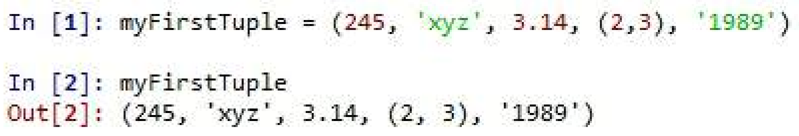
List are defined using square brackets and commas. Individual members of a list can be accessed using the square bracket notation, the same way arrays are accessed in c as follows.


```
In [6]: myFirstList[0]
Out[6]: 'xyz'

In [7]: myFirstList[1]
Out[7]: 344

In [8]: myFirstList[2]
Out[8]: 40.14

In [9]: myFirstList[3]
Out[9]: 73
```

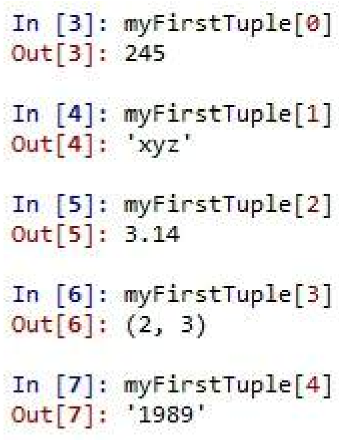
Tuple:

A Tuple is a simple ordered sequence of items, these items can be of different types and can also include sequence types. They are immutable so to change the content of a tuple object a new object with the changed value is created and the tuple variable is made to point to this new object.

Python tuples are similar to lists except that unlike lists they are enclosed within parentheses and tuple objects are immutable. Tuples can be thought of as read only lists.Creating a tuple variable is similar to the creation of a list variable but with parentheses instead of square brackets.

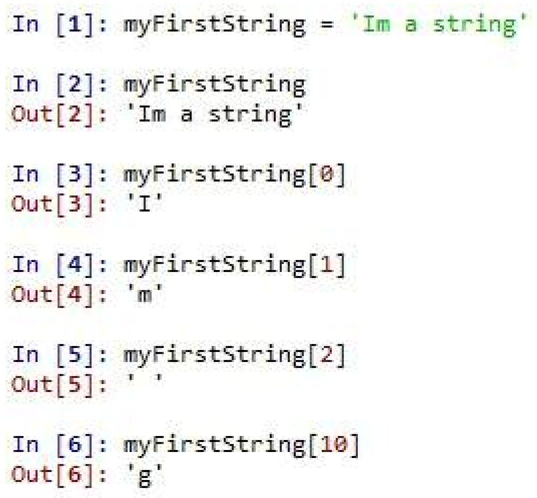

```
In [1]: myFirstTuple = (245, 'xyz', 3.14, (2,3), '1989')

In [2]: myFirstTuple
Out[2]: (245, 'xyz', 3.14, (2, 3), '1989')
```

And just like lists, individual members of a tuple can be accessed using the square bracket notation, the same way arrays are accessed in c .


```
In [3]: myFirstTuple[0]
Out[3]: 245

In [4]: myFirstTuple[1]
Out[4]: 'xyz'

In [5]: myFirstTuple[2]
Out[5]: 3.14

In [6]: myFirstTuple[3]
Out[6]: (2, 3)

In [7]: myFirstTuple[4]
Out[7]: '1989'
```

String:

Strings in python are a continuous set of characters that are enclosed within quotation marks. Python allows using either single quotes(‘’) or double quotes(“”).Strings are immutable like tuples, so the value of a string object cant be changed in place to change the value of a string object a new object is created with the changed value. Declaration of a string and accessing individual elements of a string are the same as in lists and tuples. Examples of string declaration and access are given below.



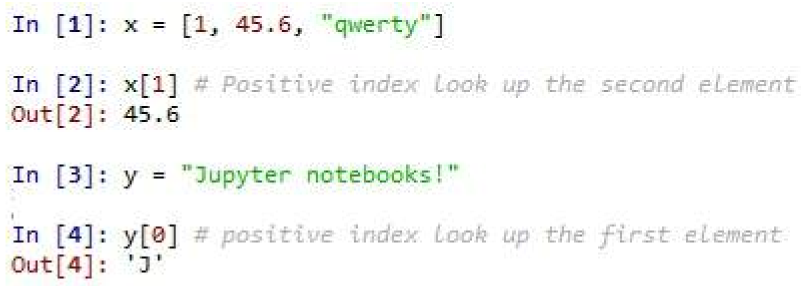
```
In [1]: myFirstString = 'Im a string'

In [2]: myFirstString
Out[2]: 'Im a string'

In [3]: myFirstString[0]
Out[3]: 'I'

In [4]: myFirstString[1]
Out[4]: 'm'

In [5]: myFirstString[2]
Out[5]: ' '

In [6]: myFirstString[10]
Out[6]: 'g'
```

1.5.3 Operations on sequence types

The operations shown here can be applied to all sequence types. Examples are shown for clarity on how they work. Most examples show the operation performed only on one or two of the sequence types but remember that these operations can be performed on all of the sequence types.

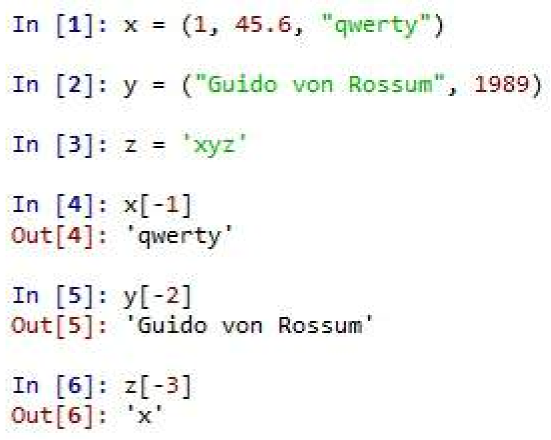
Positive and Negative index look up:

Positive index look up counts from the left and starts the count from 0.



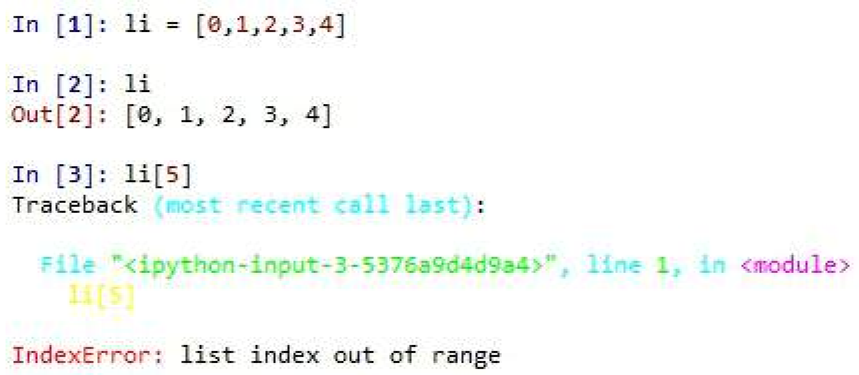
```
In [1]: x = [1, 45.6, "qwerty"]

In [2]: x[1] # Positive index look up the second element
Out[2]: 45.6

In [3]: y = "Jupyter notebooks!"

In [4]: y[0] # positive index look up the first element
Out[4]: 'J'
```

Negative index look up counts from the left and starts the count from -1.



```
In [1]: x = (1, 45.6, "qwerty")

In [2]: y = ("Guido von Rossum", 1989)

In [3]: z = 'xyz'

In [4]: x[-1]
Out[4]: 'qwerty'

In [5]: y[-2]
Out[5]: 'Guido von Rossum'

In [6]: z[-3]
Out[6]: 'x'
```

When an element that is out of range is accessed the python interpreter gives an index error, an example is shown below.



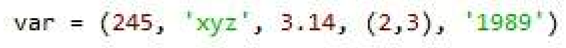
```
In [1]: li = [0,1,2,3,4]

In [2]: li
Out[2]: [0, 1, 2, 3, 4]

In [3]: li[5]
Traceback (most recent call last):

  File "<ipython-input-3-5376a9d4d9a4>", line 1, in <module>
    li[5]

IndexError: list index out of range
```

Slicing:

Slicing can be used to return a copy subset of the sequence typed object. The general form of a slicing operation is as follows

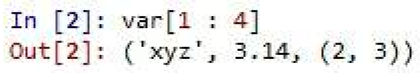
```
<Variable Name>[x : y]
```

Where ‘x’ and ‘y’ are the first and second indices of the slicing operation. The container is copied from index ‘x’ and the copying is discontinued from index y. Lets go through some examples so its made clear how the slicing operation actually works. We’ll work with the list ‘li’ given below.



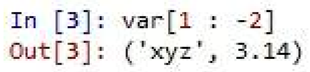
```
var = (245, 'xyz', 3.14, (2,3), '1989')
```

The below slicing operation on ‘var’ returns a copy of a subset of ‘var’. Copying starts from the first index(1) and discontinues from the second index(4).



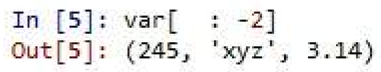
```
In [2]: var[1 : 4]
Out[2]: ('xyz', 3.14, (2, 3))
```

Negative indices can also be used when slicing like below.



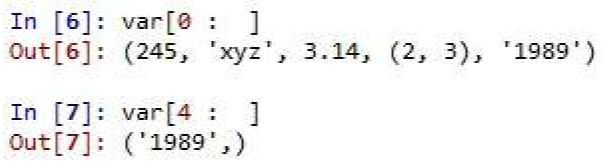
```
In [3]: var[1 : -2]
Out[3]: ('xyz', 3.14)
```

The first index can be left blank to return a copy of a subset of the that starts copying from the first element of the container.



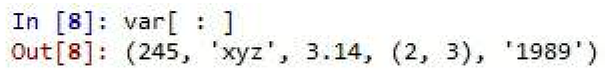
```
In [5]: var[  : -2]
Out[5]: (245, 'xyz', 3.14)
```

The second index can be left blank to start copying from the first index and continue copying till the last element.



```
In [6]: var[0 :  ]
Out[6]: (245, 'xyz', 3.14, (2, 3), '1989')

In [7]: var[4 :  ]
Out[7]: ('1989',)
```

To return a copy of the entire container both indices can be left blank



```
In [8]: var[ : ]
Out[8]: (245, 'xyz', 3.14, (2, 3), '1989')
```

From the above examples it could be understood that for any sequence typed variable say ‘A’

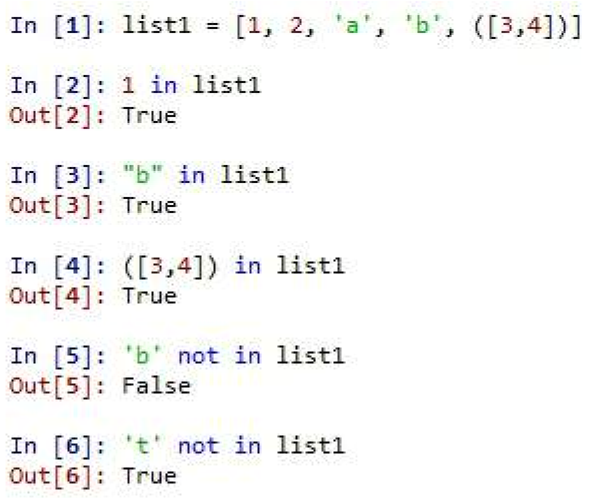
```
A[ : ] = A[0 : ] != A[0 : -1]
```

Since lists are mutable sequence types, a change in the objects value happens in place without creating a new copy of the object, but wen the slicing operator is used a new object is created and returned. Take a look at the example given below.

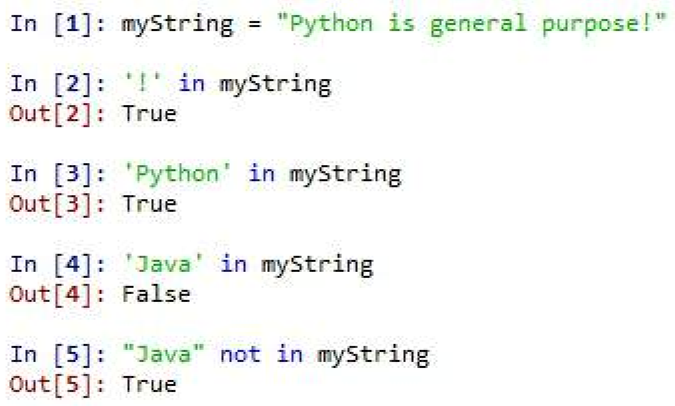
```
list1 = [1, 2, 3, 4]

list2 = list1         # 2 names referring to the same object
                      # When one name is changed the other changes as well

list2 = list1[ : ]     # Name list2 refers to a new object which is a
                      # copy of a subset list1

```

The “in” operator:

The “in” operator can be used as a Boolean test to check whether some value is inside a container object.


```
In [1]: list1 = [1, 2, 'a', 'b', ([3,4])]

In [2]: 1 in list1
Out[2]: True

In [3]: "b" in list1
Out[3]: True

In [4]: ([3,4]) in list1
Out[4]: True

In [5]: 'b' not in list1
Out[5]: False

In [6]: 't' not in list1
Out[6]: True
```

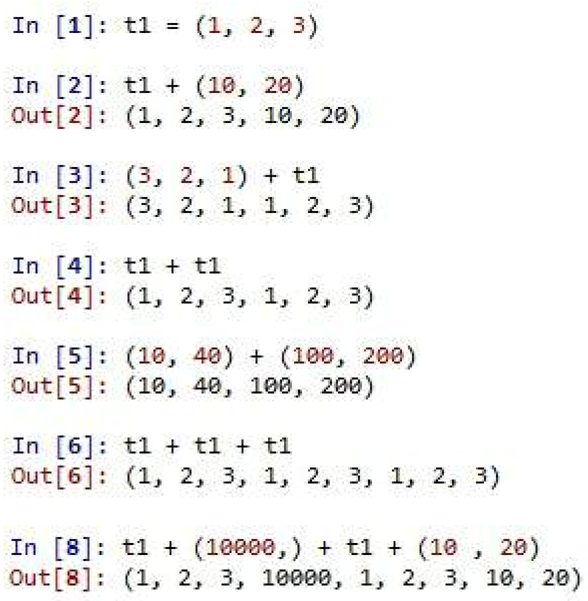
For strings the “in” operator can be used to test whether the given sub string exists in the string like in the examples given below.


```
In [1]: myString = "Python is general purpose!"

In [2]: '!' in myString
Out[2]: True

In [3]: 'Python' in myString
Out[3]: True

In [4]: 'Java' in myString
Out[4]: False

In [5]: "Java" not in myString
Out[5]: True
```

The ‘+’ operator:

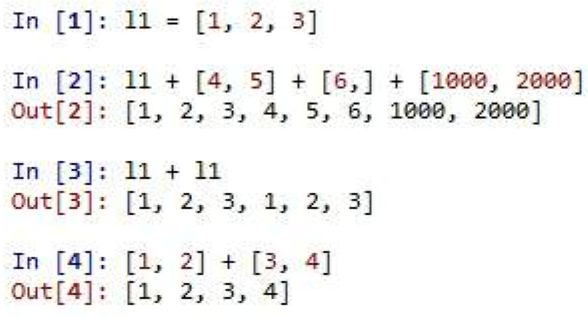
This operator is used to concatenate two tuples, strings or lists together the result obtained after use is the new object obtained after concatenation. Concatenation can not be performed with different container types. Here are some examples of the use of the ‘+’ operator with tuples, lists and strings.

With tuples:



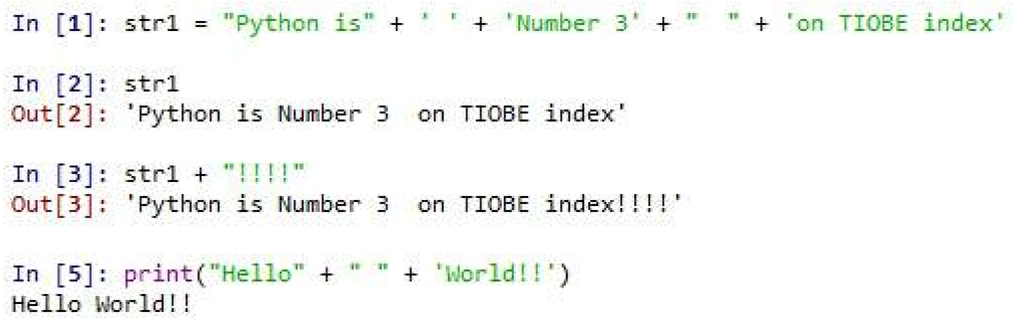
```
In [1]: t1 = (1, 2, 3)

In [2]: t1 + (10, 20)
Out[2]: (1, 2, 3, 10, 20)

In [3]: (3, 2, 1) + t1
Out[3]: (3, 2, 1, 1, 2, 3)

In [4]: t1 + t1
Out[4]: (1, 2, 3, 1, 2, 3)

In [5]: (10, 40) + (100, 200)
Out[5]: (10, 40, 100, 200)

In [6]: t1 + t1 + t1
Out[6]: (1, 2, 3, 1, 2, 3, 1, 2, 3)

In [8]: t1 + (10000,) + t1 + (10 , 20)
Out[8]: (1, 2, 3, 10000, 1, 2, 3, 10, 20)
```

With lists:(A new object is returned as the result)



```
In [1]: l1 = [1, 2, 3]

In [2]: l1 + [4, 5] + [6,] + [1000, 2000]
Out[2]: [1, 2, 3, 4, 5, 6, 1000, 2000]

In [3]: l1 + l1
Out[3]: [1, 2, 3, 1, 2, 3]

In [4]: [1, 2] + [3, 4]
Out[4]: [1, 2, 3, 4]
```

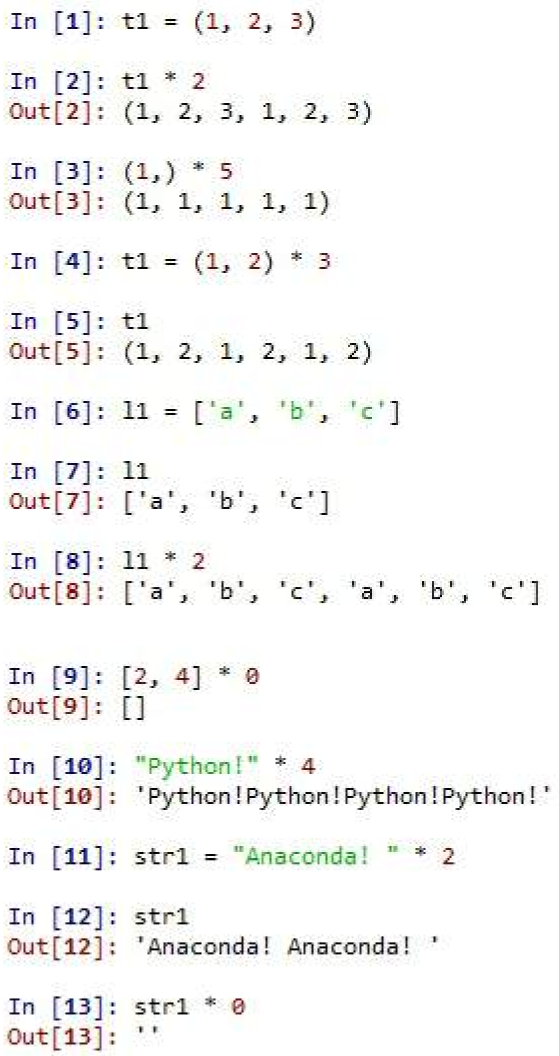
With strings:



```
In [1]: str1 = "Python is" + ' ' + 'Number 3' + "  " + 'on TIOBE index'

In [2]: str1
Out[2]: 'Python is Number 3  on TIOBE index'

In [3]: str1 + "!!!!"
Out[3]: 'Python is Number 3  on TIOBE index!!!!'

In [5]: print("Hello" + " " + 'World!!')
Hello World!!
```

The '*' operator:

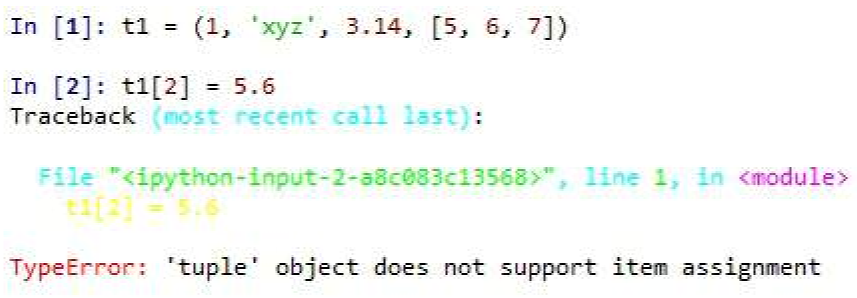
Produces a new tuple, list or string that has the original value repeated a number of times, the number of repeats is given as an operand to this operator and the other operand is the sequence itself. Lets take a look at some examples.



```
In [1]: t1 = (1, 2, 3)

In [2]: t1 * 2
Out[2]: (1, 2, 3, 1, 2, 3)

In [3]: (1,) * 5
Out[3]: (1, 1, 1, 1, 1)

In [4]: t1 = (1, 2) * 3

In [5]: t1
Out[5]: (1, 2, 1, 2, 1, 2)

In [6]: l1 = ['a', 'b', 'c']

In [7]: l1
Out[7]: ['a', 'b', 'c']

In [8]: l1 * 2
Out[8]: ['a', 'b', 'c', 'a', 'b', 'c']


In [9]: [2, 4] * 0
Out[9]: []

In [10]: "Python!" * 4
Out[10]: 'Python!Python!Python!Python!'

In [11]: str1 = "Anaconda! " * 2

In [12]: str1
Out[12]: 'Anaconda! Anaconda! '

In [13]: str1 * 0
Out[13]: ''
```

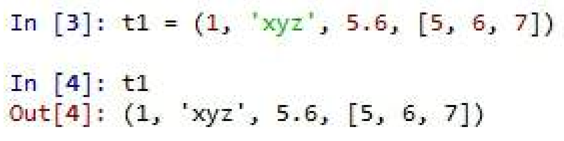
1.5.4 Lists and Tuples (Mutability vs immutability)

Tuple is an immutable type so the value/content of a tuple object can't be changed the only way to change the value of the a tuple variable is to create a new tuple object with the changed value and assign the new reference to the tuple variable. The value/content of a list can be changed in place as they are mutable, but when a value of a mutable object that is referred to by more than one variable is changed the value is changed for all variables referring to that object.Take a look at the example below.



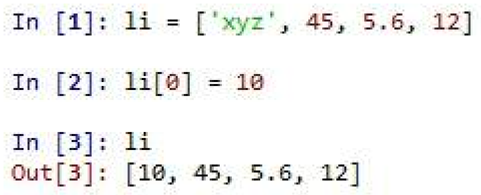
```
In [1]: t1 = (1, 'xyz', 3.14, [5, 6, 7])

In [2]: t1[2] = 5.6
Traceback (most recent call last):

  File "<ipython-input-2-a8c083c13568>", line 1, in <module>
    t1[2] = 5.6

TypeError: 'tuple' object does not support item assignment
```

Tuples can’t be changed but instead make a fresh tuple and assign the reference of it to an already existing name like below.



```
In [3]: t1 = (1, 'xyz', 5.6, [5, 6, 7])

In [4]: t1
Out[4]: (1, 'xyz', 5.6, [5, 6, 7])
```

Unlike tuples, lists are mutable and so they can be changed in place without having to change the reference to a new object



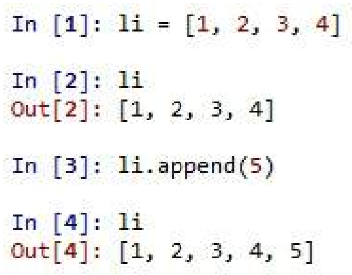
```
In [1]: li = ['xyz', 45, 5.6, 12]

In [2]: li[0] = 10

In [3]: li
Out[3]: [10, 45, 5.6, 12]
```

Like in the above example lists can be changed in place. Name “li” still points to the same memory reference when the changed is done. The mutability of lists means that they aren’t as fast as tuples.

Since lists are mutable there are some operations that are applicable only to lists and not tuples. lets go through those operations one by one. The list class has some in built methods that can be used to change the value of the list object or return a value based on the current state of the list object. This is also our first exposure to pythons method syntax.

append( ) :

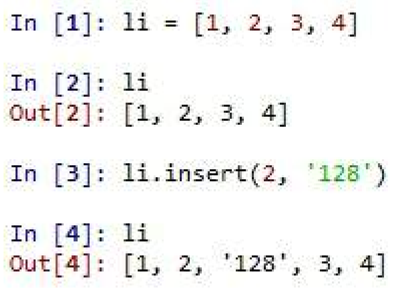
The list class has an append method that can be used to insert a new list element at the end of the list. In the example below a new name “li” is assigned a reference to a list object, then the append method of the list class is called to add the element passed as an argument to method at the end of the list object.



```
In [1]: li = [1, 2, 3, 4]

In [2]: li
Out[2]: [1, 2, 3, 4]

In [3]: li.append(5)

In [4]: li
Out[4]: [1, 2, 3, 4, 5]
```

insert( ) :

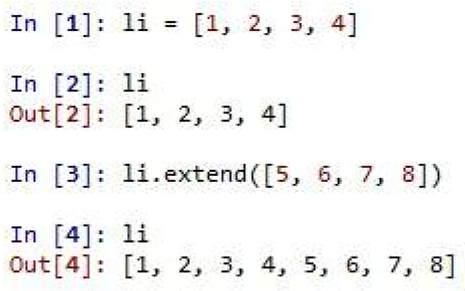
The list class has an insert method that can be used to insert an new element at any location the list. The insert method takes two parameter's, first one is the location in the list to which the the new element is to be inserted and the second is the element itself. In the below example a new element is inserted at location 2 in the list.



```
In [1]: li = [1, 2, 3, 4]

In [2]: li
Out[2]: [1, 2, 3, 4]

In [3]: li.insert(2, '128')

In [4]: li
Out[4]: [1, 2, '128', 3, 4]
```

extend( ) :

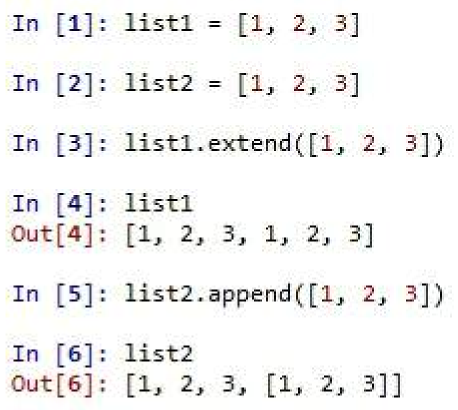
Some times instead of appending single elements at a time we may want to append a list of elements to a list, the list class has a method to do that, the extend method. The extend method takes a list as an argument and appends that list to the list object to which it was called upon. An example is given below.



```
In [1]: li = [1, 2, 3, 4]

In [2]: li
Out[2]: [1, 2, 3, 4]

In [3]: li.extend([5, 6, 7, 8])

In [4]: li
Out[4]: [1, 2, 3, 4, 5, 6, 7, 8]
```

It may look like the extend method does exactly what the ‘+’ operator does for sequence types but the difference is that the ‘+’ operator creates a new list with a new memory reference, and extend operates on the list object it’s called upon in place.

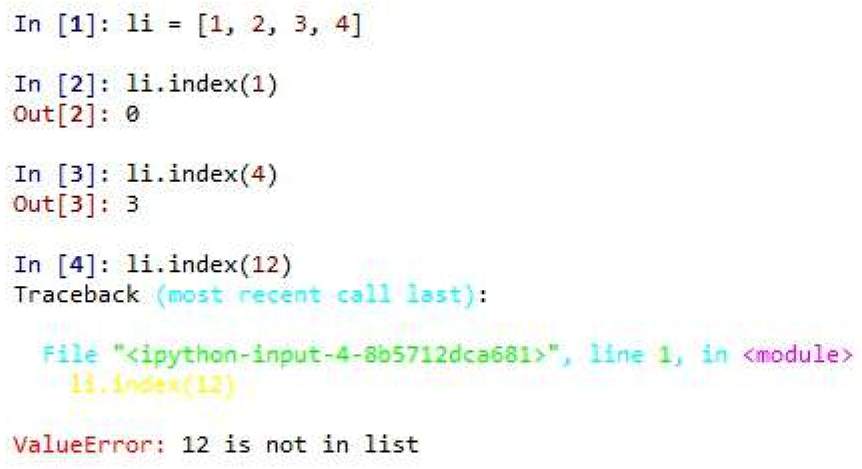
Extend method takes a list as an argument and append method takes a single object as an argument. Take a look at the example below for more clarity.



```
In [1]: list1 = [1, 2, 3]

In [2]: list2 = [1, 2, 3]

In [3]: list1.extend([1, 2, 3])

In [4]: list1
Out[4]: [1, 2, 3, 1, 2, 3]

In [5]: list2.append([1, 2, 3])

In [6]: list2
Out[6]: [1, 2, 3, [1, 2, 3]]
```

index( ) :

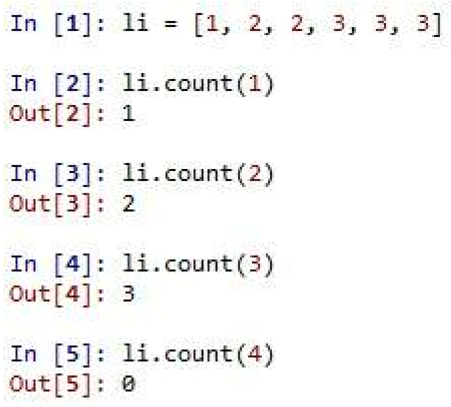
List class has an index method that returns the index of the first occurrence of the value passed as an argument to the method. If the value passed does not exist in the list then a “value error” is raised by the python interpreter. Examples are shown below.



```
In [1]: li = [1, 2, 3, 4]

In [2]: li.index(1)
Out[2]: 0

In [3]: li.index(4)
Out[3]: 3

In [4]: li.index(12)
Traceback (most recent call last):

  File "<ipython-input-4-8b5712dca681>", line 1, in <module>
    li.index(12)

ValueError: 12 is not in list
```

count( ) :

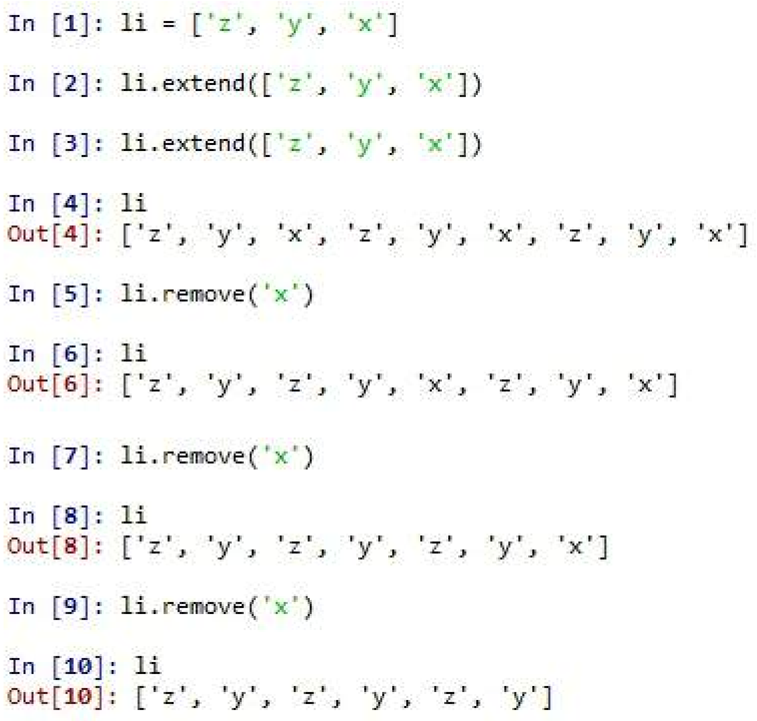
When a value is passed as an argument to the count method it returns the number of occurrences of that value in the list object on which it was called upon.



```
In [1]: li = [1, 2, 2, 3, 3, 3]

In [2]: li.count(1)
Out[2]: 1

In [3]: li.count(2)
Out[3]: 2

In [4]: li.count(3)
Out[4]: 3

In [5]: li.count(4)
Out[5]: 0
```

remove( ) :

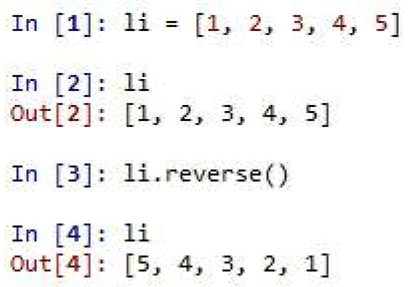
The remove method of the list class can be used to remove the first occurrence of the value passed as the argument to the method.examples are shown below.



```
In [1]: li = ['z', 'y', 'x']

In [2]: li.extend(['z', 'y', 'x'])

In [3]: li.extend(['z', 'y', 'x'])

In [4]: li
Out[4]: ['z', 'y', 'x', 'z', 'y', 'x', 'z', 'y', 'x']

In [5]: li.remove('x')

In [6]: li
Out[6]: ['z', 'y', 'z', 'y', 'x', 'z', 'y', 'x']

In [7]: li.remove('x')

In [8]: li
Out[8]: ['z', 'y', 'z', 'y', 'z', 'y', 'x']

In [9]: li.remove('x')

In [10]: li
Out[10]: ['z', 'y', 'z', 'y', 'z', 'y']
```

reverse( ) :

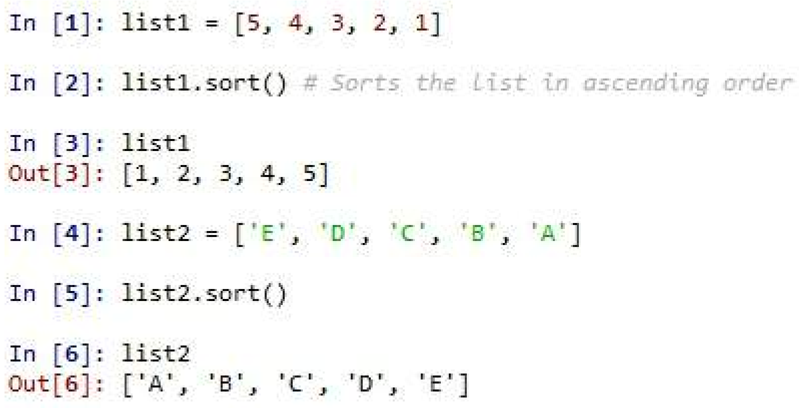
When the reverse method of the list class is called on a list object it reverses the list object in place in the memory. In the below example when the reverse method is called for a list object, the order of its content are reversed.



```
In [1]: li = [1, 2, 3, 4, 5]

In [2]: li
Out[2]: [1, 2, 3, 4, 5]

In [3]: li.reverse()

In [4]: li
Out[4]: [5, 4, 3, 2, 1]
```

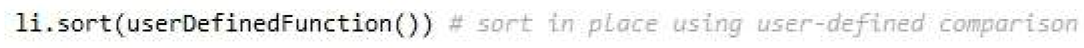
sort( ) :

The sort method of the list class can be used to sort the contents of the list in place in memory



```
In [1]: list1 = [5, 4, 3, 2, 1]

In [2]: list1.sort() # Sorts the list in ascending order

In [3]: list1
Out[3]: [1, 2, 3, 4, 5]

In [4]: list2 = ['E', 'D', 'C', 'B', 'A']

In [5]: list2.sort()

In [6]: list2
Out[6]: ['A', 'B', 'C', 'D', 'E']
```

The sort function of the list class can be passed a user defined function with it’s own approach to sorting a list.

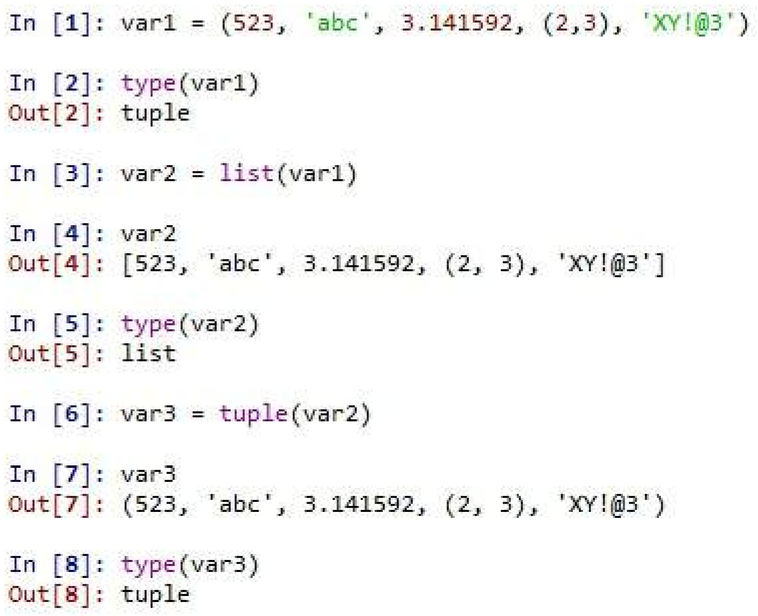


```
li.sort(userDefinedFunction()) # sort in place using user-defined comparison
```

Although lists are slower than tuples, they are much more powerful than tuples and are the most versatile types in python. Lists unlike tuples are mutable, they can be modified and they have a lot of operations that can be performed on them that make them very handy while programming. Tuples are immutable, they can’t be modified and they have much lesser features than lists, although they are faster than lists.

list( ) and tuple( ) :

Python provides the list method and tuple method to convert between lists and tuples. The list method takes a tuple and returns a new instance of a list object. The tuple method takes a list and returns a new instance of a tuple object. Some examples are shown below.



```
In [1]: var1 = (523, 'abc', 3.141592, (2,3), 'XY!@3')

In [2]: type(var1)
Out[2]: tuple

In [3]: var2 = list(var1)

In [4]: var2
Out[4]: [523, 'abc', 3.141592, (2, 3), 'XY!@3']

In [5]: type(var2)
Out[5]: list

In [6]: var3 = tuple(var2)

In [7]: var3
Out[7]: (523, 'abc', 3.141592, (2, 3), 'XY!@3')

In [8]: type(var3)
Out[8]: tuple
```

1.5.5 Dictionary

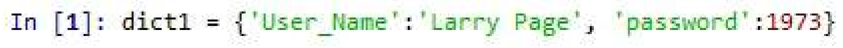
Python dictionaries are like a hash table,They store a mapping between key's and value's. They consist of key value pairs, Where the key can be any of python's immutable types, although they are usually numbers or strings. Values on the other hand can be any python object. Dictionaries are enclosed within curly braces({ }) and values can be assigned and accessed using square braces([ ]). A single dictionary can store values of different types.

Dictionaries have many operations that can be performed on them. It's possible to define, view, look up and delete the key value pairs in a dictionary object.

Declaration :

Dictionaries are enclosed in curly braces({ }), each key value pair is separated by commas(,) and a colon(:) between the key and value of each key and value pair.

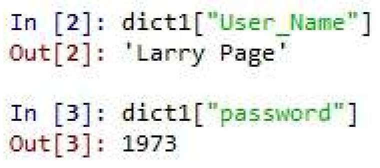
The keys can be any immutable type in in python and values can be any standard or user defined type in python. Dictionary variables are declared when a dictionary object reference is assigned to a name as in the example below.



```
In [1]: dict1 = {'User_Name':'Larry Page', 'password':1973}
```

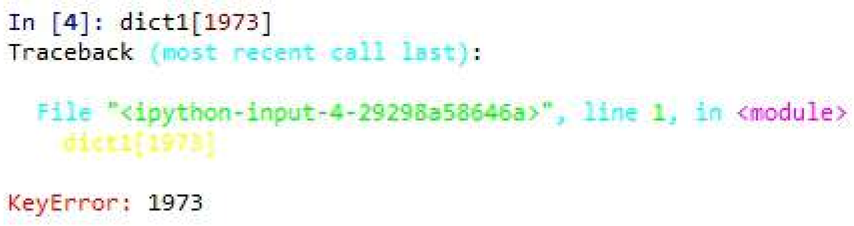
Accessing Values :

Dictionary values can be accessed using the keys enclosed in square braces like how they are accessed in tuples and lists except in the case of dictionaries the subscript is the key.Some examples are shown below.



```
In [2]: dict1["User_Name"]
Out[2]: 'Larry Page'

In [3]: dict1["password"]
Out[3]: 1973
```

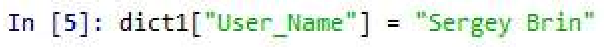
Python doesn’t support look ups with the values if a key that doesn’t exist in th e dictionary is looked up a “key error” is raised.



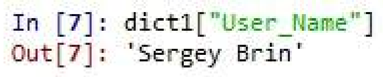
```
In [4]: dict1[1973]
Traceback (most recent call last):

  File "<ipython-input-4-29298a58646a>", line 1, in <module>
    dict1[1973]

KeyError: 1973
```

Changing values :

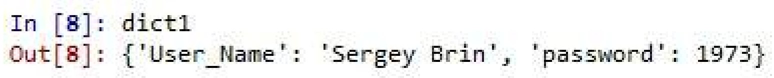
Dictionaries are mutable types so the values of the dictionary objects are changed in place in memory. A value corresponding to a dictionary key can be changed as follows



```
In [5]: dict1["User_Name"] = "Sergey Brin"
```



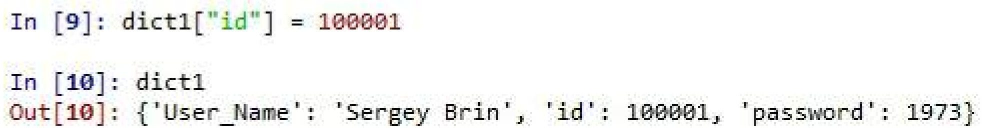
```
In [7]: dict1["User_Name"]
Out[7]: 'Sergey Brin'
```



```
In [8]: dict1
Out[8]: {'User_Name': 'Sergey Brin', 'password': 1973}
```

Inserting a new key-value pair :

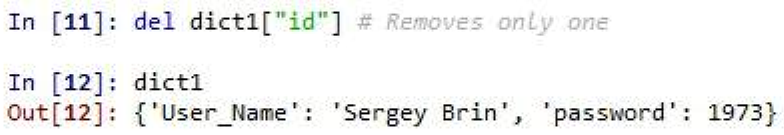
A new key-value pair can be inserted into the dictionary by assigning a value to a dictionary key that doesn't exist yet. An example is shown below.



```
In [9]: dict1["id"] = 100001

In [10]: dict1
Out[10]: {'User_Name': 'Sergey Brin', 'id': 100001, 'password': 1973}
```

`del` command :

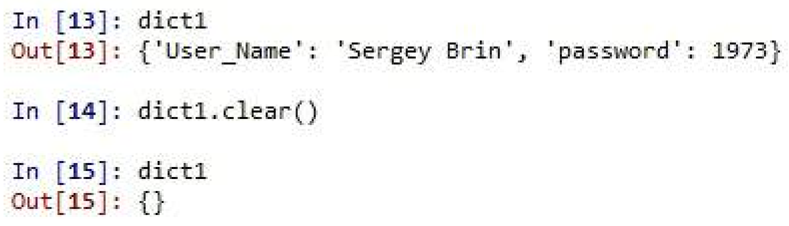
The `del` command is used to remove a single key value pair from the dictionary object, only the specified key value pair is deleted the rest of the dictionary is left intact. An example of using the `del` command is shown below.



```
In [11]: del dict1["id"] # Removes only one

In [12]: dict1
Out[12]: {'User_Name': 'Sergey Brin', 'password': 1973}
```

clear( ) :

The dictionary class has a "clear" method which when called for a dictionary object deletes all the key-value pairs in that dictionary object. It is the same as applying the del command to all keys in the dictionary

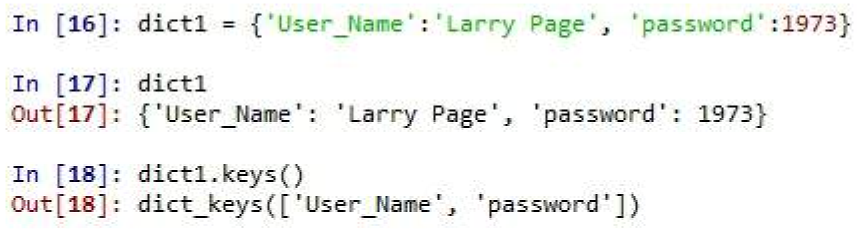


```
In [13]: dict1
Out[13]: {'User_Name': 'Sergey Brin', 'password': 1973}

In [14]: dict1.clear()

In [15]: dict1
Out[15]: {}
```

Let's discuss some of the other methods that the python dictionary class offers.

keys( ) :

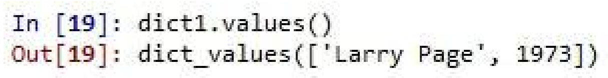
The “keys” method of python’s dictionary class when called for a method returns a list containing all the keys of that dictionary object. An example is shown below.



```
In [16]: dict1 = {'User_Name':'Larry Page', 'password':1973}

In [17]: dict1
Out[17]: {'User_Name': 'Larry Page', 'password': 1973}

In [18]: dict1.keys()
Out[18]: dict_keys(['User_Name', 'password'])
```

values( ) :

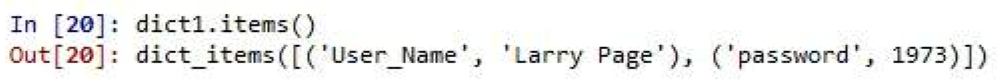
The “values” method of the dictionary class when called for a dictionary object returns a list containing all the values in the dictionary.



```
In [19]: dict1.values()
Out[19]: dict_values(['Larry Page', 1973])
```

items( ) :

Python’s dictionary method has a items class which when called on an object returns a list of all the key value pairs as tuples. An example is shown below.



```
In [20]: dict1.items()
Out[20]: dict_items([('User_Name', 'Larry Page'), ('password', 1973)])
```

1.6 Data type conversion

There may come situations where it’s need to convert between python’s built in data type, to convert between built in data types the type nae can be used as a function.

There are several built in functions to perform a conversion from one type to another type, these functions return a fresh object in memory with the converted value.

Function	Description
`int(x [,base])`	Converts x to an integer. base specifies the base if x is a string.
`long(x [,base] )`	Converts x to a long integer. base specifies the base if x is a string.
`float(x)`	Converts x to a floating-point number.
`complex(real [,imag])`	Creates a complex number.
`str(x)`	Converts object x to a string representation.
`repr(x)`	Converts object x to an expression string.
`eval(str)`	Evaluates a string and returns an object.
`tuple(s)`	Converts s to a tuple.
`list(s)`	Converts s to a list.
`set(s)`	Converts s to a set.
`dict(d)`	Creates a dictionary. d must be a sequence of (key,value) tuples.
`frozenset(s)`	Converts s to a frozen set.
`chr(x)`	Converts an integer to a character.
`unichr(x)`	Converts an integer to a Unicode character.
`ord(x)`	Converts a single character to its integer value.
`hex(x)`	Converts an integer to a hexadecimal string.
`oct(x)`	Converts an integer to an octal string.

In the next chapter we’ll look into the basic python operator’s.

1.7 Exercises

1. Will the following lines of code print the same thing? Explain why or why not.

```
x = 6
print(6)
print("6")
```

2. Will the following lines of code print the same thing? Explain why or why not.

```
x = 7
print(x)
print("x")
```

3. What is the largest floating-point value available on your system?

4. What is the smallest floating-point value available on your system?

5. What happens if you attempt to use a variable within a program, and that variable has not been assigned a value?

6. What is wrong with the following statement that attempts to assign the value ten to variable x?

```
10 = x
```

2. Once a variable has been properly assigned can its value be changed?

3. In Python can you assign more than one variable in a single statement?

9. Classify each of the following as either a legal or illegal Python identifier:

(a) fred

(b) if

(c) 2x

(d) -4

(e) sum_total

(f) sumTotal

(g) sum-total

(h) sum total

(i) sumtotal

(j) While

(k) x2

(l) Private

(m) public

(n) $16

(o) xTwo

(p) _static

(q) _4

(r) ___

(s) 10%

(t) a27834

(u) wilma's

10. What can you do if a variable name you would like to use is the same as a reserved word?

11. How is the value 2.45×10−5 expressed as a Python literal?

12. How is the value 0.000000000000000000000000449 expressed as a Python literal?

13. How is the value 56992341200000000000000000000000000000 expressed as a Python literal?

14. Can a Python programmer do anything to ensure that a variable’s value can never be changed after its initial assignment?

15. Is "i" a string literal or variable?

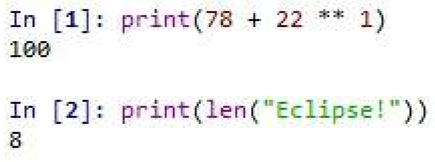
2. Statements, Expressions and operators

In this chapter we'll get understand the what a statement, expression and an operator in python is. We'll discuss the different basic operators in python and the operator precedence in python. Let's get started.

2.1 Statements and Expressions

Statements are instructions that the python interpreter could execute. So far only the assignment, del and method invocation statement has been discussed but there are other statements like "for" statements, "if" statements, "while" statements, "import" statements and others.

Expressions, they are a combination of variables, values, function calls and operators. Expressions in python need to be evaluated before they can be used. When an expression is the right hand side of an assignment statement the expression is first evaluated before it is assigned to name. When an expression is passed as the parameter to pythons "print" function the expressions is first evaluated then the result is displayed. In the below examples the expressions are first evaluated then are used.



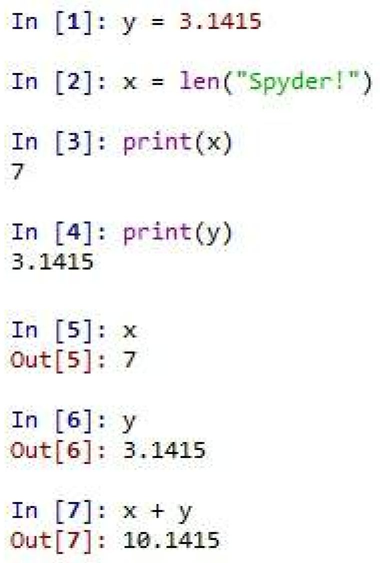
```
In [1]: print(78 + 22 ** 1)
100

In [2]: print(len("Eclipse!"))
8
```

Here len( ) is a built in python function that returns the length of the string passed as an argument. In the above code "78 + 2 ** 1" and "len("Eclipse!")" are both expressions they are evaluated before they are used by the print function.

Evaluating an expression results in a value, and that's the reason why expressions can be on the right hand side of an assignment statement. A value or variable or a function call that return a value are all simple expressions. Evaluating a variable gives the value the variable refers to. It's important to remember that expressions in python returns a value. Take a look at the code below.

```python
y = 3.1415
x = len("Spyder!")
print(x)
print(y)
```

We can see one of the differences between expressions and assignments when the above code is run in the python shell.



```
In [1]: y = 3.1415

In [2]: x = len("Spyder!")

In [3]: print(x)
7

In [4]: print(y)
3.1415

In [5]: x
Out[5]: 7

In [6]: y
Out[6]: 3.1415

In [7]: x + y
Out[7]: 10.1415
```

We can see that for assignments only the prompt is returned and no value. That's because assignments statements do not return any values. The assignment statement "`y = 3.1415`" doesn't return any value so only the prompt is returned, but the result of the simple expression "`3.1415`" is returns a value(i.e `3.1415`) which creates a new float object in memory and assigns it's reference to '`y`'. Assignment statements are simply executed and don't return any value.

When the print statement is called on '`y`' we can see the value that '`y`' is referring to as print function returns a value, also when '`y`' is entered into the shell by itself '`y`' is evaluated and returned.

2.2 What are operators and operands

Operands the tokens in python represent some kind of computation and the computation happens between the operand or operands that are given to the operator to work on.

The following are some legal expressions using arithmetic operator’s.

```
# Assignments
x = 10
y = 20
z = 30

# Valid expressions using arithmetic operators
25 + 22
x - 1
y * 60 + z
x / 60
5 ** 2
(5 + 13) * (55 + 77)
```

The operators that python supports are listed below.

- Arithmetic operators
- Relational operators(comparison operators)
- Assignment operators
- Logical operators
- Bitwise operators
- Membership operators
- Identity operators

We’ll go through each of these operators one by one.

2.3 Arithmetic operators

The seven arithmetic operators supported by python are listed and described below briefly. We’ll then go through some examples on these arithmetic operators.

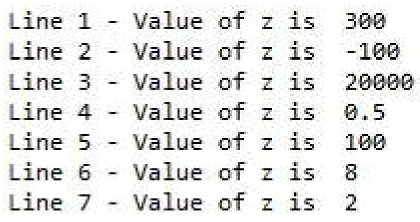
Operator	Description
+	Addition - Adds values on either side of the operator
_	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
**	Exponent - Performs exponential (power) calculation on operators
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.

Now lets look at some example code that demonstrates how the arithmetic operators work in python.

```python
#=======================================
# Assignments
#=======================================
x = 100
y = 200
z = 500

#=======================================
# Using arithmetic operators
#=======================================
z = x + y
print("Line 1 - Value of z is ", z)
z = x - y
```

```
print("Line 2 - Value of z is ", z)
z = x * y
print("Line 3 - Value of z is ", z)
z = x / y
print("Line 4 - Value of z is ", z)
z = x % y
print("Line 5 - Value of z is ", z)
x = 2
y = 3
z = x ** y
print("Line 6 - Value of z is ", z)
x = 10
y = 5
z = x//y
print("Line 7 - Value of z is ", z)
```

The above code gives the output given below.



```
Line 1 - Value of z is  300
Line 2 - Value of z is  -100
Line 3 - Value of z is  20000
Line 4 - Value of z is  0.5
Line 5 - Value of z is  100
Line 6 - Value of z is  8
Line 7 - Value of z is  2
```

2.4 Relational operators

The six relational operators that python supports are listed and described in the table below.

Operators	Description
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of

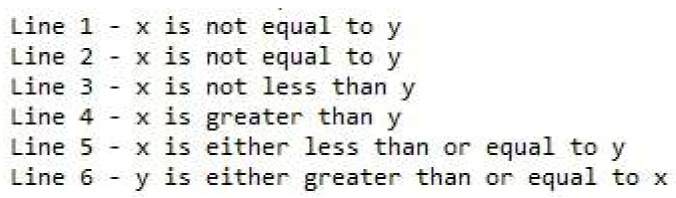
	right operand, if yes then condition becomes true.
`<`	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
`>=`	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
`<=`	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

The example code below demonstrates the use of pythons relational operators. As relational operators give a Boolean result of either true or false, if-else statements are used to demonstrate the the behaviour of relational operators in python .

```
#======================================
# Assignments
#======================================
x = 10
y = 5
z = 0

#======================================
# Using relational operators
#======================================
if( x == y ):
    print("Line 1 - x is equal to y")
else:
    print("Line 1 - x is not equal to y")
if( x != y ):
    print("Line 2 - x is not equal to y")
else:
    print("Line 2 - x is equal to y")
if( x < y ):
    print("Line 3 - x is less than y")
else:
    print("Line 3 - x is not less than y")
if( x > y ):
```

```python
    print("Line 4 - x is greater than y")
else:
    print("Line 4 - x is not greater than y")

x = 5
y = 20

if( x <= y ):
    print("Line 5 - x is either less than or equal to y")
else:
    print("Line 5 - x is neither less than nor equal to y")
if( y >= x ):
    print("Line 6 - y is either greater than or equal to x")
else:
    print("Line 6 - y is neither greater than nor equal to x")
```

The above python code is ran it gives the output shown below.



```
Line 1 - x is not equal to y
Line 2 - x is not equal to y
Line 3 - x is not less than y
Line 4 - x is greater than y
Line 5 - x is either less than or equal to y
Line 6 - y is either greater than or equal to x
```

2.5 Assignment operators

Python’s most basic assignment operator is ‘=’ which assigns a reference to the value on the right hand of the operator to the variable name on the left hand side of the operator. Python also has several other assignment operators that perform an arithmetic operation before the assignment is made. The nine different assignment operators in python are listed and described below.

Operator	Description
=	Simple assignment operator, Assigns values from right side

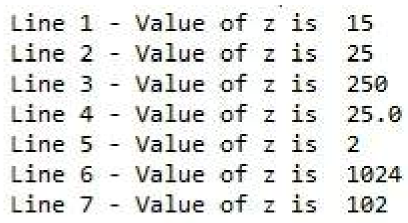
	operands to left side operand
+=	Add AND assignment operator, It adds right operand to the left operand and assigns the result to left operand
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assigns the result to left operand
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assigns the result to left operand
/=	Divide AND assignment operator, It divides left operand with the right operand and assigns the result to left operand
%=	Modulus AND assignment operator, It takes modulus using two operands and assigns the result to left operand
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assigns value to the left operand
//=	Floor Division and assigns a value, Performs floor division on operators and assigns value to the left operand

Example code is is give below that explains the working of the different assignment operators in python.

```python
#=======================================
# Assignments
#=======================================
x = 10
y = 5
z = 0

#=======================================
# Using assignment operators
#=======================================
z = x + y
print("Line 1 - Value of z is ", z)
z += x
print("Line 2 - Value of z is ", z)
z *= x
print("Line 3 - Value of z is ", z)
```

```python
z /= x
print("Line 4 - Value of z is ", z)
z = 2
z %= x
print("Line 5 - Value of z is ", z)
z **= x
print("Line 6 - Value of z is ", z)
z //= x
print("Line 7 - Value of z is ", z)
```

The above code gives the below output when executed.



```
Line 1 - Value of z is  15
Line 2 - Value of z is  25
Line 3 - Value of z is  250
Line 4 - Value of z is  25.0
Line 5 - Value of z is  2
Line 6 - Value of z is  1024
Line 7 - Value of z is  102
```

2.6 Bitwise operators

Python's bitwise operators are applied to the binary form of the values that python's numeric data types can hold.

Bitwise operators are performed bit by bit. For example when a bitwise operation is performed on "`x = 10`" and "`y = 20`", the binary equivalent of of the integer objects `10` and `20` are used instead of the decimal values(`10, 20`) that were assigned to '`x`' and '`y`', so the operands to the bitwise operation are the binary values "`1010`"(`10`) and "`10100`"(`20`). Operation is performed bit by bit.

```
X = 63 (base - 10) = 0011 1100 (base - 2)

Y = 13 (base - 10) = 0000 1101 (base - 2)

X & y = 0000 1100

X & y = 0011 1101

X & y = 0011 0001
```

The six bitwise operators that python supports are listed and briefly described in the below table.

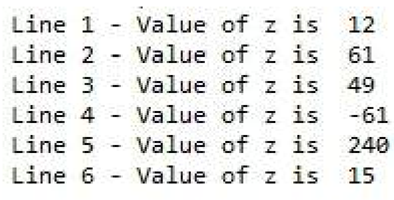
Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

The following example code makes use of each of bitwise operations that python supports.

```
#=======================================
# Assignments
#=======================================

#-----In Decimal------In Binary--------
#--------------------------------------
x =      60        # 0011 1100
y =      13        # 0000 1101
z =       0        # 0000 0000

#=======================================
# Using bitwise operators
#=======================================

z = x & y     # 12 = 0000 1100
print("Line 1 - value of z is ", z)

z = x | y     # 61 = 0011 1101
print("Line 2 - value of z is ", z)
```

```

z = x ^ y     # 49 = 0011 0001
print("Line 3 - value of z is ", z)

z = ~x        #-61 = 1100 0011
print("Line 4 - value of z is ", z)

z = x << 2    #240 = 1111 0000
print("Line 5 - value of z is ", z)

z = x >> 2    # 15 = 0000 1111
print("Line 6 - value of z is ", z)
```

When the above python script is executed it displays the following output.



```
Line 1 - Value of z is  12
Line 2 - Value of z is  61
Line 3 - Value of z is  49
Line 4 - Value of z is  -61
Line 5 - Value of z is  240
Line 6 - Value of z is  15
```

2.7 Logical operators

The logical operators and, or and not are supported by python these logical operators are described below. Unlike languages like c, logical operators are written in plain in English.

Operators	Description
and	Called logical AND operator. If both the operands are true, then the condition becomes true.
or	Called logical OR Operator. If any of the two operands are non zero, then the condition becomes true.
not	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true, then Logical NOT

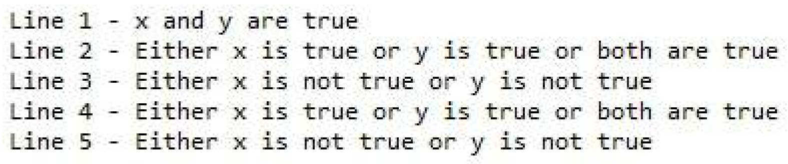
	operator will make it false.

Any non zero numerical value is considered true and the numerical value zero(`0`) is considered false. The example coed below works with python's logical operators.

```
#=====================================================================
# Assignments
#=====================================================================
x = 5
y = 10

#=====================================================================
# Using logical operators
#=====================================================================
if( x and y ):
    print("Line 1 - x and y are true")
else:
    print("Line 1 - Either x is not true or y is not true")

if( x or y ):
    print("Line 2 - Either x is true or y is true or both are true")
else:
    print("Line 2 - Neither x is true nor y is true")


x = 0

if( x and y ):
    print("Line 3 - x and y are true")
else:
    print("Line 3 - Either x is not true or y is not true")

if( x or y ):
    print("Line 4 - Either x is true or y is true or both are true")
else:
    print("Line 4 - Neither x is true nor y is true")
```

```python

if not( x and y ):
   print("Line 5 - Either x is not true or y is not true")
else:
   print("Line 5 - x and y are true")
```

The output when the above code is executed is provided below.



```
Line 1 - x and y are true
Line 2 - Either x is true or y is true or both are true
Line 3 - Either x is not true or y is not true
Line 4 - Either x is true or y is true or both are true
Line 5 - Either x is not true or y is not true
```

2.8 Membership operators

Python’s two membership operators enable testing whether a given element exists in a sequence(lists, tuples or strings). The two membership operators that python support are listed and described in the table below.

Operators	Description
`in`	Evaluates to true if it finds a variable in the specified sequence and false otherwise.
`not in`	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

The example code below may give a better understanding on what the python membership operators do.

```python
#======================================================================
# Assignments
#======================================================================
x = 6
y = 7
li = [1, 2, 3, 4, 5 ];

#======================================================================
# Using membership operators
#====================================================================
if ( x in li ):
    print("Line 1 - x is available in the given list")
else:
    print("Line 1 - x is not available in the given list")

if ( y not in li ):
    print("Line 2 - y is not available in the given list")
else:
    print("Line 2 - y is available in the given list")

x = 4

if ( x in li ):
    print("Line 3 - x is available in the given list")
else:
    print("Line 3 - x is not available in the given list")
```

The above code gives the below output.

```
Line 1 - x is not available in the given list
Line 2 - y is not available in the given list
Line 3 - x is available in the given list
```

2.9 Identity operators

Python’s identity operators can be used to check whether two names points to/references the same object in memory. There are two identity operators supported by python, they are described in the following table.

Operators	Description
`is`	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
`is not`	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

The code below demonstrates the use of identity operators in python. When you try this out in your python shell you can see how the python interpreter assigns objects to variables.

The `id( )` method takes a variable as parameter and returns the identity of the object that name references. An identity of an object is constant and is retained by the object as long as it is alive. The output of the of the above code is given below.

```python
#================================================================
# Assignments
#================================================================
x = 10
y = 10

#================================================================
# Using identity operators
#================================================================
if ( x is y ):
    print("Line 1 - x and y have same identity")
else:
    print("Line 1 - x and y do not have same identity")
if ( id(x) == id(y) ):
    print("Line 2 - x and y have same identity")
else:
    print("Line 2 - x and y do not have same identity")

x = 30

if ( x is y ):
    print("Line 3 - x and y have same identity")
else:
    print("Line 3 - x and y do not have same identity")
if ( x is not y ):
    print("Line 4 - x and y do not have same identity")

```

```python
else:
    print("Line 4 - x and y have same identity")
```

```
Line 1 - x and y have same identity
Line 2 - x and y have same identity
Line 3 - x and y do not have same identity
Line 4 - x and y do not have same identity
```

2.10 Precedence and associativity of python operators

It's already been discussed that an expression in python and most other programming languages are a combination of values, variables, function calls and operators. Lets take some instances of simple expressions in python.

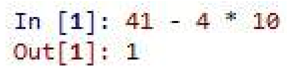
```python
# Assignments
var1 = 10
var2 = 20
var3 = "Ipython"

# Expressions with a single or no operator
var1
var2
var1 + var2
var1 % 3
len(var3) ** var1
1 + 4.67
```

Remember a valid expression always returns a value and an expression can not be used until its evaluated. In the above code every expression has either one or no operator so evaluation is very straight forward but in the cases where multiple operators are involved in the expression what is the order of evaluation to be followed? Valid expressions of this case are given in the code below.

```python
# Assignments
var1 = 10
var2 = 20
var3 = "Ipython"

```

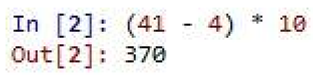
```python
# Expressions with multiple operators
var1 + var2 ** (2 % 7) // var1
var1 % 3 ** (var1 + var2)
len(var3) ** var1 % 100000
1 + 4.67 * 5 + (len(var3) * id(var2))
```

Python specifies the order of precedence of operators and associativity(in the case that there are more than one operator with the same precedence level). For example example multiplication has a higher order than subtraction.



```
In [1]: 41 - 4 * 10
Out[1]: 1
```

But we can change this order of evaluation by using parentheses(( )). parentheses have a higher precedence level.



```
In [2]: (41 - 4) * 10
Out[2]: 370
```

Python operator precedence rule (from highest precedence to lowest)

Operator	Meaning
`()`	Parentheses
`**`	Exponent
`+x, -x, ~x`	Unary plus, Unary minus, Bitwise NOT
`*, /, %, //`	Multiplication, Division, Floor division, Modulus
`+, -`	Addition, Subtraction
`<<, >>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==, !=, >, >=, <, <=, in, not in, is, is not,`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND

`or`	Logical OR

Python operator associativity

In the above table there are certain precedence levels that have multiple operators, this means that all those operators have the same level of precedence. When there are multiple operators of the same precedence in an expression the appropriate associativity rule is followed. For almost all of pythons operators the associativity rule is left to right. Take a look at the example below.

```python
# Left to right associativity
# Output: 3
print(5 * 2 // 3)          # *, // have equal precedence
                         # so follow it's associativity rule

# Shows left to right associativity
# Output: 0
print(5 * (12 // 3) % 2)   # parentheses has the highest precedence
                         # *, //, % have equal precedence
                         # so follow it's associativity rule
```

The exponent operator(**) has right to left associativity in python, so if there are multiple exponent operators they are evaluated from right to left after operators of higher precedence have already been evaluated.

```python
#================================================================
# Right to left associativity of the ** operator
#================================================================

# Output: 512
print(2 ** 3 ** 2)

# Output: 64
print((2 ** 3) ** 2) # Parentheses has the highest precedence
```

Non associative operators(Assignment and comparison operators)

There are some python operators that do not follow any associativity rule, the assignment and comparison operators. There are some special rules followed when multiple operators of this type are used in a sequence.

Comparison operators:

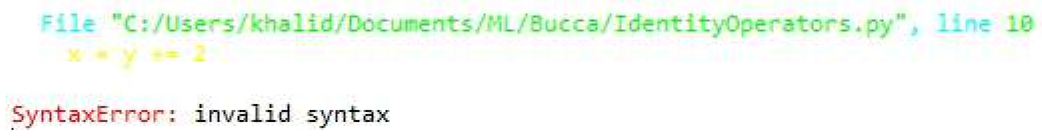
Take a conditional expression "`x < y < z`" this expression is not equivalent to "`(X < y) < z`" or "`X < (y < z)`", but instead is equivalent to "`X < y and y < z`" and this is evaluated from left to right.

Assignment operators:

Assignments of the form "`x = y = z`" are valid and are evaluated from right to left, but assignments of the form "`x = y += z`" are not considered to be a valid syntax.

```
# Initializing x, y, z
x = 1
y = 2
z = 3
# The below assignment
# expression is invalid
# (Non-associative operators)
# SyntaxError: invalid syntax

x = y += 2
```

So when the python interpreter encounters such syntax it raises a syntax error.



```
File "C:/Users/khalid/Documents/ML/Bucca/IdentityOperators.py", line 10
  x = y += 2

SyntaxError: invalid syntax
```

2.11 Exercises

1. Is the literal 4 a valid Python expression?

2. Is the variable x a valid Python expression?

3. Is x + 4 a valid Python expression?

4. What affect does the unary + operator have when applied to a numeric expression?

5. Sort the following binary operators in order of high to low precedence: +, -, *, //, /, %, =.

6. Given the following assignment:

   `x = 2`

   Indicate what each of the following Python statements would print.

   (a) `print("x")`

   (b) `print(’x’)`

   (c) `print(x)`

   (d) `print("x + 1")`

   (e) `print(’x’ + 1)`

   (f) `print(x + 1)`

7. Given the following assignments:

   `i1 = 2`

   `i2 = 5`

   `i3 = -3`

   `d1 = 2.0`

   `d2 = 5.0`

   `d3 = -0.5;`

   Evaluate each of the following Python expressions.

   (a) i1 + i2

   (b) i1 / i2

   (c) i1 // i2

   (d) i2 / i1

   (e) i2 // i1

   (f) i1 * i3

(g) d1 + d2

(h) d1 / d2

(i) d2 / d1

(j) d3 * d1

(k) d1 + i2

(l) i1 / d2

(m) d2 / i1

(n) i2 / d1

(o) i1/i2*d1

(p) d1*i1/i2

(q) d1/d2*i1

(r) i1*d1/d2

(s) i2/i1*d1

(t) d1*i2/i1

(u) d2/d1*i1

(v) i1*d2/d1

8. What is printed by the following statement:

   #print(5/3)

9. Given the following assignments:

```
i1 = 2

i2 = 5

i3 = -3

d1 = 2.0

d2 = 5.0

d3 = -0.5
```

   Evaluate each of the following Python expressions.

   (a) i1 + (i2 * i3)

   (b) i1 * (i2 + i3)

(c) i1 / (i2 + i3)

(d) i1 // (i2 + i3)

(e) i1 / i2 + i3

(f) i1 // i2 + i3

(g) 3 + 4 + 5 / 3

(h) 3 + 4 + 5 // 3

(i) (3 + 4 + 5) / 3

(j) (3 + 4 + 5) // 3

(k) d1 + (d2 * d3)

(l) d1 + d2 * d3

(m) d1 / d2 - d3

(n) d1 / (d2 - d3)

(o) d1 + d2 + d3 / 3

(p) (d1 + d2 + d3) / 3

(q) d1 + d2 + (d3 / 3)

(r) 3 * (d1 + d2) * (d1 - d3)

10. What symbol signifies the beginning of a comment in Python?

11. How do Python comments end?

12. Which is better, too many comments or too few comments?

13. What is the purpose of comments?

14. Why is human readability such an important consideration?

15. Consider the following program which contains some errors. You may assume that the comments within the program accurately describe the program's intended behavior.

```
# Get two numbers from the user
n1, n2 = eval(input()) # 1
# Compute sum of the two numbers
print(n1 + n2) # 2
# Compute average of the two numbers
print(n1+n2/2) # 3
# Assign some variables
d1 = d2 = 0 # 4
# Compute a quotient
print(n1/d1) # 5
# Compute a product
n1*n2 = d1 # 6
# Print result
print(d1) # 7
```

For each line listed in the comments, indicate whether or not an interpreter error,run-time exception, or logic error is present. Not all lines contain an error.

16. Write the shortest way to express each of the following statements.

(a) x = x + 1

(b) x = x / 2

(c) x = x - 1

(d) x = x + y

(e) x = x - (y + 7)

(f) x = 2*x

(g) number_of_closed_cases = number_of_closed_cases + 2*ncc

17. What is printed by the following code fragment?

```
x1 = 2
x2 = 2
x1 += 1
x2 -= 1
print(x1)
print(x2)
```

Why does the output appear as it does?

18. Consider the following program that attempts to compute the circumference of a circle given the radius entered by the user. Given a circle's radius, r, the circle's circumference, C is given by the formula:

C = 2pr

```
r = 0
PI = 3.14159
# Formula for the area of a circle given its radius
C = 2*PI*r
# Get the radius from the user
r = eval(input("Please enter the circle's radius: ")
# Print the circumference
print("Circumference is", C)
```

(a) The program does not produce the intended result. Why?

(b) How can it be repaired so that it works correctly?

19. Write a Python program that ...

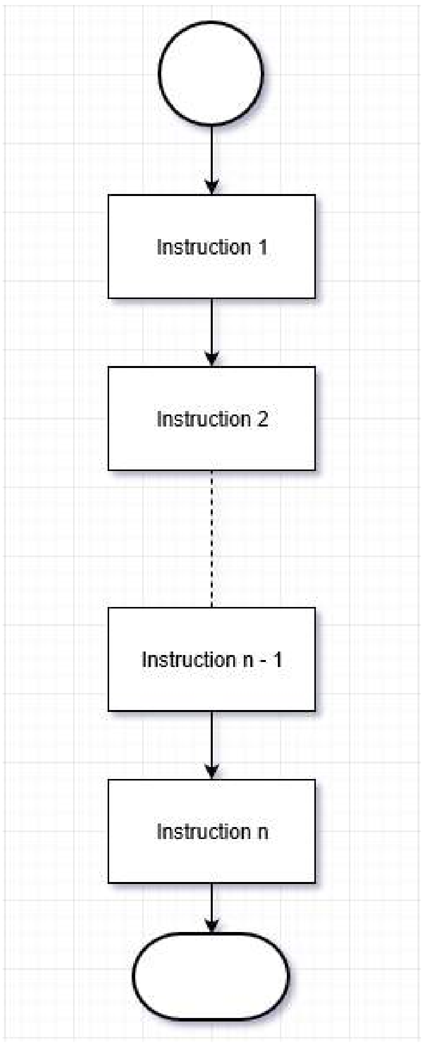
20. Write a Python program that ...

3. Pythons flow control tools

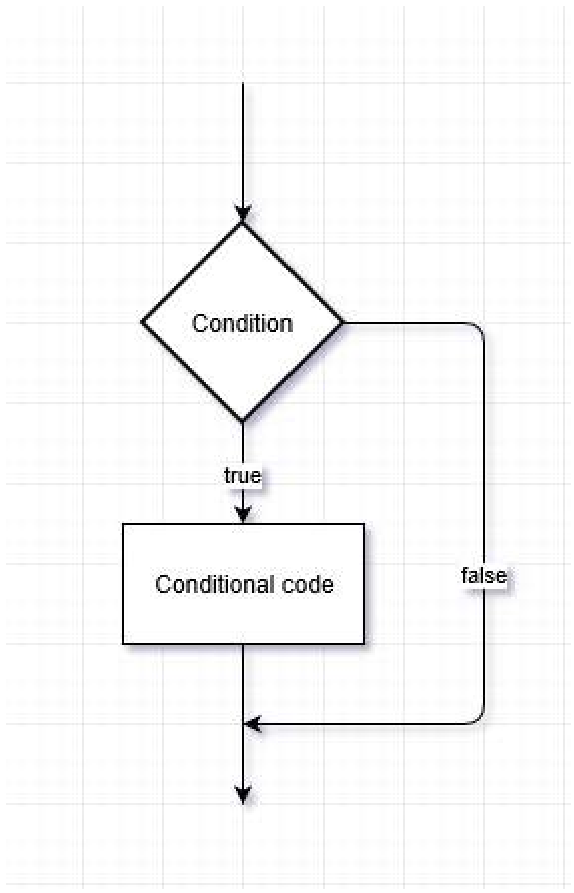
Most the code examples that were discussed prior to this chapter followed a sequential execution of statements, from start to finish(from line 1 to that last line of code) except some examples that had to involve the “if” and “if-else”for the purpose of explanation of the content being discussed. We’ll now go through each of the tools that that python offers to change the normal sequential flow of control that python programs follow.

These control flow statements can also be referred to as “branching statements”. There are two types of branching statements in python “conditional branching” and “unconditional branching”. conditional branching statements change the control flow of programs based on the result of conditional expression, unconditional branching statements change the control flow of the program when they are encountered and regardless of any condition. Unconditional branching instructions are almost always used along with conditional branching statements.

The sequential control of a python program is shown below in the flow diagram.



The following flowchart portrays the general decision making structure in almost all programming languages.



In python the numerical value "`0`" and "`null`" are considered to be "`false`" any value that is not "`0`" or "`null`" are considered to be "`true`". The types of decision making statements in python are listed and briefly described below, we'll go into details in the coming sections.

Statement	Description
if statements	An if statement consists of a conditional expression followed by one or more statements.
if-else statements	An if statement can be followed by an optional else statement, which executes when the conditional expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).

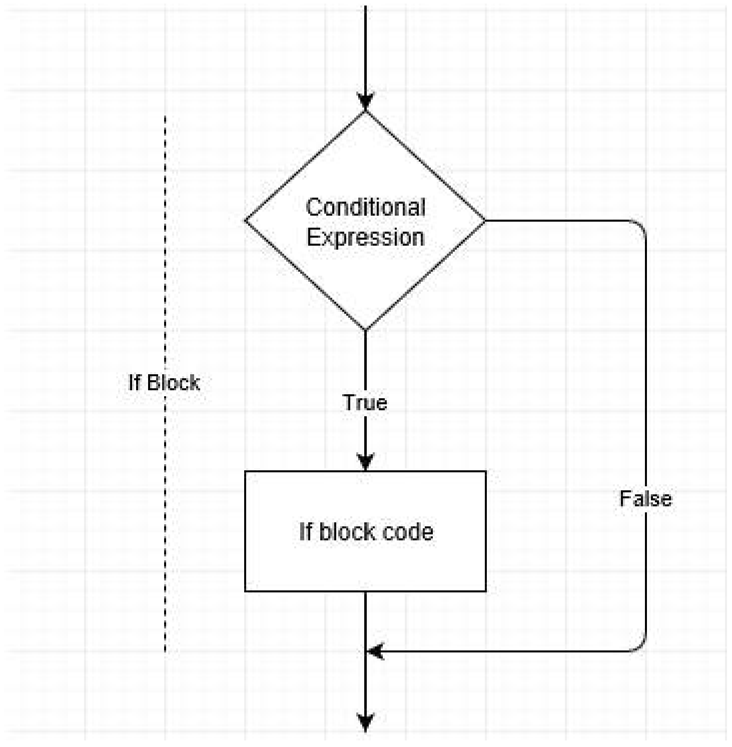
Decision making is needed in cased where a block of code should be executed only when certain condition is true, for these kind of cases that require decision making we can use "`if..elif..else`" statements.

3.1 if statements

The “if statement” has a conditional expression, when the result of the conditional expression is true the body of the “if statement” is executed, in the case that it is false body of the “if statement” is skipped. The general syntax of the if statement is given below.

```
if <Boolean Expression> :

      <if block Statement(s)>
```

In python the body of the “if statement” is specified by indentation. The body starts with an indentation and the first line without the indentation marks the end and is outside the body of the “if block”. The flowchart of pythons “if statement” is shown below.



The below example code demonstrates how the “if statement” works in python.

```python
#=================================================================
# Demonstrating the if statement.
# If the number is positive, then print the appropriate message.
#================================================================
num = 3
if num > 0:                              # if statements condition
    print(num, "is a positive number.")  # indentation marks the if block
    print("still in if block")           # indentation marks the if block
print("outside the if block.")           # outside the if block

num = -1

if num > 0:                              # if statements condition
    print(num, "is a positive number.")  # indentation marks the if block
print("outside the if block.")           # outside the if block

```

The body of the "if statement" is executed when the condition "`num > 0`" is evaluated to be true. If the condition is evaluated to be "`true`" the body of the "if statement"(marked by the indentation) gets executed, else it is skipped. In the above example code the condition of the first "if statement" is true so it's body is executed and the condition of the second "if statement" is false so it's body is skipped. The output when the above program is executed is given below.

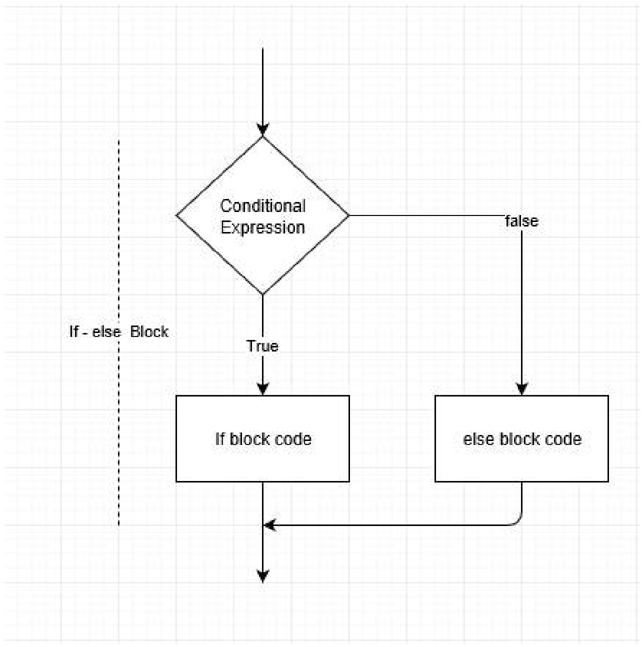
```
3 is a positive number.
still in if block
outside the if block.
outside the if block.
```

3.2 if-else statements

The "if-else statement" has a conditional expression, when the result of the conditional expression is true the body of the "if" is executed, in the case that it is false body of the "else" is executed. The general syntax of the if-else statement is given below.

```
if <Boolean Expression>:

        <if block Statement(s)>

else:
```

```
<else block Statement(s)>
```



In the given flow chart of the flow chart of the "if-else statement" we can see that when the Boolean expression is evaluated to be false the the body of the else block is executed instead.

The example code below shows an instance of using the "if-else statement", take a look for more clarity.

```python
#================================================================
# Demonstrating the if..elif..else Decision chain.
# Print the appropriate message for the different values of num
#================================================================

num = 3

# try these two variations your self
# num = -12
# num = 0

if num >= 0:
    print("num is either Positive or Zero")
else:
    print("num is a Negative number")
```

The output of the above program is given below, and try the above two variations program yourself.

```
num is either Positive or Zero
```

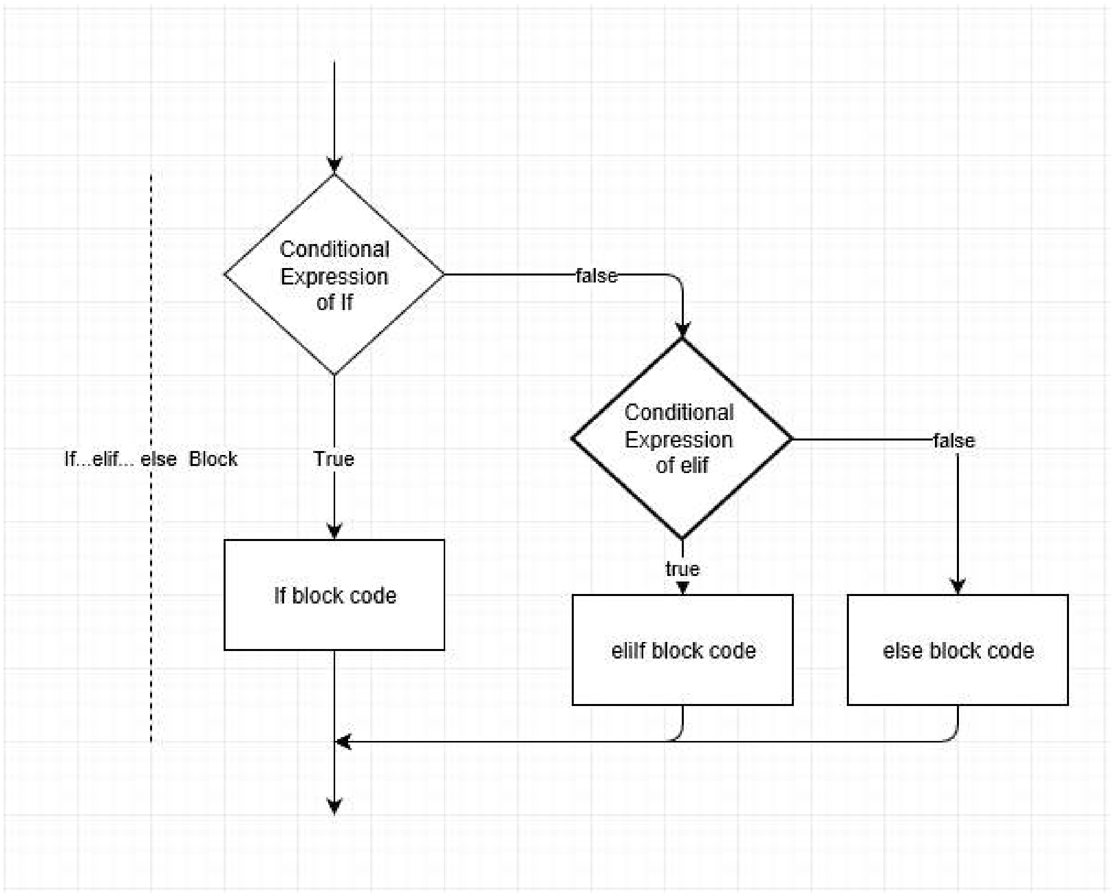
In the above program the condition of the “if statement”(`num > 0`) is true so the “if block” is executed and the “else” block code is skipped. If the value of `num` is changed to “0” the condition is still evaluated to be true so again the “if” block is executed instead. If the value of `num` is changed to “-12” the “else” block is executed instead.

3.3 if..elif..else statements

The general syntax of “if..elif..else” is given below. First the Boolean expression of “if” is evaluated, if it is evaluated to be false then the Boolean expression of elif is evaluated, if it is evaluated to be true then the body of that “elif” is executed if it is false then it’s body is skipped and control goes to the next “elif” or “else”. It is allowed to have more than one “elif” in the “if..elif..else” chain.

```
if <Boolean Expression>:

       <if block Statement(s)>

elif <Boolean Expression>:

       <elif block Statement(s)>

else:

       <else block Statement(s)>
```

The flowchart of the if..elif..else chain is shown below. Remember there can be more than one “elif” in the “if..elif..else” chain.



Lets take a look at a code example to better understand how the if..elif..else decision chain works in python.

```python
#===============================================================
# Demonstrating the if..elif..else decision chain.
# Print the appropriate message for the different values of num
#===============================================================

num = 34

# Try these two variations yourself
# num = 0
# num = -205

if num >= 0:
    print("You've entered a positive number!")
elif num == 0:
    print("You've entered zero!")
```

```
else:
    print("You've entered a negative number!")
```

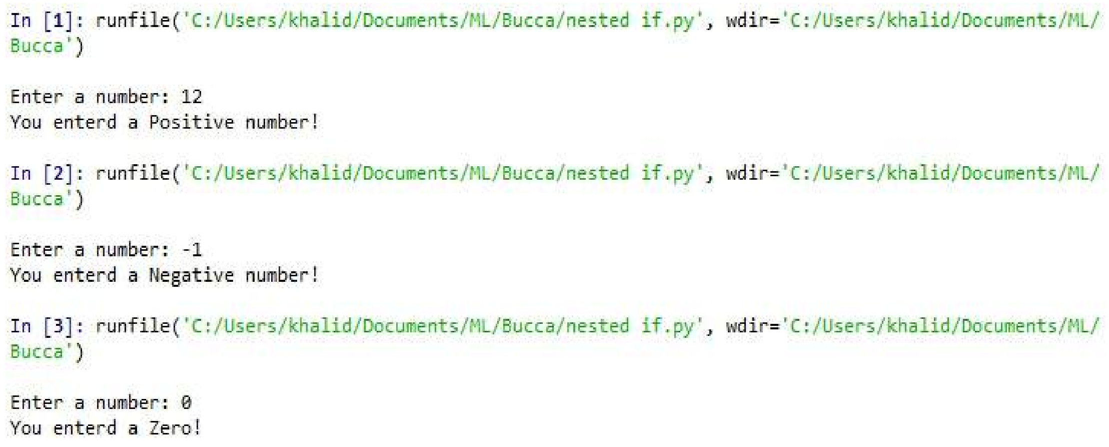
In the above program if “num” is positive then “positive number”Since the Boolean expression of “if” evaluates to be true, the if block code is executed, after execution the control goes out of the current “if..elif.else” decision chain. Try the above two variations yourself. The output of the above program is given below.

```
Positive number
```

Python permits nesting any number “if..elif..else” chains within “if..elif..else” decision chains. The only way to differentiate one chain from the other is by the indentation and so in very complex cases gets confusing to read, This is better avoided to maintain the readability of the program. Lets take a look at a code example that uses “nested if statements”.

```
#===================================================================
# Demonstrating the nested if statements.
# Print the appropriate message for the different values of num
#===================================================================

num = int(input("Enter a number: "))   # Input from the user
                                        # convert it to the type int

if num >= 0:
    if num == 0:
        print("You've entered zero!")
    else:
        print("You've entered a positive number!")
else:
    print("You've entered a negative number!")

```

The output of the above program for each possible case is given below. In the above python code the input function waits for the user to enter a value in the python shell and returns this value as a string. In this program it is expected that the user enters a numerical value, this value entered by the user is returned by the input function as a string which is converted into an int type using the int( ) method.



```
In [1]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/
Bucca')

Enter a number: 12
You enterd a Positive number!

In [2]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/
Bucca')

Enter a number: -1
You enterd a Negative number!

In [3]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/
Bucca')

Enter a number: 0
You enterd a Zero!
```

There are some statements in python that allow iterating the execution over a certain part of the code until some condition is met, these types of statements are called looping statements. Python provides two looping statements

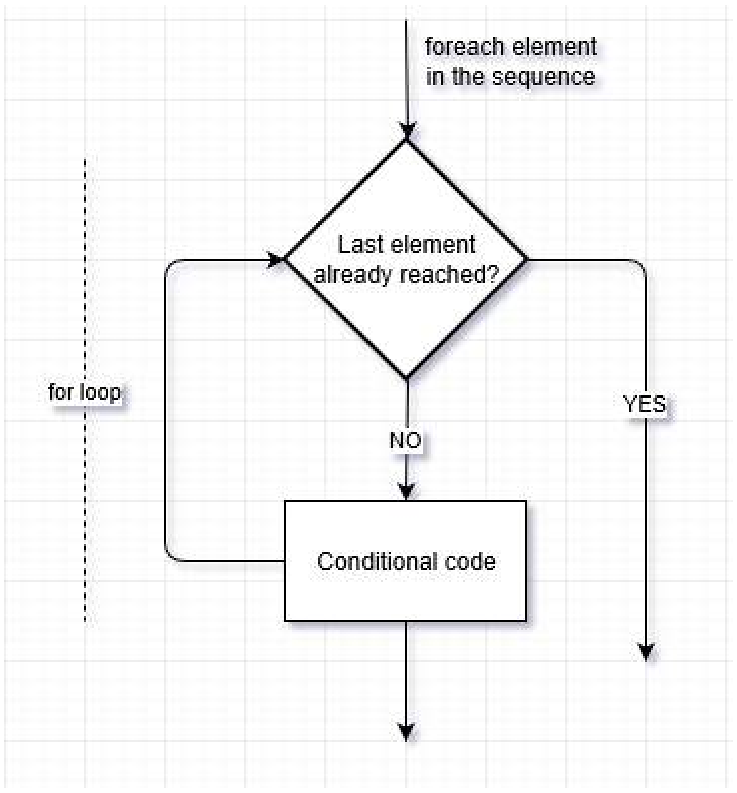
- for loop
- while loop

3.4 Pythons for loop

For loops in are very handy they are used iterate over all the elements of a sequence typed object(lists, tuples, strings) or an object that is iterated through. They are similar to the “foreach statement” which is available in languages like C#. The body of the for loop is executed once for each element of the container object. The general syntax of pythons for loop is given below.

```
for <variable> in <iterable typed object>:

        <Statements in body of for loop>
```

Here body of the for loop is identified by indentation. `<variable>` can be used in the body of the for loop and it holds the value of the element in the sequence typed object for each iteration. The loop continues until the last element in the sequence is reached. The flow chart of the for loop is shown below.


foreach element in the sequence
Last element already reached?
for loop
YES
NO
Conditional code

Lets look at a program that uses a “for loop” in it. In the example below a for loop is used to add the elements of a list and print the output. “numbers” is a a hard coded list of integers and variable “`sum`” is initialized to `0`. In every iteration the current element of the sequence of that iteration is added to the variable “`sum`”.

```python
#===============================================================
# Program to find the sum of all numbers in a list
#===============================================================


# List of integers
numlist = [10, 20, 30, 40, 50, 60, 70, 80, 90]

# variable to store the sum of elements
sum = 0

# iterate over the list
for num in numlist:
    sum = sum + num

```

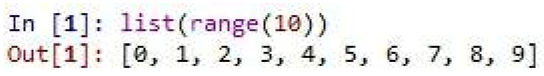
```python

# Output: The sum is 450
print("The sum of all elements is ", sum)
```

The output of the above code is given below.

```
The sum of all elements is  450
```

3.4.1 The range( ) method

Python provides a range method that can be used to generate a list of a sequence of numbers. For example if "`range(10)`" is called it can generate a list of numbers from 0 to 10.

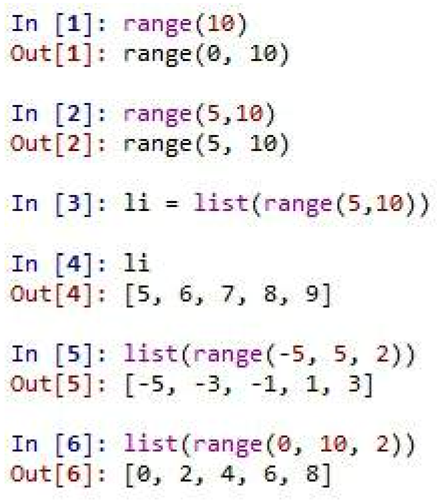


```
In [1]: list(range(10))
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The number to start from, the number to end at and the step size can be specified in the range function by passing them as parameters as "`range(start, stop, step size)`". The step size has a default value of "`1`" if not specified. The start index defaults to "`0`" if not specified. The range function can be called in the following ways.

- `range(stop)`
- `range(start, stop)`
- `range(start, stop, step size)`

Here `start` is the number to start from, `stop` is the number to stop at(not generated by range function). In python 3.7 the range function returns an object of type "`range`", which can be converted to a list using the `list( )` method. The following examples can clarify more on the `range( )` function.

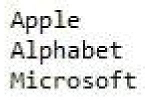


```
In [1]: range(10)
Out[1]: range(0, 10)

In [2]: range(5,10)
Out[2]: range(5, 10)

In [3]: li = list(range(5,10))

In [4]: li
Out[4]: [5, 6, 7, 8, 9]

In [5]: list(range(-5, 5, 2))
Out[5]: [-5, -3, -1, 1, 3]

In [6]: list(range(0, 10, 2))
Out[6]: [0, 2, 4, 6, 8]
```

The `range( )` function is being discussed with "for loops" because the `range( )` function is mostly used with for loops. Then can be used to generate the sequence of items that the "for loop" iterates over. In the example below the `range( )` function is combined with the `len( )` is used in a for loop to iterate through a list of items using index.

```python
#=======================================================================
# iterate through a list using indexing
#=======================================================================
Titans = ['Apple', 'Alphabet', 'Microsoft']

# Iterate through the list using the index
for i in range(len(Titans)):
    print(Titans[i])        # Accessing the i-th element of the list Titans
```

Here `len(Titans)` returns "3" which is passed as a parameter to the `range( )` function. The `range( )` method returns an range object "`range(0, 3)`" which can be iterated through by the for loop. This range can be used as the index for the "for loop". The above code generates the below output.



```
Apple
Alphabet
Microsoft
```

3.4.2 for / else statement

The “for loop” can have an optional “else” block, which is executed after all the elements in the sequence have been iterated through. In the case that a “break” statement was encountered during the execution of the “for loop” which causes an abrupt exit from the loop the “else” part is ignored and so the else part of the “for loop” is executed if there is no abrupt exit from the loop(if a break statement is not encountered while in the loop). An example that involves the “else” part of the “for loop’ is given below.

```python
#================================================================
# for / else loops example
#================================================================
temp = list(range(10))

for i in temp:
    print(i)
else:
    print("Done!")  # Executed after all iterations through
                   # the list are completed.

```

The output of the above program is given below.

```
0
1
2
3
4
5
6
7
8
9
Done!
```

3.5 Pythons while loop

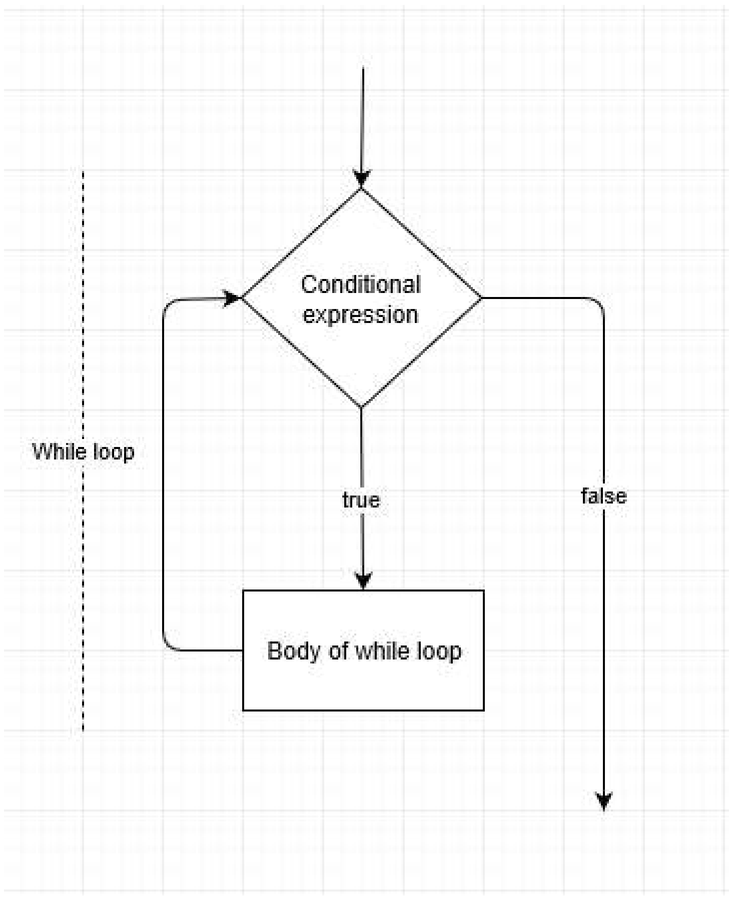
Pythons provides flow control tool, “while” statement which is similar to the “for” statement. The while loop in python allows looping/iterating over a block of code as long as it's conditional expression(condition) evaluates to be true. The difference between this loop and the "for loop" is that the for loop iterates over the elements in a sequence, the while loop on the other hand continue looping as long as its condition is true, so the while loop is mostly used in cased where the number of times to loop is unknown. The syntax of the while loop is given below.

```
while <Boolean expression>:

      <while block code>
```

Here as long as the `<boolean expression>` is evaluated to be true the `<while bloc code>` is executed. The step by step execution of the while loop is shown below.

1. The conditional expression is evaluated.
2. Executed the body of the while loop if the conditional expression is evaluated to be true.
3. Repeat steps 1 and 2 until the conditional expression is evaluated to be false.

Remember that any value other than "`0`" and "`null`" are considered to be true. The flowchart of the "while loop" is given below.

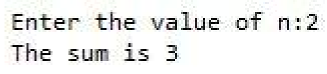


The conditional expression in the “while loop” usually contains one or many variables that is changed every time in the “while” block. This is important because without this change the “while loop” may end up looping without any end as it’s condition is always true, so its important to have one or more variables in the conditional expression of the “while loop” that approaches the terminating condition. Lets look at an example that uses the “while loop”.

```
#============================================================
# This program adds the first "n" natural numbers
#============================================================

n = int(input("Enter the value of n: "))

# Initialize sum and counter variables
i = 1
sum = 0

while i <= n:
    sum = sum + i
    i = i + 1     # Update counter

print("The sum is: ", sum)

```

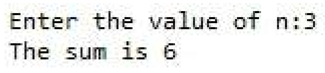
The above python script adds the first ‘`n`’ natural numbers(1, 2, 3, …), where “n” is a user provided input to add until. Here ‘`i`’ is the counter variable, it’s value is incremented in the body of the loop so the next natural number can be added in the next iteration of the loop. The conditional expression of the while loop returns false when the value of ‘`i`’ equals “`n + 1`”, this is when the control leaves the while loop. Some of the possible outputs for the above program are given below.

When “`n = 2`” the output of the above program is given below.



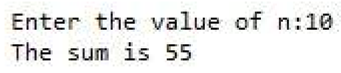
```
Enter the value of n:2
The sum is 3
```

When “`n = 3`” the output of the above program is



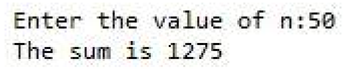
```
Enter the value of n:3
The sum is 6
```

When “`n = 10`” the output of the above program is



```
Enter the value of n:10
The sum is 55
```

When “`n = 50`” the output of the above program is



```
Enter the value of n:50
The sum is 1275
```

3.5.1 while / else statement

The just like the “for loop” the“while loop” can have an optional “else” block, which is executed after all the elements in the sequence have been iterated through. In the case that a “break” statement was encountered during the execution of the “while loop” which causes an abrupt exit from the loop the “else” part is ignored and so the else part of the “while loop” is executed if there is no abrupt exit from the loop.

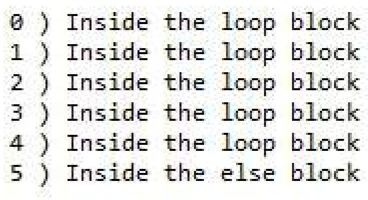
An example that uses the “else” part of the “while loop’ is given below.

```
#===============================================================
# This program illustrates the else statement with a while loop
#===============================================================
```

```python

i = 0  # Initializing the counter variable for the while loop

while i < 5:
    print(i, ") Inside the loop block")
    i = i + 1
else:
    print(i, ") Inside the else block")

```

In the above python script the conditional expression for the while loop "`i < 5`" is `true` until "`i = 5`" which is when the else block code is executed

The output for the above code example is given below.



```
0 ) Inside the loop block
1 ) Inside the loop block
2 ) Inside the loop block
3 ) Inside the loop block
4 ) Inside the loop block
5 ) Inside the else block
```

3.5.2 Infinite loops

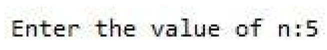
As we have already seen the "while loops" conditional expression commonly contain one or more "counter variables" that have their value changed on every iteration through the loop this allows the conditional expression to tend towards the terminating condition, which when reached, the control exits from the loop.

In the previously discussed example program for counting first n natural numbers the "while" block has a statement that increments the value of '`i`', this is very important (and sometimes forgotten) because if missed the loop executes forever with an end, this is called an infinite loop. Try the "counting the first n natural numbers" program yourself without the statement to increment the value of the counter variable.

```python
#============================================================
# This program adds the first "n" natural numbers
#============================================================

n = int(input("Enter the value of n: "))
```

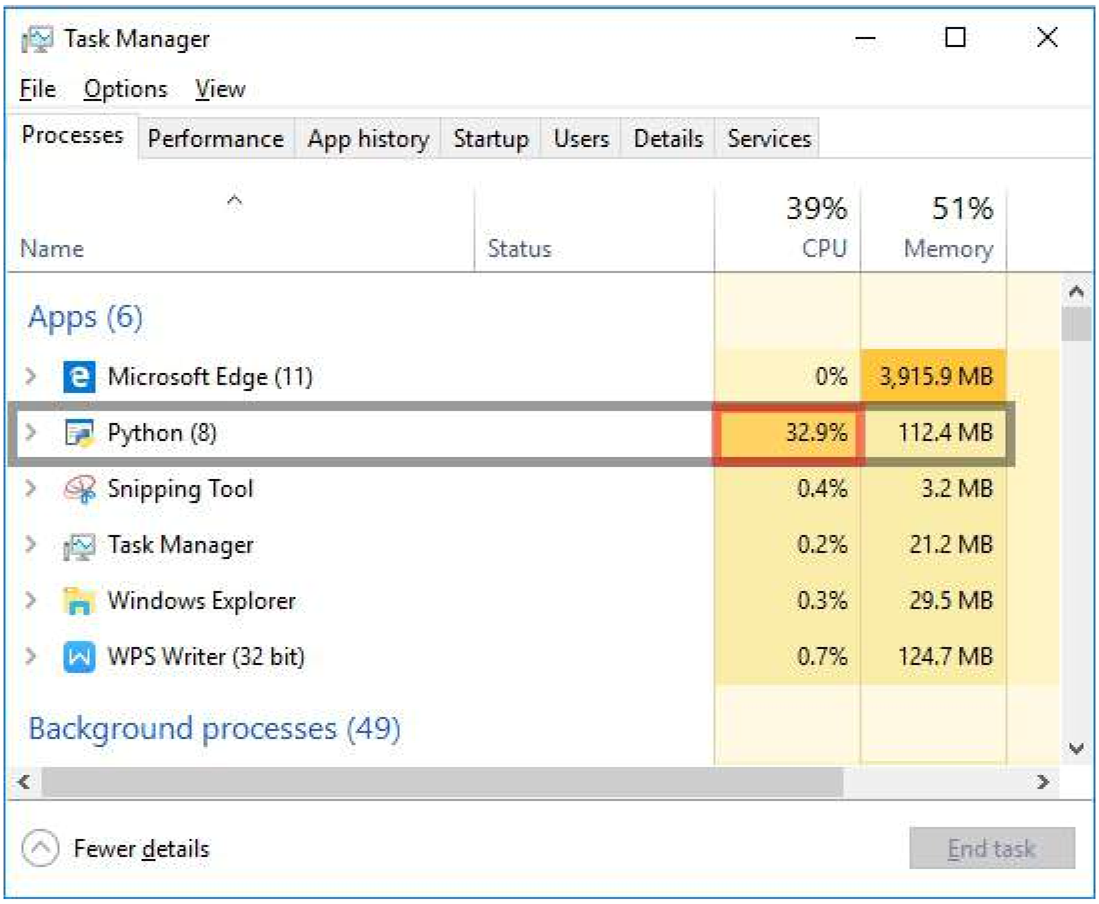
```python

# Initialize sum and counter variables
i = 1
sum = 0

while i <= n:
    sum = sum + i
    i = i + 1     # Update to the counter variable is skipped

print("The sum is: ", sum)

```

When the above program is executed, it print the output given below.



```
Enter the value of n:5
```

As you can see that after the value of ‘n’ s provided, the sum doesn’t get printed, this is because the control is still in the while loop which is an infinite loop. It’s possible to see that the program is still in execution by checking the “task manager”(ctrl + shift + esc on windows).


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	Status	39% CPU	51% Memory
Apps (6)			
Microsoft Edge (11)		0%	3,915.9 MB
Python (8)		32.9%	112.4 MB
Snipping Tool		0.4%	3.2 MB
Task Manager		0.2%	21.2 MB
Windows Explorer		0.3%	29.5 MB
WPS Writer (32 bit)		0.7%	124.7 MB
Background processes (49)			

Lets take a look at some more examples of infinite loops with while loops.

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar  2 15:41:08 2019

@author: Mohamed Khalid
"""

#===============================================================
# Examples of loops that terminate and infinite loops
#===============================================================

# Assignments
var1 = 1
var2 = 2
var3 = 3
```

```

#--------------------------------------- Loop is never executed
while (0 + 1 - var1):
    print(var1)
    var1 += 1

#--------------------------------------------- Terminating loop

while (var1 <= 10):
    print(var1)
    var1 += 1

#--------------------------------------------- Terminating loop
while (var2 >= -10):
    print(var2)
    var1 -= 5

#------------------------------------------------ Infinite loop
while (var3 != 10):
    print(var3)
    var3 += 6

#------------------------------------------------ Infinite loop
while (1):
    print(var1)
```

3.6 Break and Continue statements

Python provides two statements for abnormal termination of loops, the `break` and `continue` statements. Break and continue statements in python are used to change the control flow in a loop. Loop are used to iterate over a certain portion of the code, break statement is used to abruptly exit the loop entirely and continue is used to skip a specific iteration of the loop. Let's discuss more details about the break and continue statement.

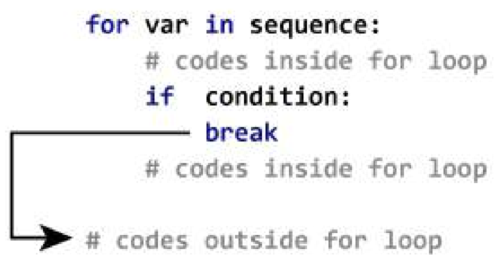
3.6.1 The break statement

The `break` statement when encountered terminates the loop that contains it. When the `break` statement is encountered by the the python interpreter passes the control to the statement immediately after the body of the loop.

In the case of nested loops(loops embedded in other loops) the the `break` statement when encountered, terminates the loop that it was encountered at. As the `break` statement changes the control flow regardless of any condition it is an unconditional branching statement. The syntax of the break statement is given below.

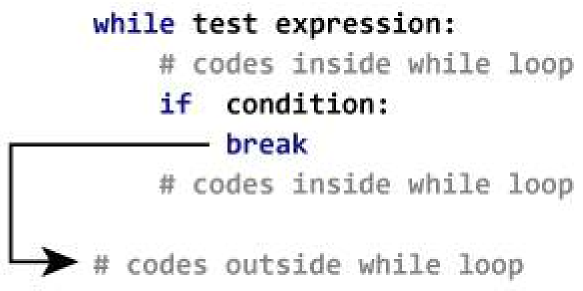
```
break
```

The general code given below illustrates what the `break` statement does when it is encountered in a "for loop".



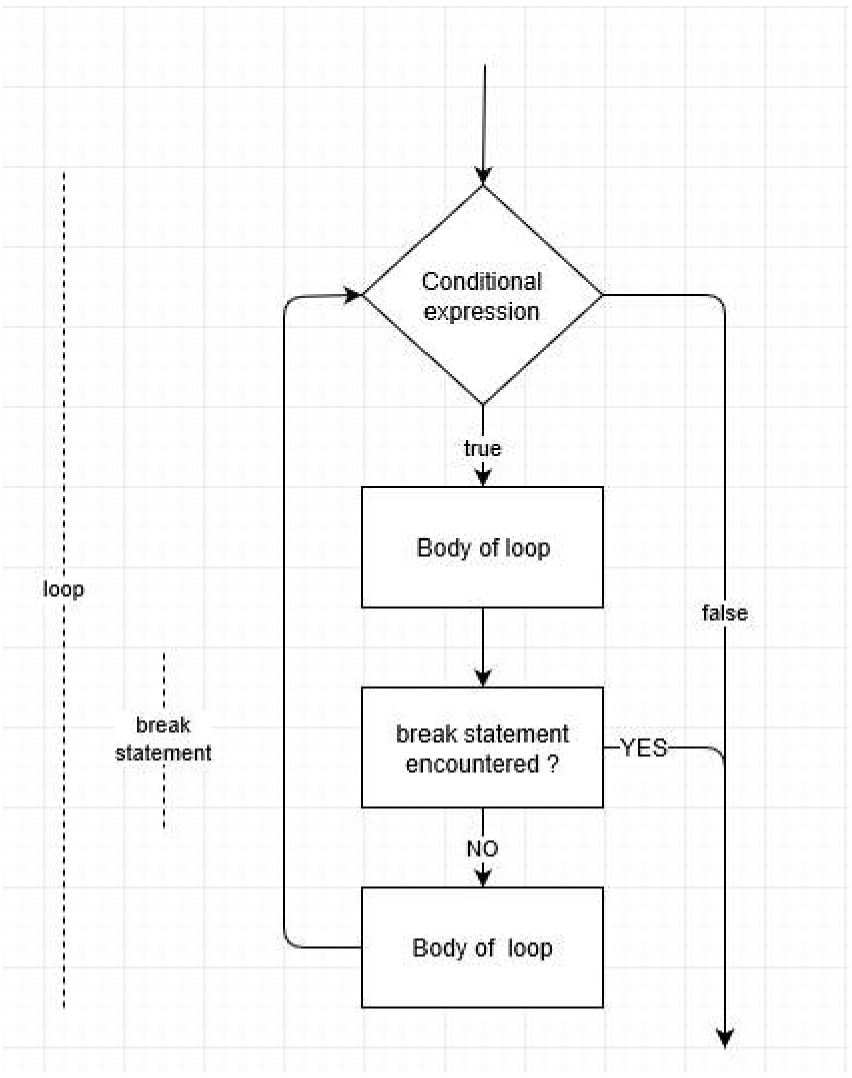
```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

The general code given below illustrates what the `break` statement does when it is encountered in a "while loop".



```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

The flow chart of the `break` statement is shown below.

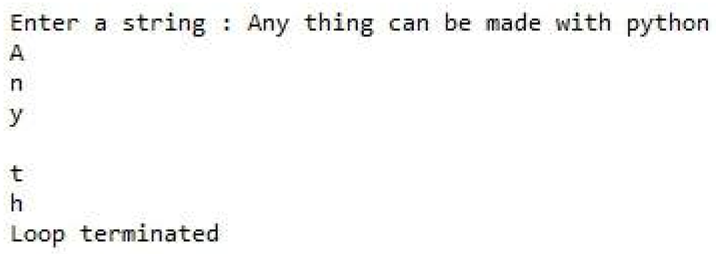


Lets look at an example that uses the `break` statement in a loop. In this python program the loop is abruptly terminated when the a specific letter is found.

```python
#===============================================================
# Print all characters of a string until the first "i" is found
#===============================================================

userString = input("Enter a string : ")

for val in userString:
    if val == "i":
        print("Loop terminated")
        break
    print(val)
else:
    print("The letter i was not found in your string")
```

In the above python script a “for loop” is used to iterate through each character of the the user provided string. On each iteration the current character is checked if it is the letter ‘i’ in the case it is not, then this character is printed, if the character is the letter ‘i’ then the python interpreter encounters the `break` statement which causes the control to exit the loop and passes the control outside the “for loop”(“else” is not executed)

The output of the above program is given below.



```
Enter a string : Any thing can be made with python
A
n
y

t
h
Loop terminated
```

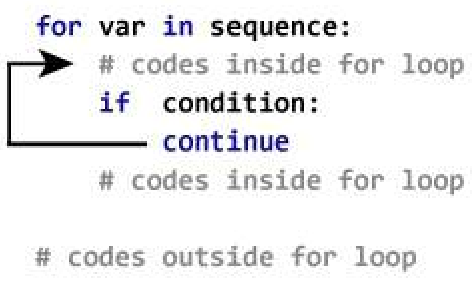
3.6.2 The continue statement

The `continue` statement when encountered abruptly terminates the current iteration loop that contains it. When the `continue` statement is encountered by the the python interpreter passes the control to the statement immediately after the body of the loop.

In the case of nested loops(loops embedded in other loops) the the `continue` statement when encountered, terminates the current iteration of the loop that it was encountered at. As the `continue` statement changes the control flow regardless of any condition it is an unconditional branching statement. The syntax of the `continue` statement is given below

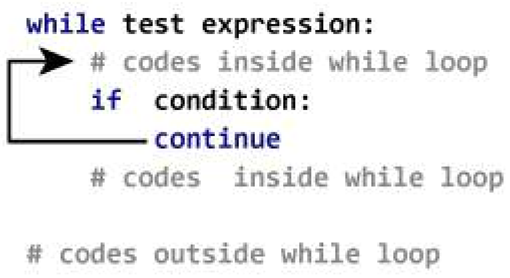
```
continue
```

The general code given below illustrates what the `continue` statement does when it is encountered in a “for loop”.



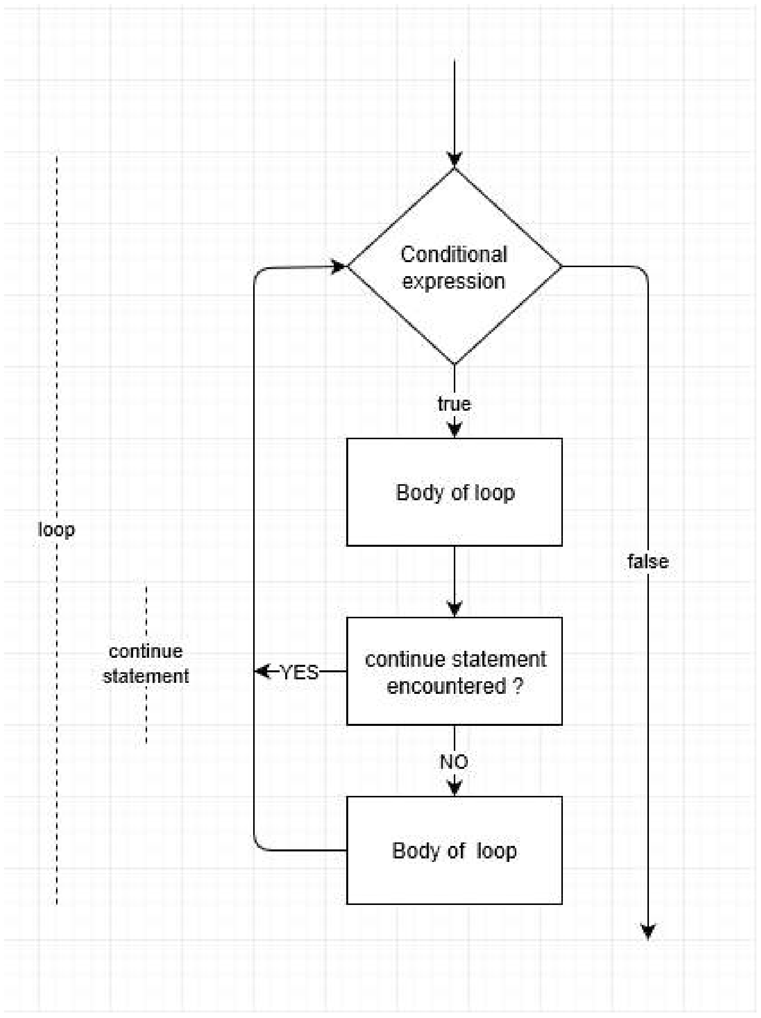
```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

The general code given below illustrates what the `continue` statement does when it is encountered in a "while loop".



```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

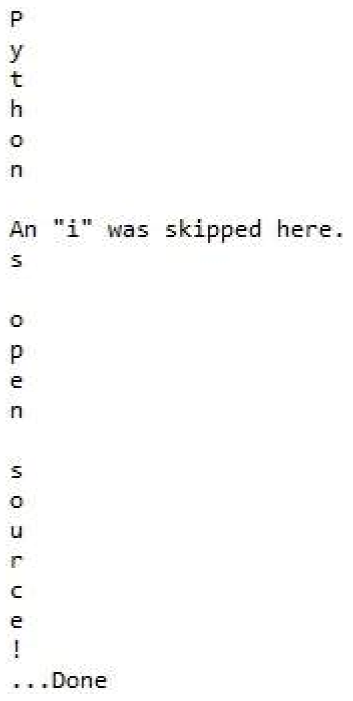
The flowchart of the `continue` statement is shown below.



Lets look at an example that uses the `continue` statement in a loop. In this python program the an iteration that contains a specific letter is skipped.

```python
#================================================================
# Print all characters of a given string except i's
#================================================================

userString = input("Enter a string : ")

for val in userString:
    if val == "i":
        print("An \"i\" was skipped here.")
        continue
    print(val)
else:
    print("...Done")
```

In the above python script a "for loop" is used to iterate through each character of the the user provided string. On each iteration the current character is checked if it is the letter 'i' in the case it is not, then this character is printed, if the character is the letter 'i' then the python interpreter encounters the `continue` statement which causes the control to exit the current iteration of the loop and passes the control to the next iteration of that loop.



```
P
y
t
h
o
n

An "i" was skipped here.
s

o
p
e
n

s
o
u
r
c
e
!
...Done
```

3.7 Pythons pass statement

Python provides the `pass` statement, which basically does not do anything, it is a null statement. It is not completely ignored by the python interpreter like the comments in python are, they simply don't do anything and are mostly used as place holders.

Nothing happens on the execution of a `pass` statement, it's execution results in a no operation(NOP). The syntax of the `pass` statement in python is given below.

```
pass
```

Let's take a look at an example that shows the use of the pass statement in python. Suppose that there is a loop or a function definition who's bodies haven't been implemented yet, python does not allow loops or function definitions to have empty bodies, so instead they can contain a pass statement that serves as a place holder as it is not ignored by the python interpreter nor does it do any operation The example below shows how the pass statement can be used as a place holder in loops.

```python
#===========================================================
# Using the pass statement as a place holder
#===========================================================

# Variable declaration
mySequence = list(range(10))

for item in mySequence:
    pass             # pass is acts as a placeholder for
                    # the functionality to be added later.

```

The pass statement can also be used as a placeholder in function definitions as shown below.

```python
def myFunction():
    pass             # pass is acts as a placeholder for
                    # the function definition to be
                    # added later.

```

The pass statement can also be used as a placeholder in class definitions as shown below.

```python
class myClass:
    pass             # pass is acts as a placeholder for
                    # the class definition to be
                    # added later.

```

3.8 Exercises

1. What possible values can a Boolean expression have?

2. Where does the term Boolean originate?

3. What is an integer equivalent to True in Python?

4. What is the integer equivalent to False in Python?

5. Is the value -16 interpreted as True or False?

6. Given the following definitions:

x, y, z = 3, 5, 7

evaluate the following Boolean expressions:

(a) x == 3

(b) x < y

(c) x >= y

(d) x <= y

(e) x != y - 2

(f) x < 10

(g) x >= 0 and x < 10

(h) x < 0 and x < 10

(i) x >= 0 and x < 2

(j) x < 0 or x < 10

(k) x > 0 or x < 10

(l) x < 0 or x > 10

7. Given the following definitions:

b1, b2, b3, b4 = true, false, x == 3, y < 3

evaluate the following Boolean expressions:

(a) b3

(b) b4

(c) not b1

(d) not b2

(e) not b3

(f) not b4

(g) b1 and b2

(h) b1 or b2

(i) b1 and b3

(j) b1 or b3

(k) b1 and b4

(l) b1 or b4

(m) b2 and b3

(n) b2 or b3

(o) b1 and b2 or b3

(p) b1 or b2 and b3

(q) b1 and b2 and b3

(r) b1 or b2 or b3

(s) not b1 and b2 and b3

(t) not b1 or b2 or b3

(u) not (b1 and b2 and b3)

(v) not (b1 or b2 or b3)

(w) not b1 and not b2 and not b3

(x) not b1 or not b2 or not b3

(y) not (not b1 and not b2 and not b3)

(z) not (not b1 or not b2 or not b3)

8. Express the following Boolean expressions in simpler form; that is, use fewer operators. x is an integer.

(a) not (x == 2)

(b) x < 2 or x == 2

(c) not (x < y)

(d) not (x <= y)

(e) x < 10 and x > 20

(f) x > 10 or x < 20

(g) x != 0

(h) x == 0

9. What is the simplest tautology?

10. What is the simplest contradiction?

11. Write a Python program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print ”OK;” otherwise, do not print anything.

12. Write a Python program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print ”OK;” otherwise, print ”Out of range.”

13. Write a Python program that allows a user to type in an English day of the week (Sunday, Monday, etc.). The program should print the Spanish equivalent, if possible.

14. Consider the following Python code fragment:

```
# i, j, and k are numbers
if i < j:
if j < k:
i = j
else:
j = k
else:
if j > k:
```

```
j = i
else:
i = k
print("i =", i, " j =", j, " k =", k)
```

What will the code print if the variables i, j, and k have the following values?

(a) i is 3, j is 5, and k is 7

(b) i is 3, j is 7, and k is 5

(c) i is 5, j is 3, and k is 7

(d) i is 5, j is 7, and k is 3

(e) i is 7, j is 3, and k is 5

(f) i is 7, j is 5, and k is 3

15. Consider the following Python program that prints one line of text:

```
val = eval(input())
if val < 10:
if val != 5:
print("wow ", end='')
else:
val += 1
else:
if val == 17:
val += 10
else:
print("whoa ", end='')
print(val)
```

What will the program print if the user provides the following input?

(a) 3

(b) 21

(c) 5

(d) 17

(e) -5

16. Write a Python program that requests five integer values from the user. It then prints the maximum and minimum values entered. If the user enters the values 3, 2, 5, 0, and 1, the program would indicate that 5 is the maximum and 0 is the minimum. Your program should handle ties properly; for example, if the user enters 2, 4 2, 3 and 3, the program should report 2 as the minimum and 4 as maximum.

17. Write a Python program that requests five integer values from the user. It then prints one of two things: if any of the values entered are duplicates, it prints "DUPLICATES"; otherwise, it prints "ALL UNIQUE".

18. Write a Python program that ...

19. How many asterisks does the following code fragment print?

```
a = 0

while a < 100:

print('*', end='')

a += 1

print()
```

20. How many asterisks does the following code fragment print?

```
a = 0

while a < 100:

print('*', end='')

print()
```

21. How many asterisks does the following code fragment print?

```
a = 0

while a > 100:

print('*', end='')
```

```
a += 1
print()
```

22. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
b = 0;
while b < 55:
print('*', end='')
b += 1
print()
a += 1
```

23. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
if a % 5 == 0:
+= 1
print()
```

24. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
b = 0
while b < 40:
if (a + b) % 2 == 0:
print('*', end='')
b += 1
print()
```

```
a += 1
```

25. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
b = 0
while b < 100:
c = 0
while c < 100:
print('*', end='')
c++;
b += 1
a += 1
print()
```

26. How many asterisks does the following code fragment print?

```
for a in range(100):
print('*', end='')
print()
```

27. How many asterisks does the following code fragment print?

```
for a in range(20, 100, 5):
print('*', end='')
print()
```

28. How many asterisks does the following code fragment print?

```
for a in range(100, 0, -2):
print('*', end='')
print()
```

29. How many asterisks does the following code fragment print?

```
for a in range(1, 1):
print('*', end='')
print()
```

30. How many asterisks does the following code fragment print?

```
for a in range(-100, 100):
print('*', end='')
print()
```

4. Functions in python

Function syntax has already been introduced in the previous chapters, we have already worked with some of functions from python s standard library like `print()`, `list()`, `tuple()`, `len()`, `id()` and some other functions. The python library has several functions that can be used for common programming tasks.

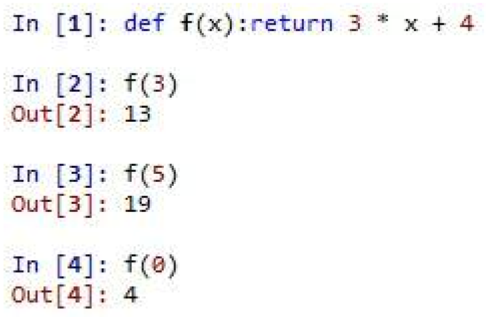
Functions in mathematics are of the form `f(x) = y`. Here function 'f' when given the input 'x' performs operations on 'x' and gives 'y' as the output. An example of function a definition in mathematics is

```
f(x) = 3x + 4
```

From this function definition we can compute the values of `f(x)` for the different values of 'x' as below.

```
f(3) = 13

f(5) = 19

f(0) = 4
```

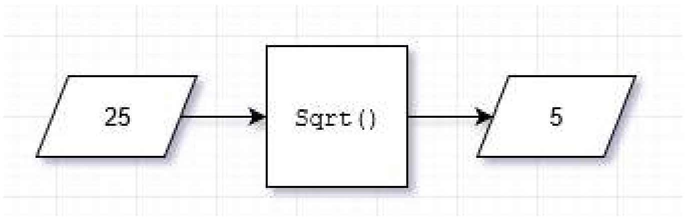
Functions in python is a block of code that is given a name(like the name of above function is 'f'). This block of code can be called any where in the program, when called control is shifted to the body of the called function. The above function 'f' can be written in python as follows.



```
In [1]: def f(x):return 3 * x + 4

In [2]: f(3)
Out[2]: 13

In [3]: f(5)
Out[3]: 19

In [4]: f(0)
Out[4]: 4
```

One example of a function in pythons standard library is the square root function, this function is named `sqrt`. The square root functions takes numeric as parameter and returns a float value, like square root of "25" is '5'.

Functions should be thought of as tools that perform a certain operation, so it's important to know "what a function does" than "how it does it". Thinking of functions this way allows the programmer to see functions as a black box that performs the desired operation. In the case of the square root function it allows the programmer to focus on the task at hand rather than focusing on the algorithm to calculate the square root of a number, like for example to make a program that return the square root of a number provided by the user, the `sqrt()` function can be used to avoid the complex logic of computing the square root of a number. The code example below returns the square root of a number provided by the user by using the `sqrt()` function instead of implementing the logic for square root calculation.



```python
from math import sqrt

# ask user for input
num = eval(input("Enter number: "))

# Compute the square root with sqrt()
# sqrt() is responsible for square root calculation,
# allowing the programmer to focus on whats important
root = sqrt(num);

# print the result
print("Square root of", num, "=", root)
```

Output for the above program is below.

```
Enter number: 25
Square root of 25 = 5.0
```

Here `sqrt(num)` is function invocation or a function call(a call to the `sqrt()` function). A function provides a service to it's the the caller(invoker of the service). The above code that we have to compute square root of a number is the "calling code" or the "client code" to the `sqrt()` function which acts like a service that can be invoked

```
root = sqrt(num)
```

The above statement invokes `sqrt()` passing it the value of num. The expression `sqrt(num)` the square root of the value of the variable "num".

Unlike the small collection of functions(like `type, str, int, range, id`) that are always available in python programs, the `sqrt()` function is available in the "math" module and can not be used unless the math module is imported, hence the purpose of the "import" statement(the first line of code)

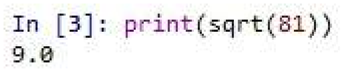
```
from math import sqrt
```

The above code imports the `sqrt()` function from the "math" module. A module is a collection of code that can be imported and used in other python programs. The "math" module is very rich in the functions it provides to perform math operations, it provides functions for trigonometric, logarithmic and several other math operations.

A function has a name followed by a set of parentheses which contain the information that the function needs to perform it's task.

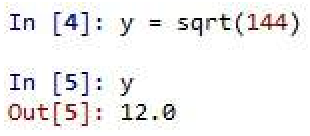
```
Sqrt( num ) # sqrt needs a parameter to return the square root of
```

`num` is the value that the `sqrt()` function operates on. The values that are passed to a function are called the parameters or arguments of the function. Functions after they perform the computations it was defined to perform can return a value back to the caller(client invoking it) like in the case of the `sqrt()` function which returns the square root of the parameter passed as a float type to its invoker. This value that is returned to the client can be used by the client just like any other value.



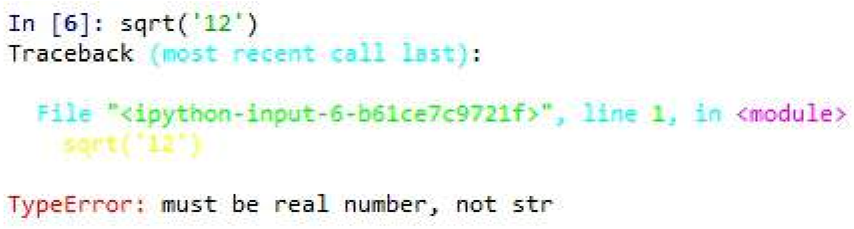
```
In [3]: print(sqrt(81))
9.0
```

The `print()` function above can use the value returned by the `sqrt()` function.



```
In [4]: y = sqrt(144)

In [5]: y
Out[5]: 12.0
```

Like in the above assignment statement, the value returned by a function can be assigned to a name. When the client code attempts to pass an argument to the function it's invoking that has a type that is different then what is expected by the function, then the interpreter raises an error.



```
In [6]: sqrt('12')
Traceback (most recent call last):

  File "<ipython-input-6-b61ce7c9721f>", line 1, in <module>
    sqrt('12')

TypeError: must be real number, not str
```

A function can be called any number of times anywhere in the program. Remember to he the client of a function, the function is like a labeled black box, the knows client knows what the function bing called does and is not concerned with how the task is performed. All functions can be treated like black boxes. A service that a function provides can be used without being concerned of the internal details.

The only way a behaviour of a function can be altered is through the functions parameters(the arguments that are passed to a function can be different). When immutable types(int, float, string, tuple) are passed as arguments to a function, a copy of the objects are passed instead of a reference to the original objects, therefore immutable objects when passed to a function as an argument can not be changed by the called function. In the case that the arguments passed are mutable objects a reference to those objects are passed instead of a new copy of those objects, therefore mutable objects when passed to a function as an argument can be changed by the invoked function.

As we already seen before functions can take more than one parameter like the `range()` function that can take one, two or three parameters.

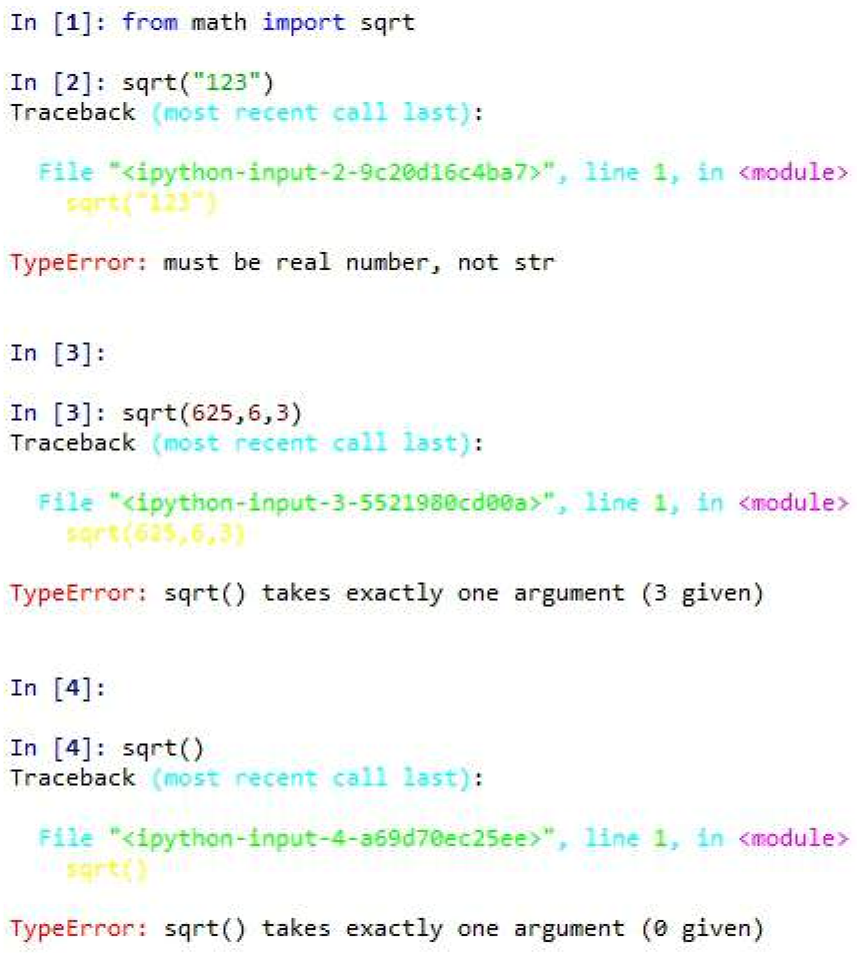
From the view of the client/caller, a function has three parts. Lets discuss eac of them in brief.

Name

A function's name is the handle that is used to identify the code to be executed when it is invoked. The same rules that apply to variable names apply to function names. Function names, like variable names are also identifiers.

Parameters

Functions are called with zero or more parameters. The types of each argument passed in the invocation of a function must be the same as the types of parameters in the function definition. If more are or less parameters are passed than what is expected the interpreter raises an error. Some examples of this these two kinds of errors are illustrated in the shell execution below.

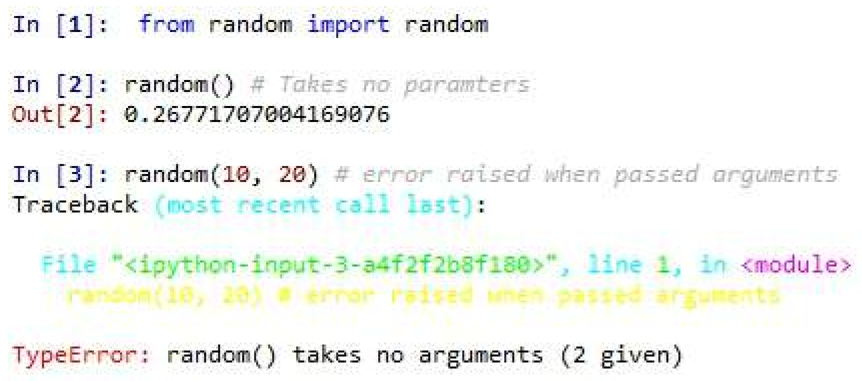


```
In [1]: from math import sqrt

In [2]: sqrt("123")
Traceback (most recent call last):

  File "<ipython-input-2-9c20d16c4ba7>", line 1, in <module>
    sqrt("123")

TypeError: must be real number, not str


In [3]:

In [3]: sqrt(625,6,3)
Traceback (most recent call last):

  File "<ipython-input-3-5521980cd00a>", line 1, in <module>
    sqrt(625,6,3)

TypeError: sqrt() takes exactly one argument (3 given)


In [4]:

In [4]: sqrt()
Traceback (most recent call last):

  File "<ipython-input-4-a69d70ec25ee>", line 1, in <module>
    sqrt()

TypeError: sqrt() takes exactly one argument (0 given)
```

Return type

A function always returns a value to it's caller. The client code should be compatible with the type of the object returned by called function. The return type and the parameter types are not related. Even in the case that the function does not specifically return any value, a special "none" value is return to the client.

There are functions that take no parameter like the `random()` function. The `random()` function returns a random floating point value and it belongs to the "random" module. Again if the arguments passed does not match with the number of parameters and the type of each parameter in the function definition then an error is raised by the interpreter. You can see in the below code that if a parameter is passed to the `random()` function an error is raised.



```
In [1]:  from random import random

In [2]: random() # Takes no paramters
Out[2]: 0.26771707004169076

In [3]: random(10, 20) # error raised when passed arguments
Traceback (most recent call last):

  File "<ipython-input-3-a4f2f2b8f180>", line 1, in <module>
    random(10, 20) # error raised when passed arguments

TypeError: random() takes no arguments (2 given)
```

One of the ways that functions can be categorized is as,

- **Functions without side effects:** These types of functions just perform the required task and return the result to the client caller and that is all it does, it makes no other changes like printing to the display, changing the variables of the caller, etc. This type of functions always return a result(not none) to the invoking client
- **Functions with side effects:** These types of variables perform the required task which may also require to change it's environment like printing to the display or changing the value of the caller, etc. This type of functions may sometimes return a special value "none" to the invoking client.

4.1 Standard mathematical functions

The "math" module provides almost all of the functionality of a scientific a scientific calculator. Some of he function of the "math" module a re listed below.

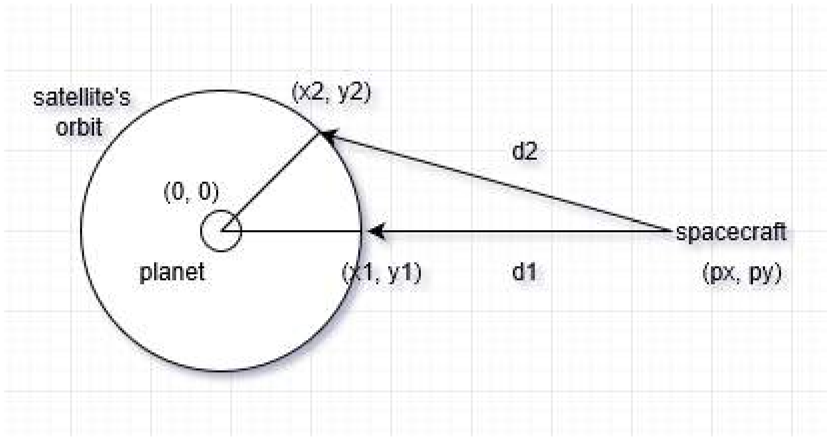
Function	description
`sqrt`	Computes the square root of a number.
`exp`	Computes e raised a power
`log`	Computes the natural logarithm of a number.
`log10`	Computes the common logarithm of a number.
`cos`	Computes the cosine of a value specified in radians: cos(x) = cos x; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyperbolic sine, and hyperbolic tangent
`pow`	Raises one number to a power of another.
`degrees`	Converts a value in radians to degrees.

`radians`	Converts a value in degrees to radians.
`fabs`	Computes the absolute value of a number.

The parameters passed to the function by the calling client code is called the actual parameters(arguments) and the parameters that are specified in the function definition are the formal parameters. When a function is invoked the first actual parameter is assigned to the first formal parameter, then the second actual parameter is assigned to the second formal parameter, so the assignment of actual parameters to formal parameters happens from left to right. The order of passing the assignment matters, for example calling the `pow()` function as `pow(10,2)` returns 100 but calling it as `pow(2, 10)` returns 1024.

Lets try to solve a trigonometry problem with the functions provided by the math module. The problem is as follows.

Suppose that there is a spacecraft at some distance from a plane. A satellite orbits this planet, the problem is to find how much farther a way the satellite would be away from the spacecraft if the the satellite moves 10 degrees along it's orbital path. The spacecraft is on the plane of the satellites orbit.



Let the center of the planet be the origin(0, 0) of the coordinate system, this is also the center of the satellites orbit. In this problem the spacecraft is located at the point (p_x, p_y), the satellite is initially at point (x_1, y_1) then moves to point (x_2 , y_2). Here the the points (x_1, y_1) and (p_x, p_y) are given and the problem is to find the the difference between d_1 and d_2. This problem can be solved in two steps as follows.

Step 1

Calculate the point (x_1, y_1). For any given point (x_1, y_1) a rotation of theta degrees gives a new point (x_2, $y_{2)}$, where

$$\begin{aligned} x_2 &= x_1 \cos\theta - y_1 \sin\theta \\ y_2 &= x_1 \sin\theta + y_1 \cos\theta \end{aligned}$$

Step 2

The distance d_1 between the two points (p_x, p_y) and (x_1, y_1) is given by,

$$d_1 = \sqrt{(x_1 - p_x)^2 + (y_1 - p_y)^2}$$

The same way, distance d_2 between the two points (p_x, p_y) and (x_2, y_2) is given by,

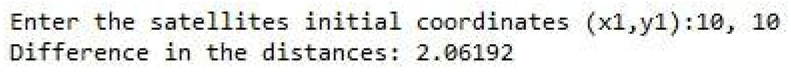
$$d_2 = \sqrt{(x_2 - p_x)^2 + (y_2 - p_y)^2}$$

The python program below solves the orbital distance problem for the spacecraft at location (100, 0) and user provided starting location for the satellite. You can see that the standard math problem was used in the program to com at the solution.

```
#====================================================================
# Python program for the orbital distance problem
#====================================================================
from math import sqrt, cos, sin, pi

# The orbiting point is (x,y)
# A fixed point is always (100, 0),
# (p_x, p_y). Change them as necessary.
p_x = 100
p_y = 0

# number of radians in 10 degrees
radians = 10 * pi/180

# The cosine and sine of 10 degrees
cos10 = cos(radians)
sin10 = sin(radians)

# ask the satellites starting point from the user
x1, y1 = eval(input("Enter the satellites initial coordinates (x1,y1):"))

# calculate the initial distance d1
d1 = sqrt((p_x - x1)**2 + (p_y - y1)**2)

```

```python
# calculate the new location (x2, y2) of the satellite after movement
x2 = x1*cos10 - y1*sin10

y2 = x1*sin10 + y1*cos10

# Compute the new distance d2
d2 = sqrt((p_x - x2)**2 + (p_y - y2)**2)

# Print the difference in the distances
print("Difference in the distances: %.5f" % (d2 - d1))

```

The output when the above program is executed for the starting location (10, 10) is,



```
Enter the satellites initial coordinates (x1,y1):10, 10
Difference in the distances: 2.06192
```

4.2 Time functions

Pythons "time" module provides a lot of functions that involve time. In this section we will take a look at two of the functions that are provided by this module the `clock()` and `sleep()` functions.

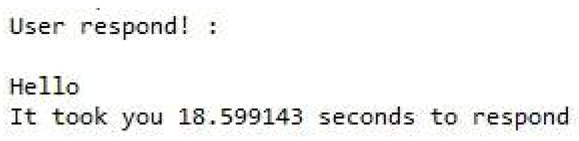
The clock function allows measuring time in seconds and it can be used anywhere in a program allowing the measure of time at any part of a program. The clock function works differently based on the operating system that it is being executed on. On unix like operating systems like linux and mac OS, the clock() function returns the number number of seconds elapsed from the start of the programs execution. On Microsoft windows the clock() function returns the number of seconds elapsed from the previous call to the clock() function. The return the number of seconds elapsed as a floating point number. The difference between the returned values of two clock function calls can be used to find the time period between the two calls.

The below program returns the amount of time it took the user to respond.

```python
#=================================================================
# This program prints the time it took the user to respond in seconds
#=================================================================
from time import clock # import the clock method
```

```python

print("User respond! : ")
startTime = clock()
input()
endTime = clock()
print("It took you %f seconds to respond " % (endTime - startTime))


```

An output a very slow user might get is,



```
User respond! :

Hello
It took you 18.599143 seconds to respond
```

Here is a another program that displays the amount of time it took to add the first ten million natural numbers.

```python
#=====================================================================
# Time taken to add the first million natural numbers
#=====================================================================
from time import clock # import the clock method

sum = 0 # initialize sum as 0

print("started adding")
startTime = clock()

for i in range(10000000):
    sum += i

endTime = clock()
print("It took %f seconds " % (endTime - startTime))

```

The output of this program is given below.

```
started adding
It took 1.363418 seconds
```

The sleep function can be used to suspend a program for a period of time in seconds. The number of seconds to suspend the program for is passed as an argument to the sleep function. The below program counts down to zero from three with one second intervals.

```
#======================================================================
# Three second count down timer!
#======================================================================
from time import sleep    # Import the sleep method

timer = int(3)            # Initialize the timer to a predefined value

print("Count down starts... %i" %timer)

while timer > 0:
    sleep(1)
    timer -= 1
    print(timer)

sleep(1)

```

The proves to be very useful when in comes to controlling the speed in graphical animations. This program when ran produces the following output.

```
count down starts... 3
2
1
0
```

4.3 Random functions

The package random provides some functions that deal with randomness. Applications that require the behaviour of randomness need these kind of functions. Almost all random number generating algorithms produce “pseudo random” numbers which means they are not truly random. pseudo random number generators when repeated long enough produce results that start repeating the exact same sequence of numbers. This not a serious problem as the length of this sequence is long enough to be considered safe for almost all practical applications.

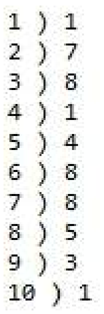
Pythons “random” module consists of a number of functions that enable the use of pseudo random numbers in programs. Few of the commonly used functions of this module are described below.

Function	Description
`random()`	Returns a pseudo random floating-point number x in the range `0 <= x < 1`
`randrange()`	Returns a pseudo random integer value within a specified range.
`seed()`	Sets the random number seed.

The `seed()` sets the value from which the sequence of the pseudo random numbers are generated. Every time the functions `random()` and `randrange()` are called they return the next value in the sequence of pseudo random values. The program below, prints ten random numbers in the range of one to ten.

```python
from random import randrange, seed

seed(10)

for i in range(10):
    print(i+1,")",randrange(1, 10))

```

This program produces the output shown below.

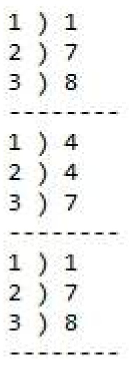


```
1 ) 1
2 ) 7
3 ) 8
4 ) 1
5 ) 4
6 ) 8
7 ) 8
8 ) 5
9 ) 3
10 ) 1
```

The function seed() sets the “seed number” that is used to generate the pseudo random number sequence and as there is a dependency if the seed value is changed then then sequence of pseudo random random numbers will also change. This dependency is in one way a problem because if the above program is run again it produces the exact same sequence of random numbers. This is illustrated in the example program shown below.

```python
from random import randrange, seed

seed(10) # The seed is 10

for i in range(3):
    print(i + 1, ")", randrange(1, 10))
print("----------")

seed(14) # Change the seed to something else

for i in range(3):
    print(i + 1, ")", randrange(1, 10))
print("----------")

seed(3)

for i in range(3):
    print(i + 1, ")", randrange(1, 10))
print("----------")
```

From this program you can see that the same sequence of pseudo random numbers are generated for the same value of the seed. Having constant value of the seed isn't a safe way to generate random numbers, but it can be useful while testing a program. When the value of the seed is not specifically mentioned, then the system time determines the seed. This allows the pseudo random numbers generated to have a more random behaviour. The out put of the above program are shown below.

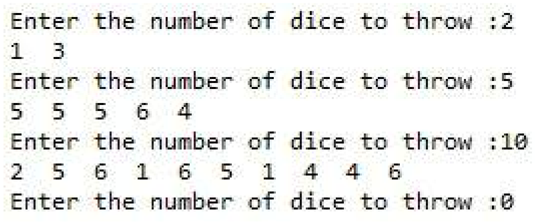


```
1 ) 1
2 ) 7
3 ) 8
--------
1 ) 4
2 ) 4
3 ) 7
--------
1 ) 1
2 ) 7
3 ) 8
--------
```

Now lets write a program that simulates the rolling of a six sided die. A die has six sides labeled one to six with one of the sides that can land on top when the die is rolled. We want to user to provide information on the number of dice to be thrown. The random number generated should lie between zero and six, so the randrange(1, 6) will be used for this, and it will be called for each die thrown.Lastly this entire setup should be placed in a loop that continue indefinitely until the user provides the number zero as input.

```python
from random import randrange

# skipping the seed setup

sides = 7

while 1:
    diceCount = int(input("Enter the number of dice to throw :"))
    for i in range(diceCount):
        dieValue = randrange(1,sides)
        print("%i " % (dieValue), end = " " )

```

An output for the above program is given below.



```
Enter the number of dice to throw :2
1  3
Enter the number of dice to throw :5
5  5  5  6  4
Enter the number of dice to throw :10
2  5  6  1  6  5  1  4  4  6
Enter the number of dice to throw :0
```

4.4 Why write your own functions

All the programs that we have seen till now where very small and come no where near the size of practical applications. When the programs size increase the complexity of the program increases as well and it starts to get difficult to manage this complexity. Programs like the ones we have gone through are "monolithic code", the program execution flows from start to finish, the these types of programs are like one big block of code,when the size of a monolithic program grows it's complexity starts to get out of hand. The solution to this is to divide the code into several almost independent smaller pieces with very specific roles. This division of a programs complexity into smaller pieces can be done using functions. A program can be made up of multiple functions, each one having the task of addressing a part of the problem and when used together solves the problem as a whole.

One big and complex monolithic code is avoided for several reasons.

- **Difficulty writing correctly:** When writing any statement, the details of the entire code must be considered. This becomes unmanageable for programmers beyond a certain point of complexity.
- **Difficulty debugging:** The larger a code block is, the harder it is to find the source of an error or an exception in that block of code. The effect of an erroneous statement might not be know until later in the program a correct statement uses the result of the erroneous statement.
- **Difficulty in modification:** Before a code sequence can be modified to achieve a different result, this code sequence needs to be understood, which becomes a very heavy task as the size of code sequence increases.

Our code can be made more manageable by dividing our code into parts and rewriting these parts of our code as functions, calling them in the program when and as needed. Using this divide and conquer strategy, a complicated code block can now be be broken down into simpler code blocks each being handled by a function. The programmer can now accomplish the same original task by delegating tasks to functions, this way a results in more manageable code blocks that are easy to read, write, debug and modify. Besides these advantages of structuring the code, functions provide several other advantages like reusing same code block for the same kind of task(reusability). Functions that provides similar functionality can be bundled together into reusable parts.

Although it's time saving to use library functions, it is sometimes necessary to write our own functions for custom behaviour. Fortunately python allows programmers to create their own custom functions and use them.

If the purpose of the custom functions are general enough and written in a proper manner then this function can even be used in other programs. In the following sections we'll take the first steps in writing functions in python.

4.5 Functions basics

Every python function has

1. **Function definition:** A functions definition describes every detail about that function including how many parameters and what type of parameters it takes, what the function does and also what the function returns.
2. **Function invocation(call):** After a function is defined, it can be called from anywhere in the program by passing the appropriate parameters to the function. A function can have only one definition but it can be invoked any number of times.

A function definition has three parts, they are listed below.

- **Name:** The name of a function is what is used to identify the block of code that this function contains. Function names are also identifiers, so they follow all the rules of python identifiers. One of the best ways to name a function is to give it the name of the task it is designed to do.
- **Parameters:** The parameters that the function accepts from callers is specified in the function definition. They appear with a set of parentheses(( )) in a comma(,) separated list. The list of parameters can be left empty if the functions definition does not specify parameters to passed, these types of function do not require information from the caller to perform the task it was designed to carry out.
- **Body:** The block of indented statements that follow after the functions header is the functions body. This is what gets executed when the function call is made. If the function is required to produce a result and return it to the client, the `return` statement can be used to return the result back to the caller.

The syntax of a function definition is given below.

```
def <function_name>(<parameter list>):

    """<docstring>"""

    <statement(s)>
```

The above function definition syntax has the following the components.

- `def` is the keyword that signifies the start of the header of a function
- `<function_name>` is the unique name that identifies the the body of the function
- `<parameter list>` is used to pass arguments to the function. They are optional.

- “`:`” signifies the end of the function header
- `<"""docstring""">` is the documentation string that can be used to describe the purpose and functionality of the function
- `<statement(s)>` are the valid python statements that make up the body of the function. All the statements in the function body have the same indentation usually four spaces(“   ”) are used.
- A `return` statement is optional and it is used to return the result back to the caller if any

The example code below illustrates a simple function definition and calling this defined function.

```python
def greet(name):
    """This function prints a hello and doesn't return anything"""
    print("Hello " + name)

greet("khalid")

```

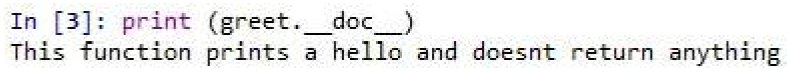
In the above code example a function “greet” is defined that takes a single string(name) as a parameter and prints a specified string onto the prompt. Then the defined function is immediately called after it’s definition, this executes the body of the greet function with the passed arguments.

4.5.1 Docstring

The first line after the function header optionally has a string called the “docstring” that is sort for “document string”. The “docstring” is used for explaining what the function does.

If you have already programmed in other languages you would be aware how programmers tend to forget what their code does, so it’s always a best practice to document your code. Even if you may not use the documentation to refresh your memory on what the function does, it can help others who will be working with your code.

The docstring of a function can be accessed with “.__doc__” after a function name. So if the following code is entered into the python shell it returns,



```
In [3]: print (greet.__doc__)
This function prints a hello and doesnt return anything
```

4.5.2 The return statement

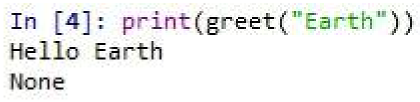
The return statement for functions is like how the break statement is for loops in a way that they both takes the control out of the block they were encountered at. The return statement exits the function, passes the control back to the place it was called from the calling code additionally the return statement contains followed by an expression list that is evaluated and returned back to the caller.

The syntax of the return statement is given below.

```
return <expression_list>
```

Here the expression in the `return` statement is evaluated and returned back to the client. In the case that there is no expression provided or the `return` statement itself is skipped then a special object `none` is returned back to the calling client.

For example try the following code in the python shell. Print the value that is return by the greet function.



```
In [4]: print(greet("Earth"))
Hello Earth
None
```

In the above call to print, which is passed a call to the greet function as the parameter. First the greet method is called which prints to the prompt then the greet method returns a “none” object back to the print function, the print function then prints the “none” object

Lets take a look at an example that illustrates the `return` statement. The program below defines and uses a function that returns the absolute value of an number that’s passed to it. You can see how the return evaluates the expression given to it and returns the result as well as the control back to the caller. Remember that an expression is any combination of values, variables, operators, function calls. Expressions always return an object. If there is no return statement in a function the execution of the function ends when the last statement of the functions code block has been executed and the “none” object as well as the control is returned back to the caller.

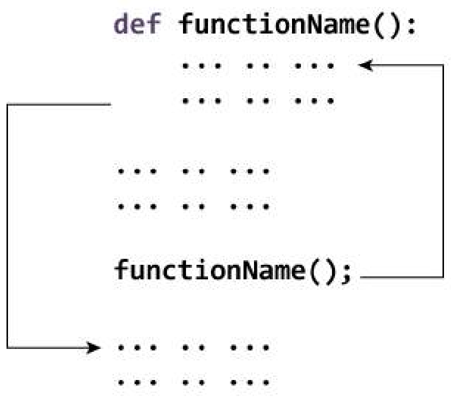
```
#===============================================================
# Evaluate absolute value of a number
#===============================================================
def absolute_value(num):
    """This function returns the absolute
    value of the number passed as an argument"""

    if num >= 0:
        return num
    else:
        return (-num)

userInput = eval(input("Enter a number: ")) # evaluate users input

print("It's absolute value is " + str(absolute_value(userInput)))

```

The output for the above program for the value “-56” is shown below.

```
Enter a number: -56
It's absolute value is 56
```

4.5.3 Flow of control at function call and return

The below figure illustrates the control flow when a function call is made and when the the execution of the functions code block completes.



```
def functionName():
    ... .. ...
    ... .. ...

... .. ...
... .. ...

functionName();

... .. ...
... .. ...
```

When a function call is made the control shifts to the body of the called function, the only two ways that control can exit from a function is by the return statement or by executing the last statement in the body of the function.

4.5.4 The scope and lifetime of variables

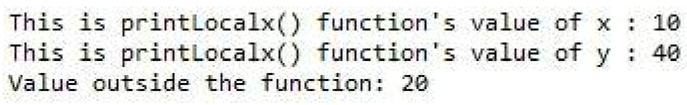
Scope of a variable in a program is the part of the program that can recognize and use the object referred to by that variable. Parameters of a function are considered to be local to that function(local variables) and so they are in the scope of that functions code block but outside the scope of the rest of the program. A functions local variables can’t be used by the parts of the program outside that functions code block hence they have local scope.

Lifetime of a variable is the period through which the variable exists in the memory. The lifetime of variables inside functions(i.e local variables) is as long as the function executes. After the function completes it’s execution all it’s variables are deleted and their associated objects are garbage collected, and because of this functions don’t have any memory between two function calls, they don’t know what the value of local variable was in the previous call.

The example code below illustrates the scope of a variable inside a function.

```python
1  #=======================================================================
2  # This program illustrates the scope of a variable inside a function
3  #=======================================================================
4  def printLocalx():
5      x = 10     # Local variable, scope lines 2 - 4
6      print("This is printLocalx() function's value of x: ", x)
7      print("This is printLocalx() function's value of y: ", y)
8      # Local variable x is deleted once the control flow reaches this point
9
10 x = 20         # Global variable, scope lines 1 - 13
11 y = 40         # Global variable, scope lines 1 - 13
12 printLocalx()
13 print("Value outside the function:", x)
14
```

The above program produces the below output when executed. Observe that even though the the global variable x is in the scope of `printLocalx()` function, the function prefers the value of it’s local variable when there is a match.



```
This is printLocalx() function's value of x : 10
This is printLocalx() function's value of y : 40
Value outside the function: 20
```

Even though function `printLocalx()` changes the value of the the variable x, it does not affect the global variable `x`. This is because variable `x` inside the `printLocalx()` function is different from the one outside of it, so remember that they are two different variables with different scopes.

In Python the `global` keyword allows modification variables that are outside the current scope. The can be used to create global variables and modify it in the local context.

Things to remember when using the global keyword.

1. A variable is by default a local to a function when it is declared inside the function
2. A variable is by default a global variable when its declared outside a function. It makes no difference using the global keyword on variables defined outside functions.
3. The global keyword can be used inside functions to read and write global variables inside functions.

Lets take a look at some examples that illustrate the use and working of the global keyword.

```python
x = 20        # x is a global variable

def printx():
    print(x) # Since there is no local variable x, the global variable x
             # can be accessed

printx()

```

This example code produces the below output.

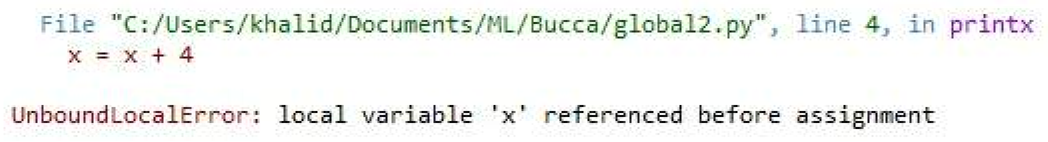
```
20
```

Here, notice that as the function `printx()` doesn't have a local variable 'x', it can read the value of the global variable 'x', but the interpreter raises an error when we try to write to the same variable.

```python
x = 20        # x is a global variable

```

```
def printx():
    x = x + 4 # Try using the value of a global variable.
              # Local variable x is referenced before
              # it's assignment
    print(x)

printx()

```

When the above program is ran it raises the error below,because the value of a global variable is being modified in the function.without using the global keyword it is only possible to read the value of a global variable inside a function

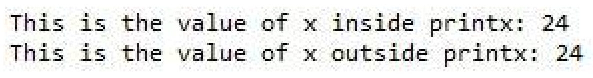


```
  File "C:/Users/khalid/Documents/ML/Bucca/global2.py", line 4, in printx
    x = x + 4

UnboundLocalError: local variable 'x' referenced before assignment
```

The `global` keyword can be used to overcome this kind of a problem where the modifications to global variables from within a function is required. The example below shows how a global keyword can be used to modify a global variable from within a function. To modify a global variable from inside a function this variable has to first to be declared as `global` inside the function it needs to be used in. This is illustrated in the example below.

```
x = 20        # x is a global variable

def printx():
    global x  # Now the global variable x can be modified
    x = x + 4 # Try using the value of the global variable.
             # you'll see there is no error raised this time
    print("This is the value of x inside function printx(): %i" %x)

printx()
print("This is the value of x outside function printx(): %i" %x)

```

The output of the above program is given below. notice that the value of the variable ‘x’ is the same inside and outside the `printx()` function. This is because variable ‘x’ inside the `printx()` function now refers to the same object as the variable ‘x’ outside this function.

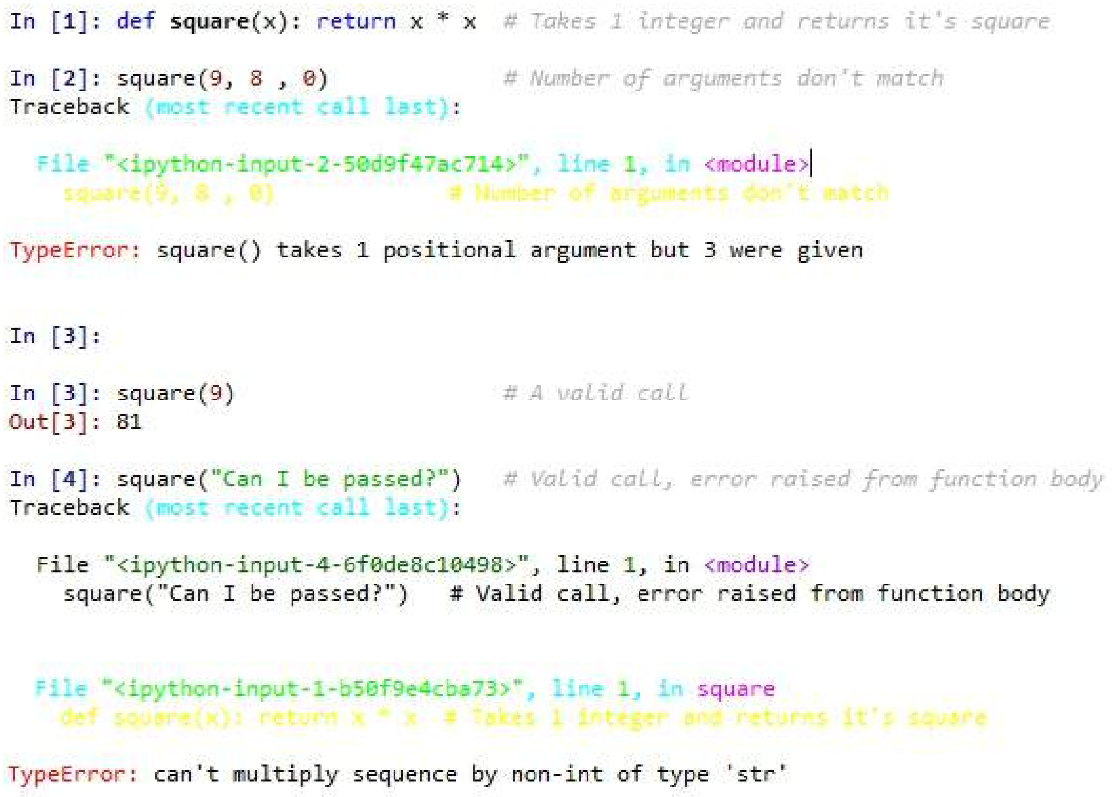


```
This is the value of x inside printx: 24
This is the value of x outside printx: 24
```

More on the global and local scopes will be discussed in the following sections.

4.5.5 Arguments

Functions may take any number of parameters(No upper or lower bound) to perform the task that it was designed for. Before calling a user defined function, this function must be defined and it’s function definition must specify the parameters it needs. When a function is called by a client code, the client caller must provide the same number of arguments as specified in the function definition. if there is any mismatch in the number of arguments passed then what was expected then the python interpreter raises an error. The examples below executed in the python shell show the errors raised when there are issues with the arguments passed to a function call



```
In [1]: def square(x): return x * x  # Takes 1 integer and returns it's square

In [2]: square(9, 8 , 0)            # Number of arguments don't match
Traceback (most recent call last):

  File "<ipython-input-2-50d9f47ac714>", line 1, in <module>
    square(9, 8 , 0)            # Number of arguments don't match

TypeError: square() takes 1 positional argument but 3 were given


In [3]:

In [3]: square(9)                   # A valid call
Out[3]: 81

In [4]: square("Can I be passed?")  # Valid call, error raised from function body
Traceback (most recent call last):

  File "<ipython-input-4-6f0de8c10498>", line 1, in <module>
    square("Can I be passed?")  # Valid call, error raised from function body


  File "<ipython-input-1-b50f9e4cba73>", line 1, in square
    def square(x): return x * x  # Takes 1 integer and returns it's square

TypeError: can't multiply sequence by non-int of type 'str'
```

Observe that the number of arguments to pass are compulsory but the types of the arguments are not. Although passing an argument of a different type than what is expected would most likely end up raising an error.

Python also allows it's users to define functions that take a variable number of parameters. Functions that take a variable number of parameters can be defined with the following types of arguments. We'll discuss each of them in the following section.

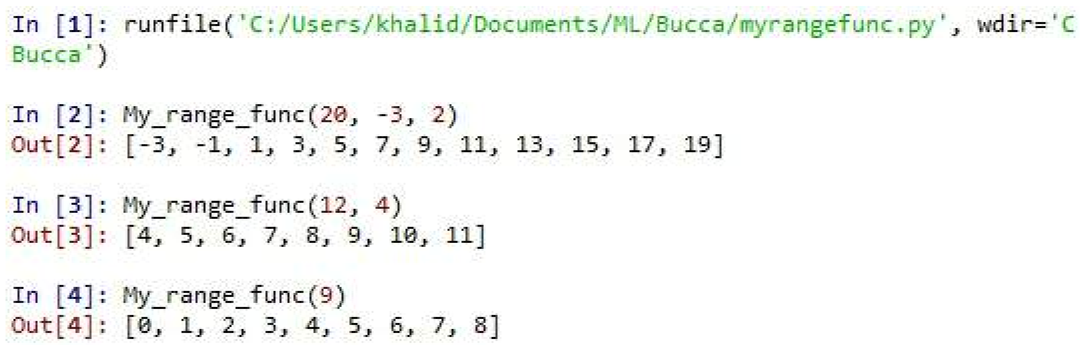
1. Default arguments
2. Keyword arguments
3. Arbitrary arguments

Default and non-default arguments

Python allows to assign default values for function parameters A default value for a parameter can be given with the "=" operator. When a parameter is assigned a default value, its argument can be skipped in the function call in which case the default value provided in the function definition gets assigned instead. In the example below we create our own range function that in turn calls python's standard range function. This example illustrates python's default arguments.

```python
def My_range_func(stop, start = 0, step = 1):
    """
    This function in turn calls
    the standard range function
    and returns it's result as
    a list
    """

    li = list(range(start, stop, step))
    return li

```

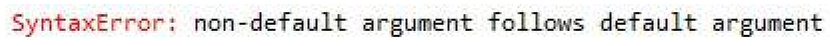
Now when this program is executed, try making calls to this function with different parameters from the python shell. Observe that the function call can be made with one, two or three parameters, this is because the second and third arguments are default arguments are default arguments and when their values are not provided they assume their default values. Here it is necessary to pass a value for parameter `stop` and failing to do so would raise an error. For illustration there are some function calls to the `My_range_func()` given below. These function calls where made from the python shell.



```
In [1]: runfile('C:/Users/khalid/Documents/ML/Bucca/myrangefunc.py', wdir='C
Bucca')

In [2]: My_range_func(20, -3, 2)
Out[2]: [-3, -1, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

In [3]: My_range_func(12, 4)
Out[3]: [4, 5, 6, 7, 8, 9, 10, 11]

In [4]: My_range_func(9)
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Here the parameter `start` does not have a default value and is mandatory to provide in a call. Default values `0` and `1` are provided for parameters `start` and `step` and so they're value can be skipped in a function call but if provided they overwrite the default values. Any number of parameters in a function can have a default value but it must be made sure that all default values are pushed to the right of the parameter list. This is needed as without it there would be difficulty differentiating between default and compulsory arguments in a function call. To ensure this the python interpreter raises an error when it encounters a function header that doesnt have all default parameters pushed to the right. An example is given below.

```
def My_range_func( start = 0, stop, step = 1): # This raises an error

```



```
SyntaxError: non-default argument follows default argument
```

Keyword and positional arguments

When a function is called by passing arguments to it, the arguments are assigned to the parameters based on it's position in the argument list which is from left to right. For instance in the above function `My_range_func()` when `(20, -3, 2)` is passed as the argument list, the assignment occurs in the following order.

1. `20` gets assigned to variable `stop`

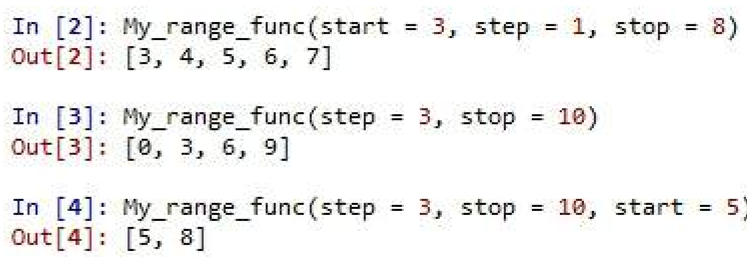
2. `-3` gets assigned to variable `start`

3. `2` gets assigned to variable `step`

These arguments are positional arguments as the position of the arguments matter in the function call. Python allows re-ordering of the arguments that are passed if the arguments are passed as keyword arguments. When function call is made with keyword arguments the arguments can be passed in any order. The general syntax of argument list when passing arguments as keyword arguments is given below.

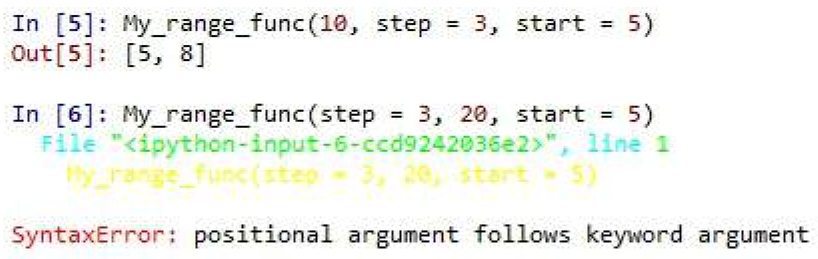
```
<function_name>(<ParamterA> = <value>, <ParameterB> = <value>,...)
```

The following are calls to the previously defined `My_range_func()` function and they are all valid.



```
In [2]: My_range_func(start = 3, step = 1, stop = 8)
Out[2]: [3, 4, 5, 6, 7]

In [3]: My_range_func(step = 3, stop = 10)
Out[3]: [0, 3, 6, 9]

In [4]: My_range_func(step = 3, stop = 10, start = 5)
Out[4]: [5, 8]
```

A function call can be made with a mixture of positional and keyword arguments but in this case the keyword arguments must be pushed to the right side of the argument list which means there should not be any positional arguments that follow keyword arguments in the argument list of a function call. The example below show a valid and invalid instance of making function calls with combination of both positional and keyword arguments.



```
In [5]: My_range_func(10, step = 3, start = 5)
Out[5]: [5, 8]

In [6]: My_range_func(step = 3, 20, start = 5)
  File "<ipython-input-6-ccd9242036e2>", line 1
    My_range_func(step = 3, 20, start = 5)

SyntaxError: positional argument follows keyword argument
```

Arbitrary arguments

There are many cases where the number of arguments that will be passed during a function call is unknown. An example of this case is a function that simply prints every argument it receives,but the number of arguments it is passed is arbitrary. An arbitrary parameter can be specified in the function definition by adding a '*' before the parameter which says that this parameter is a tuple, this tuple will contain the arbitrary number of arguments that will be passed. Lets take a look at an example that illustrates arbitrary arguments in python.

```
def sayHiTo(*people):
   """This function says hi
   to everyone in the people tuple."""

   # names is a tuple that contains all the arguments
   for person in people:
       print("Hi " + person + "!")

sayHiTo("khalid", "guido von rossum")

```

The above program defines a function that takes an arbitrary number of arguments that is available to the function through the "people" tuple. "people" tuple is a tuple that consists of all the arbitrary arguments that are passed to the function when it's called. "Hi <person name>!" is printed for each element of the "people" tuple. The output of the above program is given below.

```
Hi khalid!
Hi guido von rossum!
```

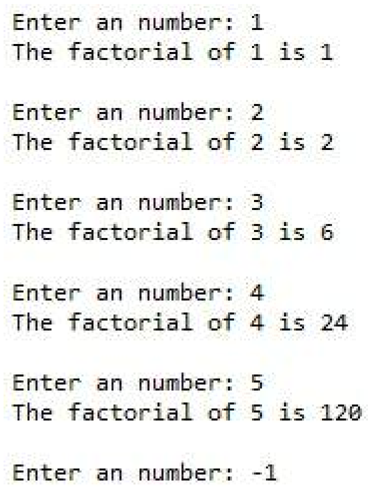
4.5.6 Recursive functions

We already know that functions in python can call other functions, which means there also can be a case where a function calls itself. These kind of functions are called as "recursive functions" as they are defined in terms of themselves. A very common example that is used to explain recursive functions is the factorial program(program that returns the factorial of a given number). The factorial of a number is the product of all integers from one to itself

For example factorial of five represented as "`5!`" is equal to "`1 * 2 * 3 * 4 * 5`". Lets take a look at an example that defines a recursive function to return the factorial of a given number.

```python
#==========================================================
# Python program to calculates the factorial of a number
#==========================================================

def factorial(x):
    """This is a recursive function
    that calculates the factorial of
    an integer"""

    if x == 1: # recursive calls terminates when this condition is met
        return 1
    else:
        return (x * factorial(x-1)) # The function calls itself

while 1:
    num = eval(input("Enter an number: "))
    if num == -1:
        break

    print("The factorial of", num, "is", factorial(num))
```

In this example the factorial() function is recursive as it makes a call to itself in the function body. Whenever the factorial function is called, it recursively calls itself decrementing the number passed as an argument each time until the terminating condition is met that is "`x == 1`" The results are continually evaluated from the last call to the first and the result of each function call is returned to it's client caller until the initial call is reached. The a possible output of the above program is given below.

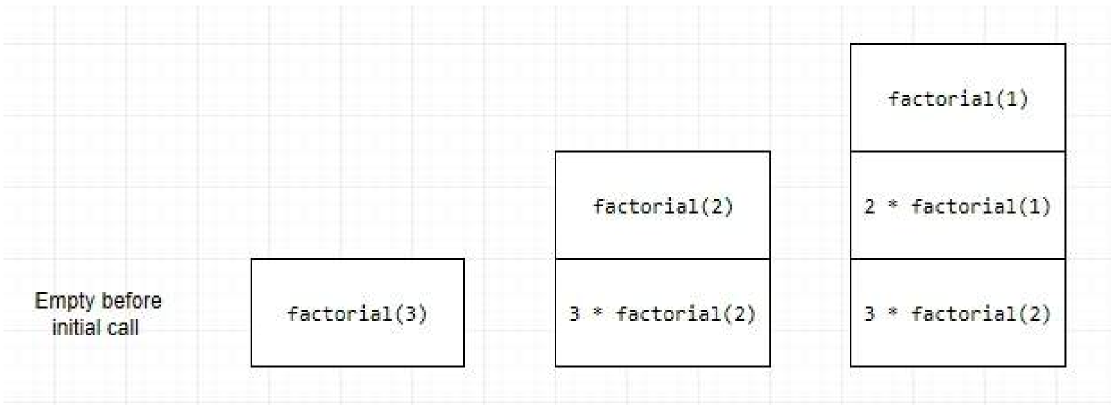


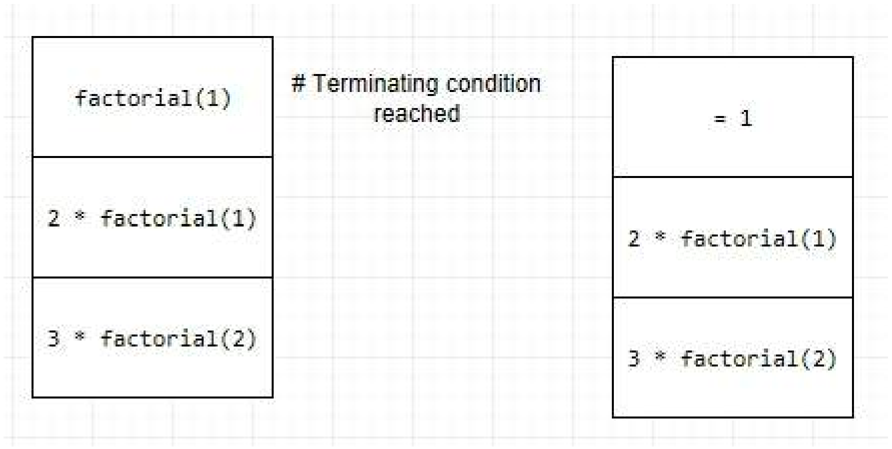
```
Enter an number: 1
The factorial of 1 is 1

Enter an number: 2
The factorial of 2 is 2

Enter an number: 3
The factorial of 3 is 6

Enter an number: 4
The factorial of 4 is 24

Enter an number: 5
The factorial of 5 is 120

Enter an number: -1
```

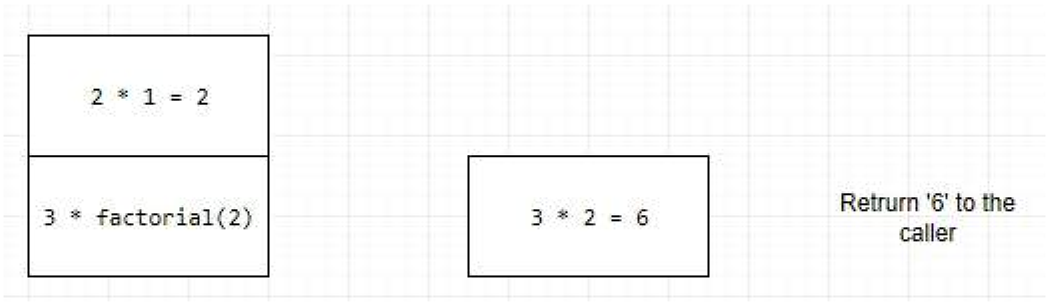
The following illustration shows the method of evaluation when the `factorial()` function is called with `5` passed as the argument.

```
factorial(4)                  # initial call with 4

4 * factorial(3)             # second call with 3

4 * 3 * factorial(2)         # third call with 2

4 * 3 * 2 * factorial(1)     # fourth call with 1

4 * 3 * 2 * 1                # fourth call returns 1

4 * 3 * 2                    # third call returns 2 * 1 = 2

4 * 6                        # second call returns 3 * 2 = 6

24                           # return from initial call 4 * 6 = 24

                             # 24 is the value returned by factorial(4)
```

Recursive function calls can be easily understood in terms of pushing and popping elements from a stack, where each call to a recursive function is pushed to the top of the stack. After the last insertion to the stack has been made the elements of the stack are popped one by one till all the elements have been popped. Before popping an element at the top of the stack the function is evaluated and the if there is a any value returned by that function it is returned to it's caller in the next top of the stack. This order of evaluation continues till there are no more elements in the stack and the if any value is returned by the last element of the stack it is returned to the initial caller of the recursive function. Lets try to visualize this with the function call `factorial(3)`.







Advantages of Recursion:

- The use of recursive functions make the code more elegant and readable.
- Recursive functions can be used to break a complex problem down to simpler problems of a repetitive kind.
- Generating a sequence is easier using recursion than using nested iteration.

Disadvantages of Recursion:

- Sometimes the logic of recursive code can be hard to follow or understand.
- Recursive calls take up more memory than it's iterative version.
- Harder to debug
-

4.5.7 Lambda functions

In this section we'll take a look at what lambda functions are in python and some examples using lambda functions.

Python allows creating functions without a name called lambda functions. Like normal functions are defined using the `def` keyword lambda functions are defined using the `lambda` keyword. Lambda functions are also called as "anonymous functions". The syntax of lambda functions in python are given below.

```
Lambda <arguments>: <Expression>
```

Lambda functions are allowed to have any number of arguments but can have only one expression as it's body. When a lambda function is called the expression is evaluated and returned. Lambda functions are used whenever function objects are needed. Let's take a look at an example that defines and uses a lambda function.

```
# Program that defines and uses a lambda function that
# returns the cube of number

cube = lambda a: a * a * a # Defining the lambda function

num = eval(input("Enter a number: "))
print(cube(num))          # Using the lambda function

```

In the above code `lambda a: a * a * a` where "a" is the argument and "a * a * a" is the expression that is evaluated and returned. This lambda function has no name that's why its called an anonymous function. The identifier cube returns the function object that is assigned to it, so it can be called a normal function.

The statement

```
cube = lambda a: a * a * a
```

Has the same meaning as the below function definition

```
def cube(a):

   return a * a * a
```

Lambda functions are usually used when a nameless function is needed for a short time. They are generally used as an argument to functions that take other functions as parameters(higher order functions). Lets go through two examples that show the practical use of lambda functions in python.

Using lambda functions in the `filter()` function

The `filter()` function in python takes a function and a list for parameters. The provided function is called for all the elements of the provided list and a new list is returned with all the elements that returned true when passed to the provided function.

The example below shows the calls the “filter” function to filters out all the zero’s from the provided list and returns a new list object with only non zero elements in the provided list.

```python
#===========================================================
# Program to filter out zero's from a list
#===========================================================
my_list = [10, 0, 7, 76, 0, 0, 39, 87]

li = list(filter(lambda a: a != 0 , my_list))

print(li)

```

The above program prints the output given below when executed.

```
[10, 7, 76, 39, 87]
```

Using lambda functions in the `map()` function

pythons map() function takes in a function object and a list object as it’s arguments and returns the a list object containing all the values returned by the provided function when it’s called on every element in the provided list.

The example code below makes a call to the map() function to return the square of all the elements in the provided list.

```python
#===========================================================
# Program to square all the elements of a given list
```

```
#============================================================
my_list = [1, 2, 4, 8, 16, 32, 64, 128]

li = list(map(lambda x: x * x, my_list))

print(li)

```

The output of this program is shown below.

```
[1, 4, 16, 64, 256, 1024, 4096, 16384]
```

4.5.8 Global, local and non local variables

Lifetime and scope of variables has already been introduces in the previous chapters, now let’s take a look at global, local and non local variables in detail.

Global variables

The parts of a python program that exists at the surface level i.e outside of any internal code blocks like “if block”, “while block”, “function definitions”, etc are said to be the global scope and any variables that are created at that level are called global variables. Variables declared in the global scope can be accessed from any part of the program as long as they are alive. In this example, ‘x’ is declared in the global scope so it’s a global variable and can be accessed from any part of the program this ability of global variable x is shown by by defining a function that uses the value of ‘x’ without it having an ‘x’ as parameter nor as a variable declared inside the function.

```
#----------------------------------------------------Global scope
x = 10     # Global variable
y = 20     # Global variable
z = 30     # Global variable
def foo(): # foo's definition, part of the global scope
#----------------------------------------------------Global scope
    #------------------foo's local scope starts
    a = 0 # foo's local variable
    print(a, x, y, z, b) # foo can access global variables
    #------------------foo's local scope ends
```

```
#----------------------------------------------------Global scope
b = 40      # Global variable
foo()       # making a call to function foo

#----------------------------------------------------Global scope

```

This program prints the output below. Note that the function foo can read the global variables.

```
0 10 20 30 40
```

Take a look at an interesting case where the python interpreter raises an error because it misunderstands what is actually meant.

```
#--------------------------------------------------------Global scope
x = 10      # Global variable
def foo():  # foo's definition, part of the global scope
#--------------------------------------------------------Global scope
    #-----------------------------------foo's local scope starts
    x = x + 40          # Trying to change the value of x ---------!
    a = 0              # foo's local variable
    print(a, x)         # foo can access global variables
    #-----------------------------------foo's local scope ends
#--------------------------------------------------------Global scope
foo()       # Calling foo
#--------------------------------------------------------Global scope

```

When the above code is executed the interpreter raises an error because, in the normal case when an assignment statement is encountered by the interpreter in a local scope, the name on the right hand side of the assignment is considered to be local variable of that local scope. So when the interpreter encounters the statement "`x = x + 40`", it checks the left hand side, creates a new name 'x' in the local scope of the function "foo" then checks the expression on the right hand side of the assignment statement tries to access the non existent object associated to the local variable 'x'. it raises an error because 'x' is referenced in an expression before it's assigned a value. When the above code is executed, it gives the following output.

```
  File "C:/Users/khalid/Documents/ML/Bucca/global scope 2.py", line 7, in foo
    x = x + 40           # Trying to change the value of x ---------!

UnboundLocalError: local variable 'x' referenced before assignment
```

To solve this problem of not being able to modify a global variable, python provides a global variable which will be discussed in the sections that follow.

Local Variables

A variable that is declared inside a local scope is a local variable, a variable local to the scope that it was created in. A local variable exists only in the scope that it was defined in and can not be accessed outside of that scope. Local variable are created just like global variables the only difference being that local variables are declared inside functions or are arguments of a function call and global variables are declared at the global scope of a python program. The example below creates a local variable inside function "foo".

```
def foo():  # foo's definition
    #------------------------------------foo's local scope starts
    a = "I'm a local variable in function foo!" # foo's local variable
    print(a)
    #------------------------------------foo's local scope ends

foo()

```

The output prints the output given below.

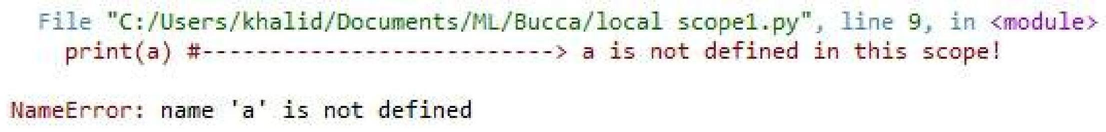
```
Im a local variable in function foo!
```

In the example below the variable 'a' that is local to the function "foo" is accessed outside of the scope of the variable which results in an error.

```
def foo():  # foo's definition
    #------------------------------------foo's local scope starts
    a = 0              # foo's local variable
    print(a)
    #------------------------------------foo's local scope ends

```

```python
#------------------------------------------------Global scope

print(a) #--------------------------> a is not defined in this scope!

#------------------------------------------------Global scope

```

The following error is raised when the above program is executed as the variable ‘a’ is defined in the local scope of function “foo” and is available for use only there, it can’t be accessed by the upper levels. This program raises the error shown below.



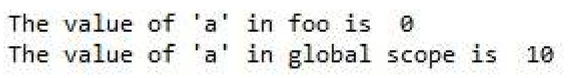
```
  File "C:/Users/khalid/Documents/ML/Bucca/local scope1.py", line 9, in <module>
    print(a) #--------------------------> a is not defined in this scope!

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

Global and local variables with the same name

When a variable is declared in the local scope of a function that has the same name as a variable in the global scope, which variable gets accessed? In such cases the python prefers the local variable over the global variable, so when a variable name in the local scope is accessed that has the same name as a variable in the global scope the object associated to that name in the local scope is returned. The code example below shows an instance of this case.

```python
#-------------------------------------------------------Global scope
a = 10          # Global variable
def foo():       # foo's definition, part of the global scope
#-------------------------------------------------------Global scope
    #-------------------------------------foo's local scope starts
    a = 0       # foo's local variable
    return a     # The local variable a is accessed (a = 0)
    #-------------------------------------foo's local scope ends

#-------------------------------------------------------Global scope
print("The value of 'a' in foo is ",foo())
print("The value of 'a' in global scope is ", a)
#-------------------------------------------------------Global scope

```

In the above code, the same name ‘a’ is used both in the global scope as well as in the local scope of function “foo”. When the values of both variables are printed we get different values as the local variables preferred over global if there is a name match



```
The value of 'a' in foo is  0
The value of 'a' in global scope is  10
```

Lets look back at the previous example where we try to change the value of a global variable inside a function. This causes an error to be raised by the python interpreter, and it’s already been mentioned that the solution python provides for this is the “global” keyword

In Python the `global` keyword allows modification variables that are outside the current scope. They can be used to create global variables and modify it in the local context.

Here are some things to remember while using the `global` keyword.

1. A variable is by default a local to a function when it is declared inside the function
2. A variable is by default a global variable when its declared outside a function. It makes no difference using the global keyword on variables defined outside functions.
3. The global keyword can be used inside functions to read and write global variables inside functions.

Lets take a look at some examples that illustrate the use and working of the global keyword.

```python
x = 20       # x is a global variable

def printx():
    print(x)  # Since there is no local variable x, the global variable x
              # can be accessed

printx()

```

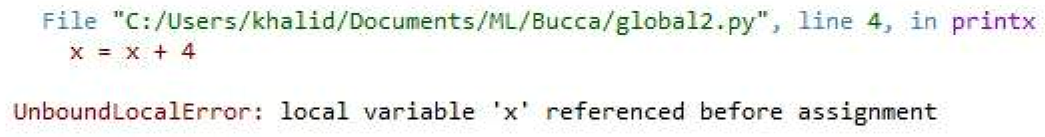
This example code produces the below output.

```
20
```

Here, notice that as the function `printx()` doesn't have a local variable 'x', it can read the value of the global variable 'x', but the interpreter raises an error when we try to write to the same variable.

```python
x = 20 # x is a global variable

def printx():
    x = x + 4  # try using the value of the global variable
    print(x)

printx()

```

When the above program is ran it raises the error below,because the value of a global variable is being modified in the function.without using the global keyword it is only possible to read the value of a global variable inside a function



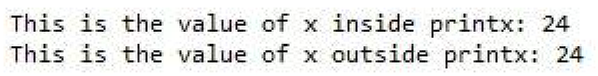
```
  File "C:/Users/khalid/Documents/ML/Bucca/global2.py", line 4, in printx
    x = x + 4

UnboundLocalError: local variable 'x' referenced before assignment
```

The `global` keyword can be used to overcome this kind of a problem where the modifications to global variables from within a function is required. The example below shows how a global keyword can be used to modify a global variable from within a function. To modify a global variable from inside a function this variable has to first to be declared as `global` inside the function it needs to be used in. This is illustrated in the example below.

```python
x = 20        # x is a global variable

def printx():
    global x   # Now the global variable x can be modified
    x = x + 4  # No error raised
    print("This is the value of x inside printx: %i" %x)

printx()
print("This is the value of x outside printx: %i" %x)
```

```

```

The output of the above program is given below. notice that the value of the variable ‘x’ is the same inside and outside the `printx()` function. This is because variable ‘x’ inside the `printx()` function now refers to the same object as the variable ‘x’ outside this function.

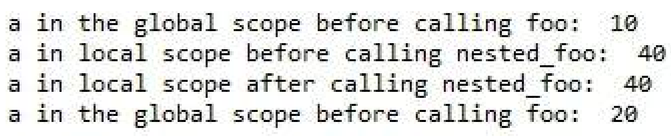


```
This is the value of x inside printx: 24
This is the value of x outside printx: 24
```

The `global` keyword can also be used in nested functions, let’s take a look at an example of using the `global` keyword in a nested function.

```python
a = 10
print("a in the global scope before calling foo: ", a)

def foo():
    a = 40
    print("a in local scope before calling nested_foo: ", a)

    def nested_foo():
        global a
        a = 20

    nested_foo()
    print("a in local scope after calling nested_foo: ", a)

foo()
print("a in the global scope before calling foo: ", a)

```

When the global keyword is used on the variable ‘a’ in the `nested_foo()` function, any change that this function makes to the variable ‘a’ in it’s scope affects the variable at the global scope. The output of the above program is given below.



```
a in the global scope before calling foo:  10
a in local scope before calling nested_foo:  40
a in local scope after calling nested_foo:  40
a in the global scope before calling foo:  20
```

Non local variables

Non local variables are used in nested functions where the local scope is not defined, so the variable can't be in either the local nor the global scope.

The keyword `nonlocal` is used to declare a non local variable in python.let's take a look at an example that explains non local variables in python.

```python
def foo():
    #------------------------------------foo's local scope starts
    a = 20          # foo's local variable
    print("a in foo: ", a)
    #------------------------------------foo's local scope ends
    def nested_foo():
        nonlocal a   # Now 'a' is a non local variable
        a = a + 20
        print("a in nested_foo: ", a)
    nested_foo()
    print("a in foo: ", a)
#------------------------------------------------Global scope
foo()
#------------------------------------------------Global scope
```

The output of the above program is given below.

```
a in foo:  20
a in nested_foo:  40
a in foo:  40
```

Here the function `nested_foo()` is defined in the local scope of the function `foo()`. The `nonlocal` keyword is used to declare 'a' to be a non local variable. Observe that when a nonlocal variable is changed, this change also reflects on the local variable 'a' of function `foo()`. This is because the variable 'a' which is in the local scope of function `foo()` is now changed by function `nested_foo()`.

4.6 Exercises

1. Suppose you need to compute the square root of a number in a Python program. Would it be a good idea to write the code to perform the square root calculation? Why or why not?

2. Which of the following values could be produced by the call random.randrange(0, 100) function?

4.5 **34** **-1** **100** **0** **99**

3. Classify each of the following expressions as legal or illegal. Each expression represents a call to a standard Python library function.

(a) math.sqrt(4.5)

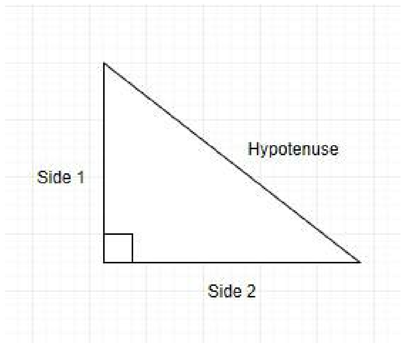
(b) math.sqrt(4.5, 3.1)

(c) random.rand(4)

(d) random.seed()

(e) random.seed(-1)

4. From geometry: Write a computer program that, given the lengths of the two sides of a right triangle adjacent to the right angle, computes the length of the hypotenuse of the triangle. (See Figure ??.) If you are unsure how to solve the problem mathematically, do a web search for the Pythagorean theorem.



5. Write a guessing game program in which the computer chooses at random an integer in the range 1. . . 100. The user's goal is to guess the number in the least

number of tries. For each incorrect guess the user provides, the computer provides feedback whether the user's number is too high or too low.

6. Extend Problem 5 by keeping track of the number of guesses the user needed to get the correct answer. Report the number of guesses at the end of the game.

7. Extend Problem 6 by measuring how much time it takes for the user to guess the correct answer. Report the time and number of guesses at the end of the game.

8. Is the following a legal Python program?

```
def proc(x):
return x + 2
def proc(n):
return 2*n + 1
def main():
x = proc(5)
main()
```

9. Is the following a legal Python program?

```
def proc(x):
return x + 2
def main():
x = proc(5)
y = proc(4)
main()
```

10. Is the following a legal Python program?

```
def proc(x):
print(x + 2)
```

```
def main():
x = proc(5)
main()
```

11. Is the following a legal Python program?

```
def proc(x):
print(x + 2)
def main():
proc(5)
main()
```

12. Is the following a legal Python program?

```
def proc(x, y):
return 2*x + y*y
def main():
print(proc(5, 4))
main()
```

13. Is the following a legal Python program?

```
def proc(x, y):
return 2*x + y*y
def main():
print(proc(5))
main()
```

14. Is the following a legal Python program?

```
def proc(x):
return 2*x
def main():
print(proc(5, 4))
main()
```

15. Is the following a legal Python program?

```
def proc(x):
print(2*x*x)
def main():
proc(5)
main()
```

16. The programmer was expecting the following program to print 200. What does it print instead? Why does it print what it does?

```
def proc(x):
x = 2*x*x
def main():
num = 10
proc(num)
print(num)
main()
```

17. Is the following program legal since the variable x is used in two different places (proc and main)? Why or why not?

```
def proc(x):
return 2*x*x
def main():
x = 10
```

```
print(proc(x))
main()
```

18. Is the following program legal since the actual parameter has a different name from the formal parameter (y vs. x)? Why or why not?

```
def proc(x):
return 2*x*x
def main():
y = 10
print(proc(y))
main()
```

19. Complete the following distance function that computes the distance between two geometric points (x1,y1) and (x2,y2):

```
def distance(x1, y1, x2, y2):
...
```

Test it with several points to convince yourself that is correct.

20. What happens if a client passes too many parameters to a function?

5. Python Modules

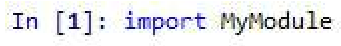
Modules in python allow a logical organization of code. When code gets too large to manage, as already discussed it's best break this code into pieces that can interact with each other, each piece having it's own independent functionality. These pieces generally have some attribute that have some relation to each other, this could be that they belong to the same class of member variables and member methods or that they belong to the same set of similar but independent functions. It would very advantageous if this code could be shared and reused in other python programs by the same or a different programmer. For this python gives modules the ability to bring in code that's already been written and this is an important way that python provides code reusability. This method of including the code in other modules in another program is called "importing". In a single sentence a module can be defined as "self contained and organized pieces of python code that can be shared".

In python modules are in the form of a python file i.e with a ".py" extension, so any file with a ".py" extension and has python code code is considered python module. Modules can be used to break down large and unmanageable code into a more structured and organized set of pieces that can be reused later in other python files.

An example of the use of function's is that all your most used functions can be defined in a python file and imported in any other python file without the need to copy all the function definitions. Let's take a look at an example that creates a python module that defines a function that returns the square of provided number.

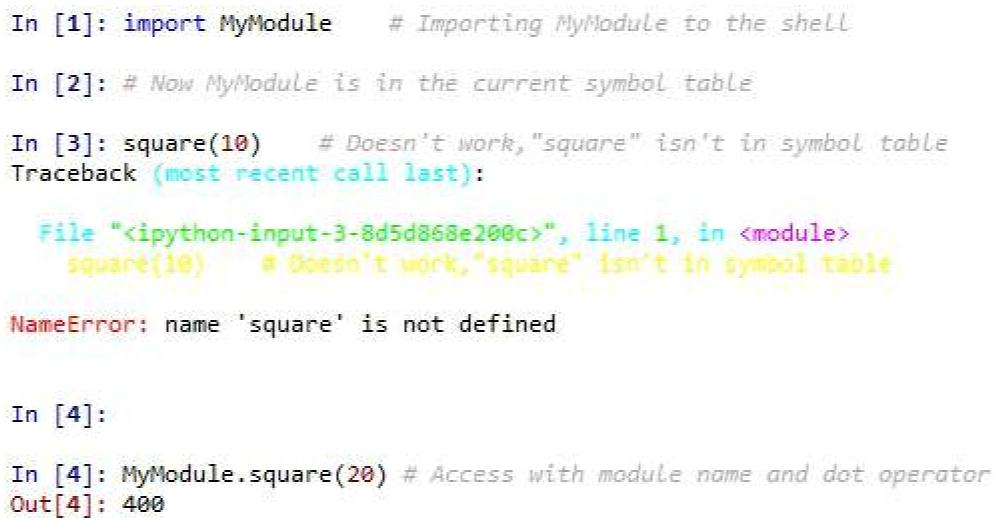
```python
#============================================================
# Module "MyModule.py"
#============================================================

def square(x):
   """This function returns
   the square of the given number"""

   result = x * x
   return result

```

This module named "MyModule" has a function "square" defined inside it, this function takes an numeric value as it's parameter and returns it's square. A module can be imported into another module or to the python shell, allowing them to use the attributes(classes and functions) that are defined inside the module. To do this python provides the `import` keyword. Try importing our previously defined module in to the python shell with the following statement(make sure the shell runs in the same directory that the module exists in).



```
In [1]: import MyModule
```

This statement doesn't add the names of the module attributes(functions and classes) to the current symbol table directly, it only adds the name "MyModule" to it, so the function defined in that module can't directly be invoked. It can however be invoked using the module name(MyModule) and the dot operator(.) like in the example below.



```
In [1]: import MyModule     # Importing MyModule to the shell

In [2]: # Now MyModule is in the current symbol table

In [3]: square(10)     # Doesn't work,"square" isn't in symbol table
Traceback (most recent call last):

  File "<ipython-input-3-8d5d868e200c>", line 1, in <module>
    square(10)     # Doesn't work,"square" isn't in symbol table

NameError: name 'square' is not defined


In [4]:

In [4]: MyModule.square(20) # Access with module name and dot operator
Out[4]: 400
```

Just like the module that we just made python provides several standard modules that are already available to use. The full list of python standard module and its description is available in the documentation at "python.org" or you can take this link docs.python.org/3/py-modindex.html.

As the standard modules are just modules that are already written, the method to import them is the same as importing user defined modules. Let's discuss the different methods to import python modules.

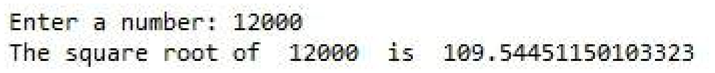
The import statement

Modules can be imported with the `import` keyword, after importing the attributes that were defined in the function can be accessed by the module that it was imported into with the module's name and the dot operator(.) as described in the previous section. Let's take an example of importing one of python's standard modules that we have already dealt with before.

```
import math

# ask user for input
```

```python
num = eval(input("Enter a number: "))
num_root = math.sqrt(num) # Using the "sqrt()" function
                          # defined in the math module
# print the result
print("The square root of", num, " is ", num_root)

```

The output of this program when it is executed is given below.



```
Enter a number: 12000
The square root of  12000  is  109.54451150103323
```

import by renaming

When a module is imported with the import statement, the module's name becomes the handle to access the attributes that are defined in the module. Python allows renaming the handle used to access the contents of a module by renaming the module handle at the import statement. This can be done with the `as` keyword that python provides. An example of it's use is shown below.

```python
import math as m

num = eval(input("Enter a number: "))

num_root = m.sqrt(num)     # Observe that the module handle is 'm'
num_root = math.sqrt(num)  # math is not recognized

print("The square root of ", num, " is ", num_root)

```

In this program the handle for the "math" module that was imported is now 'm' so the name "math " no longer recognized in our scope, making `math.sqrt()` an invalid call and `m.sqrt()` a valid call.

The from..import statement

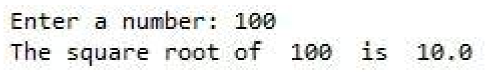
Python allows importing specific names from a module instead of instead of importing the entire module. This can be done with the `from..import` statement. The general syntax for the `from..import` statement is given below.

```
from <module_name> import <name>
```

In this example we'll just import the sqrt() function from the math module. When specific attributes of a module are imported into a python program, the that specific attribute get's added to the symbol table, so the module name or any other attributes that haven't been imported are not recognized.

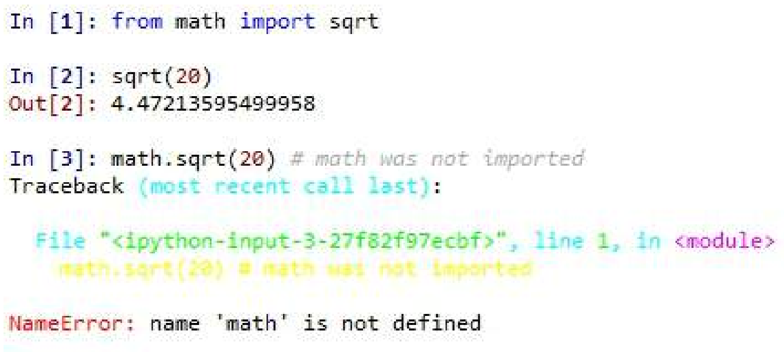
```
1  from math import sqrt   # Specifically import the sqrt() function
2                          # Now "sqrt" is in the symbol table.
3  num = eval(input("Enter a number: "))
4
5  square_root = sqrt(num) # Function sqrt() can be called
6                          # without the dot operator.
7
8  print("The square root of ", num, " is ", square_root)
9
```

One of the possible outputs of the above program is shown below.



```
Enter a number: 100
The square root of  100  is  10.0
```

Note that in the above program the square root function is called directly without using the dot operator with the module name that is because the from..import statement at line one, adds the function name "sqrt" to the symbol table allowing it to be called as though it was defined in this same file. You can also see that the name is "math" isn't recognized in the program as it wasn't added to the symbol table.



```
In [1]: from math import sqrt

In [2]: sqrt(20)
Out[2]: 4.47213595499958

In [3]: math.sqrt(20) # math was not imported
Traceback (most recent call last):

  File "<ipython-input-3-27f82f97ecbf>", line 1, in <module>
    math.sqrt(20) # math was not imported

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

Importing all names in a module

The general syntax for importing all the names in a module is shown below.

```
from <module_name> import *
```

When an import is performed this way all the names in the module imported from gets added to the symbol table, so they can be used as if they were defined in the python file that imported them. This method to import isn't a good practice as it affects the readability of the program and may some time result in unwanted duplicate names.

```python
# Not a good practice to import all names
from math import *  # import all names in the math module
# Warning 1 - Unable to detect undefined names

print("the square root of 12000 is ", sqrt(12000))
# Warning 2 - sqrt may be undefined or defined from start imports.

```

The output of the above program is given below.

```
the square root of 12000 is  109.54451150103323
```

Observe that there are two warnings that have been put up.

1) When all the names from a module is imported, names that are undefined could be from the star import

2) Function `sqrt()` could either be undefined or defined in the start imports

Another problem that could arise from using star imports is that programmer might not know whether the names that is declared in the program have already been declared in the module that is being imported, this could cause duplicate definitions for an identifier. In the code below the identifier "pow" is being redefined even though it has already been imported from the math module.

```python
# Not a good practice to import all names
from math import *  # import all names in the math module
# Warning 1 - Unable to detect undefined names

def pow(x, y):
    return y ** x

```

```python
print(pow(8, 2))
# pow(x ,y) in the math module returns x ** y
# function pow(x, y) has been redefined to return  y ** x

```

The output of this program is `256`(since `2 ** 8`) instead of `64.0` (in which case `8 ** 2`) if the call was made to `pow()` function defined in the math module.

Module search path

When the python interpreter encounters an import statement where is requested python module searched for? Python searches several locations for the module requested. The search first starts by checking if the module that was requested is a built in method, if it is, then interpreter has the module that it needs, in the case that it isn't the python interpreter searches the directories that are specified in `sys.path` , the search proceeds in the following order

1. Search the current directory.
2. Search the list of directories in the `PYTHONPATH` environment variable.
3. Search the installation dependent default variable.

The list of directories in `sys.path` can be modified to add user specified paths.

Reloading modules

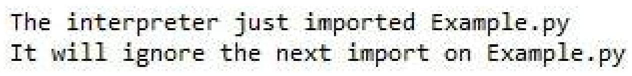
Python interpreters during it's session doesn't import the same module more than once, yes it can import several modules that have different names. Python interpreter when it encounters an import statement, does not import the specified module if it already imported a module with the same name during that same session. This behaviour of the interpreter is illustrated in the code example given below. For this we'll be using two python files, one is a python module to import(`Example.py`) and the other one that imports the previously created module multiple times(`ModuleReloading.py`).

```python
#======================================================
# Module "Example.py"
#======================================================

""" Using this module to illustrate how the interpreter
 reacts multiple imports of the same module """

```

```python
print("The interpreter just imported Example.py")
print("It will ignore the next import on Example.py")

```

```python
#======================================================
# "ModuleReloading.py"
# This program shows the python interpreters behaviour
# when there are multiple imports on a module.
#======================================================
import imp

import Example
import Example
import Example
import Example

```

The output when ModuleReloading.py is executed is given below. Observe that the module "Example.py" is imported only once.



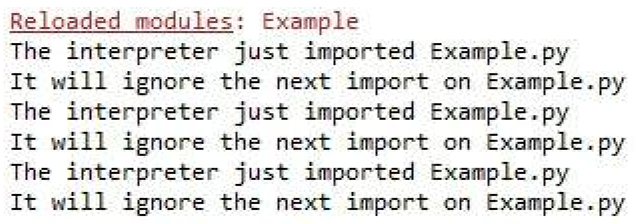
```
The interpreter just imported Example.py
It will ignore the next import on Example.py
```

There are some cases where it may be required to import the same module twice like when there's a change in a module during execution, a solution to this could be to restart the interpreter so it forgets that it has imported that module, but this does not help much. Python has a better way to solve this problem, it provides a `reload()` function inside the `imp` module to reload a module. Now let's try running the same program again by using the reload function to reload the modules instead.

```python
#======================================================
# "ModuleReloading.py"
# This program shows the python interpreters behaviour
# when there are multiple imports on a module.
#======================================================
import imp

import Example
```

```python
imp.reload(Example)
imp.reload(Example)

```

In the output below observe that the modules have been reloaded.



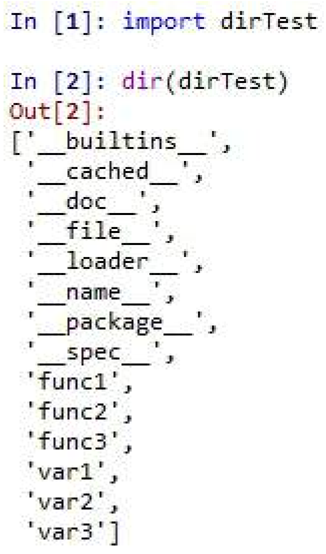
```
Reloaded modules: Example
The interpreter just imported Example.py
It will ignore the next import on Example.py
The interpreter just imported Example.py
It will ignore the next import on Example.py
The interpreter just imported Example.py
It will ignore the next import on Example.py
```

The `dir()` function

Python provides as a built in method that can be used to find out what names a module has defined inside it. For the illustration of the dir() function we'll create a python module named "dirTest" as follows.

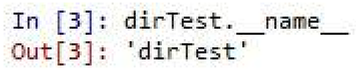
```python
#============================================================
# Module "dirTest.py"
#============================================================
# Variable names
var1 = 1
var2 = 2
var3 = 3

# Function definition
def func1():
    print("This is func1")

def func2():
    print("This is func2")

def func3():
    print("This is func3")

```

Calling the function `dir()` on a module's name returns a sorted list of names. In this list there is a set of names that begin with an underscore(_), these names are default python attributes that are associated to a module and are not user defined names. The rest of the names are user defined, and contains all the names that where defined in the module. Calling the `dir()` for the above module returns the sorted list of names shown below. The `dir()` function won't recognize the module name until it's imported, so import the module before making the call to `dir()`.



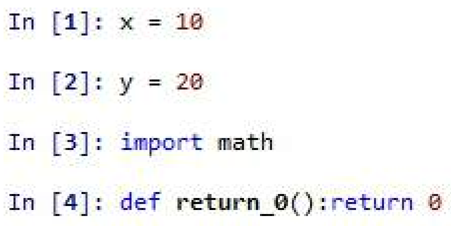
```
In [1]: import dirTest

In [2]: dir(dirTest)
Out[2]:
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'func1',
 'func2',
 'func3',
 'var1',
 'var2',
 'var3']
```

The names in the list that start with an underscore are default to the module and aren't user defined. For example the name "__name__" is the name of the module.

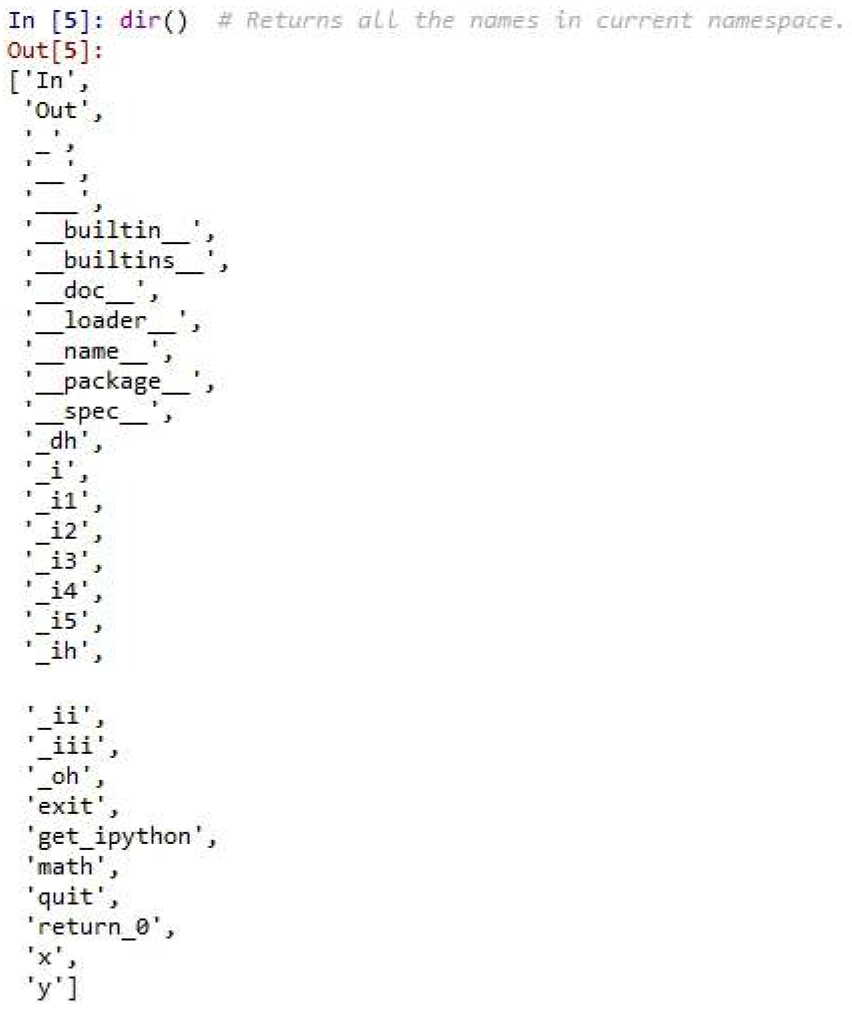


```
In [3]: dirTest.__name__
Out[3]: 'dirTest'
```

The dir() when called without any arguments passed returns all the names in the current namespace. Try declaring some names, making some imports in the python shell and see what it returns.



```
In [1]: x = 10

In [2]: y = 20

In [3]: import math

In [4]: def return_0():return 0
```



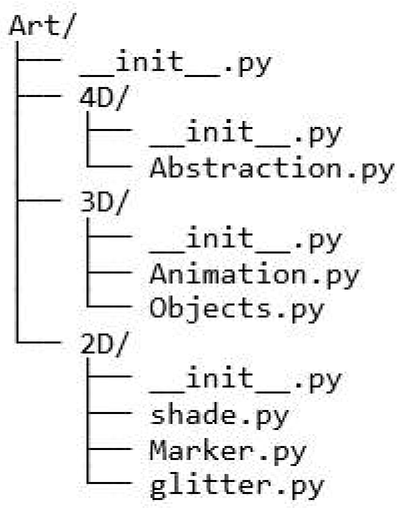
```
In [5]: dir()  # Returns all the names in current namespace.
Out[5]:
['In',
 'Out',
 '_',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i2',
 '_i3',
 '_i4',
 '_i5',
 '_ih',

 '_ii',
 '_iii',
 '_oh',
 'exit',
 'get_ipython',
 'math',
 'quit',
 'return_0',
 'x',
 'y']
```

5.1 Packages

Files are easy to organize as a hierarchy of folders/directories, where each directory can have several other files and sub directories. Generally files that share a similarity in some a placed into the same directory. The advantage of directories is that it allows the organizing the files based on their relevance or some other criteria. Python enables organizing modules as a hierarchy of directories with “packages” for directories and “modules” for files.

A good practice while working on a python project is t always organize relevant modules by placing them in the same package and by placing irrelevant modules in different packages, this practice proves to remove much of the confusion when dealing with projects that are on a larger scale. Just like directories can have other sub-directories and other files, a package in python can have other packages and modules.

The directories that have a module named “`__init__.py`” are considered to be python packages. Although this file can be left empty, it is a good practice to have code that initializes the package in this module. The figure below shows a helpful visualization of valid python package.



```
Art/
├── __init__.py
├── 4D/
│   ├── __init__.py
│   └── Abstraction.py
├── 3D/
│   ├── __init__.py
│   ├── Animation.py
│   └── Objects.py
└── 2D/
    ├── __init__.py
    ├── shade.py
    ├── Marker.py
    └── glitter.py
```

The above package hierarchy could be a possible organization of modules when developing a paint application with 2D, 3D and 4D features. A directory isn’t considered to be a package without the “__init__.py” module. Observe that each package in the above visualization has an “__init__.py” module. Now let’s look at the different ways that packages can be handled in a python program.

Importing modules from packages

Importing specific packages from a package has the same general syntax as importing specific names from a module. Importing specific modules from a package can be done with the dot operator(.). From the above package hierarchy to import the “Abstraction” module from the “4D” sub package, the following statement is used.

```
import Art.4D.Abstraction
```

Let’s assume that this module “Abstraction” has a function named `draw()` to used this function the entire path should be used to reference it as shown below.

```
Art.4D.Abstraction.draw(<argument list>)
```

This construct to reference the function is lengthy so instead the module can be imported without the package prefix as,

```
from Art.4D import Abstraction
```

When imported this way the function can be called in a more easier way as

```
Abstraction.draw(<argument list>)
```

To specifically import the draw() function from the Abstraction module we can use the import statement below.

```
from Art.4D.Abstraction import draw
```

Now the draw function can be called as,

```
draw(<argument list>)
```

This is obviously easier to write looking at the amount of typing to do but not recommended. Using the full namespace while calling makes it clear exactly which function is being called and reduces the possibility of a collision of two   name in your code.

6. Python classes and objects

This chapter discusses classes, objects and object oriented programming in python. In python any construct that contains a value of any type is an object. Python is an "Object oriented programming language"(OOP language) but there is no compulsion to use OOP in a program, which means as we've already seen in the previous examples, python programs can be written without using any class definitions and class instances although it's recommended to use them. Object oriented programming eliminates many of the problems encountered when the programming approach is procedural.

Imperative programming is a programming pattern or model where statements are used to change the state of a program. Just like how we express commands in our natural language, an imperative program consists of a sequence of commands or statements(can include branching statements) for a computer to execute. An example of this is the Assembly language. Procedural programming is a kind of imperative programming where the programming is built using procedures(we call them functions in python). As concepts of classes and objects weren't discussed or used yet, all examples that has been dealt with till now are can all be abstracted to have a procedural model or pattern.

Python is a multi-paradigm programming language a python program be either written using a procedural approach or an object oriented approach. Although using an object oriented approach is optional in python, it is highly encouraged.

Several of pythons built in types have been demonstrated so far, it's time to define our own types(user defined types). For now we'll define a type called "Point_4D" that represents a point in a four dimensional space. Mathematically a four dimensional point is of the form given below

```
(w, x, y, z)
```

Here w, x , y, z represent the values of the point on each dimension. There are three ways we can represent this four dimensional point in python

- The values of w, x , y, z can be stored into four distinct variables.
- The values of w, x , y, z can be stored in a list or a tuple.
- A four dimensional point type can be defined to create objects that can appropriately represent an instance of `(w, x, y, z)`

Even though defining classes(creating a new type) and using their instances instead is a more complicated task, it is a good practice because it comes with several advantages that will be discussed in the following sections.

A “user defined type” can also be called a “class”. A class definitions header is shown below.

```
class <type_name>
    “““<DocString>”””
```

An object in python has two characteristics, which are

- Attributes(variables)
- Behaviour(Methods)

So in the previous example of the four dimensional point

- w, x ,y, z being the variables
- Plot() being the method used to draw the point based on the values of w, x, y, z.

Object oriented programming places the focus on creating code that can be reused.The concept of creating reusable code is DRY, which stands for don’t repeat yourself.

Object oriented programming in python follows some basic principles that are listed and briefly described in the table given below.

Principle	Description
Inheritance	A class can Inherit the attributes and behaviors of another class without modifying it’s own details.
Encapsulation	Hiding the details of a class from other objects.
polymorphism	A single method that shows different behaviors depending on the parameters passed to it by the caller.

In the following section we’ll briefly discuss the key concepts of object oriented programming in python.

6.1 Introduction to object oriented programming

In this section we’ll discuss the key concepts of object oriented programming in python briefly. The concepts that will be gone through here are

- Classes
- Objects
- Methods

- Inheritance
- Encapsulation
- Polymorphism

6.1.1 Classes

A class for it's object is a blueprint(description / definition). For example if a class is being defined for "human" objects than the class will have details i.e the name declarations for details like name, gender, weight, height, eye color, hair color, skin etc, and the class would also have functions(behaviors) defined specifically for the class like work, sleep, etc. The general syntax to define a class in python is shown below. The body of the class definition is identified by it's indentation, the first unindented line marks the end of the class body. Remember that a type's name must start with a letter or an underscore.

```
class <type_name>

   <docstring>

   <statement 1>

   .

   .

   .

   <statement n>
```

Class is way to bundle together data and functionality. Making a new class creates a new type of object allowing the creation of new instances of objects of that type. Every instance of a class has attributes associated with it to maintain the state that it is in, in addition to this class instances can have methods that are defined in this class for modifying the state of the class instance.

The statements inside a class definition are usually function definitions although other kinds of statements are allowed. When a class header is encountered by the python interpreter it creates a new namespace, which is the local scope. All the local names are a part of this namespace, especially function definition in the classes body are bound to this namespace.

When the python interpreter leaves class definition normally(the end of the definition identified by the indentation),

- A "class object" is created. This class object is wrapper around the contents created by the class namespace.

- The control returns to the original local scope which existed before the class definition was entered
- The “class object” is bound to this original local scope with the class name that was specified in the class header

6.1.2 Class objects

There are two operation that are supported by by class objects which are “attribute references” and “instantiation”. The standard syntax used for all attribute references, is used to reference attributes of class objects(i.e the dot operator). All the names that were in a classes namespace when it’s class object was created are valid attributes of that class.

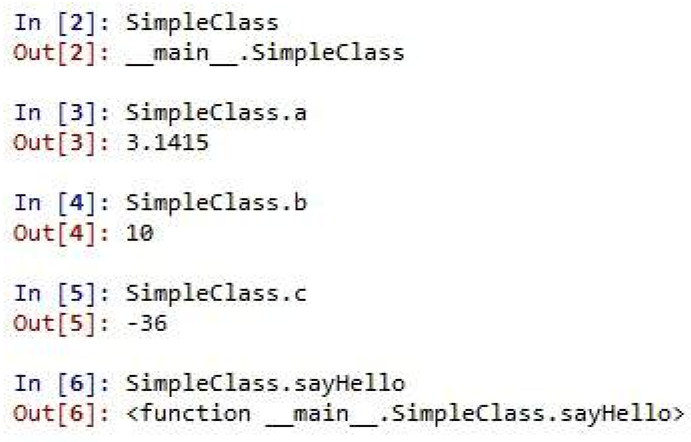
```python
#==========================================================
# Defining a simple class
#==========================================================

class SimpleClass:
    """An Example of a simple class"""

    # Variable assignments
    a = 3.1415
    b = "Ipython"
    c = -12 * 3

    # Method definitions
    def sayHello(self):
        return "Hello user!!"

```

In the above example definition of a class the following attribute references are valid.

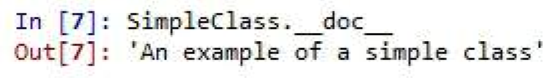
- `SimpleClass.a` - Returns an integer object
- `SimpleClass.b` - Returns an string object
- `SimpleClass.c` - Returns an integer object

- `SimpleClass.sayHello` - Returns a function object

The attributes of the class object that is created when the python interpreter leaves the body of the above class definition can have it's attributes changed by assignment. `__doc__` is also a valid attribute of the class object. This attribute has the reference to the docstring of the class and returns the docstring when used as or in an expression. The examples below that were executed in the python shell illustrates what was discussed about class objects. Here, first the above program is executed.

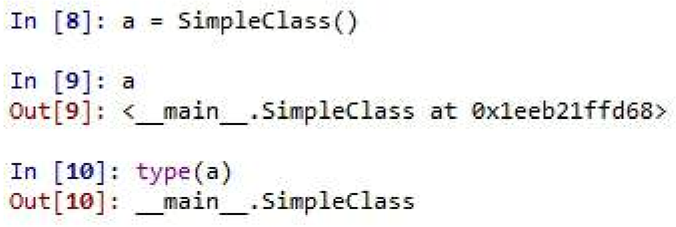


```
In [2]: SimpleClass
Out[2]: __main__.SimpleClass

In [3]: SimpleClass.a
Out[3]: 3.1415

In [4]: SimpleClass.b
Out[4]: 10

In [5]: SimpleClass.c
Out[5]: -36

In [6]: SimpleClass.sayHello
Out[6]: <function __main__.SimpleClass.sayHello>
```



```
In [7]: SimpleClass.__doc__
Out[7]: 'An example of a simple class'
```

Note that here the class object belongs to the "main" namespace, the attributes of the "SimpleClass" class object belongs to the "simpleClass" namespace. A class instantiation takes the form of a function. For the sake of understanding consider the class object as parameterless function that when called as a function , it return a new instance of it's class. For example for the above class.



```
In [8]: a = SimpleClass()

In [9]: a
Out[9]: <__main__.SimpleClass at 0x1eeb21ffd68>

In [10]: type(a)
Out[10]: __main__.SimpleClass
```

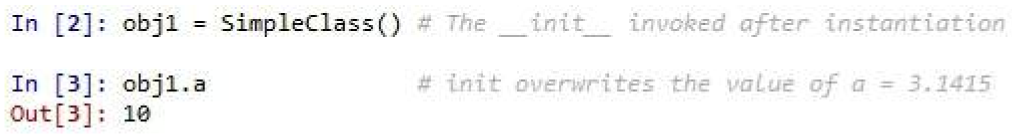
The above statement-8 creates a new instance of class "SimpleClass" and assigns this object to the local variable 'a', also you can see that the variable 'a' is of type "__main__.SimpleClass" and belongs to the "__main__" namespace.

An instantiation operation performed by calling the class object creates an empty object. There are cases where there is a requirement to create objects that are instantiated to a specific initial state. For this a special method "__init__" can be defined in the class definition. An example for it's definition is shown below.

```
#========================================================
# Defining a simple class
```

```python
#==========================================================

class SimpleClass:
    """An Example of a simple class"""

    # Variable assignments
    a = 3.1415
    b = "Ipython"
    c = -12 * 3

    # Method definitions
    def __init__(self): # Automatically invoked at class instantiation
        self.a = 10

    def sayHello(self):
        return "Hello user!!"

```

When a class defines an "__init__" method, the method is invoked immediately for a newly created class instance. The example below illustrates this effect of using an "__init__" method in the class definition.

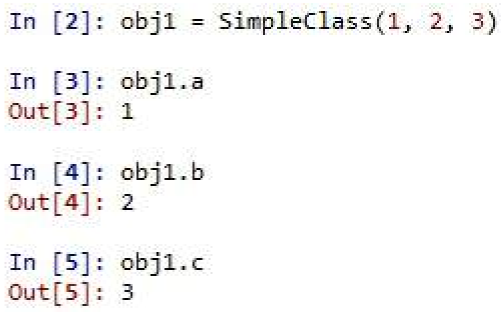


```
In [2]: obj1 = SimpleClass() # The __init__ invoked after instantiation

In [3]: obj1.a               # init overwrites the value of a = 3.1415
Out[3]: 10
```

For the sake of flexibility __init__ method may have arguments that are passed to it. In this case the arguments that are passed to the class instantiation operator also gets passed to the classes __init__ method. An example for this case is shown below for the above class definition.

```python
    def __init__(self, x, y, z):
        self.a = x
        self.b = y
        self.c = z

```

You can see below how making the above change to the __init__ method of the "SimpleClass" class effects the initial state of the instance of the class.



```
In [2]: obj1 = SimpleClass(1, 2, 3)

In [3]: obj1.a
Out[3]: 1

In [4]: obj1.b
Out[4]: 2

In [5]: obj1.c
Out[5]: 3
```

6.1.3 Instance objects

There is only a single operation that can be understood by an instance object, and that it attribute reference. For an instance object there are two kinds of attribute names that are valid, and they are data attributes and methods. Data attributes are the same as “instance variables”, there is no need to declare them and like local variables they come into existence when they are assigned a value. The other kind of attribute reference that can be made is a reference to a method defined in the class. A method is a function that is defined in and belongs to a class. Note that the term method is not unique to instance of classes as other data types in python can also have methods like how the list type has the methods like sort, remove, insert, append defined inside it’s definition.

Valid method names on a class instance(i.e an object) are the functions that are defined in the body of the class. All the attributes of a class which is a function object are “methods” for the the instances of that class. So from our previous example of the class “SimpleClass”, if “obj1” is an instance object of this class then valid attribute references would be

- obj1.a
- Obj1.b
- Obj1.c
- Obj1.sayHello

And invalid attribute references are the references to names that have not been defined in the class definition like x, var1, func, etc. Take a look at the below example that was executed in the python shell.

```
In [2]: obj1 = SimpleClass(1, 2, 3)

In [3]: obj1.a # Valid attribute reference
Out[3]: 1

In [4]: obj1.b # Valid attribute reference
Out[4]: 2

In [5]: obj1.c # Valid attribute reference
Out[5]: 3

In [6]: obj1.sayHello() # Valid attribute reference
Out[6]: 'Hello user!'

In [7]: obj1.func() # Invalid attribute reference
Traceback (most recent call last):

  File "<ipython-input-7-77114d42163c>", line 1, in <module>
    obj1.func() # Invalid attribute reference

AttributeError: 'SimpleClass' object has no attribute 'func'
```

6.1.4 Method object

Usually methods of an instance object are invoked right after it gets bound to this object for example a method can be called as follows

```
In [2]: obj1 = SimpleClass(10, 20, 30) # obj1 is an instance object

In [3]: obj1.sayHello()                # This is a method object
Out[3]: 'Hello user!'
```

Method objects aren't necessarily invoked right away, they can also be stored and invoked later in the program like in the example below where the method object of the "obj1" class instance is stored in a local variable and used later in the following statements.

```
In [4]: methObj1 = obj1.sayHello

In [5]: methObj1() # The name methObj1 has a reference to the method object
Out[5]: 'Hello user!'
```

In the above statement the variable "methObj1" has a reference to the method object "sayHello" of the instance object "Obj1"

Let's go through what exactly happens on a method call in python. In the previous example the statement `obj1.sayHello()` was used. This statement invokes the `sayHello()` method of the `obj1` instance object. You may have noticed that this method is being invoked without any arguments passed although the method definition inside the definition of the "SimpleClass" type specifies the function to be called by passing the argument "self". The python interpreter raises an error if there is a mismatch in the number of positional arguments in the call and the number of positional parameters specified in the function invocation. The interpreter doesnt raise an error here because unlike normal functions, when a method is called the instance object is passed as the first argument of the method, and therefore the call `Obj1.sayHello()` is exactly the same as `SimpleClass(obj1)`. Generally speaking we can say that calling a method with a list of 'n' arguments is the same as calling it's corresponding function with the same list of arguments with the instance objects added at the beginning of the list.

To put it more clear the process of invoking an objects method is given below.

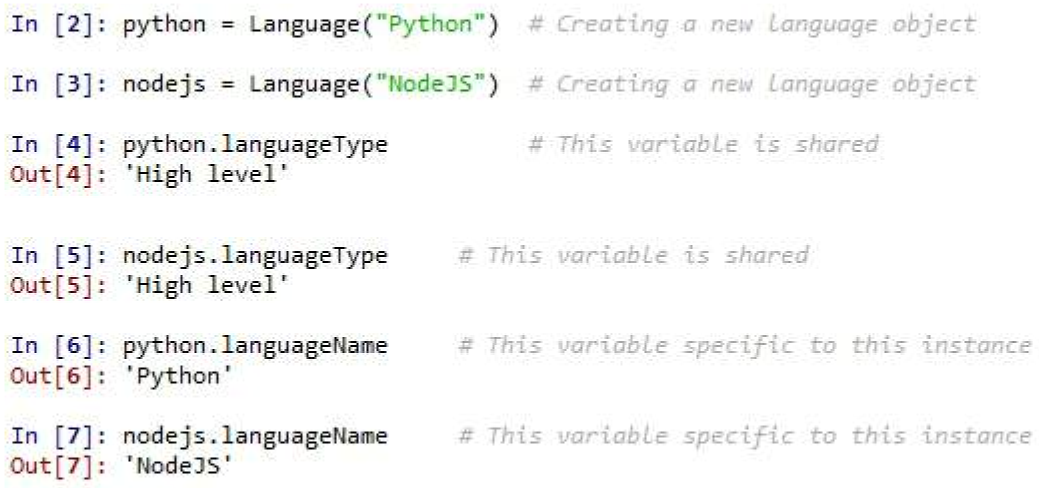
- If a non data attribute of an instance object is used then this instance object's class is searched.
- If this name is a valid class attribute which is a function object then an abstract object is made by putting together the instance object and the abstract object that was just found.
- If a method object s called along with an argument list then create a new argument list using the instance object and the argument list then call the function object with this new argument list that was created.

6.1.5 Class and instance variables

Instance variables are the data attributes that are unique(doesn't mean different) to each object instance, and class variables are the data attributes and method attributes that are shared by all instances of the class. Lets take a look at an example that illustrates instance and class variables.

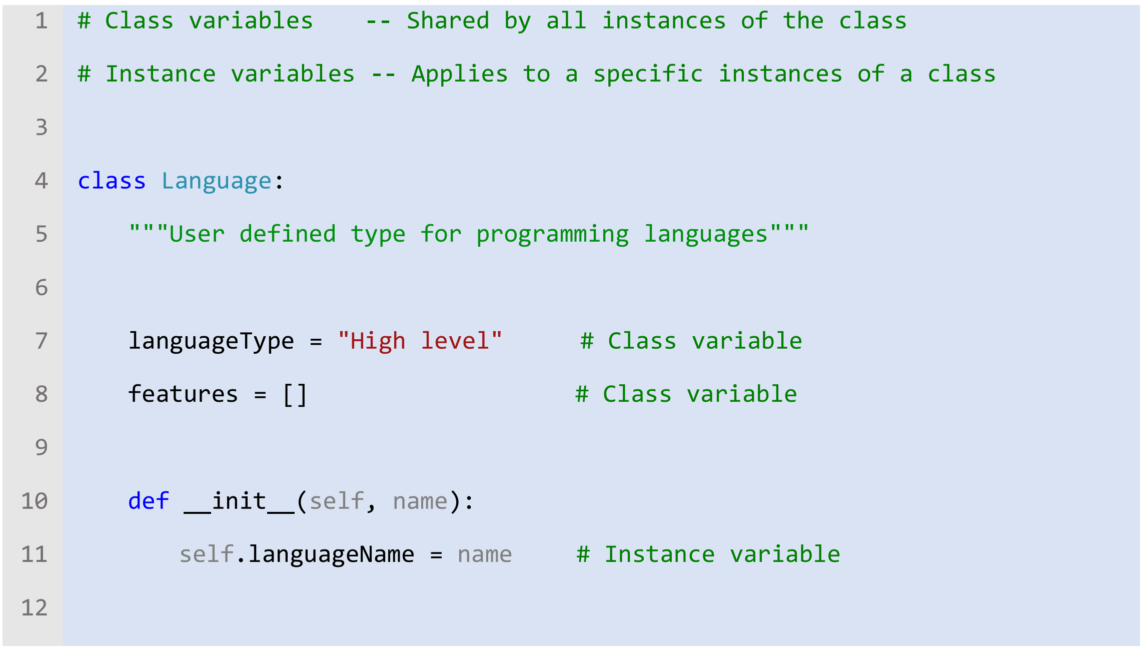
```python
# Class variables    -- Shared by all instances of the class
# Instance variables -- Applies to a specific instances of a class

class Language:
    """User defined type for programming languages"""

    languageType = "High level"   # Class variable

    def __init__(self, name):
        self.languageName = name  # Instance variable

```

The above python code defines a class “Language” which two data attributes. The variable “languageType” is a class variable that has the same value for every instance object that is created from this class definition. The variable “languageName” is an instance type which is specific to an instance object which means that the value of this instance variable depends on the argument that is passed by the caller of the class object to create an instance object. Let’s create some object instances of the above class and check the values of the instance variables and the class variables of the objects that were create. In the below examples from the python shell see how the class variables of all the instance objects are same for all the newly created objects.



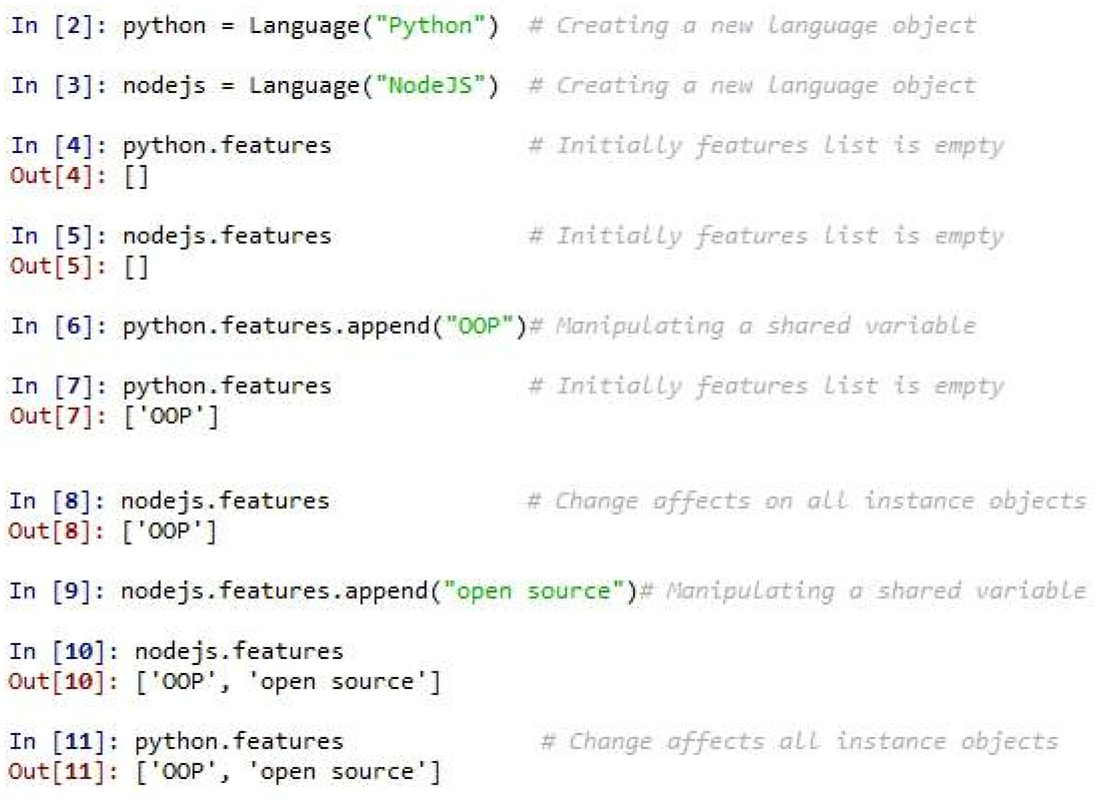
```
In [2]: python = Language("Python")  # Creating a new language object

In [3]: nodejs = Language("NodeJS")  # Creating a new language object

In [4]: python.languageType          # This variable is shared
Out[4]: 'High level'


In [5]: nodejs.languageType     # This variable is shared
Out[5]: 'High level'

In [6]: python.languageName     # This variable specific to this instance
Out[6]: 'Python'

In [7]: nodejs.languageName     # This variable specific to this instance
Out[7]: 'NodeJS'
```

When the value of a class variable is reassigned in an instance object, this change is noticed by all instance objects of that class as the class variable is shared by all the instance objects, shared meaning there the object that is referenced by the class variable is the same object for every instance object. On the other hand an instance objects, instance variable refers to an object that is different from objects referenced by the instance variables of other instance objects. This can be made more clear by going through the example shown below.



```
# Class variables    -- Shared by all instances of the class
# Instance variables -- Applies to a specific instances of a class

class Language:
    """User defined type for programming languages"""

    languageType = "High level"      # Class variable
    features = []                    # Class variable

    def __init__(self, name):
        self.languageName = name     # Instance variable

    def addFeature(self, feature):

        self.feature.append(feature) # Class variable

```

In the above program a new class variable “features” was added to the “Language” class. This class variable represents the list of features of the programming language as this variable is shared by all the instance objects, when the list s appended with a feature by an instance object, the value of the list is changed for all the instance objects of that class. Although in reality this is obviously not true for programming languages the example is just used for the sake of explanation. The statements below that were executed on the python shell illustrates this behaviour of class variables. In other programming languages like C# , this class variables are similar to “static variables”.



```
In [2]: python = Language("Python")  # Creating a new language object

In [3]: nodejs = Language("NodeJS")  # Creating a new language object

In [4]: python.features              # Initially features list is empty
Out[4]: []

In [5]: nodejs.features              # Initially features list is empty
Out[5]: []

In [6]: python.features.append("OOP")# Manipulating a shared variable

In [7]: python.features              # Initially features list is empty
Out[7]: ['OOP']


In [8]: nodejs.features              # Change affects on all instance objects
Out[8]: ['OOP']

In [9]: nodejs.features.append("open source")# Manipulating a shared variable

In [10]: nodejs.features
Out[10]: ['OOP', 'open source']

In [11]: python.features              # Change affects all instance objects
Out[11]: ['OOP', 'open source']
```

6.1.6 Inheritance

Inheritance is an important feature f object oriented programming, so python classes do support inheritance. The general python syntax to define a derived class is given below.

```
class <name of derived type>(<name of the base type>):

    <statement 1>
    .

    .

    .

    <statement n>
```

It’s necessary for the “base class” to be defined in a scope that contains the definition of the “derived class”. python allows using other arbitrary expressions to be used in place of “`<name of the base type>`” in the class header of the derived class. This enables us to derive from classes that are defined in other modules like the class header shown below.

```
class <name of derived type>(<module name>.<name of the base type>):
```

The execution of a derived class definition is the same as the execution of the base class definition. When a class object is created for a derived class the base class is remembered along with it for the sake of resolving attribute references. When an attribute is requested, the attribute is first searched in the derived class, if it’s not found then the derived class’s base class is searched for the requested attribute, as long as the requested attribute is not found and the inheritance chain didn’t end this search process recursively continues upward along the inheritance chain.

The instantiation of a derived class is the same as the instantiation of a normal class. Calling `<name of derived type>()`, creates a new instance of the derived class. Resolving method references proceeds as follows, the corresponding class attribute is searched starting from the derived class and the search continue up the inheritance chain until the method is found or the end of the chain is reached in which case the python interpreter raises an error that the requested attribute is not defined.

When derived classes define a method attribute that has already been defined in it’s base class the method defined by the base class is considered by the interpreter. As there is no priority for calls to methods in the same class, there is a possibility that a method call may end up calling a method from one of it’s sub-classes.

There could be cases where a derived classes method that overrides a method from a base wants to extend the base class method instead of replacing the method. In such cases the base class method can be called with the following syntax given that the base class is accessible from the scope it is being used in.

```
<base class name>.<method name>(self,<argument list>)
```

There are two built in functions that python provides that work with inheritance and they are listed and described briefly in the table below.

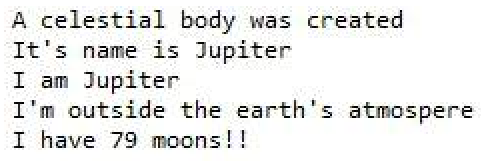
Method	**Description**

`isinstance()`	This function is used to check an instance object's type. <u>Example:</u> `isinstance(obj, int)` The above function call returns a true only when the type of the object "obj" is "int"

`issubclass()`	This function is used check whether the first argument is the subclass of the second argument and on this basis returns either a "True" or "False"
	
	<u>Example:</u>
	
	`issubclass(bool, int)`
	
	The above function call returns "True" because the type "bool" is a subclass of type "int"
	
	`issubclass(str, int)`
	
	The above function call returns "False" because the type "str" is not a subclass of type "int"

Lets take a look at an example program that illustrates how inheritance works in python

```python
#===================================================================
# Python program that illustrates how inheritance works in python
#===================================================================

#------------------------class CelestialBody----------------------
class CelestialBody:
    """All the data attributes and method attributes
    of this class are inherited by the class "DerivedClass" """

    def __init__(self):
        "The constructor of the class"
        print("A celestial body was created")


    def whoAmI(self):
        print("I am a celestial body")

    def speak(self):
        print("I'm outside the earth's atmosphere")
#------------------------class CelestialBody-----------------------

#------------------------class Jupiter-----------------------------
class Jupiter(CelestialBody):
    """This class inherits the
    attributes of the class "BaseClass"
    defined above"""

    def __init__(self):
        "The constructor of the class"
        super().__init__() # Call the constructor of the parent
        print("It's name is Jupiter")

    def whoAmI(self):
        """This method overrides the
        whoisThis() method of the parent class"""
        print("I am Jupiter")

    def brag(self):
        print("I have 79 moons!!")
#------------------------class Jupiter-----------------------------

#----------------Instantiation and method calls--------------------
jupyter = Jupiter()   # Calls Jupiter's constructor and creates an
                     # creates a Jupiter instance object
jupyter.whoAmI()      # This method is defined in class Jupiter
jupyter.speak()       # This method is defined in class CestialBody
jupyter.brag()        # This method is defined in class Jupiter
#----------------Instantiation and method calls--------------------
```
The output of the above program is given below.



```
A celestial body was created
It's name is Jupiter
I am Jupiter
I'm outside the earth's atmospere
I have 79 moons!!
```

In the above program there are two classes defined which are “CelestialBody”(the base class) and “Jupiter”(the derived class). An instance object was created from the class object of class “Jupiter”. as Jupiter inherits from the class “CelestialBody” all it’s data attributes and method attributes are inherited by class “Jupiter”, which make them accessible by instances of “Jupiter” objects. This explains why the instance object “jupyter” can invoke the method `speak()` which wasn’t defined in class Jupiter’s definition. The whoAmI() method that the instance object invokes is it’s own method and not the method of it’s parent(CelestialBody) as method attribute resolution starts from the bottom(class Jupiter) of the inheritance chain and continues to the top(class CelestialBody). The `__init__()` method of class Jupiter uses the super() function which allows an instance object to specifically access the attributes of it’s base class. The `brag()` method of the Jupiter class extends the functionality of the class “CelestialBody”.

6.1.7 Multiple inheritance

Python also allows a form of “multiple inheritance” which enables a class to inherit from more than one base class the general syntax for a class definition of a derived class that inherits from more than one base class is shown below.

```
class <name of derived type>(<base 1>, <base 2>, . . .,<base x> ):

    <statement 1>

    .

    .

    .

    <statement n>
```

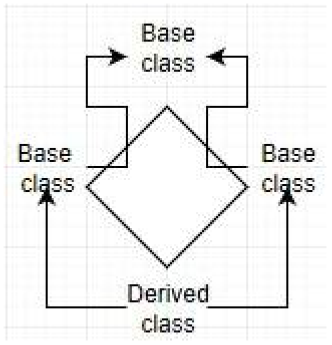
For the simplest and most common cases the search for attributes that were inherited from a parent is a depth first search that doesn’t search the same class twice in case of an overlap in the hierarchy. The process of search for an attribute proceeds as follows.

1. Search for the attribute in the derived class if the requested attribute is not found then do step 2.

2. Search for the requested attribute in the left most parent in the derived classes header which in this case is “base-1”, if the attribute is not found then do step 3.

3. Search recursively in all the base classes of "base-1". If the requested attribute is not found in then the next base class in the base class list of the derived class is searched as specified in steps 2 and 3.

The method resolution order is slightly more complex than the resolution approach described above. The order for resolving methods changes dynamically for the purpose of supporting cooperative invocations to the `super()` function. This approach in other programming languages is referred to as "call next method" and is in many ways better than the "super call" that is found in several "single inheritance" programming languages. Allowing the feature of multiple inheritance in a programming language creates the possibility of the problem of "diamond relationships". diamond relationship is an inheritance chain where there is more than one path from a class in this chain to one of it's predecessors as shown in the diagram below.



Dynamic resolution is needed because in python every case of multiple inheritance has at least diamond relationship as for example all classes in python inherit from the `object` class so any multiple inheritance case has more than one path to the `object` class. To prevent base classes from being accessed more than a dynamic algorithm is used to linearizes the order of resolution in a way that the ordering(left to right) that was specified in the derived class's definition.

6.1.8 Private variables

"Private variables" are variables that belong to an object, that can be accessed from within the object but not from outside of the object. Object oriented programming languages like C++, Java, C# and others support private variables, python is not one of those languages, but there is a convention which is used, which is that a name prefixed with an underscore is treated as non-public

Private variables are useful as they avoid name conflicts with the names that are defined by the sub-classes of a class. Although python doesn't support private variables as of version 3.7 there is limited support offered called "name mangling". private variables in other programming languages can't be accessed from outside of a class and so, instance objects of only that specific class can access private variables of that class. This also means that instance objects of derived classes can't access private variables of the base classes. Name mangling on the other hand textually replaces names of a specific format so the sub-classes are unaware of the name's existence.

Identifiers with that are prefixed with at least two leading underscores and at most one trailing underscore(example: `__var`, `__var2_`) are textually replaced with

```
_<classname>__<attribute name>
```

Here `<classname>` is the name of the class with any leading underscores removed. The the replacement is performed regardless of the position of the identifier as long as it occurs within the body of the class.

Name mangling allows derived classes to override methods but without breaking intra-class method calls in the base class. Lets take a look at an example that illustrates "Name mangling" in python.

```python
#------------------------class MapIterable-------------------------
class MapIterable:
    def __init__(self, iterable):
        self.items = []
        self.__mapper(iterable)

    def mapper(self, iterable):
        for item in iterable:
            self.items.append(item)

    __mapper = mapper # Private copy of original the mapper() method

#------------------------class MapIterable-------------------------

#--------------------class MapIterableDerived----------------------
class MapIterableDerived(MapIterable):

    def mapper(self, keys, values):
        # Gives a new signature for the mapper() method
        # but does not break the __init__() method of the base class
        for item in zip(keys, values):
            self.items.append(item)

```

```
#-------------------class MapIterableDerived----------------------

```

The above example would still work even if the class `MapIterableDerived` defines a method called `__mapper()` in the class definition because the identifier `__mapper` in the base class is textually replaced by `_MapIterable__mapper`, and the identifier `__mapper` in the derived class is textually replaced by `_MapIterableDerived__mapper`.

Remember that python supports mangling mainly to avoid accidents. These variables can still be accessed(for both reading and writing) from outside an instance object. Still, accessing these variables from outside of an object in some cases can be useful for debugging.

6.1.9 Encapsulation

An important feature that object oriented programming languages provide is “Encapsulation” which allows restricting access to certain names in a program. Restricting access using name mangling has just been discussed in the previous subsection. In object oriented languages languages the private variables of a class can’t directly be accessed but they can be accessed by using getter and setter methods, this way the freedom to use these variables is limited to what is provided by the getter and setter methods of of each private variable. Let’s take a look at an example below that is given below.

```
#=======================================================================
# Python program that demonstrates encapsulation in python
#=======================================================================

#-----------------------class Subscription---------------------------
class Subscription:

    #Variable

    def __init__(self):
        "The constructor of the class"
        print("You created a new subscription")
```

```python
        self.__billingPeriod = 1 # Default period in months

    def GetBillingPeriod(self):
        print("Current billing period: {}".format(self.__billingPeriod))

    def SetBillingPeriod(self):
        billingPeriod = eval(input("Enter the new billing period: "))
        self.__billingPeriod = billingPeriod
        print("The new billing period: ", self.__billingPeriod)
#-------------------------class Subscription----------------------------


#--------------------Instantiation and method calls---------------------
sub1 = Subscription()


#print("The current billing period: ",sub1.__billingPeriod)
                        # can't directly access the private variable
                        # So interpreter raises error when executed


sub1.GetBillingPeriod()   # Accessing the private variable with the
                        # with the logic provided by the getter
                        # method


sub1.__billingPeriod = 12 # Can't directly access the private variable


print("An attempt was made to change the private variable to 12")
sub1.GetBillingPeriod()


sub1.SetBillingPeriod()   # Accessing the private variable with the
                        # with the logic provided by the setter
                        # method
sub1.GetBillingPeriod()
#--------------------Instantiation and method calls---------------------
```

In the example above a class is created with a constructor and two methods(`GetBillingPeriod` and `SetBillingPeriod`) defined. The instance variable that we want to focus on here is `__billingPeriod`. The feature of private variables in python is partially provided with the help of name mangling. As the instance variable `__billingPeriod` follows the private variable syntax as it is prefixed with two underscores, it will be considered for name mangling.

Name mangling makes a textual replacement of a variable name as discussed in the previous section and therefore when the the instance object is created from the "subscription" class, the instance variable it possesses is `_Subscription__billingPeriod` and not `__billingPeriod` which is the name that is specified in "subscription" class definition.

When the method `GetBillingPeriod()` of an instance object is invoked,it prints the current billing period of the object which can be accessed through the variable `_Subscription__billingPeriod`. When the SetBillingPeriod() method of an instance object is invoked the user is prompted for an input which is then assigned to the variable `_Subscription__billingPeriod`, which is then printed back to the user.

Remember that because of the name mangling feature the variable name `__billingPeriod` does not exist in the `Subscription` class object as it was textually replaced by `_Subscription__billingPeriod`. As this variable doesn't exist it cant be accessed from outside it's instance object, not even by the classes that inherit from the `Subscription` class.

When instance object "sub1" is created, the `__init__()` method is called which assigns the numerical value 1 to the instance variable `_Subscription__billingPeriod`, note that as of now the name `__billingPeriod` does not exist in the scope of the instance object. When this name is accessed from within the class definition, name mangling textually replaces this name as previously discussed, and so when a private variable is accessed from within the class as in line 20 of the above program

```
20          self.__billingPeriod = billingPeriod
```

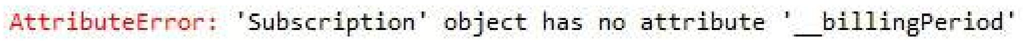
This is seen by the interpreter as

```
20          self._Subscription__billingPeriod = billingPeriod
```

When we try to read the value of `__billingPeriod` from outside the class definition the python interpreter raises an error is as the name was does not exist `__billingPeriod` yet.

```
27  print("The current billing period: ", sub1.__billingPeriod)
```

If the above line of python code is encountered by the interpreter, it raises the following error.

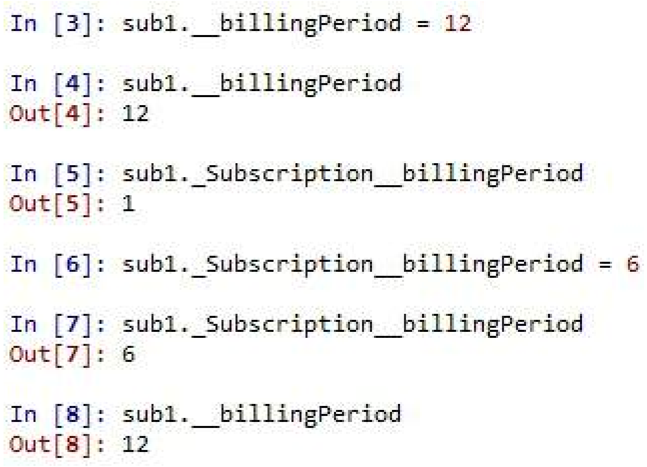


```
AttributeError: 'Subscription' object has no attribute '__billingPeriod'
```

When we try to assign a value to the name , a new name `__billingPeriod` is created in memory and is given a reference to the object that was assigned to it. This occurs at line 35.

```
35  sub1.billingPeriod = 12
```

The above line of code creates a new instance variable __billing in the instance object sub1. This variable is different from the private instance variable `_Subscription__billingPeriod`, which was created at the time of instantiation. This can be verified as shown in the shell execution below.



```
In [3]: sub1.__billingPeriod = 12

In [4]: sub1.__billingPeriod
Out[4]: 12

In [5]: sub1._Subscription__billingPeriod
Out[5]: 1

In [6]: sub1._Subscription__billingPeriod = 6

In [7]: sub1._Subscription__billingPeriod
Out[7]: 6

In [8]: sub1.__billingPeriod
Out[8]: 12
```

As direct access of the private variable is prevented by name mangling, private variable `__BillingPeriod` can instead be accessed with the class provided getter and setter methods(`GetBillingPeriod` and `SetBillingPeriod`) of the private variable `__BillingPeriod`.

A possible output of the above program(when user input is 12) is given below.

```
You created a new subscription
Current billing period: 1
An attempt was made to change the private varaible to 12
Current billing period: 1

Enter the new billing period: 12
The new billing period:  1
Current billing period: 1
```

In the next section we'll discuss another important feature that object oriented programming provides.

6.1.10 Polymorphism and method overriding

Literally speaking the term "polymorphism" means the ability to take multiple forms. Polymorphism in python allows a derived class to define methods with the same name as defined in one of their base classes.

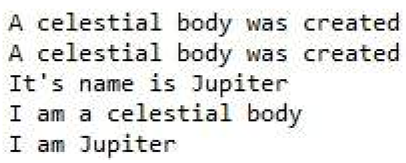
As already discussed a derived class inherits all the attributes of one of it's base classes. There may be cases where some of the methods of the base class that were inherited may not be suitable for objects of the derived class to use in such cases the derived class may want to redefine the method inherited from it's base class. The process of redefining an already existing method which was inherited from a parent class is called as method overriding. As method resolution starts from the base class and proceeds up the chain of inheritance if a method requested is found in the derived class the definition in the derived class is used regardless of whether it has already been defined in the base class

If a method was overriden in the derived class, then the version of the method that will be called depends on the type of the objects that made the method call.In the case that a derived class object is used to call the method then the derived class version of the method is called on the other hand if the base class object is used to call the method that was overriden then the base classes version of this method is called instead. Let's take a look at an example that demonstrates method overriding in python.

```python
#===================================================================
# Python program that demonstrates polymorphism in python
#===================================================================

#------------------------class CelestialBody-----------------------
class CelestialBody:
    """All the data attributes and method attributes
    of this class are inherited by the class "DerivedClass" """

    def __init__(self):
        "The constructor of the class"
        print("A celestial body was created")

    def whoAmI(self):
        print("I am a celestial body")

    def speak(self):
        print("I'm outside the earth's atmosphere")
#------------------------class CelestialBody----------------------


#------------------------class Jupiter----------------------------
class Jupiter(CelestialBody):
    """This class inherits the
    attributes of the class "BaseClass"
    defined above"""

    def __init__(self):
        "The constructor of the class"
        super().__init__() # Call the constructor of the parent
        print("It's name is Jupiter")

    def whoAmI(self):
        """This method overrides the
        whoisThis() method of the parent class"""
        print("I am Jupiter")

    def brag(self):
        print("I have 79 moons!!")
#------------------------class Jupiter----------------------------


#----------------Instantiation and method calls--------------------
celestial_body = CelestialBody() # Instantiating the object
```

```python
jupiter = Jupiter()              # Instantiating the object

celestial_body.whoAmI()          # Calls the base class method
jupiter.whoAmI()                 # Calls the derived class method
#----------------Instantiation and method calls--------------------

```

The above program when executed produces the output shown below.

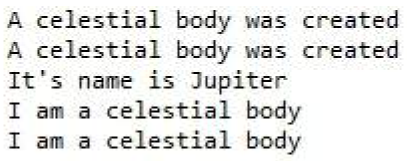


```
A celestial body was created
A celestial body was created
It's name is Jupiter
I am a celestial body
I am Jupiter
```

The celestial_body object is an instance object of type `CelestialBody` and so class “CelestialBody” version of the `whoAmI()` method is called on the other hand the `jupiter` object is an instance object of type “Jupiter” and so the class “Jupiter” version of the `whoAmI()` method is called instead. There are cases which requires using the base class version of the overriden method, in such cases the `super()` function can be used. An example of the use of the super function is shown in the example below.

```python
#==================================================================
# Python program that demonstrates the use of the super() function
#==================================================================

#------------------------class CelestialBody----------------------
class CelestialBody:
    """All the data attributes and method attributes
    of this class are inherited by the class "DerivedClass" """

    def __init__(self):
        "The constructor of the class"
        print("A celestial body was created")

    def whoAmI(self):
        print("I am a celestial body")

    def speak(self):
        print("I'm outside the earth's atmosphere")
#------------------------class CelestialBody-----------------------

#------------------------class Jupiter-----------------------------
class Jupiter(CelestialBody):
    """This class inherits the
    attributes of the class "BaseClass"
    defined above"""

    def __init__(self):
        "The constructor of the class"
        super().__init__() # Call the constructor of the parent
        print("It's name is Jupiter")

    def whoAmI(self):
        """This method overrides the
        whoisThis() method of the parent class"""
        print("I am Jupiter")

    def brag(self):
        print("I have 79 moons!!")
#------------------------class Jupiter-----------------------------

#----------------Instantiation and method calls--------------------
celestial_body = CelestialBody() # Instantiating the object
jupiter = Jupiter()             # Instantiating the object

celestial_body.whoAmI()         # Calls the base class method
jupiter.whoAmI()                # Calls the derived class method
#----------------Instantiation and method calls--------------------

```

In the above program that now the `whoAmI()` method of class Jupiter in turn makes a call to the `whoAmI()` method of class "CelestialBody" using the `super()` function The output when the above program is executed is given below.



```
A celestial body was created
A celestial body was created
It's name is Jupiter
I am a celestial body
I am a celestial body
```

Polymorphism provides the ability to use a common interface for varying functionalities. Let's say for example there is a requirement to add a fill color for a shape, and although there could be several different shapes the same method name can be invoked to fill color to the given shape, regardless of what shape it is. Polymorphism provides this ability. Let's take a look at another example to make things clear.

```python
#=======================================================================
# Python program that illustrates polymorphism in python
#=======================================================================

#------------------------class CelestialBody-----------------------
class CelestialBody:
    """All the data attributes and method attributes
    of this class are inherited by the class "DerivedClass" """

    def __init__(self):
        "The constructor of the class"
        print("A celestial body was created")

    def whoAmI(self):
        print("I am a celestial body")

    def speak(self):
        print("I'm outside the earth's atmosphere")
#------------------------class CelestialBody-----------------------

#------------------------class Jupiter-----------------------------
class Jupiter(CelestialBody):
    """This class inherits the
    attributes of the class "BaseClass"
    defined above"""

    def __init__(self):
        "The constructor of the class"
        super().__init__() # Call the constructor of the parent
        print("It's name is Jupiter")

    def whoAmI(self):
        """This method overrides the
        whoisThis() method of the base class"""
        print("I am Jupiter")

    def brag(self):
        print("I have 79 moons!!")
#------------------------class Jupiter----------------------------

#------------------------class Saturn-----------------------------
class Saturn(CelestialBody):
    """This class inherits the
    attributes of the class "BaseClass"
    defined above"""

    def __init__(self):
        "The constructor of the class"
        super().__init__() # Call the constructor of the parent
        print("It's name is Saturn")

    def whoAmI(self):
        """This method overrides the
        whoisThis() method of the base class"""
        print("I am Saturn")

    def brag(self):
        print("My rings are made up of ice")
#------------------------class Saturn-----------------------------

```

```
#---------------------function definition--------------------------
def callBrag(Celestial_Body):
    """This function is a common interface,
    it calls brag method of the Celestial body
    passed as an argument"""
    Celestial_Body.brag()

#---------------------function definition--------------------------

#---------------Instantiation and function calls-------------------
# Instantiating the objects
jupiter = Jupiter()
saturn = Saturn()

# Pass the objects as arguments to the common interface
# observe that the appropriate method instance ins called
# based on the argument passed
callBrag(jupiter)
callBrag(saturn)

#---------------Instantiation and function calls-------------------
```

The Above program defines three class `CelestialbBody`, `Jupiter`, `Saturn` with the class CelestialBody as the base class from which both classes `Jupiter` and `Saturn` inherit. Both these two class have a common method `brag()`, However the what these two methods do is different. To show polymorphism a common function was defined named `callBrag()` that takes an object of type `CelestialBody` and as class classes `Jupiter` and `Saturn` inherit from class `CelestialBody`, instance objects of these two classes make for valid arguments to this common function `callBrag()`.

When the callBrag() function is called passing the instance objects of type Jupiter and type Saturn the appropriate method is invoked by the interpreter. The output of the above program when executed is given below.

```
A celestial body was created
It's name is Jupiter
A celestial body was created
It's name is Saturn
I have 79 moons!!
My rings are made up of ice
```

6.2 The object Class

All classes in python whether user defined or inbuilt implicitly inherit from the object class which makes the following two class definitions the same.

```python
# Here both the classes X and Y inherit from the object class

class X:         # Implicitly inherits from the object class
    pass

class Y(object):  # Explicitly inherits from the object class
    pass

```

The object class in python is an inbuilt class that has special methods who's names have two preceeding and trailing underscores that are inherited by all the classes. Three most important classes that are provided by the object class are given below.

1. `__new__()`

2. `__init__()`

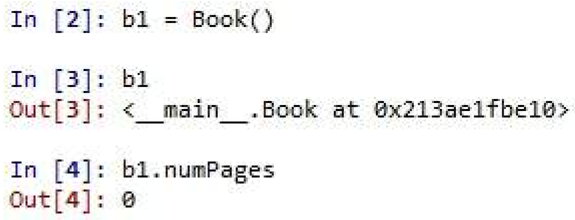
3. `__str__()`

The __new__() method of the object class is defined to create the new instance object, then make a call to the __init__() method so the attributes of the object are initialized. The __new__() method of the object class isn't usually overridden but if there is a need to change the way an instance object is created it most definitely can be overridden.

The __init__() method is responsible for initializing the attributes of the newly created instance object. This method is called from the __new__() method after the object has been created. The __init__() method is almost always overriden to specify how the attributes of the instance object will be initialized. Let's look at an example of how the __init__() method can be overriden by a programmer to instantiate the class attributes.

```python
class Book:
    numPages =    0
```

The above class has a single data attribute numPages which is assigned a default value of zero. When an instance object of this class is created the value of it’s variable numPages gets assigned the already specified value zero.

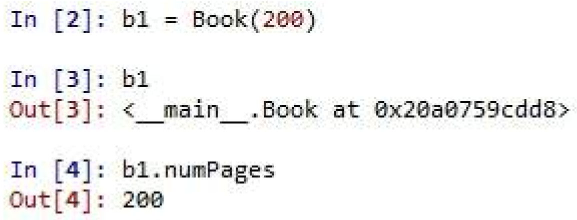


```
In [2]: b1 = Book()

In [3]: b1
Out[3]: <__main__.Book at 0x213ae1fbe10>

In [4]: b1.numPages
Out[4]: 0
```

We can override the __init__() method that our book class inherited from the object class to initialize the variables as needed.

```python
class Book:
    numPages =    0

    def __init__(self, num_pages):
        self.numPages = num_pages

```

The shell execution below after the above program below shows that the class variable numPages was changed when the __new__() method called the __init__() method.

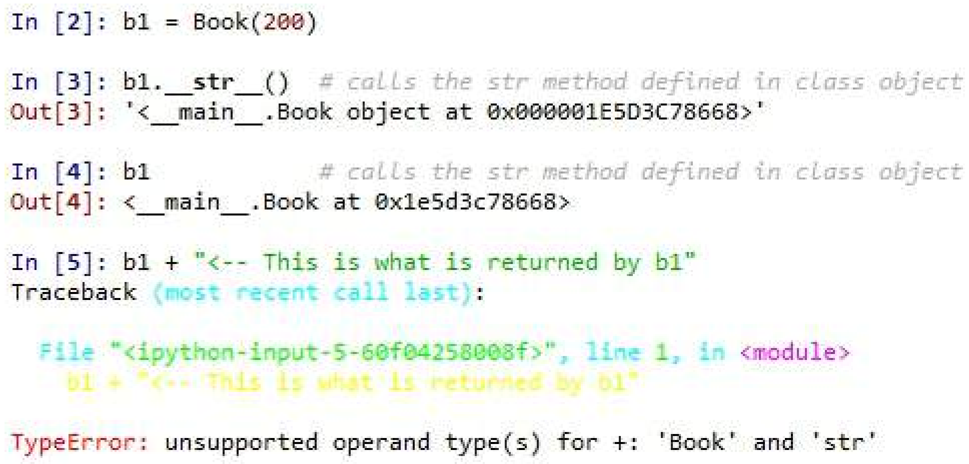


```
In [2]: b1 = Book(200)

In [3]: b1
Out[3]: <__main__.Book at 0x20a0759cdd8>

In [4]: b1.numPages
Out[4]: 200
```

The __str__() method is defined to return a well formatted string representation of the instance object it belongs to. The object class defines the __str__() method to return a string in this format

```
<namespace>.<class name> object at <object’s memory address in hexadecimal>
```

The shell execution below calls the __str__() method for an instance object of the class defined above. Note that simply using an object's name returns the string that is returned by the original __str__() method before overriding(This doesn't mean that it can be used in expressions as a string type).

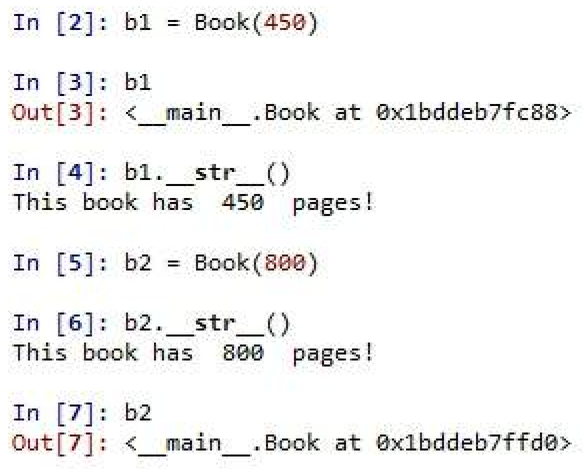


```
In [2]: b1 = Book(200)

In [3]: b1.__str__()  # calls the str method defined in class object
Out[3]: '<__main__.Book object at 0x000001E5D3C78668>'

In [4]: b1            # calls the str method defined in class object
Out[4]: <__main__.Book at 0x1e5d3c78668>

In [5]: b1 + "<-- This is what is returned by b1"
Traceback (most recent call last):

  File "<ipython-input-5-60f04258008f>", line 1, in <module>
    b1 + "<-- This is what is returned by b1"

TypeError: unsupported operand type(s) for +: 'Book' and 'str'
```

This format of the string may not usually be very helpful, so it of course can be changed by overriding the method in the derived class(which is every class in this case as all classes inherit from the object class). An example of this is shown in the class definition given below.

```python
class Book:
    numPages =    0

    def __init__(self, num_pages):
        self.numPages = num_pages

    def __str__(self):
        print("This book has ", self.numPages, " pages!")
```

Now lets make some calls to the __str__() method of instance objects of type "Book", this is shown in the shell execution shown below.



```
In [2]: b1 = Book(450)

In [3]: b1
Out[3]: <__main__.Book at 0x1bddeb7fc88>

In [4]: b1.__str__()
This book has  450  pages!

In [5]: b2 = Book(800)

In [6]: b2.__str__()
This book has  800  pages!

In [7]: b2
Out[7]: <__main__.Book at 0x1bddeb7ffd0>
```

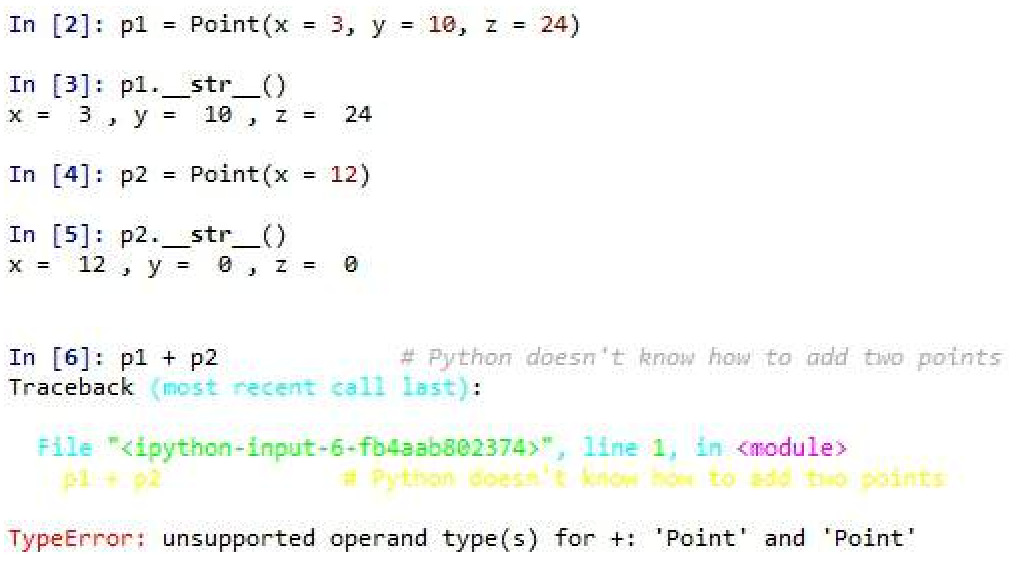
6.3 Operator overloading in python

Python operators work in a different way for different types like how the '+' operator will perform arithmetic addition when it comes to integer types, will merge when it comes to lists and will concatenate for strings.

This feature of python that lets operators to perform differently when it comes to different types is called operator overloading. What does an operator do when it used on user defined types. Lets take a look at this with the user defined type "Point" which is defined below.

```python
class Point:
    def __init__(self, x = 0, y = 0, z = 0):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        print("x = ", self.x, ", y = ", self.y, ", z = ", self.z)

```

When we create two objects out of this class add try to use the '+' operator on them, this is the output obtained.



```
In [2]: p1 = Point(x = 3, y = 10, z = 24)

In [3]: p1.__str__()
x =  3 , y =  10 , z =  24

In [4]: p2 = Point(x = 12)

In [5]: p2.__str__()
x =  12 , y =  0 , z =  0


In [6]: p1 + p2               # Python doesn't know how to add two points
Traceback (most recent call last):

  File "<ipython-input-6-fb4aab802374>", line 1, in <module>
    p1 + p2               # Python doesn't know how to add two points

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

A type error is raised by the python interpreter as it's not known how the '+' operator can be applied to objects of type "Point". This information can be specified by overloading the '+' operator for the class "Point". Before going through how this can be done, lets take a look at special functions in python.

6.3.1 Special functions in python

Any class function that begins with a double underscore is a special function in python. The __init__() function that was defined in our previous class "Point" is a special function, and as already discussed this function __init__ gets called every time a new object is created. There are many special functions that python provides. A class can use special functions to make itself compatible with built in function that python provides. In the Point class example above if the __str__() method hasn't been overloaded then the call to the __str__() method would invoke the __str__() method defined in the object class.

```
class Point:
    def __init__(self, x = 0, y = 0, z = 0):
        self.x = x
        self.y = y
        self.z = z

#   def __str__(self):
#       print("x = ", self.x, ", y = ", self.y, ", z = ", self.z)

```

```
In [2]: p1 = Point(1,2,3)

In [3]: print(p1)
<__main__.Point object at 0x000001C7EAED07F0>
```

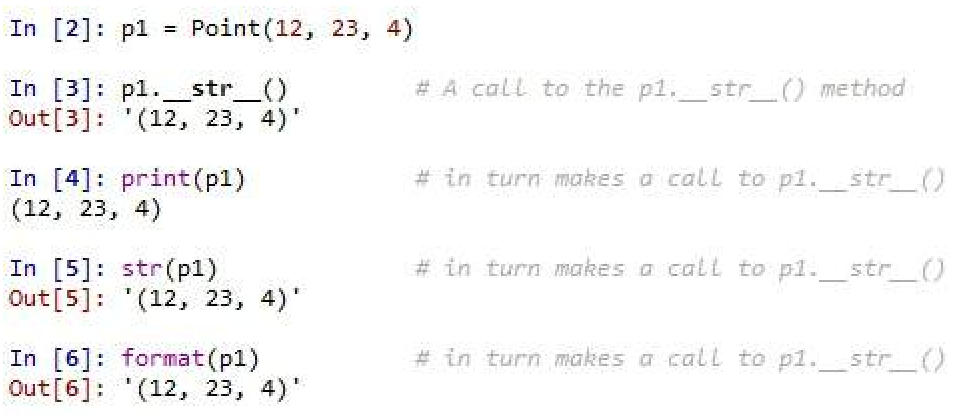
Here the built in function print() prints to the console the value that is returned by the __str__() special function which in this case doesn't seem to provide any useful information. This can be changed by overriding what the __str__method does and what the __str__() method returns to it's caller.

```
class Point:
    def __init__(self, x = 0, y = 0, z = 0):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        """This function returns the point object
        in a tuple format as string
        Example: "(7, 12, 0)" """
        #print("x = ", self.x, ", y = ", self.y, ", z = ", self.z)
        return "({0}, {1}, {2})".format(self.x, self.y, self.z)

```

Observe that calling the object "p1" from the shell makes a call to the definition of the __str__() method in the object class, where as when provided to the print() built in function the print() function makes a call to the __ str__() special method that was implemented by the 'Point" class and concatenates the value returned by __str__() to the string to be printed onto the console.

```
In [2]: p1 = Point(12, 23, 4)

In [3]: print(p1)          # print() inturn makes a call to p1.__str__()
(12, 23, 4)

In [4]: p1.__str__()
Out[4]: '(12, 23, 4)'

In [5]: p1
Out[5]: <__main__.Point at 0x2bdb9caf4e0>
```

This new string that is returned by the __str__() method is gives information that is more relevant to what we may want to know about an instance object of the class Point. This is also the same method that is invoked when the built in functions str() or format() is used.



```
In [2]: p1 = Point(12, 23, 4)

In [3]: p1.__str__()          # A call to the p1.__str__() method
Out[3]: '(12, 23, 4)'

In [4]: print(p1)             # in turn makes a call to p1.__str__()
(12, 23, 4)

In [5]: str(p1)               # in turn makes a call to p1.__str__()
Out[5]: '(12, 23, 4)'

In [6]: format(p1)            # in turn makes a call to p1.__str__()
Out[6]: '(12, 23, 4)'
```

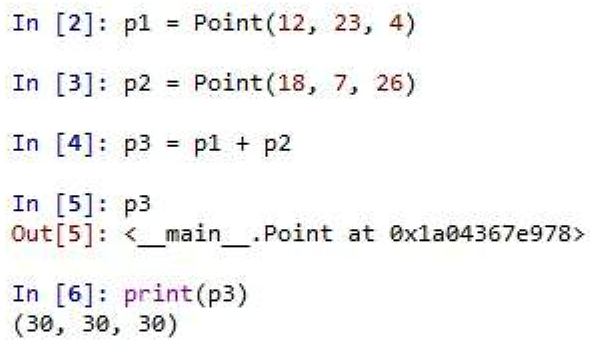
From the above example observe that for str(obj) and format(obj) what is actually done is obj.__str__(), Hence the name "special functions".

6.3.2 Overloading the '+' operator in python

For python to know how to apply the '+ operator on objects of type Point there needs to be an implementation of the __add__() method in the class definition. Let's define our __add__() function to return the point object which is the coordinate sum.

```python
class Point:
    def __init__(self, x = 0, y = 0, z = 0):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        """This function returns the point object
        in a tuple format as string
        Example: "(7, 12, 0)" """
        #print("x = ", self.x, ", y = ", self.y, ", z = ", self.z)
        return "({0}, {1}, {2})".format(self.x, self.y, self.z)

    def __add__(self,other):
        """ This method defines how the +
        operator is applied on two Point objects
        p1 + p2 => p1.__add__(p2) => Point.__add__(p1, p2)"""
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Point(x, y, z)

```

The above definition of the user defined type “Point” implements the special function __add__() which is invoked when the ‘+’ operator is used on two point objects. In the shell execution below observe that now as there is an implementation of the __add__() method in the “Point” class, the ‘+’ operator can now be applied to instance objects of type “Point”. The result of this operation is another point as the __add__() method returns a point object that is the result of the summation.



```
In [2]: p1 = Point(12, 23, 4)

In [3]: p2 = Point(18, 7, 26)

In [4]: p3 = p1 + p2

In [5]: p3
Out[5]: <__main__.Point at 0x1a04367e978>

In [6]: print(p3)
(30, 30, 30)
```

What actually happens here is that when the python interpreter encounters `p1 + p2` , it calls `p1.__add__(p2)`, which in turn calls `Point.__add__(p1, p2)`. Just as we have overloaded the ‘+’ operator other operators can be overloaded as well. The table given below lists all the special functions that can be implemented.

Operator	Use in an Expression	What it means
Addition	`x + y`	`x.__add__(y)`
Subtraction	`x - y`	`x.__sub__(y)`
Multiplication	`x * y`	`x.__mul__(y)`
Power	`x ** y`	`x.__pow__(y)`
Division	`x / y`	`x.__trudiv__(y)`
Floor division	`x // y`	`x.__floordiv__(y)`
modulo	`x % y`	`x.__mod__(y)`
Bitwise left shift	`x << y`	`x.__lshift__(y)`
Bitwise right shift	`x >> y`	`x.__rshift__(y)`

Bitwise AND	`x & y`	`x.__and__(y)`
Bitwise OR	`x | y`	`x.__or__(y)`
Bitwise XOR	`x ^ y`	`x.__xor__(y)`
Bitwise NOT	`~x`	`x.__invert__(y)`

6.3.3 Overloading comparison operators in python

The ability to override operators doesn't end at with just arithmetic operators, python allows overloading comparison operators as well. Let's take an example of implementing the "greater than or equal to" (>=) operator for the class "Point" that was previously defined.

For any two point objects '`x`' and '`y`', for python to be able to evaluate the expression "`x >= y`", an implementation of the special function `__ge__()` must be present in the body of the "Point" class definition. Lets define this function to return "`True`" if the magnitude of '`x`' from the origin is greater than or equal to the magnitude of 'y' from the origin and "False" in the other case. The magnitude of a point `(x, y, z)` from the origin `(0, 0, 0)` is given by,

$$magnitude(x, y, z) = x^2 + y^2 + z^2$$

The class "Point" with the implementation of the special function `__ge__()` is given below.

```python
class Point:
    def __init__(self, x = 0, y = 0, z = 0):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        """This function returns the point object
        in a tuple format as string
        Example: "(7, 12, 0)" """
        #print("x = ", self.x, ", y = ", self.y, ", z = ", self.z)
        return "({0}, {1}, {2})".format(self.x, self.y, self.z)

    def __add__(self,other):
        """ This method defines how the +
        operator is applied on two Point objects
        p1 + p2 => p1.__add__(p2) => Point.__add__(p1, p2)"""
        x = self.x + other.x
        y = self.y + other.y
        z = self.z + other.z
        return Point(x, y, z)

    def __ge__(self,other):
        """ This method defines how the +
        operator is applied on two Point objects
        p1 >= p2 => p1.__ge__(p2) => Point.__ge__(p1, p2)"""
        # Calculate the magnitude of self
        self_magnitude = (self.x ** 2) + (self.y ** 2) + (self.z ** 2)
        # Calculate the magnitude of other
        other_magnitude = (other.x ** 2) + (other.y ** 2) + (other.z ** 2)
        return self_magnitude >= other_magnitude

```

For the above class definition lets take a look at some example statements with expressions that involve the “greater than or equal to” comparison operator in the python shell.

```
In [2]: p1 = Point(123, 223, 543)  # A point far from the origin

In [3]: p2 = Point(6, 12, 25)      # A point closer to the origin

In [4]: p1 >= p2                   # Here, p1 is greater than p2
Out[4]: True

In [5]: p2 >= p1                   # Here, p2 is less than p1
Out[5]: False

In [6]: Point(-1, 0 , -1) >= Point(10, 10, 10)
Out[6]: False
```

The special function that need to be implemented to overload comparison operators are listed in the table below.

Operator	Use in an Expression	What it means
Less than	`x < y`	`x.__lt__(y)`

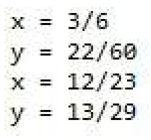
Less than or equal to	`x <= y`	`x.__le__(y)`
Greater than	`x > y`	`x.__gt__(y)`
Greater than or equal to	`x >= y`	`x.__ge__(y)`
Equal to	`x == y`	`x.__eq__(y)`
Not equal to	`x != y`	`x.__ne__(y)`

6.4 Exercises

Some of the exercise questions in this exercise involve the type definition for rational numbers that is available in the python program given below.

```python
#================================================================
# RationalNumber.py
#================================================================
class RationalNumber:
    """
    Objects of this class can represent a rational number
    """
    def __init__(self, n, d):
        self.__numerator = n       # Private variable
        if d != 0:
            self.__denominator = d # Private variable
        else:
            print("It's not a good idea to divide by zero")

    def GetNumerator(self):
        """ Returns the numerator of the fraction. """
        return self.__numerator

    def GetDenominator(self):
        """ Returns the denominator of the fraction. """
        return self.__denominator


    def SetNumerator(self, num):
        """ Sets the numerator of the fraction to n. """
        self.__numerator = num

    def SetDenominator(self, den):
        """
        Sets the denominator to den,unless d
        is zero, in which case the method
        prints an error message
        """
        if den != 0:
            self.__denominator = den
        else:
            print("Error: zero denominator!")

    def __str__(self):
        """
        Make a string representation of a Rational object
        and return it to the caller
        """
        num = self.GetNumerator()
        den = self.GetDenominator()
        return str(num) + "/" + str(den)

# Client code that uses Rational objects
def main():
    x = RationalNumber(3, 6)
    y = RationalNumber(22, 60)

    print("x =", x)
    print("y =", y)

    x.SetNumerator(12)
    x.SetDenominator(23)
    y.SetNumerator(13)
    y.SetDenominator(29)

    print("x =", x)
    print("y =", y)

main()
```

The output for the above program is given below.



```
x = 3/6
y = 22/60
x = 12/23
y = 13/29
```

Some of the exercise questions that are given below refer to the Rational Number class that is defined above and the Point class that was previously defined, feel free to refer to them when needed to get through the exercise.

1. Given the definition of the Rational number class, complete the function named add:

```python
def add(r1, r2):
# Details go here
```

   that returns the rational number representing the sum of its two parameters.

2. Given the definition of the geometric Point class, complete the function named distance:

```python
def distance(r1, r2):
# Details go here
```

   that returns the distance between the two points passed as parameters.

3. Given the definition of the Rational number class, complete the following function named reduce:

```python
def reduce(r):
# Details go here
```

that returns the rational number that represents the parameter reduced to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

4. What is the purpose of the __init__ method in a class?
5. What is the parameter named self that appears as the first parameter of a method?

6. Given the definition of the Rational number class, complete the following method named reduce:

```python
class Rational:
# Other details omitted here ...
# Returns an object of the same value reduced
# to lowest terms
def reduce(self):
# Details go here
```

that returns the rational number that represents the object reduced to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

7. Given the definition of the Rational number class, complete the following method named reduce:

```python
class Rational:
# Other details omitted here ...
# Reduces the object to lowest terms
def reduce(self):
# Details go here
```

that reduces the object on whose behalf the method is called to lowest terms; for example, the fraction 10/20 would be reduced to 1/2.

8. Given the definition of the geometric Point class, add a method named distance:

```
class Point:
# Other details omitted
# Returns the distance from this point to the
# parameter p
double distance(self, p):
# Details go here
```

that returns the distance between the point on whose behalf the method is called and the parameter p.

7. Files and input/output

In this chapter we'll discuss how python can be used to access files and other input and output capabilities of python. We'll start by explaining the "File objects", which includes it's attributes, built-in functions and built-in methods, then we'll discuss in brief about standard files, accessing the file system and persistent storage modules.

7.1 File objects

File objects are what is used to access normal files that are on the disk. They can be used to access anything that has the file abstraction which means that other objects ca be accessed using file-like interfaces in the same way the that normal files on the disk are accessed in python. Files are a simply a contiguous sequence of bytes. If data needs to be sent anywhere this will involve a byte stream, where the byte stream could either be in the form of individual bytes or blocks of data.The unix operating system even use files as an underlying architectural interface for communication. In the following sub sections we will be discussing in detail about the built-in functions, built-in methods and attributes for "File objects".

7.1.1 Built-in function(open())

The open() function is one of the built in functions that are provided by python. It can be used to initiate a file input/output process. If the open() function is able to successfully open a file that was requested then it returns a file object which can be used as a handle for subsequent interactions with the file. If the open() function encountered a problem opening the requested file then this results in an error. When such failure occurs python the raises the exception `IOERROR`. We'll discuss in detail about errors and exceptions in the next chapter. The general syntax of the open() function is given bellow.

```
<file object name> = open(<file name>,
                 access_mode = <access mode>), buffering = <buffer size>)
```

Here `<file object name>` is the name that refers to file object that is returned by the open() function. The `<file name> argument` is of type string and it has the name of the file that the open function needs to open. `<file name>` can be a relative or a full path name. `access_mode` is an optional parameter that can be specified in the call to the open function, it is also of type string. The caller can use it by specifying a flag that indicates which mode to open the file with. Usually there are three modes that a file is opened with, and they are

- "`r`" - read
- "`w`" - write
- "`a`" - append

If the open() function is used on a file in the "read" then the file must exist or there won't be a file to read from. If the open() function is used to open a file in the write mode then the file is truncated(all data in the file is deleted) first if the file exists, then the file gets recreated. If a file that does not exist is opened in the write mode then a new file is created and a handle to it is returned to the caller. If the open function is used to open a file in the append mode, then if the file exists the file is opened and the initial position of the file for write access is set to end of file(EOF). In the case that the file didn't exist when opened in this mode, the file with the provided name is created making it as though it was opened using the write mode. If the access mode is not specified in the call to the open function then the mode is assumed to be "read".

There are some more modes that the fopen() built in function supports that can work with the open() function. Some of these other modes include,

- "`+`" - for read and write access
- "`b`" - for binary access

The other optional argument that can be passed is the "buffering", which can be used to specify the type of buffering that is requested to be performed during the access to the file. The buffer argument is of type int and can have the following mentioned values,

- `0` - implies that there is no buffering
- `1` - implies line buffering
- `>1` - implies a buffered I/O with the specified value for the buffer size

Here as can be seen there is no negative value, which means that the default buffering scheme of the system should be used. In the normal case a buffering value is not provided in which case the system default is used.

Mode of operation	Operation to be performed
`r`	Open to read
`w`	Open to write and truncate if needed

a	Open to write, start at the eof and create file if needed
r+	Open for read and write
w+	Open for read and write, w as mentioned above
a+	Open for read and write , a as mentioned above
rb	Open for binary read
wb	Open for binary write, w as mentioned above
ab	Open for binary append, a as mentioned above
rb+	Open for binary read and write, r+ as mentioned above
wb+	Open for binary read and write, w+ as mentioned above
ab+	Open for binary read and write, a+ as mentioned above

Now lets take a look at some examples of using the built in open() function to obtain files objects as handles that can be used later in the program for reading or writing to the file.

```
fileHandle1 = open('/mod/five')             # This file gets opened for reading
                                            # by default as the access mode is
                                            # not specified in the call

fileHandle2 = open('client', 'w')           # This file gets opened for writing

fileHandle3 = open('MINER', 'r+')           # This file gets opened for both
                                            # Reading and writing

fileHandle4 = open('c:/tell/speak', 'wb')  # This binary file is opened for
                                            #reading

```

In the above examples the names fileHandle1, fileHandle2, fileHandle3, fileHandle4 are assigned references to the file objects that are returned by the open() functions.

7.1.2 Built-in methods of file objects

After the open() function is used to successfully obtain a handle to the file object associated with the requested file, any subsequent accesses to the file can be done using the file object that was returned by the open() function. The built in methods of a file object can be used to perform operations on the file like reading from the file or writing to a file. There are four different categories of methods for a file which are listed below.

- Input
- Output
- Intra-file movement (movement within a file)
- miscellaneous

Input

The following methods are provided by a file object to be used to get input from a file,

- `read()`
- `readline()`
- `readlines()`
- `readinto()` - Not supported

The `read()` method reads bytes directly into a string object reading up to the number of bytes that was specified if available. In the case that there was no size specified the default value of "-1" for size is assumed in which case the entire file is read till the end. The `readline()` method returns the next line from the file that was called upon which means it return all bytes until a NEWLINE character is encountered. This NEWLINE character is also included in the string that is returned back to the caller of the `readline()` method. The `readlines()` method performs a similar operation, the difference being that it returns list of all the lines in the file where each line in the file is a single string element in the returned list. The `readinto()` method reads the number of bytes that was specified in the function call and stores it in a writable buffer object, This is the type of object that is returned by the built in function `buffer()`. `buffer()` is not supported and so the built in function `readinto()` is also unsupported.

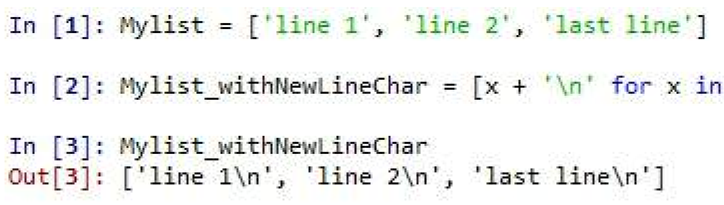
Output

The following methods are provided by a file object to be used to write data to a file,

- `write()`

- `writelines()`

The write() method is a built in method who’s functionality is almost the opposite of the read() and readline() methods. The write() method takes either a string that has one to many lines of text or it could take a block of bytes, then it writes this data that it was passed to the file. Just as how readlines() method operated on a list of strings, so does the write line function. It takes a list of strings and writes them to the file. Writelines does not add a NEWLINE character at the end of every string so it’s the responsibility of the programmer to manually add them to the end of each line before calling the writelines() method . A simple way to add the NEWLINE character at the end of every string in the list is shown in the sell execution below.



```
In [1]: Mylist = ['line 1', 'line 2', 'last line']

In [2]: Mylist_withNewLineChar = [x + '\n' for x in Mylist]

In [3]: Mylist_withNewLineChar
Out[3]: ['line 1\n', 'line 2\n', 'last line\n']
```

The writeline() method isn’t provided as it’s functionality would be the same as calling the write method by passing it a single line string that is terminated with a new line character.

Intra-file movement

The following methods are provided by a file object to be used to write data to a file,

- `seek()`
- `tell()`

The seek() method can be used to move the file pointer to a different position within the file. The seek() method takes the offset in bytes as well as the relative offset position as parameters. The relative offset location can take the following possible values. The value 0 specifies that the distance is taken from the beginning of the file. The offset when measured from the file beginning is called the “absolute offset”. The value 1 specifies that that the relative offset position is the current current location. The value 2 specifies that the relative offset location is the end of the file.

Other built in methods

The file object provides some other miscellaneous built in methods, we’ll discuss a few of them in brief.

The close() method is used when the a file is longer needed, it’s used stop the access to the file by closing the file. The file is also closed when the garbage collector deletes it’s file object when the file objects reference count is reduced to zero. An instance of this case is that if there exists only one reference to a file which was created using the assignment below.

```
fileHandler1 = open(‘x’)
```

Now if the variable `fileHandler1` is assigned another file object to reference, it will no longer refer to the file object that was returned by the above call to the open function on file 'x'. When this happens there are no more variables that refers to this file object, so it's reference count becomes zero. The garbage collector deletes all objects who's reference count is zero, so the file object returned by the `open('x')` is deleted and the file is closed.

The fileno() method can be used to return a file's integer file descriptor. The flush() method flushes the internal buffer for the file. The isatty() returns the value 1 in the case that the file is a tty like device, otherwise it returns a 0. The truncate() method can be used to truncate the file to the specified number of bytes or to zero if number of bytes is not specified.

Now let's take a look at some examples that use built in file objects and methods. The example program that is given below requests the user to enter a file name then it attempts to open the file specified by the user. If it succeeds in opening the file, the readlines() method is used to retrieve all the lines of the file as a list of strings which is stored in a variable, then the file is closed by calling the close method. After the file is closed, all the string in the list are printed to the console.

```
#======================================================================
# Program to print the contents of a  user specified file
#======================================================================

filepath = input("which file: ") # Ask the user for a file path
print(filepath)
fileObj = open(filepath, 'r')    # open the requested file
                                 # in read access mode

lines = fileObj.readlines()      # Read all lines in the file.
                                 # returns a list of strings

fileObj.close()                  # close the file

for line in lines:               # print elements of the "lines" list
    print(line)

```

The above program is different from standard file access because all the lines of the file are read from the file and is available in memory before any line is printed on to the console. This becomes a problem when dealing with very large files as the user is faces a large delay(which is not user friendly) in such cases it is better to take the proper way that is to read and display one line at a time. A possible output of the above program is given below.

```
which file: c:/Users/khalid/Documents/poetry.txt

Not like the brazen giant of Greek fame,

With conquering limbs astride from land to land;

Here at our sea-washed, sunset gates shall stand

A mighty woman with a torch, whose flame

Is the imprisoned lightning, and her name
```

The example program below solves the problem that we could have had with the above program by printing a line as soon as it is read from the file.

```python
#=================================================================
# Program to print the contents of a  user provided file
#=================================================================

filepath = input("which file: ")   # Ask the user for a file path.

fileObj = open(filepath, 'r')      # Open the requested file
                                   # with read access mode.

EOF_Flag = 0                       # This file can be used to know
                                   # if the end of the files has been
                                   # reached.

while EOF_Flag != 1:               # Print each line of the file
                                   # until EOF_Flag remains a 0.

    nextLine = fileObj.readline()  # Read the next line of the file

    if nextLine != "":             # If end of file is not reached then
        print(nextLine)           # print the line,
    else:                         # else,
        EOF_Flag = 1                # set the EOF_Flag to 1

fileObj.close()                     # close the file

```

In this program as we are unaware of when the file would end we use a variable EOF_Flag, it's value tells says whether or not the end of the file has been reached. This variable is obviously initially assigned the value 0 (which is represents "false", meaning the end of the file hasn't been reached yet). The termination criteria of the while loop depends on the variable EOF_Flag()

7.1.3 Built-in attributes

In addition to methods file objects also have some data attributes. These attributes have auxiliary data that are related to the file object that these attributes belong to. File objects have four data attributes and they are

1. `file.name` : name of the file
2. `file.closed` : whether the file is closed
3. `file.mode` : access mode that the file was opened with
4. `file.softspace` : this is a flag that tells whether an additional space character is required before printing successive data items using the print statement

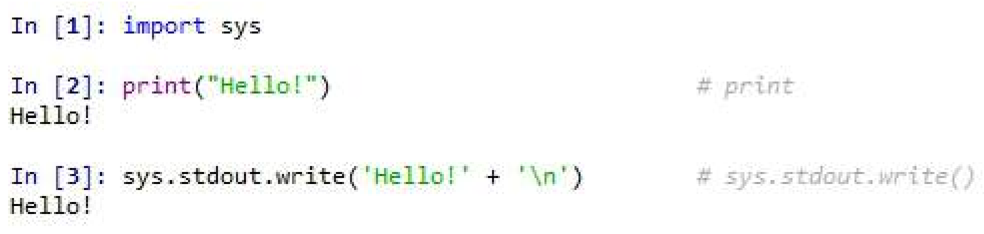
File object attribute	Description
`file.name`	The file's name
`file.closed`	Set to 1 if the file is closed otherwise set to 0
`file.mode`	Access mode that the file was opened with
`file.softspace`	Set to 1 if space is not explicitly required with print otherwise 1

7.2 Standard Files

When a program starts there are three standard files that are made available and they are

1. `stdin` : The standard input file (usually keyboard)
2. `stdout` : The standard output file (buffered output to the display screen)
3. `stderr` : The standard error file (unbuffered output to the display screen)

You may have noticed that here buffered/unbuffered refers to the third argument to the open() function. These names (`stdin, stdout, stderr`) are taken from the c language naming scheme. These standard files are available when the program starts implies that when the program starts these files are opened for you when the program starts and that you can access them using the provided file handlers. The file handlers for these standard files are made available in the sys module. Once you import the sys module, these file handlers can be accessed with the dot operator as (`sys.stdin, sys.stdout, sys.stderr`). The print() function outputs to the stdout file.



```
In [1]: import sys

In [2]: print("Hello!")                        # print
Hello!

In [3]: sys.stdout.write('Hello!' + '\n')      # sys.stdout.write()
Hello!
```

Notice that the sys.stdout.write() method requires that the newline character is specified, but this is not required with the print method.

```
import sys
sys.stdout.write("Enter your string: ")
my_string = sys.stdin.readline()
sys.stdout.write(my_string)
```

In the above program observe that we do not explicitly provide the NEWLINE character as the readline() executed on sys.stdin preserves this readline.

7.3 Command-line Arguments

The sys module through sys.argv provides access to the command-line arguments. Command-line arguments are the arguments that are passed to the program in addition to the script name when the program is ran. These arguments are called command-line arguments because in earlier text based environments these arguments were passed using the command-line. In newer GUI based environments this isn’t the case. IDE’s these days usually provide textbox controls which accept values that are in turn passed as command-line arguments to the program when it is started.

“argv” stands for “argument vector” which is a list of strings of all command-line arguments passed. “argc” stands for argument count which isn’t made available in the sys module, but this argument count can be obtained by applying the len() function on the sys.argv list

- sys.argv can be used to access the list of command -line arguments that were passed to the programs
- len(sys.argv) the integer returned is the number of command-line arguments that were passed

Lets take a look at an example program named “arguments.py” that uses command line arguments

```python
import sys

print("You're arguments =>", str(sys.argv))
print("Argument count = ", len(sys.argv))
```

Here is a possible invocation of the above program and the output of the invocation

```
% arguments.py 12 python 24
You're arguments => [‘12’, ‘python’, ‘24’]
Argument count = 4
```

7.4 The file system

Most of the the access to the file system occurs mainly with the “os” module. It is the primary interface to the operating system’s services and it’s facilities. The os module acts as the front end for the real module that is loaded which is different for the different operating systems like posix for unix, NT for windows, mac for Macintosh, dos for dos, os2 for OS/2.

It’s best if you never import those modules directly. The appropriate module for your operating system is loaded when the os module is imported. Accessing operating system features and facilities through this module is more convenient for programmers and data-scientists as it hides the underlying details. Remember that some features may be unavailable based on the operating system that your using.

In addition to the task of managing processes and process execution environments, the os module performs most of the file system operations which includes removing files, renaming files, traversing the directory tree, managing file accessibility and several other operations which are listed in the tables below.

os module file(file processing)	Operation
`remove() / unlink()`	Delete a file
`rename()`	Rename the file
`*stat()`	Return statistics of the file
`symlink()`	Create a symbolic link
`utime()`	Update time-stamp

os module file(Directories/Folders)	Operation
`chdir()`	Change working directory
`listdir()`	List files in directory
`getcwd()`	Return current working directory
`mkdir()/mkdirs()`	Create directory/directories
`rmdir()/removedirs()`	Remove directory/directories

There os another module that is available that can perform operations on specific paths. This is the os.path module which is available through the os module. This module includes several functions to manipulate and manage file path name components, for obtaining directory or file information, for making file path inquiries and several other functions most important of which are listed and described in the tables below.

os.path Pathname function(separation)	**Operation**
`basename()`	Return leaf name removing directory path
`dirname()`	Return directory path removing leaf name
`join()`	Join separate components into a single pathname
`split()`	return (*dirname(), basename()*) tuple
`splitdrive()`	return (*drivename, pathname*) tuple
`splitext()`	return (*filename, extension*) tuple

os.path Pathname function(information)	**Operation**
`getmtime()`	Return the last file's modification time
`getatime()`	Return the last file's access time
`join()`	Return file's size(in bytes)

os.path Pathname function(inquiry)	**Operation**
`exists()`	Returns whether pathname(file or directory) exists
`isdir()`	Returns whether pathname exists and is it a directory
`isfile()`	Returns whether pathname exists and is it a file
`islink()`	Returns whether pathname exists and is it a symbolic link

`samefile()`	Do both the file name provided point to the same file

7.5 Persistent storage modules

Most of the example programs that we've dealt with so far has required user input. After many repetitions of entering the same data as input over and over this starts to get frustrating, also when the user just entered a lot of information it may not be feasible to ask the user to re-enter that data when the program is run again. In this case it is useful to use the persistent storage to store the data for reuse or archive the entered data, so it can be accessed without re-entry. This is one of the several scenarios where data persistence proves to be very useful.

There are some cases where simple disk files that store strings aren't enough for the task at hand and relational databases might be an overkill for the same task. Persistent storage solves this problem. Most of the persistent storage modules deals only with strings of data, but there are some modules that deal with the persistence of python objects.

7.5.1 Pickle and marshal modules

Python has several modules that enable a programmer to persist information. Two of them are marshal and pickle. The modules marshal and pickle allow pickling of python objects. "Pickling" is the process where objects more complex than primitive type are are converted to a binary set of bytes that can be stored or transmitted over the network and again be converted back to it's original form as python object. Pickling is synonymous with the terms "flattening", "serializing", "marshalling"

The modules pickle and marshal can be used to flatten objects but they don't completely provide persistent storage as they don't give a namespace for the objects and they don't give concurrent write access to the persistent objects. What these python modules are good for is to pickle python objects so they can be stored and/or transmitted over the network and be reconstructed back to it's original form. The difference between marshal and pickle is that marshal can only handle simple python objects like numbers, sequences and mappings while pickle on the other hand can handle almost any kind of python object including user defined classes and object instances, recursive objects and objects that are multi-referenced from different places.

Pickle allows storing python objects to files without the extra step of converting them into strings or writing them out as binary files using low level file access methods. What the pickle modules does is it creates a python only binary version of the object which allows the programmer to easily read and write the object without worrying about all the file level details. All that is required is a valid file handle to read or write objects to the disk.

Pickle module has two main functions and they are,

- dump( ) : This function takes as arguments, a file handle and an object and stores this object to the file that was passed
- load( ) : This function is used to load a stored object back to it's original form that it was in before it was saved to the disk as a file

7.6 Exercises

1: *File Filtering.* Display all lines of a file, except those that start with a pound sign (#), the comment character for Python, Perl, Tcl, and most other scripting languages.

2: *File Access.* Prompt for a number *N* and file *F,* and display the first *N* lines of *F.*

3: *File Information.* Prompt for a filename and display the number of lines in that text file.

4: *File Access.* Write a "pager" program. Your solution should prompt for a file name, and display the text file 25 lines at a time, pausinLg each time to ask the user to "press a key to continue."

5: *Test Scores.* Update your solution to the test scores problems (Exercises 5–3 and 6–4) by allowing a set of test scores be loaded from a file. We leave the file format to your discretion.

6: *File Comparison.* Write a program to compare two text files. If they are different, give the line and column numbers in the files where the first difference occurs.

7: *Parsing Files.* Windows users: create a program that parses a Windows .ini file. Unix users: create a program that parses the /etc/services file. All other platforms: create a program that parses an system file with some kind of structure to it.

8: *Module Introspection.* Extract module attribute information. Prompt the user for a module name (or accept it from the command-line). Then, using dir() and other built-in functions, extract all its attributes, and display their names, types, and values.

9: "*PythonDoc.*" Go to the directory where your Python standard library modules are located. Examine each .py file and determine whether a __doc__ string is available for that module. If so, format it properly and catalog it. When your program has completed, it should present a nice list of those modules which have documentation strings and what they are. There should be a trailing list showing which modules do not have documentation strings (the shame list). EXTRA CREDIT: extract documentation for all classes and functions within the standard library modules.

10: *Home Finances.* Create a home finance manager. Your solution should be able to manage savings, checking, money market, certificate of deposit (CD), and similar accounts. Provide a menu-based interface to each account as well as operations such as deposits, withdrawals, debits, and credits. An option should be given to a user to remove transactions as well. The data should be stored to file when the user quits the application (but randomly during execution for backup purposes).

11: *Web Site Addresses.*

(a)Write a URL bookmark manager. Create a text-driven menu-based application which allows the user to add, update, or delete entries. Entries include a site name, Website URL address, and perhaps a one-line description (optional). Allow search functionality so that a search "word" looks through both names and URLs for possible matches. Store the data to a disk file when the user quits the application, and load up the data when the user restarts.

(b)Upgrade your solution to part (a) by providing output of the bookmarks to a legible and syntactically correct HTML file (.htm or .html) so that users can then point their browser to this output file and be presented with a list of their bookmarks. Another feature to implement is allowing the creation of "folders" to allow grouping of related bookmarks. EXTRA CREDIT: Read the literature on regular expressions and the Python re module. Add regular expression validation of URLs that users enter into their database.

12: *Users and Passwords.*

(a)Do Exercise 7-5, which keeps track of usernames and passwords. Update your code to support a "last login time." See the documentation for the time module to obtain timestamps for when users "login" to the system. Also, create the concept of an "administrative" user which can dump a list of all the users, their passwords (you can add encryption on top of the passwords if you wish), and their last login times. The database should be stored to disk, one line at a time, with fields delimited by colons

(:), i.e., "joe:boohoo:953176591.145," for each user. The number of lines in the file will be the number of users which are part of your system.

(b)Further update your example such that instead of writing out one line at a time, you "pickle" the entire database object and write that out instead. Read the documentation on the pickle module to find out how to "flatten" or "serialize" your object, as well as how to perform I/O using picked objects. With the addition of this new code, your solution should take up fewer lines than your solution in part (a).

13: *Command-line arguments.*

(a)What are they, and why might they be useful?

(b)Write code to display the command-line arguments which were entered.

14: *Logging Results.* Convert your calculator program (Exercise 5-6) to take input from the command-line, i.e., % calc.py 1 + 2 Output the result only. Also, write each expression and result to a disk file. Issuing a command of…

```
% calc.py print
```

reset/truncated. Here is an example session:

```
% calc.py 1 + 2

3

% calc.py 3 ^ 3

27

% calc.py print

1 + 2

3

3 ^ 3

27

% calc.py print

%
```

15: *Copying Files.* Prompt for two file names (or better yet, use command-line arguments). The contents of the first file should be copied to the second file.

16: *Text Processing.* You are tired of seeing lines on your e-mail wrap because people type lines which are too long for your mail reader application. Create a program to scan a text file for all lines longer than 80 characters. For each of the offending lines, find the closest word before 80 characters and break the line there, inserting the remaining text to the next line (and pushing the previous next line down one). When you are done, there should no longer be lines longer than 80 characters.

17: *Text Processing.* Create a crude and elementary text file editor. Your solution is menu-driven, with the following options: (1) create file [prompt for file name and any number of lines of input], (2) display file [dump its contents to the screen], (3) edit file (prompt for line to edit and allow user to make changes), (4) save file, and (5) quit.

18: *Searching Files.* Obtain a byte value (0-255) and a file name. Display the number of times that byte appears in the file.

19: *Generating Files.* Create a sister program to the previous problem. Create a binary data file with random bytes, but one particular byte will appear in that file a set number of times. Obtain the following three values: (1) a byte value (0–255), (2) the number of times that byte should appear in the data file, and (3) the total number of bytes that make up the data file. Your job is to create that file, randomly scatter the request byte across the file, and to ensure that there are no duplicates and that the file contains exactly the number of occurrences that byte was requested for, and that the resulting data file is exactly the size requested.

8. Errors and Exceptions

8.1 What are errors and exceptions

Errors and exceptions are part of a programmers life, before we get into the details, lets look at what errors and exceptions are.

Errors

When it comes to software development errors could either be syntactical or logical.

Syntactical errors occur when there is an issue with the programs construct in this case the compiler cant compile the code and the interpreter cant execute the statement with the syntax error

Once the program is syntactically correct the only errors that remain in the program are logical errors. The causes of these are

- Lack of input to the program
- Invalid input
- Not being ale to calculate the desired result based on the input

Logical errors are sometimes also referred to as domain or range failures. When an error is detected by the python interpreter it indicates that it reached a point in the program flow where execution is no longer possible.This is the point where exceptions come in.

Exceptions

Exceptions in python are actions that are taken outside of the normal control flow. They occur in two phases. The first being that an error causes an exception and the second phase being the detection of the exception and the possible resolution of it.

The first phase takes place when an exceptional condition occurs. When this happens the interpreter detects the error and recognizes the exceptional condition. This I when the interpreter "raises" an exception. The words triggers", "throws" or "generates" can be used and it is the way that the interpreter makes it know that an exceptional condition has occurred. Python also lets the programmers raise exception and how ever this exception was raised either by the programmer or the interpreter it sends a signal that there was an exceptional condition and takes the program of the normal control flow this is the second phase of an exception where the programmer can specify how to handle the error.

The second phase is the exception handling part. When an exception is raised the exception can be handled in several ways like ignoring the error, simply logging the error, performing corrective measures, etc. Each of these methods of handling the error takes a different path in the programs control flow. Here the advantage of error handling is that the programmer can dictate how an error should be handled.

Exceptions are mostly caused by unforeseen possibilities that happen(mostly through user input). Python a language that supports raising and handling exceptions help the programmer by giving them a way to handle these unforeseen possibilities. This increases a programs robustness as the logical errors are handled at runtime.

8.2 Exceptions in python

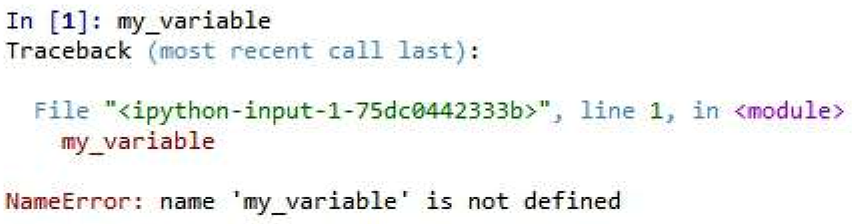
you might have already encountered some exceptions as you were trying out the examples and exercises in this book. Recall that a “traceback” notice was printed to the screen along with as much information that the interpreter can provide. This notice includes

- Name of the error
- Reason for the error
- Could also include the exact or near line number

All errors regardless of whether they are syntactical or logical are caused by behavioral incompatibility with the python interpreter and they cause errors to be raised. Let’s take a look at some of pythons exceptions below.

NameError : Trying to access a variable that is not declared

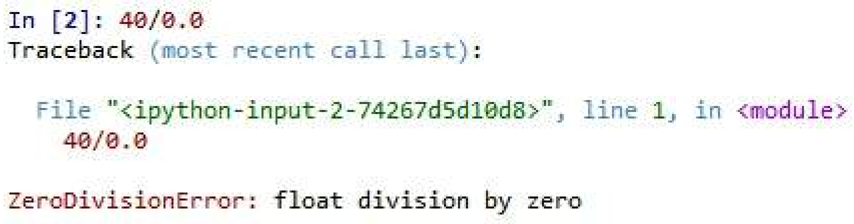
This exception is raised when there was a access to a variable that wasn’t initialized yet. The accessed identifier was not found in the symbol table. Take a look at the example below.



```
In [1]: my_variable
Traceback (most recent call last):

  File "<ipython-input-1-75dc0442333b>", line 1, in <module>
    my_variable

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

ZeroDivisionError : Division operation with a numeric zero

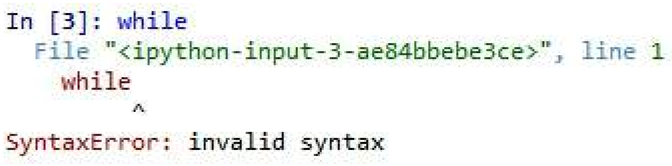
Any numeric division by zero raises this error. An example is given below.



```
In [2]: 40/0.0
Traceback (most recent call last):

  File "<ipython-input-2-74267d5d10d8>", line 1, in <module>
    40/0.0

ZeroDivisionError: float division by zero
```

SyntaxError : A syntax error

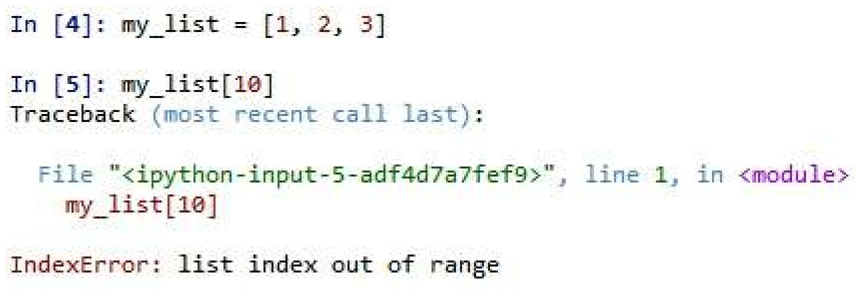
These are the only type of errors that occur at run-time.The indicate that there is some part of the program that does not adhere to pythons syntax definition. An example is given below.



```
In [3]: while
  File "<ipython-input-3-ae84bbebe3ce>", line 1
    while
         ^
SyntaxError: invalid syntax
```

IndexError : Requesting an index that is out of range on a sequence object

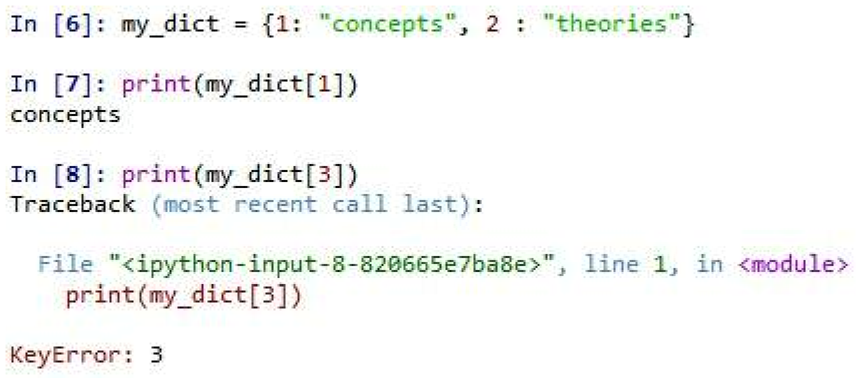
Raised when an index that is outside the available range is requested for a sequence typed object. An example is given below.



```
In [4]: my_list = [1, 2, 3]

In [5]: my_list[10]
Traceback (most recent call last):

  File "<ipython-input-5-adf4d7a7fef9>", line 1, in <module>
    my_list[10]

IndexError: list index out of range
```

KeyError : Requesting a dictionary key that does not exist

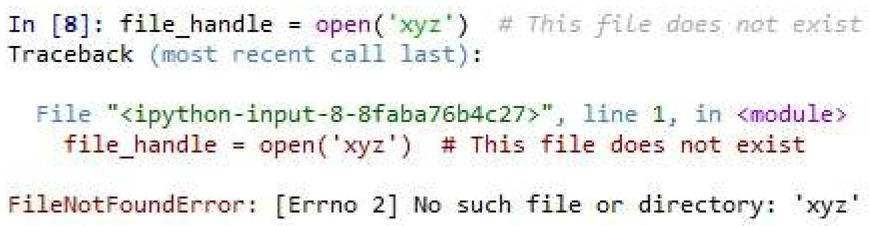
Keys are required when accessing data from mapping types like dictionaries, and the values from the mapping type can not be retrieved if the key is incorrect or nonexistent. In such cases a KeyError is raised. An example is given below.



```
In [6]: my_dict = {1: "concepts", 2 : "theories"}

In [7]: print(my_dict[1])
concepts

In [8]: print(my_dict[3])
Traceback (most recent call last):

  File "<ipython-input-8-820665e7ba8e>", line 1, in <module>
    print(my_dict[3])

KeyError: 3
```

IOError : An input or output error

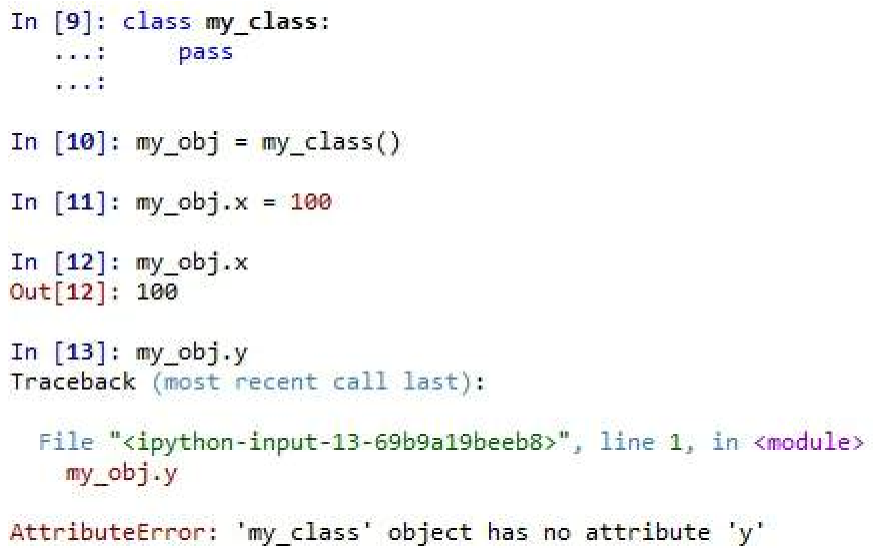
An IOError exception is raised on any kind of input or output error like trying to access a file that does not exist . An example is given below.



```
In [8]: file_handle = open('xyz')  # This file does not exist
Traceback (most recent call last):

  File "<ipython-input-8-8faba76b4c27>", line 1, in <module>
    file_handle = open('xyz')  # This file does not exist

FileNotFoundError: [Errno 2] No such file or directory: 'xyz'
```

AttributeError : Attempting to access an unknown attribute of an object

Observe that in the example below when an uninitialized instance variable is accessed the python interpreter raises an AttributeError exception .



```
In [9]: class my_class:
   ...:     pass
   ...:

In [10]: my_obj = my_class()

In [11]: my_obj.x = 100

In [12]: my_obj.x
Out[12]: 100

In [13]: my_obj.y
Traceback (most recent call last):

  File "<ipython-input-13-69b9a19beeb8>", line 1, in <module>
    my_obj.y

AttributeError: 'my_class' object has no attribute 'y'
```

8.3 Detecting and handling exceptions

Exceptions in python are detected using the “try” statement. The body(code suite) of the try statement is always monitored for exceptions. The try statement comes in two forms and they are,

1. Try-except
2. Try-finally

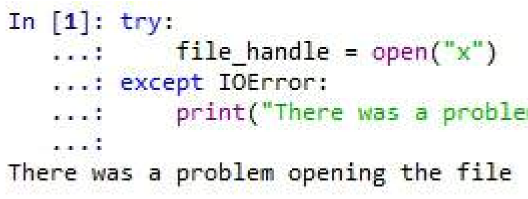
These two types of try statements are mutually exclusive which means python doesnt allow using them as a hybrid. The try-except statement detects and also handles an exception the try is responsible for detecting an exception and the except part of the statement is responsible for handling the exception. This statement can have one or more except clauses and can also include an else clause which is the case that no error occurred. The try-finally statement has one try clause that covers the code that requires monitoring of exceptions and and exactly one finally clause which does not allow handling the exception but can only perform some form of obligatory cleanup.

The try-except statement

The try-except statement syntax allows the programmer to define a section of code that needs to be monitored for exceptions and also a section of code that handles the cases when exceptions occur. The syntax of the try-except statement is given below.

```
try :
      <try suite> # Monitor for exceptions here
except  <exception>:
      <except suite> # Handle exceptions here
```

Lets take a look again at the example that we previously discussed by applying the try-except statement to it



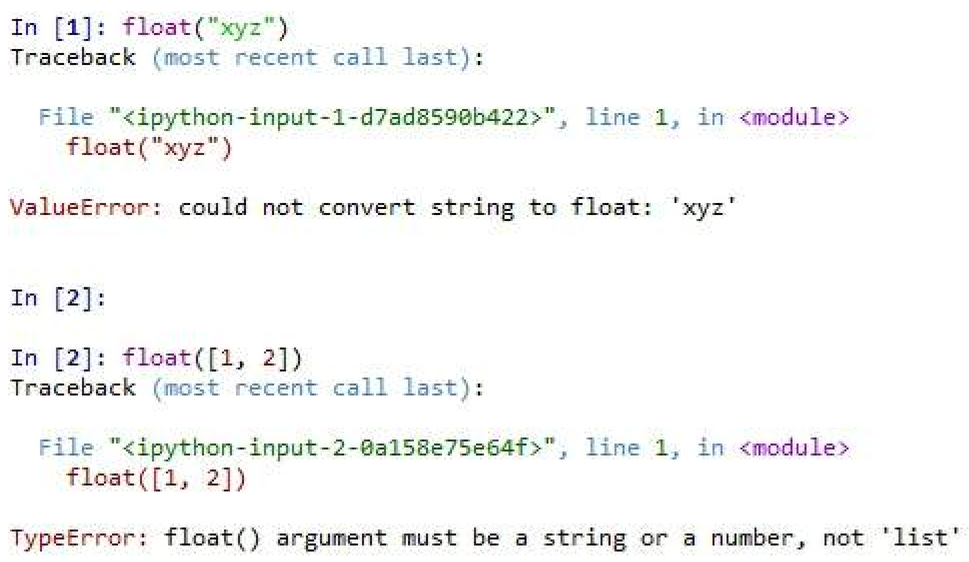
```
In [1]: try:
   ...:     file_handle = open("x")
   ...: except IOError:
   ...:     print("There was a problem opening the file")
   ...:
There was a problem opening the file
```

Observe that the error does not occur like in the last example. In actuality the python interpreter does raise an “IOError” exception at line 2 as the file x does not exist and there is an attempt to open the file, but as this code suite is monitored for exceptions, as soon as an exception is raised the exception is handled by transferring the flow of control to the code suite of the appropriate except clause.All the remaining code in the code suite of the try block is skipped, but in this case there was no code skipped as there were no more instructions in the code suite of the try clause.

Once the error is raised the search for the appropriate handler begins and if an appropriate handler(except clause) is found the flow of control is passed to the body of the handler. If there wasn’t an appropriate handler found in the search then the exception is passed to the caller’s level to handle the exception this means that the exception is passed to the stack frame that is immediately preceeding the current frame. If again no appropriate handler was found then the exception is again passed to its caller. In this way is the top most level is reached then the python interpreter considers the exception to be unhandled and the python interpreter displays the usual traceback notice before exiting.

Wrapping a built in function using try-except

Lets go through an interactive example that starts simply with just detecting an error and continuously improve on what we have to make it more robust. Our example involves the float( ) function which can convert any numeric type to a floating point value. The float( ) function can also convert a number that is in it’s string representation(like the string “100”) to it’s float equivalent. The float( ) does not accept bad input like in the below examples below.



```
In [1]: float("xyz")
Traceback (most recent call last):

  File "<ipython-input-1-d7ad8590b422>", line 1, in <module>
    float("xyz")

ValueError: could not convert string to float: 'xyz'


In [2]:

In [2]: float([1, 2])
Traceback (most recent call last):

  File "<ipython-input-2-0a158e75e64f>", line 1, in <module>
    float([1, 2])

TypeError: float() argument must be a string or a number, not 'list'
```

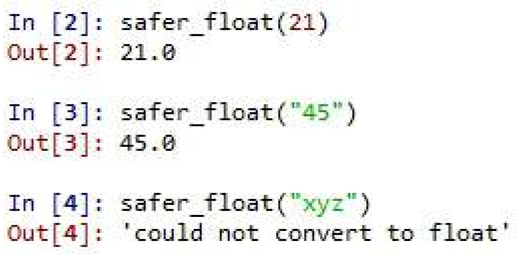
Note that in the first instruction the argument is a string which is an acceptable type but the string was not a valid number so the ValueError exception was raised. In the second instruction the argument was not of the right type so the TypeError was raised. The exercise that we’re going to go through involves making the call to float( ) function more “safe”, which means that we want to ignore the cases that turn up error as they are not relevant to our task of converting strings into floating point number and the cases are not severe enough that the python interpreter has to abandon the execution. To accomplish this objective we will use a wrapper function with the help of the try-except statement and we’ll call this function “safer_float( )”. In our first iteration we’ll handle the ValueError exception as this is one of the common exceptions that the float function results in.

```python
def safer_float(object):
    try:
        return float(object)  # return the float object
    except ValueError:
        pass                  # do nothing
```

Here in our first iteration we solved the problem of making the float function safer as now with the safer_float( ) function the program will no longer abruptly halt. Here there is a ValueError exception raised the exception is detected but as there is nothing done in the except clause there is no handling done, which means that we are just ignoring the error. One problem here is that in the case of an error there is no value returned back to the caller of the function that means a "None" is returned to the caller. This prevents the caller from knowing if there was anything wrong during the executing the function. Let's change the above code so we can indicate to the caller that there was some error in the case that there were.

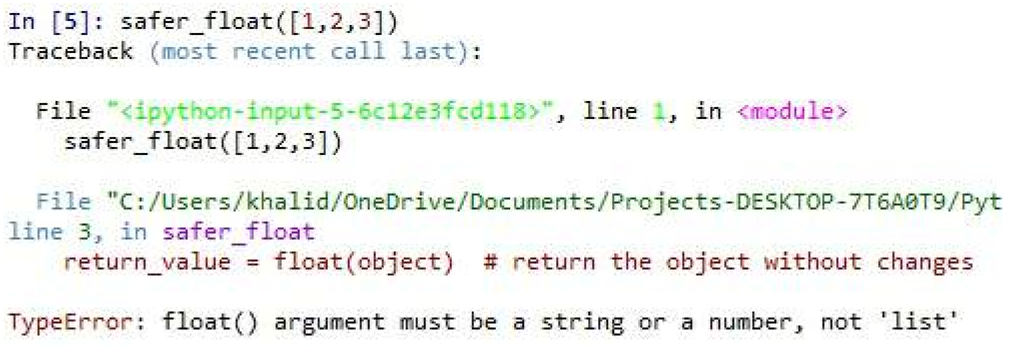
```python
def safer_float(object):
    try:
        return_value = float(object)
    except ValueError:
        return_value = "could not convert to float"
    return return_value
```

Lets try calling the function we just designed with different arguments.



```
In [2]: safer_float(21)
Out[2]: 21.0

In [3]: safer_float("45")
Out[3]: 45.0

In [4]: safer_float("xyz")
Out[4]: 'could not convert to float'
```

A problem with the function that just designed is that it is able to handle the ValueError exceptions if raised but it can't handle TypeError exceptions that would be raised if an object other than a string or numeric type is passed.



```
In [5]: safer_float([1,2,3])
Traceback (most recent call last):

  File "<ipython-input-5-6c12e3fcd118>", line 1, in <module>
    safer_float([1,2,3])

  File "C:/Users/khalid/OneDrive/Documents/Projects-DESKTOP-7T6A0T9/Pyt
line 3, in safer_float
    return_value = float(object)  # return the object without changes

TypeError: float() argument must be a string or a number, not 'list'
```

Lets discuss how to overcome this problem.

try-except statements with multiple except clauses

We've already introduced try-except statements of the following kind with only one except clause, this construct is mostly used to handle exceptions of type "Exception".

```
try :

      <try suite> # Monitor for exceptions here

except  <exception>:

      <except suite> # Handle exceptions here
```

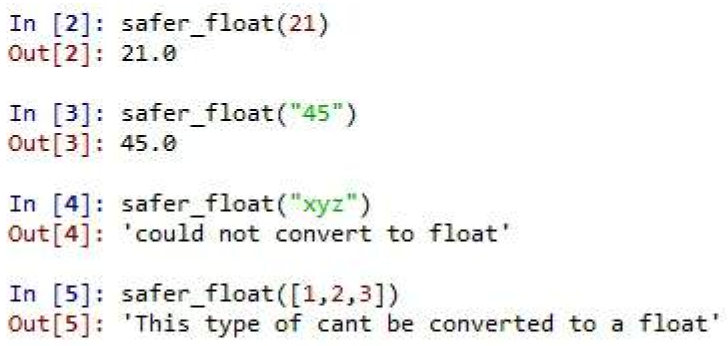
Try-except statements also allow chaining together multiple except clauses together to handle multiple different types of exceptions and handling the exceptions differently based on what kind of exception was raised. The syntax for try-except statements with multiple except statements is given below.

```
try :

      <try suite> # Monitor for exceptions here

except  <exception1>:

      <except suite> # Handle exception1 here

except  <exception2>:

      <except suite> # Handle exceptions2 here

      .

      .

      .
```

Let us apply what we have just learned to make the safer_float( ) function able to handle exceptions of type TypeError. Take a look at the updated code below.

```python
def safer_float(object):
    try:
        return_value = float(object)
    except ValueError:
        return_value = "could not convert to float"
    except TypeError:
        return_value = "This type of cant be converted to a float"
    return return_value
```

Lets try calling the function we just designed with different arguments.



```
In [2]: safer_float(21)
Out[2]: 21.0

In [3]: safer_float("45")
Out[3]: 45.0

In [4]: safer_float("xyz")
Out[4]: 'could not convert to float'

In [5]: safer_float([1,2,3])
Out[5]: 'This type of cant be converted to a float'
```

Observe that the function now handles the TypeError exception by returning a predefined string.

except statements with multiple exception types

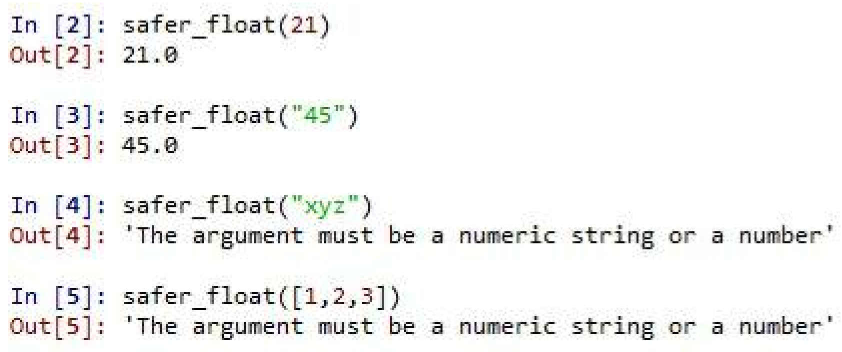
Except clause can be used to handle multiple exception types. These type of except clauses require the set of exception types to be handled be provided as a tuple. The syntax of such except clause is given below.

```
except  (<exception1>, <exception2>, …):
        <except suite> # Handle those exceptions here
```

The safer_float( ) function can be re-written as follows.

```
def safer_float(object):
    try:
        return_value = float(object)
    except (ValueError, TypeError) :
        return_value = "The argument must be a numeric string or a number"
    return return_value
```

Lets try calling the function we just designed with different arguments.



```
In [2]: safer_float(21)
Out[2]: 21.0

In [3]: safer_float("45")
Out[3]: 45.0

In [4]: safer_float("xyz")
Out[4]: 'The argument must be a numeric string or a number'

In [5]: safer_float([1,2,3])
Out[5]: 'The argument must be a numeric string or a number'
```

try-except statements with no named exceptions

This type of try-except statements is one that does not specify the name of the exception and it’s syntax is show below.

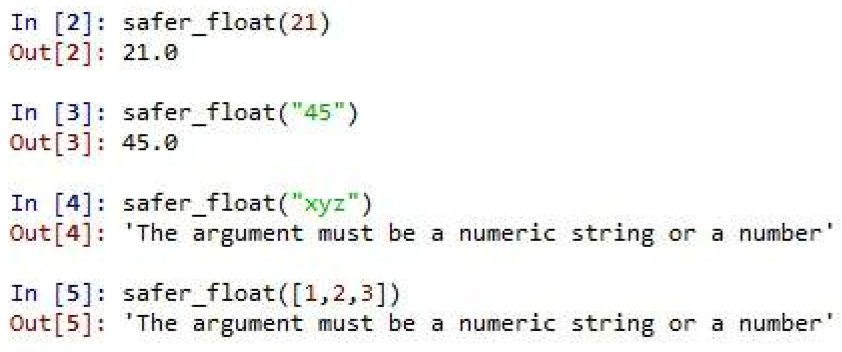
```
try :

      <try suite> # Monitor for exceptions here

except :

      <except suite> # Handle all exceptions here
```

Even though this approach to using the try-except statement would catch almost all exception it is highly recommended never to use this approach as this reveals nothing regarding the root cause of the occurring exceptions. It is better for the programmer to be have code the helps the programmer to investigate and discover what type of errors can occur to prevent them in the future.

Let’s look at our safer_float( ) function with this kind of except clause.

```python
def safer_float(object):
    try:
        return_value = float(object)
    except :
        return_value = "The argument must be a numeric string or a number"
    return return_value
```

Lets try calling the function we just designed with different arguments.



```
In [2]: safer_float(21)
Out[2]: 21.0

In [3]: safer_float("45")
Out[3]: 45.0

In [4]: safer_float("xyz")
Out[4]: 'The argument must be a numeric string or a number'

In [5]: safer_float([1,2,3])
Out[5]: 'The argument must be a numeric string or a number'
```

The else statement

This type of try-except statements is one that does not specify the name of the exception and it’s syntax is show below.

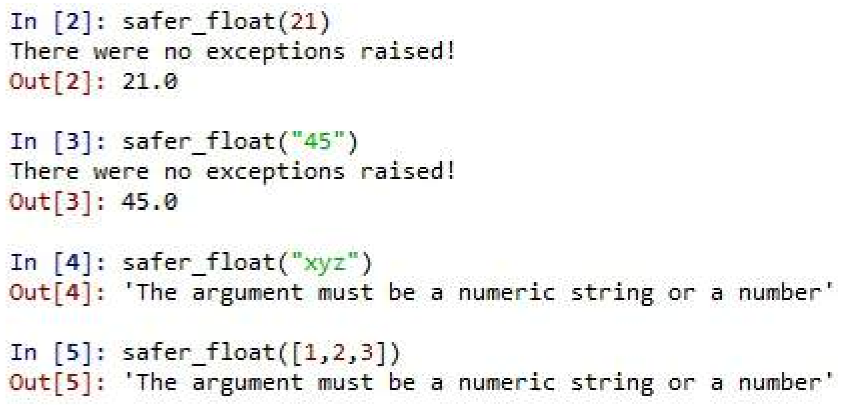
```
<try-except statement>
else:
   <block that gets executed if there is no error>
```

Even though this approach to using the try-except statement would catch almost all exception it is highly recommended never to use this approach as this reveals nothing regarding the root cause of the occurring exceptions. It is better for the programmer to be have code the helps the programmer to investigate and discover what type of errors can occur to prevent them in the future.

Let’s look at our safer_float( ) function with this kind of except clause.

```
def safer_float(object):
    try:
        return_value = float(object)
    except (ValueError, TypeError) :
        return_value = "The argument must be a numeric string or a number"
    else:
        print("There were no exceptions raised!")
    return return_value
```

Lets try calling the function we just designed with different arguments.



```
In [2]: safer_float(21)
There were no exceptions raised!
Out[2]: 21.0

In [3]: safer_float("45")
There were no exceptions raised!
Out[3]: 45.0

In [4]: safer_float("xyz")
Out[4]: 'The argument must be a numeric string or a number'

In [5]: safer_float([1,2,3])
Out[5]: 'The argument must be a numeric string or a number'
```

Exceptional arguments

When exceptions are raised additional arguments can be passed along with them to the exception handler. The purpose of these additional arguments is to provide more aid to the exception handler. Even though exception arguments are completely optional, standard built in exceptions provide at least one argument which is an error string that indicates what the cause of the exception is. To access the exception argument a variable is required to hold the argument, this argument is provided on the except header and follows the exception type that is being handled. The syntax for adding arguments to the except clause is given below.

```
# For a single exception

except <exception>, <argument>:

      <except suite> # Handle exceptions here

# For a multiple exceptions

except (<exception1>, <exception2>, …), <argument>:

      <except suite> # Handle exceptions here
```

Unless the exception that was raised was a string exception. The "argument" here is an instance of a class that contains diagnostic information from the code that caused the error. The exception arguments are in a tuple which is an attribute of the exception object that was instantiated. In the general syntax that is shown above, "argument" is an instance of the exception class.

When it comes to most of the standard built-in exceptions(meaning exception that inherit from StandardError class), the tuple has a string that is used to help specify the cause of the error

try-finally statements

These statements differ from try-except statements as they don't handle the raised exceptions. Their purpose is to ensure consistent behaviour regardless of the whether there was an exception raised. the "finally" suite executes regardless of whether there was an error raised in the "try" suite. The syntax of the try-finally statement is shown below.

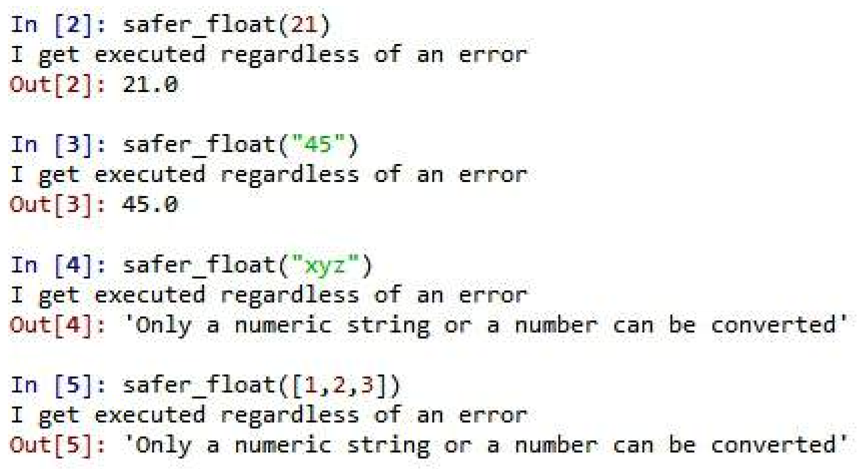
```
try :

       <try suite> # Monitor for exceptions here

finally  <exception>:

       <finally suite> # Execute regardless of exception
```

In the case that there was an exception that was raised in the try suite the control flow is immediately passed to the finally suite. When the finally suite has completed execution the exception is raised again to be handled ny the higher level. This is the reason that it is very common to see try-finally statements embedded within try-except statements.

Let's look at our safer_float( ) function with the try-finally statement.

```
def safer_float(object):
    return_value = 0
    try:
        try:
            return_value = float(object)
        finally :
            print("I get executed regardless of an error")
    except (ValueError, TypeError):
        return_value = "Only a numeric string or a number can be converted"
    return return_value
```

Lets try calling the function we just designed with different arguments.



```
In [2]: safer_float(21)
I get executed regardless of an error
Out[2]: 21.0

In [3]: safer_float("45")
I get executed regardless of an error
Out[3]: 45.0

In [4]: safer_float("xyz")
I get executed regardless of an error
Out[4]: 'Only a numeric string or a number can be converted'

In [5]: safer_float([1,2,3])
I get executed regardless of an error
Out[5]: 'Only a numeric string or a number can be converted'
```

List of standard exception in python

The table below lists and describes pythons standard exceptions.

Exception Name	Description
`Exception`	Base class for all exception classes
`StopIteration`	Exception raised when an iterators next( ) method doesnt point to an object
`SystemExit`	Sys.exit( ) function method raises this exception
`StandardError`	This is the base class for all of the built in exceptions excluding excluding StopIteration and SystemExit.
`ArithmeticError`	This is the base class for all errors that occur during a numeric calculation
`OverflowError`	When a calculation exceeds maximum limit that a numeric type can handle this exception is raised
`FloatingPointError`	Raised if a floating point calculation fails
`ZeroDivisionError`	Raised if a division by zero or modulo by zero takes place
`AssertionError`	Raised if an assert statement fails
`AttributeError`	Raised when accessing an attribute of an object fails
`EOFError`	Raised in the case where there is no input from the input( ) function and the end of the file is reached

`ImportError`	Raised on failure to import a module
`KeyboardInterrupt`	Raised if user interrupts the programs execution. This us usually done by pressing ctrl + c
`LookupError`	This is the base class for all look up errors
`IndexError`	Raised when an index could not be found when indexing a sequence typed object
`KeyError`	Raised when the indicated key was not found when search a dictionary type object
`NameError`	Raised when a name could not be found in the local or global namespace
`UnboundLocalError`	Raised when trying to access a local variable of a function that does not have a value assigned to it yet
`EnvironmentError`	This is the base class for all of the exceptions that occur outside the python environment
`IOError`	Raised when an I/O operation fails like when the open function can not find the specified file
`SyntaxError`	Raised when the program is inconsistent with python's grammatical syntax
`Indentation error`	Raised when there are problems with the indentation of some instructions
`SystemError`	Raised when the python interpreter finds an internal error. The interpreter does not exit when this error is raised
`TypeError`	Raised if an function or an operation is carried out for data type that it is not designed to handle
`ValueError`	Raised when a built in function for a data type has the valid type of arguments but the arguments have invalid values provided.
`RuntimeError`	Raised if the error that was generated does not fall into any category.
`NotImplementedError`	Raised if the abstract method that was required to be implemented in an inherited class wasn't implemented

8.4 Raising exceptions

In whatever we have discussed so the python interpreter was responsible for raising the exceptions. These exceptions are raised when there is an error during execution. When a programmer is writing an API there may be some cases where the programmer wants to raise an exception on an erroneous input. Python allows a programmer to explicitly raise exceptions using the raise statement.

The raise statement is very flexible when it comes to the arguments that it supports. This makes it allow a large number of different formats. The raise statements general syntax is given below.

```
raise [Exception [, args[,traceback]]]
```

Here the first argument which is “Exception” is the name of the exception that is to be raised. If it is present then it should either be a string, object or a class. The “Exception” argument is required if the the other arguments are present. The second expression is the args(which are parameter and values) for the exception is optional and is either a single object or a tuple of objects.In the case that an exception is detected the exception argument is always returned as a tuple. In the case that “args” is a single object then the tuple consists of just that one object and if “args” is tuple of objects then the tuple is is the tuple of arguments that are passed to the handler. The majority of cases is that single objects passed will have a string that indicates what the cause of the error was. When a tuple is passed the usual case is an error string, error number and an error location(which file).

The last argument here which is “traceback” is an optional argument that is rarely used and is the traceback object that is used for the exception. This argument is useful if the programmer wants to re-raise an exception. Arguments can be left absent by using the value ”None”.

Lets take a look an example of using the raise statement in a simple python program given below.

```
def safer_float(object):
    return_value = 0
    try:
        if ((type(object) != str and type(object) != int)) :
            ex = Invalid_type("Incorrect type", "cant convert an invalid Type")
```

```python
            raise ex
        return_value = float(object)  # not executed if exception raised
    except Invalid_type:
        return_value = ex.error
    except ValueError:
        return_value = "your string is not a valid number"
    else:
        print("No errors were raised")
    return return_value


class Invalid_type(Exception):
    def __init__(self, message, errors):

        # Call  base class constructor with  parameters it requires
        super().__init__(message)

        # custom code...
        self.error = errors

```

Lets try testing our code using different arguments below.

```
In [2]: safer_float(21)
No errors were raised
Out[2]: 21.0

In [3]: safer_float("45")
No errors were raised
Out[3]: 45.0

In [4]: safer_float("xyz")
Out[4]: 'your string is not a valid number'

In [5]: safer_float([1,2,3])
Out[5]: 'cant convert an invalid Type'
```

8.5 Assertions

Assertions in python re predicates that are required to evaluate to the boolean value "true" or 1 otherwise and exception is raised to indicate that an assertion was evaluated to be "false" or 0. they are used for diagnostic and testing purposes. Assertions in python are similar to the "assert" macros in the c language.

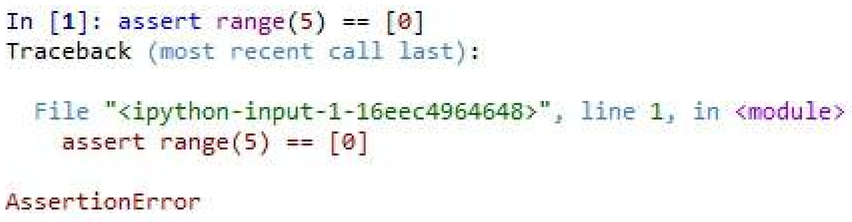
The easiest way to understand assertions in python is to understand them as an imaginary “raise-if-not” statement which means an expression is evaluated and if the result is false then an exception is raised. The exception raised if an assertion is evaluated to be false is the `AssertionError`. Assertions are used in python with the assert statement, which was introduced in python 1.5 .

assert statement

Lets take a look at a few examples of the assert statement below.

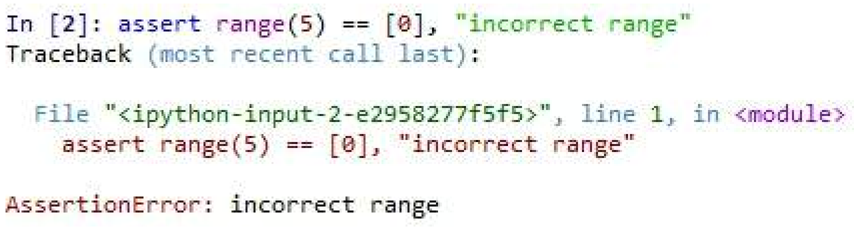
```python
#=======================================================
# All these assertions evaluate to true
#=======================================================
assert 1 != 0
assert range(5) == [0, 1, 2, 4, 5]
assert 1
assert len(['this string', 2]) < 15
#=======================================================
# All these assertions raise an AssertionError exception
#=======================================================
assert 1 != 1
assert range(5) == [0]
assert 0
assert len(['this string', 2]) > 15
```

AssertionError that is raised can be handled just as other exceptions can be handled for example by using the try-except statement, but when they are not handled the interpreter raises a traceback notice and terminates the program as shown in the example below.



```
In [1]: assert range(5) == [0]
Traceback (most recent call last):

  File "<ipython-input-1-16eec4964648>", line 1, in <module>
    assert range(5) == [0]

AssertionError
```

Just as we provided an argument to our raise statement in the previous section we can provide a argument to our assertion as shown below.



```
In [2]: assert range(5) == [0], "incorrect range"
Traceback (most recent call last):

  File "<ipython-input-2-e2958277f5f5>", line 1, in <module>
    assert range(5) == [0], "incorrect range"

AssertionError: incorrect range
```

Now lets use a try-except statement to catch the AssertionException in the code below.

```python
try:
    assert (2 + 2) == 5
except AssertionError:
    print("2 + 2 is not 5")

```

In the above program the assertion is evaluated to 0 therefore the interpreter raises the AssertionError which is caught and handled in the except suite.

For a better understanding of how the assertion statement works lets design a function that performs the same function as the assert statement.

```python
def function_assert(expr, args=None):
    if __debug__ and not expr:
        raise AssertionError
```

The first if statement here makes sure that the syntax of the assert statement is preserved(expr must be an expression) The second part here evaluates the expression and raises the exception if the result of the expression was a zero. Under normal circumstances the built in variable __debug__ is 1 and 0 in the case that optimization is requested.

8.6 Exercises

1: Raising Exceptions. Which of the following can RAISE exceptions during program execution? Note that this question does not ask what may CAUSE exceptions.

a) the user

b) the interpreter

c) the program

d) all of the above

e) only (b) and (c)

f) only (a) and (c) \

2: Raising Exceptions. Referring to the list in the problem above, which could raise exceptions while running within the interactive interpreter?

3: Keywords. Name the keyword(s) which is(are) used to raise exceptions.

4: Keywords. What is the difference between **try-except** and **try-finally?**

5: Exceptions. Name the exception that would result from executing the following pieces of Python code from within the interactive interpreter (refer back to Table 10.2 for a list of all built-in exceptions):

a)

```
>>> if 3 < 4 then: print '3 IS less than 4!'
```

b)

```
>>> aList = ['Hello', 'World!', 'Anyone', 'Home?']
>>> print 'the last string in aList is:', aList[len(aList)]
```

c)

```
>>> x
>>> x = 4 % 0
```

e)

```
>>> import math
>>> i = math.sqrt(-1)
```

6: Improving open(). Create a wrapper for the open() function. When a program opens a file successfully, a file handle will be returned. If the file open fails, rather than generating an error, return None to the callers so that they can open files without an exception handler.

7: Exceptions. What is the difference between Python pseudo-code snippets (a) and (b)? Answer in the context of statements A and B, which are part of both pieces of code.

a)

```
try:
statement_A
except …:
…
else:
statement_B
```

b)

```
try:
statement_A
statement_B
except …:
…
```

8: Improving raw_input(). In the beginning of this chapter, we presented a "safe" version of the float() built-in function to detect and handle two different types of exceptions which float() generates. Likewise, the raw_input() function can generate two different exceptions, either EOFError or KeyboardInterrupt on end-of-file (EOF) or cancelled input, respectively. Create a wrapper function, perhaps safe_input(); rather than raising an exception if the user entered EOF (^D in Unix or ^Z in DOS) or attempted to break out using ^C, have your function return None that the calling function can check for.

9: Improving math.sqrt(). The math module contains many functions and some constants for performing various mathematics-related operations. Unfortunately, this module does not recognize or operate on complex numbers, which is the reason why the cmath module was developed. Rather than suffering the overhead of importing an entire module for complex numbers which you do not plan on using in your application, you want to just use the standard arithmetic operators which work fine with complex numbers, but really want just a square root function that can provide a complex number result when given a negative argument. Create a function, perhaps safe_sqrt(), which wraps math.sqrt(), but is smart enough to handle a negative parameter and return a complex number with the correct value back to the caller

```
self.file =
self.fingerpri
self.logdupes =
self.debug = de
self.logger =
if path:
    self.file
    self.file.
    self.fing

@classmethod
def from_setting
    debug = sett
    return cls(j

def request_see
    fp = self.
```

[illegible]