Learning The Python Language

Syed Asad Ali Zaidi

1 The Keywords

In computer language, understanding and applying these keywords is probably the most important aspect. When keywords are used in a code , the computer would act in that manner.Perhaps thinking and using these keywords are the actual crux of programming. Truly knowing and fully understand how these keywords would work makes the programming lot easier. Otherwise by not fully understanding these keywords and rushing and jumping to learn faster would later raise confusion and chaos.In learning any computer language it is lot better to first fully focus on these keywords.Why they act ? What are their types? & How they are used? In Which way they are applied? and after applying what exactly would their output be?

For instance 'while' is used in python as a loop, it follows the command given. Similarly when 'if' is used logical condition is used, when 'def' is used the computer would perform specific custom function.

Keywords are words by which computer understands that particular language. They are used in specific order of commands, and are used in different programming languages accordingly. In python one can type help('keywords') in the python terminal and the list appears. Keywords in alphabetical order are in the following list.

1.1 and , as , assert
break
class , continue
def , del
elif , else , except , exec
finally, for, from
global
if, import, in, is
lambda
not
or
pass, print
raise, return
try
while, with
yield.

2 The Statements

Instruction given to python are called statements.

2.1 Example:-

$$x = y + 1$$

Here x is variable '=' is an operator, 'y+1' is an expression, '1' is a constant

$$x = "one"$$

Here "one" is a string.

$$print(x)$$

Here print is a predefined function.

3 Variables

Type of variables in python.

4 Conditions

Those statements which have results logically true or false. They uses either if or else, and are called conditions.But they actually are three if, elif and else. If can be used twice in a statement, and else is used in the end, it act as a true when if is false and vice versa. else is a kind of closing argumnet. elif can be used in between if and else as many times until the program finds it as true. Otherwise the program would end on else and the condition would be executed. if , elif, and else also uses : (indentation), which is very important in Python. Without indentation Traceback error would appear.

4.1 Example:-

```
x = 42
if x > 1 :
        print('More than one')
        if x < 100 :
                print('Less than hundred')
print('All Done')
```

4.2 Example:-

```
x = 7
if x < 2 :
        print('small')
elif x < 10 :
        print('medium')
else :
    print ('large')
print('All done')
```

4.3 Example:-

```
a = 33
b = 33
if b > a :
        print("b is greater than a")
elif a == b :
        print("a and b are equal")
```

```
a = 200
b = 33
if b > a :
        print("b is greater than a")
elif a == b :
        print("a and b are equal")
else :
        print("a is greater than b")
```

Indentation is the space of four or the tab which starts after conditions or loops or exception etc. When give : after any condition or loop or exception etc text editor like ATOM automatically indent. Python would give a Trace back error if indentation is not followed.

5 Loops & Iteration

The purpose of loops is give a command to the computer to repeat.This can be a anything from numeric digits to list of strings. There are three kinds of loops in Python, The while loop, the for loop and the nest loop.The moment we use the keyword 'while' or 'for', python would start the process of repeating or Iteration. The statement used in the loop has indentation and the variable constructed to control the loop is called Iteration variable.

5.1 while loops-the indefinite loop

First we shall discuss the while loop. While loop act in true or false. Incremental or decremental value can be used to for true and false of the statement, bad loop programing would act toward infinity and looping will not stop.Usually in while loop a value is given to the variable in the begining. Then when the while key word is used that variable is constructed with an operator lie $<$ or $>$ or $==$ to make a logical statement of the while keyword which will act as loop. Indentation : is used and inside that indented block statements like if (with another indentation :) or functions like print() can be used.The calculation should be in a way that false should be achieved in the end to avoid unstopable loop.Few of the simple examples of while loop are as follows.

5.1.1 Example:-

```
x = 5
while x  > 0 :
        print(x)
        x = x - 3
```

5.1.2 Example:-

$$i = 1$$

while i < 6 :

print(i)

$$i = i + 1$$

5.1.3 Example:-

$$n = 5$$

while n > 0 :

print(n)

$$n = n - 1$$

print('Blastoff!!!')

There are couple of ways to stop the loop. One is the break keyword. Here is a complex example to logically use the loop for an input and stoping it by the keyword break. In this example line is given the value done in a way that it shall finish and exit the code.

5.1.4 Example:- Using keyword break to stop loop

```
while True:
    line = input ('>')
    if line == 'done' :
        break
    print(line)
print('Done!')
```

5.1.5 Result

```
>Hello there
Hello there
>How r u?
How r u?
>finish it !!!
finish it !!!
>done
Done!

Process finished with exit code 0
```

If we carefully analyze the Example 4.1.4, the True keyword is used here as an expression of the the while loop, to start the iteration process. As we know that the whole iterational process of while loop is based on true and false. As the keyword while is used, the indentation in the indented block would make any statement act as an iteration accordingly.And as we write below line after indentation: with an input()fuction, the program enter's in > prompt.

line = input ('>')

In the obove line code by assigining line with the value of an input function('>'). This would give a display > as a prompt.

In this code, the given value of an input with the string is '>'. Using inverted comas turn this sign '>' into a string, and input() function has converted it to take input values, like the example below.

>hello there

Hello there

We know that < & > sign are operators but inside the inverted comas '>' this operator become a string . This '>' sign after it has been assigned with inverted comas now act as a class string , here this '>' does not hold anymore of the mathematical value and becomes or act as a prompt in the output.

In this example the iteration is happening in the back end as a type of > prompt.

This > prompt is actually the iteration!

Exiting this > prompt is only possible by writing the word 'done' which is the execution of the break keyword.

The interesting thing here is conditionality of the word 'done' given by the if statement. Done is the value of the line, which was previously given the value true (of the input), and now this value is changed, and is recorded as a false to stop the loop or pullout from the > prompt.

In simple words it has been assigned false value for the while loop.

Think closely , in this line inside the indentation block with the if condition's purpose is to end this loop. In this line the given value is 'done', so the program remember this particular word and will execute for the keyword break.

In the result we see that any line written like Hello there , How r u?, and finish it, are repeated as input statements. This is because of the input() function , the iteration is acting as a shell, which take these input value in the > prompt.

The if condition is working in two ways. Firstly reserving a word and secondly using this particular word to stop the loop.

So when we write the word 'done' the loop finished with exit code 0.

Otherwise one can not exit this prompt. This is also one kind of running loop of true in the back end and stoping it to false by the keyword break. Here we can pull out only from > prompt by stopping the loop using this break command. Below is the code line.

if line == 'done':

break

and the meaning of this line is break the process or end the loop using the word done.

This would false the while iteration which begins on true and end's on false, and brings out from the > prompt.

One thing to note here is that the iteration process in this particular example is different in process, than the other while examples given before.

Lastly this code is not completed until print() command is given for the line in the indentation of if statement. This print() is responsible for the output of any word that has to be repeated back in the run, Outside this indentation again print is given for the word done to print Done! and end the process.

print(line)

print('Done!')

Another keyword that can be used in the while loop is continue, unlike break keyword , which stops the loop , continue stop the iteration and bring back on top of the loop.

5.1.6 Example of Using keyword break to stop iteration

```
while True:
    line = input('>')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```

5.1.7 Result

```
>#
>print
print
>done
Done!
```

It is noted here that the '#' value which is made a string just like '>' and is assigned the keyword continue inside its indentation block , when written in the run does not act in any way. Neither it print back like the word print nor it exit out like the word done. This is the difference between continue and break.

5.2 for loops-the definite loop

for loops are also used for repetition but instead of the while loop it is more finite and easy to use.This is because it act on sets. These sets can be numbers, or words or any given data unlike the while loop which works on true or false. for loop take more like a ready made list in a set then define it. With for loop usually the keyword 'in' is used. A very simple example of this is

5.2.1 Example

```
for i in [5,4,3,2,1] :
    print(i)
print('blastoff!!!')
```

Here i is the variable, and which is assigned the value of the set which contain these numbers from 5 to 1. Every time as the for loop would iterate these values would change and by giving the command to print i, i.e the variable, the list would be printed.

5.2.2 Result

```
5
4
3
2
1
blastoff!!!
```

comparing to the while loop in Example 4.1.3 this Example 4.2.1 is written in three line code and is easily handeled because of the finite number of values available in a set. Another example with words or strings

5.2.3 Example

```
friends = ['asad','safeer','sanaullah']
for friend in friends :
    print('Eid Mubarak',friend)
print('Done!!')
```

A bit confusing but for better understanding , here the variable is friend. Python does not understand singular or plural but in the for loop the variable is going to iterate, hence printing the string and the variable.

5.2.4 Result

```
Eid Mubarak asad
Eid Mubarak safeer
Eid Mubarak sanaullah
Done!!
```

5.2.5 Example

The coming few examples are inter-related with each other in order to understand how looping can be used for calculations or pulling out certain data from a list. In this example print command is given to print the string 'Before', then thing the for loop variable is assigned the

values in the list. By giving the print command to the variable the list is printed .Finally ending with printing the string again, but this time the after.

```
print('Before')
for thing in [9,41,12,3,74,15] :
    print(thing)
print('After')
```

5.2.6 Result

```
Before
9
41
12
3
74
15
After
```

Keeping in mind the previous example ,If we wish to pull out the largest number from the same list then. Here is a more complex example.

5.2.7 Example The complex one, would be repeated in the end

```
largest_so_far = -1
print('Before',largest_so_far)
for the_num in [9,41,12,3,74,15] :
    if the_num > largest_so_far :
        largest_so_far = the_num
    print(largest_so_far,the_num)

print('After',largest_so_far)
```

In this Example a kind of negative value is given to the variable largest_so_far i.e -1 then string is printed that is Before, but after this statement the for loop is used now with a new iteration variable the_num which is greater than -1, here we are using the if condition which has a logical value of true or false .By declaring the value of the_num variable greater than largest_so_far which of course is True, the first value printed is 9 which is true. The iteration of that for's variable is greater and this iteration will work on the whole list.

5.2.8 Result

```
Before -1
9 9
41 41
41 12
41 3
74 74
74 15
After 74
```

Then inside the indentation block of if largest_so_far variable is equalise to the iteration variable the_num. What will happen from now on is that the program would seek true or false for the whole list. When the statement if the_num > largest_so_far: is false it shall write the previous greater number.

This list of largest_so_far is printed on the true and false logic.

The dual list is be printed by giving print comand (outside the indentation block of if and inside the indentation block of for), because both variables are separated by coma and are written in the first print command.

What we see in the result is that, largest_so_far is printed parallel to the_num, although with greater value in sequence, and Is this because of the if statement ?

Why?.

And the answer is that this code is going back & forth making equals and finding greater.

After assigning the equal it went back for the greater again and stuck into the greater.

Then the next command is to print the list of both ,think what can be largest_so_far now , its no longer smaller than the_num and is printed in the list as 9,41 but not 12. Here the 41 is repeated instead of 12, and this is because 41 is greater than 12 which is true here. It would repeat 41 again because its greater.

if the_num > largest_so_far:

iteration is working as 9 does is greater than -1 so it is written in the list of largest_so_far. then comes the 41 , iteration is working again,and 41 is greater than 9 which is true next comes 12 and 12 is not greater than 41 so its not true and the largest_so_far is 41, and 41 is written again as iteration would go to last number to complete the list and also would print, so it print 41 again.

largest_so_far = the_num

Now this line is inside the indentation of if statement so it shall follow logical condition of equalising the values of largest_so_far = the_num. Note two operation are going on one that if the_num > largest_so_far and then largest_so_far = the_num so the list becomes 9,41,41,41,74,74. The three numbers 12 which is not greater than 41, 3 again not greater than 41 and 15 not greater than 74 are skipped.

if the_num > largest_so_far:

It is still on the run, so yes 41 is greater than 12, iteration is still going on, next it skips the 12, writes 41 instead again and then skip 3 as it is smaller than 41 again and than find

74 which of course is greater than 41 and is now 74 written is greater in the largest_so_far list.It skips 15 and the last value remaining for largest_so_far is 74.

Lastly with the string 'After' the largest_so_far is written for print and 74 is printed with , After. Actually 74 which is the greatest value in the list of the_num is also now the value of largest_so_far, and we have achieved the pulling out of greatest number from a list of this "for loop" code.

Another simplified example with the same but easier notion is here.

This example is a continuity of the previous example.

See how the last value of the variable zork is 6 and not the initial value that was 0. How the formula zork+1 worked.

And ofcourse by not using the if statement this example is much simpler & easier to understand.

Example

```
zork = 0
print('Before',zork)
for thing in [9,41,12,3,74,15] :
    zork = zork + 1
    print(zork,thing)
print('After', zork)
```

Carefully study the result now

Result

```
Before 0
1 9
2 41
3 12
4 3
5 74
6 15
After 6
```

See the resemblance but its simpler & obviously without the if condition much easier to understand. Now observe this next example.

Example

```
zork = 0
print ('Before', zork)
for thing in [9,41,12,3,74,15] :
    zork = zork + thing
    print (zork,thing)
print('After',zork)
```

Here instead of +1 , + thing is used so the result is as follows

Result

```
Before 0
9 9
50 41
62 12
65 3
139 74
154 15
After 154
```

So things are being simplified in this much simpler code. Now carefully see the next example

5.2.9 Example

```
count = 0
sum = 0
print ('Before',count,sum)
for value in [9,41,12,3,74,15] :
    count = count + 1
    sum = sum + value
    print (count,sum,value)
print('After',count,sum,sum/count)
```

Here we have achieved the average also by simply dividing the variable sum from the variable count in the print function execution. Follow the result

5.2.10 Result

```
Before 0 0
1 9 9
2 50 41
3 62 12
4 65 3
5 139 74
6 154 15
After 6 154 25.666666666666668
```

All of these examples are the sequence of the complex example of the first in which we have used if statement in the for loop indentation. In these three example we have not used the if satement but this all is leading towardthe same complex code so now see the next example

5.2.11 Example

```
print('Before')
for value in [9,41,12,3,74,15]:
    if value > 20 :
        print('Largenumber',value)
print('After')
```

So we have put the if condition again with the greater sign. And in result we have achieved two greater numbers i.e 41 and 74. Value is the iterational variable, we have set it in for loop and by using if statement inside the for loop indentation , have given the logic of that this variable i.e 'Value' is greater than 20. Now only two numbers are greater than 20 and they are in the following result.

5.2.12 Result

```
 Before
Largenumber 41
Largenumber 74
After
```

As we have discussed earlier that if condition works on true and false, so here is analysis of the same but in simpler values as we are assigning the words true and false for our better understanding. But here instead of pulling out the largest number , we wish to pull out the smallest number. So the logic is changed in this classic example. there is a serious error in this example as you would notice, but we have alot of learning through that error.

5.2.13 Example

```
found = False
print ('Before' , found)
for value in [9,41,12,3,74,15] :
    if value == 3 :
        found = True
    print(found, value)
print('After',found)
```

5.2.14 Result

```
Before False
False 9
False 41
False 12
True 3
True 74
True 15
After True
```

but as you see in the result the true went to the 3 but continues to 74, and 15. This is the error. How would we get rid of this error. The error exist because after the program has found 3 as its true value it shall continue to take later the 74 and 15 to be true also, and this is because that now the variable 'found' has become true, computer does not under stand what value has to be given to which digit, it only runs on statement or commands once its true it shall follow the true untill an other statement of elif is used to nullify or make computer understand that 74 and 15 are not true. see the below example.

5.2.15 Example

```
smallest = None
print('Before')
for value in [9,41,12,3,74,15] :
    if smallest is None :
        smallest = value
    elif value < smallest :
        smallest = value
    print(smallest, value)
print('After', smallest)
```

5.2.16 Result

```
Before
9 9
9 41
9 12
3 3
3 74
3 15
After 3
```

This is how a smallest number can be pulled out from a list using thsi for loop code. Now we come back to our complex example again and try to understand this same logic again.But this time time changing the value of our variable largest_so_far to initial value of 9 in the list.

This is the last example of this sequence but the goal here is to achieve the smallest number. The code is simmilar of that the first complex example.

5.2.17 Example -Repeat of the complex one but for better Understanding with smaller value

```
largest_so_far = 9
print('Before',largest_so_far)
for the_num in [9,41,12,3,74,15] :
    if the_num < largest_so_far :
        largest_so_far = the_num
    print(largest_so_far,the_num)

print('After',largest_so_far)
```

5.2.18 Result

We can assign largest_so_far any greater value, maybe hundred, but what if we make a wiser decision by assigning the valuse of the first digit which is 9 here. By using 9 as the value of largest_so_far , we pulled out the smallest number.

```
Before 9
9 9
9 41
9 12
3 3
3 74
3 15
After 3
```

So this is how the a for loop can be used to pull out data of greater or smaller values from different lists. These can be very long and complex lists and by using logic one can find greater or smaller values or any other complex values out.

The for loop works in range function also few of the simpler exaples are as follows.

5.2.19 Example

$$x = range(6)$$

for n in x :

print(n)

5.2.20 Example

$$x = range(6)$$

for n in x :

print(n)

6 Exception

6.1 Try and Except

When writing a code program errors can be control by using try ,except and finally. The try block lets you test a block of code without displaying the Traceback error. The except block lets you handle the error but the logic for try has to be given.

6.1.1 Example

try:

print(x)

except:

print('There is something wrong')

Here x is not defined and would certainly give a Traceback error.the Try: command doesnot allow that to happen unless the logical reason is given by the except: command.One thing to note here is that try and except are not like if and else.Except must give a logical value to complete the try command.

6.1.2 Example

```
x = 'abc'
try:
    y = int(x)
except:
        y = -5
print ((x,y))
```

6.2 finally

The finally block lets you execute code, regardless of the result of the try and except blocks.The finally block will always be executed, no matter if the try block raises an error or not.

6.2.1 Example

```
try:
    x > 3
except:
        print("Something went wrong")
else:
      print("Nothing went wrong")
finally:
        print("The try...except block is finished")
```

7 Functions()

A function is a block of code which only runs when it is called.There are two kind of functions, First category is the one which is pre-defined. The pre-defined function has a built-in code in it. The other type is defined by the user. A function in python is preceded by parentheses.

7.1 Built-in functions()

Here is a list of built-in functions in python3. One main difference between btween the pre-defined(or built-in) and the defined function is indentation ':' the pre defined or built-in function does not use indentation after the parenthesis.

7.1.1 abs(), all(), any(), ascii().
bin(), bool(), breakpoint(), bytearray(), bytes().
callable(), chr(), classmethod(), compile(), complex().
delattr(), dict(), dir(), divmode().
enumerate(), eval(), exec().
filter(), float(), format(), frozenset().
getattr(), globals().
hasattr(), hash(), help(), hex().
id(), input(), int(), isinstance(), issubclass(), iter(), _import_().
len(), list(), locals().
map(), max(), memoryview(), min().
next().
object(), oct(), open(), ord().
pow(), print(), property(),
range(), repr(), reversed(), round(),
set(), setattr(), slice(), sorted(), staticmetod(), str(), sum(), super(),
tupil(), type().
vars().
zip().

Here are few mentions of bulit-in functions.

7.1.2 print()

print() is probably one of the most used function(). It is used for output, wheater on screen or any other outputdevice.

7.1.3 Example

```
print("Hello World")
```

7.1.4 type()

type is a another function , by which one can define variables. It usually does not display in python interpreters like pycharm but can be displayed in IDLE or in python terminal.

7.1.5 Example

```
x=1
type(x)
```

Result in IDLE or python commandline/terminal

<class 'int'>

7.1.6 input()

Input is a function which allow user to enter data. When input() function is written in a statement, a blank space is created which allow user to enter data, then on filling that blank space with data would show the result with programmers statement plus data accordingly.

7.1.7 Example

```
print('What is your name')
name = input()
print('Your name is ' + name)
```

7.1.8 range()

In python3 range is used with the for loop to define any set whether names, numbers or alphabets. In python2 its function was to simply print the output in a set. But in python3 a variable is also defined along with the for loop in order to execute the range() function.

7.1.9 This is how the range works:(start,stop,step)

7.1.10 Example

```
for i in range(6):
    print(i)
```

7.1.11 Example

```
for i in range(1,20,2):
    print(i)
```

7.1.12 Example

```
x = range(6)
for n in x:
    print(n)
```

As range() is a function it must have parenthesis(), but because it is used with the for loop so the statement must end with an indentation : Also there must be a variable to complete the statements as i,x and n are in 6.4's examples.

7.1.13 int()

int is built-in function which can tun any number wheather it is in decimal or not into an integer. Similarly it can turn a number as a string also into integer.

7.1.14 Example

```
x = int(5.55)
print(x)
```

7.1.15 Example

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

7.1.16 floats()

Unlike int() function, the float function turn's an integer or a number as a string into decimal numbers.

7.1.17 Example

```
x = float(5)
print(x)
```

7.1.18 Example

```
x = ("13")
print(x)
```

7.2 Defined-function as Custom functions

In python functions have parenthesis () after the keyword as their recognition. When the word def is used before, any function name can be assigned and made a custom fuction, but except, not those which are built-in fuctions or are already defined.

Although there are many the built-in functions,the need for a custom function to use in computer language is must & crucial.That is when you make your own function, according to the need. But making of one's own fuction is very complex and need to be understood in a very critical manner. A custom function is something that resembles a macro in excel or an action in photoshop. Instructions are written but in an extreamly complex and crictical manner.

A defined function has a 'def' keyword in the begining. This is how a custom fuction can be written. These functions are user's defined functions and have indentation ':' in the end. Between the def keyword and the parenthesis one can use any word but not the keyword as a function name.

Then comes the following statement inside the indentation block, it can be a variable like

```
x = a+b
```

Or can be a print function, which can be invoked by giving two line space outside the indentation block.

7.2.1 Example

```
def fruit(a,b):
    a = "apple"
    b = "banana"
    print(a,b)


fruit('a','b')
```

The result is

apple banana

If we analyze this example 6.2.1, def keyword is used followed by the word fruit which has now become a function. This means that we have assigned function properties to the name fruit by using keyword def before the name fruit . And in python's memory fruit is now a function. Indentation is given and the next line inside the indentation block has a role. This line is a statement which is defining what this function. As in the parenthesis we have given two different alphabets that are actually variables to be defined later and here in the parenthesis are called parameters. They are a and b. Now the statement after the indentation is giving value to the variable a(variable) =(operator) & string ("apple"). Then is b defined in the similar manner. The the instruction to print a & b is given. If we dont write 2 lines below fruit ('a','b') outside the indentation block, nothing would happen. This is because the function written or customize would work only after it is recalled in two lines below as function name with parenthesis and inside this parenthesis the variables which now are called arguments. These arguments should have inverted comas like string otherwise traceback error would take place. Here we are giving the instruction to print, but inside the parenthesis of print we have two give two strings, if only one string is given there would be a traceback error. Other than print() function , here the print() function won't work until or unless the new custom function fruit is written below down two spaces Then writting the name of def fuction along with the parenthesis () in the bottom outside the indentation with 'two lines' distance would be known as recalling the function,without re-call or invoke the defined function won't work. Sometimes def() function can be a bit confusing. The def() function is not predefined so it has alot of options which can be confusing. Few points to understand this and to avoid the confusions are folows. Parameters can be written inside the parenthesis. After the indentation the strings in the statement can be written. This statement is very importatnt to understand and can be according to requirements. Then finally is the invocation or calling of the function. Here in parenthesis () the writing is called arguments.

The following example is the simplest to understand the def() function.

7.2.2 Example

```
def x():
        print('This is a define function')
        print('It has indentation in it')
        print('while built-in function has no indentation in it')

x()
```

We can write numbers or strings inside the parenthesis as variables,these variables inside the def() parenthesis are called parameter.Parameters are different from arguments which are written in recalling parenthesis after the statement of the def() function. Parameter are a kind of input variables .While arguments are a kind of input value. if understood this confusion can save alot of time. Another confusion arises from the statement after the def() function, sometime its print() sometimes its return. Other time it can be logial statements like conditions or many others. Here the understanding to use statement so that the def() function can be recalled is also very important to understand.
This is an example of def() function in which the statement is in condition.

7.2.3 Example

```
def great(lang):
    if lang == 'es':
        print('Hola')
    elif lang == 'fr':
        print('bonjour')
    else:
        print('Hello')


great('en')
great('es')
great('fr')
```

7.2.4 Result

```
Hello
Hola
bonjour
```

In the Example 6.2.2 the statement used in def() function is condition of if, elif & else. Here great is the def function. Inside the parenthesis (lang) is the variable and then is the indentation : now inside-alongside the indentation all statements are part of this def() function and are recorded in a way that that the function would remember these statements and would be recalled on re-calling of the function as

great('en')

great('es')

great('fr')

7.2.5 Example

```
def place(country,city):
    print(f"country is {country},& the city is {city}")


place("Pakistan","Abbottabad")
```

7.2.6 Result

```
country is Pakistan,& the city is Abbottabad
```

Understanding Example 6.2.4 is the key. def place() is the defined function. country,city are variable called the parameters inside the parenthesis of the def() function. print() here is the statement inside this def() function, although print() itself is a pre-defined function,yet here it act like the statement inside this def() function.So the python remember's these paramenters.Parameters are input variables in this case as country and city.Inside the parenthesis() of the statement are strings along with the braces} which will allows to fill in the input value later.. Using "f" string inside this the print() parenthesis, works as a format command which allows python to understand the "input values that would be inserted in the arguments of the recall later", are to be inside the braces }, which would be country and city here.Then getting down two lines and writing the recall, which is place() and the arguments("Pakistan","Abbottabad"). The arguments are the input value inside the parenthesis of the recall i.e place("Pakistan","Abbottabad").
This completes this def() function and gives the Result 6.2.5.

7.2.7 Example

```
def income(Sale,Profit):
    income = Sale + Profit
    print("Income is Sale plus Profit")
    print(income)
income(5000,4000)
```

7.2.8 Result

```
Income is Sale plus Profit
9000
```

This is another example. Here the def() fuction define the income which contain's two parameters sales & profit.In the preceding statement income=sale+profit, the expression sale+profit are the input variables in which input values can be added later in the argumnet when def() function is recalled. Then the print's string is used to elobrate in the recall as Income is Sale plus Profit and again the print is used to print the income which is defined earlier in the parameter of the def paranthesis. Finally writting the recall of def() function with the arguments(5000,4000) gives the result 9000.

7.2.9 Return-keyword

Usually return keyword or return statement work's with the custom made function like def(), in order to complete the call or end the execution invokation made by the def() or any other custom function.

7.2.10 Example

```
def num ():
    return 3+3
print (num())
```

Below is a complex example of def() function with 'if' condition and using return keyword.

7.2.11 Example

```
def greet(lang):
    if lang == 'eng':
        return "Hello"
    elif lang == 'urdu':
        return "Salam"
    else:
        return "Bonjour"


print(greet('eng'),'Glen')
print(greet('urdu'),'Asad')
print(greet('French'),'John')
```

7.2.12 Result

```
Hello Glen
Salam Asad
Bonjour John
```

When writing a def()function, one has to be careful in giving the function name after the def(). Then inside the parenthesis is (parameter's list) and then indentation':'. After def()function line and indentation use carefully the statement. This statement is the function's body. This can be a variable, or a string, and is also very important to understand and knowing how to use ,inside that indentation variables can be written to give value or strings to use later. Also the def() function parameters can have any logical relationship with this statement i.e the function's body.

7.2.13 Example

```
def addtwo(a,b):
    add = a + b
    return add

x = addtwo(3,5)
print(x)
```

Example 6.2.12 is elaborated briefly. Here the def is the keyword of the define function added is the name of the function.(a,b) are the parameters of the fuction.Now in the next line within the indentation block is the statement i.e is the function body, in this case this is add = a+b. Note that this function body here is like a variable, it neither has parenthesis nor inverted comas like a string. As we are giving value to the word add making it a variable which's value is a+b, the def function is recording it. After this, the return keyword is used . One thing again to be noted is that return is a keyword its not a statemnet and doesnot have = or other opertors, also it does not have a parenthesis like a fuction. But this return here is also recording what to do, and that in this case return add, which is add = a+b. Note that we have not given value to a & b yet which would be given when the function is recalled. The next step is to use or recall this function. Another variable which previously is not defined yet is x . This x was not used before, in the function or in the function body and is now used in recall process. Here the value given to the x is the function name which is addtwo , this addtwo have been previously defined in the def function and now is the value of the x outside the indentation after two step space in the recall. i.e x=addtw0(3,5), here one can use arguments in the parenthesis , these arguments are the value of those a & b which we had inserted earlier in the parameters in side the parenthesis of function name addtwo. By giving space of two and recalling the function which name is addtwo. Inside the parenthesis are the arguments 3 & 5, which are the value of a & b. After the def keyword is given, python is recording each and every next command that would be given back or in computer narration revoke(re-called). Lastly we give the print command and inside its parenthesis is the variable x. This all is a process of which the computer understand this custom function.

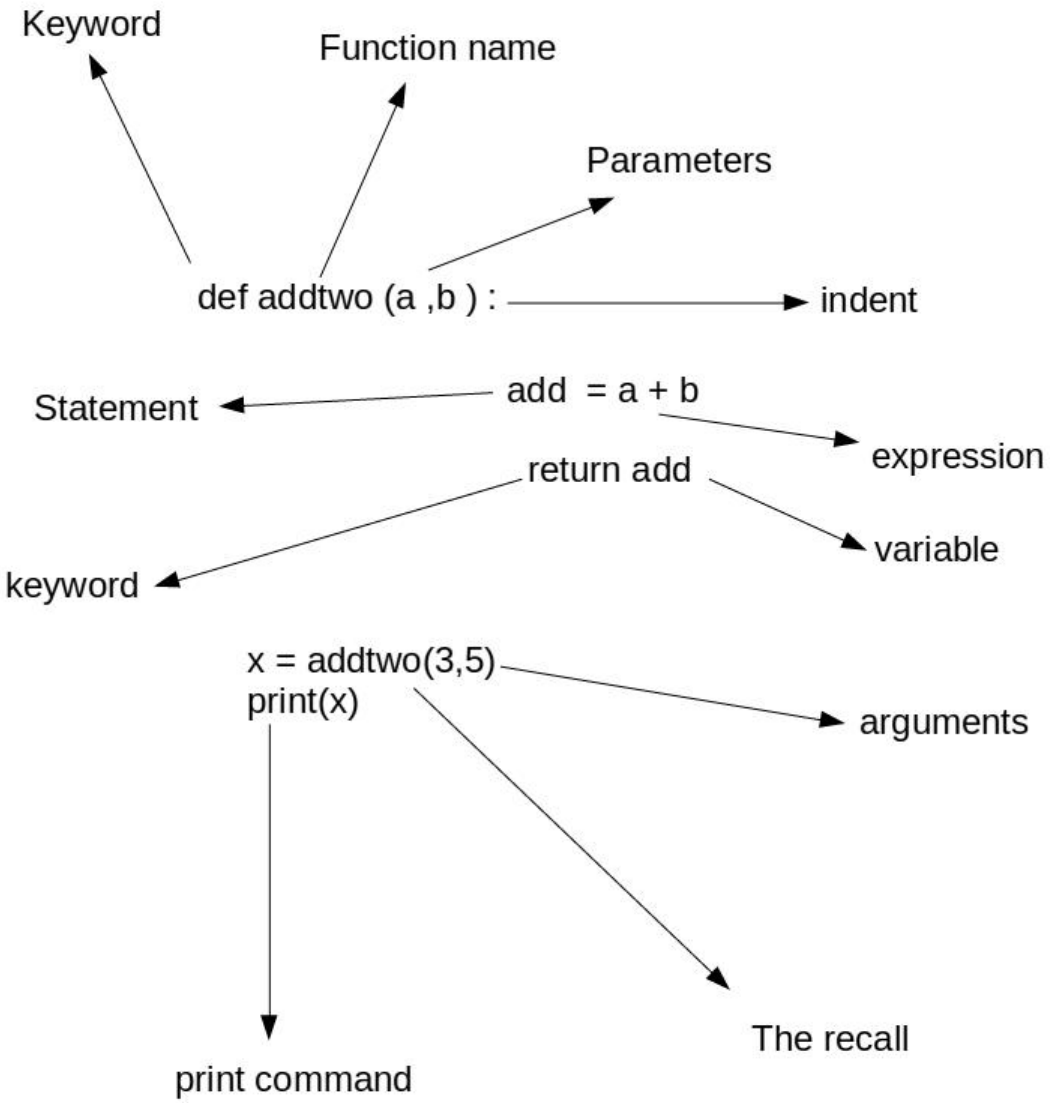

Figure 1: Diagram of the example 6.2.12

8 Strings

When we give value to a variable with words in an inverted comas ” or "" these ate the strings

x = "Hello World"

Here "Hello Word" is a string and x is a variable.

8.1 String as Data

We can store digits in the quotes in the strings beside words also.

Age = "43"

We can use this stings with numbers inside the quote to add to any number by using int() function

```
length = '5'
width = 4
area = width * int(length)
print (area)
```


8.1.1 Another Example

```
name = input ('Enter:')
print (name)

apple = input ('Enter:')
x = int (apple) - 10
print(x)
```

8.1.2 Example

```
str1 = "Hello"
str2 = "there"
bob = str1 + str2
print(bob)
str3 ='123'
x = int(str3) + 1
print(x)
```

8.2 Index of strings

In python every string has an indexing starting from 0.

8.2.1 Example

b a n a n a

0 1 2 3 4 5

By giving index value to the square brackets we can see which character has that index value. We can also use variables to assign the values of the strings and by printing those variables with the square bracket that special character can be printed.

8.2.2 Example

```
fruit = 'banana'
letter = fruit[1]
print(letter)

Answer = a

x = 3
w = fruit [ x - 1 ]
print(w)

Answer = n
```

8.3 Length of a string

Length of a string is different than the index of string. Very important to remember is that any string starts with 0 as index but when length is counted it is from 1. We can also find the length of the string by simply using the len() function.

```
>>> fruit = 'banana'
>>> x = len(fruit)
>>> print(x)
6
```

8.3.1 Example

len(banana)

6

8.3.2 Example

```
name = 'maria'
print (len(name))


Answer 5
```

8.4 Looping with Strings

We can use loops with string, this can be for retrieving data, or look in the string.

8.4.1 Example-while loop with strings

While loops are lengthy than for loops in strings and can be difficult in writing codes in comparison to for loops. How ever they can be used under certain circumstances, when needed.

```
fruit = 'banana'
index = 0
while index < len(fruit) :
    letter = fruit [index]
    print (index, letter)
    index = index + 1


Answer
0 b
1 a
2 n
3 a
4 n
5 a
```

8.4.2 Example-for with in with strings

Using for loop in much much easier than while loop in strings. This is because the for loop needs much less writing of codes than the while loop here is a side by side example of both achieving the same results

```
fruit = 'banana'
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1

Answer
b
a
n
a
n
a


for letter in 'banana' :
    print(letter)

Answer
b
a
n
a
n
a
```

Another example with an iteration variable is

8.4.3 Example

```
fruit ='banana'
for letter in fruit :
    print(letter)
Answer
b
a
n
a
n
a
```

We can use loops to find the index value of any letter in a string and their quantity, here is a classic example of using for loop to find how many 'a' are in the word banana.

```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        count = count + 1
print(count)

Answer = 3
```

Running this code would give the answer 3. There are 3 'a's in the banana. And this shows how power ful loops can be to find anything in lists or if we use broader term datas. A simple code can sort out required information from huge datas.

In this example, we can see that variable is made that is word for the string banana. Then another variable is made as the count with the value 0 . Now for loop is inserted with another iterational variable letter which would loop the the variable word's string 'banana'. Here additinaly a conditional statement of if is used to give the iterational variable equal value of the index of alphabet 'a', then count variable which wa assigned the 0 value is again assigned count+1, this is done because count = 0 and as the iteration would run one by one the + 1 will add to the 0 value of the count. thi sis of a and in the result we get three a because banana has three a. The iteration of for is going through each and every word as in the previous example it printed banana. But here the if condition has increased the notion and count is inside the indentation of if. See this modified code

```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        print(letter)

Answer
a
a
a
```

Now one understands the logic of the variable count which inside the indentation of if would sum up these three a. In the string banana b = 0 and a = 1, this is their index value And the index value by new assignment of count = count + 1 i.e total of a+a+a is 3.

```
word = 'banana'
count = 0
for letter in word :
    if letter == 'a' :
        count = count + 1
print(count)

Answer = 3
```

8.5 Slicing Strings

By using square brakets [ ] and double colons : strings can be sliced or pull out in pieces. This is also a very powerful mechanism as special words or even sentences can be obtained from heavy data.

8.5.1 Example

```
s = 'asad ali'
print(s[0:8])
print(s[6:7])
print(s[:20])
print(s[0:2])
print(s[0:])
print(s[:])

Result
asad ali
l
asad ali
as
asad ali
asad ali
```

```
>>> s = 'Monty Python'
>>> print(s[:2])
Mo
>>> print(s[8:])
thon
>>> print(s[:])
Monty Python
```

Here s is the variable and the numbers between the colon's : are the index numbers of the strings. The number on the left is the start number & the number on the right of the colon is "upto" the number, "upto" means that not the index number but one before (this is important any need to be remembered). One can use only one digit either side of the colon.

8.6 String with plus "+" Operator

We can use plus '+' operator between strings. This is known as string concatenation.

8.6.1 Example

```
a = 'Hello'
b = 'there'
print ( a + b )
c = a + '  ' + b
print(c)

Answer
Hellothere
Hello there
```

Space is also counted in the indexing of strings and can be defined between single inverted comas in the value of variables also.

8.7 'in' as a logical operator

Previously we were using in keyword with for loop. But 'in' can be used as logical operator like = or >, etc. It can be used directly with the strings or also with the if statement.

8.7.1 Example

```
fruit = 'banana'
'n' in fruit
if 'a' in fruit :
    print('Found it')

Answer is

Found it
```

8.8 Strings Comparison

Strings can be compared like upper case and lower case have different values. Similarly character set plays a role in these values.

8.8.1 Example

```
word = 'orange'
if word == 'banana':
    print('All right, bananas.')

if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Answer is
Your word,orange, comes after banana.

```
word = 'apple'
if word == 'banana':
    print('All right, bananas.')

if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Answer is
Your word,apple, comes before banana.

8.9 String Library

Strings have lot of functions(). They can be used to make changes with in strings. Here is an example of changing from upper to lower case and vice versa.

8.9.1 Example

```
greet = 'Hello Bob'
zap = greet.lower()
print (zap)
print(greet)
print('Hello There'.lower())

Answer
hello bob
Hello Bob
hello there


greet = 'Hello Bob'
nnn = greet.upper()
print(nnn)
www = greet.lower()
print(www)

Answer
HELLO BOB
hello bob
```

We can use type function in variable definition of a string also

```
>>> stuff = 'Hello world'
>>> type(stuff)
<class 'str'>
```

By typing dir() and putting the string inside the parenthesis, one can see the list of string functions that can be applied.

dir(stuff) 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill'. We can find the index value of any character within a string. One little example is below.

8.9.2 Example of find() function

```
fruit = 'banana'
pos = fruit.find('na')
print(pos)
aa = fruit.find('z')
print(aa)

Answer is
2
-1
```

Here -1 mean's that it does'nt exist.

8.10 Replace fuction()

Strings can be replaced using replace() function. Indeed practically a very useful function.

```
greet = 'Hello␣Bob'
nstr = greet.replace('Bob','Jane')
print(nstr)
print(greet)
nstr = greet.replace('o','x')
print(nstr)
Answer

Hello Jane
Hello Bob
Hellx Bxb
```

8.10.1 Whitespace—the srtip() function

Sometimes spaces are needed and sometimes spaces have to be either reduced or eliminated as required.Nothing space is called whitespace. For this strip() function is used with letter l for left and r for right.

```
>>> greet = '␣␣␣Hello␣Bob␣␣'
>>> greet.lstrip()
'Hello␣Bob␣␣'
>>> greet.rstrip()
'␣␣␣Hello␣Bob'
>>> greet.strip()
'Hello␣Bob'
>>>
```

8.11 Prefixes

```
>>> line = 'Please␣have␣a␣nice␣day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

Here startwith() has a logical output as True or False

8.12 Extraction

Now this is a classical example of utilising string functions() to extract certain data in this example the email domain name of the school.

```
>>> data ='From␣stephen.marquard@uct.ac.za␣Sat␣Jan␣5␣09:14:16␣2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find('␣',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1 : sppos]
>>> print(host)
uct.ac.za
```

9 Files

One of the learning step is readinding files. Usually text files with different character coding according to the operating system. in linux ususally UTF-8 is used its better to convert any file to UTF-8 before reading. However the standard is ASCII which is almost for every platform.

9.1 Open() function

In order to read a file a function called open()function is used. open function can have a file and a mode inside its parenthesis. There are four types of modes, read, write, append & create.

9.1.1 Type of modes

r is for read file.

a is for append file.

w is for write file.

x is to create file.

In addition you can specify if the file should be handled as binary or text mode.
"t" - Text - Default value. Text mode
"b" - Binary - Binary mode (e.g. images)

9.1.2 Example

x = open (filename, mode)

print (x)

9.1.3 Example

```
fhand = open ('mbox.txt')
print (fhand)
<_io.TextIOWrapper name='mbox.txt' mode='r' encoding='UTF-8'>
```

9.2 Newline reader

There is character in python known as back slash chrachter with n.

```
\n
```

by using this character we can break in new line

9.2.1 Example

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print(stuff)
Hello
World!
>>> stuff = 'X\nY'
>>> print(stuff)
X
Y
>>> len(stuff)
3
```

One thing to not here is that

\n

is a single character as we see by applying the len() function in X & Y example that it returned three instead of four.

9.3 Reading files in Python

The variable which is created to open file is known as file handeler. We can use loop to read a file. A very easy and simple example is as follows:-

9.3.1 Example

```
xfile = open('mbox.txt')
for cheese in xfile :
    print(cheese)
```

here the iteration variable cheese is acutally lines in the file. The for loop would read lines.

9.4 Counting

By using loops we can count lines in a file.

9.4.1 Example

```
fhand = open('mbox.txt')
count = 0
for line in fhand :
    count = count + 1
print ('Line count:', count)

Line count: 132045
```

In this example count is a variable holding value 0, purposely inserted for later use with the for loop to count lines. We can also count the whole file meaning all the chracters in a file.

9.4.2 Example

```
fhand = open('mbox-short.txt')
inp = fhand.read()
print (len(inp))
print(inp[:20])

94626
From stephen.marquar
```

We have a read() function in python. Here we assign read() function to inp variable we created to read fhand, which is a variable holding the open function of the subject file. Then by using len() function inside the print parenthesis we can count the character of the said file. We can also print a specific characters by using square brackets and colon to specify specific line.

9.4.3 Startswith()function

We can use startswith()function with an if statement inside a for loop to write any specific word incase it starts with that particular word.

9.4.4 Example

```
fhand = open('mbox-short.txt')
for line in fhand :
    if line.startswith('From:') :
        print (line)
```

We can avoid the extra space printed in between by the lines by useinf function rstrip(). By assigning rstrip() to the iterating variable of the for loop we can omit the white space.

9.4.5 Example

```
fhand = open('mbox-short.txt')
for line in fhand :
    line = line.rstrip()
    if line.startswith('From:') :
        print(line)
```

We can achieve this same result by modifying it with the key word not & continue.
Continue purpose is to to skip specific value iteration but unlike break.
Not purpose is to reverse the true to false or vice versa.
Below is an example.

9.4.6 Example

```
fhand = open('mbox-short.txt')
for line in fhand :
    line = line.rstrip()
    if not line.startswith('From:') :
        continue
    print(line))
```

We can use input() functions, to search inside the file , we can search specific value by mentioning it in the code.

9.4.7 Example

```
fname = input('Enter the file name:  ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print('There were', count, 'subject lines in', fname)
```

In order to avoid the Traceback error of missing or unidentified file we can use Try and except block. Here is an example.

9.4.8 Example

```
fname = input('Enter the file name:  ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fname)
    quit()

count = 0
for line in fhand :
    if line.startswith('Subject:') :
        count = count + 1
print('There were', count, 'subject lines in', fname)
```

If we remove the quit() function here the loop would iterate and would give in an other error, which would be of not defining the variable fhand.

9.4.9 quit function()

One way quit() is used as a function is in try & except, here one of the variable is unidentified and error is generated even after try & except because of the loop started after the try & except block. Notice that we have brought the count in the second block.

10 Lists

Data in a structured form is saved in a set this set can be assigned to variables these data sets are called collections. There are three kind of collection, they are as follows. 1.Lists 2.Dictionaries 3.Tuples

10.1 Lists definition

Lists are a type of collection. They are one of the kind of collection in which Data can be stored in square brackets [ ]. Strings and constants in large quantity can be stored in the list. We can use the len() function just like we do it for strings, but in collection the len() function would count the number of item in that particular collection , like the example of list here.

Example

```
friends = ['asad','asim','nasir']
print(friends)
print(range(len(friends)))

Result

['asad', 'asim', 'nasir']
range(0, 3)
```

In python3 the range function can display the items in the starting from with coma to the total item if used to print the list. Lists can be used with the for loop. And a single item can be printed by giving index number in square brackets inside the print parenthesis. We can also use a coma in the print parenthesis to add a string or constant.

10.1.1 Example

```
fruits = ['apple','banana','orange',pomegranatee','mango']
for␣fruit␣in␣fruits␣:
␣␣␣␣print(fruit)


print('The Last fruit is ',␣fruits[5])

Result

apple
banana
orange
pomegranate
mango
The␣Last␣fruit␣is␣␣mango
```

10.2 List Concatenating

List can be concatenated

10.2.1 Example

```
a = [1,2,3]
b = [4,5,6]
c = a + b
print(c)
print(a)
print(b)

Result
[1, 2, 3, 4, 5, 6]
[1, 2, 3]
[4, 5, 6]
```

10.3 List are Mutable

Unlike strings and Tuples lists can be changed, individual items can be changed in side the list.

10.3.1 Example

```
even = [2,4,5,8,10]
print (even)
even[2] = 6
print (even)

Result
[2, 4, 5, 8, 10]
[2, 4, 6, 8, 10]
```

Like strings lists can be sliced and an items can be pulled out.

10.3.2 Example

```
t = [2,5,7,9,13,22,1]
print(t[0:3])
Result
[2, 5, 7]
```

10.4 List Methods

$$x = list()$$

$$type(x)$$

$$< type'list' >$$

$$dir(x)$$

$$['append','count','extend','index','insert','pop','remove','reverse','sort']$$

10.5 Building Lists

We can build and fill empty lists using the append method.

10.5.1 Example

```
>>> stuff = list()
>>> stuff.append('book')
>>> stuff.append(45)
>>> print(stuff)
['book', 45]
```

Similarly sort() method can be used to sort the list.

10.5.2 Example

```
some = [9,12,3,74,42,15]
some.sort()
print(some)
[3, 9, 12, 15, 42, 74]
```

list can also use 'in' operators, which shall result in True or False values.

10.5.3 Example

```
>>> some = [9,42,12,3,74,15]
>>> 9 in some
True
>>> 44 in some
False
>>> 44 not in some
True
```

10.6 Few of List functions

Here are few of lists function but it is better to see all in python documentation.

10.6.1 Example

```
>>> some = [9,42,12,3,74,15]
>>> print(some[1])
9
>>> print(max(some))
74
>>> print(min(some))
3
>>> print(sum(some))
155
>>> print(sum(some)/len(some))
25.833333333333332
```

10.7 Lists with while loop

Here is an example of list using while loop.We can fill the list with input()function and append()method.By using this code we can find the average of the value of the list. The loops end with the word done.

10.7.1 Example

```
numlist = list()
while True :
    inp = input ('Enter a number :::')
    if inp == 'done' : break
    value = float(inp)
    numlist.append(value)
average = sum(numlist)/len(numlist)
print('Average :', average)
```

10.7.2 Result

```
Enter a number :    12
Enter a number :    13
Enter a number :    14
Enter a number :    done
Average : 13.0
```

10.8 List with split function

We can use the split function to break down the string and convert it into a list.The split() function recognises the white spaces and would split the sentence presumed to be a single string in differnt strings in a square bracket list. We can use the indexing to find any particular

word, we can also use for loop to print these now separate words in a verticl list. Here is a code forthis.Split not even omit single space but it also omit multiple spaces.

10.8.1 Example

```
stuff = abc.split()
print(stuff)
print(stuff[0])
print(len(stuff))
for words in stuff :
    print(words)
```

10.8.2 Result

```
['With', 'three', 'words']
With
3
With
three
words
```

By using arguments in the inside the split() parenthesis any thing other tham white space can be omitted also, for example in the below example semicolon ; are removed.

10.8.3 Example

```
>>> line = 'A lot of        spaces in   there  for   you'
>>> line.split()
['A', 'lot', 'of', 'spaces', 'in', 'there', 'for', 'you']
>>> line = 'first;second;third'
>>> len(line)
18
>>> thing = line.split()
>>> print(len(thing))
1
>>> print(thing)
['first;second;third']
>>> print(len(thing))
1
>>> thing = line.split(';')
>>> print(thing)
['first', 'second', 'third']
>>> print(len(thing))
3
```

10.9 extracting words in email

By using split() we can count the white spaces and extract a wod from an email or a paragraph, here is an example.

10.9.1 Example

```
fhand = open('mbox-short.txt')
for line in fhand :
    line = line.rstrip()
    if not line.startswith('From ') : continue
    words = line.split()
    print(words[2])
```

10.9.2 Result

```
Fri
Thu
Thu
Thu
Thu
Thu
Thu
```

We can pull out desired data in the below example the email address by using different methods like split in a list.

10.9.3 Example

```
line = ('From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008')
words = line.split()
email = words[1]
pieces = email.split('@')
print(pieces[1])
```

10.9.4 Result

```
uct.ac.za
```

11 Dictionaries

After lists now the second type of collections are dictionaries. Dictionary differs from lists in two aspect,

1. Dictionaries have labels. These labels are known as keys.
2. They can be put in a collection In-organised unlike the list, because of their label or keys, the items in the dictionary can be find out easily.

11.1 labels of dictionaries

The key to dictionaries are their label. They are written on the left side of the variable. Just like the list we can start with an empty dictionary and append different items in it. But along with the variable of the dictionary ,the label must be defined on the left side. This label or key is a string, this string holds a value with a colons : on the right side. Making this key a string holding values in constant.

11.1.1 Example

```
purse = dict()
purse['money']   = 12
purse['candy']   = 14
purse['tissues'] = 11
print(purse)
print(purse['candy'])

purse['candy']   = purse['candy'] + 2
print(purse)
```

11.1.2 Result

```
{'money': 12, 'candy': 14, 'tissues': 11}
14
{'money': 12, 'candy': 16, 'tissues': 11}
```

One of the main difference between dictionaries and list are, key and indexing in other words the indexing mechanism. Items in a list can be find by indexing, however in a dictionary it can be find out by the label(key) that is assigned to an item. Here is a very simple example.

11.1.3 Example

```
ddd = dict()
ddd['age'] = 21
ddd['course'] = 14
print(ddd)
```

11.1.4 Result

```
{'age': 21, 'course': 14}
```

Out put of dictionary in represented in curly brackets.Empty curly brackets can represent an empty dictionary. Similarly individual keys value can be added by using addition of constants as in the example below.

{ }

11.1.5 Example

```
jjj = {'chuck:1','fred:44','jan:100'}
print(jjj)
ooo = {}
print(ooo)

ccc = dict()
ccc['cwen'] = 1
ccc['csew'] = 1
print(ccc)
ccc ['csew'] = ccc['csew'] + 1
print(ccc)
```

11.1.6 Result

```
{'chuck:1', 'jan:100', 'fred:44'}
{}
{'cwen': 1, 'csew': 1}
{'cwen': 1, 'csew': 2}
```

We can look into the key which is not in the dictionary, we can use 'in' operators to find the True or False value weather that key is available in that particular dictionary.

11.1.7 Example

```
count = dict()
names = ['asad','maria','ammar','hamza','khadija','asad','asad']
for name in names :
    if name not in count :
        count[name] = 1
    else :
        count[name]= count[name] + 1

print(count)
```

11.1.8 Result

```
{'asad': 3, 'maria': 1, 'ammar': 1, 'hamza': 1, 'khadija': 1}
```

11.2 The Get Method

By using get() ,method , we can find the key that exist in the dictionary or not. It reduces the piece of code and simply get that specific label or key needed. Here is an example.

11.2.1 Example without get method

```
count = dict()
names = ['asad','maria','ammar','hamza','khadija','asad','asad']
for name in names :
    #print(name)
    if name not in count :
            #print(count)
            count[name] = 1
    else:
        count[name] = count[name]+1


print(count)
```

11.2.2 Example with get method

```
count = dict()
names = ['asad','maria','ammar','hamza','khadija','asad','asad']
for name in names :
    count[name] = count.get(name,0) + 1
print(count)
```

11.2.3 Same Result

```
{'asad': 3, 'maria': 1, 'ammar': 1, 'hamza': 1, 'khadija': 1}
```

Here the purpose of using zero with the key'name' is to make a histogram so that multiple keys add in zero and give the number of keys in this dictionary.

11.2.4 Example-Counting words in a Text

We can count words in text file.

```
count = dict()
print('Enter a line of Text :')
line = input('')

words = line.split()
print('Words:', words)
print('Counting...')
```

```
for word in words :
    count[word] = count.get(word,0) + 1
print('Count:', count)
```

11.2.5 Result

```
Enter a line of Text :
hello
Words: ['hello']
Counting...
Count: {'hello': 1}
```

for loops are used extensilvely in dictionaries the iteration variable in the for loop represents the key of the dictionary.

11.2.6 Example

```
counts = {'chuck': 1,'fred' : 42, 'jan' : 100}
for key in counts :
    print(key,counts[key])
```

11.2.7 Result

```
chuck 1
fred 42
jan 100
```

Value of dictionaries and key s of dictionaries can be converted to list easily.

11.2.8 Example-Converting to lists and finding items

```
jjj = {'chuck':1,'fred':42, 'jan':100}
print(list(jjj))
print(jjj.keys())
print(jjj.values())
print(jjj.items())
```

11.2.9 Result

```
['chuck', 'fred', 'jan']
dict_keys(['chuck', 'fred', 'jan'])
dict_values([1, 42, 100])
dict_items([('chuck', 1), ('fred', 42), ('jan', 100)])
```

Beside text, we can do the same technique in a file.

11.2.10 Example

```
name = input ('Enter File : ')
handel = open (name)

counts = dict()
for line in handel :
    words = line.split()
    for word in words :
        counts[word] = counts.get(word,0) + 1

bigcount = None
bigword = None
for word, count in counts.items() :
    if bigcount is None or count > bigcount :
        bigword = word
        bigcount = count

print(bigword, bigcount)
```

11.2.11 Result

```
Enter File : mbox.txt
2007 20413
```

12 Tuples

Tuples are the same like lists, except that they are immutable, i.e they can not be changed or edited. Unlike lists tupels have round brackets called parenthesis. In below exampls both list and tuples are used , lists with the square brackets and tuples with the round brackets.

12.0.1 Example

```
x = ['asad','maria','maleeha']
print(x[2])
y = (1,9,2)
print(y)
print(max(y))
for iter in y :
    print(iter)


x = [9,8,7]
x[2] = 6
print(x)
```

12.0.2 Result

```
maleeha
(1, 9, 2)
9
1
9
2
[9, 8, 6]
```

There are some limitations to tuples if we compare them to list.

12.0.3 Example

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no attribute 'reverse'
```

One reason to use tuples is that they are much more efficient then lists, like consuming lesser memory. There are many things that you can do with lists but can not with the tuple. But still tuples have there significance and are used lot in coding. The built in functions of lists are much more than tuples.Because of Python's internal organizational structures make them much more efficient than lists. Better performance is achieved on tuples in cases where they are needed rather than lists.

Another reason to use tuples instead of lists is that tuples can be used on the left side of an assignment statement. Here in the below example, two assignments can be made in one statement.

12.0.4 Example

```
(x,y) = (4,'fred')
print(y)
(a,b) = (90,95)
print(a)
```

12.0.5 Result

```
fred
90
```

As tuples are more efficient than lists in term of memory and performance lets not forget their lesser built-in functions capability also.

12.0.6 Example

```
>>> l = list()
>>> dir(l)
['append','count','extend','index','insert',
'pop','remove','reverse','sort']
>>> t = tuple()
>>> dir(t)
['count', 'index']
```

12.1 Tuples with Dictionaries

Tuples can be used with the dictionaries. In the below example an empty dictionary is made and then it is assigned values using the for loop. In the next block the for loop variable , which hold's item is assigned to tuples. Here we can see the relationship of dictionaries and tuples.

12.1.1 Example

```
#Tuple with dictionary
d = dict()
d ['asad'] = 2
d ['maria'] = 3
for (k,v) in d.items() :
    print(k,v)

print(d)
#Now this is tuple
tups = d.items()
print(tups)
```

12.1.2 Result

```
asad 2
maria 3
{'asad': 2, 'maria': 3}
dict_items([('asad', 2), ('maria', 3)])
```

12.2 Sorting list of tuples via dictionaries

We cannot use sort function directly in tuples but we can sort them via dictionaries. This is because dictionaries have relationship of key and values. So we have to make an additional step in order to transfer the data in tuples.

As list of tuples, we must remember that here we are taking advantage of the comparison operators that can be used with tuples. Here we can convert a dictionary to list and then sort that list. Here also while sorting the first item in the dictionary would come first.

12.2.1 Example

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> d.items()
dict_items([('a', 10), ('c', 22), ('b', 1)])
>>> sorted(d.items())
[('a', 10), ('b', 1), ('c', 22)]
```

In this same situation if we use the for loop we can make a sequence of key and values.

12.2.2 Example

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = sorted(d.items())
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

```
>>> for k, v in sorted(d.items()):
...     print(k, v)
...
a 10
b 1
c 22
```

12.3 Comparability of Tuples

One interesting things about tuples are that they are comparable. Meaning that they are same comparable as are strings and comparison operators with values can be used to identify between two tuples.

How ever there are some rules here, when two tuples are compared python compare's only the first value of the two tuples and would not even see the preceding values.

But if the first value is equal it would go for the next and then compare the two. Below is an example of how these operators work's in tuples.

12.3.1 Example

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Sam')
True
>>> ( 'Jones', 'Sally') > ('Adams', 'Sam')
True
```

Below is a classic example of sorting most common words in Text using tuples. This piece of code can be used to count words in any text file. The file should be in the same folder in which this code is saved as python file. This code has three blocks in the first block file is open, and 'count' empty dictionary is created. Then for loop is used to split the lines in this text file. Inside the indentation of the first for loop another for loop is created with the loop variable 'word'. Now list is created for the count which is defined as dictionary. here idiom is used to assign the value of 'counts' i.e a dictionary as lists of 'word'.

Now in the second block lst is defined as list. Again for loop is used with the items function, and a tuple is created in the next indentation line. append function is used to include key and value in the newtup tuple.

Last block is used for sorting and printing. With the sort function reverse is also used.

12.3.2 Example

```
fhand = open('file.txt')
counts = {}
for line in fhand :
    words = line.split()
    for word in words :
```

```
        counts[word] = counts.get(word,0) + 1

lst = []
for key,val in counts.items() :
    newtup = (val,key)
    lst.append(newtup)

lst = sorted(lst, reverse = True)
for val,key in lst[:10] :
    print(key,val)
```

Any text file can be used to count the number of words from this code, by replacing the file name and its presence inside the same folder of this python file.

A very interesting single line code to read and sort dictionary to tuples can be seen in the below example, showing the power & easiness in python.

12.3.3 Example

```
c = {'a':10,'b':1,'c':22}
print(sorted([(v,k) for k,v in c.items()]))
```

Here is another example with the input command and if the conditions not met than pulling data from another file name as an option.

Here 'filename' is optionally be changed. Folder of the python file and the text file should be same.

12.3.4 Example

```
fname = input ('Enter file name: ')
if len(fname) < 1 : fname = 'filename.txt'
hand = open(fname)

di = dict()
for lin in hand :
    lin = lin.rstrip()
    wds = lin.split()
    for w in wds :
        di[w] = di.get(w,0) + 1

print(di)

x = sorted(di.items())
print(x)
```

Same example but now the out put in a vertical list more compehensively defined.

12.3.5 Example

```
fname = input ('Enter file name: ')
if len(fname) < 1 : fname = 'filename.txt'
hand = open(fname)

di = dict()
for lin in hand :
    lin = lin.rstrip()
    wds = lin.split()
    for w in wds :
        di[w] = di.get(w,0) + 1

#print(di)

tmp = list()
for k,v in di.items() :
    newt = (v,k)
    tmp.append(newt)

#print('Flipped',tmp)
tmp = sorted(tmp, reverse = True)
#print('Sorted',tmp[:5])

for v,k in tmp[:5] :
    print(k,v)
```

13 Comments#

Comments# are not any kind of code. A comment# in python is represented by a hash sign #. They are programmer's comments# or descriptions written in a way that it does not play any role in execution of the code or program. They are written for remembering or for other programmers.The interpreter ignores the comments# .They can be used in helping other programmers in understanding complex executions.

13.1 Example

```
#This is a comment
print('Hello World')
```

One of the most powerful feature of comments is debugging. If a programmer's code has an error, there are several ways to debug. One way is by using print function before the error line indicated in the Traceback, if we can find the bug then we can later turn that print() function which has sought the bug and is now no longer needed in to a comment.I Instead of deleting it we can turn that into a comment by simply using # before the now not needed print() command. We can also write guardian pattern and problems encounter during the code block in text of comments , which is readable to other programmer's but would not effect the program itself.