**About Python**

Python was created by Guido Van Rossum in 1991 as a general-purpose programming language. It is an interactive and interpreted language. ***It supports both structured and object-oriented*** style of programming. Python is an open source language under General Public License(GPL).

Advantage of Python is that it has wide range of libraries and built-in functions which aid in rapid development of applications. Python is easy to learn and follows a simple syntax, so it is a good choice for beginners. All the codes are executing at the runtime.No compilation required.

Python is scripting, general-purpose, high-level, and interpreted programming language. It also provides the object-oriented programming approach. The filename extension of Python can be various types such as .py, .pyw, .pyc, .pyd, .pyz.

**What is a scripting language?**

The scripting language is referred to perform the task based on automating a repeated task. It includes same types of steps while implementing the procedure or program. It reduces time and cuts the costs further. ***The scripting languages are interpreted language instead of a compiled language***. The name of few scripting languages is [Perl](https://www.javatpoint.com/perl-tutorial), Visual Basic, [JavaScript](https://www.javatpoint.com/javascript-tutorial), [Python](https://www.javatpoint.com/python-tutorial), Unix Shell Scripts, ECMAScript, and [Bash](https://www.javatpoint.com/bash) etc.

**Features of Scripting language:**

Below are the features of the scripting language.

* It runs faster and complete task efficiently.
* It is easy to learn and easy to use.
* No IDEs are required to write code.
* It is suitable for automation tasks.
* Scripting language doesn't require the memory to run the program.
* Less line code requires completing the task as compared to other languages.

# Prerequisites

Knowledge on below concepts of any programming language will be useful.

* Identifiers and keywords
* Variables
* Datatypes
* Branching statements
* Looping statements
* Functions
* Type conversion

# Following are some of the important features of Python:

* **Python is open source:** The Python implementation is under an open source license that makes it freely usable and distributable, even for commercial use.
* **Python is interpreted:** Python is a high level language which is interpreted by a Python interpreter.
* **Python is cross platform compatible:** Python can be executed on all major platforms like Windows, Linux/Unix, OS/2, Mac and others.
* **Python is Object-Oriented:**In Python, we encapsulate data within the objects as it supports the object-oriented style of programming.
* **Python is a great choice for new learners:** Python is easy to learn and follows a simple syntax, so it is a good choice for beginner programmers. Python also supports wide range of application development.
* **Python is extensible:** Python has a wide range of libraries and built-in functions which helps in easy and rapid development of applications.
* **Python is interactive:** Python users are provided a command prompt where they can interact directly with the interpreter to write programs.
* **Database connectivity:** Python provides interfaces required to connect to all major databases like Oracle, MySQL, PostgreSQL and others.

# Identifiers:

In Python, variables, functions, classes, modules and objects are identified using a name known as an identifier. An identifier can start with an uppercase or lowercase character or an underscore (\_) followed by any number of underscores, letters and digits. ***All identifiers in Python are case sensitive.***

**Example:**weight=10

In the above example, weight is an identifier.

# Keywords:

Keywords are the **reserved words** in Python. So keywords cannot be used as identifiers or to name variables and functions. Few of the keywords are listed below.

**Example:** if, else, elif, for, where, break, continue

variables are like containers for data (i.e. they hold the data) and the value of the variable can vary throughout the program.

**Declaring a variable:**

**Syntax:**var\_name = literal\_value

where var\_name is the name given to the container holding the value specified as literal\_value in the syntax above.

**Example:** weight=10

In the above example, *weight* is the container holding the value *10*  which can change during the execution of the program.

Python may have data belonging to different types. Common data types used in programming are listed below:

| **Category** | **Datatype** | **Example** |
| --- | --- | --- |
| Numeric | int | 123 |
| long | 1237126381763817 |
| Numeric with decimal point | float | 123.45 |
| double | 123123.32345324 |
| Alphanumeric | char | A |
| string | Hello |
| Boolean | boolean | True, False |

**Python is a dynamically typed language!**

In the above example, no **datatype** was mentioned at the time of declaring variable. In Python, the datatype of a variable is decided automatically at the time of execution based on the value assigned to it. This is called as **dynamic typing**.

num=65 #line 1

num="A" #line 2

In line 1, variable *num* is assigned a value *65* which is an integer, so the data type of *num* variable is integer in line 1.

In line 2, variable *num* is assigned a value “*A*” which is a string, so the data type of the *num* variable is string in line 1.

**Note:** To check the datatype of the variable we can use **type(var\_name)**which in turn returns the **datatype** of the variable.

**Example:**

1. num=65
2. print(num,type(num))
3. num="A"
4. print(num,type(num))

**Output:**

65<class'int'>  
A <class 'str'>

# The input() function:

Python provides the **input()** built-in function to read an input from the user using the standard input device (i.e. keyboard). The input() function always returns string data irrespective of the type of data entered through the keyboard.

**Syntax:**var\_name = input([“interactive statement”])

where,

**var\_name** is the variable assigned with the string value which is read using input method.

**Interactive statement** is the statement displayed to the user expecting the response from them.

**Example:**

1. input\_var=input("please enter the value")
2. print(input\_var)

**sample output:**

please enter the value 100

100

# The print() function:

Python provides the **print()** built-in function to display the output onto the standard output device (i.e. Monitor)

**Syntax:**print(“var\_name1, var\_name2, …”, [end=”value1”, sep=”value2”])

where,

**var\_name1, var\_name2** are the variable names or the literals you want to print or output

**end** is used to specify the separator between two print statements which is ‘\n’ by default

**sep** is used to specify the separator between multiple variables displayed using a single print statement.

**Example:**

1. a="cat"
2. b=20.127
3. c=10
4. print(a,b,c)
5. print(a,b,c,sep=":")
6. print(a,b,c,end=" ")
7. print(a,b,c)
8. print("b=%0.2f" %b)
9. print("c=%8d" %c)
10. print("c=%-8d" %c)

**Output:**

cat 20.127 10

cat:20.127:10 #seperator between variables changed to ‘:’

cat 20.127 10 cat 20.127 10 #seperator between two print statement changed to " "

b=20.13 #as the format is 0.2 value is rounded of two decimal digits

c= 10 #right aligned within the reserved 8 spaces

c=10 #left aligned within the reserved 8 spaces as there is – symbol

Operators in Python are the symbols used to indicate the operation to be performed. Some of the most common operators used in Python are listed below:

| **Category** | **Operators** |
| --- | --- |
| Arithmetic Operators | +,-,\*,/, %,// |
| Relational Operators | ==,!=,>,<,>=,<= |
| Assignment Operators | =,+=,-=,\*=,/=,%= |
| Logical Operators | and,or,not |

# Arithmetic operators:

| **Operator** | **Explanation** | **Example** |
| --- | --- | --- |
| + | Used for addition operation | "+" is used as addition operator where 11+2 is evaluated as 13 |
| - | Used for subtraction operation | "-" is used as subtraction operator where 11-2 is evaluated as 9, 2-11 is evaluated as -9 |
| \* | Used for multiplication operation | "\*" is used as multiplication operator where 11\*2 is evaluated as 22 |
| / | Used for division operation | "/" is used as division operator where 11/2 is evaluated as 5.5 |
| // | Used for integer division operation | "//" is used for integer division where 11//2 is evaluated as 5 |
| % | Used for modulo operation. Consider the expression  num1% num2 which finds the remainder after dividing num1 by num2 | "%" is used as modulo operator where 11%2 is evaluated as 1, 9%11 is evaluated as 9 |

# Relational operators:

| **Operator** | **Explanation** | **Example** |
| --- | --- | --- |
| == | Used for checking the equality of two values/variables | 10 == 10 is evaluated as True  10 == 100 is evaluated as False |
| != | Used for checking the inequality of two values/variables | 10 != 10 is evaluated as False  10 != 100 is evaluated as True |
| > | Used for checking if num1 is greater than num2 and is represented as num1 > num2 | 10 > 10 is evaluated as False  100 > 10 is evaluated as True |
| < | Used for checking if num1 is lesser than num2 and is represented as num1 < num2 | 10 < 10 is evaluated as False  10 < 100 is evaluated as True |
| >= | Used for checking if num1 is greater than or equal to num2 and is represented as num1 >= num2 | 10 >= 10 is evaluated as True  10 >= 100 is evaluated as True |
| <= | Used for checking if num1 is lesser than or equal to num2 and is represented as num1 <= num2 | 10 <= 10 is evaluated as True  100 <= 10 is evaluated as False |

# 

# Assignment operators:

| **Operator** | **Explanation** | **Example** |
| --- | --- | --- |
| = | Used for assigning a value to a variable | num=5  Here num is assigned the value 5 |
| += | Used as short hand assignment operator for addition | num=num+1 can also be represented as num+=1 |
| -= | Used as short hand assignment operator for subtraction | num=num-1 can also be represented as num-=1 |
| \*= | Used as short hand assignment operator for multiplication | num=num\*1 can also be represented as num\*=1 |
| /= | Used as short hand assignment operator for division | num=num/1 can also be represented as num/=1 |
| %= | Used as short hand assignment operator for modulo operation | num=num%1 can also be represented as num%=1 |

# Logical Operators:

These operators are used to combine one or more relational expressions.

| **Operators** | **Description** |
| --- | --- |
| AND | Result will be true, if both the expressions are true. If any one or both the expressions are false, the result will be false |
| OR | Result will be true, even if one of the expression is true. If both the expressions are false, the result will be false |
| NOT | If the expression is true, result will be false and vice versa |

If A and B are two relational expressions, say A = (Num1>2000), B= (Num2>100), the result of combining A and B using logical operator is based on the result of A and B as shown below:

| **A** | **B** | **A and B** |
| --- | --- | --- |
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

| **A** | **B** | **A or B** |
| --- | --- | --- |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

| **A** | **not A** |
| --- | --- |
| True | False |
| False | True |

**Implicit and Explicit type conversions:**

When we perform any operation on variables of different datatypes, the data of one variable will be converted to a higher datatype among the two variables and the operation is completed. This conversion is done by interpreter automatically and it is known as ***implicit type conversion***.

But Python does not support implicit type conversion and it will throw an error.

**Example:**

1. num1=10
2. num2="20"
3. result=num1+num2
4. print(result)

Output:

 result=num1+num2  
TypeError: unsupported operand type(s) for +: 'int' and 'str'

If we have to avoid this, then we have to explicitly convert the datatype of one variable into the required datatype to complete the operation. This is known as **explicit type conversion.**

**Example:**

1. num1=10
2. num2="20"
3. result=num1+int(num2)
4. print(result)

**Output:**

30

**Note:**

Programming languages define their own rules for implicit and explicit conversions. These rules do change from language to language.

Similarly, one has to be careful in explicit conversions as well. For example,

* Converting a floating point value to integer would result in loss of decimal point values
* A larger data type if converted to smaller data type will result in loss of data as the number will be truncated

***Comments are the lines which are skipped during execution of a program.***

There are two types of comments available in Python:

* First one is **single line comment** which starts with ‘#’ symbol and extends till the end of line. Comments can start from the beginning of the line and middle of the line, but it should not be a part of string literal.
* Second one is **multi line comment** which starts with ''' or """ and ends with ''' or """ respectively. This type of comment is mainly used for documentation purpose.

**Example:**

1. '''
2. used for: Demonstrating comments
3. This is the first way of using multi-line comment
4. '''
5. """
6. used for: Demonstrating comments
7. second way of using multi-line comment
8. """
9. *#program to demonstrate explicit type conversion*
10. num1=10 *#variable of integer type*
11. num2="20" *#variable of string type*
12. result=num1+int(num2) *#using explicit conversion*
13. print(result)

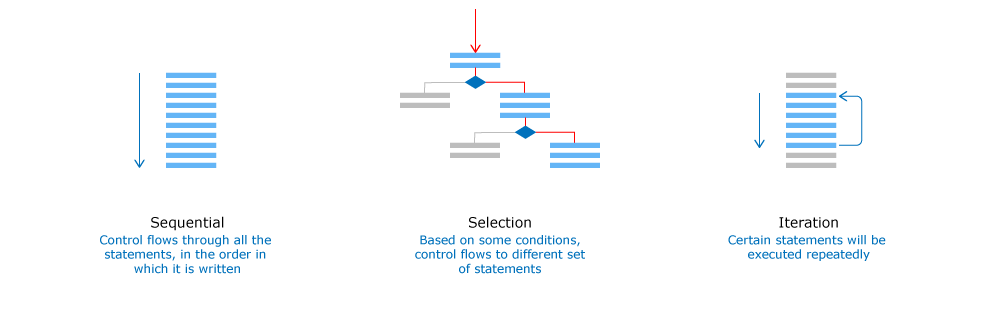
**Output:**

30

# Control structures:

Control Structures are used to control the flow of execution in a Python program. Sometimes, it is required to skip execution of few statements based on the logic and in some cases, a set of statements may be needed to get executed repeatedly for a finite number of times or based on the value of the variable.

Following are the commonly used control structures in Python:

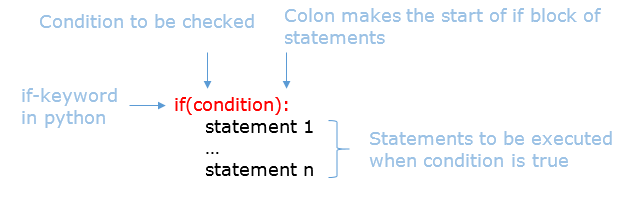


During the execution of the program, we may not wish to execute all set of statements sequentially. Sometimes, we may wish to select between various set of statements based on some conditions. Depending on the test condition evaluation, the flow is determined within the program. Let’s have look at few of them.

# if (simple if):

It is a conditional statement used for decision making in Python. In **if** statement, the test condition is evaluated and the statements inside the **if** block are executed only if the evaluated condition is True. In the below **if** statement, we have only one set of statements to select based on the test condition. So, it is also called as**one-way selection** statement.

Below is the syntax of the simple **if** statement:



**Example:**

1. a=10
2. if(a>0):
3. print("positive integer")

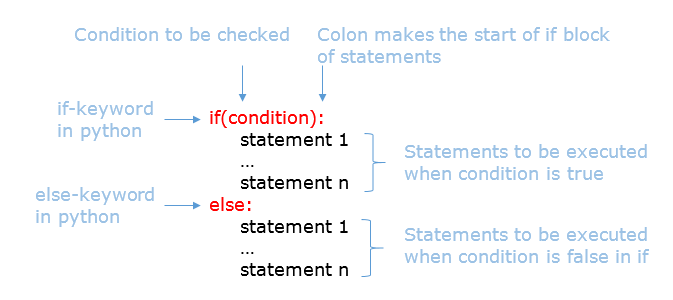
**Output:**

positive integer

# if-else:

It is a conditional statement used for selection between two set of statements based on the evaluation of test condition. The statements inside the **if** block are executed only if the evaluated condition is True. Otherwise statements inside the **else** block are executed. As we have two set of statements to select based on the test condition, it is also called as **two-way selection**statement.

Below is the syntax of if-else statement:



**Example:**

1. a=-10
2. if(a>0):
3. print("positive integer")
4. else:
5. print("Not a positive integer")

**Output:**

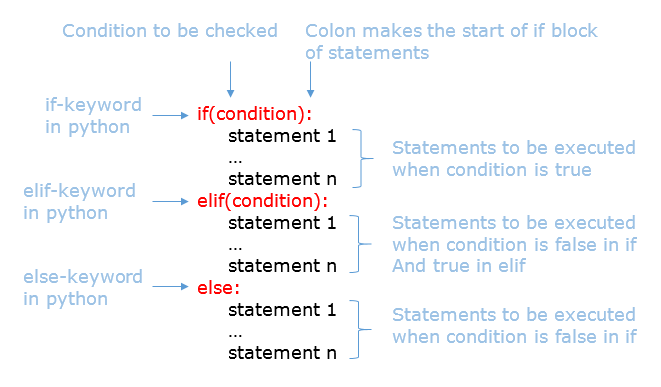
Not a positive integer

# else-if ladder:

It is a conditional statement used for selection between multiple set of statements based on multiple test conditions. The various test conditions are provided inside each **if-elif** statement. Whenever the test condition is evaluated as True, the statements inside the corresponding block are executed and the control comes out of the else-if ladder. If none of the test conditions are evaluated as True, the statements inside the **else** block are executed. As we have multiple set of statements to select based on the test conditions, it is also called as **multi-way selection** statement.

In **else-if** ladder, the conditions are evaluated from the top of the ladder downwards. As soon as a True condition is found, the set of statements associated with it get executed skipping the rest of the ladder.

Below is the syntax of else-if ladder statement:



**Example:**

1. a=0
2. if(a>0):
3. print("positive integer")
4. elif(a<0):
5. print("negative integer")
6. else:
7. print("it’s zero")

**Output:**

it’s zero

# Nested if:

An **if** statement within another **if** statement is known as **nested if** statement. Similarly, any decision logic can be written within an **else** statement too.

Have a look at the below example of nested if:

1. num1=10
2. num2=20
3. num3=30
4. if(num1>num2):
5. if(num1>num3):
6. print("num1 is greater")
7. else:
8. print("num3 is greater")
9. elif(num2>num3):
10. print("num2 is greater")
11. else:
12. print("num3 is greater")

**Output:**

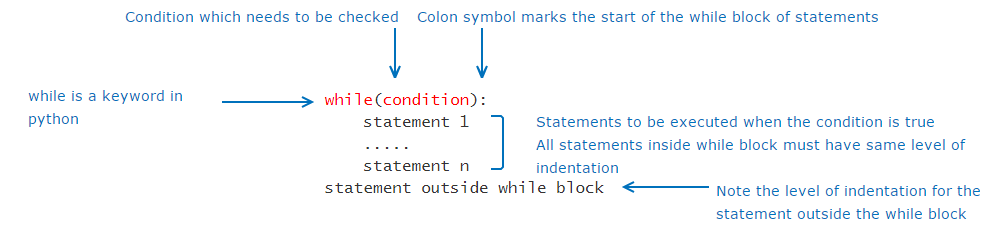
num3 is greater

***Looping statements*** are used to execute the same block of code multiple times in Python based on the test condition. Let’s have look some of the looping statements in Python.

# while loop:

        The **while** loop is used to execute a piece of code as long as the test condition is True. The **while** loop is preferred whenever the number of iterations is not known.

**Syntax:**



**Example:**

1. num=5
2. count=1
3. while count <= num:
4. print("the current number is:",count)
5. count+=1

**Output:**

the current number is: 1

the current number is: 2

the current number is: 3

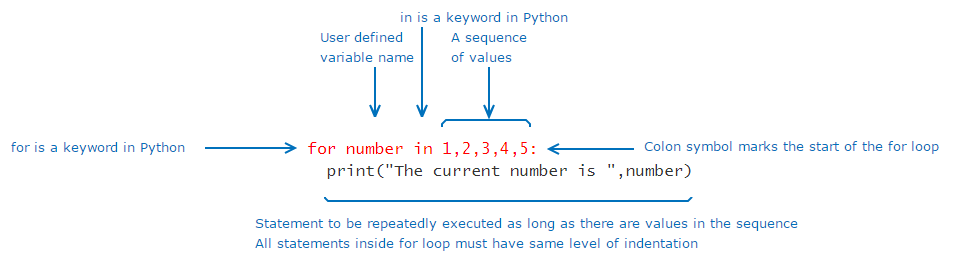
the current number is: 4

the current number is: 5

# for loop:

In Python, the **for** loop allows the loop to run over a specific sequence of values. In other words, for every value in the sequence, the loop runs once. Thus, we can avoid infinite loops by using a **for** loop.

**Syntax**:



**Example:**

1. for number in 1,2,3,4,5:
2. print("The current number is ",number)

**Output:**

The current number is  1

The current number is 2

The current number is 3

The current number is 4  
The current number is  5

***Another variation!***

In Python, there is an easy way to achieve this by using range(x, y, step). It creates a sequence from x to y-1 with a difference of step between each value.

**Example:**

1. start=1
2. end=10
3. step=2
4. for number in range (start, end, step):
5. print("The current number is ", number)

**Output:**

The current number is  1

The current number is 3

The current number is 5

The current number is 7  
The current number is  9

# Nested loop:

A loop within a loop is known as a nested loop.

Assume that there are 5 passengers and each of them have 2 baggages. The below code will make sure that all baggages of each passenger have undergone the security check.

**Example:**

1. number\_of\_passengers=5
2. number\_of\_baggage=2
3. security\_check=True
4. for passenger\_count in range(1, number\_of\_passengers+1):
5. for baggage\_count in range(1,number\_of\_baggage+1):
6. if(security\_check==True):
7. print("Security check of passenger:", passenger\_count, "-- baggage:", baggage\_count,"baggage cleared")
8. else:
9. print("Security check of passenger:", passenger\_count, "-- baggage:", baggage\_count,"baggage not cleared")

Output:

Security check of passenger: 1 -- baggage: 1 baggage cleared  
Security check of passenger: 1 -- baggage: 2 baggage cleared  
Security check of passenger: 2 -- baggage: 1 baggage cleared  
Security check of passenger: 2 -- baggage: 2 baggage cleared  
Security check of passenger: 3 -- baggage: 1 baggage cleared  
Security check of passenger: 3 -- baggage: 2 baggage cleared  
Security check of passenger: 4 -- baggage: 1 baggage cleared  
Security check of passenger: 4 -- baggage: 2 baggage cleared  
Security check of passenger: 5 -- baggage: 1 baggage cleared  
Security check of passenger: 5 -- baggage: 2 baggage cleared

The same code in the inner loop can also be written using the **while** loop instead of the **for**loop as shown below:

number\_of\_passengers=5

1. number\_of\_baggage=2
2. security\_check=True
3. for passenger\_count in range(1, number\_of\_passengers+1):
4. baggage\_count =1
5. while (baggage\_count<=number\_of\_baggage):
6. if(security\_check==True):
7. print("Security check of passenger:", passenger\_count, "-- baggage:", baggage\_count,"baggage cleared")
8. else:
9. print("Security check of passenger:", passenger\_count, "-- baggage:", baggage\_count,"baggage not cleared")
10. baggage\_count+=1

Similarly, the outer loop can also be written using while loop.

# Looping Control Statements:

The flow inside looping statements are controlled using the looping control statements like **pass**, **break** and **continue**.

When we want to stop a loop or break out of it, we can use the **break** statement.

When we want to skip the remaining statements in the current loop and continuewith the next iteration, we can use **continue** statement.

**Example:**

Go through the below code, Assume A – Adult passenger, C- Child, FC – Flight Captain, FA – Flight Attendant, SP – Suspicious passenger.   
Also, assume the following conditions:

* Flight captains and attendants do not require to check-in
* In case suspicious passengers are found, need to declare an emergency at the airport
* For other passengers such as adults and children, need to proceed with normal security check

1. for passenger in "A","A", "FC", "C", "FA", "SP", "A", "A":
2. if(passenger=="FC" or passenger=="FA"):
3. print("No check required")
4. continue
5. if(passenger=="SP"):
6. print("Declare emergency in the airport")
7. break
8. if(passenger=="A" or passenger=="C"):
9. print("Proceed with normal security check")
10. print("Check the person")
11. print("Check for cabin baggage")

**Output:**

Proceed with normal security check  
Check the person  
Check for cabin baggage  
Proceed with normal security check  
Check the person  
Check for cabin baggage  
No check required  
Proceed with normal security check  
Check the person  
Check for cabin baggage  
No check required  
Declare emergency in the airport

In Python,**pass** is a null statement which is used to do create empty blocks. When **pass** is executed, it results in no-operation and the control will move to the next statement applicable. Below example shows how **pass** can be used to create an empty **if** block.

**Example:**

1. num=10
2. count=0
3. while(count <= num):
4. if(count%2 == 0):
5. pass
6. else:
7. print(count)
8. count+=1

**Output:**

1  
3  
5  
7  
9

# Functions in Python:

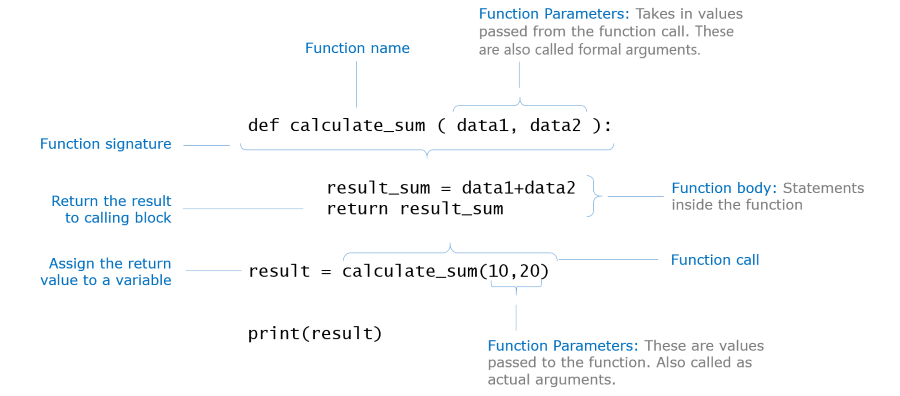
Functions are set of instructions to perform a specific task. Below is the syntax of functions in Python.

**Syntax:**

def function\_name([arg1,...,argn]):   
    #statements   
    [return value]   
variable\_name = function\_name([val1,...,valn])

**Note:** Anything enclosed in [ ] (square bracket) is optional.

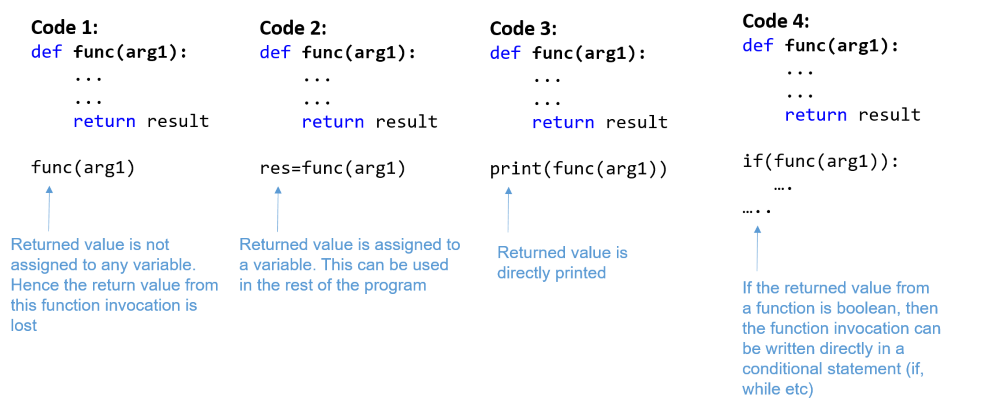
**Example:**



* Function call
* Actual arguments being copied to formal arguments
* Execution of function body
* Return from function

1. observe1="What's happening!!"
2. def passport\_check(passport\_no):
3. observe4="actual copied to formal"
4. observe5="func. execution starts"
5. if(len(passport\_no)==8):
6. if(passport\_no[0]>="A" and passport\_no[0]<="Z"):
7. status="valid"
8. else:
9. status="invalid"
10. else:
11. status= "invalid"
12. observe6="func. execution ends"
13. return status
14. observe2="function with formal arg."
15. observe3="calling with actual arg."
16. passport\_status=passport\_check("M9993471")
17. print("Passport is",passport\_status)
18. *#observe1,2,3,4,5,6 are temporary variables used to explain this concept*

Let's see how we can use values returned from a function.

ar

**Arguments Passing in Python:**

In programming, there are two ways in which arguments can be passed to functions:

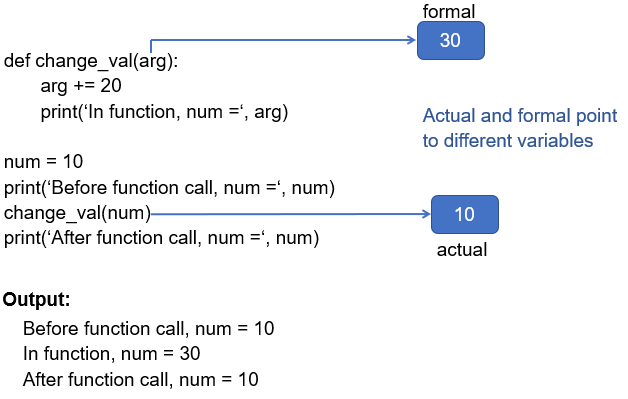
* pass by value
* pass by reference

Some languages use pass by value by default while others use pass by reference. Some languages support both and allow you to choose.

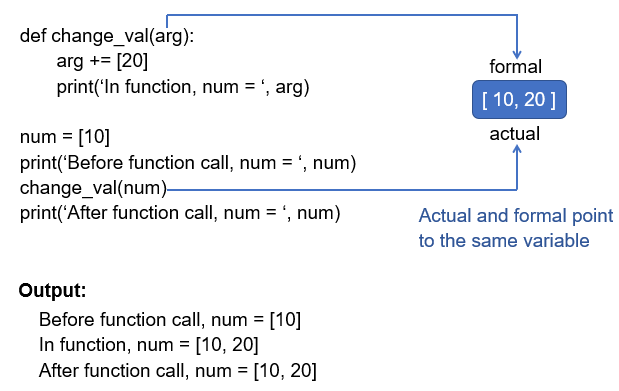
In Python, we don't have to think about pass by value and pass by reference as it does that automatically for you. To emulate this using Python, we use the concept of mutability. If the argument passed is immutable then it follows pass by value, else if the argument passed is mutable then it follows pass by reference.

**Note:**Till now we have seen int, float, string data types which are immutable. Mutable data types will be discussed later in this course.

**Pass by value method:**

****

**Pass by reference method:**

****

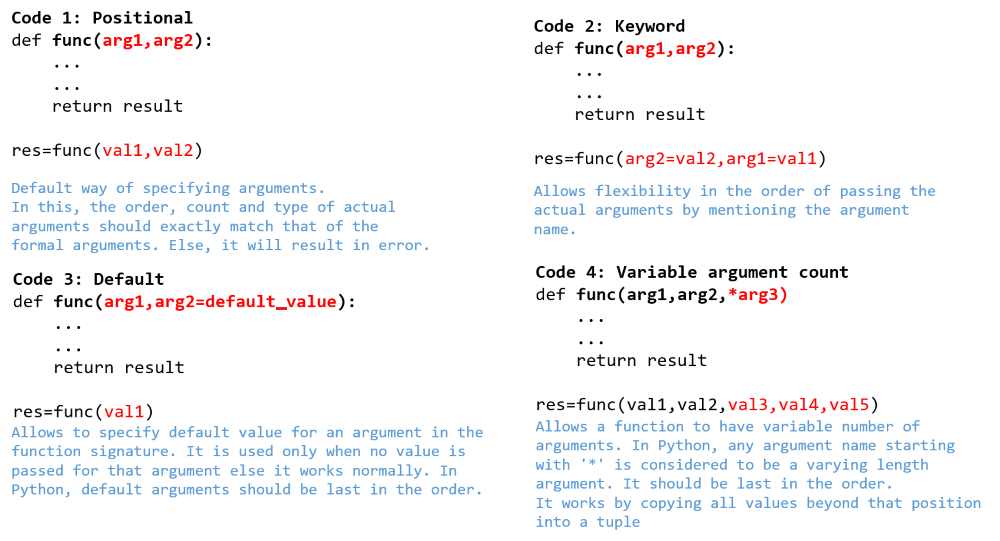
**Example:**

1. def change\_number(num):
2. num+=10
3. def change\_list(num\_list):
4. num\_list.append(20)
5. num\_val=10
6. print("\*\*\*\*\*\*\*\*\*effect of pass by value\*\*\*\*\*\*\*\*\*")
7. print("num\_val before function call:", num\_val)
8. change\_number(num\_val)
9. print("num\_val after function call:", num\_val)
10. print("-----------------------------------------------")
11. val\_list=[5,10,15]
12. print("\*\*\*\*\*\*\*\*\*effect of pass by reference\*\*\*\*\*\*\*\*\*")
13. print("val\_list before function call:", val\_list)
14. change\_list(val\_list)
15. print("val\_list after function call:", val\_list)

**Types of Arguments :**

Programming languages allow controlling the ordering and default values of arguments.

In Python, we will observe the following:



Example:

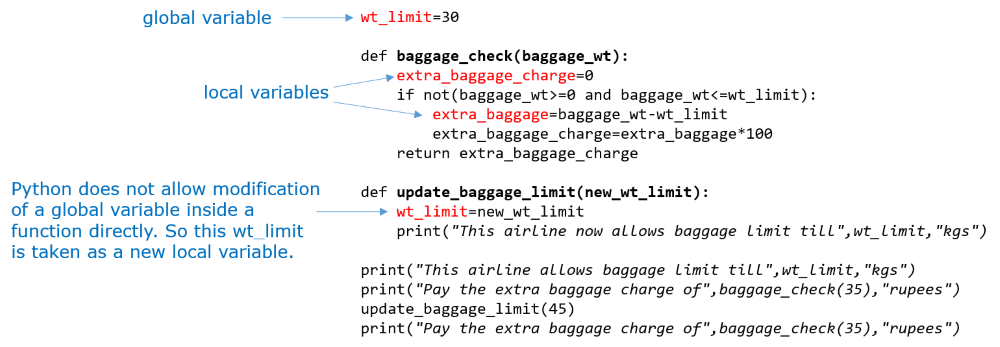
1. def display1(flight\_number, seating\_capacity):
2. print("Flight Number:", flight\_number)
3. print("Seating Capacity:", seating\_capacity)
4. print("code-1: positional arguments")
5. display1("FN789",200)
6. *#Uncomment and execute the below function call statement and observe the output*
7. *#display1(300,"FN123")*
8. def display2(flight\_number, seating\_capacity):
9. print("Flight Number:", flight\_number)
10. print("Seating Capacity:", seating\_capacity)
11. print("-------------------------------------------------")
12. print("code-2: keyword arguments")
13. display2(seating\_capacity=250, flight\_number="FN789")
14. def display3(flight\_number, flight\_make="Boeing", seating\_capacity=150):
15. print("Flight Number:", flight\_number)
16. print("Flight Make:", flight\_make)
17. print("Seating Capacity:", seating\_capacity)
18. print("-------------------------------------------------")
19. print("code-3: default arguments")
20. display3("FN789","Eagle")
21. *#Uncomment and execute the below function call statements one by one and observe the output*
22. *#display3("FN234")*
23. *#display3("FN678","Qantas",200)*
24. def display4(passenger\_name, \*baggage\_tuple):
25. print("Passenger name:",passenger\_name)
26. total\_wt=0
27. for baggage\_wt in baggage\_tuple:
28. total\_wt+=baggage\_wt
29. print("Total baggage weight in kg:", total\_wt)
30. print("-------------------------------------------------")
31. print("code-4: variable argument count")
32. display4("Jack",12,8,5)
33. *#Uncomment and execute the below function call statements one by one and observe the output*
34. *#display4("Chan",20,12)*
35. *#display4("Henry",23)*

**Variable and its scope:**

The below code has been written to represent the baggage weight check process based on the weight limit specified by an airline.

Go through the below code and guess the output.

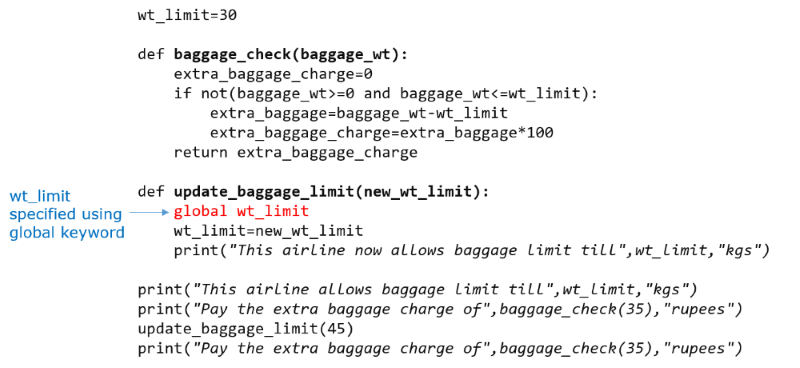
1. wt\_limit=30
2. def baggage\_check(baggage\_wt):
3. extra\_baggage\_charge=0
4. if not(baggage\_wt>=0 and baggage\_wt<=wt\_limit):
5. extra\_baggage=baggage\_wt-wt\_limit
6. extra\_baggage\_charge=extra\_baggage\*100
7. return extra\_baggage\_charge
8. def update\_baggage\_limit(new\_wt\_limit):
9. wt\_limit=new\_wt\_limit
10. print("This airline now allows baggage limit till",wt\_limit,"kgs")
11. print("This airline allows baggage limit till",wt\_limit,"kgs")
12. print("Pay the extra baggage charge of",baggage\_check(35),"rupees")
13. update\_baggage\_limit(45)
14. print("Pay the extra baggage charge of",baggage\_check(35),"rupees")

Let us go through the code now.  


Variables **extra\_baggage** and **extra\_baggage\_charge** are created inside the function **baggage\_check**(). Hence they are local to that function or in other words, they are **local variables**. They are created when the owning function starts execution and remains in memory till the function finishes execution. They can be accessed only inside that function.

**wt\_limit** is created outside the functions. Hence it is a **global variable**. Global variables are created when the program execution starts and remains in memory till the program terminates. They can be read anywhere in the program - within a function or outside. But they are protected from modification inside a function. As it is available throughout the program, use of global variable should be restricted to avoid accidental misuse by developers and to minimize memory usage.

In cases where a global variable needs to be modified inside a function, like in function **update\_baggage\_limit**(), Python allows you to do that using the **global** keyword.



# Collections in Python:

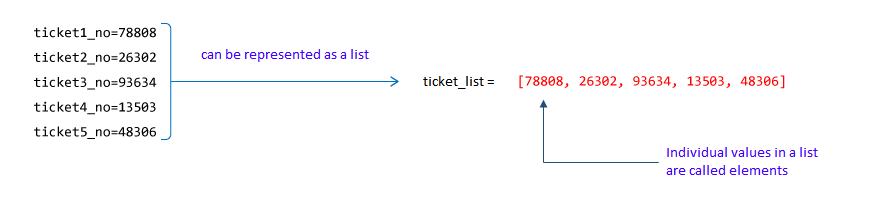
When we want to treat some data as a group, it would not be good to create individual variables for each data. We can store them together as a collection.

There are many collection data types which are supported by Python.

* List
* Tuple
* String
* Set
* Dictionary

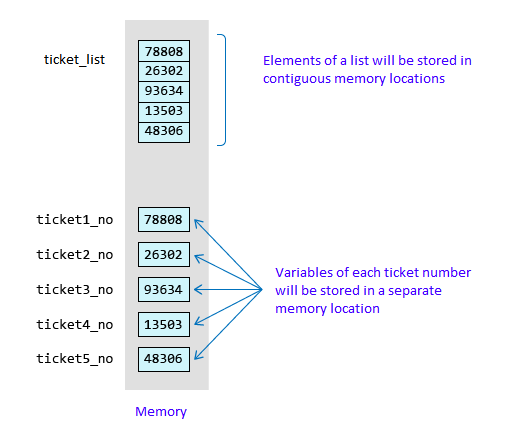
**List – Introduction:**

Let’s start by exploring one of the most common collection data type – list. **List**can be used to store a group of elements together in a sequence.  
  
If we want to store the ticket numbers allocated to each passenger traveling in a flight, instead of using separate variables for each ticket number, we can use a list as shown below.



**List – Storage of Elements:**

In case of having different variables for each ticket number, variables will be stored in separate memory locations. Whereas in case of list, the elements will be stored in contiguous memory locations as illustrated below.



**List – Indexing:**

Each element in the list has a position in the list known as an **index**. The list index starts from **zero**. It’s like having seat numbers starting from 0!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Element** | 78808 | 26302 | 93634 | 13503 | 48306 |
| **Index** | 0 | 1 | 2 | 3 | 4 |

If seat number 2 is allocated to the passenger with ticket number 93634, the passenger can directly go to seat number 2 without having to go through other seats. Similarly, index positions actually help us to **directly access** a value from the list.

**list\_name[index]** can be used to directly access the list element at the mentioned index position.  
  
If we want to allocate a different passenger to seat number 3, we can do it as ticket\_list[3]=13504. Thus, in addition to using the index to access an element directly, we can also use it to **directly modify**an element in the list.  
  
Just like how we cannot allocate 101st seat to a passenger in a 100 seat plane, we cannot access values beyond the total number of elements in the list.  
  
For example: *print(ticket\_list[5])* will result in **index out of bound error**.

 Let’s see how to create a list in Python and perform some operations on it.

**Creating a list:**

|  |  |  |
| --- | --- | --- |
| Creating an empty list | sample\_list=[] |  |
| Creating a list with known size and known elements | sample\_list1=["Mark",5,"Jack",9, "Chan",5]  sample\_list2=["Mark","Jack", "Chan"] | List can store both homogeneous and heterogeneous elements |
| Creating a list with known size and unknown elements | sample\_list=[None]\*5 | None denotes an unknown value in Python |
| Length of the list | len(sample\_list) | Displays the number of elements in the list |

**Random access of elements:**

|  |  |  |
| --- | --- | --- |
| Random read | print(sample\_list[2]) |  |
| Random write | sample\_list[2]=“James” | List is mutable i.e., the above statement will rewrite the existing  value at index position 2 with “James”. |

**Other Operations:**

|  |  |  |
| --- | --- | --- |
| Adding an element to the end of the list | sample\_list.append("James") | List need not have a fixed size, it can grow dynamically |
| Concatenating two lists | new\_list=["Henry","Tim"]  sample\_list+=new\_list sample\_list=sample\_list+new\_list | sample\_list+=new\_list, concatenates new\_list to sample\_list  sample\_list=sample\_list+new\_list, creates a new list named sample\_list containing the concatenated elements from the original sample\_list and new\_list   Observe this difference while visualizing |

As elements are stored sequentially in a list, we can use a loop to access these elements.

1. list\_of\_airlines=["AL1","AL2","AL3"]
2. print("Iterating the list using range()")
3. for index in range(0,len(list\_of\_airlines)):
4. print(list\_of\_airlines[index])
5. print("Iterating the list using keyword in")
6. for airline in list\_of\_airlines:
7. print(airline)

If we just want to find out whether an element is there in a list or not, we need not iterate through the list. Instead we can check using **if..in** syntax. Try out the code and observe the results.

1. list\_of\_airlines=["AL1","AL2","AL3"]
2. airline="AL3"
3. if airline in list\_of\_airlines:
4. print("Airline found")
5. else:
6. print("Airline not found")

**List -Slicing:**

Assume that the names of the airlines operating from an airport are stored in a list as shown below.  
Suppose, we need to extract the airlines from the 1st to the 3rd index positions in the list. How can we do that?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **List of Airlines** | AL1 | AL2 | AL3 | AL4 | AL5 |
| **Index** | 0 | 1 | 2 | 3 | 4 |
| **Negative Index** | -5 | -4 | -3 | -2 | -1 |

Python offers a simple solution in the form of slicing:

sub\_list=list\_of\_airlines[1:4]

The above line provides a sub-list from index position 1 to 3. That is, from 1 to (4-1).   
  
**For example:**To fetch the second last airline in the list, we can write list\_of\_airlines[-2]. This is equivalent to list\_of\_airlines[len(list\_of\_airlines)-2].   
  
Indices may also be considered negative as shown above. This is normally used to count from right. Negative indices can also be used for slicing.

**For example:** list\_of\_airlines[-4:-1] will give us the same output as list\_of\_airlines[1:4]

**Collections - tuple:**

Suppose it is mandatory to have the following types of food in the lunch menu of the passengers.

Welcome Drink, Veg Starter, Non-Veg Starter, Veg Main Course, Non-Veg Main Course, Dessert

How can we store it such that no one can modify the elements?

Of course, we can use a list but anybody can modify an element in the list. This is where we can use another collection data type known as **tuple**.

Like list, tuple can also store a sequence of elements but the value of the individual elements cannot be changed. (i.e. tuples are **IMMUTABLE**). Elements can be homogeneous or heterogeneous but they are **READ-ONLY**.

|  |  |  |
| --- | --- | --- |
| **Creating a tuple** | lunch\_menu=("Welcome Drink","Veg Starter","Non-Veg Starter","Veg Main Course","Non-Veg Main Course","Dessert") | () are optional, a set of values separated by comma is also considered to be a tuple.  sample\_tuple="A","B","C" Although () are optional, it is a good practice to have them for readability of code.  If we need to create a tuple with a single element, then we need to include a comma as shown below: sample\_tuple=("A",) |
| **Random Write** | lunch\_menu[0]="" | This will result in an error as tuple is immutable. Hence random write is not possible in tuple. |

All the remaining operations are similar to lists.

**Collections - Strings:**

In a program, not all values are numerical. We also have alphabetical or alpha numerical values. Such values are called strings.

**Example:**"Hello World", "AABGT6715H"

Each value in a string is called a **character**. Just like list elements, we can access the characters in a string based on its index position.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **String** | "AABGT6715H" | | | | | | | | | |
| **Character** | A | A | B | G | T | 6 | 7 | 1 | 5 | H |
| **index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

In Python, string is a data type and anything enclosed in a single quote or double quote is considered to be a string. All the remaining operations are similar to lists. But like tuple, strings are also **IMMUTABLE**.

Example:

1. *#Creating a string*
2. pancard\_number="AABGT6715H"
3. *#Length of the string*
4. print("Length of the PAN card number:", len(pancard\_number))
5. *#Concatenating two strings*
6. name1 ="PAN "
7. name2="card"
8. name=name1+name2
9. print(name)
10. print("Iterating the string using range()")
11. for index in range(0,len(pancard\_number)):
12. print(pancard\_number[index])
14. print("Iterating the string using keyword in")
15. for value in pancard\_number:
16. print(value)
17. print("Searching for a character in string")
18. if "Z" in pancard\_number:
19. print("Character present")
20. else:
21. print("Character is not present")
22. *#Slicing a string*
23. print("The numbers in the PAN card number:", pancard\_number[5:9])
24. print("Last but one 3 characters in the PAN card:",pancard\_number[-4:-1])
25. pancard\_number[2]="A" *#This line will result in an error, i.e., string is immutable*
26. print(pancard\_number)

**Collections – Set:**

A set is an unordered group of values with no duplicate entries. Set can be created by using the keyword **set** or by using curly braces **{}**. The **set** function is used to eliminate duplicate values in a list.

|  |  |  |
| --- | --- | --- |
| Creating a set | flight\_set={500,520,600,345,520,634,600,500,200,200} | Removes the duplicates from the given group of values to create the set. |
| Eliminating duplicates from a list | passengers\_list=["George", "Annie", "Jack", "Annie", "Henry", "Helen", "Maria", "George", "Jack", "Remo"] unique\_passengers=set(passengers\_list) | set function - removes the duplicates from the list and returns a set |
| Common elements between set A and set B | set A & set B | Creates a new set which has common elements from set A and set B |
| Elements that are only in set A | set A - set B | Creates a new set which has only unique elements of set A |
| Merges elements of set A and set B | set A | set B | Creates a new set which has all the elements |

**Collections – Dictionary:**

A dictionary can be used to store an unordered collection of key-value pairs. The key should be unique and can be of any data type. Like lists, dictionaries are mutable.

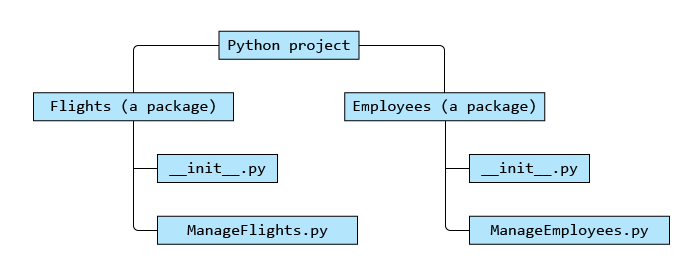
|  |  |  |
| --- | --- | --- |
| Creating a dictionary | crew\_details= { "Pilot":"Kumar", "Co-pilot":"Raghav", "Head-Strewardess":"Malini", "Stewardess":"Mala" } | First element in every pair is the key and the second element is the value. |
| Accessing the value using key | crew\_details["Pilot"] | This will return the corresponding value for the specified key |
| Iterating through the dictionary | for key,value in crew\_details.items():      print(key,":",value) | items() function gives both key and value, which can be used in a for loop. |

# Libraries and built functions in Python:

**Code organization:**

In Python, code organization and reusability is managed through modules and packages. A module is a Python file with a .py extension. A package is a folder which contains a file whose name is \_\_init\_\_.py. Package can store many Python modules/files in it. The import  statement is used to load a library module into the program's memory.

Sample Python project structure:



**Packages:**

Let's assume that the **ManageFlights.py** inside Flights package has the below code:

1. airline="Smart Airlines"
2. def add(no\_of\_flights):
3. print(no\_of\_flights," flights added to the fleet")

If we need to access the function **add()**, inside the ManageFlights module in some other module, then we can import the ManageFlights module and use it.

Import can be done in two ways:

**Method 1:**

1. from Flights import ManageFlights *#from packagename import modulename*
2. ManageFlights.add(10)

**Method 2:**

1. import Flights.ManageFlights *#import packagename.modulename*
2. Flights.ManageFlights.add(10)

**Packages Naming Conflict:**

Consider a scenario where two modules have the same name which are present in different packages and both of them have the same function ‘add’.

Flights -> Manage.py -> add()

Employees -> Manage.py -> add()

To avoid naming conflicts during import, we can use one of the below techniques:

**(i)**

1. import Flights.Manage
2. import Employees.Manage
3. Flights.Manage.add()
4. Employees.Manage.add()

**(ii)**

1. from Flights import Manage as FManage
2. from Employees import Manage as EManage
3. FManage.add()
4. EManage.add()

**(iii)**

1. from Flights.Manage import add as add1
2. from Employees.Manage import add as add2
3. add1()
4. add2()

**Buitl-in Modules – random and Math:**

**The Random Module:**

Python has many inbuilt packages and modules. One of the most useful modules is **random**. This module helps in generating random numbers.

The code given below generates a random number between x and y-1 (both inclusive) using the **randrange** function of the random module.

Try out the below code and observe the output.

1. import random
2. x=10
3. y=50
4. print(random.randrange(x,y))

**Sample output:**

33

46

**The Math Module:**

**math** is another useful module in Python. Once you have imported the math module, you can use some of the below functions:

| **Function** | **Explanation** |
| --- | --- |
| math.ceil(x) | Smallest integer greater than or equal to x |
| math.floor(x) | Largest integer smaller than or equal to x |
| math.factorial(x) | Factorial of x |
| math.fabs(x) | Gives absolute value of x |

Try out  the below code and observe the output.

1. import math
2. num1=234.01
3. num2=6
4. num3=-27.01
5. print("The smallest integer greater than or equal to num1,",num1,":",math.ceil(num1))
6. print("The largest integer smaller than or equal to num1,",num1,":",math.floor(num1))
7. print("The factorial of num2,",num2,":", math.factorial(num2))
8. print("The absolute value of num3",num3,":",math.fabs(num3))

**String Functions:**

String data type in Python has many inbuilt functions which makes it easier to work with strings.

Consider the string, name="Raghav".

| **Function** | **Output** | **Explanation** |
| --- | --- | --- |
| name.count("a") | 2 | Returns the count of a given set of characters. Returns 0 if not found |
| name.replace("a","A") | RAghAv | Returns a new string by replacing a set of characters with another set of characters. It does not modify the original string |
| name.find("a") | 1 | Returns the first index position of a given set of characters |
| name.startswith("Ra") | True | Checks if a string starts with a specific set of characters, returns true or false accordingly. |
| name.endswith("ha") | False | Checks if a string ends with a specific set of characters, returns true or false accordingly. |
| name.isdigit() | False | Checks if all the characters in the string are numbers, returns true or false accordingly. |
| name.upper() | RAGHAV | Converts the lowercase letters in string to uppercase |
| name.lower() | raghav | Converts the uppercase letters in string to lowercase |
| name.split("a") | ['R', 'gh', 'v'] | Splits string according to delimiter and returns the list of substring. Space is considered as the default delimiter. |

Example:

1. boarding\_call="Good Evening, this is the final call to AL passengers for the flight AL 466 which is planned to take off at 8.40A.M."
2. if(boarding\_call.startswith("Good Evening")):
3. print(boarding\_call.replace("Good Evening","Good Morning"))
4. if(boarding\_call.find("AL"))>=0:
5. print("Welcome to Air Lines.")
6. if(boarding\_call.endswith("A.M.")):
7. print("Passengers are requested to have their breakfast.")
8. a=boarding\_call.split(" ")
9. for i in a:
10. if(i.isdigit()):
11. print("Flight Number is specified to the passengers.")
12. print("Total number of times flight service name is specified in the boarding call:",boarding\_call.count("AL"))
13. message="Thank you all..Have a nice journey!"
14. print(message.upper())
15. print(message.lower())

**List Functions:**

List data type in Python has many inbuilt functions.  
Consider a list, num\_list=[10,20,30,40,50]

| **Function** | **Output** | **Explanation** |
| --- | --- | --- |
| num\_list.append(60) | [10,20,30,40,50,60] | Adds an element to end of list |
| num\_list.index(10) | 0 | Returns the index position of the element. In case of multiple occurrence of the element, returns the index of the first occurrence. Throws ValueError, if the element is not found |
| num\_list.insert(3,60) | [10,20,30,60,40,50] | Inserts an element at a given position |
| num\_list.pop(3) | 40 | Removes and returns the element at given index position from the list |
| num\_list.remove(30) | [10,20,40,50] | Removes the first occurring element whose value is 30 |
| num\_list.sort() | [10,20,30,40,50] | Sorts the list in ascending order |
| num\_list.reverse() | [50,40,30,20,10] | Reverses the list |

Example:

1. crew\_details={
2. "Pilot":"Kumar",
3. "Co-pilot":"Raghav",
4. "Head-Strewardess":"Malini",
5. "Stewardess":"Mala"
6. }
7. print("Before update:")
8. print("Co-pilot:",crew\_details.get("Co-pilot"))
9. crew\_details.update({"Flight Attendant":"Jane", "Co-pilot":"Henry"})
10. print("\nAfter update:")
11. print("Co-pilot:",crew\_details.get("Co-pilot"))
12. print("Flight Attendant:",crew\_details["Flight Attendant"])

**Dictionary Functions:**

Dictionary in Python has many inbuilt functions.

Consider a dictionary:

1. crew\_details={
2. "Pilot":"Kumar",
3. "Co-pilot":"Raghav",
4. "Head-Strewardess":"Malini",
5. "Stewardess":"Mala"
6. }

| **Function** | **Output** | **Explanation** |
| --- | --- | --- |
| crew\_details.get("Pilot") | Kumar | Returns the value for given key. If the given key is not found, returns None |
| crew\_details.update({"Flight Attendant":"Jane", "Co-pilot":"Henry"}) | No output, dictionary will be updated | Updates the dictionary with the given key-value pairs. If a key-value pair is already existing, it will be overwritten, otherwise it will be added to the dictionary |

**Regular Excpressions:**

Many times a lot of data would be stored in files and we may have to pick and change only relevant portions from a file. Even though there are string functions which allow us to manipulate strings, when dealing with more complicated requirements, we would need more powerful tools.

Regular Expressions are used to check and extract relevant portions of a string based on a pattern and modify if required.   
Python has a module named 're' for regular expressions.

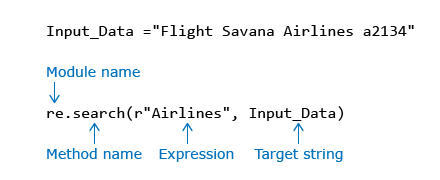
**Working with re module:**

Two commonly used methods in the **re** module are **search** and **sub**. Search is used to find a pattern and sub is used to perform a substitution.

input\_data = “Flight Savana Airlines a2134”

The requirement is to search for the pattern "Airlines" in the input\_data. Let’s understand how this

canbe achieved using re methods.



Here, **r** in front of the search pattern indicates 'raw string' where the special characters are treated as normal characters.  
The output will be 'None' if the pattern is not found.

If you want to search the pattern "Air" in the given string "Airline" here are few example for that.

**Example 1:**

1. import re
2. if(re.search(r"Air","Airline")!=None):
3. print("Pattern found")
4. else:
5. print("Pattern not found")

**Output:**

Pattern found

**Example 2:**

1. import re
2. if(re.search(r"Air","airline")!=None):
3. print("Pattern found")
4. else:
5. print("Pattern not found")

**Output:**

Pattern not found

**More on regular Expressions:**

Go through metacharacters and the respective examples given below.

| **Metacharacter** | **Usage** | **Requirement** | **Solution** | **Remarks** |
| --- | --- | --- | --- | --- |
| . | Used to match one occurrence of any character | To search the pattern having two characters in between A and l in the given string "Aopline". | import re if(re.search(r"A..l","Aopline")!=None):     print("Pattern found") else:     print("Pattern not found") | “.” stands for any character. If any two characters are there between A and l, then the pattern has matched |
| \d | Used to match one occurrence of any digit from 0-9 | To search for a digit between A and l in the given string "A2line". | import re if(re.search(r"A\dl","A2line")!=None):     print("Pattern found") else:     print("Pattern not found") | \d checks for a digit. If any digit is found between A and l, then the pattern has matched |
| \* | Used to match zero or more occurrences of previous character | To check if a number is found 0 or n times after A in the given string. | import re if(re.search(r"A\d\*","A2234line")!=None):     print("Pattern found") else:     print("Pattern not found") | Checks if a number is found 0 or n times after A |
| + | Used to match one or more occurrences of previous character | To check if a number is found 1 or n times after A in the given string. | import re if(re.search(r"A\d+","Airline")!=None):     print("Pattern found") else:     print("Pattern not found") | Checks if a number is found 1 or n times after A |
| ? | Used to match zero or one occurrence of previous character | To check if a number is found 0 or 1 times after A in the given string. | import re if(re.search(r"A\d?i","Airline")!=None):     print("Pattern found") else:     print("Pattern not found") | Checks if a number is found 0 or 1 times after A |
| {n} | Used to match exactly n occurrences of previous character | To check if 3 digits are present after A in the given string. | import re if(re.search(r"A\d{3}i","A223irline")!=None):     print("Pattern found") else:     print("Pattern not found") | {n} checks if the preceding character appears exactly n times. Here we are checking if there are 3 digits after A |
| [] | Used to match one occurrence of any characters present within square brackets | To search for a number between 4 and 8 in between A and l in the given string. | import re if(re.search(r"A[4-8]l","A2line")!=None):     print("Pattern found") else:     print("Pattern not found") | [] does a single character substitution. We can specify a sequence of values. If any of the values are found, then the pattern has matched |
| ^ | Used to match a pattern at the beginning of a string | To check if the given string is starting with A. | import re if(re.search(r"^A","Airline")!=None):     print("Pattern found") else:     print("Pattern not found") | ^ checks if a pattern is at the beginning of the string. Here we check if string begins with “A” |
| $ | Used to match a pattern at the end of a string | To check if the given string is ending with e. | import re if(re.search(r"e$","Airline")!=None):     print("Pattern found") else:     print("Pattern not found") | $ checks if a pattern is at the end of the string. Here we check if string ends with “e” |
| \w | Used to match an word character which includes alphabets(a-zA-Z), digits(0-9) and '\_' | To check whether last character is alphanumeric or not. | import re if(re.search(r"\w$","Airline%")!=None):     print("Pattern found") else:     print("Pattern not found") | \w checks for a-z,A-Z,0-9,\_ Here we check if the last character is an alphanumeric character |
| \s | Used to match single space or sequence of spaces (including \t \n) | To check for the space after "Air" in the given string "Airline". | import re if(re.search(r"Air\s","Airline")!=None):     print("Pattern found") else:     print("Pattern not found") | \s indicates a space. Here we are checking if there is a space after Air |
| | | Used to match any one pattern which is on either side of it | To search for the pattern "Hell" or "Fell" in the given string "Fellow". | import re if(re.search(r"Hell|Fell","Fellow")!=None):     print("Pattern found") else:     print("Pattern not found") | | acts like ‘or’ operator. If Hell or Fell is found in the string, the pattern is found |

**re -Replacing data:**

We can use the function ‘sub’ to perform a substitution.

Go through the below example

1. import re
2. flight\_details="Flight Savana Airlines a2134"
3. print(flight\_details)
4. print(re.sub(r"Flight",r"Plane",flight\_details))

Output:

Flight Savana Airlines a2134

Plane Savana Airlines a2134

# File handling in python

**File handling in python:**

Files can be accessed from a python program for reading the data from a file and writing the data into a file.

**Opening a file in python:**

Python provides a function called **open()** to open a file programmatically. open() function returns a file object using which other file operations like reading, writing and closing of the file are done.

**Syntax:**

file\_object = open(file\_name\_path [, access\_mode])

where,

* **file\_name\_path** is the filename or path of the file to be opened.
* **access\_mode** is the mode in which the file has to be opened which is optional. If not specified, default value will be read.
* **file\_object** is the object returned by open method which is used for further file operations.

**Access Mode:**

**r**: If the access mode specified is ‘r’, then specified file is opened in **read mode**, if the file exists. Otherwise if the file does not exist it will throw an error.

**w**: If the access mode specified is ‘w’, then specified file is opened in **write mode.** If the file exists, then the content present in the file is truncated. Otherwise if the file does not exist, it will create a new file and open it for writing.

**a**: If the access mode specified is ‘a’, then specified file is opened in **append mode.** If the file exists, the content present in the file are preserved. Otherwise if the file does not exist, it will create a new file and open it for appending.

**Example:**

fhr=open("data.txt","r")

**Closing a file**

The number of files that can be simultaneously opened by a program is limited. So it is very important to close all the files, once the operations are completed.

**Syntax:**

file\_object.close()

**Example:**

fhr.close()

**Reading from file**

Consider a file data.txt with below data.

this is first line  
you are reading the second line  
now you are dealing with third line

**Reading single line**

Python provides **readline()** function to read the single line from the file at a time. When the end of the file reached it returns an empty string.

**Syntax:**

var\_name= file\_object.readline()

**Example:**

1. fhr=open("data.txt","r")
2. line1=fhr.readline()
3. print(line1,end="")
4. line2=fhr.readline()
5. print(line2,end="")
6. line3=fhr.readline()
7. print(line3,end="")

**Output:**

this is first line  
you are reading the second line  
now you are dealing with third line

**Reading the contents of a file into a list:**

Python provides **readlines()** function to read entire content of file into a variable where each line will present as an element of the list.

**Syntax:**

var=file\_object.readlines()

**Example:**

1. fhr=open("data.txt","r")
2. list\_var=fhr.readlines()
3. for line in list\_var:
4. print(line,end="")
5. fhr.close()

**Output:**

this is first line  
you are reading the second line  
now you are dealing with third line

**Reading the contents of a file into a String:**

Python provides **read(size)** function to read the specified size(number) of characters from the file as a string into a variable. If no size is specified, then entire content of the file is read as a string into a variable.

**Syntax:**

var\_name= file\_object.read(size)

**Example 1:**

1. fhr=open("data.txt","r")
2. data =fhr.read(10)
3. print(data)
4. fhr.close()

**Output:**

this is fi

**Example 2:**

1. fhr=open("data.txt","r")
2. data =fhr.read()
3. print(data)
4. fhr.close()

**Output:**

this is first line  
you are reading the second line  
now you are dealing with third line

**Iterating through file object read the content line by line**

We can also iterate through file object to read the content of the file line by line in a simple way. This method is fast and efficient compared to other methods of reading the file contents.

**Example:**

1. fhr=open("data.txt","r")
2. for line in fhr:
3. print(line,end="")
4. fhr.close()

**Output:**

this is first line  
you are reading the second line  
now you are dealing with third line

**Writing into file:**

Python provides **write(data)** function to write the given data which is a string into the file and it returns the number of characters written into file.

**Syntax:**

var=file\_object.write(data)

where,

* **data** is the content which is a string to be written into a file
* **var** is a variable assigned with number characters written into the file

**Example:**

1. fhr=open("data.txt","r")
2. data =fhr.read()
3. print("Before writing:")
4. print(data)
5. fhr.close()
6. fhw=open("data.txt","w")
7. num=fhw.write("this new first line written\n")
8. num1=fhw.write("this new second line written\n")
9. print("num:",num)
10. print("num1:",num1)
11. fhw.close()
12. fhr=open("data.txt","r")
13. data =fhr.read()
14. print("After writing:")
15. print(data)
16. fhr.close()

**Output:**

Before writing:  
this is first line  
you are reading the second line  
now you are dealing with third line  
num: 28  
num1: 29  
After writing:  
this new first line written  
this new second line written

**Note:** The file **‘data.txt’** is opened in **write mode**, so previous content of file is **truncated** and it contains only two lines after writing into it.

**Another example:**

1. fhr=open("data.txt","r")
2. data =fhr.read()
3. print("Before writing:")
4. print(data)
5. fhr.close()
6. fhw=open("data.txt","a")
7. num=fhw.write("this new first line written\n")
8. num1=fhw.write("this new second line written\n")
9. print("num:",num)
10. print("num1:",num1)
11. fhw.close()
12. fhr=open("data.txt","r")
13. data =fhr.read()
14. print("After writing:")
15. print(data)
16. fhr.close()

**Output:**

Before writing:  
this is first line  
you are reading the second line  
now you are dealing with third line  
num: 28  
num1: 29  
After writing:  
this is first line  
you are reading the second line  
now you are dealing with third line  
this new first line written  
this new second line written

**Note:** The file **'data.txt'** is opened in **append mode**, so the previous content of file is **preserved** and it contains all five lines after writing into it.

**Getting current position of the file object pointer:**

Python provides **tell()** method to get current position which is pointed by file object within the file.

Syntax:

file\_object.tell()

**Example:**

1. fhr=open("data.txt","r")
2. cur\_pos=fhr.tell()
3. print(cur\_pos)
4. data =fhr.readline()
5. print(data)
6. cur\_pos=fhr.tell()
7. print(cur\_pos)
8. data =fhr.readline()
9. print(data)
10. fhr.close()

**Output:**

0  
this new first line written

29  
this new second line written

**Navigating the file object pointer**

Python provides **seek()** function to navigate the file object pointer to the required position specified.

**Syntax:**

file\_object.seek(offset,[whence])

where,

**file\_object** indicates the file object pointer to be navigated

**offset** indicates which position the file object pointer is to be navigated

if offset is,

* **positive** navigation is done in forward direction
* **negative** navigation is done in backward direction

**whence** represent reference point for navigating the file object pointer. whence is optional, if not specified default value is 0.

If whence value is

* **0**, navigation will take the reference of beginning of file (absolute positioning)
* **1**, navigation will take the reference of current position (relative positioning) of the file object pointer
* **2**, navigation will take the reference of end of file (relative positioning)

**Note:**If you are working with is a text file, then the access mode of the should be ‘rb+’ (which opens a file for reading and writing in binary format) otherwise relative positioning will misbehave.

**Example:**

1. fhr=open("data.txt","rb+")
2. print(fhr.tell())
3. fhr.seek(12) *#navigates to 12th position from beginning of the file*
4. print(fhr.tell())
5. fhr.seek(3,1) *#navigates to 3rd position from current position of the file object position*
6. print(fhr.tell())
7. fhr.seek(-3,2)#navigates to 3rd position from end of the file in backward direction
8. print(fhr.tell())
9. fhr.close()

**Output:**

0  
12  
15  
56

**File object attributes:**

file\_object.**closed:**closed attribute returns true if the file is closed else it will return false.

file\_object.**mode:**mode attribute returns mode in which the file has been opened.

file\_object.**name:**name attribute returns the name of the file opened.

**Example:**

1. fhr=open("data.txt","rb+")
2. print("file name:",fhr.name)
3. print("access mode:",fhr.mode)
4. print("closed?",fhr.closed)
5. fhr.close()
6. print("after closing the file closed?",fhr.closed)

**Output:**

file name: data.txt  
access mode: rb+  
closed? False  
after closing the file closed? True

# Exception handling in python

Sometimes the programs may misbehave or terminate/crash unexpectedly due to some unexpected events during the execution of a program. These unexpected events are called as **exceptions** and the process of handling them to avoid misbehavior or crashing the program is called as **exception handling**.

Let’s execute the below code in python playground and have a look at the output.

1. def calculate\_expenditure(list\_of\_expenditure):
2. total=0
3. for expenditure in list\_of\_expenditure:
4. total+=expenditure
5. print(total)
6. list\_of\_values=[100,200,300,"400",500]
7. calculate\_expenditure(list\_of\_values)

Above code will give an error, one way to take care of such error situation is to use selection constructs. The error was due to addition of a string (“400”) to an integer. If we add a condition to check whether the expenditure is of type int, that would solve this error.

But that can cause further issues. Let's see that by executing the below code in python playground.

1. def calculate\_expenditure(list\_of\_expenditure):
2. total=0
3. for expenditure in list\_of\_expenditure:
4. if(type(expenditure) is int):
5. total+=expenditure
6. else:
7. print("Wrong data type")
8. break
9. print(total)
10. list\_of\_values=[100,200,300,"400",500]
11. calculate\_expenditure(list\_of\_values)

Although we have handled this error using if statement, the function itself returns wrong output when there is error in the input.   
The ideal situation would be if the function can tell us that something went wrong.

In python we can create a try and except block of code to handle exceptions.  
If any exception occurs in the try block of code, it will jump to except block of code.  
Once the except block is executed, the code continues to execute other statements outside except block.

1. def calculate\_expenditure(list\_of\_expenditure):
2. total=0
3. try:
4. for expenditure in list\_of\_expenditure:
5. total+=expenditure
6. print(total)
7. except:
8. print("Some error occured")
9. print("Returning back from function.")
10. list\_of\_values=[100,200,300,"400",500]
11. calculate\_expenditure(list\_of\_values)

With this we will not get incorrect output like before.

**Built-in Exception in python:**

Python has many kinds of exceptions predefined as part of the language. Here are some of the common types.

| **Built-in exception** | **When it will be raised** | **Example** |
| --- | --- | --- |
| ZeroDivisionError | When a value is divided by zero | num\_list=[] total=0 avg=total/len(num\_list) |
| TypeError | When we try to do an operation with incompatible data types | total=10 total+="20" |
| NameError | When we try to access a variable which is not defined | avg=total/10 where total is not defined |
| IndexError | When we try to access an index value which is out of range | num\_list=[1,2,3,4] value=num\_list[4] |
| ValueError | When we use a valid data type for an argument of a built-in function but passes an invalid value for it | #string is a valid data type for int() but the value “A” is invalid, as "A" can't be converted into int. value="A" num=int(value) |

Python also allows us to handle different exceptions that can occur separately. That means you can have a different action or message for every unique exception that occurs.

Here is the same expenditure calculation code with additional average expenditure calculation.

1. def calculate\_expenditure(list\_of\_expenditure):
2. total=0
3. try:
4. for expenditure in list\_of\_expenditure:
5. total+=expenditure
6. print("Total:",total)
7. avg=total/num\_values
8. print("Average:",avg)
9. except ZeroDivisionError:
10. print("Divide by Zero error")
11. except TypeError:
12. print("Wrong data type")
13. except:
14. print("Some error occured")
15. list\_of\_values=[100,200,300,"400",500]
16. num\_values=0
17. calculate\_expenditure(list\_of\_values)

**Note:**

1. Default except block is the one without any type mentioned.
2. If an error occurs and the matching except block is found, then that is executed.
3. If an error occurs and the matching except block is not found, it executes the default except block.
4. If an error occurs and the matching except block is not found and if the default except block is also not found, the code crashes.
5. The default except block, if present should be the last except block, otherwise it will result in a runtime error.

**Exception handling inside a function :**

If an exception occurs inside a function and if the exception is not caught inside it, then the exception is transferred to the function call. We have another opportunity to catch it, if we write function call inside another try and except block.

Try the below code in python playground and observe the output.

1. def calculate\_sum(list\_of\_expenditure):
2. total=0
3. try:
4. for expenditure in list\_of\_expenditure:
5. total+=expenditure
6. print("Total:",total)
7. avg=total/no\_values
8. print("Average:",avg)
9. except ZeroDivisionError:
10. print("Divide by Zero error")
11. except TypeError:
12. print("Wrong data type")
13. try:
14. list\_of\_values=[100,200,300,400,500]
15. num\_values=len(list\_of\_values)
16. calculate\_sum(list\_of\_values)
17. except NameError:
18. print("Name error occured")
19. except:
20. print("Some error occured")

**finally:**

Sometimes in programming we need to execute some code irrespective of whether the primary program logic itself succeeds or fails to do its job. In Python we can achieve this using a **finally** block. A finally block of statement is an optional part of the try-except statements. A code written inside the finally block will **ALWAYS** be executed.

finally block is majorly used to close the database connections in the programs which involves database connectivity.

Try the below code in python playground and observe the output.

1. balance=1000
2. amount="300Rs"
3. def take\_card():
4. print("Take the card out of ATM")
5. try:
6. if balance>=int(amount):
7. print("Withdraw")
8. else:
9. print("Invalid amount")
10. except TypeError:
11. print("Type Error Occurred")
12. except ValueError:
13. print("Value Error Occurred")
14. except:
15. print("Some error Occurred")
16. finally:
17. take\_card()

**Module:**

Module in python is basically a file which contains python definitions and statements

**Modules in Python Scripting:**

* Sys module
* Os module
* Subprocess
* Math
* Random
* Date Time
* JSoN

Let us go with some of the methods from these modules

**Sys Module:**

import sys

Print(sys.version) 🡺 To print current version

Print(sys.argv) 🡺 To print command line arguments?

**os Module:**

Import os

Print(os.getcwd) 🡺 To print current working directory

os.chdir (“pathname”) 🡺To change directory

os.mkdir(“pathname”) 🡺 Make directories

os.rmdir(“pathname”) 🡺 To remove directory.

os.remove(“pathname”) 🡺 To remove file.

Print(os.path.join(“path1”,”path2”)) 🡺 To join the path

Print(os.path.split(“pathname”)) 🡺 To split the path

Print(os.path.exists(“pathname”)) 🡺 To check that path is available or not

**Math Module:**

Import math

Print(math.pi) 🡺To print the value of pi

Print(math.e) 🡺To print the value of e

Print(math.degree(0.1)) 🡺To print the value of angle in degrees.

Print(math.acos(0.5)) 🡺To print the value of cos value.

Print(math.asin(0.5)) 🡺To print the value of sin value.

**Random Module:**

import random

print(random.randomrange(0,50,10)) 🡺To print the values in between 0,50 with step size 10.

print(random.randint(0,20)) 🡺To print the integer values in between 0,20.

**Date Time Module:**

Import datetime

Print(datetime.date.today()) 🡺 To print current date.

Print(datetime.date.today()) 🡺To perform some arithmetic function of date time objects.