**MCA20204 : PYTHON PROGRAMMING**

**UNIT-I**

**CHAPTER-1 : PYTHON BASICS,OBJECTS**

**1). Introduction to Python.**

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other scripting languages.

***Python Features* :-** Python's features include:

* **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
* **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
* **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
* **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
* **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
* **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
* **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
* **Databases:** Python provides interfaces to all major commercial databases.
* **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
* **Scalable:** Python provides a better structure and support for large programs than shell scripting.

**Apart from the above-mentioned features, Python has a big list of good features, few are listed below:**

* It supports functional and structured programming methods as well as OOP.
* It can be used as a scripting language or can be compiled to byte-code for building large applications.
* It provides very high-level dynamic data types and supports dynamic type checking.
* It supports automatic garbage collection.
* It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

**2). Python Objects.**

Python is an object-oriented programming language. Everything is in Python treated as an object, including variable, function, list, tuple, dictionary, set, etc. Every object belongs to its class. For example - An integer variable belongs to integer class. An object is a real-life entity. An object is the collection of various data and functions that operate on those data. An object contains the following properties.

* **State -** The attributes of an object represents its state. It also reflects the properties of an object.
* **Behavior -** The method of an object represents its behavior.
* **Identity -** Each object must be uniquely identified and allow interacting with the other objects.

The classes and objects are the essential key to the object-oriented programming. Classes are the blueprint of the object. Classes are used to bundling the data and functionality together. Each newly created class must have its object. To understand this consider the real-life example of class and object.

A human is a class which may have may attributes such as walking, sleeping, thinking, etc. Suppose we want to name and age of 100 humans, so we don't need to create a class for every person. We just need to instantiate the multiple objects of that particular class.

The class contains the user-defined data structure that holds the own data members such as variables, constructs, and member functions, which can be accessed by creating an object of the class.

The syntax of creating a class is given below. The syntax of creating a class is given below.

### Syntax:

1. **class** ClassName:
2. #statement\_suite

The class keyword is used to define the class and the user-define class name replaces ClassName.

### Creating an Object of class

The object is created using the class name. The syntax is given below.

**Syntax:**

1. <object-name> = <**class**-name>(<arguments>)

In the following example, we have created the object of Person class.

**Example :**

1. **class** Person:
2. name = "John"
3. age = 24
4. **def** display (self):
5. **print**("Age: %d \nName: %s"%(self.age,self.name))
6. # Creating a emp instance of Employee class
7. per = Person()
8. per.display()

**Output :**

**Age: 24**

**Name: John**

**3). Standard data types in Python (Built-in Types).**

Python has five standard data types :

* Numbers
* Strings
* Lists
* Tuples
* Dictionary

**Python Numbers:-**

Number data types store numeric values. Number objects are created when you assign a value to them. For example :

var1 = 1

var2 = 10

**Python Strings :-**

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

**Python Lists :-**

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

**Python Tuples :-**

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

**Python Dictionary :-**

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

**Along with the above, Python also supports the following other Built-in Types.**

* Type
* None
* File
* Function
* Module
* Class
* Class Instance
* Method

**Types and the type() Built-in Function**

The syntax is as follows:

type*(object)*

The type() built-in function takes object and returns its type. The return object is a type object.

Example:-

>>> type(4) #int type

<type 'int'>

>>>

>>> type('Hello World!') #string type

<type 'string'>

>>>

>>> type(type(4)) #type type

<type 'type'>

**4). Internal Types in Python.**

The following are the internal types in Python :

* Code
* Frame
* Traceback
* Slice
* Ellipsis
* Xrange

The general application programmer would typically not interact with these objects directly,

**Code Objects :-**

Code objects are executable pieces of Python source that are byte-compiled, usually as return values from calling the compile() built-in function. Such objects are appropriate for execution by either **exec** or by the eval() built-in function. Code objects themselves do not contain any information regarding their execution environment, but they are at the heart of every user-defined function, all of which *do* contain some execution context.

**Frames :-**

These are objects representing execution stack frames in Python. Frame objects contain all the information the Python interpreter needs to know during a runtime execution environment.

**Tracebacks :-**

When we make an error in Python, an exception is raised. If exceptions are not caught or "handled," the interpreter exits with some diagnostic information similar to the output shown below:

Traceback (innermost last):

File "<stdin>", line *N?,* in ???

*ErrorName: error reason*

The traceback object is just a data item that holds the stack trace information for an exception and is created when an exception occurs. If a handler is provided for an exception, this handler is given access to the traceback object.

**Slice Objects:-**

Slice objects are created when using the Python extended slice syntax. This extended syntax allows for different types of indexing. These various types of indexing include *stride indexing,* multi-dimensional indexing, and indexing using the Ellipsis type. The syntax for multi-dimensional indexing is

*sequence*[*start1 : end1, start2 : end2*],

or using the ellipsis, *sequence*[…, *start1 : end1*]. Slice objects can also be generated by the slice() built-in function. Stride indexing for sequence types allows for a third slice element that allows for "step"-

like access with a syntax of *sequence*[*starting\_index : ending\_index : stride*].

**Ellipsis :-**

Ellipsis objects are used in extended slice notations as demonstrated above. These objects are used to represent the actual ellipses in the slice syntax (…). Like the Null object, ellipsis objects also have a single name, Ellipsis, and has a Boolean *true* value at all times.

**Xranges :-**

XRange objects are created by the built-in function xrange(), a sibling of the range() built-in function and used when memory is limited and for when range() generates an unusually large data set.

**5). Standard Type Operators (or) Python Operators.**

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language. Python provides a variety of operators, which are described as follows.

* Arithmetic operators
* Comparison operators
* Assignment Operators
* Logical Operators
* Bitwise Operators
* Membership Operators
* Identity Operators

## Arithmetic Operators :-

Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), \*(multiplication), /(divide), %(reminder), //(floor division), and exponent (\*\*) operators.

Consider the following table for a detailed explanation of arithmetic operators.

|  |  |
| --- | --- |
| Operator | Description |
| **+ (Addition)** | It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30 |
| **- (Subtraction)** | It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 10 => a - b = 10 |
| **/ (divide)** | It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0 |
| **\* (Multiplication)** | It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a \* b = 200 |
| **% (reminder)** | It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0 |
| **\*\* (Exponent)** | It is an exponent operator represented as it calculates the first operand power to the second operand. |
| **// (Floor division)** | It gives the floor value of the quotient produced by dividing the two operands. |

## Comparison operators :-

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly. The comparison operators are described in the following table.

|  |  |
| --- | --- |
| Operator | Description |
| == | If the value of two operands is equal, then the condition becomes true. |
| != | If the value of two operands is not equal, then the condition becomes true. |
| <= | If the first operand is less than or equal to the second operand, then the condition becomes true. |
| >= | If the first operand is greater than or equal to the second operand, then the condition becomes true. |
| > | If the first operand is greater than the second operand, then the condition becomes true. |
| **<** | If the first operand is less than the second operand, then the condition becomes true. |

## Assignment Operators :-

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

|  |  |
| --- | --- |
| Operator | Description |
| = | It assigns the value of the right expression to the left operand. |
| += | It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30. |
| -= | It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10. |
| \*= | It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a\* = b will be equal to a = a\* b and therefore, a = 200. |
| %= | It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0. |
| \*\*= | a\*\*=b will be equal to a=a\*\*b, for example, if a = 4, b =2, a\*\*=b will assign 4\*\*2 = 16 to a. |
| //= | A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a. |

## Bitwise Operators :-

The bitwise operators perform bit by bit operation on the values of the two operands. Consider the following example.

|  |  |
| --- | --- |
| Operator | Description |
| & (binary and) | If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied. |
| | (binary or) | The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1. |
| ^ (binary xor) | The resulting bit will be 1 if both the bits are different; otherwise, the resulting bit will be 0. |
| ~ (negation) | It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa. |
| << (left shift) | The left operand value is moved left by the number of bits present in the right operand. |
| >> (right shift) | The left operand is moved right by the number of bits present in the right operand. |

## Logical Operators :-

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

|  |  |
| --- | --- |
| Operator | Description |
| and | If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true. |
| or | If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true. |
| not | If an expression **a** is true, then not (a) will be false and vice versa. |

## Membership Operators :-

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

|  |  |
| --- | --- |
| Operator | Description |
| in | It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary). |
| not in | It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary). |

## Identity Operators :-

The identity operators are used to decide whether an element certain class or type.

|  |  |
| --- | --- |
| Operator | Description |
| is | It is evaluated to be true if the reference present at both sides point to the same object. |
| is not | It is evaluated to be true if the reference present at both sides do not point to the same object. |

### 6). Standard Type Built-in Functions

Along with generic operators which , Python also provides some built-in functions that can be applied to all the basic object types: cmp(), repr(), str(), type(), and the single reverse or back quotes ( '' ) operator, which is functionally-equivalent to repr().

**Function** **Operation**

cmp(obj1,obj2) compares obj1 and obj2, returns integer i where:

i < 0 if obj1 < obj2

i > 0 if obj1 > obj2

i == 0 if obj1 == obj2

repr(obj)/'obj' returns evaluatable string representation of obj

str(obj) returns printable string representation of obj

##### **7). Categorizing the Standard Types and Unsupported Types in Python**

Standard Types can be categorized as follows :

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Storage Model** | **Update Model** | **Access Model** |
| Numbers | Scalar | Immutable | Direct |
| Strings | Scalar | Immutable | Sequence |
| Lists | Container | Mutable | Sequence |
| Tuples | Container | Immutable | Sequence |
| Dictionaries | Container | Mutable | Mapping |

| **Unsupported Types in Python:-**  The following are the list of types that are not supported by Python.  **Boolean :-**  Unlike Pascal or Java, Python does not feature the Boolean type. Use integers instead.  **char or byte :-** Python does not have a char or byte type to hold either single character or 8-bit integers. Use strings of length one for characters and integers for 8-bit numbers.  **Pointer :-**  Since Python manages memory for you, there is no need to access pointer addresses. The closest to an address that you can get in Python is by looking at an object's identity using the id() built-in function. Since you have no control over this value, it's a moot point.  **int vs. short vs. Long :-**  Python's plain integers are the universal "standard" integer type, obviating the need for three different integer types, i.e., C's int, short, and long. For the record, Python's integers are implemented as C longs. For values larger in magnitude than regular integers (usually your system architecture size, i.e., 32-bit), use Python's long integer.  **float vs. Double :-**  C has both a single precision float type and double-precision double type. Python's float type is actually a C double. Python does not support a single-precision floating point type because its benefits are outweighed by the overhead required to support two types of floating point types. |
| --- |

**CHAPTER-2 : NUMBERS**

**1). Numbers in Python.**

Number data types store numeric values. They are immutable data types, which means that changing the value of a number data type results in a newly allocated object.

Different types of Number data types are :

* int
* float
* complex

**Int type :-**

int (Integers)are the whole number, including negative numbers but not fractions. In Python, there is no limit to how long an integer value can be.

|  |
| --- |
| **Example 1:-**Creating int and checking type num = -8  # print the data type  print(type(num)) |

**Output :-**

<class 'int'>

**Example 2:**Performing arithmetic Operations on int type

|  |
| --- |
| a = 5  b = 6  # Addition  c = a + b  print("Addition:",c)  d = 9  e = 6  # Subtraction  f = d - e  print("Subtraction:",f)  g = 8  h = 2  # Division  i = g // h  print("Division:",i)  j = 3  k = 5  # Multiplication  l = j \* k  print("Multiplication:",l)  m = 25  n = 5  # Modulus  o = m % n  print("Modulus:",o)  p = 6  q = 2  # Exponent  r = p \*\* q  print("Exponent:",r) |

**Output :-**

Addition: 11

Subtraction: 3

Division: 4

Multiplication: 15

Modulus: 0

Exponent: 36

**Float type :-**

This is a real number with floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation. Some examples of numbers that are represented as floats are 0.5 and -7.823457.

They can be created directly by entering a number with a decimal point, or by using operations such as division on integers. Extra zeros present at the number’s end are ignored automatically.

**Ex :-**Creating float and checking type

|  |
| --- |
| num = ¾   # print the data type  print(type(num)) |

**Output:**

<class 'float'>

As we have seen, dividing any two integers produces a float.

A float is also produced by running an operation on two floats, or a float and an integer.

|  |
| --- |
| num = 6 \* 7.0  print(type(num)) |

**Output:**

<class 'float'>

|  |
| --- |
| **Example 2:**Performing arithmetic Operations on float type  a = 5.5  b = 3.2  # Addition  c = a + b  print("Addition:", c)  # Subtraction  c = a-b  print("Subtraction:", c)  # Division  c = a/b  print("Division:", c)    # Multiplication  c = a\*b  print("Multiplication:", c) |

**Output**

Addition: 8.7

Subtraction: 2.3

Division: 1.71875

Multiplication: 17.6

**Complex type :-**

A complex number is a number that consists of the real and imaginary parts. For example, 2 + 3j is a complex number where 2 is the real component, and 3 multiplied by j is an imaginary part.

**Example 1:** Creating Complex and checking type

|  |
| --- |
| num = 6 + 9j  print(type(num)) |

**Output:**

<class 'complex'>

**Example 2:**Performing arithmetic operations on complex type

|  |
| --- |
| a = 1 + 5j  b = 2 + 3j  # Addition  c = a + b  print("Addition:",c)  d = 1 + 5j  e = 2 - 3j   # Subtraction  f = d - e  print("Subtraction:",f)  g = 1 + 5j  h = 2 + 3j  # Division  i = g / h  print("Division:",i)  j = 1 + 5j  k = 2 + 3j  # Multiplication  l = j \* k  print("Multiplication:",l) |

**Output :-**

Addition: (3+8j)

Subtraction: (-1+8j)

Division: (1.307692307692308+0.5384615384615384j)

Multiplication: (-13+13j)

**2). Types of Built-in functions applied on Numbers.**

* **Standard Type Functions :-**

The cmp(), str(), and type() built-in functions that apply for all standard types. For numbers, these functions will compare two numbers, convert numbers into strings, and gives a number's type output, respectively. Here are some examples of using these functions:

>>> cmp(-6, 2)

-1

>>> cmp(-4.333333, -2.718281828)

-1

>>> cmp(0xFF, 255)

0

>>> str(0xFF)

'255'

>>> str(55.3e2)

'5530.0'

>>> type(0xFF)

<type 'int'>

>>> type(98765432109876543210L)

<type 'long int'>

>>> type(2-1j)

<type 'complex'>

* **Numeric Type Functions :-**

Python currently supports different sets of built-in functions for numeric types. Some convert from one numeric type to another while others are more operational, performing some type of calculation on their numeric arguments.

**Conversion :-**

The int(), long(), float(), and complex() built-in functions are used to convert from any numeric type to another. Starting in Python 1.5, these functions will also take strings and return the numerical value represented by the string.

The following are some examples using the numeric type conversion built-ins:

>>> int(4.25555)

4

>>> long(42)

42L

>>> float(4)

4.0

>>> complex(4)

(4+0j)

* **Operational :-**

Python has five operational built-in functions for numeric types: abs(), coerce(), divmod(), pow(), and round().

**abs()** returns the absolute value of the given argument.

>>> abs(-1)

1

**coerce()**  just returns a tuple containing the converted pair of numbers. Here are some examples:

>>> coerce(1, 2)

(1, 2)

>>>

>>> coerce(1.3, 134L)

(1.3, 134.0)

>>>

**divmod()** built-in function combines division and modulus operations into a single function call that returns the pair (quotient, remainder) as a tuple.

>>> divmod(10,3)

(3, 1)

**pow()** function perform exponentiation.

>>> pow(2,5)

32

>>>

**round()** built-in function rounds a floating point number to the nearest integral number and returns that result as a float.

>>> round(3)

3.0

>>> round(3.45)

3.0

>>> round(3.4999999)

* **Integer Type Built-in Functions :-**

The following are the different types of Integer Type Built-in Functions.

|  |
| --- |
| Function operation |
| hex( *num* ) converts *num* to hexadecimal and return as string |
| oct( *num* ) converts *num* to octal and return as string |
| chr( *num* ) takes ASCII value *num* and returns ASCII character as string; |
| 0 <= *num* <=255 only |
| ord( *chr* ) takes ASCII *chr* and returns corresponding ordinal ASCII value; |
| *chr* must be a string of length 1 |

**3). Numeric Type Related Modules**

There are a number of modules in the Python standard library that add-on to the functionality of the operators and built-in functions for numeric types. The following table lists the key modules for use with numeric types.

**Table : Numeric Type Related Modules**

|  |  |
| --- | --- |
| Module | Contents |
| array | implements array types… a restricted sequence type |
| math/cmath | supplies standard C library mathematical functions; most functions available in math are implemented for complex numbers in the cmath module |
| operator | contains numeric operators available as function calls, i.e.,operator.sub(m, n) is equivalent to the difference (m - n) for numbers m and n |
| random | is default RNG module for Python… obsoletes rand and whrandom |

**CHAPTER-3 : SEQUENCES**

**1). Python Sequences.**

Python supports the following 3 different types of sequences. :

1. S**trings**

2. L**lists**

**3.** T**uples**

1. **Python Strings** :-

Strings are a **group of characters** written inside a **single or double-quote**s. Python does not have a **character type** so a single character inside quotes is also considered as a **string**.

**Ex:-**

name = “ MCA Students”

type(name)

**Output:**

**<class ‘str’>**

Strings are **immutable** in nature so we can **reassign** a variable to a new string but we **can’t** make any **changes** in the **string**.

**Ex:-**

city= ‘India’

print(city[2])

city[2] = ‘a’

**Output:**

Traceback (most recent call last):

File “<stdin>”, line 3, in <module>

TypeError: ‘str’ object does not support item assignment

#### 2.Python Lists :-

Python lists are similar to an array but they allow us to **create** a **heterogeneous collection** of items inside a **list**. A list can contain **numbers**, **strings**, **lists**, **tuples**, **dictionaries**, **objects**, etc. Lists are declared by using **square brackets** around **comma-separated** items.

**Ex:-**

list1 = [1,2,3,4]

list2 = [‘red’, ‘green’, ‘blue’]

list3 = [‘hello’, 100, 3.14, [1,2,3] ]

**Lists** are **mutable** which makes it easier to **change** and we can quickly **modify** a list by directly **accessing** it.

**Ex:-**

list = [10,20,30,40]

list[1] = 100

print( list)

**Output:** [10, 100, 30, 40]

#### 3.Python Tuples :-

Tuples are also a **sequence** of **Python objects**. A tuple is created by **separating** items with a **comma**. They can be optionally put inside the**parenthesis ()** but it is necessary to put parenthesis in an **empty tuple**. A **single item tuple** should use a**comma** in the end.

**Ex:-**

tup = ()

print( type(tup) )

tup = (1,2,3,4,5)

tup = ( “78 Street”, 3.8, 9826 )

print(tup)

**Output:**

<class ‘tuple’>

(’78 Street’, 3.8, 9826)

Tuples are also **immutable** like **strings** so we can only **reassign** the **variable** but we cannot **change**, **add** or **remove elements** from the **tuple**.

**Ex:-Output:**

Traceback (most recent call last):

File “<stdin>”, line 2, in <module>

TypeError: ‘tuple’ object does not support item assignment

**2). Mapping Types (or) Dictionaries in Python**

The mapping objects are used to map hash table values to arbitrary objects. In python there is mapping type called **dictionary**. It is mutable. The keys of the dictionary are arbitrary. As the value, we can use different kind of elements like lists, integers or any other mutable type objects.

Some dictionary related methods and operations are :

## Method len(d)

The len() method returns the number of elements in the dictionary.

## Operation d[k]

It will return the item of d with the key ‘k’. It may raise **KeyError** if the key is not mapped.

## Method get(key[, default])

The get() method will return the value from the key. The second argument is optional. If the key is not present, it will return the default value.

## Method items() 🡪 It will return the items using (key, value) pairs format.

Method keys() 🡪 Return the list of different keys in the dictionary.

Method values() 🡪 Return the list of different values from the dictionary**.**

## Method update(elem) 🡪 Modify the element elem in the dictionary.

## Example Code

## myDict = {'ten' : 10, 'twenty' : 20, 'thirty' : 30, 'forty' : 40}

## print(myDict)

## print(list(myDict.keys()))

## print(list(myDict.values()))

## #create items from the key-value pairs

## print(list(myDict.items()))

## myDict.update({'fifty' : 50})

## print(myDict)

## Output

{'ten': 10, 'twenty': 20, 'thirty': 30, 'forty': 40}

['ten', 'twenty', 'thirty', 'forty']

[10, 20, 30, 40]

[('ten', 10), ('twenty', 20), ('thirty', 30), ('forty', 40)]

{'ten': 10, 'twenty': 20, 'thirty': 30, 'forty': 40, 'fifty': 5

**3). Set Type in Python**

The sets are basically an unordered collection of distinct hash-table objects. We can use the set for some mathematical operations like set union, intersection, difference etc. We can also use set to remove duplicates from a collection.

The set do not record the element position. It does not support the indexing, slicing or other sequence related operations.

In python there are basically two types of sets. The **set** and the **frozenset**. The set type is mutable, whether the frozenset is immutable. We can perform add(), remove() and these kind of operations on set, but it is not possible for frozenset.

Some of the set related methods and operations are as follows :

## Method len(s)

The len() method returns the number of elements in the set.

## Operation (x in s) or (y not in s)

The **in** and **not in** operations are used to check the membership of an element in the set. In the first statement (x in s), it will return true, when the value x is available in set s. The second one (y not in s) will return true, when the element y is not present in the set.

## Method isdisjoint(other\_set)

This method will check whether the other\_set is disjoint with the current set or not. If at least one element is common for both of them, the method will return false.

## Method issuperset(other\_set)

This function returns true, when all of the elements in other\_set set is also present in the current set. It basically checks whether the current set is superset of the other\_set or not.

## Method union(other\_set)

The union() function returns a new set by collecting all of the elements from current set and the other\_set.

## Method intersection(other\_set)

The intersection() function returns a new set by collecting common elements from the current set and the other\_set.

## Method difference(other\_set)

The difference() method will return a set, where the final set contains all of the elements of the first set except the common elements of those two sets.

## Method add(elem)

Add the element elem in the set.

## Method discard(elem)

Remove the element elem from the set. This will work when the elem is present in the set. There is another method called remove(). In the remove(), it will raise KeyError if the item is not present in the set.

## Ex:-

## mySet1 = {1, 2, 5, 6}

## mySet2 = {8, 5, 3, 4}

## mySet3 = set(range(15)) # all elements from 0 to 14 in the set

## mySet4 = {10, 20, 30, 40}

## print(set(mySet1.union(mySet2)))

## print(set(mySet1.intersection(mySet2)))

## print(set(mySet1.difference(mySet2)))

## print(mySet3.issuperset(mySet1))

## print(mySet1.isdisjoint(mySet4))

## mySet4.add(45)

## print(mySet4)

## mySet4.discard(40)

## print(mySet4)

## Output

set([1, 2, 3, 4, 5, 6, 8])

set([5])

set([1, 2, 6])

True

True

set([40, 10, 20, 45, 30])

set([10, 20, 45, 30])

**4). Decision making (or) Conditional Control Statements in Python**

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.

|  |
| --- |
| In python, decision making is performed by the following statements. |
| Statement Description |
| If Statement: The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed. |
| If - else Statement: The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed. |
| Nested if Statement: Nested if statements enable us to use if ? else statement inside an outer if statement |

* **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 :**

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.

**if condition:**

**#block of statements**

**else:**

**#another block of statements (else-block)**

**Example :** 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 elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

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 :** Program to print the largest of the three numbers.

a = int(input("Enter a? "));

b = int(input("Enter b? "));

c = int(input("Enter c? "));

if a>b and a>c:

print("a is largest");

elif b>c:

print("b is largest");

else:

print("c is largest");

# 5). Python Loops (or) Iterative (or) Repetitive Control Structures.

The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.

For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times.

* **Python For Loop**:-

The for loop in Python is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of **for** loop in python is given below.

**for iterating\_var in sequence:**

**statement(s)**

**Example-1:** Iterating string using for loop

str = "Python"

for i in str:

print(i)

**Output:**

P

y

t

h

o

n

**Example-2:** Program to print the sum of the given list.

list = [10,30,23,43,65,12]

sum = 0

for i in list:

sum = sum+i

print("The sum is:",sum)

**Output:**

The sum is: 183

* **For loop Using range() function**

**The range() function**

The range() function is used to generate the sequence of the numbers. If we pass the range(10), it will generate the numbers from 0 to 9. The syntax of the range() function is given below.

**Syntax:-**

**range(start,stop,step size)**

* The start represents the beginning of the iteration.
* The stop represents that the loop will iterate till stop-1. The range(1,5) will generate numbers 1 to 4 iterations. It is optional.
* The step size is used to skip the specific numbers from the iteration. It is optional to use. By default, the step size is 1. It is optional.

**Example :** Program to print table of given number.

n = int(input("Enter the number "))

for i in range(1,11):

c = n\*i

print(n,"\*",i,"=",c)

**Output:**

Enter the number 10

10 \* 1 = 10

10 \* 2 = 20

10 \* 3 = 30

10 \* 4 = 40

10 \* 5 = 50

10 \* 6 = 60

10 \* 7 = 70

10 \* 8 = 80

10 \* 9 = 90

# Python While loop:-

The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a pre-tested loop.

It can be viewed as a repeating if statement. When we don't know the number of iterations then the while loop is most effective to use.

The syntax is given below.

**while expression:**

**statements**

Here, the statements can be a single statement or a group of statements. The expression should be any valid Python expression resulting in true or false. The true is any non-zero value and false is 0.

**Example:** Program to print 1 to 10 using while loop

i=1

#The while loop will iterate until condition becomes false.

While(i<=10):

print(i)

i=i+1

**Output:**

1

2

3

4

5

6

7

8

9

10

**UNIT-II**

**CHAPTER-1 : FILES**

**1). File Objects and File Built-in Function.**

**File objects** can be used not only to access normal disk files, but also any other type of "file" that uses that abstraction. The **open()** built-in function returns a file object which is then used for all succeeding operations on the file.

**File Built-in Function [open()]**

The **open()** built-in function provides a general interface to initiate the file input/output (I/O) process. **open()** returns a file object on a successful opening of the file or else results in an error situation. When a failure occurs, Python generates or *raises* an **IOError** exception.The basic syntax of the open() built-in function is:

**file\_object = open(file\_name, access\_mode='r', buffering= -1).**

The *file\_name* is a string containing the name of the file to open.The *access\_mode* optional variable is also a string, consisting of a set of flags indicating which mode to open the file with. Generally, files are opened with the modes "r," "w," or "a," representing read, write, and append, respectively.

Any file opened with mode "r" must exist. Any file opened with "w" will be truncated first if it exists, and then the file is (re)created. Any file opened with "a" will be opened for write. If the file exists, the initial position for file (write) access is set to the end-of file.

There are other modes supported by fopen() that will work with Python's open(). These include the "+" for read-write access and "b" for binary access.

The other optional argument, *buffering,* is used to indicate the type of buffering that should be performed when accessing the file. A value of 0 means no buffering should occur, a value of 1 signals line buffering, and any value greater than 1 indicates buffered I/O with the given value as the buffer size. Under normal circumstances, a *buffering* value is not given, thus using the system default.

Here are some examples for opening files:

fp = open('/etc/motd') #open file read

fp = open('test', 'w') #open file for Write

fp= open('data', 'r+') #open file for read/write

fp = open('c:\io.sys', 'rb') #open binary file for read

**2). File Built-in Methods**

File methods come in four different categories: input, output, movement within a file, which we will call "intra-file motion," and miscellaneous. A summary of all file methods can be shown following table.

|  |  |
| --- | --- |
| **Table : Methods for File Objects** | |
| **File Object Method** | **Operation** |
| file.close() | close file |
| file.fileno() | return integer file descriptor (FD) for file |
| file.flush() | flush internal buffer for file |
| file.read (size=-1) | read all or size bytes of file as a string and return it |
| file.readinto(buf, size) | read size bytes from file into buffer buf |
| file.readline() | read and return one line from file (includes trailing "\n") |
| file.readlines() | read and returns all lines from file as a list (includes all trailing "\n" characters) |
| file.seek(off, whence) | move to a location within file, off bytes offset from  whence (0 == beginning of file, 1 == current location, or  2 == end of file) |
| file.tell() | return current location within file |
| file.truncate(size=0) | truncate file to 0 or size bytes |
| file.write(str) | write string str to file |
| file.writelines(list) | write list of strings to file |

## 3). File Built-in Attributes

File objects also have data attributes in addition to its methods. These attributes hold auxiliary data related to the file object they belong to, such as the file name (*file.*name), the mode with which the file was opened (*file.*mode), whether the file is closed (*file.*closed), and a flag indicating whether an additional space character needs to be displayed before successive data items when using the **print** statement (*file.*softspace). The following table gives these attributes along with a brief description of each.

|  |  |
| --- | --- |
| **Table : Attributes for File Objects** | |
| ***File Object Attribute*** | ***Description*** |
| *file.*closed | 1 if *file* is closed, 0 otherwise |
| *file.*mode | access mode with which *file* was opened |
| *file.*name | name of *file* |
| *file.*softspace | 0 if space explicitly required with print, 1 otherwise; rarely used by the programmer—generally for internal use only |

## 4). Standard Files

There are generally three standard files which are made available to you when your program starts. These are standard input (usually the keyboard), standard output (buffered output to the monitor or display), and standard error (unbuffered output to the screen). (The "buffered" or "unbuffered" output refers to that third argument to open()). These files are named stdin, stdout, and stderr and take after their names from the C language.

Python makes these file handles available to you from the sys module. Once we import sys, we have access to these files as sys.stdin, sys.stdout, and sys.stderr. The **print** statement normally outputs to sys.stdout while the raw\_input() built-in function receives its input from sys.stdin.

### Ex :-

### sys.stdout.write()

**import** sys

sys.stdout.write('Hello World!' + '\n)

### sys.stdin.readline()

**import** sys

sys.stdout.write('Enter a string: ')

aString = sys.stdin.readline() sys.stdout.write(aString)

## 5) . Command-line Arguments

The sys module also provides access to any *command-line arguments* via the sys.argv. Command-line arguments are those arguments given to the program in addition to the script name on invocation. Historically, of course, these arguments are so named because they are given on the command-line along with the program name in a text-based environment like a Unix- or DOS-shell. However, in an IDE or GUI environment, this would not be the case. Most IDEs provide a separate window with which to enter

"command-line arguments." These, in turn, will be passed into the program to start the application from the command-line.

sys.argv is the list of command-line arguments

len(sys.argv) *is the number of command-line arguments (a.k.a. argc)*

For example to create a small test program called argv.py with the following lines:

**import** sys

**print** 'you entered', len(sys.argv), 'arguments…'

**print** 'they were:', str(sys.argv)

Here is an example invocation and output of this script:

% argv.py 76 tales 85 hawk y if we enter 5 arguments…

they were: ['argv.py', '76', 'tales', '85', 'hawk']

Command-line arguments allow a programmer or administrator to start a program perhaps with different behavioral characteristics. Much of the time, this execution takes place in the middle of the night and run as a batch job without human interaction. Command-line arguments and program options enable this type of functionality. As long as there are computers sitting idle at night and plenty of work to be done, there will always be a need to run programs in the background on our very expensive "calculators."

Python features a getopt module that helps you parse command-line options and arguments.

## 6). File System, File Execution, Persistent Storage Modules, Related Modules.

## File System :-

Access to the file system occurs mostly through the Python os module. This module serves as the primary interface to the operating system facilities and services from Python. The os module is actually a front-end to the real module that is loaded, a module that is clearly operating system-dependent. This "real" module may be one of the following: posix (Unix), nt (Windows), mac (Macintosh), dos (DOS), os2 (OS/2), etc.

In addition to managing processes and the process execution environment, the os module performs most of the major file system operations that the application developer may wish to take advantage of. These features include removing and renaming files, traversing the directory tree, and managing file accessibility. The following table lists some of the more common file or directory operations available from the os module.

|  |  |
| --- | --- |
| **Table : os Module File/Directory Access Functions** | |
| **os *Module File/Directory Function*** | ***Operation*** |
| ***File Processing*** |  |
| remove()/unlink() | delete file |
| rename() | rename file |
| \*stat()[a] | return file statistics |
| symlink() | create symbolic link |
| utime() | update timestamp |
| ***Directories/Folders*** |  |
| chdir() | change working directory |
| listdir() | list files in directory |
| getcwd() | return current working directory |
| mkdir()/makedirs() | create directory(ies) |

|  |  |
| --- | --- |
| rmdir()/removedirs() | remove directory(ies) |
| ***Access/Permissions (available only on Unix***  ***or Windows )*** |  |
| access() | verify permission modes |
| chmod() | change permission modes |
| umask() | set default permission modes |

## File Execution :-

Whether we want to simply run an operating system command, invoke a binary executable, or another type of script (perhaps a shell script, Perl, or Tcl/Tk), this involves executing another file somewhere else on the system. Even running other Python code may call for starting up another Python interpreter, although that may not always be the case.

## Persistent Storage Modules :-

Python provides a variety of modules which implement minimal persistent storage. One set of modules (marshal and pickle) allows for pickling of Python objects. Pickling is the process whereby objects more complex than primitive types can be converted to a binary set of bytes that can be stored or transmitted across the network, then be converted back to their original object forms. Pickling is also known as flattening, serializing, or marshalling. Another set of modules (dbhash/bsddb, dbm, gdbm, dumbdbm) and their "manager" (anydbm) can provide persistent storage of Python strings only. The last module (shelve) can do both.

## Related Modules :-

There are plenty of other modules related to files and input/output, all of which work on most of the major platforms. The following table lists some of the file-related modules.

|  |  |
| --- | --- |
| **Table : Related File Modules** | |
| ***Module(s)*** | ***Contents*** |
| Fileinput | iterates over lines of multiple input text files |
| Getopt | provides command-line argument parsing/manipulation |
| glob/fnmatch | provides Unix-style wildcard character matching |
| gzip/zlib/zipfile[a] | allows file access to include automatic de/compression |
| Shutil | offers high-level file access functionality |
| c/StringIO | implements file-like interface on top of string objects |
| Tempfile | generates temporary file names or files |

**CHAPTER-2 : EXCEPTIONS**

**1). Exceptions in Python.**

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program. Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. An exception is a Python object that represents an error.Some of the examples of exceptions are given below :

**NameError: attempt to access an undeclared variable**

>>> foo

Traceback (innermost last):

File "<interactive input>", line 0, in ?

NameError: foo

NameError indicates access to an uninitialized variable. The offending identifier was not found in the Python interpreter's symbol table.

**ZeroDivisionError: division by any numeric zero**

>>> 12.4/0.0

Traceback (innermost last):

File "<interactive input>", line 0, in ?

ZeroDivisionError: float division

The example above used floats, but in general, any numeric division-by-zero will result in a ZeroDivisionError exception.

**SyntaxError: Python interpreter syntax error**

>>> **for**

File "<string>", line 1

**for**

**^**

SyntaxError: invalid syntax

SyntaxError exceptions are the only ones which do not occur at run-time. They indicate an improperly constructed piece of Python code which cannot execute until corrected.

**IndexError: request for an out-of-range index for sequence**

>>> aList = []

>>> aList[0]

Traceback (innermost last):

File "<stdin>", line 1, in ?

IndexError: list index out of range

IndexError is raised when attempting to access an index which is outside the valid range of a sequence.

**KeyError: request for a non-existent dictionary key**

>>> aDict = {'host': 'earth', 'port': 80}

>>> **print** aDict['server']

Traceback (innermost last):

File "<stdin>", line 1, in ?

KeyError: server

Mapping types such as dictionaries depend on keys to access data values. Such values are not retrieved if an incorrect/nonexistent key is requested. In this case, a KeyError is raised to indicate such an incident has occurred.

**IOError: input/output error**

>>> f = open("blah")

Traceback (innermost last):

File "<interactive input>", line 1, in ?

IOError: [Errno 2] No such file or directory: 'blah'

Attempting to open a non-existent disk file is one example of an operating system input/output (I/O) error. Any type of I/O error raises an IOError exception.

**2). Built-in (or) Standard Exceptions in Python**

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

**Common Exceptions**:-

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

## The problem without handling exceptions:-

Suppose we have two variables **a** and **b**, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a **ZeroDivision** exception. Consider the following example.

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b

print("a/b = %d" %c)

#other code:

print("Hi I am other part of the program)

**OUTPUT**:-

Enter a:10

Enter b:0

Traceback (most recent call last):

File "exception-test.py", line 3, in <module>

c = a/b;

ZeroDivisionError: division by zero.

The above program is syntactically correct, but it through the error because of unusual input. That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution. That's why an exception-handling plays an essential role in handling these unexpected exceptions.

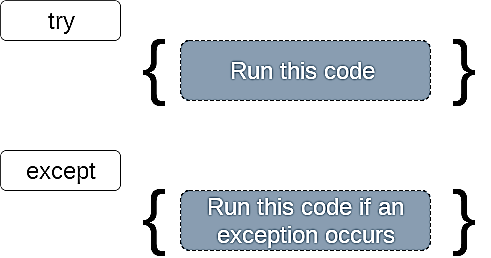
**3). Detecting and Handling Exceptions**

Exceptions can be detected by incorporating them as part of a **try** statement. Any code suite of a **try** statement will be monitored for exceptions. There are two main forms of the **try** statement: **try-except** and **try-finally.** These statements are mutually exclusive, meaning that we pick only one of them. A **try** statement is either accompanied by one or more **except** clauses or exactly one **finally** clause.

**try-except** statements allow one to detect and handle exceptions. There is even an optional else clause for situations where code needs to run only when no exceptions are detected. Meanwhile, **try-finally** statements allow only for detection and processing of any obligatory clean-up (whether or not exceptions occur), but otherwise has no facility in dealing with exceptions.

**The try-expect statement:-**

If the Python program contains suspicious code that may throw the exception, we must place that code in the try block. The try block must be followed with the except statement, which contains a block of code that will be executed if there is some exception in the try block.



**Syntax:-**

try:

#block of code

except Exception1:

#block of code

except Exception2:

#block of code

#other code

**Consider the following example.**

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b

except:

print("Can't divide with zero")

Output:

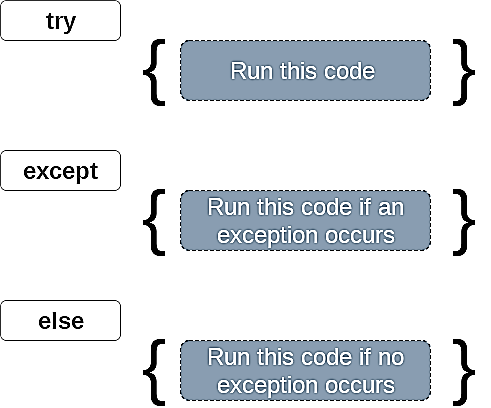
Enter a:10

Enter b:0

Can't divide with zero

We can also use the else statement with the **try-except** statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

The syntax to use the **else** statement with the try-except statement is given below.



try:

#block of code

except Exception1:

#block of code

else:

#this code executes if no except block is executed

**Consider the following program.**

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b

print("a/b = %d"%c)

# Using Exception with except statement. If we print(Exception) it will return exception class

except Exception:

print("can't divide by zero")

print(Exception)

else:

print("Hi I am else block")

**Output:-**

Enter a:10

Enter b:0

can't divide by zero

<class 'Exception'>

**Example :- (File not Found Exception)**

try:

#this will throw an exception if the file doesn't exist.

fileptr = open("file.txt","r")

except IOError:

print("File not found")

else:

print("The file opened successfully")

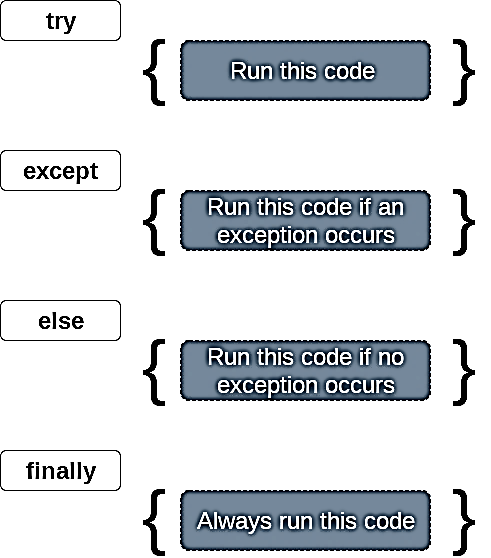
fileptr.close()

**Output :-**

File not found

* **The try...finally block**

Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

We can use the finally block with the try block in which we can place the necessary code, which must be executed before the try statement throws an exception. The syntax to use the finally block is given below.

try:

# block of code

# this may throw an exception

finally:

# block of code

# this will always be executed

**Example:-**

try:

fileptr = open("file2.txt","r")

try:

fileptr.write("Hi I am good")

finally:

fileptr.close()

print("file closed")

except:

print("Error")

**Output:-**

file closed

Error

**4). Raising Exceptions**

An exception can be raised forcefully by using the **raise** clause in Python. It is useful in in that scenario where we need to raise an exception to stop the execution of the program.

The syntax to use the raise statement is given below.

**raise Exception\_class,<value>**

**Example 1:**

try:

age = int(input("Enter the age:"))

if(age<18):

raise ValueError

else:

print("the age is valid")

except ValueError:

print("The age is not valid")

**Output:**

Enter the age:17

The age is not valid

**Example 2:**  Raise the exception with message

try:

num = int(input("Enter a positive integer: "))

if(num <= 0):

# we can pass the message in the raise statement

raise ValueError("That is a negative number!")

except ValueError as e:

print(e)

**Output:**

Enter a positive integer: -5

That is a negative number!

**Example 3:**

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

if b is 0:

raise ArithmeticError

else:

print("a/b = ",a/b)

except ArithmeticError:

print("The value of b can't be 0")

**Output:**

Enter a:10

Enter b:0

The value of b can't be 0

**5). Exceptions as Strings**

Prior to Python 1.5, standard exceptions were implemented as strings. However, this became limiting in that it did not allow for exceptions to have any relationships to each other. With the advent of exception classes, this is no longer the case. As of 1.5, all standard exceptions are now classes. It is still possible for programmers to generate their own exceptions as strings, but we recommend using exception classes from now on. For backwards compatibility, it is possible to revert to string-based exceptions. Starting the Python interpreter with the command-line option -X will provide you with the standard exceptions as strings. This feature will be obsoleted beginning with Python 1.6. If we must use string exceptions, we will now show how to do it right. The following piece of code may or may not work:

# this may not work… risky!

**try:**

:

**raise** 'myexception'

:

**except** 'myexception'

suite\_to\_handle\_my\_string\_exception

**except:**

suite\_for\_other\_exceptions

The reason why the above code may not work is because exceptions are based on object identity as opposed to object value. There are two different string objects above, both with the same value. To rectify the potential problem, create a static string object with which to use:

# this is a little bit better

myexception = 'myexception'

**try:**

:

**raise** myexception

:

**except** myexception:

suite\_to\_handle\_my\_string\_exception

**except:**

suite\_for\_other\_exceptions

With this update, the same string object is used. However, if you are going to use this

code, you might as well use an exception class. Substitute the myexception assignment

above with:

# this is the best choice

**class** MyException(Exception):

**pass**

:

**try:**

:

**raise** MyException

:

**except** MyException:

suite\_to\_handle\_my\_string\_exception

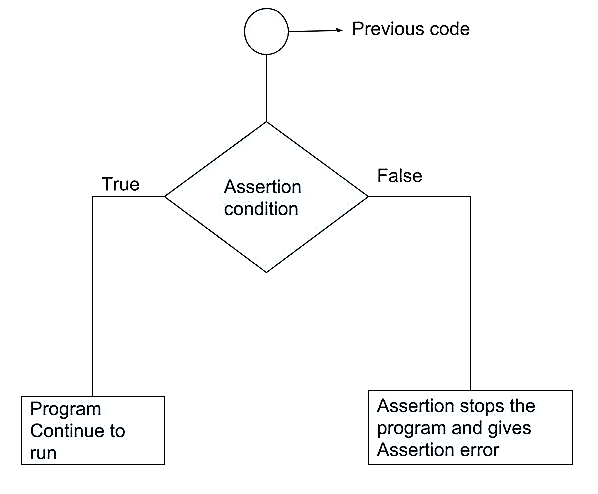
**except:**

suite\_for\_other\_exceptions

So you see, there really is no reason not to use exception classes from now on when creating your own exceptions. Be careful, however, because you may end up using an external module which may still have exceptions implemented as strings.

**6). Assertions in Python.**

Assertions are statements that assert or state a fact confidently in the program. For example, while writing a division function, we're confident the divisor shouldn't be zero, we assert divisor is not equal to zero. Assertions are simply boolean expressions that check if the conditions return true or not. If it is true, the program does nothing and moves to the next line of code. However, if it is false, the program stops and throws an error. It is also a debugging tool as it halts the program as soon as an error occurs and displays it.



* **Python assert Statement**

Python has built-in assert statement to use assertion condition in the program. Assert statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an Assertion Error.

**Syntax for using Assert in Pyhton:**

assert <condition>

assert <condition>,<error message>

In Python we can use assert statement in two ways as mentioned above.

1).assert statement has a condition and if the condition is not satisfied the program will stop and give Assertion Error.

2).assert statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives Assertion Error along with the error message.

Example : Using assert with error message

def avg(marks):

assert len(marks) != 0,"List is empty."

return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]

print("Average of mark2:",avg(mark2))

mark1 = []

print("Average of mark1:",avg(mark1))

When we run the above program, the output will be:

Average of mark2: 78.0

Assertion Error: List is empty.

We passed a non-empty list mark2 and also an empty list mark1 to the avg() function and we got output for mark2 list but after that we got an error AssertionError: List is empty. The assert condition was satisfied by the mark2 list and program to continue to run. However, mark1 doesn't satisfy the condition and gives an Assertion Error.

**7). Exceptions and the sys Module**

An alternative way of obtaining exception information is by accessing the **exc\_info()** function in the **sys** module. This function provides a 3-tuple of information, more than what we can achieve by simply using only the exception argument. Consider the following example what we get using **sys.exc\_info():**

try:

float('abc123')

except:

import sys

exc\_tuple = sys.exc\_info()

print (exc\_tuple)

(<class exceptions.ValueError at f9838>, <exceptions.ValueError

instance at 122fa8>,

<traceback object at 10de18>)

>>> for eachItem in exc\_tuple:

print (eachItem)

exceptions.ValueError

invalid literal for float(): abc123

<traceback object at 10de18>

What we get from sys.exc\_info() in a tuple are:

exception class object

(this) exception class instance object

traceback object

The first two items we are familiar with: the actual exception class and this particular exception's instance. The third item, a traceback object, is new. This object provides the execution context of where the exception occurred. It contains information such as the execution frame of the code that was running and the line number where the exception occurred.

In older versions of Python, these three values were available in the sys module as sys.exc\_type, sys.exc\_value, and sys.exc\_traceback. Unfortunately, these three are global variables and not thread-safe.

**CHAPTER-3 : MODULES**

**1).Modules in Python**

A module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

**Example**

The Python code for a module named **aname** normally resides in a file named **aname.py**. Consider an example of a simple module, support.py

def print\_func( par ):

print ("Hello : ", par)

return

**Loading the module in python code :-**

We need to load the module in the python code to use its functionality. Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement

* **The import Statement :-**

We can use any Python source file as a module by executing an **import** statement in some other Python source file. The import has the following syntax:

**import module1[, module2[,... moduleN]**

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, we need to put the following command at the top of the script.

**import support**

# Now we can call defined function that module as follows

support.print\_func("Hai")

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

Hello : Hai

A module is loaded only once, regardless of the number of times it is imported.

## The from-import statement :-

## Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

## from < module-name> import <name 1>, <name 2>..,<name n>

## Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

## calculation.py:

## #place the code in the calculation.py

## def summation(a,b):

## return a+b

## def multiplication(a,b):

## return a\*b;

## def divide(a,b):

## return a/b;

## Main.py:

## from calculation import summation

## #it will import only the summation() from calculation.py

## a = int(input("Enter the first number"))

## b = int(input("Enter the second number"))

## print("Sum = ",summation(a,b)) #we do not need to specify the module name while accessing summation()

## Output:

## Enter the first number10

## Enter the second number20

## Sum = 30

**2).Namespaces and Scoping**

Variables are names (identifiers) that map to objects. A namespace is a dictionary of variable names (keys) and their corresponding objects (values). A Python statement can access variables in a local namespace and in the global namespace. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions. Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local. Therefore, in order to assign a value to a global variable within a function, we first use the global statement.

The statement global VarName tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable Money in the global namespace. Within the function Money, we assign Money a value, therefore Python assumes Money as a local variable. However, we accessed the value of the local variable Money before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

Money = 2000

def AddMoney():

Money = Money + 1

print (Money)

AddMoney()

print (Money)

**3). Module Built-In Functions**

The following are the some of the module built-in functions in Python.

**The dir( ) Function :-**

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example:

# Import built-in module math

import math

content = dir(math)

print (content)

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

['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp','fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log','log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']

Here, the special string variable \_\_name\_\_ is the module's name, and \_\_file\_\_ is the filename from which the module was loaded.

**The globals() and locals() Functions :-**

The globals() and locals() functions can be used to return the names in the global and local namespaces depending on the location from where they are called.If locals() is called from within a function, it will return all the names that can be accessed locally from that function.If globals() is called from within a function, it will return all the names that can be accessed globally from that function.The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

**The reload() Function :-**

When the module is imported into a script, the code in the top-level portion of a module is executed only once. Therefore, if we want to reexecute the top-level code in a module, we can use the reload() function. The reload() function imports a previously imported module again. The syntax of the reload() function is as follows :

reload(module\_name)

Here, module\_name is the name of the module we want to reload and not the string containing the module name. For example, to reload hello module, do the following :

reload(hello)

**4). Python Module Attributes: name, doc, file, dict**

Python module has its attributes that describes it. Attributes perform some tasks or contain some information about the module. Some of the important attributes are explained below:

**\_\_name\_\_ Attribute :-**

The \_\_name\_\_ attribute returns the name of the module. By default, the name of the file (excluding the extension .py) is the value of \_\_name\_\_attribute.

Example: \_\_name\_\_ Attribute

>>> import math

>>> math.\_\_name\_\_

'math'

In the same way, it gives the name of the custom module.

Example: \_\_name\_\_ Attribute

>>> hello.\_\_name\_\_

'hello'

However, this can be modified by assigning different strings to this attribute. Change hello.py as shown below.

Example: Set \_\_name\_\_

def SayHello(name):

print ("Hi {}! How are you?".format(name))

\_\_name\_\_="SayHello"

And check the \_\_name\_\_ attribute now.

>>> import hello

>>> hello.\_\_name\_\_

'SayHello'

The value of the \_\_name\_\_ attribute is \_\_main\_\_ on the Python interactive shell.

>>> \_\_name\_\_

'\_\_main\_\_'

When we run any Python script (i.e. a module), its \_\_name\_\_ attribute is also set to \_\_main\_\_. For example, create the following welcome.py in IDLE.

Example: welcome.py Copy

print("\_\_name\_\_ = ", \_\_name\_\_)

Run the above welcome.py in IDLE by pressing F5. Then python produces the following result.

Output in IDLE:

>>> \_\_name\_\_ = \_\_main\_\_

However, when this module is imported, its \_\_name\_\_ is set to its filename. Now, import the welcome module in the new file test.py with the following content.

Example: test.py Copy

import welcome

print("\_\_name\_\_ = ", \_\_name\_\_)

Now run the test.py in IDLE by pressing F5. The \_\_name\_\_ attribute is now "welcome".

Example: test.py Copy

\_\_name\_\_ = welcome

This attribute allows a Python script to be used as an executable or as a module.

**\_\_doc\_\_ Attribute :-**

The \_\_doc\_\_ attribute denotes the documentation string (docstring) line written in a module code.

Example: Copy

>>> import math

>>> math.\_\_doc\_\_

'This module is always available. It provides access to the mathematical functions defined by the C standard.'

Consider the the following script is saved as test.py module.

test.py Copy

"""This is docstring of test module"""

def SayHello(name):

print ("Hi {}! How are you?".format(name))

return

The \_\_doc\_\_ attribute will return a string defined at the beginning of the module code.

Example: Copy

>>> import test

>>> test.\_\_doc\_\_

'This is docstring of test module'

**\_\_file\_\_ Attribute :-**

\_\_file\_\_ is an optional attribute which holds the name and path of the module file from which it is loaded.

Example: \_\_file\_\_ Attribute Copy

>>> import io

>>> io.\_\_file\_\_

'C:\\python37\\lib\\io.py'

**\_\_dict\_\_ Attribute :-**

The \_\_dict\_\_ attribute will return a dictionary object of module attributes, functions and other definitions and their respective values.

Example: \_\_dict\_\_ Attribute Copy

>>> import math

>>> math.\_\_dict\_\_

{'\_\_name\_\_': 'math', '\_\_doc\_\_': 'This module is always available. It provides access to the\nmathematical functions defined by the C standard.',

**5). Packages in Python**

We organize a large number of files in different folders and subfolders based on some criteria, so that we can find and manage them easily. In the same way, a package in Python takes the concept of the modular approach to next logical level. As we know, a [module](https://www.tutorialsteacher.com/python/python-module) can contain multiple objects, such as classes, functions, etc. A package can contain one or more relevant modules. Physically, a package is actually a folder containing one or more module files.

To create a package named mypackage, using the following steps:

* Create a new folder named **D:\MyApp.**
* Inside MyApp, create a subfolder with the name 'mypackage'.
* Create an empty \_\_init\_\_.py file in the mypackage folder.
* Using a Python-aware editor like IDLE, create modules greet.py and functions.py with the following code:

greet.py

def SayHello(name):

print("Hello ", name)

functions.py

def sum(x,y):

return x+y

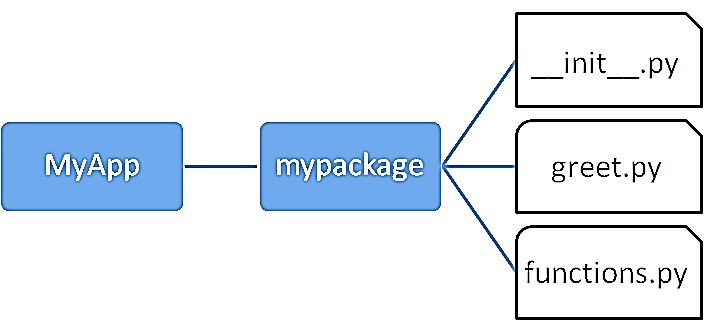
def average(x,y):

return (x+y)/2

def power(x,y):

return x\*\*y

We have created our package called mypackage. The following is a folder structure:



Importing a Module from a Package

Now, to test our package, navigate the command prompt to the MyApp folder and invoke the Python prompt from there.

D:\MyApp>python

Import the functions module from the mypackage package and call its power() function.

>>> from mypackage import functions

>>> functions.power(3,2)

9

It is also possible to import specific functions from a module in the package.

>>> from mypackage.functions import sum

>>> sum(10,20)

30

>>> average(10,12)

Traceback (most recent call last):

File "<pyshell#13>", line 1, in <module>

NameError: name 'average' is not defined.

**UNIT-III**

**CHAPTER-1 : REGULAR EXPRESSIONS**

**1). Python Regular Expressions - Special Symbols and Characters**

The regular expressions can be defined as the sequence of characters which are used to search for a pattern in a string. The module **re** provides the support to use **regex** in the python program. The **re** module throws an exception if there is some error while using the regular expression.The **re** module must be imported to use the **regex** functionalities in python.

import re

**Regex Functions :-**

The following regex functions are used in the python.

**Function** **Description**

match This method matches the regex pattern in the string with the optional flag. It returns

true if a match is found in the string otherwise it returns false.

search This method returns the match object if there is a match found in the string.

findall It returns a list that contains all the matches of a pattern in the string.

split Returns a list in which the string has been split in each match.

sub Replace one or many matches in the string.

**Forming a regular expression:-**

A regular expression can be formed by using the mix of **meta-characters, special sequences, and sets.**

**Meta-Characters:-**Metacharacter is a character with the specified meaning.

**Metacharacter** **Description** **Example**

[] It represents the set of characters. "[a-z]"

\ It represents the special sequence. "\r"

. It signals that any character is present at some specific place. "Ja.v."

^ It represents the pattern present at the beginning of the string. "^Java"

$ It represents the pattern present at the end of the string. "point$"

\* It represents zero or more occurrences of a pattern in the string. "hello\*"

+ It represents one or more occurrences of a pattern in the string. "hello+"

{} The specified number of occurrences of a pattern the string. "java{2}"

| It represents either this or that character is present. "java|point"

() Capture and group

**Special Sequences:-**Special sequences are the sequences containing \ followed by one of the characters.

**Character** **Description**

\A It returns a match if the specified characters are present at the beginning of the string.

\b It returns a match if the specified characters are present at the beginning or the end of the string.

\B It returns a match if the specified characters are present at the beginning of the string but not at the end.

\d It returns a match if the string contains digits [0-9].

\D It returns a match if the string doesn't contain the digits [0-9].

\s It returns a match if the string contains any white space character.

\S It returns a match if the string doesn't contain any white space character.

\w It returns a match if the string contains any word characters.

\W It returns a match if the string doesn't contain any word.

\Z Returns a match if the specified characters are at the end of the string.

**Sets:-**A set is a group of characters given inside a pair of square brackets. It represents the special meaning.

**Set** **Description**

[arn] Returns a match if the string contains any of the specified characters in the set.

[a-n] Returns a match if the string contains any of the characters between a to n.

[^arn] Returns a match if the string contains the characters except a, r, and n.

[0123] Returns a match if the string contains any of the specified digits.

[0-9] Returns a match if the string contains any digit between 0 and 9.

[0-5][0-9] Returns a match if the string contains any digit between 00 and 59.

[a-zA-Z] Returns a match if the string contains any alphabet (lower-case or upper-case).

**The findall() function :-**

This method returns a list containing a list of all matches of a pattern within the string. It returns the patterns in the order they are found. If there are no matches, then an empty list is returned. Consider the following example:

import re

str = "How are you. How is everything"

matches = re.findall("How", str)

print(matches)

Output:

['How', 'How']

**The match object :-**

The match object contains the information about the search and the output. If there is no match found, the None object is returned.

**Example :**

import re

str = "How are you. How is everything"

matches = re.search("How", str)

print(type(matches))

print(matches) #matches is the search object

**Output:**

<class '\_sre.SRE\_Match'>

<\_sre.SRE\_Match object; span=(0, 3), match='How'>

**The Match object methods :-**The following methods are associated with the Match object.

**span():-** It returns the tuple containing the starting and end position of the match.

**string()**:- It returns a string passed into the function.

**group()**:- The part of the string is returned where the match is found.

**Example :**

import re

str = "How are you. How is everything"

matches = re.search("How", str)

print(matches.span())

print(matches.group())

print(matches.string)

**Output:**

(0, 3)

How

How are you. How is everything

**2).Multi Threading-Introduction-advantages-Processes and Threads**

A thread is the smallest unit of a program or process executed independently or scheduled by the Operating System. In the computer system, an Operating System achieves multitasking by dividing the process into threads. A thread is a lightweight process that ensures the execution of the process separately on the system. In Python 3, when multiple processors are running on a program, each processor runs simultaneously to execute its tasks separately.

**Python Multithreading :-**

Multithreading is a threading technique in Python programming to run multiple threads concurrently by rapidly switching between threads with a CPU help (called context switching). Besides, it allows sharing of its data space with the main threads inside a process that share information and communication with other threads easier than individual processes. Multithreading aims to perform multiple tasks simultaneously, which increases performance and speed and improves of the application.

The Python **Global Interpreter Lock (GIL)** allows running a single thread at a time, even the machine has multiple processors.

### Benefits of Multithreading in Python :-

Following are the benefits to create a multithreaded application in Python :

1. It ensures effective utilization of computer system resources.
2. Multithreaded applications are more responsive.
3. It shares resources and its state with sub-threads (child) which makes it more economical.
4. It makes the multiprocessor architecture more effective due to similarity.
5. It saves time by executing multiple threads at the same time.
6. The system does not require too much memory to store multiple threads.

### When to use Multithreading in Python :-

It is a very useful technique for time-saving and improving the performance of an application. Multithreading allows the programmer to divide application **tasks** into sub-tasks and simultaneously run them in a program. It allows threads to communicate and share resources such as files, data, and memory to the same processor. Furthermore, it increases the user's responsiveness to continue running a program even if a part of the application is the length or blocked.

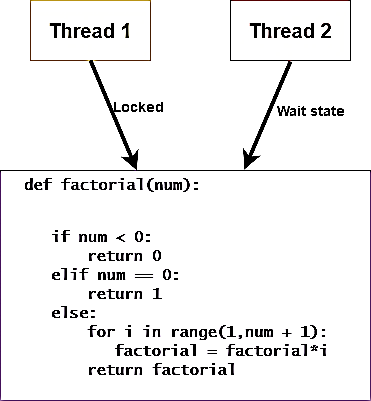
**3). Global Interpreter Lock (GIL)**

A global interpreter lock (GIL) is a mechanism to apply a global lock on an interpreter. It is used in computer-language interpreters to synchronize and manage the execution of threads so that only one native thread (scheduled by the operating system) can execute at a time.

In a scenario where we have multiple threads, what can happen is that both the thread might try to acquire the memory at the same time, and as a result of which they would overwrite the data in the memory. Hence, arises a need to have a mechanism that could help prevent this phenomenon.

Some popular interpreters that have GIL are CPython and Ruby MRI. As we know that Python is an interpreted language, it has various distributions like CPython, Jython, IronPython. Out of these, GIL is supported only in CPython, and it is also the most widely used implementation of Python. CPython has been developed in both C and Python language primarily to support and work with applications that have a lot of C language underneath the hood.

Even if the processor has multiple cores, a global interpreter will allow only one thread to be executed at a time. This is because, when a thread starts running, it acquires the global interpreter lock. When it waits for any I/O operation ( reading/writing data from/to disk ) or a CPU bound operation ( vector/matrix multiplication ), it releases the lock so that other threads of that process can run. Hence, it prevents us from running the other threads at the same time.



In the above diagram. there is a factorial function and two threads 1 and 2, thread 1 is in the locked state while thread 2 is in a wait state. This means that only one of the threads is able to access the function. Now, let's assume that the factorial function takes 2 seconds to complete. Then, in an ideal case, both the threads should be able to finish the execution in 2 seconds. However, this is not the case in Python, both the threads will run serially and not parallel to each other.

The threads 1 and 2 calling the factorial function may take twice as much time as a single thread calling the function twice. This also tells that the memory manager governed by the interpreter is not thread-safe, which means that multiple threads fail to access the same shared data simultaneously.

**4). Achieving multithreading in Python-Thread Module and Threading Module**

There are two main modules of multithreading used to handle threads in Python

* **The thread module**
* **The threading module**

**The Thread module :-**

It is started with Python 3, designated as obsolete, and can only be accessed with \_thread that supports backward compatibility. The Syntax is as follows :

thread.start\_new\_thread ( function\_name, args[, kwargs] )

To implement the thread module in Python, we need to import a thread module and then define a function that performs some action by setting the target with a variable.

Thread.py

import thread # import the thread module

import time # import time module

def cal\_sqre(num): # define the cal\_sqre function

print(" Calculate the square root of the given number")

for n in num:

time.sleep(0.3) # at each iteration it waits for 0.3 time

print(' Square is : ', n \* n)

def cal\_cube(num): # define the cal\_cube() function

print(" Calculate the cube of the given number")

for n in num:

time.sleep(0.3) # at each iteration it waits for 0.3 time

print(" Cube is : ", n \* n \*n)

arr = [4, 5, 6, 7, 2] # given array

t1 = time.time() # get total time to execute the functions

cal\_sqre(arr) # call cal\_sqre() function

cal\_cube(arr) # call cal\_cube() function

print(" Total time taken by threads is :", time.time() - t1) # print the total time

Output:

Calculate the square root of the given number

Square is: 16

Square is: 25

Square is: 36

Square is: 49

Square is: 4

Calculate the cube of the given number

Cube is: 64

Cube is: 125

Cube is: 216

Cube is: 343

Cube is: 8

Total time taken by threads is: 3.005793809890747

**The Threading Module :-**

The threading module is a high-level implementation of multithreading used to deploy an application in Python. To use multithreading, we need to import the threading module in Python Program

* **Thread Class Methods**

**Methods Description**

start() A start() method is used to initiate the activity of a thread. And it calls only once for each thread so that the execution of the thread can begin.

run() A run() method is used to define a thread's activity and can be overridden by a class that extends the threads class.

join() A join() method is used to block the execution of another code until the thread terminates.

The following steps are used to implement the threading module in Python Multithreading:

**Import the threading module :-** Create a new thread by importing the threading module, as shown.

Syntax:-

**import threading**

A threading module is made up of a Thread class, which is instantiated to create a Python thread.

**Declaration of the thread parameters:-** It contains the target function, argument, and kwargs as the parameter in the Thread() class.

**Target:** It defines the function name that is executed by the thread.

**Args:** It defines the arguments that are passed to the target function name.

**Example :**

import threading

def print\_hello(n):

print("Hello, how old are you ", n)

t1 = threading.Thread( target = print\_hello, args =(18, ))

In the above code, we invoked the print\_hello() function as the target parameter. The print\_hello() contains one parameter n, which passed to the args parameter.

**Start a new thread**: -To start a thread in Python multithreading, call the thread class's object. The start() method can be called once for each thread object; otherwise, it throws an exception error.

**Example:**

t1.start()

t2.start()

**Join method**:- It is a join() method used in the thread class to halt the main thread's execution and waits till the complete execution of the thread object. When the thread object is completed, it starts the execution of the main thread in Python.

**Joinmethod.py**

import threading

def print\_hello(n):

Print("Hello, how old are you? ", n)

T1 = threading.Thread( target = print\_hello, args = (20, ))

T1.start()

T1.join()

Print("Thank you")

**Output:-**

Hello, how old are you? 20

Thank you

When the above program is executed, the join() method halts the execution of the main thread and waits until the thread t1 is completely executed. Once the t1 is successfully executed, the main thread starts its execution.

If we do not use the join() method, the interpreter can execute any print statement inside the Python program. Generally, it executes the first print statement because the interpreter executes the lines of codes from the program's start.

**CHAPTER-2 : GUI PROGRAMMING**

**1). Introduction to GUI Programming**

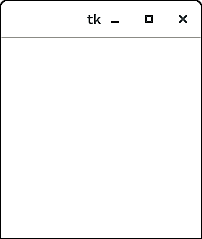
Python provides various options for developing graphical user interfaces (GUIs). Most important are listed below.

* **Tkinter** − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python.
* **wxPython** − This is an open-source Python interface for wxWindows [http://wxpython.org](http://wxpython.org/).
* **JPython** − JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine [http://www.jython.org](http://www.jython.org/).

**2). Tkinter Programming**

Python provides the standard library Tkinter for creating the graphical user interface for desktop based applications.

Developing desktop based applications with python Tkinter is not a complex task. An empty Tkinter top-level window can be created by using the following steps.

1. import the Tkinter module.
2. Create the main application window.
3. Add the widgets like labels, buttons, frames, etc. to the window.
4. Call the main event loop so that the actions can take place on the user's computer screen.

## Example :

## from tkinter import \*

## #creating the application main window.

## top = Tk()

## #Entering the event main loop

## top.mainloop()

## Output:

## 3) Tkinter widgets

## There are various widgets like button, canvas, checkbutton, entry, etc. that are used to build the python GUI applications.

|  |  |  |
| --- | --- | --- |
| **SN0** | **Widget** | **Description** |
| 1 | [Button](https://www.javatpoint.com/python-tkinter-button) | The Button is used to add various kinds of buttons to the python application. |
| 2 | [Canvas](https://www.javatpoint.com/python-tkinter-canvas) | The canvas widget is used to draw the canvas on the window. |
| 3 | [Checkbutton](https://www.javatpoint.com/python-tkinter-checkbutton) | The Checkbutton is used to display the CheckButton on the window. |
| 4 | [Entry](https://www.javatpoint.com/python-tkinter-entry) | The entry widget is used to display the single-line text field to the user. It is commonly used to accept user values. |
| 5 | [Frame](https://www.javatpoint.com/python-tkinter-frame) | It can be defined as a container to which, another widget can be added and organized. |
| 6 | [Label](https://www.javatpoint.com/python-tkinter-label) | A label is a text used to display some message or information about the other widgets. |
| 7 | [ListBox](https://www.javatpoint.com/python-tkinter-listbox) | The ListBox widget is used to display a list of options to the user. |
| 8 | [Menubutton](https://www.javatpoint.com/python-tkinter-menubutton) | The Menubutton is used to display the menu items to the user. |
| 9 | [Menu](https://www.javatpoint.com/python-tkinter-menu) | It is used to add menu items to the user. |
| 10 | [Message](https://www.javatpoint.com/python-tkinter-message) | The Message widget is used to display the message-box to the user. |
| 11 | [Radiobutton](https://www.javatpoint.com/python-tkinter-radiobutton) | The Radiobutton is different from a checkbutton. Here, the user is provided with various options and the user can select only one option among them. |
| 12 | [Scale](https://www.javatpoint.com/python-tkinter-scale) | It is used to provide the slider to the user. |
| 13 | [Scrollbar](https://www.javatpoint.com/python-tkinter-scrollbar) | It provides the scrollbar to the user so that the user can scroll the window up and down. |
| 14 | [Text](https://www.javatpoint.com/python-tkinter-text) | It is different from Entry because it provides a multi-line text field to the user so that the user can write the text and edit the text inside it. |
| 14 | [Toplevel](https://www.javatpoint.com/python-tkinter-toplevel) | It is used to create a separate window container. |
| 15 | [Spinbox](https://www.javatpoint.com/python-tkinter-spinbox) | It is an entry widget used to select from options of values. |
| 16 | [PanedWindow](https://www.javatpoint.com/python-tkinter-panedwindow) | It is like a container widget that contains horizontal or vertical panes. |
| 17 | [LabelFrame](https://www.javatpoint.com/python-tkinter-labelframe) | A LabelFrame is a container widget that acts as the container |
| 18 | [MessageBox](https://www.javatpoint.com/python-tkinter-messagebox) | This module is used to display the message-box in the desktop based applications. |

## 4). Python Tkinter Geometry

## The Tkinter geometry specifies the method by using which, the widgets are represented on display. The python Tkinter provides the following geometry methods.

## The pack() method

## The grid() method

## The place() method

## Python Tkinter pack() method :-

## The pack() widget is used to organize widget in the block. The positions widgets added to the python application using the pack() method can be controlled by using the various options specified in the method call.

## However, the controls are less and widgets are generally added in the less organized manner.

## The syntax to use the pack() is given below.

## widget.pack(options)

A list of possible options that can be passed in pack() is given below.

* **expand:** If the expand is set to true, the widget expands to fill any space.
* **Fill:** By default, the fill is set to NONE. However, we can set it to X or Y to determine whether the widget contains any extra space.
* **size:** it represents the side of the parent to which the widget is to be placed on the window.

## Example :

## from tkinter import \*

## parent = Tk()

## redbutton = Button(parent, text = "Red", fg = "red")

## redbutton.pack( side = LEFT)

## greenbutton = Button(parent, text = "Black", fg = "black")

## greenbutton.pack( side = RIGHT )

## bluebutton = Button(parent, text = "Blue", fg = "blue")

## bluebutton.pack( side = TOP )

## blackbutton = Button(parent, text = "Green", fg = "red")

## blackbutton.pack( side = BOTTOM)

## parent.mainloop()

## ****Output:****

## Python Tkinter

## ****5). Python Tkinter grid() method****

## **The grid() geometry manager organizes the widgets in the tabular form. We can specify the rows and columns as the options in the method call. We can also specify the column span (width) or rowspan(height) of a widget.This is a more organized way to place the widgets to the python application. The syntax to use the grid() is given below :**

## **widget.grid(options)**

A list of possible options that can be passed inside the grid() method is given below.

* **Column**  
  The column number in which the widget is to be placed. The leftmost column is represented by 0.
* **Columnspan**  
  The width of the widget. It represents the number of columns up to which, the column is expanded.
* **ipadx,ipady**  
  It represents the number of pixels to pad the widget inside the widget's border.
* **padx,pady**  
  It represents the number of pixels to pad the widget outside the widget's border.
* **row**  
  The row number in which the widget is to be placed. The topmost row is represented by 0.
* **rowspan**  
  The height of the widget, i.e. the number of the row up to which the widget is expanded.
* **Sticky**  
  If the cell is larger than a widget, then sticky is used to specify the position of the widget inside the cell. It may be the concatenation of the sticky letters representing the position of the widget. It may be N, E, W, S, NE, NW, NS, EW, ES.

## ****Example :****

## **from tkinter import \***

## **parent = Tk()**

## **name = Label(parent,text = "Name").grid(row = 0, column = 0)**

## **e1 = Entry(parent).grid(row = 0, column = 1)**

## **password = Label(parent,text = "Password").grid(row = 1, column = 0)**

## **e2 = Entry(parent).grid(row = 1, column = 1)**

## **submit = Button(parent, text = "Submit").grid(row = 4, column = 0)**

## **parent.mainloop()**

## ****Output:****

## **Python Tkinter**

## **Python Tkinter place() method**

## **The place() geometry manager organizes the widgets to the specific x and y coordinates.**

## **Syntax**

## **widget.place(options)**

A list of possible options is given below.

* **Anchor:** It represents the exact position of the widget within the container. The default value (direction) is NW (the upper left corner)
* **bordermode:** The default value of the border type is INSIDE that refers to ignore the parent's inside the border. The other option is OUTSIDE.
* **height, width:** It refers to the height and width in pixels.
* **relheight, relwidth:** It is represented as the float between 0.0 and 1.0 indicating the fraction of the parent's height and width.
* **relx, rely:** It is represented as the float between 0.0 and 1.0 that is the offset in the horizontal and vertical direction.
* **x, y:** It refers to the horizontal and vertical offset in the pixels.

## ****Example :****

## **from tkinter import \***

## **top = Tk()**

## **top.geometry("400x250")**

## **name = Label(top, text = "Name").place(x = 30,y = 50)**

## **email = Label(top, text = "Email").place(x = 30, y = 90)**

## **password = Label(top, text = "Password").place(x = 30, y = 130)**

## **e1 = Entry(top).place(x = 80, y = 50)**

## **e2 = Entry(top).place(x = 80, y = 90)**

## **e3 = Entry(top).place(x = 95, y = 130)**

## **top.mainloop()**

## ****Output :****

## Python Tkinter

## UNIT-IV

## CHAPTER-1 : WEB PROGRAMMING

### 1). Web Surfing with Python: Creating Simple Web Clients

A web browser is only one type of Web client. Any application that makes a request for data from a Web server is considered a "client.". It is possible to create other clients that retrieve documents or data off the Internet. One important reason to do this is that a browser provides only limited capacity, i.e., it is used primarily for viewing and interacting with Web sites. A client program, on the other hand, has the ability to do more it can not only download data, but it can also store it, manipulate it, or perhaps even transmit it to another location or application.

Applications that use the **urllib** module to download or access information on the Web can be considered a simple Web client. Simple Web surfing involves using Web addresses called URLs (Uniform Resource Locators). Such addresses are used to locate a document on the Web or to call a CGI program to generate a document for the client. URLs are part of a larger set of identifiers known as URIs (Uniform Resource Identifiers). A URL uses the following format:

prot\_sch://net\_loc/path;params?query#frag

The following table describes each of the components.

|  | |
| --- | --- |
| **URL Component** | **Description** |
| prot\_sch | Network protocol or download scheme |
| net\_loc | Location of server (and perhaps user information) |
| Path | Slash ( / ) delimited path to file or CGI application |
| Params | Optional parameters |
| Query | Ampersand ( & ) delimited set of "key=value" pairs |
| Frag | Fragment to a specific anchor within document |

net\_loc can be broken down into several more components, some required, others optional. The net\_loc string looks like this:

user:passwd@host:port

These individual components are described in following Table

|  | |
| --- | --- |
| **net\_loc Component** | **Description** |
| User | User name or login |
| Passwd | User password |
| Host | Name or address of machine running Web server [required] |
| Port | Port number (if not 80, the default) |

The host name is the most important. The port number is necessary only if the Web server is running on a different port number from the default. User names and perhaps passwords are used only when making FTP connections.

Python supplies two different modules, each dealing with URLs in completely different functionality and capacities. One is urlparse, and the other is urllib.

#### urlparse Module

The urlparse module provides basic functionality with which to manipulate URL strings. These functions include urlparse(), urlunparse(), and urljoin().

**urlparse.urlparse() :-** urlparse() breaks up a URL string into some of the major components described above. It has the following syntax:

urlparse(urlstr, defProtSch=None, allowFrag=None)

* **urlparse.urlunparse() :-** urlunparse() does the exact opposite of urlparse(),it merges a 6-tuple (prot\_sch, net\_loc, path, params, query, frag)urltup, which could be the output of urlparse(), into a single URL string and returns it. The syntax of urlunparse() is as follows:

urlunparse(urltup)

##### **urlparse.urljoin() :-** The urljoin() function is useful in cases where many related URLs are needed, for example, the URLs for a set of pages to be generated for a Web site. The syntax for urljoin() is:

urljoin(baseurl, newurl, allowFrag=None)

urljoin() takes baseurl and joins its base path (net\_loc plus the full path up to, but not including, a file at the end) with newurl. For example:

The urllib module provides functions to download data from given URLs as well as encoding and decoding strings to make them suitable for including as part of valid URL strings. The functions include: urlopen(), urlretrieve(), quote(), unquote(), quote\_plus(), unquote\_plus(), and urlencode().

* **urllib.urlopen()**

urlopen() opens a Web connection to the given URL string and returns a file-like object. It has the following syntax:

urlopen(urlstr, postQueryData=None)

urlopen() opens the URL pointed to by urlstr. If no protocol or download scheme is given, or if a "file" scheme is passed in, urlopen() will open a local file.

| When a successful connection is made, urlopen() returns a file-like object as if the destination was a file opened in read mode. If our file object is f, for example, then our "handle" would support the expected read methods such as f.read(), f.readline(), f.readlines(), f.close(), and f.fileno(). | |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

##### **urllib.urlretrieve()**

Rather than reading from the URL like urlopen() does, urlretrieve() will simply download the entire HTML file located at urlstr to your local disk. It will store the downloaded data into localfile if given or a temporary file if not. If the file has already been copied from the Internet or if the file is local, no subsequent downloading will occur. Here is the syntax for urlretrieve():

urlretrieve(urlstr, localfile=None, downloadStatusHook=None)

Rather than reading from the URL like urlopen() does, urlretrieve() will simply download the entire HTML file located at urlstr to your local disk. It will store the downloaded data into localfile if given or a temporary file if not. If the file has already been copied from the Internet or if the file is local, no subsequent downloading will occur.

* **urllib.quote() and urllib.quote\_plus()**

The quote\*() functions take URL data and "encodes" them so that they are "fit" for inclusion as part of a URL string. In particular, certain special characters that are unprintable or cannot be part of valid URLs acceptable to a Web server must be converted. Both quote\*() functions have the following syntax:

quote(urldata, safe='/')

##### **urllib.unquote() and urllib.unquote\_plus()**

As you have probably guessed, the unquote\*() functions do the exact opposite of the quote\*() functionsthey convert all characters encoded in the "%xx" fashion to their ASCII equivalents. The syntax of unquote\*() is as follows:

unquote\*(urldata)

Calling unquote() will decode all URL-encoded characters in urldata and return the resulting string. unquote\_plus() will also convert plus signs back to space characters.

##### **urllib.urlencode()**

urlencode(), added to Python back in 1.5.2, takes a dictionary of key-value pairs and encodes them to be included as part of a query in a CGI request URL string. The pairs are in "key=value" format and are delimited by ampersands ( & ). Furthermore, the keys and their values are sent to quote\_plus() for proper encoding.

### 2). Advanced Web Clients in Python

Web browsers are basic Web clients. They are used primarily for searching and downloading documents from the Web. Advanced clients of the Web are those applications which do more than download single documents from the Internet.

One example of an advanced Web client is a crawler (a.k.a. spider, robot). These are programs which explore and download pages from the Internet for different reasons, some of which include:

* Indexing or cataloging into a large search engine such as Google, Alta Vista, or Yahoo!,
* Offline browsing—downloading documents onto a local hard disk and rearranging hyperlinks to create almost a mirror image for local browsing,
* Downloading and storing for historical or archival purposes, or
* Web page caching ...

**3). Python CGI (Common Gateway Interface)**

The word CGI is the acronyms of the **"Common Gateway Interface"**, which is used to define how to exchange information between the web server and a custom script. The Common Gateway Interface is a standard for external gateway programs to interface with the server, such as HTTP Servers.

In simple words, it is the collection of methods used to set up a dynamic interaction between the server, and the client application. When a client sends a request to the webserver, the CGI programs execute that particular request and send back the result to the webserver.

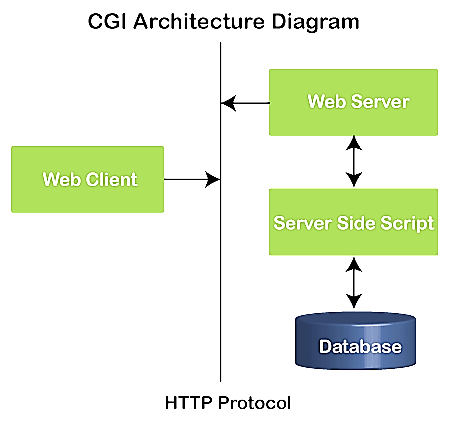
The users may submit the information in web browser by using HTML [<form>](https://www.javatpoint.com/html-form) or [<isindex>](https://www.javatpoint.com/html-isindex-tag) element. There is a server's special directory called cgi-bin, where cgi script is generally stored. When a client makes a request, the server adds the additional information to request. This additional information can be the hostname of the client, the query string, the requested URL, and so on. The webserver executes and sends the output to the web browser (or other client application).

**Example :**

1. print("<title> This is a CGI script output</title>")
2. print("<h1>This is my first CGI script</h1>")
3. print("Hello, Python!")

## Web Browsing :-

Before understanding the CGI concepts, we should know the internal process of web page or URL when we click on the given link.

* The client (web browser) communicates with the HTTP server and asks for the URL i.e., filename.
* If the web browser finds that requested file, then it sends back to the client (web browser), otherwise it sends the error message to the client as error file.
* Here the responsibility of the web browser to display either the received file or an error message.

## CGI Architecture :-

## CGI module :-

Python provides the **cgi** module, which consists of many useful built-in functions. We can use them by importing the **cgi** module.

import cgi

Now, we can write further script.

import cgi

cgitb.enable()

The above script will stimulate an exception handler to show the detailed report in the web browser of occurred errors. We can also save the report by using the following script.

import cgitb

cgitb.enable(display=0, logdir="/path/to/logdir")

The above feature of the cgi module is helpful during the script development. These reports help us to debug the script effectively. When we get the expected output, we can remove this.

Python provides the FieldStorage class. We can apply the encoding keyword parameter to the document if the form contains the non-ASCII character. We will find the content <META> tag in the <HEAD> section in our HTML document. The FieldStorage class reads the form information from the standard input or the environment. A FieldStorage instance is the same as the Python dictionary.

We can use the len() and all dictionary function in the FieldStorage instance. It overlooks fields with empty string values. We can also consider the empty values using the optional keyword parameter keep\_blank\_values by setting True.

Example :

form = cgi.FieldStorage()

if ("name" not in form or "addr" not in form):

print("<H1>Error</H1>")

print("Please enter the information in the name and address fields.")

return

print("<p>name:", form["name"].value)

print("<p>addr:", form["addr"].value)

#Next lines of code will execute here...

In the above example, we have used the form ["name"], here name is key. This is used to extract the value which is entered by the user.

**CHAPTER-2 : DATABASE PROGRAMMING**

**1). Introduction to Database Programming in Python**

The Python programming language has powerful features for database programming. Some of them are given below :

* Python supports various databases like MySQL, Oracle, Sybase, PostgreSQL, etc.
* Python also supports Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query Statements.
* For database programming, the Python DB API is a widely used module that provides a database application programming interface.
* The Python standard for database interfaces is the Python DB-API.
* You can choose the right database for your application.
* Python Database API supports a wide range of database servers such as –GadFlymSQL MySQL, Postgre SQL, Microsoft SQL Server 2000, Informix, InterbaseOracleSybase

**2). Benefits of Python for database programming**

* Programming in Python is **more efficient and faster compared** to other languages.
* Python is famous for its **portability.**
* It is **platform independent.**
* Python supports **SQL cursors**.
* In many programming languages, the application developer needs to take care of the open and closed connections of the database, to avoid further exceptions and errors. **In Python,** these connections **are taken care of Python s**upports relational database systems.
* Python **database APIs** are compatible with **various databases,** so it is very easy to migrate and port **database application interfaces.**

**3). Introduction to Python DB API module**

We must **download** a separate **DB API module** for each database we need to access. **For example**, if we need to access an Oracle database as well as a MySQL database,we must download both the Oracle and the MySQL database modules. The DB API provides a **minimal standard for working with databases** using Python structures and syntax wherever possible.**This API includes the following :**

1.Importing the API module.

2.Acquiring a connection with the database.

3.Issuing SQL statements and stored procedures.

4.Closing the connection

**Python Database**:-

* Data is retrieved from a database system using the SQL language.
* Data is everywhere and software applicationsuse that. Data is either in memory, files or databases.
* Python has bindingsfor many database systems includingMySQL, Postregsql, Oracle, Microsoft SQL Server and Maria DB.
* One of these database management systems (DBMS) is called SQLite.
* SQLite was created in the year 2000and is one of the many management systems in the database zoo.

**4).The Python Standard Database interface (Python DB-API)**

The Python Standard Database interface is a Python DB-API, which provides a database application programming interface.

**Python DB-API usage process:**

* The API module is introduced.
* Get the connection to the database.
* Execute the SQL statement and the stored procedure.
* Turn off the database connection.

Python3 is connected to the MySQL database via the Pymysql module, usepip3 install PymySQL installation module.

## **import pymysql**

## **# Establish a database connection**

## **db = pymysql.connect(host='127.0.0.1', user='admin', password='123456', database='test', port=3306, charset='utf8')**

## **try:**

## **# Create a cursor object**

## **with db.cursor() as cursor:**

## **# Execute SQL operation**

## **cursor.execute("SELECT VERSION()")**

## **# Extract set**

## **data = cursor.fetchall()**

## **print("Mysql Version:", data[0][0])**

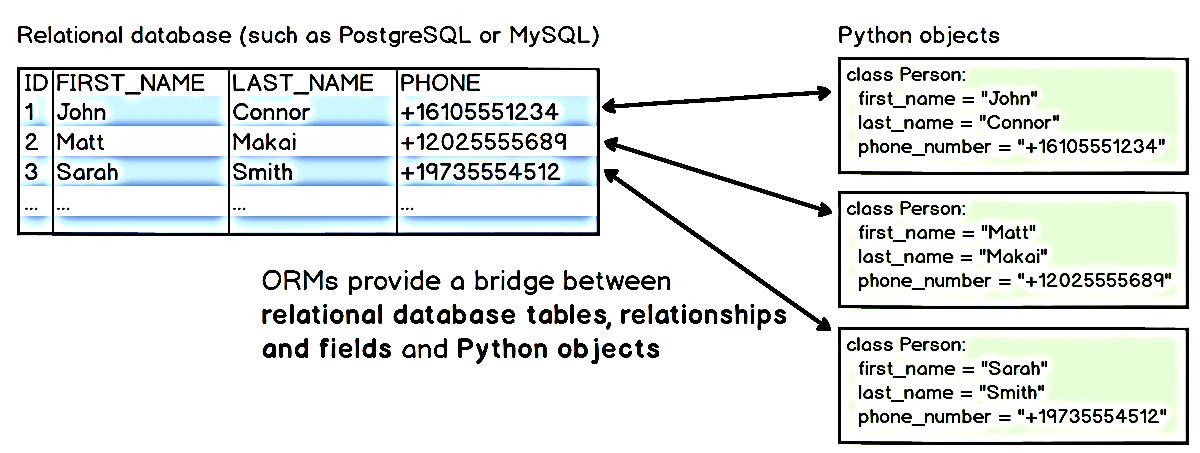
## **finally：**

## **# Turn off database connection**

## **db.close()**

# 5). Object-relational Managres or Mappers (ORMs)

An object-relational manager (ORM) is a code library that automates the transfer of data stored in relational database tables into objects that are more commonly used in application code.



## **ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.**

## The ability to write Python code instead of SQL can speed up web application development, especially at the beginning of a project. The potential development speed boost comes from not having to switch from Python code into writing declarative paradigm SQL statements. While some software developers may not mind switching back and forth between languages, it's typically easier to knock out a prototype or start a web application using a single programming language.

## Python ORM Implementations :-

There are numerous ORM implementations written in Python, including

* [SQLAlchemy](https://www.fullstackpython.com/sqlalchemy.html)
* [Peewee](https://www.fullstackpython.com/peewee.html)
* [The Django ORM](https://www.fullstackpython.com/django-orm.html)
* [PonyORM](https://www.fullstackpython.com/pony-orm.html)
* [SQLObject](http://sqlobject.org/)
* [Tortoise ORM](https://tortoise-orm.readthedocs.io/en/latest/) ([source code](https://github.com/tortoise/tortoise-orm/))