Statements and Syntax

Python Statements:

A statement is an instruction that a Python interpreter can execute. So, in simple words, we can say anything written in Python is a statement. For example, a = 1 is an assignment statement, if statement, for statement, while statement, etc.
In other words, Instructions written in the source code for execution are called statements. There are different types of statements in the Python programming language like Assignment statements, Conditional statements, Looping statements, etc. These all help the user to get the required output. For example, n = 50 is an assignment statement.
Python supports two types in sentences

a) Single line simple statement
b) Multiline statements

Multi-Line Statements: Statements in Python can be extended to one or more lines using parentheses (), braces {}, square brackets [], semi-colon (;), continuation character slash (\). When the programmer needs to do long calculations and cannot fit his statements into one line, one can make use of these characters.

Example of Multi-Line Statements:

a) Declared using Continuation Character (\):

```
s = 1 + 2 + 3 + \
        4 + 5 + 6 + \
        7 + 8 + 9
```

b) Declared using parentheses ():

```
n = (1 * 2 * 3 + 7 + 8 + 9)
```

c) Declared using square brackets []:

```
Footballer = ['MESSI',
              'NEYMAR',
              'SUAREZ']
```

d) Declared using braces {}:

```
x = {1 + 2 + 3 + 4 + 5 + 6 +
        7 + 8 + 9}
```

e) Declared using semicolons (;):

```
Flag = 2; ropes = 3; pole = 4
```

Python Assignments:

Assignment is fundamental to Python; assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects.
We create and change variables primarily with the *assignment* statement. This statement provides an expression and a variable name which will be used to label the value of the expression.

Variable = expression

Different Forms of Assignment Statements in Python:
We use Python assignment statements to assign objects to names. The target of an assignment statement is written on the left side of the equal sign (=), and the object on the right can be an arbitrary expression that computes an object.
There are some important properties of assignment in Python:-

- Assignment creates object references instead of copying the objects.
- Python creates a variable name the first time when they are assigned a value.
- Names must be assigned before being referenced.
- There are some operations that perform assignments implicitly.

Assignment statement forms:-

1. Basic form:

This form is the most common form

```
student = 'Geeks'
print(student)
```

2. Tuple assignment:

```
# equivalent to: (x, y) = (50, 100)
x, y = 50, 100
print('x = ', x)
print('y = ', y)
```

When we code a tuple on the left side of the =, Python pairs objects on the right side with targets on the left by position and assigns them from left to right. Therefore, the values of x and y are 50 and 100 respectively.

3. List assignment:

This works in the same way as the tuple assignment.

```
[x, y] = [2, 4]
print('x = ', x)
print('y = ', y)
Output would be
x = 2
y = 4
```

4. Sequence assignment:

In recent version of Python, tuple and list assignment have been generalized into instances of what we now call sequence assignment – any sequence of names can be assigned to any sequence of values, and Python assigns the items one at a time by position.

```
a, b, c = 'HEY'
print('a = ', a)
print('b = ', b)
print('c = ', c)
Output would be
a = H
b = E
c = Y
```

5. Extended Sequence unpacking:

It allows us to be more flexible in how we select portions of a sequence to assign.

```
p, *q = 'Hello'
print('p = ', p)
print('q = ', q)
```

Here, p is matched with the first character in the string on the right and q with the rest. The starred name (*q) is assigned a list, which collects all items in the sequence not assigned to other names.

```
Output would be
p = H
q = ['e', 'l', 'l', 'o']
```

This is especially handy for a common coding pattern such as splitting a sequence and accessing its front and rest part.

```
ranks = ['A', 'B', 'C', 'D']
first, *rest = ranks
print("Winner: ", first)
print("Runner ups: ", ', '.join(rest))
Output would be
Winner: A
Runner ups: B, C, D
```

6. Multiple- target assignment:

```
x = y = 75
print(x, y)
```

In this form, Python assigns a reference to the same object (the object which is rightmost) to all the target on the left.

```
Output would be
75 75
```

7. Augmented assignment:

The augmented assignment is a shorthand assignment that combines an expression and an assignment.

```
x = 2
# equivalent to: x = x + 1
x += 1
print(x)
Output would be
```

```
3
portfolio = 0
portfolio += 150 * 2 + 1/4.0
portfolio += 75 * 1 + 7/8.0
print portfolio
```

Expressions in Python:

An expression is a combination of operators and operands that is interpreted to produce some other value. In any programming language, an expression is evaluated as per the precedence of its operators. So that if there is more than one operator in an expression, their precedence decides which operation will be performed first. We have many different types of expressions in Python.

1. Constant Expressions:

These are the expressions that have constant values only.
Example:

```
# Constant Expressions
x = 15 + 1.3
print(x)
Output would be
16.3
```

2. Arithmetic Expressions:

An arithmetic expression is a combination of numeric values, operators, and sometimes parenthesis. The result of this type of expression is also a numeric value. The operators used in these expressions are arithmetic operators like addition, subtraction, etc. Here are some arithmetic operators in Python:

Operators	Syntax	Functioning
+	x + y	Addition
–	x – y	Subtraction
*	x * y	Multiplication
/	x / y	Division
//	x // y	Quotient
%	x % y	Remainder
**	x ** y	Exponentiation

```
# Arithmetic Expressions
x = 40
y = 12
add = x + y
sub = x - y
pro = x * y
div = x / y
```

```
print(add)
print(sub)
print(pro)
print(div)
Output would be
52
28
480
3.3333333333333335
```

3. Integral Expressions:

These are the kind of expressions that produce only integer results after all computations and type conversions.

```
# Integral Expressions
a = 13
b = 12.0
c = a + int(b)
print(c)
Output would be
25
```

4. Floating Expressions:

These are the kind of expressions which produce floating point numbers as result after all computations and type conversions.

```
# Floating Expressions
a = 13
b = 5
c = a / b
print(c)
Output would be
2.6
```

5. Relational Expressions:

In these types of expressions, arithmetic expressions are written on both sides of relational operator (> , < , >= , <=). Those arithmetic expressions are evaluated first, and then compared as per relational operator and produce a boolean output in the end. These expressions are also called Boolean expressions.

```
# Relational Expressions
a = 21
b = 13
c = 40
d = 37
p = (a + b) >= (c - d)
print(p)
Output would be
True
```

6. Logical Expressions:

These are kinds of expressions that result in either True or False. It basically specifies one or more conditions. For example, (10 == 9) is a condition if 10 is equal to 9. As we know it is not correct, so it will return False. Studying logical expressions, we also come across some logical operators which can be seen in logical expressions most often. Here are some logical operators in Python:

Operator	Syntax	Functioning
and	P and Q	It returns true if both P and Q are true otherwise returns false
or	P or Q	It returns true if at least one of P and Q is true
not	not P	It returns true if condition P is false

Example

```
P = (10 == 9)
Q = (7 > 5)
# Logical Expressions
R = P and Q
S = P or Q
T = not P
print(R)
print(S)
print(T)
Output would be
False
True
True
```

7. Bitwise Expressions:

These are the kind of expressions in which computations are performed at bit level.

```
# Bitwise Expressions
a = 12
x = a >> 2
y = a << 1
print(x, y)
Output would be
3 24
```

8. Combinational Expressions:

We can also use different types of expressions in a single expression, and that will be termed as combinational expressions.

Example

```
# Combinational Expressions
a = 16
b = 12
c = a + (b >> 1)
print(c)
Output would be
22
```

Python Print Function

Python **print() function** prints the message to the screen or any other standard output device. The message can be a string, or any other object, the object will be converted into a string before written to the screen.

Syntax

print*(object(s)*, sep=*separator*, end=*end*, file=*file*, flush=*flush*)

Parameter	Description
object(s)	Any object, and as many as you like. Will be converted to string before printed
sep='*separator*'	Optional. Specify how to separate the objects, if there is more than one. Default is ' '
end='*end*'	Optional. Specify what to print at the end. Default is '\n' (line feed)
file	Optional. An object with a write method. Default is sys.stdout
flush	Optional. A Boolean, specifying if the output is flushed (True) or buffered (False). Default is False

```
print("Hello", "how are you?")
```

```
x = ("apple", "banana", "cherry")
print(x)
```

Example Use of Separator

```
print("Hello", "how are you?", sep="---")
```

```
# Python 3.x program showing
# how to print data on
# a screen

# One object is passed
print("GeeksForGeeks")

x = 5
# Two objects are passed
print("x =", x)
# code for disabling the softspace feature
print('G', 'F', 'G', sep='')
# using end argument
print("Python", end='@')
print("GeeksforGeeks")
Output would be
GeeksForGeeks
x = 5
GFG
Python@GeeksforGeeks
```

Conditional Statements if, Multiway Branching

Decision-making is as important in any programming language as it is in life. Decision-making in a programming language is automated using conditional statements, in which Python evaluates the code to see if it meets the specified conditions.
The conditions are evaluated and processed as true or false. If this is found to be true, the program is run as needed. If the condition is found to be false, the statement following the If condition is executed.
Conditional Statement in Python perform different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

Python has six conditional statements that are used in decision-making:-

1. If the statement
2. If else statement
3. Nested if statement
4. If…Elif ladder
5. Short Hand if statement
6. Short Hand if-else statement

If Statement

The If statement is the most fundamental decision-making statement, in which the code is executed based on whether it meets the specified condition. It has a code body that only executes if the condition in the if statement is true. The statement can be a single line or a block of code.
The if statement in Python has the subsequent syntax:

```
if expression
 Statement

#If the condition is true, the statement will be executed.
```

Example – 1

```
num = 5
if num > 0:
    print(num, "is a positive number.")
print("This statement is true.")
#When we run the program, the output will be:
5 is a positive number.
This statement is true.
```

Example – 2

```
a = 25
b = 170
if b > a:
  print("b is greater than a")
output : b is greater than a
```

If Else Statement

This statement is used when both the true and false parts of a given condition are specified to be executed. When the condition is true, the statement inside the if block is executed; if the condition is false, the statement outside the if block is executed.
The if…Else statement in Python has the following syntax:

```
 if condition :
              #Will executes this block if the condition is true
```

```
    else :
                #Will executes this block if the condition is false
Example -1
num = 5
if num >= 0:
    print("Positive or Zero")
else:
     print("Negative number")
output : Positive or Zero
```

If…Elif..else Statement

In this case, the If condition is evaluated first. If it is false, the Elif statement will be executed; if it also comes false, the Else statement will be executed.

The If…Elif..else statement in Python has the subsequent syntax:

```
if condition :
    Body of if
elif condition :
    Body of elif
else:
    Body of else
```

Example -1

```
num = 7
if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
output:  Positive number
```

Nested IF Statement

A Nested IF statement is one in which an If statement is nestled inside another If statement. This is used when a variable must be processed more than once. If, If-else, and If…elif…else statements can be used in the program. In Nested If statements, the indentation (whitespace at the beginning) to determine the scope of each statement should take precedence.

The Nested if statement in Python has the following syntax

```
if (condition1):
#Executes if condition 1 is true
if (condition 2):
  #Executes if condition 2 is true
  #Condition 2 ends here
#Condition 1 ends here
```

Example-1

```
num = 8
if num >= 0:
    if num == 0:
        print("zero")
    else:
        print("Positive number")
else:
    print("Negative number")
output: Positive number
```

Short Hand if statement

Short Hand if statement is used when only one statement needs to be executed inside the if block. This statement can be mentioned in the same line which holds the If statement.
The Short Hand if statement in Python has the following syntax:

```
if condition: statement
```

Example -1

```
i=15
# One line if statement
if i>11 : print ("i is greater than 11")
The output of the program : "i is greater than 11."
```

Short Hand if-else statement

It is used to mention If-else statements in one line in which there is only one statement to execute in both if and else blocks. In simple words, If you have only one statement to execute, one for if, and one for else, you can put it all on the same line.
Examples - 1

```
#single line if-else statement
a = 3
b = 5
print("A") if a > b else print("B")
output: B
```

```
#single line if-else statement, with 3 conditions
a = 3
b = 5
print("A is greater") if a > b else print("=") if a == b else print("B is
greater")
output: B is greater
```

Multiway Branching

In typical programing environment if then else can be used as multiway branching, however for complex situation or to avoid repetitive typing and to have good debugging we may use SWITCH statement.
Switch statement refers to multiway branching. For Python versions below 3.10 does not have switch statement.

Writing if-else statements as multiway branching:

Let's assume that we have a variable named "choice" that takes some string values and based on the value of the variable will then print a float number.
Using a series of if-else statements that would look like the code snippet shown below:

```
if choice == 'optionA':
    print(1.25)
elif choice == 'optionB':
    print(2.25)
elif choice == 'optionC':
    print(1.75)
elif choice == 'optionD':
    print(2.5)
else:
    print(3.25)
```

Writing a switch statement using a dictionary

Another possibility (if you are using Python < 3.10) are dictionaries since they can be easily and efficiently indexed. For instance, we could create a mapping where our options correspond to the keys of the dictionaries and the values to the desired outcome or action.

```
choices = {
    'optionA': 1.25,
    'optionB': 2.25,
    'optionC': 1.75,
    'optionD': 2.5,
}
```

Finally, we can pick the desired choice by providing the corresponding key:

```
choice = 'optionA'
print(choices[choice])
```

Now we deal with the case where our choice is not included in the specified dictionary. If we try to provide a non-existent key, we will get back a KeyError. There are essentially two ways we can handle this situation.

The first option requires the use of get() method when selecting the value from the dictionary that also allows us to specify the default value when the key is not found. For example,

```
>>> choice = 'optionE'
>>> print(choices.get(choice, 3.25)
3.25
```

We can simplify the above complexity by using switch statement

```
choice:
    case 'optionA':
        print(1.25)
    case 'optionB':
        print(2.25)
    case 'optionC':
        print(1.75)
    case 'optionD':
        print(2.5)
    case _:
        print(3.25)
```

The above programing style is called multiway branching.

Note that we can even combine several choices in a single case or pattern using | ('or') operator as demonstrated below:

```
case 'optionA' | 'optionB':
    print(1.25)
```

Looping controls while, for loop coding techniques

Python While Loop:

Python While Loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

Syntax:

```
while expression:
    statement(s)
```

When a while loop is executed, expr is first evaluated in a Boolean context and if it is true, the loop body is executed. Then the expr is checked again, if it is still true then the body is executed again and this continues until the expression becomes false.

Example -1

```
# Python program to illustrate
# while loop
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
```

Output would be

```
Hello Geek
Hello Geek
Hello Geek
```

In the above example, the condition for while will be True as long as the counter variable (count) is less than 3

Example - 2: Python while loop with list

```
# checks if list still
# contains any element
a = [1, 2, 3, 4]

while a:
    print(a.pop())
```

Output would be

```
4
3
2
1
```

Example - 3: Single statement while block

```
# Python program to illustrate
# Single statement while block
count = 0
while (count < 5): count += 1; print("Hello Geek")
```

Output would be

```
Hello Geek
Hello Geek
Hello Geek
Hello Geek
Hello Geek
```

Example 4: Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Continue Statement

Python Continue Statement returns the control to the beginning of the loop.

Example: Python while loop with continue statement

```
# Prints all letters except 'e' and 's'
i = 0
a = 'geeksforgeeks'

while i < len(a):
    if a[i] == 'e' or a[i] == 's':
        i += 1
        continue

    print('Current Letter :', a[i])
    i += 1
```

Output would be

```
Current Letter : g
Current Letter : k
Current Letter : f
Current Letter : o
Current Letter : r
Current Letter : g
Current Letter : k
```

Break Statement

Python Break Statement brings control out of the loop.

Example: Python while loop with a break statement

```
# break the loop as soon it sees 'e'
# or 's'
i = 0
a = 'geeksforgeeks'
while i < len(a):
    if a[i] == 'e' or a[i] == 's':
        i += 1
        break

    print('Current Letter :', a[i])
    i += 1
```

Output would be

```
Current Letter : g
```

Python For Loops

Python For loop is used for sequential traversal i.e. it is used for iterating over an iterable like String, Tuple, List, Set or Dictionary.
In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is "for" loop which is similar to each loop in other languages. Let us learn how to use for in loop for sequential traversals

Syntax

```
for var in iterable:
    # statements
```

Here the iterable is a collection of objects like lists, tuples. The indented statements inside the for loops are executed once for each item in an iterable. The variable var takes the value of the next item of the iterable each time through the loop.
Examples of For Loops in Python

Example 1: Using For Loops in Python List

```
# Python program to illustrate
# Iterating over a list
l = ["geeks", "for", "geeks"]
for i in l:
    print(i)
```

Output:

```
Geeks
for
geeks
```

Example 2: Using For Loops in Python Dictionary

```
# Iterating over dictionary
print("Dictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d:
    print("% s % d" % (i, d[i]))
```

Output:

```
Dictionary Iteration
xyz  123
abc  345
```

Example 3: Using For Loops in Python String

```
# Iterating over a String
print("String Iteration")
s = "Geeks"
for i in s:
    print(i)
```

Output:

```
String Iteration
G
e
e
k
s
```

Continue Statement in Python
Python continue Statement returns the control to the beginning of the loop.
Example: Python for Loop with Continue Statement

```
# Prints all letters except 'e' and 's'
for letter in 'geeksforgeeks':
    if letter == 'e' or letter == 's':
        continue
    print('Current Letter :', letter)
```

Output:

```
Current Letter : g
Current Letter : k
Current Letter : f
Current Letter : o
Current Letter : r
Current Letter : g
Current Letter : k
```

Break Statement in Python
Python break statement brings control out of the loop.
Example: Python For Loop with Break Statement

```
for letter in 'geeksforgeeks':

    # break the loop as soon it sees 'e'
    # or 's'
    if letter == 'e' or letter == 's':
        break

print('Current Letter :', letter)
```

Output:

```
Current Letter : e
```

Python Iterators

An iterator is an object that contains a countable number of values.
An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().

Iterator vs Iterable
Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.
All these objects have a iter() method which is used to get an iterator:
Example:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
```

```
print(next(myit))
print(next(myit))
print(next(myit))
```

Example

Strings are also iterable objects, containing a sequence of characters:

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

Example

Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

Create an Iterator

To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called __init__(), which allows you to do some initializing when the object is being created.

The __iter__() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The __next__() method also allows you to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)
```

```
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
Output
1
2
3
4
5
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
Example:
Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
Without list comprehension you will have to write a
for statement with a conditional test inside:
Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

Output

```
['apple', 'banana', 'mango']
```

Range iterators

Python range function generates a list of numbers which are generally used in many situation for iteration as in for loop or in many other cases. In python range objects are not iterators. range is a class of a list of immutable objects. The iteration behavior of range is similar to iteration behavior of list in list and range we can not directly call next function. We can call next if we get an iterator using iter.
Example

```
# Python program to understand range
# this creates a list of 0 to 5
# integers
demo = range(6)
# print the demo
print(demo)
# it will generate error
print(next(demo))
Output
range(0, 6)
```

```
In above example you will get following runtime error

Traceback (most recent call last):
  File "/home/6881218331a293819d2a4c16029084f9.py", line 13, in
    print(next(demo))
TypeError: list object is not an iterator
```

Note: Above runtime error clearly indicates that python range is not a iterator.
Example:

```
# Python program to understand range

# creates a demo range
demo = range(1, 31, 2)

# print the range
print(demo)

# print the start of range
print(demo.start)

# print step of range
print(demo.step)

# print the index of element 23
print(demo.index(23))

# since 30 is not present it will give error
print(demo.index(30))
```

Output

```
range(1, 31, 2)
1
2
11
```

Runtime Error: Since element 30 is not present it will rise an error

```
 Traceback (most recent call last):
  File "/home/cddaae6552d1d9288d7c5ab503c54642.py", line 19, in
    print(demo.index(30))
ValueError: 30 is not in range
```

The Map Zip and Filter function

Python provides some special functions which are iterable over a range

Map Function:

map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

Syntax:

```
map(fun, iter)
```

Parameters:

fun : It is a function to which map passes each element of given iterable.
iter : It is a iterable which is to be mapped.

NOTE: You can pass one or more iterable to the map() function.
Returns:

```
Returns a list of the results after applying the given function
to each item of a given iterable (list, tuple etc.)
```

Example 1:

```
# Python program to demonstrate working
# of map.

# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Output:

```
[2, 4, 6, 8]
```

Example 2:
We can also use lambda expressions with map to achieve above result.

```
# Double all numbers using map and lambda
numbers = (1, 2, 3, 4)
result = map(lambda x: x + x, numbers)
print(list(result))
```

Output:

```
[2, 4, 6, 8]
```

Example 3:

```
# Add two lists using map and lambda
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```

Output:

```
[5, 7, 9]
```

Example 4:

```
def myfunc(a, b):
  return a + b

x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon',
'pineapple'))
print(x)
#convert the map into a list, for readability:
print(list(x))
```

Output:

```
<map object at 0x034244F0>
['appleorange', 'bananalemon', 'cherrypineapple']
```

Zip and Filter Functions:

When you need to filter out data based on a condition, you can easily do so by using python's built-in **filter** function.

If you want to combine two lists (or any other iterables) into one, you can use Python's built-in **zip** function.

Filter function

As you can see from the below syntax, the filter function takes just two arguments: a function and an iterable to filter out the data as per your need.

Syntax

```
filter(function or None, iterable)
```

- function / None: first, you need to write a function that contains the condition to filter the data. When this function is used with the filter it returns those elements of iterable for which function returns true.
- iterable: input data on which you need to perform the filter operation. It can be any iterable like sets, lists, tuples, dictionaries, etc.

Result: Filter function returns an iterator object having only the filtered data as per your conditions (as written in the function/lambda).

Example

```
>>> marks = [35, 85, 90, 30, 55, 99, 75, 66, 45]
>>> first_class = filter(lambda x: x>= 60, marks)
>>> print("\nFirst Class Student Marks")
>>> print("--"*15)
>>> for mark in first_class:
...     print(mark)
```

Output:

```
First Class Student Marks
------------------------------
85
90
99
75
66
```

In short:

filter() - The filter() function takes in a provided sequence or iterable along with a filtering criteria (a function or lambda). It then tests every element in the sequence to determine if the element fits the filtering criteria, returning only the elements that match that criteria.

Zip Function:

zip() - The zip() function takes two iterable objects and returns a tuple of paired elements. The first item in both iterables is paired, the second item in both iterables is paired together, and so on.

Python zip() method takes iterable or containers and returns a single iterator object, having mapped values from all the containers.
It is used to map the similar index of multiple containers so that they can be used just using a single entity.

Example 1:

```
name = [ "Manjeet", "Nikhil", "Shambhavi", "Astha" ]
roll_no = [ 4, 1, 3, 2 ]
# using zip() to map values
mapped = zip(name, roll_no)
print(set(mapped))
```

Output:

```
{('Shambhavi', 3), ('Nikhil', 1), ('Astha', 2), ('Manjeet', 4)}
```

Example 2:

```
names = ['Mukesh', 'Roni', 'Chari']
ages = [24, 50, 18]

for i, (name, age) in enumerate(zip(names, ages)):
    print(i, name, age)
```

Output:

```
0 Mukesh 24
1 Roni 50
2 Chari 18
```

Multiple V/s Single Iterators

Basically logically iterators are like for loop with default step and works like function
Example:
>>>Nums = [1, 2, 3, 4, 5, 6]
#above is a list of numbers
#now let we define iterator1st with iter function as
>>>it1 =iter(Nums)
#now let we define iterator 2nd with iter function as
>>>it2 =iter(Nums)
#We have two iterators it1 and it2 both are applied over a list
#Say we moved it1 with next as
>>>next(it1)
#this will print 1, again we use
>>>next(it1)
#this will print 2, again we use
>>>next(it1)
#this will print 3, again we use
#now we will use
>>>next(it2)
#this will print 2, again we use
>>>next(it2)

The above example shows that on a list if we define two separate iterators they behave independently.

Generators in Python

Python provides a generator to create your own iterator function. A generator is a special type of function which does not return a single value, instead, it returns an iterator object with a sequence of values. In a generator function, a **yield** statement is used **rather than a return** statement. The following is a simple generator function.

Example: Generator Function

```
def mygenerator():
    print('First item')
    yield 10

    print('Second item')
    yield 20

    print('Last item')
    yield 30
```

In the above example, the mygenerator() function is a generator function. It uses yield instead of return keyword. So, this will return the value against the yield keyword each time it is called. However, you need to create an iterator for this function, as shown below.

Example: next()

```
>>> gen = mygenerator()
>>> next(gen)
First item
10
>>> next(gen)
Second item
20
>>> next(gen)
Last item
30
```

The generator function cannot include the return keyword. If you include it, then it will terminate the function. The difference between yield and return is that yield returns a value and pauses the execution while maintaining the internal states, whereas the return statement returns a value and terminates the execution of the function.

Timer in Python

A timer in Python is a time-tracking program. Python developers can create timers with the help of Python's time modules. There are two basic types of timers: timers that count up and those that count down.

Stopwatches

Timers that count up from zero are frequently called stopwatches. We can use these to record the amount of time it takes to complete a task.

Runners often use stopwatch timers to record how long it takes them to complete a lap around a course or finish an entire run. You can use a stopwatch to track how long it takes to complete any task, such as coding a simple program.

Programmers often use stopwatch timers to compare the performance of various Python solutions. By seeing how long each solution takes to execute, you can choose the program that runs the fastest. Over time, this will allow you to better understand computational complexity and build intuition for choosing efficient solutions. These skills go a long way towards becoming a proficient programmer.

Countdown Timers

There are also countdown timers, set with a specific amount of time that depletes until the timer reaches zero. You can use Python to build countdown timers that serve different purposes.

For instance, you can build a cooking countdown timer to ensure you don't leave food in the oven for too long. Countdown timers can also work to display the time remaining in a sporting event or during an exam. They are also used to count down the hours and minutes to a movie release or big event. The possibilities are endless!

Example-

```
import time
seconds = time.time()
print("Time in seconds since the epoch:", seconds)
local_time = time.ctime(seconds)
print("Local time:", local_time)
```

Example-

```
import time
print("This is the start of the program.")
time.sleep(5)
print("This prints five seconds later.")
```

Example-

```
import datetime
current = datetime.datetime.now()
print ("Current date:", str(current))
one_year = current + datetime.timedelta(days = 365)
print("The date in one year:", str(one_year))
```

Functions: (Function Scope), (Function Arguments), (Function Types) (Recursion Function)

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters/arguments, into a function.
- A function can return data as a result.

Creating a Function

```
def my_function():
  print("Hello from a function")
```

Calling a Function

```
def my_function():
  print("Hello from a function")

my_function()
```

Function Scope:

There are two type of variables associated with python

Local Variables
Global Variables

Local Variables have local scope
Global Variables have global scope

Local Variable:

A variable created inside a function belongs to the ***local scope*** of that function, and can only be used inside that function.

Example

A variable created inside a function is available inside that function:

```
def myfunc():
  x = 300
  print(x)

myfunc()
```

In above example of myfunc() function x is defined and assigned value of 300 inside the function definition therefore it has local scope and called local variable.

Global Variable:

A variable created in the main body of the Python code is a global variable and belongs to the global scope.

Global variables are available from within any scope, global and local.

Example

```
A variable created outside of a function is global and can be used by anyone:
x = 300
def myfunc():
  print(x)

myfunc()

print(x)
```

Example Global Vs Local Variable

```
x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

Output would be:

```
200
300
```

In some cases global Keyword can be used to declare global variable

Example

```
def myfunc():
  global x
  x = 300

myfunc()

print(x)
```

Example

```
x = 300
def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

Function Arguments

The terms ***parameter*** and *argument* can be used for the same thing: information that are passed into a function.

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that is sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Ankit", "Patil")
```

Functions with Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

Example

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Function with default Argument value

The following example shows how to use a default parameter value.
If we call the function without argument, it uses the default value:

Example

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Function Types

There are many types of Python Functions. Each of the function type is important in its own way. The following are the different types of Python Functions:

- Python Built-in Functions
- Python Recursion Functions
- Python Lambda Functions
- Python User-defined Functions

Python Built-in Functions

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, **print() function** prints the given object to the standard output device (screen) or to the text stream file. In Python 3.6, there are 68 built-in functions.

Python Recursion Functions

When a function calls itself inside its body then such arrangement is called as Python Recursion Functions

Python Lambda Functions

In Python, an anonymous function is a function that is defined without a name.
While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword. Hence, anonymous functions are also called lambda functions.
A Lambda function in python has the following syntax:

```
lambda arguments: expression
```

```
# Program to show the use of lambda functions

double = lambda x: x * 2

# Output: 10
print(double(5))
```

The statement double = lambda x: x * 2 is nearly the same as

```
def double(x):
   return x * 2
```

Python User-defined Functions

Functions that we define ourselves to do the certain specific task are referred to as user-defined functions. If we use functions written by others in the form of the library, it can be termed as library functions.
All the functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Recursion Function

Recursion is the process of defining something in terms of itself.
In Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.
Following is an example of a recursive function to find the factorial of an integer.
Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 5 (denoted as 5!) is 1*2*3*4*5 = 120.

```
# An example of a recursive function to
# find the factorial of a number

def calc_factorial(x):
    <em>"""This is a recursive function
    to find the factorial of an integer"""

    </em>if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))

num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

In the above example, **calc_factorial()** is a recursive function as it calls itself.

Function Objects:

In Python, functions behave like any other object, such as an int or a list. That means that you can use functions as arguments to other functions, store functions as dictionary values, or return a function from another function.
Example

```
def square(x):
    """Square of x."""
    return x*x

def cube(x):
    """Cube of x."""
    return x*x*x


# create a dictionary of functions
funcs = {
    'square': square,
    'cube': cube,
}
```

Anonymous functions:

When using functional style, there is often the need to create small specific functions that perform a limited task as input to a HOF (Higher Ordered Functions) such as map or filter. In such cases, these functions are often written as anonymous or lambda functions. If you find it hard to understand what a lambda function is doing, it should probably be rewritten as a regular function.

Example using regular function:

```
# Using standard regular function
def square(x):
    return x*x

print map(square, range(5))
```

Output

```
[0, 1, 4, 9, 16]
```

The above example using anonymous function can be written as:

```
# Using an anonymous function
print map(lambda x: x*x, range(5))
```

Output

```
[0, 1, 4, 9, 16]
```

Units:

In many cases we do calculations in terms of units. Sometimes we need to have unit conversion too. In such cases python comes with many unit related libraries amongst them unit is one of the library which can be used for direct units calculations and even conversions. However it is not a standard part of python's package you need to install the unit library.
To install units library use

```
pip install units
```

To use the unit library you need to import the library
Example

```
>>> from units import unit
>>> metre = unit('m')
>>> print(metre(7) + metre(11))
18.00 m
```