Unit1

1.Python Variables

Python Variable is containers which store values. Python is not “statically typed”. We do not need to declare variables before using them or declare their type. A variable is created the moment we first assign a value to it. A Python variable is a name given to a memory location. It is the basic unit of storage in a program.

Example of Python Variables

```
Var = "Geeksforgeeks"

print(Var)
```

Output:

```
Geeksforgeeks
```

Rules for creating variables in Python

- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable names are case-sensitive (name, Name and NAME are three different variables).
- The reserved words(keywords) cannot be used naming the variable.
-

Declare the Variable

Let’s see how to declare the variable and print the variable.

```
# declaring the var
```

```
Number = 100

# display

print( Number)
```

Output:

```
100
```

Re-declare the Variable

We can re-declare the python variable once we have declared the variable already.

- Python3

```
# declaring the var

Number = 100

 # display

print("Before declare: ", Number)

# re-declare the var

Number = 120.3

print("After re-declare:", Number)
```

Output:

```
Before declare:  100

After re-declare: 120.3
```

Variable type in Python

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

Following are the standard or built-in data type of Python:

- Numeric
- Sequence Type
- Boolean
- Set
- Dictionary

Identifiers in Python

We can define identifiers in Python in few ways:

- An identifier is a user-defined name to represent a variable, a function, a class, a module, or any other object.
- It is a programmable entity in Python- one with a name.
- It is a name given to the fundamental building blocks in a program.

1. Rules for naming Identifiers in Python

a. A Python identifier can be a combination of lowercase/ uppercase letters, digits, or an underscore. The following characters are valid

- Lowercase letters (a to z)
- Uppercase letters (A to Z)
- Digits (0 to 9)
- Underscore (_)

An identifier cannot begin with a digit.

Some valid names:

- _9lives
- lives9

c. We cannot use special symbols in the identifier name. Some of these are:

!

@

#

$

%

d. We cannot use a keyword as an identifier.

Keywords are reserved names in Python and using one of those as a name for an identifier will result in a SyntaxError.

Reserved Classes of Python Identifiers

1. Single Leading Underscore (_*)

We use this identifier to store the result of the last evaluation in the interactive interpreter.
This result is stored in the __builtin__ module. Importing a module as *from module import ** does not import such private variables.

2. Leading and Trailing Double Underscores (__*__)

These are system-defined names (by the **interpreter**).

3. Leading Double Underscores (__*)

These are class-private names. Within a class definition, the interpreter rewrites (mangles) such a name to avoid name clashes between the private attributes of base and derived classes.

Lexical Definitions in Python Identifiers

- **identifier ::= (letter | "_") (letter | digit | "_")*** #It has to begin with a letter or an underscore; letters, digits, or/and underscores may follow

- **letter ::= lowercase | uppercase** #Anything from a-z and from A-Z

- **lowercase ::= "a" … "z"** #Lowercase letters a to z

- **uppercase ::= "A" … "Z"** #Uppercase letters A to Z

- **digit ::= "0" … "9"** #Integers 0 to 9

3.Python Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division.
There are 7 arithmetic operators in Python :

1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Modulus
6. Exponentiation
7. Floor division

1. Addition Operator : In Python, **+** is the addition operator. It is used to add 2 values.
Example :

```
val1 = 2

val2 = 3

# using the addition operator

res = val1 + val2

print(res)
```

Output :
```
5
```

2. Subtraction Operator : In Python, **–** is the subtraction operator. It is used to subtract the second value from the first value.
Example :

```
val1 = 2

val2 = 3

# using the subtraction operator

res = val1 - val2

print(res)
```

Output :
```
-1
```

3. Multiplication Operator : In Python, ***** is the multiplication operator. It is used to find the product of 2 values.
Example :

```
val1 = 2

val2 = 3

# using the multiplication operator

res = val1 * val2

print(res)
```

Output :

```
6
```

4. Division Operator : In Python, **/** is the division operator. It is used to find the quotient when first operand is divided by the second.
Example :

```
val1 = 3

val2 = 2

# using the division operator

res = val1 / val2

print(res)
```

Output :

```
1.5
```

5. Modulus Operator : In Python, **%** is the modulus operator. It is used to find the remainder when first operand is divided by the second.
Example :

```
val1 = 3

val2 = 2

# using the modulus operator

res = val1 % val2

print(res)
```

Output :

```
1
```

6. Exponentiation Operator : In Python, ****** is the exponentiation operator. It is used to raise the first operand to power of second.

Example :

```
val1 = 2

val2 = 3

# using the exponentiation operator

res = val1 ** val2

print(res)
```

Output :

```
8
```

7. Floor division : In Python, // is used to conduct the floor division. It is used to find the floor of the quotient when first operand is divided by the second.

Example :

```
val1 = 3
```

```
val2 = 2

# using the floor division

res = val1 // val2

print(res)
```

Output :
```
1
```

Below is the summary of all the 7 operators :

Operator	Description	Syntax
+	Addition: adds two operands	x + y
–	Subtraction: subtracts two operands	x – y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when first operand is divided by the second	x % y

4.Python Booleans

In programming you often need to know if an expression is `True` or `False`.

You can evaluate any expression in Python, and get one of two answers, `True` or `False`.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

Example

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

```
output

true

false

fALSE
```

When you run a condition in an if statement, Python returns `True` or `False`:

Evaluate Values and Variables

The `bool()` function allows you to evaluate any value, and give you `True` or `False` in return,

Example

Evaluate a string and a number:

```
print(bool("Hello"))
print(bool(15))
output
```

```
True
True
```

Most Values are True

Almost any value is evaluated to `True` if it has some sort of content.

Any string is `True`, except empty strings.

Any number is `True`, except `0`.

Any list, tuple, set, and dictionary are `True`, except empty ones.

unctions can Return a Boolean

You can create functions that returns a Boolean Value:

Example

Print the answer of a function:

```
def myFunction() :
  return True

print(myFunction())
```

OUTPUT

```
True
```

5.Operator Precedence in Python

An expression in python consists of variables, operators, values, etc. When the Python interpreter encounters any expression containing several operations, all operators get evaluated according to an ordered hierarchy, called operator precedence.

Python Operators Precedence Table

Listed below is the table of operator precedence in python, increasing from top to bottom and decreasing from bottom to top.

Operator	Description
:=	Assignment expression
lambda	Lambda expression

Operator	**Description**
if-else	Conditional expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
<, <=, >, >=,	Comparison operators
!=, ==	Equality operators
in, not in, is, is not,	Identity operators, membership operators
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Left and right Shifts
+, –	Addition and subtraction
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+x, -x, ~x	Unary plus, Unary minus, bitwise NOT
**	Exponentiation
await x	Await expression
x[index], x[index], x(arguments…), x.attribute	Subscription, slicing, call, attribute reference
() Parentheses	(Highest precedence)

Python Operators Precedence Rule - PEMDAS

Operator precedence in python follows the PEMDAS rule for arithmetic expressions. The precedence of operators is listed below in a high to low manner.

Firstly, parantheses will be evaluated, then exponentiation and so on.

- **P** – Parentheses
- **E** – Exponentiation
- **M** – Multiplication
- **D** – Division
- **A** – Addition
- **S** – Subtraction

In the case of tie means, if two operators whose precedence is equal appear in the expression, then the associativity rule is followed.

Associativity Rule

All the operators, except exponentiation(**) follow the left to right associativity. It means the evaluation will proceed from left to right, while evaluating the expression.

Example- (43 + 13 - 9 / 3 * 7)(43+13−9/3∗7)

In this case, the precedence of multiplication and division is equal, but further, they will be evaluated according to the left to right associativity.

Let's try to solve this expression by breaking it out and applying the precedence and associativity rule.

- Interpreter encounters the parenthesis (. Hence it will be evaluated first.
- Later there are four operators ++, -−, *∗ and //.
- Precedence of (/,(/, and *) >∗)> Precedence of (+, -)(+,−).
- We can use the operator precedence only if all operators belong to different-different levels from the hierarchy table, which is not the case in our example.
- Associativity rule will be followed for operators with the same precedence.
- This expression will be evaluated from left to right, 9 / 3 * 79/3∗7 = 3 * 73∗7 = 2121.
- Now, our expression has become 43+13-2143+13−21.
- Left to right Associativiy rule will be followed again. So, our final value of expression will be, 43+13-2143+13−21 = 56-2156−21 = 3535.

6.CONDITIONLS IF-ELSE CONSTRUCTS

Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

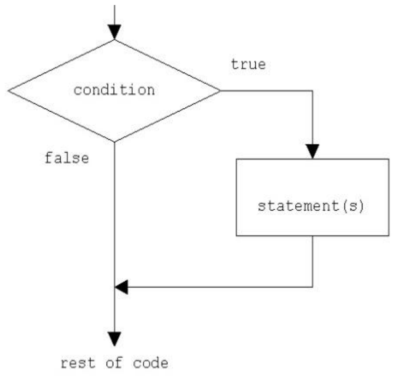
Indentation in Python

In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.

Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation.

The if statement

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

```
if expression:
    statement
```

Example 1

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even")
```

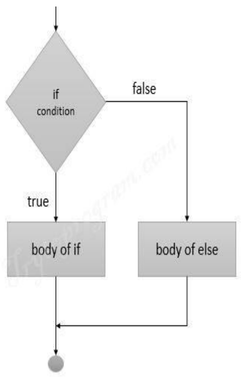
Output:

```
enter the number?10
Number is even
```

The if-else statement

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



The syntax of the if-else statement is given below.

```
if condition:
    #block of statements
else:
    #another block of statements (else-block)
```

Example 1 : Program to check whether a person is eligible to vote or not.

```
age = int (input("Enter your age? "))
if age>=18:
    print("You are eligible to vote !!");
else:
    print("Sorry! you have to wait !!");
```

Output:

```
Enter your age? 90
You are eligible to vote !!
```

The elif statement

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

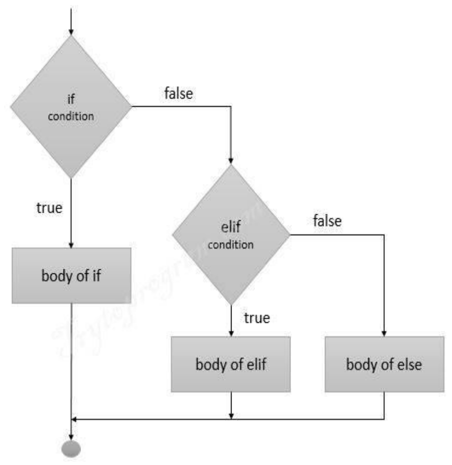
The syntax of the elif statement is given below.

```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements

elif expression 3:
    # block of statements

else:
    # block of statements
```



Example 1

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
```

```
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

Output:

```
Enter the number?15
number is not equal to 10, 50 or 100
```

7.Recursion in Python

The term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly.

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

Syntax:

```
def func(): <--
              |
              | (recursive call)
              |
    func() ----
```

Example 1: A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8….

```
# Program to print the fibonacci series upto n_terms
# Recursive function

def recursive_fibonacci(n):

  if n <= 1:

      return n

  else:

      return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))



n_terms = 10

 # check if the number of terms is valid

if n_terms <= 0:

  print("Invalid input ! Please input a positive value")

else:

  print("Fibonacci series:")

for i  in range(n_terms):

    print(recursive_fibonacci(i))
```

Output

```
Fibonacci series:
0
1
1
```

```
2

3

5

8

13

21

34
```

What is Tail-Recursion?

A unique type of recursion where the last procedure of a function is a recursive call. The recursion may be automated away by performing the request in the current stack frame and returning the output instead of generating a new stack frame. The tail-recursion may be optimized by the compiler which makes it better than non-tail recursive functions.

UNIT2

1.Python - Lists

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

When the above code is executed, it produces the following result −

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

Note − append() method is discussed in subsequent section.

When the above code is executed, it produces the following result −

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

When the above code is executed, it produces following result −

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Basic List Operations

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

ndexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sect

2.Python – Tuples

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing −

```
tup1 = ();
```

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print "tup1[0]: ", tup1[0];
print "tup2[1:5]: ", tup2[1:5];
```

When the above code is executed, it produces the following result −

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3;
```

When the above code is executed, it produces the following result −

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example −

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);
print tup;
del tup;
print "After deleting tup : ";
print tup;
```

Basic Tuples Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

3.Python - Sets

Mathematically a set is a collection of items not in any particular order. A Python set is similar to this mathematical definition with below additional conditions.

- The elements in the set cannot be duplicates.
- The elements in the set are immutable(cannot be modified) but the set as a whole is mutable.

- There is no index attached to any element in a python set. So they do not support any indexing or slicing operation.

Set Operations

The sets in python are typically used for mathematical operations like union, intersection, difference and complement etc. We can create a set, access it’s elements and carry out these mathematical operations as shown below.

Creating a set

A set is created by using the set() function or placing all the elements within a pair of curly braces.

Example

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
Months={"Jan","Feb","Mar"}
Dates={21,22,17}
print(Days)
print(Months)
print(Dates)
```

Output

When the above code is executed, it produces the following result. Please note how the order of the elements has changed in the result.

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
set(['Jan', 'Mar', 'Feb'])
set([17, 21, 22])
```

Accessing Values in a Set

We cannot access individual values in a set. We can only access all the elements together as shown above. But we can also get a list of individual elements by looping through the set.

Example

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])

for d in Days:
   print(d)
```

Output

When the above code is executed, it produces the following result −

```
Wed
Sun
Fri
Tue
```

```
Mon
Thu
Sat
```

Adding Items to a Set

We can add elements to a set by using add() method. Again as discussed there is no specific index attached to the newly added element.

Example

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])

Days.add("Sun")
print(Days)
```

Output

When the above code is executed, it produces the following result −

```
set(['Wed', 'Sun', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Removing Item from a Set

We can remove elements from a set by using discard() method. Again as discussed there is no specific index attached to the newly added element.

Example

```
Days=set(["Mon","Tue","Wed","Thu","Fri","Sat"])

Days.discard("Sun")
print(Days)
```

Output

When the above code is executed, it produces the following result.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Union of Sets

The union operation on two sets produces a new set containing all the distinct elements from both the sets. In the below example the element “Wed” is present in both the sets.

Example

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA|DaysB
print(AllDays)
```

Output

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed', 'Fri', 'Tue', 'Mon', 'Thu', 'Sat'])
```

Intersection of Sets

The intersection operation on two sets produces a new set containing only the common elements from both the sets. In the below example the element “Wed” is present in both the sets.

Example

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA & DaysB
print(AllDays)
```

Output

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Wed'])
```

Difference of Sets

The difference operation on two sets produces a new set containing only the elements from the first set and none from the second set. In the below example the element “Wed” is present in both the sets so it will not be found in the result set.

Example

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Wed","Thu","Fri","Sat","Sun"])
AllDays = DaysA - DaysB
print(AllDays)
```

Output

When the above code is executed, it produces the following result. Please note the result has only one “wed”.

```
set(['Mon', 'Tue'])
```

Compare Sets

We can check if a given set is a subset or superset of another set. The result is True or False depending on the elements present in the sets.

Example

```
DaysA = set(["Mon","Tue","Wed"])
DaysB = set(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"])
SubsetRes = DaysA <= DaysB
SupersetRes = DaysB >= DaysA
print(SubsetRes)
print(SupersetRes)
```

Output

When the above code is executed, it produces the following result −

```
True
True
```

4.Python - Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. For example −

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring. For example −

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

When the above code is executed, it produces the following result −

```
var1[0]:  H
var2[1:5]:  ytho
```

Updating Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether. For example −

```
#!/usr/bin/python

var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
```

When the above code is executed, it produces the following result −

```
Updated String :-  Hello Python
```

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1

r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example −

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

When the above code is executed, it produces the following result −

```
My name is Zara and weight is 21 kg!
```

Unicode String

Normal strings in Python are stored internally as 8-bit ASCII, while Unicode strings are stored as 16-bit Unicode. This allows for a more varied set of characters, including special characters from most languages in the world. I'll restrict my treatment of Unicode strings to the following −

```
#!/usr/bin/python

print u'Hello, world!'
```

When the above code is executed, it produces the following result −

```
Hello, world!
```

As you can see, Unicode strings use the prefix u, just as raw strings use the prefix r

5.Python - Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following result −

```
dict['Name']:  Zara
dict['Age']:  7
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following result −

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

example −

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

```
OUTPUT
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

A.More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.

B. Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

6.Loading and using modules

Python module refers to a piece of Python code that is designed to execute a spesific task. Technically, modules are simply Python script files (file extension `.py`) that contain function definitions and other statements. Python packages are a way of organizing modules into larger entities.

Loading modules

Python modules can be loaded in a number of different ways.

Let's start simple with the math module. Here, we'll load the math module using the `import` statement and try out some of the functions in the module, such as the square root function `sqrt`.

```
import math
math.sqrt(81)
```

```
9.0
```

Here we have loaded the math module by typing `import math`, which tells Python to read in the functions in the math module and make them available for use. In our example, we see that we can use a function within the math library by typing the name of the module first, a period, and then the name of function we would like to use afterward (e.g., `math.sqrt()`).

Renaming imported modules

We can also rename modules when they are imported. This can be helpful when using modules with longer names. Let's import the `math` module but rename it to `m` using the format `import module as name`. Then we can using the `sqrt` function from the math library and check the type of our module named `m`.

```
import math as m
m.sqrt(49)
```

```
7.0
```

```
type(m)
```

```
module
```

Here, we imported the `math` module to be usable with the name `m` instead of `math`. We will see other examples later in the course where using an alternate name is rather useful. For example, next week we will start using the `pandas` library for data analysis. It is customary to import pandas as `pd`:

```
import pandas as pd
```

Importing a single function

It is also possible to import only a single function from a module, rather than the entire module. This is sometimes useful when needing only a small piece of a large module. We can do this using the form `from module import function`. Let's import the `sqrt` function from the `math` module using this form. Again, we can test using our resulting function afterward.

```
from math import sqrt
sqrt(121)
```

```
11.0
```

Though this can be useful, it has the drawback that **the imported function could conflict with other built-in or imported function names**, and you lose the information about which module contains the imported function. You should only do this when you truly need to.

Importing a submodule

Some modules have submodules that can also be imported without importing the entire module. We may see examples of this later when making data plots using the pyplot sub-module of the Matplotlib module.

Using module functions

As we see above, the easiest way to use a module is to import it an then use its functions by typing `modulename.functionname()` and providing the necessary arguments. Yes, it is that simple.

However, there are times you may not know the names of all of the functions in a given module, or which are part of a module. You can view the list of functions that are part of a module by using the `dir()` function.

```
print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh',
```

7.Making your own Modules

Creating your own modules is easy, you've been doing it all along! Every Python program is also a module. You just have to make sure it has a `.py` extension. The following example should make it clear.

Creating your own Modules

ExampleÂ 8.3.Â How to create your own module

```
#!/usr/bin/python
# Filename: mymodule.py

def sayhi():
   print 'Hi, this is mymodule speaking.'

version = '0.1'

# End of mymodule.py
```

The above was a sample *module*. As you can see, there is nothing particularly special about compared to our usual Python program. We will next see how to use this module in our other Python programs.

Remember that the module should be placed in the same directory as the program that we import it in, or the module should be in one of the directories listed in `sys.path` .

```
#!/usr/bin/python
# Filename: mymodule_demo.py

import mymodule

mymodule.sayhi()
print 'Version', mymodule.version
```

Output

```
$ python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

How It Works

Notice that we use the same dotted notation to access members of the module. Python makes good reuse of the same notation to give the distinctive 'Pythonic' feel to it so that we don't have to keep learning new ways to do things.

from..import

Here is a version utilising the `from..import` syntax.

```
#!/usr/bin/python
# Filename: mymodule_demo2.py

from mymodule import sayhi, version
# Alternative:
# from mymodule import *

sayhi()
print 'Version', version
```

The output of `mymodule_demo2.py` is same as the output of `mymodule_demo.py`.

8.Python Standard Library

We know that a module is a file with some Python code, and a package is a directory for sub packages and modules. But the line between a package and a Python library is quite blurred.

A Python library is a reusable chunk of code that you may want to include in your programs/ projects.

Python Standard Library

The Python Standard Library is a collection of exact syntax, token, and semantics of Python. It comes bundled with core Python distribution. We mentioned this when we began with an introduction.

It is written in C, and handles functionality like I/O and other core modules. All this functionality together makes Python the language it is.

More than 200 core modules sit at the heart of the standard library. This library ships with Python.

But in addition to this library, you can also access a growing collection of several thousand components from the Python Package Index (PyPI). We mentioned it in the previous blog.

Note:Remaining in 4 unit

Unit3

1.Opening and Closing Files in Python

Python provides basic functions and methods necessary to manipulate files by default. You can do most of the **file** manipulation using a file object.

The open Function

Before you can read or write a file, you have to open it using Python's built-in open() function. This function creates a **file** object, which would be utilized to call other support methods associated with it.

Syntax

file object = open(file_name [, access_mode][, buffering])

Here are parameter details −

- **file_name** − The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode** − The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
- **buffering** − If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default(default behavior).
- Here is a list of the different modes of opening a file −

Sr.No	Modes & Description
1	**r**Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb**Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+**Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+**Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w**Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
6	**wb**Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

Sr.No	Modes & Description
7	**w+**Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
8	**wb+**Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
9	**a**Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
10	**ab**Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.
11	**a+**Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.
12	**ab+**Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

- ## The file Object Attributes
- Once a file is opened and you have one file object, you can get various information related to that file.
- Here is a list of all attributes related to file object −

Sr.No	Modes & Description
1	**file.closed**Returns true if file is closed, false otherwise.
2	**file.mode**Returns access mode with which file was opened.

Sr.No	Modes & Description
3	**file.name**Returns name of the file.
4	**file.softspace**Returns false if space explicitly required with print, true otherwise.

- ## Example

```
#!/usr/bin/python

# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

- ## Output
- This produces the following result −

```
Name of the file: foo.txt
Closed or not : False
Opening mode : wb
Softspace flag : 0
```

- ## The close() Method
- The close() method of a file object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.
- ## Syntax

```
fileObject.close()
```

- ## Example

```
#!/usr/bin/python
```

```
# Open a file
fo = open("foo.txt", "wb")
print "Name of the file: ", fo.name
# Close opend file
fo.close()
```

- Output
- This produces the following result −
- Name of the file: foo.txt

2.File Positions

There are two more methods of file objects used to determine or get files positions.

tell()

seek()

tell():

This method is used to tell us the current position within the file which means the next read or write operation will be performed that many bytes away from the start of the file.

Syntax :

obj.tell()

Example :

```
#create file object and show the use tell()

object=open("itvoyagers.txt",'w')

object.write("first statement n")

object.write("second statement n")
```

```
object=open("itvoyagers.txt",'r')

s=11

c=object.read(s)

print(object.tell()) #tells the position based on parameter passed in read operation

g=object.read()

print(object.tell()) #tells position after performing read() on entire file
```

seek():

This method is used to change the current position of file.This method has two main parameters offset and from.

Syntax :

```
obj.seek(offset,from)
```

Here, **offset** argument means the number of bytes to be moved.
from argument is used to specify the reference position from where the bytes needs to be moved.

Points to Remember for "from" argument

- if from is set to 0 ,it means use the beginning of the file as reference position

- if from is set to 1 ,it means use the current position of the file as reference position

- if from is set to 2 ,it means use the end of the file as reference position

Example :

```
#create file object and show the use seek()

with open("itvoyagers.txt","r") as f:

s=10
```

```
c=f.read(s) #reads till 10 char

print(c,end=' ') #adds space after first read()

f.seek(0,0) #goes to start position in file

c=f.read(s)

print(c)

f.seek(0,1) #goes to current position in file

c=f.read(s)

print(c)

f.seek(0,2) #goes to end position in file

c=f.read(s)

print(c)
```

4.Errors and Exceptions in Python

Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.
Two types of Error occurs in python.

1. Syntax errors
2. Logical errors (Exceptions)

Syntax errors

When the proper syntax of the language is not followed then a syntax error is thrown.
Example

- Python3

```
# initialize the amount variable

amount = 10000



# check that You are eligible to

#  purchase Dsa Self Paced or not

if(amount>2999)

    print("You are eligible to purchase Dsa Self Paced")
```

Output:

It returns a syntax error message because after the if statement a colon: is missing. We can fix this by writing the correct syntax.

logical errors(Exception)

When in the runtime an error that occurs after passing the syntax test is called exception or logical type. For example, when we divide any number by zero then the ZeroDivisionError exception is raised, or when we import a module that does not exist then ImportError is raised.

Example 1:

```
# initialize the amount variable

marks = 10000
```

```
# perform division with 0

a = marks / 0

print(a)
```

Exception	Description
IndexError	When the wrong index of a list is retrieved.
AssertionError	It occurs when the assert statement fails
AttributeError	It occurs when an attribute assignment is failed.
ImportError	It occurs when an imported module is not found.
KeyError	It occurs when the key of the dictionary is not found.
NameError	It occurs when the variable is not defined.
MemoryError	It occurs when a program runs out of memory.
TypeError	It occurs when a function and operation are applied in an incorrect type

Error Handling

When an error and an exception are raised then we handle that with the help of the Handling method.

- **Handling Exceptions with Try/Except/Finally**
  We can handle errors by the Try/Except/Finally method. we write unsafe code in the try, fall back code in except and final code in finally block.
  Example

- Python3

```
# put unsafe operation in try block

try:

     print("code start")



     # unsafe operation perform

     print(1 / 0)



# if error occur the it goes in except block

except:

     print("an error occurs")



# final code in finally block

finally:
     print("GeeksForGeeks")
```

- **Output:**

```
code start
an error occurs
GeeksForGeeks
```

5.Multiple Exception Handling in Python

Given a piece of code that can throw any of several different exceptions, and one needs to account for all of the potential exceptions that could be raised without creating duplicate code or long, meandering code passages.

If you can handle different exceptions all using a single block of code, they can be grouped together in a tuple as shown in the code given below :

Code #1 :

```
try:

    client_obj.get_url(url)

except (URLError, ValueError, SocketTimeout):

    client_obj.remove_url(url)
```

The `remove_url()` method will be called if any of the listed exceptions occurs. If, on the other hand, if one of the exceptions has to be handled differently, then put it into its own except clause as shown in the code given below :
Code #2 :

```
try:

    client_obj.get_url(url)
```

```
except (URLError, ValueError):

    client_obj.remove_url(url)

except SocketTimeout:

    client_obj.handle_url_timeout(url)
```

Many exceptions are grouped into an inheritance hierarchy. For such exceptions, all of the exceptions can be caught by simply specifying a base class

For example, instead of writing code as shown in the code given below –

Code #3 :

```
try:

    f = open(filename)

except (FileNotFoundError, PermissionError):

    ...
```

Except statement can be re-written as in the code given below. This works because `OSError` is a base class that's common to both the `FileNotFoundError` and **PermissionError** exceptions.

Code #4 :

```
try:

    f = open(filename)

except OSError:

    ...
```

Although it’s not specific to handle multiple exceptions per **se**, it is worth noting that one can get a handle to the thrown exception using them as a keyword as shown in the code given below.

Code #5 :

```
try:

    f = open(filename)



except OSError as e:

    if e.errno == errno.ENOENT:

        logger.error('File not found')

    elif e.errno == errno.EACCES:

        logger.error('Permission denied')

    else:

        logger.error('Unexpected error: % d', e.errno)
```

The **e** variable holds an instance of the raised OSError. This is useful if the exception has to be invested further, such as processing it based on the value of the additional status code. The except clauses are checked in the order listed and the first match executes.

Code #6 : Create situations where multiple except clauses might match

```
f = open('missing')
```

Output :

```
Traceback (most recent call last):

File "", line 1, in

FileNotFoundError: [Errno 2] No such file or directory: 'miss
```

```
try:

    f = open('missing')

    except OSError:

        print('It failed')

    except FileNotFoundError:

        print('File not found')
```

Output :

```
Failed
```

Here the except `FileNotFoundError` clause doesn't execute because the `OSError` is more general, matches the FileNotFoundError exception, and was listed first.

Unit4

Python Plotly tutorial

As we know **Plotly Dash** is most popular framework to build interactive dashboard in python. You can use Dash if want to build web application specifically for interactive dashboard.
But if you want to build a web application and you want to serve other things (like video, music etc.) along with interactive dashboard then you have to choose some other framework in Python.

Python Plotly Library is an open-source library that can be used for data visualization and understanding data simply and easily. Plotly supports various types of plots like line charts, scatter plots, histograms, cox plots, etc. So you all must be wondering why Plotly over other visualization tools or libraries? Here’s the answer –

- Plotly has hover tool capabilities that allow us to detect any outliers or anomalies in a large number of data points.
- It is visually attractive that can be accepted by a wide range of audiences.
- It allows us for the endless customization of our graphs that makes our plot more meaningful and understandable for others.

Package Structure of Plotly

There are three main modules in Plotly. They are:

- plotly.plotly
- plotly.graph.objects
- plotly.tools

plotly.plotly acts as the interface between the local machine and Plotly. It contains functions that require a response from Plotly’s server.

plotly.graph_objects module contains the objects (Figure, layout, data, and the definition of the plots like scatter plot, line chart) that are responsible for creating the plots. The Figure can be represented either as dict or instances of **plotly.graph_objects.Figure** and these are serialized as JSON before it gets passed to plotly.js. Consider the below example for better understanding.

Django is most popular framework in python to build such kind of complex web application.

Django is a high-level Python web framework that enables rapid development of secure and maintainable websites. Built by experienced developers, Django takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It is free and open source, has a thriving and active community, great documentation, and many options for free and paid-for support.

Django helps you write software that is:

- Complete
- Versatile
- Secure
- Scalable
- Maintainable
- Portable

Flask Python

Flask is a web framework, it’s a Python module that lets you develop web applications easily. It’s has a small and easy-to-extend core: it’s a microframework that doesn’t include an ORM (Object Relational Manager) or such features.

It does have many cool features like url routing, template engine. It is a WSGI web app framework.

Flask?

Flask is a web application framework written in Python. It was developed by Armin Ronacher, who led a team of international Python enthusiasts called Poocco. Flask is based on the Werkzeg WSGI toolkit and the Jinja2 template engine.Both are Pocco projects.

WSGI

The Web Server Gateway Interface (Web Server Gateway Interface, WSGI) has been used as a standard for Python web application development. WSGI is the specification of a common interface between web servers and web applications.

Werkzeug

Werkzeug is a WSGI toolkit that implements requests, response objects, and utility functions. This enables a web frame to be built on it. The Flask framework uses Werkzeg as one of its bases.

jinja2

jinja2 is a popular template engine for Python.A web template system combines a template with a specific data source to render a dynamic web page.

Microframework

Flask is often referred to as a microframework. It is designed to keep the core of the application simple and scalable.

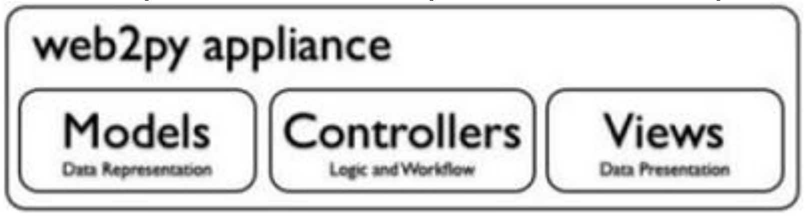
Instead of an abstraction layer for database support, Flask supports extensions to add such capabilities to the application.

Web2py

web2py is defined as a free, open-source web framework for agile development which involves database-driven web applications; it is written in Python and programmable in Python. It is a full-stack framework; it consists of all the necessary components, a developer needs to build a fully functional web application.

web2py framework follows the **Model-View-Controller** pattern of running web applications unlike traditional patterns.

- **Model** is a part of the application that includes logic for the data. The objects in model are used for retrieving and storing the data from the database.
- **View** is a part of the application, which helps in rendering the display of data to end users. The display of data is fetched from Model.
- **Controller** is a part of the application, which handles user interaction. Controllers can read data from a view, control user input, and send input data to the specific model.



- **web2py** has an in-built feature to manage cookies and sessions. After committing a transaction (in terms of SQL), the session is also stored simultaneously.
- **web2py** has the capacity of running the tasks in scheduled intervals after the completion of certain actions. This can be achieved with **CRON**.

Unit5

1.Methods in Python

In Object-Oriented Programming, we have (drum roll please) objects. These objects consist of properties and behavior. Furthermore, properties of the object are defined by the attributes and the behavior is defined using **methods**. These methods are defined inside a class. These methods are the reusable piece of code that can be invoked/called at any point in the program.

Python offers various types of these methods. These are crucial to becoming an efficient programmer and consequently are useful for a data science professional.

Types of Methods in Python

There are basically three types of methods in Python:

- Instance Method
- Class Method
- Static Method

Instance Methods

The purpose of instance methods is to set or get details about instances (objects), and that is why they're known as instance methods. They are the most common type of methods used in a Python class.

They have one default parameter- **self,** which points to an instance of the class. *Although you don't have to pass that every time.* You can change the name of this parameter but it is better to stick to the convention i.e **self.**

```
class My_class:
def instance_method(self):
return "This is an instance method."
```

In order to call an instance method, you've to create an object/instance of the class. With the help of this object, you can access any method of the class.

```
obj = My_class()
obj.instance_method()
```

2. Class Methods

The purpose of the class methods is to set or get the details (status) of the class. That is why they are known as class methods. They can't access or modify specific instance data. They are bound to the class instead of their objects. Two important things about class methods:

- In order to define a class method, you have to specify that it is a class method with the help of the @classmethod decorator
- Class methods also take one default parameter- **cls,** which points to the class. Again, this not mandatory to name the default parameter "cls". But it is always better to go with the conventions

Now let's look at how to create class methods:

```
class My_class:
@classmethod
  def class_method(cls):
    return "This is a class method."
```

we'll create the instance of this My_class as well and try calling this class_method():

```
obj = My_class()
obj.class_method()
```

USE; The most common use of the class methods is for creating **factory methods**. Factory methods are those methods that return a class object (like a constructor) for different use cases

3. Static Methods

Static methods cannot access the class data. In other words, they do not need to access the class data. They are self-sufficient and can work on their own. Since they are not attached to any class attribute, they cannot get or set the instance state or class state. In order to define a static method, we can use the @staticmethod decorator (in a similar way we used @classmethod decorator). Unlike instance methods and class methods, we do not need to pass any special or default parameters. Let's look at the implementation:

```
class My_class:

  @staticmethod
  def static_method():
    return "This is a static method."
```

USE:S**tatic Method –** They are used for creating utility functions. For accomplishing routine programming tasks we use utility functions

2. Python Inheritance

Like any other OOP languages, Python also supports the concept of class inheritance.

Inheritance allows us to create a new class from an existing class.

The new class that is created is known as **subclass** (child or derived class) and the existing class from which the child class is derived is known as **superclass** (parent or base class).

Python Inheritance Syntax

Here's the syntax of the inheritance in Python,

```
# define a superclass
class super_class:
    # attributes and method definition

# inheritance
class sub_class(super_class):
    # attributes and method of super_class
    # attributes and method of sub_class
```

Example: Python Inheritance

```
class Animal:

    # attribute and method of the parent class
    name = ""

    def eat(self):
        print("I can eat")

# inherit from Animal
class Dog(Animal):

    # new method in subclass
    def display(self):
        # access name attribute of superclass using self
        print("My name is ", self.name)

# create an object of the subclass
labrador = Dog()

# access superclass attribute and method
labrador.name = "Rohu"
labrador.eat()

# call subclass method
labrador.display()
```
Run Code

Output

```
I can eat
My name is  Rohu
```

is-a relationship

In Python, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an **is-a** relationship between two classes. For example,

1. **Car** is a **Vehicle**
2. **Apple** is a **Fruit**
3. **Cat** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Apple** can inherit from **Fruit**, and so on.

Method Overriding in Python Inheritance

In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Python.

Example: Method Overriding

```
class Animal:

    # attributes and method of the parent class
    name = ""

    def eat(self):
        print("I can eat")

# inherit from Animal
class Dog(Animal):

    # override eat() method
    def eat(self):
        print("I like to eat bones")

# create an object of the subclass
labrador = Dog()
```

```
# call the eat() method on the labrador object
labrador.eat()
```
Run Code

Output

```
I like to eat bones
```

Uses of Inheritance

1. Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
2. Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
3. Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.

3.Encapsulation in Python

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP), including abstraction, inheritance, and polymorphism.

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP), including abstraction, inheritance, and polymorphism. This lesson will cover what encapsulation is and how to implement it in Python.

What is Encapsulation in Python?

Encapsulation in Python describes the concept of **bundling data and methods within a single unit**. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.

ENCAPSULATION is a way to can restrict access to methods and variables from outside of class. Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.

Access Modifiers in Python

Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don't have direct access modifiers like public, private, and protected. We can achieve this by using single **underscore** and **double underscores**.

- **Public Member**: Accessible anywhere from otside oclass.
- **Private Member**: Accessible within the class
- **Protected Member**: Accessible within the class and its sub-classes

Public Member

Public data members are accessible within and outside of a class. All member variables of the class are by default public.

Private Member

We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.

Private members are accessible only within the class, and we can't access them directly from the class objects.

We can access private members from outside of a class using the following two approaches

- Create public method to access private members
- Use name mangling

Protected Member

Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore `_`.

Protected data members are used when you implement inheritance and want to allow data members access to only child classes.

Getters and Setters in Python

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members.

In Python, private variables are not hidden fields like in other programming languages. The getters and setters methods are often used when:

- When we want to avoid direct access to private variables

- To add validation logic for setting a value

Advantages of Encapsulation

- **Security**: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
- **Data Hiding**: The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
- **Simplicity**: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
- **Aesthetics**: Bundling data and methods within a class makes code more Jobs

z

4.Class method vs Static method in Python

.

What is Class Method in Python?

The @classmethod decorator is a built-in function decorator that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition. A class method receives the class as an implicit first argument, just like an instance method receives the instance

Syntax Python Class Method:

```
class C(object):
    @classmethod
    def fun(cls, arg1, arg2, ...):
       ....
fun: function that needs to be converted into a class method
returns: a class method for function.
```

- A class method is a method that is bound to the class and not the object of the class.
- They have the access to the state of the class as it takes a class parameter that points to the class and not the object instance.
- It can modify a class state that would apply across all the instances of the class. For example, it can modify a class variable that will be applicable to all the instances.

What is the Static Method in Python?

A static method does not receive an implicit first argument. A static method is also a method that is bound to the class and not the object of the class. This method can't access or modify the class state. It is present in a class because it makes sense for the method to be present in class.

Syntax Python Static Method:

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...
returns: a static method for function fun.
```

Class method vs Static Method

The difference between the Class method and the static method is:

- A class method takes cls as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can't access or modify it.
- In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

When to use the class or static method?

- We generally use the class method to create factory methods. Factory methods return class objects (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions.

How to define a class method and a static method?

To define a class method in python, we use @classmethod decorator, and to define a static method we use @staticmethod decorator.

Let us look at an example to understand the difference between both of them. Let us say we want to create a class Person. Now, python doesn't support method overloading like C++ or Java so we use class methods to create factory methods. In the below example we use a class method to create a person object from birth year.

As explained above we use static methods to create utility functions. In the below example we use a static method to check if a person is an adult or not.

One simple Example :

class method:

- Python3

```
class MyClass:

    def __init__(self, value):

        self.value = value



    def get_value(self):

        return self.value

 # Create an instance of MyClass

obj = MyClass(10)

 # Call the get_value method on the instance

print(obj.get_value())  # Output: 10
```

Output

```
10
```

Static method:-

- Python3

```
class MyClass:

    def __init__(self, value):

        self.value = value
    @staticmethod

    def get_max_value(x, y):

        return max(x, y)



# Create an instance of MyClass

obj = MyClass(10)

print(MyClass.get_max_value(20, 30))

print(obj.get_max_value(20, 30))
```

Output

```
30

30
```

Output:

```
21

25

True
```

5.Polymorphism in Python

What is Polymorphism: The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.

Example of inbuilt polymorphic functions:

- Python3

```
# Python program to demonstrate in-built poly-

# morphic functions



# len() being used for a string

print(len("geeks"))

 # len() being used for a list

print(len([10, 20, 30]))
```

Output:

```
5

3
```

Polymorphism with class methods:
The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

Polymorphism with Inheritance:
In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

- Python3

```
class Bird:
  def intro(self):
    print("There are many types of birds.")

  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

class ostrich(Bird):
  def flight(self):
    print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()
 obj_bird.intro()
obj_bird.flight()
 obj_spr.intro()
obj_spr.flight()

 obj_ost.intro()

obj_ost.flight()
```

Output:

```
There are many types of birds.

Most of the birds can fly but some cannot.

There are many types of birds.

Sparrows can fly.

There are many types of birds.

Ostriches cannot fly.
```

Polymorphism with a Function and objects:

It is also possible to create a function that can take any object, allowing for polymorphism. In this example, let's create a function called "func()" which will take an object which we will name "obj". Though we are using the name 'obj', any instantiated object will be able to be called into this function. Next, let's give the function something to do that uses the 'obj' object we passed to it. In this case, let's call the three methods, viz., capital(), language() and type(), each of which is defined in the two classes 'India' and 'USA'. Next, let's create instantiations of both the 'India' and 'USA' classes if we don't have them already. With those, we can call their action using the same func() function:

```
def func(obj):
        obj.capital()
        obj.language()
        obj.type()


obj_ind = India()
obj_usa = USA()


func(obj_ind)
```

func(obj_usa)

Code: Implementing Polymorphism with a Function

- Python3

```
class India():

    def capital(self):

        print("New Delhi is the capital of India.")



    def language(self):

        print("Hindi is the most widely spoken language of India.")



    def type(self):

        print("India is a developing country.")



class USA():

    def capital(self):

        print("Washington, D.C. is the capital of USA.")



    def language(self):

        print("English is the primary language of USA.")
    def type(self):
        print("USA is a developed country.")


def func(obj):
    obj.capital()
    obj.language()
    obj.type()


obj_ind = India()
obj_usa = USA()
func(obj_ind)
func(obj_usa)
```

Output:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

7.Creating Classes in Python

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

Syntax

```
class ClassName:
    #statement_suite
```

In Python, we must notice that each class is associated with a documentation string which can be accessed by using **<class-name>.__doc__.** A class contains a statement suite including fields, constructor, function, etc. definition.

Consider the following example to create a class **Employee** which contains two fields as Employee id, and name.

The class also contains a function **display()**, which is used to display the information of the **Employee.**

Example

```
class Employee:
    id = 10
    name = "Devansh"
    def display (self):
        print(self.id,self.name)
```

Here, the **self** is used as a reference variable, which refers to the current class object. It is always the first argument in the function definition. However, using **self** is optional in the function call.

The self-parameter

The self-parameter refers to the current instance of the class and accesses the class variables. We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.

8.Python object persistence (shelve)

The shelve module in Python’s standard library is a simple yet effective tool for persistent data storage when using a relational database solution is not required. The shelf object defined in this module is dictionary-like object which is persistently stored in a disk file. This creates afile similar to dbm database on UNIX like systems. Only string data type can be used as key in this special dictionary object, whereas any picklable object can serve as value.

The shelve module defines three classes as follows −

Sr.No.	Module & Description
1	**Shelf**This is the base class for shelf implementations. It is initialized with dict-like object.
2	**BsdDbShelf** This is a subclass of Shelf class. The dict object passed to its constructor must support first(), next(), previous(), last() and set_location() methods.
3	**DbfilenameShelf** This is also a subclass of Shelf but accepts a filename as parameter to its constructor rather than dict object.

asiest way to form a Shelf object is to use open() function defined in shelve module which return a DbfilenameShelf object.

```
open(filename, flag = 'c', protocol=None, writeback = False)
```

The filename parameter is assigned to the database created.

Default value for flag parameter is ‘c’ for read/write access. Other flags are ‘w’ (write only) ‘r’ (read only) and ‘n’ (new with read/write)

Protocol parameter denotes pickle protocol writeback parameter by default is false. If set to true, the accessed entries are cached. Every access calls sync() and close() operations hence process may be slow.

This will create test.dir file in current directory and store key-value data in hashed form. The Shelf object has following methods available −

Sr.No.	Method & Description
1	**close()**synchronise and close persistent dict object.
2	**sync()**Write back all entries in the cache if shelf was opened with writeback set to True.
3	**get()**returns value associated with key

Sr.No.	Method & Description
4	**items()**list of tuples – each tuple is key value pair
5	**keys()**list of shelf keys
6	**pop()**remove specified key and return the corresponding value.
7	**update()**Update shelf from another dict/iterable
8	**values()**list of shelf values

To access value of a particular key in shelf.

```
>>> s=shelve.open('test')
>>> s['age']
23
>>> s['age']=25
>>> s.get('age')
25
```

The items(), keys() and values() methods return view objects.

```
>>> list(s.items())
[('name', 'Ajay'), ('age', 25), ('marks', 75)]
>>> list(s.keys())
['name', 'age', 'marks']
>>> list(s.values())
['Ajay', 25, 75]
```

To remove a key-value pair from shelf

```
>>> s.pop('marks')
75
>>> list(s.items())
[('name', 'Ajay'), ('age', 25)]
```

Notice that key-value pair of marks-75 has been removed.

To merge items of another dictionary with shelf use update() method

```
>>> d={'salary':10000, 'designation':'manager'}
>>> s.update(d)
>>> list(s.items())
[('name', 'Ajay'), ('age', 25), ('salary', 10000), ('designation', 'manager')]
```