Python Course Syllabus

* Introduction
* History of python
* Python Features
* Key words or reserved words
* Data types
* Variables
* Operators
* Python control structures
* Numbers
* Strings
* Lists and Tuples
* Dictionaries
* Data &Time
* Functions
* Files I/O
* Exceptions
* OOPS Concept
* Regular Expressions
* Multi threading
* GUI Programming

Python Course

**Introduction:**

Python is a high-level, interpreted, interactive and object-oriented scripting language.

Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

**Python is Interpreted** − Python is processed at runtime by the interpreter.

You do not need to compile your program before executing it. This is similar to PERL and PHP.

**Python is Interactive** − You can actually sit at a Python prompt and interact

with the interpreter directly to write your programs.

**Python is Object-Oriented** − Python supports Object-Oriented style or

technique of programming that encapsulates code within objects.

**Python is a Beginner's Language** − Python is a great language for the

beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

**History of python:**

## 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, and Unix shell and other scripting languages.**

## Python is now maintained by a core development team at the institute, although Guido van Rossum still holds a vital role in directing its progress.

**Python Features:**

* **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.

## Python Keywords

Keywords are the reserved words in Python.

We cannot use a keyword as a [variable](https://www.programiz.com/python-programming/variables-datatypes) name, [function](https://www.programiz.com/python-programming/function) name or any other identifier. They are used to define the syntax and structure of the Python language.

In Python, keywords are case sensitive.

There are 33 keywords in Python 3.7. This number can vary slightly over the course of time.

All the keywords except True, False and None are in lowercase and they must be written as they are. The list of all the keywords is given below.

`

| False | await | else | import | pass |
| --- | --- | --- | --- | --- |
| None | break | except | in | raise |
| True | class | finally | is | return |
| and | continue | For | lambda | try |
| as | def | from | nonlocal | while |
| assert | del | global | not | with |
| async | elif | If | or | yield |

Looking at all the keywords at once and trying to figure out what they mean might be overwhelming.

**Data Types:**

The data stored in memory can be of many types. For example, a person's age is stored

as a numeric value and his or her address is stored as alphanumeric characters. Python

has various standard data types that are used to define the operations possible on them

and the storage method for each of them.

Python has four standard data types −

* Numbers
* String
* List
* Tuple

Data types in Python

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below.

## Python Numbers

Integers, floating point numbers and complex numbers fall under [Python numbers](https://www.programiz.com/python-programming/numbers) category. They are defined as int, float and complex classes in Python.

We can use the type() function to know which class a variable or a value belongs to. Similarly, the isinstance() function is used to check if an object belongs to a particular class.

a = 5

print(a, "is of type", type(a))

a = 2.0

print(a, "is of type", type(a))

a = 1+2j

print(a, "is complex number?", isinstance(1+2j,complex))

**Output**

5 is of type <class 'int'>

2.0 is of type <class 'float'>

(1+2j) is complex number? True

Integers can be of any length, it is only limited by the memory available.

A floating-point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is an integer, 1.0 is a floating-point number.

Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part. Here are some examples.

>>> a = 1234567890123456789

>>> a

1234567890123456789

>>> b = 0.1234567890123456789

>>> b

0.12345678901234568

>>> c = 1+2j

>>> c

(1+2j

## Python List

[List](https://www.programiz.com/python-programming/list) is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

a = [1, 2.2, 'python']

We can use the slicing operator [ ] to extract an item or a range of items from a list. The index starts from 0 in Python.

a = [5,10,15,20,25,30,35,40]

# a[2] = 15

print("a[2] = ", a[2])

# a[0:3] = [5, 10, 15]

print("a[0:3] = ", a[0:3])

# a[5:] = [30, 35, 40]

print("a[5:] = ", a[5:])

**Output**

a[2] = 15

a[0:3] = [5, 10, 15]

a[5:] = [30, 35, 40]

Lists are mutable, meaning, the value of elements of a list can be altered.

a = [1, 2, 3]

a[2] = 4

print(a)

## Python Tuple

[Tuple](https://www.programiz.com/python-programming/tuple) is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than lists as they cannot change dynamically.

It is defined within parentheses () where items are separated by commas.

t = (5,'program', 1+3j)

We can use the slicing operator [] to extract items but we cannot change its value.

t = (5,'program', 1+3j)

# t[1] = 'program'

print("t[1] = ", t[1])

# t[0:3] = (5, 'program', (1+3j))

print("t[0:3] = ", t[0:3])

# Generates error

# Tuples are immutable

t[0] = 10

**Output**

t[1] = program

t[0:3] = (5, 'program', (1+3j))

Traceback (most recent call last):

File "test.py", line 11, in <module>

t[0] = 10

TypeError: 'tuple' object does not support item assignment

## Python Strings

[String](https://www.programiz.com/python-programming/string) is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

s = "This is a string"

print(s)

s = '''A multiline

string'''

print(s)

**Output**

This is a string

A multiline

string

Just like a list and tuple, the slicing operator [ ] can be used with strings. Strings, however, are immutable.

s = 'Hello world!'

# s[4] = 'o'

print("s[4] = ", s[4])

# s[6:11] = 'world'

print("s[6:11] = ", s[6:11])

# Generates error

# Strings are immutable in Python

s[5] ='d'

**Output**

s[4] = o

s[6:11] = world

Traceback (most recent call last):

File "<string>", line 11, in <module>

TypeError: 'str' object does not support item assignment

## Python Set

[Set](https://www.programiz.com/python-programming/set) is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

a = {5,2,3,1,4}

# printing set variable

print("a = ", a)

# data type of variable a

print(type(a))

**Output**

a = {1, 2, 3, 4, 5}

<class 'set'>

We can perform set operations like union, intersection on two sets. Sets have unique values. They eliminate duplicates.

a = {1,2,2,3,3,3}

print(a)

**Output**

{1, 2, 3}

Since, set are unordered collection, indexing has no meaning. Hence, the slicing operator [] does not work.

>>> a = {1,2,3}

>>> a[1]

Traceback (most recent call last):

File "<string>", line 301, in runcode

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

TypeError: 'set' object does not support indexing

## Python Dictionary

[Dictionary](https://www.programiz.com/python-programming/dictionary) is an unordered collection of key-value pairs.

It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

>>> d = {1:'value','key':2}

>>> type(d)

<class 'dict'>

We use key to retrieve the respective value. But not the other way around.

d = {1:'value','key':2}

print(type(d))

print("d[1] = ", d[1])

print("d['key'] = ", d['key'])

# Generates error

print("d[2] = ", d[2])

**Output**

<class 'dict'>

d[1] = value

d['key'] = 2

Traceback (most recent call last):

File "<string>", line 9, in <module>

KeyError: 2

## Conversion between data types

We can convert between different data types by using different type conversion functions like int(), float(), str(), etc.

>>> float(5)

5.0

Conversion from float to int will truncate the value (make it closer to zero).

>>> int(10.6)

10

>>> int(-10.6)

-10

Conversion to and from string must contain compatible values.

>>> float('2.5')

2.5

>>> str(25)

'25'

>>> int('1p')

Traceback (most recent call last):

File "<string>", line 301, in runcode

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

ValueError: invalid literal for int() with base 10: '1p'

We can even convert one sequence to another.

>>> set([1,2,3])

{1, 2, 3}

>>> tuple({5,6,7})

(5, 6, 7)

>>> list('hello')

['h', 'e', 'l', 'l', 'o']

To convert to dictionary, each element must be a pair:

>>> dict([[1,2],[3,4]])

{1: 2, 3: 4}

>>> dict([(3,26),(4,44)])

{3: 26, 4: 44}

## Type Conversion

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1. Implicit Type Conversion
2. Explicit Type Conversion

## Implicit Type Conversion

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

### **Example 1: Converting integer to float**

num\_int = 123

num\_flo = 1.23

num\_new = num\_int + num\_flo

print("datatype of num\_int:",type(num\_int))

print("datatype of num\_flo:",type(num\_flo))

print("Value of num\_new:",num\_new)

print("datatype of num\_new:",type(num\_new))

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

datatype of num\_int: <class 'int'>

datatype of num\_flo: <class 'float'>

Value of num\_new: 124.23

datatype of num\_new: <class 'float'>

In the above program,

* We add two variables num\_int and num\_flo, storing the value in num\_new.
* We will look at the data type of all three objects respectively.
* In the output, we can see the data type of num\_int is an integer while the data type of num\_flo is a float.
* Also, we can see the num\_new has a float data type because Python always converts smaller data types to larger data types to avoid the loss of data.

Now, let's try adding a string and an integer, and see how Python deals with it.

### **Example 2: Addition of string(higher) data type and integer(lower) datatype**

num\_int = 123

num\_str = "456"

print("Data type of num\_int:",type(num\_int))

print("Data type of num\_str:",type(num\_str))

print(num\_int+num\_str)

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

Data type of num\_int: <class 'int'>

Data type of num\_str: <class 'str'>

Traceback (most recent call last):

File "python", line 7, in <module>

TypeError: unsupported operand type(s) for +: 'int' and 'str'

In the above program,

* We add two variables num\_int and num\_str.
* As we can see from the output, we got TypeError. Python is not able to use Implicit Conversion in such conditions.
* However, Python has a solution for these types of situations which is known as Explicit Conversion.

## Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.

This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax :

<required\_datatype>(expression)

Typecasting can be done by assigning the required data type function to the expression.

### **Example 3: Addition of string and integer using explicit conversion**

num\_int = 123

num\_str = "456"

print("Data type of num\_int:",type(num\_int))

print("Data type of num\_str before Type Casting:",type(num\_str))

num\_str = int(num\_str)

print("Data type of num\_str after Type Casting:",type(num\_str))

num\_sum = num\_int + num\_str

print("Sum of num\_int and num\_str:",num\_sum)

print("Data type of the sum:",type(num\_sum))

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

Data type of num\_int: <class 'int'>

Data type of num\_str before Type Casting: <class 'str'>

Data type of num\_str after Type Casting: <class 'int'>

Sum of num\_int and num\_str: 579

Data type of the sum: <class 'int'>

In the above program,

* We add num\_str and num\_int variable.
* We converted num\_str from string(higher) to integer(lower) type using int() function to perform the addition.
* After converting num\_str to an integer value, Python is able to add these two variables.
* We got the num\_sum value and data type to be an integer.

Key Points to Remember

1. Type Conversion is the conversion of object from one data type to another data type.
2. Implicit Type Conversion is automatically performed by the Python interpreter.
3. Python avoids the loss of data in Implicit Type Conversion.
4. Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.
5. In Type Casting, loss of data may occur as we enforce the object to a specific data type.

## Python Variables

A variable is a named location used to store data in the memory. It is helpful to think of variables as a container that holds data that can be changed later in the program. For example,

number = 10

Here, we have created a variable named number. We have assigned the value 10 to the variable.

You can think of variables as a bag to store books in it and that book can be replaced at any time.

number = 10

number = 1.1

Initially, the value of number was 10. Later, it was changed to 1.1.

**Note**: In Python, we don't actually assign values to the variables. Instead, Python gives the reference of the object(value) to the variable.

## Assigning values to Variables in Python

As you can see from the above example, you can use the assignment operator = to assign a value to a variable.

### **Example 1: Declaring and assigning value to a variable**

website = "apple.com"

print(website)

**Output**

apple.com

In the above program, we assigned a value apple.com to the variable website. Then, we printed out the value assigned to website i.e. apple.com

**Note**: Python is a [type-inferred](https://en.wikipedia.org/wiki/Type_inference) language, so you don't have to explicitly define the variable type. It automatically knows that apple.com is a string and declares the website variable as a string.

### **Example 2: Changing the value of a variable**

website = "apple.com"

print(website)

# assigning a new value to website

website = "programiz.com"

print(website)

**Output**

apple.com

programiz.com

### **Example 3: Assigning multiple values to multiple variables**

a, b, c = 5, 3.2, "Hello"

print (a)

print (b)

print (c)

If we want to assign the same value to multiple variables at once, we can do this as:

x = y = z = "same"

print (x)

print (y)

print (z)

The second program assigns the same string to all the three variables x, y and z.

## Constants

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.

You can think of constants as a bag to store some books which cannot be replaced once placed inside the bag.

## Assigning value to constant in Python

In Python, constants are usually declared and assigned in a module. Here, the module is a new file containing variables, functions, etc which is imported to the main file. Inside the module, constants are written in all capital letters and underscores separating the words.

### **Example 3: Declaring and assigning value to a constant**

Create a **constant.py**:

PI = 3.14

GRAVITY = 9.8

Create a **main.py**:

import constant

print(constant.PI)

print(constant.GRAVITY)

**Output**

3.14

9.8

In the above program, we create a **constant.py** module file. Then, we assign the constant value to PI and GRAVITY. After that, we create a **main.py** file and import the constant module. Finally, we print the constant value.

**Note**: In reality, we don't use constants in Python. Naming them in all capital letters is a convention to separate them from variables, however, it does not actually prevent reassignment.

## Rules and Naming Convention for Variables and constants

1. Constant and variable names should have a combination of letters in lowercase (a to z) or uppercase (**A to Z**) or digits (**0 to 9**) or an underscore (**\_**). For example:
2. snake\_case
3. MACRO\_CASE
4. camelCase

CapWords

1. Create a name that makes sense. For example, vowel makes more sense than v.
2. If you want to create a variable name having two words, use underscore to separate them. For example:
3. my\_name

current\_salary

1. Use capital letters possible to declare a constant. For example:
2. PI
3. G
4. MASS
5. SPEED\_OF\_LIGHT

TEMP

1. Never use special symbols like !, @, #, $, %, etc.
2. Don't start a variable name with a digit.

Literals

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows:

## Numeric Literals

Numeric Literals are immutable (unchangeable). Numeric literals can belong to 3 different numerical types: Integer, Float, and Complex.

### **Example 4: How to use Numeric literals in Python?**

a = 0b1010 #Binary Literals

b = 100 #Decimal Literal

c = 0o310 #Octal Literal

d = 0x12c #Hexadecimal Literal

#Float Literal

float\_1 = 10.5

float\_2 = 1.5e2

#Complex Literal

x = 3.14j

print(a, b, c, d)

print(float\_1, float\_2)

print(x, x.imag, x.real)

**Output**

10 100 200 300

10.5 150.0

3.14j 3.14 0.0

In the above program,

* We assigned integer literals into different variables. Here, a is binary literal, b is a decimal literal, c is an octal literal and d is a hexadecimal literal.
* When we print the variables, all the literals are converted into decimal values.
* 10.5 and 1.5e2 are floating-point literals. 1.5e2 is expressed with exponential and is equivalent to 1.5 \* 102.
* We assigned a complex literal i.e 3.14j in variable x. Then we use **imaginary** literal (x.imag) and **real** literal (x.real) to create imaginary and real parts of complex numbers.

To learn more about Numeric Literals, refer to [Python Numbers](https://www.programiz.com/python-programming/numbers).

String literals

A string literal is a sequence of characters surrounded by quotes. We can use both single, double, or triple quotes for a string. And, a character literal is a single character surrounded by single or double quotes.

### **Example 7: How to use string literals in Python?**

strings = "This is Python"

char = "C"

multiline\_str = """This is a multiline string with more than one line code."""

unicode = u"\u00dcnic\u00f6de"

raw\_str = r"raw \n string"

print(strings)

print(char)

print(multiline\_str)

print(unicode)

print(raw\_str)

**Output**

This is Python

C

This is a multiline string with more than one line code.

Ünicöde

raw \n string

In the above program, This is Python is a string literal and C is a character literal.

The value in triple-quotes """ assigned to the multiline\_str is a multi-line string literal.

The string u"\u00dcnic\u00f6de" is a Unicode literal which supports characters other than English. In this case, \u00dc represents Ü and \u00f6 represents ö.

r"raw \n string" is a raw string literal.

## Boolean literals

A Boolean literal can have any of the two values: True or False.

### **Example 8: How to use boolean literals in Python?**

x = (1 == True)

y = (1 == False)

a = True + 4

b = False + 10

print("x is", x)

print("y is", y)

print("a:", a)

print("b:", b)

**Output**

x is True

y is False

a: 5

b: 10

In the above program, we use boolean literal True and False. In Python, True represents the value as 1 and False as 0. The value of x is True because 1 is equal to True. And, the value of y is False because 1 is not equal to False.

Similarly, we can use the True and False in numeric expressions as the value. The value of a is 5 because we add True which has a value of 1 with 4. Similarly, b is 10 because we add the False having value of 0 with 10.

## Special literals

Python contains one special literal i.e. None. We use it to specify that the field has not been created.

### **Example 9: How to use special literals in Python?**

drink = "Available"

food = None

def menu(x):

if x == drink:

print(drink)

else:

print(food)

menu(drink)

menu(food)

**Output**

Available

None

In the above program, we define a menu function. Inside menu, when we set the argument as drink then, it displays Available. And, when the argument is food, it displays None.

## Literal Collections

There are four different literal collections List literals, Tuple literals, Dict literals, and Set literals.

### **Example 10: How to use literals collections in Python?**

fruits = ["apple", "mango", "orange"] #list

numbers = (1, 2, 3) #tuple

alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary

vowels = {'a', 'e', 'i' , 'o', 'u'} #set

print(fruits)

print(numbers)

print(alphabets)

print(vowels)

**Output**

['apple', 'mango', 'orange']

(1, 2, 3)

{'a': 'apple', 'b': 'ball', 'c': 'cat'}

{'e', 'a', 'o', 'i', 'u'}

In the above program, we created a list of fruits, a tuple of numbers, a dictionary dict having values with keys designated to each value and a set of vowels.

## operators in python:

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example:

>>> 2+3

5

Here, + is the operator that performs addition. 2 and 3 are the operands and 5 is the output of the operation.

## Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

| Operator | Meaning | Example |
| --- | --- | --- |
| + | Add two operands or unary plus | x + y+ 2 |
| - | Subtract right operand from the left or unary minus | x - y- 2 |
| \* | Multiply two operands | x \* y |
| / | Divide left operand by the right one (always results into float) | x / y |
| % | Modulus - remainder of the division of left operand by the right | x % y (remainder of x/y) |
| // | Floor division - division that results into whole number adjusted to the left in the number line | x // y |
| \*\* | Exponent - left operand raised to the power of right | x\*\*y (x to the power y) |

### **Example 1: Arithmetic operators in Python**

x = 15

y = 4

# Output: x + y = 19

print('x + y =',x+y)

# Output: x - y = 11

print('x - y =',x-y)

# Output: x \* y = 60

print('x \* y =',x\*y)

# Output: x / y = 3.75

print('x / y =',x/y)

# Output: x // y = 3

print('x // y =',x//y)

# Output: x \*\* y = 50625

print('x \*\* y =',x\*\*y)

**Output**

x + y = 19

x - y = 11

x \* y = 60

x / y = 3.75

x // y = 3

x \*\* y = 50625

Comparison operators

Comparison operators are used to compare values. It returns either True or False according to the condition.

| Operator | Meaning | Example |
| --- | --- | --- |
| > | Greater than - True if left operand is greater than the right | x > y |
| < | Less than - True if left operand is less than the right | x < y |
| == | Equal to - True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to - True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to - True if left operand is less than or equal to the right | x <= y |

### **Example 2: Comparison operators in Python**

x = 10

y = 12

# Output: x > y is False

print('x > y is',x>y)

# Output: x < y is True

print('x < y is',x<y)

# Output: x == y is False

print('x == y is',x==y)

# Output: x != y is True

print('x != y is',x!=y)

# Output: x >= y is False

print('x >= y is',x>=y)

# Output: x <= y is True

print('x <= y is',x<=y)

**Output**

x > y is False

x < y is True

x == y is False

x != y is True

x >= y is False

x <= y is True

Logical operators

Logical operators are the and, or, not operators.

| Operator | Meaning | Example |
| --- | --- | --- |
| And | True if both the operands are true | x and y |
| Or | True if either of the operands is true | x or y |
| Not | True if operand is false (complements the operand) | not x |

### **Example 3: Logical Operators in Python**

x = True

y = False

print('x and y is',x and y)

print('x or y is',x or y)

print('not x is',not x)

**Output**

x and y is False

x or y is True

not x is False

Here is the [truth table](https://www.programiz.com/python-programming/keyword-list#and_or_not) for these operators.

Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name.

For example, 2 is 10 in binary and 7 is 111.

**In the table below:** Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

| Operator | Meaning | Example |
| --- | --- | --- |
| & | Bitwise AND | x & y = 0 (0000 0000) |
| | | Bitwise OR | x | y = 14 (0000 1110) |
| ~ | Bitwise NOT | ~x = -11 (1111 0101) |
| ^ | Bitwise XOR | x ^ y = 14 (0000 1110) |
| >> | Bitwise right shift | x >> 2 = 2 (0000 0010) |
| << | Bitwise left shift | x << 2 = 40 (0010 1000) |

Assignment operators

Assignment operators are used in Python to assign values to variables.

a = 5 is a simple assignment operator that assigns the value 5 on the right to the variable a on the left.

There are various compound operators in Python like a += 5 that adds to the variable and later assigns the same. It is equivalent to a = a + 5.

| Operator | Example | Equivalent to |
| --- | --- | --- |
| = | x = 5 | x = 5 |
| += | x += 5 | x = x + 5 |
| -= | x -= 5 | x = x - 5 |
| \*= | x \*= 5 | x = x \* 5 |
| /= | x /= 5 | x = x / 5 |
| %= | x %= 5 | x = x % 5 |
| //= | x //= 5 | x = x // 5 |
| \*\*= | x \*\*= 5 | x = x \*\* 5 |
| &= | x &= 5 | x = x & 5 |
| |= | x |= 5 | x = x | 5 |
| ^= | x ^= 5 | x = x ^ 5 |
| >>= | x >>= 5 | x = x >> 5 |
| <<= | x <<= 5 | x = x << 5 |

Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

### **Identity operators**

is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

| Operator | Meaning | Example |
| --- | --- | --- |
| Is | True if the operands are identical (refer to the same object) | x is True |
| is not | True if the operands are not identical (do not refer to the same object) | x is not True |

### **Example 4: Identity operators in Python**

x1 = 5

y1 = 5

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

# Output: False

print(x1 is not y1)

# Output: True

print(x2 is y2)

# Output: False

print(x3 is y3)

**Output**

False

True

False

Here, we see that x1 and y1 are integers of the same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).

But x3 and y3 are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

### **Membership operators**

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence ([string](https://www.programiz.com/python-programming/string), [list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [set](https://www.programiz.com/python-programming/set) and [dictionary](https://www.programiz.com/python-programming/dictionary)).

In a dictionary we can only test for presence of key, not the value.

| Operator | Meaning | Example |
| --- | --- | --- |
| in | True if value/variable is found in the sequence | 5 in x |
| not in | True if value/variable is not found in the sequence | 5 not in x |

### **Example #5: Membership operators in Python**

x = 'Hello world'

y = {1:'a',2:'b'}

# Output: True

print('H' in x)

# Output: True

print('hello' not in x)

# Output: True

print(1 in y)

# Output: False

print('a' in y)

**Output**

True

True

True

False

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

**Control structures in Python:**

Decision making is required when we want to execute a code only if a certain condition is satisfied.

The if…elif…else statement is used in Python for decision making.

### **Python if Statement Syntax**

if test expression:

statement(s)

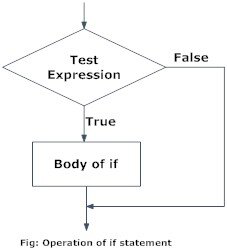
Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True.

If the test expression is False, the statement(s) is not executed.

In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end.

Python interprets non-zero values as True. None and 0 are interpreted as False.

### **Python if Statement Flowchart**

Flowchart of if statement in Python programming

### **Example: Python if Statement**

# find voting or not

age=int(input("enter age"))

if(age>=18):

print("voting possible")

## Python if...else Statement

### **Syntax of if...else**

if test expression:

Body of if

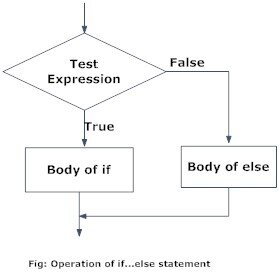
else:

Body of else

The if..else statement evaluates test expression and will execute the body of if only when the test condition is True.

If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

### **Python if..else Flowchart**



Program:

# find voting or not

age=int(input("enter age"))

if(age>=18):

print("voting possible")

else:

print(" not possible voting")

**Program:**

# find even or odd

n=int(input("enter a number"))

if(n%2==0):

print("even")

else:

print("odd")

**Program:**

# find leap year or not

year=int(input("enter a year"))

if(year%4==0):

print("leap year")

else:

print("not leap year")

## Python if...elif...else Statement

### **Syntax of if...elif...else**

if test expression:

Body of if

elif test expression:

Body of elif

else:

Body of else

The elif is short for else if. It allows us to check for multiple expressions.

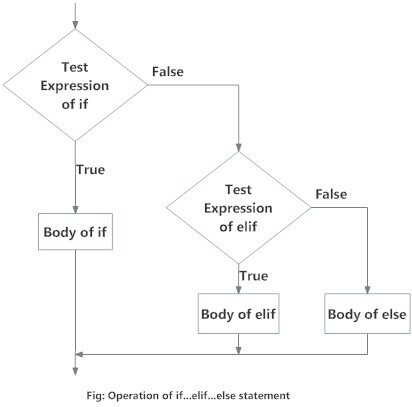
If the condition for if is False, it checks the condition of the next elif block and so on.

If all the conditions are False, the body of else is executed.

Only one block among the several if...elif...else blocks is executed according to the condition.

The if block can have only one else block. But it can have multiple elif blocks.

### **Flowchart of if...elif...else**



Program:

# Student Marks Card:

s=str(input("enter a student name\n"))

r=int(input("enter a roll number\n"))

t=int(input("enter a telugu marks\n"))

h=int(input("enter a hindhi marks\n"))

si=int(input("enter a science marks\n"))

e=int(input("enter a english marks\n"))

m=int(input("enter a maths marks\n"))

so=int(input("enter a social marks\n"))

sum=int(t+h+si+e+m+so)

avg=float(sum/6)

print("Name:-",s)

print("Roll Number:-",r)

print("Telugu-",t)

print("Hindhi-",h)

print("English-",e)

print("Science-",si)

print("Social-",so)

print("Maths-",m)

print("---------------")

print("Total marks=",sum)

print("---------------")

print("percentage=",avg)

if(avg>=75):

print(" status -Distinction")

elif(avg>=65):

print(" status -First class")

elif(avg>=55):

print(" status -second class")

elif(avg>45):

print(" status -third class")

else:

print("status-fail")

**Program:**

# Restaurent Bill Using if and elif

import os

ch=int(input("Chilles restaurent\n1.Chicken biryani\n2.Mutton biryani\n3.Roties\n4.Chickencurry\n5.Exit\nplease give your order:"))

if(ch==1):

price\_ofChickenbiryani=130

qty=int(input("how many Chicken biryanies"))

p= (price\_ofChickenbiryani\*qty)

print("you pay just rupees:",p)

input()

os.system('cls')

elif(ch==2):

price\_ofMuttonbiryani =200

qty=int(input("how many Mutton biryani "))

p= (price\_ofMuttonbiryani\*qty)

print("you pay just rupees:",p)

input()

os.system('cls')

elif(ch==3):

price\_ofRoties =15

qty=int(input("how many roties "))

p= (price\_ofRoties\*qty)

print("you pay just rupees:",p)

input()

os.system('cls')

elif(ch==4):

price\_ofchickencurry=180

qty=int(input("how many chicken curry packs\n"))

p= (price\_ofchickencurry\*qty)

print("you can pay just only:")

input()

os.system('cls')

elif ch==5:

print("ARE YOU SURE EXIT(y/n):")

if opt == "y" :

ch=5

input()

os.system('cls')

else :

ch=0

input()

os.system('cls')

else:

print("Out of choice")

Program:

# Given Character Alaphet or Vowel:

ch=str(input("enter a character\n"))

if(ch=='a'):

print(" your enter character is vowel")

elif(ch=='e'):

print(" your enter character is vowel")

elif(ch=='i'):

print(" your enter character is vowel")

elif(ch=='o'):

print(" your enter character is vowel")

elif(ch=='u'):

print(" your enter character is vowel")

else:

print(" your enter character is consonants")

## Python Nested if statements

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

### **Python Nested if Example**

'''In this program, we input a number

check if the number is positive or

negative or zero and display

an appropriate message

This time we use nested if statement'''

num = float(input("Enter a number: "))

if num >= 0:

if num == 0:

print("Zero")

else:

print("Positive number")

else:

print("Negative number")

**Loop Statements in Python:**

## for loop in Python?

The for loop in Python is used to iterate over a sequence ([list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [string](https://www.programiz.com/python-programming/string)) or other iterable objects. Iterating over a sequence is called traversal.

### **Syntax of for Loop**

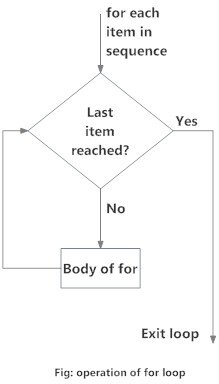
for val in sequence:

Body of for

Here, val is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

### **Flowchart of for Loop**

Flowchart of for Loop in Python

Programs:

#Required Number sum:

n=int(input("enter a n value\n"))

i=int(input("enter a i value\n"))

sum=0

print("Required Numbers")

for i in range(n,i):

print(i)

sum=sum+i

print("------------")

print("sum=",sum)

print("------------")

Program:

# print of given Number:

n=int(input("enter a number is="))

for i in range(1,n+1):

print(i)

**Program**

# full triangle:

num=int(input("enter the number of the rows"))

for i in range (1,num+1):

for j in range (1,num-i+1):

print(end=" ")

for j in range(i,0,-1):

print(j,end=" ")

for j in range(2,i+1):

print(j,end=" ")

print()

**Program:**

# Factorial of given Number:

n=int(input("enter a factirial of the number is="))

factorial=1

for i in range(1,n+1):

factorial=factorial\*i

print("factorial of the number :\n",n,"!","=",factorial)

**Program:**

# Table :

j=int(input("enter a value\n"))

print("youe multification of table is:",j)

for i in range(1,21):

k=j\*i

print(j,'\*',i,"=",j\*i)

**Program:**

#prime or not enterd number:

a=int(input("enter a number \n"))

i=2

if(a>1):

for i in range(2,a):

if((a%i)==0):

print("your enter is not prime number",a)

break

else print("your enter is a prime number")

### **Example: Python for Loop**

# Program to find the sum of all numbers stored in a list

# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

sum = sum+val

print("The sum is", sum)

When you run the program, the output will be:

The sum is 48

The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start, stop,step\_size). step\_size defaults to 1 if not provided.

The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len and \_\_getitem\_\_ operations.

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The following example will clarify this.

print(range(10))

print(list(range(10)))

print(list(range(2, 8)))

print(list(range(2, 20, 3)))

**Output**

range(0, 10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[2, 3, 4, 5, 6, 7]

[2, 5, 8, 11, 14, 17]

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example.

# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index

for i in range(len(genre)):

print("I like", genre[i])

**Output**

I like pop

I like rock

​I like jazz

## for loop with else

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts.

The break keyword can be used to stop a for loop. In such cases, the else part is ignored.

Hence, a for loop's else part runs if no break occurs.

Here is an example to illustrate this.

digits = [0, 1, 5]

for i in digits:

print(i)

else:

print("No items left.")

When you run the program, the output will be:

0

1

5

No items left.

Here, the for loop prints items of the list until the loop exhausts. When the for loop exhausts, it executes the block of code in the else and prints No items left.

This for...else statement can be used with the break keyword to run the else block only when the break keyword was not executed. Let's take an example:

# program to display student's marks from record

student\_name = 'Soyuj'

marks = {'James': 90, 'Jules': 55, 'Arthur': 77}

for student in marks:

if student == student\_name:

print(marks[student])

break

else:

print('No entry with that name found.')

## while loop in Python?

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know the number of times to iterate beforehand.

### **Syntax of while Loop in Python**

while test\_expression:

Body of while

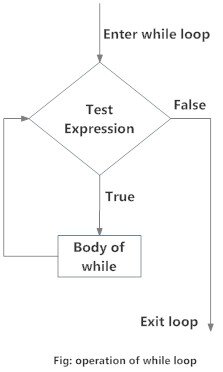
In the while loop, test expression is checked first. The body of the loop is entered only if the test\_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test\_expression evaluates to False.

In Python, the body of the while loop is determined through indentation.

The body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as False.

### **Flowchart of while Loop**

Flowchart for while loop in Python

**Program:**

n=int(input("enter a number"))

i=1

while i<=n:

print(i)

i=i+1

print("while loop end")

**Program:**

n=10

sum=0

i=1

while i<=n:

sum=sum+i

i=i+1

print("the sum",sum**)**

**Program:**

#find palindrome or not

n=int(input("enter number"))

temp=n

rev=0

while(n>0):

dig=n%10

rev=rev\*10+dig

n=n//10

if(temp==rev):

print("the number is a palindrome",temp)

else:

print("the number is not palindrome",temp)

**Program:**

#find amstrong or not

n=int(input("enter number"))

sum=0

temp=n

while temp>0:

dig=temp%10

sum+=dig\*\*3

temp//=10

if(n==sum):

print("the number is a palindrome",n)

else:

print("the number is not palindrome",n)

**Program:**

# Hcm and Lcm:

num1=int(input("enter a number\n"))

num2=int(input("enter a number\n"))

if(num1=='x'):

exit()

else:

number1=int(num1)

number2=int(num2)

temp1=number1

temp2=number2

while(temp2!=0):

t=temp2

temp2=temp1%temp2

temp1=t

hcf=temp1

lcm=(number1\*number2)/hcf

print("HCF=",hcf)

print("LCM=",lcm)

## While loop with else

Same as with [for loops](https://www.programiz.com/python-programming/for-loop), while loops can also have an optional else block.

The else part is executed if the condition in the while loop evaluates to False.

The while loop can be terminated with a [break statement](https://www.programiz.com/python-programming/break-continue). In such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

Here is an example to illustrate this.

'''Example to illustrate

the use of else statement

with the while loop'''

counter = 0

while counter < 3:

print("Inside loop")

counter = counter + 1

else:

print("Inside else")

**Output**

Inside loop

Inside loop

Inside loop

Inside else

Here, we use a counter variable to print the string Inside loop three times.

On the fourth iteration, the condition in while becomes False. Hence, the else part is executed.

**Unconditional Statements:**

## Python break statement

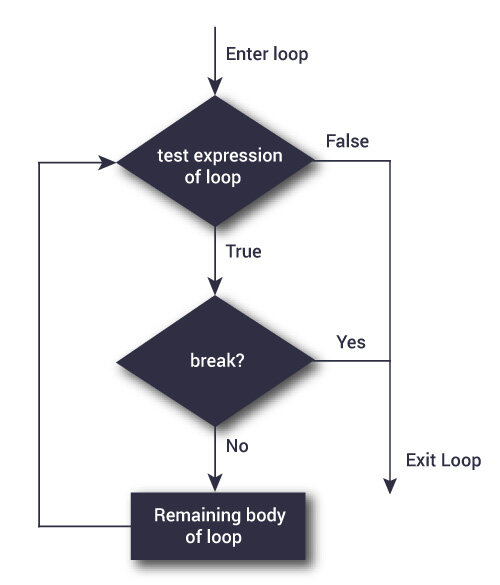
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

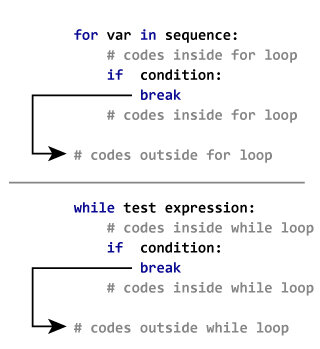
### **Syntax of break**

break

### **Flowchart of break**

Flowchart of break statement in Python

The working of break statement in [for loop](https://www.programiz.com/python-programming/for-loop) and [while loop](https://www.programiz.com/python-programming/while-loop) is shown below.

Working of the break statement

### **Example: Python break**

# Use of break statement inside the loop

for val in "string":

if val == "i":

break

print(val)

print("The end")

**Output**

s

t

r

The end

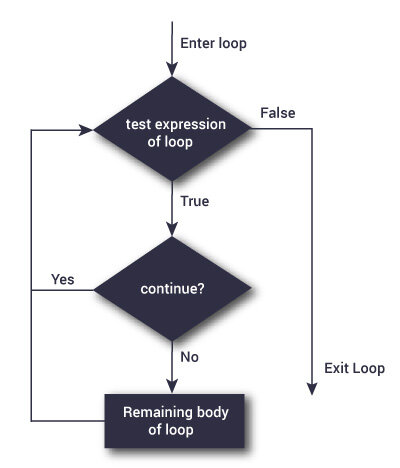
In this program, we iterate through the "string" sequence. We check if the letter is i, upon which we break from the loop. Hence, we see in our output that all the letters up till i gets printed. After that, the loop terminates.

Python continue statement:

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

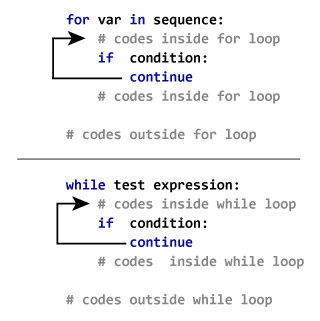
### **Syntax of Continue**

continueFlowchart of continue



Flowchart of continue statement in Python

The working of the continue statement in for and while loop is shown below.

How continue statement works in python

### **Example: Python continue**

# Program to show the use of continue statement inside loops

for val in "string":

if val == "i":

continue

print(val)

print("The end")

**Output**

s

t

r

n

g

The end

This program is same as the above example except the break statement has been replaced with continue.

We continue with the loop, if the string is i, not executing the rest of the block. Hence, we see in our output that all the letters except i gets printed.

## pass statement in Python?

In Python programming, the pass statement is a null statement. The difference between a [comment](https://www.programiz.com/python-programming/statement-indentation-comments) and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored.

However, nothing happens when the pass is executed. It results in no operation (NOP).

### **Syntax of pass**

pass

We generally use it as a placeholder.

Suppose we have a [loop](https://www.programiz.com/python-programming/for-loop) or a [function](https://www.programiz.com/python-programming/function) that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would give an error. So, we use the pass statement to construct a body that does nothing.

### **Example: pass Statement**

'''pass is just a placeholder for

functionality to be added later.'''

sequence = {'p', 'a', 's', 's'}

for val in sequence:

pass

We can do the same thing in an empty [function](https://www.programiz.com/python-programming/function) or [class](https://www.programiz.com/python-programming/class) as well.

def function(args):

pass

class Example:

pass

## function in Python?

In Python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

### **Syntax of Function**

def function\_name(parameters):

"""docstring"""

statement(s)

Above shown is a function definition that consists of the following components.

* Keyword def that marks the start of the function header.
* A function name to uniquely identify the function. Function naming follows the same [rules of writing identifiers in Python](https://www.programiz.com/python-programming/keywords-identifier#rules).
* Parameters (arguments) through which we pass values to a function. They are optional.
* A colon (:) to mark the end of the function header.
* Optional documentation string (docstring) to describe what the function does.
* One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
* An optional return statement to return a value from the function.

### **Example of a function**

def greet(name):

"""

This function greets to

the person passed in as

a parameter

"""

print("Hello, " + name + ". Good morning!")

### **How to call a function in python?**

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

>>> greet('Paul')

Hello, Paul. Good morning!

**Note:** Try running the above code in the Python program with the function definition to see the output.

def greet(name):

"""

This function greets to

the person passed in as

a parameter

"""

print("Hello, " + name + ". Good morning!")

greet('Paul')

Docstrings

The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does.

Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as the \_\_doc\_\_ attribute of the function.

**For example**:

Try running the following into the Python shell to see the output.

>>> print(greet.\_\_doc\_\_)

This function greets to

the person passed in as

a parameter

To learn more about docstrings in Python, visit [Python Docstrings](https://www.programiz.com/python-programming/docstrings).

The return statement

The return statement is used to exit a function and go back to the place from where it was called.

### **Syntax of return**

return [expression\_list]

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

**For example:**

>>> print(greet("May"))

Hello, May. Good morning!

None

Here, None is the returned value since greet() directly prints the name and no return statement is used.

### **Example of return**

def absolute\_value(num):

"""This function returns the absolute

value of the entered number"""

if num >= 0:

return num

else:

return -num

print(absolute\_value(2))

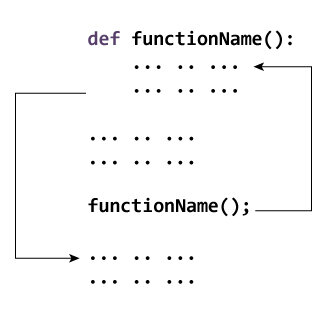
print(absolute\_value(-4))

**Output**

2

4

## How Function works in Python?

Working of functions in Python

## Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.

The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

def my\_func():

x = 10

print("Value inside function:",x)

x = 20

my\_func()

print("Value outside function:",x)

**Output**

Value inside function: 10

Value outside function: 20

Here, we can see that the value of x is 20 initially. Even though the function my\_func() changed the value of x to 10, it did not affect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

Types of Functions

Basically, we can divide functions into the following two types:

* [Built-in functions](https://www.programiz.com/python-programming/methods/built-in) - Functions that are built into Python.
* [User-defined functions](https://www.programiz.com/python-programming/user-defined-function) - Functions defined by the users themselves.

Programs:

# Demo for functions:

def sum(a,b):

return(a+b)

def diff(a,b):

return(a-b)

def multi(a,b):

return(a\*b)

def div(a,b):

return(a/b)

def r(a,b):

return(a//b)

print("sum:",(4+5))

print("diff",(45-9))

print("multi",(45\*9))

print("diff",(38/9))

print("diff",(3619//60))

**Program:**

# to calculate electricity bill

name=str(input("enter name of the consumer:"))

number=int(input("enter consumer number:"))

unit=int(input("Enter the unit consumed: "))

print("consumer name:",name)

print("consumer:no",number)

def Bill\_Calc(unit):

if(unit<=50):

pay=unit\*1.50

fixedcharge=25

t=pay+fixedcharge

print("you total electricity bill:",t)

elif(unit<=150):

pay=(50\*1.5)+(unit-50)\*2.00

fixedcharge=50

t=pay+fixedcharge

print("you total electricity bill:",t)

elif((unit<=250)):

pay=(50\*1.5)+((150-50)\*2.00)+((unit-150)\*3.00)

fixedcharge=75

t=pay+fixedcharge

print("you total electricity bill:",t)

elif(unit<=350):

pay=(50\*1.5)+((150-50)\*2.0)+((250-150)\*3.00)+((unit-350)\*4)

fixedcharge=90

t=pay+fixedcharge

print("you total electricity bill:",t)

else:

pay=(50\*1.5)+((150-50)\*2.00)+((250-150)\*3.00)+(350-250)\*4+(unit-350)\*5

fixedcharge=90

t=pay+fixedcharge

print("you total electricity bill:",t)

Bill\_Calc(unit)

**Program:**

def chilles\_b(bill):

if bill == 1:

price\_ofChickenbiryani=130

qty=int(input("how many Chicken biryanies"))

p= (price\_ofChickenbiryani\*qty)

if qty >= 5:

dis= 0.3\*p

price\_ofChickenbiryani=130

print("you Discount you pay:",(p-dis))

else :

print("you pay just rupees:",p)

if bill == 2:

price\_ofMuttonbiryani =200

qty=int(input("how many Mutton biryani "))

p= (price\_ofMuttonbiryani\*qty)

if qty >= 5:

dis= 0.3\*p

price\_ofChickenbiryani=130

print("you Discount you pay:",(p-dis))

else :

print("you pay just rupees:",p)

if bill == 3:

price\_ofRoties =15

qty=int(input("how many roties "))

p= (price\_ofRoties\*qty)

if qty >= 5:

dis= 0.3\*p

price\_ofChickenbiryani=130

print("you Discount you pay:",(p-dis))

else :

print("you pay just rupees:",p)

if bill == 4:

price\_ofchickencurry=180

qty=int(input("how many chicken curry packs"))

p= (price\_ofchickencurry\*qty)

if qty >= 5:

dis= 0.3\*p

price\_ofChickenbiryani=130

print("you Discount you pay:",(p-dis))

else:

print("you pay just rupees:",p)

import os

ch=0

while ch !=5 :

ch=int(input("Chilles restaurent\n1.Chicken biryani\n2.Mutton biryani\n3.Roties\n4.ChickenCurry\n5.Exit\nplease give your order:"))

if ch == 1:

chilles\_b(1)

input()

os.system('cls')

elif ch == 2:

chilles\_b(2)

input()

os.system('cls')

elif ch == 3:

chilles\_b(3)

input()

os.system('cls')

elif ch == 4:

chilles\_b(4)

input()

os.system('cls')

elif ch == 5:

opt=input("ARE YOU SURE EXIT(y/n)")

if opt == 'y':

ch = 5

input()

os.system('cls')

else:

ch = 0

input()

os.system('cls')

else :

print(" \*\*\*\*\*You Choose Correct Selection \*\*\*\*\*")

input()

os.system('cls')

Program:

#Calculator Using function:

def calculator(func):

a=int(input("enter a A value"))

b=int(input("enter a B value"))

if func=='add':

c=a+b

print(" ----------------------------")

print(" Adding a value is",c)

print(" ----------------------------")

elif func=='sub':

c=a-b

print(" ----------------------------")

print(" Subraction value:",c)

print(" ----------------------------")

elif func=='m':

c=a%b

print(" ----------------------------")

print("|","Multification value is:",c,"|")

print(" -----------------------------")

elif func=='d':

c=a\*b

print(" ----------------------------")

print(" Division value",c)

print(" ----------------------------")

ch = 0

while(ch!=5):

ch=int(input("Calculator\n1.Adding\n2.Subraction\n3.Multiplying\n4.Dividing\n5.Exit\nEnter your Choice:\n"))

if ch==1:

calculator('add')

elif ch==2:

calculator('sub')

elif ch==3:

calculator('m')

elif ch==4:

calculator('d')

elif ch==5:

opt =input("Are you sure to exit(y/n)")

if opt=='y':

ch=5

else:

ch=0

else:

print("\_\_\_\_\_\_\_\_\_\_\_\_Invalid choice\_\_\_\_\_\_\_\_\_\_\_\_")

**What is the Recursion:**

Recursion is the process of defining something in terms of itself.

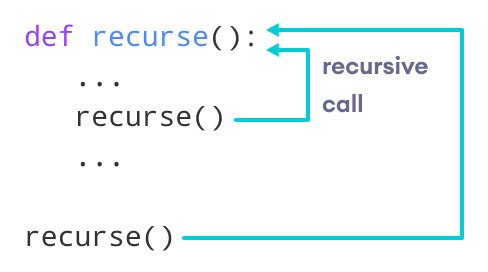
A function calls itself is called recursion.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python Recursive Function

In Python, we know that a [function](https://www.programiz.com/python-programming/function) can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

The following image shows the working of a recursive function called recurse.

Recursive Function in Python

Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1\*2\*3\*4\*5\*6 = 720.

### **Example of a recursive function**

def factorial(x):

"""This is a recursive function

to find the factorial of an integer"""

if x == 1:

return 1

else:

return (x \* factorial(x-1))

num = 3

print("The factorial of", num, "is", factorial(num))

**Output**

The factorial of 3 is 6

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

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

factorial(3) # 1st call with 3

3 \* factorial(2) # 2nd call with 2

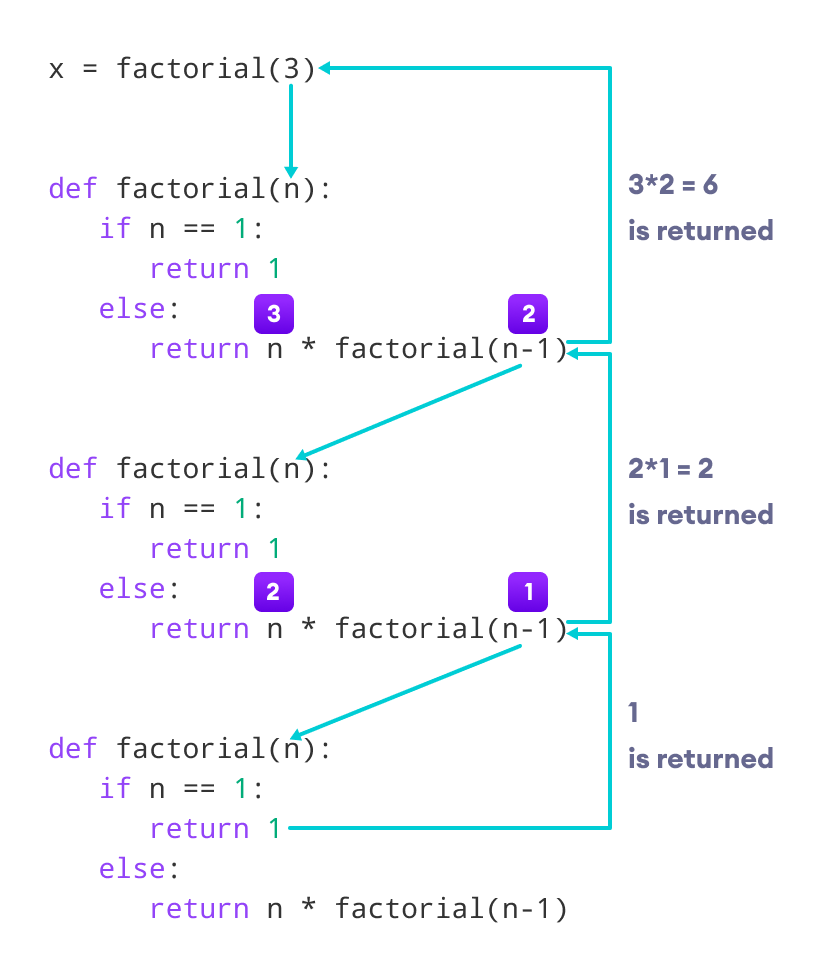
3 \* 2 \* factorial(1) # 3rd call with 1

3 \* 2 \* 1 # return from 3rd call as number=1

3 \* 2 # return from 2nd call

6 # return from 1st call

Let's look at an image that shows a step-by-step process of what is going on:

Working of a recursive factorial function

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError. Let's look at one such condition.

def recursor():

recursor()

recursor()

**Output**

Traceback (most recent call last):

File "<string>", line 3, in <module>

File "<string>", line 2, in a

File "<string>", line 2, in a

File "<string>", line 2, in a

[Previous line repeated 996 more times]

RecursionError: maximum recursion depth exceeded

## Advantages of Recursion

* Recursive functions make the code look clean and elegant.
* A complex task can be broken down into simpler sub-problems using recursion.
* Sequence generation is easier with recursion than using some nested iteration.

## Disadvantages of Recursion

* Sometimes the logic behind recursion is hard to follow through.
* Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
* Recursive functions are hard to debug.

## **lambda functions in Python?**

In Python, an anonymous function is a [function](https://www.programiz.com/python-programming/function) that is defined without a name.

While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.

Hence, anonymous functions are also called lambda functions.

## How to use lambda Functions in Python?

A lambda function in python has the following syntax.

### **Syntax of Lambda Function in python**

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

### **Example of Lambda Function in python**

Here is an example of lambda function that doubles the input value.

# Program to show the use of lambda functions

double = lambda x: x \* 2

print(double(5))

**Output**

10

In the above program, lambda x: x \* 2 is the lambda function. Here x is the argument and x \* 2 is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier double. We can now call it as a normal function. The statement

double = lambda x: x \* 2

is nearly the same as:

def double(x):

return x \* 2

## Use of Lambda Function in python

We use lambda functions when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as [arguments](https://www.programiz.com/python-programming/function-argument)). Lambda functions are used along with built-in functions like filter(), map() etc.

### **Example use with filter()**

The filter() function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Here is an example use of filter() function to filter out only even numbers from a list.

# Program to filter out only the even items from a list

my\_list = [1, 5, 4, 6, 8, 11, 3, 12]

new\_list = list(filter(lambda x: (x%2 == 0) , my\_list))

print(new\_list)

**Output**

[4, 6, 8, 12]

### **Example use with map()**

The map() function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of map() function to double all the items in a list.

# Program to double each item in a list using map()

my\_list = [1, 5, 4, 6, 8, 11, 3, 12]

new\_list = list(map(lambda x: x \* 2 , my\_list))

print(new\_list)

**Output**

[2, 10, 8, 12, 16, 22, 6, 24]

**Program:**

# Program to show the use of lambda functions

double = lambda x: x \* 2

print(double(5))

**Program:**

# circle areas and perimeters with lambda.

radious=[20,24,34,56,78,90,23,4,]

areac=list(map(lambda r:3.14\*r\*r,radious))

permc=list(map(lambda r:3.14\*2\*r,radious))

print("\_\_\_\_AREA OF THE GIVEN CIRCLES\_\_\_")

print(areac)

print('''

''')

print('\*\*\*\*\*\*\*Perimeter of the given radious\*\*\*\*\*\*\*\*\*')

print(permc)

**Program:**

# Program to double each item in a list using map()

my\_list = [1, 5, 4, 6, 8, 11, 3, 12]

new\_list = list(map(lambda x: x \* 2 , my\_list))

print(new\_list)

**Program:**

# Program to filter out only the even items from a list

my\_list = [1, 5, 4, 6, 8, 11, 3, 12]

new\_list = list(filter(lambda x: (x%2 == 0) , my\_list))

print(new\_list)

**Program:**

m1=['Samsung','Vivo','Nokia','Oppo','Redme','Lenovo','Honor']

sl=['Samsung Galax M40','Samsung Galaxy M20','Samsung J7 pro','Samsung W2019','Samsung Galaxy S5 Sport','Samsung Galaxy Folder 2','Samsung Galaxy On8 (2018)']

sr=[20000,9990,14000,198000,16000,16400,11900]

vl=['VivoY69','VivoV11','Vivo Y83','Vivo Y21L','Vivo V9Pro','Vivo V5Plus']

vr=[9879,19090,11990,7400,17790,25990] # Mobiles PriceList using lambda with filter,mapand zip:

nl=['Nokia 8.1','Nokia6.1(2018)','Nokia5.1Plus','Nokia2.2','Nokia Asha','Nokia Asha 501','Nokia 6.1 ','Nokia 6.1']

nr=[26000,10400,9336,7738,5000,4500,5490,10699]

os='OPPO'

ol=['OPPO A83','OPPO A5S','OPPO F11 Pro','OPPO A5S','OPPO A5S blue']

os=[6490,8990,23990,9990,12990]

r='REDMI'

rl=['Redmi 6A (Black)','Redmi 7','Redmi Y3','Redmi Note 7 Pro','Redmi Note 7 Pro','Redmi 7 ecllipse',]

rs=[5990,7999,11999,20999,6499,8000]

ch=int(input("\_\_\_\_\_\_Arjun Mobile Stores\_\_\_\_\n1.list\n2.Search\n3.Exit\nEnter your choices:\n"))

if(ch==1):

print('........Name of the mobiles in the shop..........')

print('1:-',m1[0])

print('2:-',m1[1])

print('3:-',m1[2])

print('4:-',m1[3])

print('5:-',m1[4])

print('6:-',m1[5])

print('7:-',m1[6])

ch=int(input("\_\_\_\_\_\_Arjun Mobile Stores\_\_\_\_\n1.list\n2.Search\n3.Exit\nEnter your choices:\n"))

if(ch==2):

cho=int(input('1.Samsung\n2.VIVO\n3.NOKIA\n4.OPPO\n5.REDMI\n6.LENOVO\n7.HONOR\nEnter your Favourate Mobile Name:'))

if(cho==1):

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_")

print("{0:>0}{1:>15}{2:>15}{3:>5}".format('Mobile Name','|','Price','|'))

print("{0:>0}{1:>9}{2:>15}{3:>5}".format(sl[0],'|',sr[0],'|'))

print("{0:>0}{1:>8}{2:>14}{3:>6}".format(sl[1],'|',sr[1],'|'))

print("{0:>0}{1:>12}{2:>15}{3:>5}".format(sl[2],'|',sr[2],'|'))

print("{0:>0}{1:>13}{2:>16}{3:>4}".format(sl[3],'|',sr[3],'|'))

print("{0:>0}{1:>3}{2:>15}{3:>5}".format(sl[4],'|',sr[4],'|'))

print("{0:>0}{1:>3}{2:>15}{3:>5}".format(sl[5],'|',sr[5],'|'))

print("{0:>0}{1:>1}{2:>15}{3:>5}".format(sl[6],'|',sr[6],'|'))

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ ")

elif(cho==2):

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_")

print("{0:>0}{1:>5}{2:>15}{3:>5}".format('Mobile Name','|','Price','|'))

print("{0:>0}{1:>9}{2:>15}{3:>5}".format(vl[0],'|',vr[0],'|'))

print("{0:>0}{1:>9}{2:>16}{3:>4}".format(vl[1],'|',vr[1],'|'))

print("{0:>0}{1:>8}{2:>16}{3:>4}".format(vl[2],'|',vr[2],'|'))

print("{0:>0}{1:>7}{2:>15}{3:>5}".format(vl[3],'|',vr[3],'|'))

print("{0:>0}{1:>6}{2:>16}{3:>4}".format(vl[4],'|',vr[4],'|'))

print("{0:>0}{1:>5}{2:>16}{3:>4}".format(vl[5],'|',vr[5],'|'))

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_")

elif(cho==3):

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_")

print("{0:>0}{1:>7}{2:>15}{3:>5}".format('Mobile Name','|','Price','|'))

print("{0:>0}{1:>9}{2:>15}{3:>5}".format(nl[0],'|',nr[0],'|'))

print("{0:>0}{1:>4}{2:>15}{3:>5}".format(nl[1],'|',nr[1],'|'))

print("{0:>0}{1:>6}{2:>14}{3:>6}".format(nl[2],'|',nr[2],'|'))

print("{0:>0}{1:>10}{2:>14}{3:>6}".format(nl[3],'|',nr[3],'|'))

print("{0:>0}{1:>8}{2:>14}{3:>6}".format(nl[4],'|',nr[4],'|'))

print("{0:>0}{1:>4}{2:>14}{3:>6}".format(nl[5],'|',nr[5],'|'))

print("{0:>0}{1:>8}{2:>14}{3:>6}".format(nl[6],'|',nr[6],'|'))

print("{0:>0}{1:>9}{2:>15}{3:>5}".format(nl[7],'|',nr[7],'|'))

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_")

elif(cho==4):

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_")

print("{0:>0}{1:>6}{2:>15}{3:>5}".format('Mobile Name','|','Price','|'))

print("{0:>0}{1:>9}{2:>14}{3:>6}".format(ol[0],'|',os[0],'|'))

print("{0:>0}{1:>9}{2:>14}{3:>6}".format(ol[1],'|',os[1],'|'))

print("{0:>0}{1:>5}{2:>15}{3:>5}".format(ol[2],'|',os[2],'|'))

print("{0:>0}{1:>9}{2:>14}{3:>6}".format(ol[3],'|',os[3],'|'))

print("{0:>0}{1:>4}{2:>15}{3:>5}".format(ol[4],'|',os[4],'|'))

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ ")

elif(cho==5):

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ ")

print("{0:>0}{1:>8}{2:>15}{3:>5}".format('Mobile Name','|','Price','|'))

print("{0:>0}{1:>3}{2:>15}{3:>5}".format(rl[0],'|',rs[0],'|'))

print("{0:>0}{1:>12}{2:>15}{3:>5}".format(rl[1],'|',rs[1],'|'))

print("{0:>0}{1:>11}{2:>16}{3:>4}".format(rl[2],'|',rs[2],'|'))

print("{0:>0}{1:>3}{2:>16}{3:>4}".format(rl[3],'|',rs[3],'|'))

print("{0:>0}{1:>3}{2:>15}{3:>5}".format(rl[4],'|',rs[4],'|'))

print("{0:>0}{1:>3}{2:>15}{3:>5}".format(rl[5],'|',rs[5],'|'))

print("\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_")

elif(cho==6):

print("""\*\*\*\*\*\*\*\*\*\*\*\*\*

OUT OF STOCK

\*\*\*\*\*\*\*\*\*\*\*\*\*""")

elif(cho==7):

print("""\*\*\*\*\*\*\*\*\*\*\*\*\*

OUT OF STOCK

\*\*\*\*\*\*\*\*\*\*\*\*\*""")

elif(ch==3):

print (" ARE you Sure want to exit ")

else:

print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ SORRY \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

Program:

# Calculate BMI with their statuses.

import os

n=['Arjun','Rohan','Laxman','Riya','Sathvika']

w=[70,100,90,85,70]

h=[1.65,1.89,2.00,2.15,2.2]

b=list(map(lambda x,y:(x)/(y\*y),w,h))

print('''

''')

print("\_\_\_\_\_\_\_\_Status of the peoples\_\_\_\_\_\_")

ch=0

while(ch!=6):

ch=int(input("enter candidate number"))

if(ch==1):

print("""Candidate-1:-""")

print("Name:",n[0])

print("weight:",w[0])

print("Hieght:",h[0])

print("BMI:",b[0])

s1=list(filter(lambda x:x>=25,b))

print('status: The candidate is over weight')

input()

os.system("cls")

elif(ch==2):

print("""Candidate-2:-""")

print("Name:",n[1])

print("weight:",w[1])

print("Hieght:",h[1])

print("BMI:",b[1])

s2=list(filter(lambda x:x>=25,b))

print('status: The candidate is Obsity')

input()

os.system("cls")

elif(ch==3):

print("""Candidate-3:-""")

print("Name:",n[2])

print("weight:",w[2])

print("Hieght:",h[2])

print("BMI:",b[2])

s2=list(filter(lambda x:x>=22 and x==23,b))

print('status: The candidate is normal weight')

input()

os.system("cls")

elif(ch==4):

print("""Candidate-4:-""")

print("Name:",n[3])

print("weight:",w[3])

print("Hieght:",h[3])

print("BMI:",b[3])

s3=list(filter(lambda x:x<=20,b))

print('status: The candidate is under weight')

input()

os.system("cls")

elif(ch==5):

print("""Candidate-5:-""")

print("Name:",n[4])

print("weight:",w[4])

print("Hieght:",h[4])

print("BMI:",b[4])

s4=list(filter(lambda x:x<=20,b))

print('status: The candidate is under weight')

input()

os.system("cls")

else:

print("exit your information list")

## Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Let's see an example of how a global variable is created in Python.

### **Example 1: Create a Global Variable**

x = "global"

def foo():

print("x inside:", x)

foo()

print("x outside:", x)

**Output**

x inside: global

x outside: global

In the above code, we created x as a global variable and defined a foo() to print the global variable x. Finally, we call the foo() which will print the value of x.

What if you want to change the value of x inside a function?

x = "global"

def foo():

x = x \* 2

print(x)

foo()

**Output**

UnboundLocalError: local variable 'x' referenced before assignment

The output shows an error because Python treats x as a local variable and x is also not defined inside foo().

To make this work, we use the global keyword. Visit [Python Global Keyword](https://www.programiz.com/python-programming/global-keyword) to learn more.

## Local Variables

A variable declared inside the function's body or in the local scope is known as a local variable.

### **Example 2: Accessing local variable outside the scope**

def foo():

y = "local"

foo()

print(y)

**Output**

NameError: name 'y' is not defined

The output shows an error because we are trying to access a local variable y in a global scope whereas the local variable only works inside foo() or local scope.

Let's see an example on how a local variable is created in Python.

### **Example 3: Create a Local Variable**

Normally, we declare a variable inside the function to create a local variable.

def foo():

y = "local"

print(y)

foo()

**Output**

local

Let's take a look at the [earlier problem](https://www.programiz.com/python-programming/global-local-nonlocal-variables#change-x) where x was a global variable and we wanted to modify x inside foo().

Global and local variables

Here, we will show how to use global variables and local variables in the same code.

### **Example 4: Using Global and Local variables in the same code**

x = "global "

def foo():

global x

y = "local"

x = x \* 2

print(x)

print(y)

foo()

**Output**

global global

local

In the above code, we declare x as a global and y as a local variable in the foo(). Then, we use multiplication operator \* to modify the global variable x and we print both x and y.

After calling the foo(), the value of x becomes global global because we used the x \* 2 to print two times global. After that, we print the value of local variable y i.e local.

### **Example 5: Global variable and Local variable with same name**

x = 5

def foo():

x = 10

print("local x:", x)

foo()

print("global x:", x)

**Output**

local x: 10

global x: 5

In the above code, we used the same name x for both global variable and local variable. We get a different result when we print the same variable because the variable is declared in both scopes, i.e. the local scope inside foo() and global scope outside foo().

When we print the variable inside foo() it outputs local x: 10. This is called the local scope of the variable.

Similarly, when we print the variable outside the foo(), it outputs global x: 5. This is called the global scope of the variable.

## Nonlocal Variables

Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.

Let's see an example of how a nonlocal variable is used in Python.

We use nonlocal keywords to create nonlocal variables.

### **Example 6: Create a nonlocal variable**

def outer():

x = "local"

def inner():

nonlocal x

x = "nonlocal"

print("inner:", x)

inner()

print("outer:", x)

outer()

**Output**

inner: nonlocal

outer: nonlocal

In the above code, there is a nested inner() function. We use nonlocal keywords to create a nonlocal variable. The inner() function is defined in the scope of another function outer().

**Note** : If we change the value of a nonlocal variable, the changes appear in the local variable.

**Exception Handling:**

We can make certain mistakes while writing a program that lead to errors when we try to run it. A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:

* Syntax errors
* Logical errors (Exceptions)

## Python Syntax Errors

Error caused by not following the proper structure (syntax) of the language is called **syntax error** or **parsing error**.

Let's look at one example:

>>> if a < 3

File "<interactive input>", line 1

if a < 3

^

SyntaxError: invalid syntax

As shown in the example, an arrow indicates where the parser ran into the syntax error.

We can notice here that a colon : is missing in the if statement.

Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called **exceptions** or **logical errors**.

For instance, they occur when we try to open a file(for reading) that does not exist (FileNotFoundError), try to divide a number by zero (ZeroDivisionError), or try to import a module that does not exist (ImportError).

Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Let's look at how Python treats these errors:

>>> 1 / 0

Traceback (most recent call last):

File "<string>", line 301, in runcode

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

ZeroDivisionError: division by zero

>>> open("imaginary.txt")

Traceback (most recent call last):

File "<string>", line 301, in runcode

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

FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'

Python Built-in Exceptions

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur. We can view all the built-in exceptions using the built-in local() function as follows:

print(dir(locals()['\_\_builtins\_\_']))

locals()['\_\_builtins\_\_'] will return a module of built-in exceptions, functions, and attributes. dir allows us to list these attributes as strings.

Some of the common built-in exceptions in Python programming along with the error that cause them are listed below:

| Exception | Cause of Error |
| --- | --- |
| AssertionError | Raised when an assert statement fails. |
| AttributeError | Raised when attribute assignment or reference fails. |
| EOFError | Raised when the input() function hits end-of-file condition. |
| FloatingPointError | Raised when a floating point operation fails. |
| GeneratorExit | Raise when a generator's close() method is called. |
| ImportError | Raised when the imported module is not found. |
| IndexError | Raised when the index of a sequence is out of range. |
| KeyError | Raised when a key is not found in a dictionary. |
| KeyboardInterrupt | Raised when the user hits the interrupt key (Ctrl+C or Delete). |
| MemoryError | Raised when an operation runs out of memory. |
| NameError | Raised when a variable is not found in local or global scope. |
| NotImplementedError | Raised by abstract methods. |
| OSError | Raised when system operation causes system related error. |
| OverflowError | Raised when the result of an arithmetic operation is too large to be represented. |
| ReferenceError | Raised when a weak reference proxy is used to access a garbage collected referent. |
| RuntimeError | Raised when an error does not fall under any other category. |
| StopIteration | Raised by next() function to indicate that there is no further item to be returned by iterator. |
| SyntaxError | Raised by parser when syntax error is encountered. |
| IndentationError | Raised when there is incorrect indentation. |
| TabError | Raised when indentation consists of inconsistent tabs and spaces. |
| SystemError | Raised when interpreter detects internal error. |
| SystemExit | Raised by sys.exit() function. |
| TypeError | Raised when a function or operation is applied to an object of incorrect type. |
| UnboundLocalError | Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. |
| UnicodeError | Raised when a Unicode-related encoding or decoding error occurs. |
| UnicodeEncodeError | Raised when a Unicode-related error occurs during encoding. |
| UnicodeDecodeError | Raised when a Unicode-related error occurs during decoding. |
| UnicodeTranslateError | Raised when a Unicode-related error occurs during translating. |
| ValueError | Raised when a function gets an argument of correct type but improper value. |
| ZeroDivisionError | Raised when the second operand of division or modulo operation is zero. |

## Exceptions in Python

Python has many [built-in exceptions](https://www.programiz.com/python-programming/exceptions) that are raised when your program encounters an error (something in the program goes wrong).

When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

For example, let us consider a program where we have a [function](https://www.programiz.com/python-programming/function) A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.

If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

Catching Exceptions in Python

In Python, exceptions can be handled using a try statement.

The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

We can thus choose what operations to perform once we have caught the exception. Here is a simple example.

Program:

# Function which returns a/b

def AbyB(a , b):

try:

c = ((a+b) / (a-b))

print(c)

except ZeroDivisionError:

print( "a/b result in 0")

# Driver program to test above function

AbyB(5.0, 3.0)

AbyB(3.0, 3.0)

**Output:**

4.0

a/b result in 0

**Program:**

# Python program to handle simple runtime error

a = [1, 2, 3]

try:

print ("Second element = %d" %(a[1]))

# Throws error since there are only 3 elements in array

print ("Fourth element = %d" %(a[3]) )

except IndexError:

print ("An error occurred")

**Output:**

Second element = 2

An error occurred

# import module sys to get the type of exception

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except:

print("Oops!", sys.exc\_info()[0], "occurred.")

print("Next entry.")

print()

print("The reciprocal of", entry, "is", r)

**Output**

The entry is a

Oops! <class 'ValueError'> occurred.

Next entry.

The entry is 0

Oops! <class 'ZeroDivisionError'> occured.

Next entry.

The entry is 2

The reciprocal of 2 is 0.5

In this program, we loop through the values of the randomList list. As previously mentioned, the portion that can cause an exception is placed inside the try block.

If no exception occurs, the except block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (first and second values).

Here, we print the name of the exception using the exc\_info() function inside sys module. We can see that a causes ValueError and 0 causes ZeroDivisionError.

Since every exception in Python inherits from the base Exception class, we can also perform the above task in the following way:

# import module sys to get the type of exception

import sys

randomList = ['a', 0, 2]

for entry in randomList:

try:

print("The entry is", entry)

r = 1/int(entry)

break

except Exception as e:

print("Oops!", e.\_\_class\_\_, "occurred.")

print("Next entry.")

print()

print("The reciprocal of", entry, "is", r)

This program has the same output as the above program.

Catching Specific Exceptions in Python

In the above example, we did not mention any specific exception in the except clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause should catch.

A try clause can have any number of except clauses to handle different exceptions, however, only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

try:

# do something

pass

except ValueError:

# handle ValueError exception

pass

except (TypeError, ZeroDivisionError):

# handle multiple exceptions

# TypeError and ZeroDivisionError

pass

except:

# handle all other exceptions

pass

Raising Exceptions in Python

In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the raise keyword.

We can optionally pass values to the exception to clarify why that exception was raised.

>>> raise KeyboardInterrupt

Traceback (most recent call last):

...

KeyboardInterrupt

>>> raise MemoryError("This is an argument")

Traceback (most recent call last):

...

MemoryError: This is an argument

>>> try:

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

... if a <= 0:

... raise ValueError("That is not a positive number!")

... except ValueError as ve:

... print(ve)

...

Enter a positive integer: -2

That is not a positive number!

## Python try with else clause

In some situations, you might want to run a certain block of code if the code block inside try ran without any errors. For these cases, you can use the optional else keyword with the try statement.

**Note**: Exceptions in the else clause are not handled by the preceding except clauses.

Let's look at an example:

# program to print the reciprocal of even numbers

try:

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

assert num % 2 == 0

except:

print("Not an even number!")

else:

reciprocal = 1/num

print(reciprocal)

**Output**

If we pass an odd number:

Enter a number: 1

Not an even number!

If we pass an even number, the reciprocal is computed and displayed.

Enter a number: 4

0.25

However, if we pass 0, we get ZeroDivisionError as the code block inside else is not handled by preceding except.

Enter a number: 0

Traceback (most recent call last):

File "<string>", line 7, in <module>

reciprocal = 1/num

ZeroDivisionError: division by zero

Python try...finally

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.

For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).

In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.

Here is an example of [file operations](https://www.programiz.com/python-programming/file-operation) to illustrate this.

try:

f = open("test.txt",encoding = 'utf-8')

# perform file operations

finally:

f.close()

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

## Object Oriented Programming

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

* attributes
* behavior

Let's take an example:

A parrot is an object, as it has the following properties:

* name, age, color as attributes
* singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

1. Class
2. Object
3. Data Abstraction
4. Data Encapsulation
5. Inheritance
6. Polymorphism
7. Message Passing

**Class:**

A class is data abstraction of real world entities.

A class is combined together of data items and functions into a single unit.

A class is technical blue print of the object.

**Object:**

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

**Data Abstraction:**

Data Abstraction is process of showing external details and hiding background details.

**Data Encapsulation:**

Data Encapsulation is nothing but a together of data items and functions into a single unit.

Inheritance:

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Polymorphism:

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

Message Passing:

One object to another objects transfer messages is called Message Passing.

## Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

class Parrot:

pass

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

## Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

obj = Parrot()

Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

### **Example 1: Creating Class and Object in Python**

class Parrot:

# class attribute

species = "bird"

# instance attribute

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instantiate the Parrot class

blu = Parrot("Blu", 10)

woo = Parrot("Woo", 15)

# access the class attributes

print("Blu is a {}".format(blu.\_\_class\_\_.species))

print("Woo is also a {}".format(woo.\_\_class\_\_.species))

# access the instance attributes

print("{} is {} years old".format( blu.name, blu.age))

print("{} is {} years old".format( woo.name, woo.age))

**Output**

Blu is a bird

Woo is also a bird

Blu is 10 years old

Woo is 15 years old

In the above program, we created a class with the name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the \_\_init\_\_ method of the class. It is the initializer method that is first run as soon as the object is created.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

We can access the class attribute using \_\_class\_\_.species. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

To learn more about classes and objects, go to [Python Classes and Objects](https://www.programiz.com/python-programming/class)

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

### **Example 2 : Creating Methods in Python**

class Parrot:

# instance attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instance method

def sing(self, song):

return "{} sings {}".format(self.name, song)

def dance(self):

return "{} is now dancing".format(self.name)

# instantiate the object

blu = Parrot("Blu", 10)

# call our instance methods

print(blu.sing("'Happy'"))

print(blu.dance())

**Output**

Blu sings 'Happy'

Blu is now dancing

In the above program, we define two methods i.e sing() and dance(). These are called instance methods because they are called on an instance object i.e blu.

## Constructors in Python

Class functions that begin with double underscore \_\_ are called special functions as they have special meaning.

Of one particular interest is the \_\_init\_\_() function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

class ComplexNumber:

def \_\_init\_\_(self, r=0, i=0):

self.real = r

self.imag = i

def get\_data(self):

print(f'{self.real}+{self.imag}j')

# Create a new ComplexNumber object

num1 = ComplexNumber(2, 3)

# Call get\_data() method

# Output: 2+3j

num1.get\_data()

# Create another ComplexNumber object

# and create a new attribute 'attr'

num2 = ComplexNumber(5)

num2.attr = 10

# Output: (5, 0, 10)

print((num2.real, num2.imag, num2.attr))

# but c1 object doesn't have attribute 'attr'

# AttributeError: 'ComplexNumber' object has no attribute 'attr'

print(num1.attr)

**Output**

2+3j

(5, 0, 10)

Traceback (most recent call last):

File "<string>", line 27, in <module>

print(num1.attr)

AttributeError: 'ComplexNumber' object has no attribute 'attr'

In the above example, we defined a new class to represent complex numbers. It has two functions, \_\_init\_\_() to initialize the variables (defaults to zero) and get\_data() to display the number properly.

An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute attr for object num2 and read it as well. But this does not create that attribute for object num1.

## Deleting Attributes and Objects

Any attribute of an object can be deleted anytime, using the del statement. Try the following on the Python shell to see the output.

>>> num1 = ComplexNumber(2,3)

>>> del num1.imag

>>> num1.get\_data()

Traceback (most recent call last):

...

AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.get\_data

>>> num1.get\_data()

Traceback (most recent call last):

...

AttributeError: 'ComplexNumber' object has no attribute 'get\_data'

We can even delete the object itself, using the del statement.

>>> c1 = ComplexNumber(1,3)

>>> del c1

>>> c1

Traceback (most recent call last):

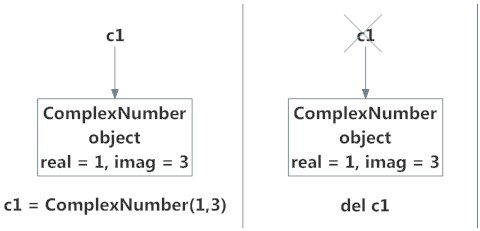
...

NameError: name 'c1' is not defined

Actually, it is more complicated than that. When we do c1 = ComplexNumber(1,3), a new instance object is created in memory and the name c1 binds with it.

On the command del c1, this binding is removed and the name c1 is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.

This automatic destruction of unreferenced objects in Python is also called garbage collection.

Deleting objects in Python removes the name binding

**Program:**

# Demo for class:

class employe:

def \_\_init\_\_(self,name,id):

self.name=name

self.id=id

def display(self):

print("ID:%d\nName:%s"%(self.id,self.name))

emp1=employe("Arjun",397)

emp1.display()

emp2=employe("Mallikarjun",398)

emp2.display()

**Output:**

ID:397

Name:Arjun

ID:398

Name:Mallikarjun

## Inheritance in Python

Inheritance is a powerful feature in object oriented programming.

It refers to defining a new [class](https://www.programiz.com/python-programming/class) with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

### **Python Inheritance Syntax**

class BaseClass:

Body of base class

class DerivedClass(BaseClass):

Body of derived class

Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code.

### **Example of Inheritance in Python**

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

class Polygon:

def \_\_init\_\_(self, no\_of\_sides):

self.n = no\_of\_sides

self.sides = [0 for i in range(no\_of\_sides)]

def inputSides(self):

self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

def dispSides(self):

for i in range(self.n):

print("Side",i+1,"is",self.sides[i])

This class has data attributes to store the number of sides n and magnitude of each side as a list called sides.

The inputSides() method takes in the magnitude of each side and dispSides() displays these side lengths.

A triangle is a polygon with 3 sides. So, we can create a class called Triangle which inherits from Polygon. This makes all the attributes of Polygon class available to the Triangle class.

We don't need to define them again (code reusability). Triangle can be defined as follows.

class Triangle(Polygon):

def \_\_init\_\_(self):

Polygon.\_\_init\_\_(self,3)

def findArea(self):

a, b, c = self.sides

# calculate the semi-perimeter

s = (a + b + c) / 2

area = (s\*(s-a)\*(s-b)\*(s-c)) \*\* 0.5

print('The area of the triangle is %0.2f' %area)

However, class Triangle has a new method findArea() to find and print the area of the triangle. Here is a sample run.

>>> t = Triangle()

>>> t.inputSides()

Enter side 1 : 3

Enter side 2 : 5

Enter side 3 : 4

>>> t.dispSides()

Side 1 is 3.0

Side 2 is 5.0

Side 3 is 4.0

>>> t.findArea()

The area of the triangle is 6.00

We can see that even though we did not define methods like inputSides() or dispSides() for class Triangle separately, we were able to use them.

If an attribute is not found in the class itself, the search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Single Level Inheritance:

One base class one child class have inheritance is called Single Level Inheritance.

# Single Inheritance demo:

class student:

def \_\_init\_\_(self,name,id):

self.name=name

self.id=id

def show(self):

print("Name of the Student :-",self.name)

print("ID of the Student :-",self.id)

class details(student):

def \_\_init\_\_(self,age,marks,grade):

self.age=age

self.marks=marks

self.grade=grade

def shows(self):

print("Age of the student :-",self.age)

print("Student marks :-",self.marks)

print("Grade :-",self.grade)

st=student("ARJUN",34)

st.show()

s1=details(23,456,"A")

s1.shows()

## Python Multilevel Inheritance

We can also inherit from a derived class. This is called multilevel inheritance. It can be of any depth in Python.

In multilevel inheritance, features of the base class and the derived class are inherited into the new derived class.

An example with corresponding visualization is given below.

class Base:

pass

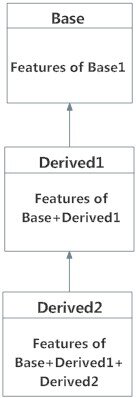
class Derived1(Base):

pass

class Derived2(Derived1):

pass

Here, the Derived1 class is derived from the Base class, and the Derived2 class is derived from the Derived1 class.



#Multilevel Inheritance:

class account:

def \_\_init\_\_(self,name,acc):

self.name=name

self.acc=acc

class add(account):

def \_\_init\_\_(self,name,acc,balance,mail):

account.\_\_init\_\_(self,name,acc)

self.balance=balance

self.mail=mail

def display(self):

print("Account Balance :-", self.balance)

print("Mail of the accountholder :-",self.mail )

class extra(add):

def \_\_init\_\_(self,name,acc,balance,mail,aa):

add.\_\_init\_\_(self,name,acc,balance,mail)

self.aa=aa

def see(self):

print("Name of the Account holder:-",self.name)

print("Account number :-",self.acc)

print("Account Balance :-", self.balance)

print("Mail of the accountholder :-",self.mail )

print("Aadhar of Account holder :-", self.aa)

ac =extra("Arjun",33030662839,3500.00,"arjunkumar0397ak@gmail.com",801571141878)

print(ac.see())

**Output:**

Name of the Account holder:- Arjun

Account number :- 33030662839

Account Balance :- 3500.0

Mail of the accountholder :- arjunkumar0397ak@gmail.com

Aadhar of Account holder :- 801571141878

None

## Python Multiple Inheritance

A [class](https://www.programiz.com/python-programming/class) can be derived from more than one base class in Python, similar to C++. This is called multiple inheritance.

In multiple inheritance, the features of all the base classes are inherited into the derived class. The syntax for multiple inheritance is similar to single [inheritance](https://www.programiz.com/python-programming/inheritance).

### **Example**

class Base1:

pass

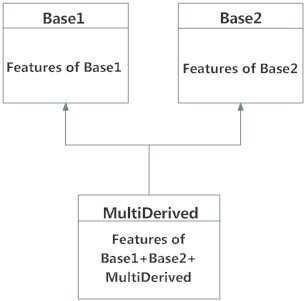
class Base2:

pass

class MultiDerived(Base1, Base2):

pass

Here, the MultiDerived class is derived from Base1 and Base2 classes.



The MultiDerived class inherits from both Base1 and Base2 classes.

Deriving properties from more than one parent class to one childs class is called Multiple inheritance.

Program:

# multiple\_inheritance:

class samsung:

def \_\_init\_\_(self):

self.str1="SAMSUNG"

return self.str1

class iphone:

def \_\_init\_\_(self):

self.str2="IPHONE"

return self.str2

class VIVO(samsung,iphone):

def \_\_init\_\_(self):

samsung.\_\_init\_\_(self)

iphone.\_\_init\_\_(self)

def print\_prop(self):

print(type(self).\_\_name\_\_,":",self.str1,"+",self.str2 )

x = VIVO()

x.print\_prop()

Output:

VIVO : SAMSUNG + IPHONE

## **Method Overriding**

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.

### Example

**class** Animal:

**def** speak(self):

**print**("speaking")

**class** Dog(Animal):

**def** speak(self):

**print**("Barking")

d = Dog()

d.speak()

**Output:**

Barking

**Program:**

# polymorphism Method Overriding:

class Bird:

def intro(self):

print("There are many types of birds.")

def flight(self):

print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

def flight(self):

print("Sparrows can fly.")

class ostrich(Bird):

def flight(self):

print("Ostriches cannot fly.")

obj\_bird = Bird()

obj\_spr = sparrow()

obj\_ost = ostrich()

obj\_bird.intro()

obj\_bird.flight()

obj\_spr.intro()

obj\_spr.flight()

obj\_ost.intro()

obj\_ost.flight()

**Output:**

There are many types of birds.

Most of the birds can fly but some cannot.

There are many types of birds.

Sparrows can fly.

There are many types of birds.

Ostriches cannot fly.

## Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

### **Example : Using Polymorphism in Python**

class Parrot:

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin:

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

flying\_test(peggy)

**Output**

Parrot can fly

Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have a common fly() method. However, their functions are different.

To use polymorphism, we created a common interface i.e flying\_test() function that takes any object and calls the object's fly() method. Thus, when we passed the blu and peggy objects in the flying\_test() function, it ran effectively.

# **Python File Handling**

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

The file handling plays an important role when the data needs to be stored permanently into the file. A file is a named location on disk to store related information. We can access the stored information (non-volatile) after the program termination.

The file-handling implementation is slightly lengthy or complicated in the other programming language, but it is easier and shorter in Python.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

* Open a file
* Read or write - Performing operation
* Close the file

## **Opening a file**

Python provides an **open()** function that accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

**Syntax:**

file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

| **SN** | **Access mode** | **Description** |
| --- | --- | --- |
| 1 | r | It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed. |
| 2 | rb | It opens the file to read-only in binary format. The file pointer exists at the beginning of the file. |
| 3 | r+ | It opens the file to read and write both. The file pointer exists at the beginning of the file. |
| 4 | rb+ | It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file. |
| 5 | w | It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file. |
| 6 | wb | It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file. |
| 7 | w+ | It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file. |
| 8 | wb+ | It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file. |
| 9 | a | It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name. |
| 10 | ab | It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name. |
| 11 | a+ | It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name. |
| 12 | ab+ | It opens a file to append and read both in binary format. The file pointer remains at the end of the file. |

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

## **Example**

#opens the file file.txt in read mode

fileptr = open("file.txt","r")

**if** fileptr:

**print**("file is opened successfully")

**Output:**

<class '\_io.TextIOWrapper'>

file is opened successfully

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement

## **The close() method**

Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done.

The syntax to use the **close()** method is given below.

**Syntax**

fileobject.close()

Consider the following example.

# opens the file file.txt in read mode

fileptr = open("file.txt","r")

**if** fileptr:

**print**("file is opened successfully")

#closes the opened file

fileptr.close()

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

**try**:

 fileptr = open("file.txt")

 # perform file operations

**finally**:

fileptr.close()

## **The with statement**

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with the statement is given below.

1. with open(<file name>, <access mode>) as <file-pointer>:
2. #statement suite

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the **close()** function. It doesn't let the file to corrupt.

Consider the following example.

### Example

1. with open("file.txt",'r') as f:
2. content = f.read();
3. **print**(content)

## **Writing the file**

To write some text to a file, we need to open the file using the open method with one of the following access modes.

**w:** It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

**a:** It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Consider the following example.

### Example

1. # open the file.txt in append mode. Create a new file if no such file exists.
2. fileptr = open("file2.txt", "w")
4. # appending the content to the file
5. fileptr.write('''''Python is the modern day language. It makes things so simple.
6. It is the fastest-growing programing language''')
8. # closing the opened the file
9. fileptr.close()

**Output:**

File2.txt

Python is the modern-day language. It makes things so simple. It is the fastest growing programming language.

**Snapshot of the file2.txt**

We have opened the file in **w** mode. The **file1.txt** file doesn't exist, it created a new file and we have written the content in the file using the **write()** function.

### Example 2

1. #open the file.txt in write mode.
2. fileptr = open("file2.txt","a")
4. #overwriting the content of the file
5. fileptr.write(" Python has an easy syntax and user-friendly interaction.")
7. #closing the opened file
8. fileptr.close()

**Output:**

Python is the modern day language. It makes things so simple.

It is the fastest growing programing language Python has an easy syntax and user-friendly interaction.

**Snapshot of the file2.txt**

We can see that the content of the file is modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from the file. It can read the data in the text as well as a binary format.

The syntax of the **read()** method is given below.

**Syntax:**

1. fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

### Example

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r")
3. #stores all the data of the file into the variable content
4. content = fileptr.read(10)
5. # prints the type of the data stored in the file
6. **print**(type(content))
7. #prints the content of the file
8. **print**(content)
9. #closes the opened file
10. fileptr.close()

**Output:**

<class 'str'>

Python is

In the above code, we have read the content of **file2.txt** by using the **read()** function. We have passed count value as ten which means it will read the first ten characters from the file.

If we use the following line, then it will print all content of the file.

1. content = fileptr.read()
2. **print**(content)

**Output:**

Python is the modern-day language. It makes things so simple.

It is the fastest-growing programing language Python has easy an syntax and user-friendly interaction.

### Read file through for loop

We can read the file using for loop. Consider the following example.

1. #open the file.txt in read mode. causes an error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #running a for loop
4. **for** i **in** fileptr:
5. **print**(i) # i contains each line of the file

**Output:**

Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.

## **Read Lines of the file**

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

### Example 1: Reading lines using readline() function

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #stores all the data of the file into the variable content
4. content = fileptr.readline()
5. content1 = fileptr.readline()
6. #prints the content of the file
7. **print**(content)
8. **print**(content1)
9. #closes the opened file
10. fileptr.close()

**Output:**

Python is the modern day language.

It makes things so simple.

We called the **readline()** function two times that's why it read two lines from the file.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

### Example 2: Reading Lines Using readlines() function

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r");
4. #stores all the data of the file into the variable content
5. content = fileptr.readlines()
7. #prints the content of the file
8. **print**(content)
10. #closes the opened file
11. fileptr.close()

**Output:**

['Python is the modern day language.\n', 'It makes things so simple.\n', 'Python has easy syntax and user-friendly

## **Creating a new file**

The new file can be created by using one of the following access modes with the function open().

**x:** it creates a new file with the specified name. It causes an error a file exists with the same name.

**a:** It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

**w:** It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

### Example 1

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","x")
3. **print**(fileptr)
4. **if** fileptr:
5. **print**("File created successfully")

**Output:**

<\_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>

File created successfully

## **File Pointer positions**

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. Consider the following example.

1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
4. #initially the filepointer is at 0
5. **print**("The filepointer is at byte :",fileptr.tell())
7. #reading the content of the file
8. content = fileptr.read();
10. #after the read operation file pointer modifies. tell() returns the location of the fileptr.
12. **print**("After reading, the filepointer is at:",fileptr.tell())

**Output:**

The filepointer is at byte : 0

After reading, the filepointer is at: 117

## **Modifying file pointer position**

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.

The syntax to use the seek() method is given below.

**Syntax:**

1. <file-ptr>.seek(offset[, **from**)

The seek() method accepts two parameters:

**offset:** It refers to the new position of the file pointer within the file.

**from:** It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

### Example

1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
4. #initially the filepointer is at 0
5. **print**("The filepointer is at byte :",fileptr.tell())
7. #changing the file pointer location to 10.
8. fileptr.seek(10);
10. #tell() returns the location of the fileptr.
11. **print**("After reading, the filepointer is at:",fileptr.tell())

**Output:**

The filepointer is at byte : 0

After reading, the filepointer is at: 10

## **Python OS module**

### Renaming the file

The Python **os** module enables interaction with the operating system. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the rename() method to rename the specified file to a new name. The syntax to use the **rename()** method is given below.

**Syntax:**

1. rename(current-name, new-name)

The first argument is the current file name and the second argument is the modified name. We can change the file name bypassing these two arguments.

**Example 1:**

1. **import** os
3. #rename file2.txt to file3.txt
4. os.rename("file2.txt","file3.txt")

**Output:**

The above code renamed current **file2.txt** to **file3.txt**

### Removing the file

The os module provides the **remove()** method which is used to remove the specified file. The syntax to use the **remove()** method is given below.

1. remove(file-name)

**Example 1**

1. **import** os;
2. #deleting the file named file3.txt
3. os.remove("file3.txt")

## **Creating the new directory**

The **mkdir()** method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

**Syntax:**

1. mkdir(directory name)

**Example 1**

1. **import** os
3. #creating a new directory with the name new
4. os.mkdir("new")

## **The getcwd() method**

This method returns the current working directory.

The syntax to use the getcwd() method is given below.

**Syntax**

1. os.getcwd()

**Example**

1. **import** os
2. os.getcwd()

**Output:**

'C:\\Users\\DEVANSH SHARMA'

## **Changing the current working directory**

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is given below.

**Syntax**

1. chdir("new-directory")

### Example

1. **import** os
2. # Changing current directory with the new directiory
3. os.chdir("C:\\Users\\DEVANSH SHARMA\\Documents")
4. #It will display the current working directory
5. os.getcwd()

**Output:**

'C:\\Users\\DEVANSH SHARMA\\Documents'

## **Deleting directory**

The rmdir() method is used to delete the specified directory.

The syntax to use the rmdir() method is given below.

**Syntax**

1. os.rmdir(directory name)

**Example 1**

1. **import** os
2. #removing the new directory
3. os.rmdir("directory\_name")

It will remove the specified directory.

## **The file related methods**

The file object provides the following methods to manipulate the files on various operating systems.

| **SN** | **Method** | **Description** |
| --- | --- | --- |
| 1 | file.close() | It closes the opened file. The file once closed, it can't be read or write anymore. |
| 2 | File.fush() | It flushes the internal buffer. |
| 3 | File.fileno() | It returns the file descriptor used by the underlying implementation to request I/O from the OS. |
| 4 | File.isatty() | It returns true if the file is connected to a TTY device, otherwise returns false. |
| 5 | File.next() | It returns the next line from the file. |
| 6 | File.read([size]) | It reads the file for the specified size. |
| 7 | File.readline([size]) | It reads one line from the file and places the file pointer to the beginning of the new line. |
| 8 | File.readlines([sizehint]) | It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function. |
| 9 | File.seek(offset[,from) | It modifies the position of the file pointer to a specified offset with the specified reference. |
| 10 | File.tell() | It returns the current position of the file pointer within the file. |
| 11 | File.truncate([size]) | It truncates the file to the optional specified size. |
| 12 | File.write(str) | It writes the specified string to a file |
| 13 | File.writelines(seq) | It writes a sequence of the strings to a file. |

## **Python Generator?**

Python Generators are the functions that return the traversal object and used to create iterators. It traverses the entire items at once. The generator can also be an expression in which syntax is similar to the list comprehension in Python.

There is a lot of complexity in creating iteration in Python; we need to implement **\_\_iter\_\_()** and **\_\_next\_\_()** method to keep track of internal states.

It is a lengthy process to create iterators. That's why the generator plays an essential role in simplifying this process. If there is no value found in iteration, it raises **StopIteration** exception.

## **How to Create Generator function in Python?**

It is quite simple to create a generator in Python. It is similar to the normal function defined by the **def** keyword and uses a **yield** keyword instead of return. Or we can say that if the body of any function contains a **yield** statement, it automatically becomes a generator function. Consider the following example:

1. def simple():
2. **for** i in range(10):
3. **if**(i%2==0):
4. yield i
6. #Successive Function call using **for** loop
7. **for** i in simple():
8. print(i)

**Output:**

0

2

4

6

8

### yield vs. return

The **yield** statement is responsible for controlling the flow of the generator function. It pauses the function execution by saving all states and yielded to the caller. Later it resumes execution when a successive function is called. We can use the multiple yield statement in the generator function.

The return statement **returns** a value and terminates the whole function and only one return statement can be used in the function.

**Using multiple yield Statement**

We can use the multiple yield statement in the generator function. Consider the following example.

1. def multiple\_yield():
2. str1 = "First String"
3. yield str1
5. str2 = "Second string"
6. yield str2
8. str3 = "Third String"
9. yield str3
10. obj = multiple\_yield()
11. print(next(obj))
12. print(next(obj))
13. print(next(obj))

**Output:**

First String

Second string

Third String

### Difference between Generator function and Normal function

* Normal function contains only one L**return** statement whereas generator function can contain one or more **yield** statement.
* When the generator functions are called, the normal function is paused immediately and control transferred to the caller.
* Local variable and their states are remembered between successive calls.
* StopIteration exception is raised automatically when the function terminates.

### Generator Expression

We can easily create a generator expression without using user-defined function. It is the same as the lambda function which creates an anonymous function; the generator's expressions create an anonymous generator function.

The representation of generator expression is similar to the Python list comprehension. The only difference is that **square bracket is replaced by round parentheses**. The list comprehension calculates the entire list, whereas the generator expression calculates one item at a time.

Consider the following example:

1. list = [1,2,3,4,5,6,7]
3. # List Comprehension
4. z = [x\*\*3 **for** x in list]
6. # Generator expression
7. a = (x\*\*3 **for** x in list)
9. print(a)
10. print(z)

**Output:**

<generator object <genexpr> at 0x01BA3CD8>

[1, 8, 27, 64, 125, 216, 343]

In the above program, list comprehension has returned the list of cube of elements whereas generator expression has returned the reference of calculated value. Instead of applying a **for loop**, we can also call **next()** on the generator object. Let's consider another example:

1. list = [1,2,3,4,5,6]
3. z = (x\*\*3 **for** x in list)
5. print(next(z))
7. print(next(z))
9. print(next(z))
11. print(next(z))

**Output:**

1

8

27

64

#### Note:- When we call the next(), Python calls \_\_next\_\_() on the function in which we have passed it as a parameter.

In the above program, we have used the **next()** function, which returned the next item of the list.

**Example:** Write a program to print the table of the given number using the generator.

1. def table(n):
2. **for** i in range(1,11):
3. yield n\*i
4. i = i+1
6. **for** i in table(15):
7. print(i)

**Output:**

15

30

45

60

75

90

105

120

135

150

In the above example, a generator function is iterating using for loop.

## **Advantages of Generators**

There are various advantages of Generators. Few of them are given below:

### 1. Easy to implement

Generators are easy to implement as compared to the iterator. In iterator, we have to implement **\_\_iter\_\_()** and **\_\_next\_\_()** function.

### 2. Memory efficient

Generators are memory efficient for a large number of sequences. The normal function returns a sequence of the list which creates an entire sequence in memory before returning the result, but the generator function calculates the value and pause their execution. It resumes for successive call. An infinite sequence generator is a great example of memory optimization. Let's discuss it in the below example by using **sys.getsizeof()** function.

1. **import** sys
2. # List comprehension
3. nums\_squared\_list = [i \* 2 **for** i in range(1000)]
4. print(sys.getsizeof("Memory in Bytes:"nums\_squared\_list))
5. # Generator Expression
6. nums\_squared\_gc = (i \*\* 2 **for** i in range(1000))
7. print(sys.getsizeof("Memory in Bytes:", nums\_squared\_gc))

**Output:**

Memory in Bytes: 4508

Memory in Bytes: 56

We can observe from the above output that list comprehension is using 4508 bytes of memory, whereas generator expression is using 56 bytes of memory. It means that generator objects are much efficient than the list compression.

### 3. Pipelining with Generators

Data Pipeline provides the facility to process large datasets or stream of data without using extra computer memory.

Suppose we have a log file from a famous restaurant. The log file has a column (4th column) that keeps track of the number of burgers sold every hour and we want to sum it to find the total number of burgers sold in 4 years. In that scenario, the generator can generate a pipeline with a series of operations. Below is the code for it:

1. with open('sells.log') as file:
2. burger\_col = (line[3] **for** line in file)  per\_hour = (**int**(x) **for** x in burger\_col **if** x != 'N/A')
3. print("Total burgers sold = ",sum(per\_hour))

# **Python Regular Expressions**

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

| **SN** | **Function** | **Description** |
| --- | --- | --- |
| 1 | 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. |
| 2 | search | This method returns the match object if there is a match found in the string. |
| 3 | findall | It returns a list that contains all the matches of a pattern in the string. |
| 4 | split | Returns a list in which the string has been split in each match. |
| 5 | 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.

| **SN** | **Set** | **Description** |
| --- | --- | --- |
| 1 | [arn] | Returns a match if the string contains any of the specified characters in the set. |
| 2 | [a-n] | Returns a match if the string contains any of the characters between a to n. |
| 3 | [^arn] | Returns a match if the string contains the characters except a, r, and n. |
| 4 | [0123] | Returns a match if the string contains any of the specified digits. |
| 5 | [0-9] | Returns a match if the string contains any digit between 0 and 9. |
| 6 | [0-5][0-9] | Returns a match if the string contains any digit between 00 and 59. |
| 10 | [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.

**Example**

1. **import** re
3. str = "How are you. How is everything"
5. matches = re.findall("How", str)
7. **print**(matches)
9. **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

1. **import** re
3. str = "How are you. How is everything"
5. matches = re.search("How", str)
7. **print**(type(matches))
9. **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

There are the following methods associated with the Match object.

1. **span():** It returns the tuple containing the starting and end position of the match.
2. **string():** It returns a string passed into the function.
3. **group():** The part of the string is returned where the match is found.

### Example

1. **import** re
3. str = "How are you. How is everything"
5. matches = re.search("How", str)
7. **print**(matches.span())
9. **print**(matches.group())
11. **print**(matches.string)

**Output:**

(0, 3)

How

How are you. How is everything

# **Python Tkinter Tutorial**

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

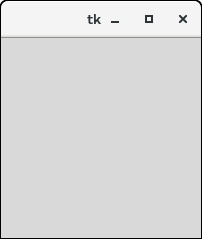
Developing desktop based applications with python Tkinter is not a complex task. An empty Tkinter top-level window can be created by using the following steps.

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

1. # !/usr/bin/python3
2. **from** tkinter **import** \*
3. #creating the application main window.
4. top = Tk()
5. #Entering the event main loop
6. top.mainloop()

**Output:**



## **Tkinter widgets**

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

| **SN** | **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. |

## **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.

1. The pack() method
2. The grid() method
3. The place() method

Let's discuss each one of them in detail.

### 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.

### syntax

1. 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

1. # !/usr/bin/python3
2. **from** tkinter **import** \*
3. parent = Tk()
4. redbutton = Button(parent, text = "Red", fg = "red")
5. redbutton.pack( side = LEFT)
6. greenbutton = Button(parent, text = "Black", fg = "black")
7. greenbutton.pack( side = RIGHT )
8. bluebutton = Button(parent, text = "Blue", fg = "blue")
9. bluebutton.pack( side = TOP )
10. blackbutton = Button(parent, text = "Green", fg = "red")
11. blackbutton.pack( side = BOTTOM)
12. parent.mainloop()

**Output:**



### 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.

### Syntax

1. 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

1. # !/usr/bin/python3
2. **from** tkinter **import** \*
3. parent = Tk()
4. name = Label(parent,text = "Name").grid(row = 0, column = 0)
5. e1 = Entry(parent).grid(row = 0, column = 1)
6. password = Label(parent,text = "Password").grid(row = 1, column = 0)
7. e2 = Entry(parent).grid(row = 1, column = 1)
8. submit = Button(parent, text = "Submit").grid(row = 4, column = 0)
9. parent.mainloop()

**Output:**



### Python Tkinter place() method

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

### Syntax

1. widget.place(options)

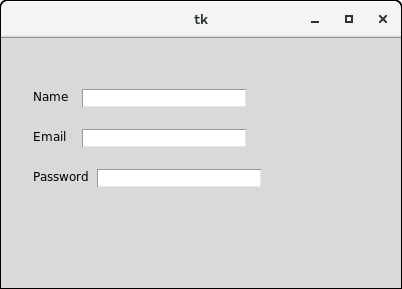
A list of possible options is given below.

* **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

1. # !/usr/bin/python3
2. **from** tkinter **import** \*
3. top = Tk()
4. top.geometry("400x250")
5. name = Label(top, text = "Name").place(x = 30,y = 50)
6. email = Label(top, text = "Email").place(x = 30, y = 90)
7. password = Label(top, text = "Password").place(x = 30, y = 130)
8. e1 = Entry(top).place(x = 80, y = 50)
9. e2 = Entry(top).place(x = 80, y = 90)
10. e3 = Entry(top).place(x = 95, y = 130)
11. top.mainloop()

**Output:**



# **Python Tkinter Button**

The button widget is used to add various types of buttons to the python application. Python allows us to configure the look of the button according to our requirements. Various options can be set or reset depending upon the requirements.

We can also associate a method or function with a button which is called when the button is pressed.

The syntax to use the button widget is given below.

### Syntax

1. W = Button(parent, options)

A list of possible options is given below.

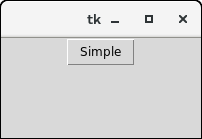
| **SN** | **Option** | **Description** |
| --- | --- | --- |
| 1 | activebackground | It represents the background of the button when the mouse hover the button. |
| 2 | activeforeground | It represents the font color of the button when the mouse hover the button. |
| 3 | Bd | It represents the border width in pixels. |
| 4 | Bg | It represents the background color of the button. |
| 5 | Command | It is set to the function call which is scheduled when the function is called. |
| 6 | Fg | Foreground color of the button. |
| 7 | Font | The font of the button text. |
| 8 | Height | The height of the button. The height is represented in the number of text lines for the textual lines or the number of pixels for the images. |
| 10 | Highlightcolor | The color of the highlight when the button has the focus. |
| 11 | Image | It is set to the image displayed on the button. |
| 12 | justify | It illustrates the way by which the multiple text lines are represented. It is set to LEFT for left justification, RIGHT for the right justification, and CENTER for the center. |
| 13 | Padx | Additional padding to the button in the horizontal direction. |
| 14 | pady | Additional padding to the button in the vertical direction. |
| 15 | Relief | It represents the type of the border. It can be SUNKEN, RAISED, GROOVE, and RIDGE. |
| 17 | State | This option is set to DISABLED to make the button unresponsive. The ACTIVE represents the active state of the button. |
| 18 | Underline | Set this option to make the button text underlined. |
| 19 | Width | The width of the button. It exists as a number of letters for textual buttons or pixels for image buttons. |
| 20 | Wraplength | If the value is set to a positive number, the text lines will be wrapped to fit within this length. |

### Example

1. #python application to create a simple button
3. **from** tkinter **import** \*

6. top = Tk()
8. top.geometry("200x100")
10. b = Button(top,text = "Simple")
12. b.pack()
14. top.mainaloop()

**Output:**



### Example

1. **from** tkinter **import** \*
3. top = Tk()
5. top.geometry("200x100")
7. **def** fun():
8. messagebox.showinfo("Hello", "Red Button clicked")

11. b1 = Button(top,text = "Red",command = fun,activeforeground = "red",activebackground = "pink",pady=10)
13. b2 = Button(top, text = "Blue",activeforeground = "blue",activebackground = "pink",pady=10)
15. b3 = Button(top, text = "Green",activeforeground = "green",activebackground = "pink",pady = 10)
17. b4 = Button(top, text = "Yellow",activeforeground = "yellow",activebackground = "pink",pady = 10)
19. b1.pack(side = LEFT)
21. b2.pack(side = RIGHT)
23. b3.pack(side = TOP)
25. b4.pack(side = BOTTOM)
27. top.mainloop()

**Output:**


# **Python Tkinter Checkbutton**

The Checkbutton is used to track the user's choices provided to the application. In other words, we can say that Checkbutton is used to implement the on/off selections.

The Checkbutton can contain the text or images. The Checkbutton is mostly used to provide many choices to the user among which, the user needs to choose the one. It generally implements many of many selections.

The syntax to use the checkbutton is given below.

### Syntax

1. w = checkbutton(master, options)

A list of possible options is given below.

7.2M

164

Exception Handling in Java - Javatpoint

| **SN** | **Option** | **Description** |
| --- | --- | --- |
| 1 | activebackground | It represents the background color when the checkbutton is under the cursor. |
| 2 | activeforeground | It represents the foreground color of the checkbutton when the checkbutton is under the cursor. |
| 3 | Bg | The background color of the button. |
| 4 | bitmap | It displays an image (monochrome) on the button. |
| 5 | Bd | The size of the border around the corner. |
| 6 | command | It is associated with a function to be called when the state of the checkbutton is changed. |
| 7 | Cursor | The mouse pointer will be changed to the cursor name when it is over the checkbutton. |
| 8 | disableforeground | It is the color which is used to represent the text of a disabled checkbutton. |
| 9 | Font | It represents the font of the checkbutton. |
| 10 | Fg | The foreground color (text color) of the checkbutton. |
| 11 | Height | It represents the height of the checkbutton (number of lines). The default height is 1. |
| 12 | highlightcolor | The color of the focus highlight when the checkbutton is under focus. |
| 13 | Image | The image used to represent the checkbutton. |
| 14 | Justify | This specifies the justification of the text if the text contains multiple lines. |
| 15 | offvalue | The associated control variable is set to 0 by default if the button is unchecked. We can change the state of an unchecked variable to some other one. |
| 16 | onvalue | The associated control variable is set to 1 by default if the button is checked. We can change the state of the checked variable to some other one. |
| 17 | Padx | The horizontal padding of the checkbutton |
| 18 | Pady | The vertical padding of the checkbutton. |
| 19 | Relief | The type of the border of the checkbutton. By default, it is set to FLAT. |
| 20 | selectcolor | The color of the checkbutton when it is set. By default, it is red. |
| 21 | selectimage | The image is shown on the checkbutton when it is set. |
| 22 | State | It represents the state of the checkbutton. By default, it is set to normal. We can change it to DISABLED to make the checkbutton unresponsive. The state of the checkbutton is ACTIVE when it is under focus. |
| 24 | underline | It represents the index of the character in the text which is to be underlined. The indexing starts with zero in the text. |
| 25 | variable | It represents the associated variable that tracks the state of the checkbutton. |
| 26 | Width | It represents the width of the checkbutton. It is represented in the number of characters that are represented in the form of texts. |
| 27 | wraplength | If this option is set to an integer number, the text will be broken into the number of pieces. |

## **Methods**

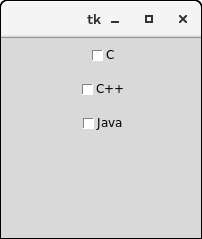
The methods that can be called with the Checkbuttons are described in the following table.

| **SN** | **Method** | **Description** |
| --- | --- | --- |
| 1 | deselect() | It is called to turn off the checkbutton. |
| 2 | flash() | The checkbutton is flashed between the active and normal colors. |
| 3 | invoke() | This will invoke the method associated with the checkbutton. |
| 4 | select() | It is called to turn on the checkbutton. |
| 5 | toggle() | It is used to toggle between the different Checkbuttons. |

### Example

1. **from** tkinter **import** \*
3. top = Tk()
5. top.geometry("200x200")
7. checkvar1 = IntVar()
9. checkvar2 = IntVar()
11. checkvar3 = IntVar()
13. chkbtn1 = Checkbutton(top, text = "C", variable = checkvar1, onvalue = 1, offvalue = 0, height = 2, width = 10)
15. chkbtn2 = Checkbutton(top, text = "C++", variable = checkvar2, onvalue = 1, offvalue = 0, height = 2, width = 10)
17. chkbtn3 = Checkbutton(top, text = "Java", variable = checkvar3, onvalue = 1, offvalue = 0, height = 2, width = 10)
19. chkbtn1.pack()
21. chkbtn2.pack()
23. chkbtn3.pack()
25. top.mainloop()

**Output:**



# **Python Tkinter Listbox**

The Listbox widget is used to display the list items to the user. We can place only text items in the Listbox and all text items contain the same font and color.

The user can choose one or more items from the list depending upon the configuration.

The syntax to use the Listbox is given below.

1. w = Listbox(parent, options)

A list of possible options is given below.

| **SN** | **Option** | **Description** |
| --- | --- | --- |
| 1 | Bg | The background color of the widget. |
| 2 | Bd | It represents the size of the border. Default value is 2 pixel. |
| 3 | Cursor | The mouse pointer will look like the cursor type like dot, arrow, etc. |
| 4 | Font | The font type of the Listbox items. |
| 5 | Fg | The color of the text. |
| 6 | Height | It represents the count of the lines shown in the Listbox. The default value is 10. |
| 7 | highlightcolor | The color of the Listbox items when the widget is under focus. |
| 8 | highlightthickness | The thickness of the highlight. |
| 9 | Relief | The type of the border. The default is SUNKEN. |
| 10 | selectbackground | The background color that is used to display the selected text. |
| 11 | selectmode | It is used to determine the number of items that can be selected from the list. It can set to BROWSE, SINGLE, MULTIPLE, EXTENDED. |
| 12 | Width | It represents the width of the widget in characters. |
| 13 | xscrollcommand | It is used to let the user scroll the Listbox horizontally. |
| 14 | yscrollcommand | It is used to let the user scroll the Listbox vertically. |

## **Methods**

There are the following methods associated with the Listbox.

| **SN** | **Method** | **Description** |
| --- | --- | --- |
| 1 | activate(index) | It is used to select the lines at the specified index. |
| 2 | curselection() | It returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple. |
| 3 | delete(first, last = None) | It is used to delete the lines which exist in the given range. |
| 4 | get(first, last = None) | It is used to get the list items that exist in the given range. |
| 5 | index(i) | It is used to place the line with the specified index at the top of the widget. |
| 6 | insert(index, \*elements) | It is used to insert the new lines with the specified number of elements before the specified index. |
| 7 | nearest(y) | It returns the index of the nearest line to the y coordinate of the Listbox widget. |
| 8 | see(index) | It is used to adjust the position of the listbox to make the lines specified by the index visible. |
| 9 | size() | It returns the number of lines that are present in the Listbox widget. |
| 10 | xview() | This is used to make the widget horizontally scrollable. |
| 11 | xview\_moveto(fraction) | It is used to make the listbox horizontally scrollable by the fraction of width of the longest line present in the listbox. |
| 12 | xview\_scroll(number, what) | It is used to make the listbox horizontally scrollable by the number of characters specified. |
| 13 | yview() | It allows the Listbox to be vertically scrollable. |
| 14 | yview\_moveto(fraction) | It is used to make the listbox vertically scrollable by the fraction of width of the longest line present in the listbox. |
| 15 | yview\_scroll (number, what) | It is used to make the listbox vertically scrollable by the number of characters specified. |

### Example 1

1. # !/usr/bin/python3
3. **from** tkinter **import** \*
5. top = Tk()
7. top.geometry("200x250")
9. lbl = Label(top,text = "A list of favourite countries...")
11. listbox = Listbox(top)
13. listbox.insert(1,"India")
15. listbox.insert(2, "USA")
17. listbox.insert(3, "Japan")
19. listbox.insert(4, "Austrelia")
21. lbl.pack()
22. listbox.pack()
24. top.mainloop()

**Output:**



### Example 2: Deleting the active items from the list

1. # !/usr/bin/python3
3. **from** tkinter **import** \*
5. top = Tk()
7. top.geometry("200x250")
9. lbl = Label(top,text = "A list of favourite countries...")
11. listbox = Listbox(top)
13. listbox.insert(1,"India")
15. listbox.insert(2, "USA")
17. listbox.insert(3, "Japan")
19. listbox.insert(4, "Austrelia")
21. #this button will delete the selected item from the list
23. btn = Button(top, text = "delete", command = **lambda** listbox=listbox: listbox.delete(ANCHOR))
25. lbl.pack()

28. listbox.pack()
30. btn.pack()
31. top.mainloop()

**Output:**



After pressing the delete button.



# **Python Tkinter Radiobutton**

The Radiobutton widget is used to implement one-of-many selection in the python application. It shows multiple choices to the user out of which, the user can select only one out of them. We can associate different methods with each of the radiobutton.

We can display the multiple line text or images on the radiobuttons. To keep track the user's selection the radiobutton, it is associated with a single variable. Each button displays a single value for that particular variable.

The syntax to use the Radiobutton is given below.

### Syntax

1. w = Radiobutton(top, options)

| **SN** | **Option** | **Description** |
| --- | --- | --- |
| 1 | activebackground | The background color of the widget when it has the focus. |
| 2 | activeforeground | The font color of the widget text when it has the focus. |
| 3 | anchor | It represents the exact position of the text within the widget if the widget contains more space than the requirement of the text. The default value is CENTER. |
| 4 | Bg | The background color of the widget. |
| 5 | bitmap | It is used to display the graphics on the widget. It can be set to any graphical or image object. |
| 6 | borderwidth | It represents the size of the border. |
| 7 | command | This option is set to the procedure which must be called every-time when the state of the radiobutton is changed. |
| 8 | Cursor | The mouse pointer is changed to the specified cursor type. It can be set to the arrow, dot, etc. |
| 9 | Font | It represents the font type of the widget text. |
| 10 | Fg | The normal foreground color of the widget text. |
| 11 | Height | The vertical dimension of the widget. It is specified as the number of lines (not pixel). |
| 12 | highlightcolor | It represents the color of the focus highlight when the widget has the focus. |
| 13 | highlightbackground | The color of the focus highlight when the widget is not having the focus. |
| 14 | Image | It can be set to an image object if we want to display an image on the radiobutton instead the text. |
| 15 | Justify | It represents the justification of the multi-line text. It can be set to CENTER(default), LEFT, or RIGHT. |
| 16 | Padx | The horizontal padding of the widget. |
| 17 | Pady | The vertical padding of the widget. |
| 18 | Relief | The type of the border. The default value is FLAT. |
| 19 | selectcolor | The color of the radio button when it is selected. |
| 20 | selectimage | The image to be displayed on the radiobutton when it is selected. |
| 21 | State | It represents the state of the radio button. The default state of the Radiobutton is NORMAL. However, we can set this to DISABLED to make the radiobutton unresponsive. |
| 22 | Text | The text to be displayed on the radiobutton. |
| 23 | textvariable | It is of String type that represents the text displayed by the widget. |
| 24 | underline | The default value of this option is -1, however, we can set this option to the number of character which is to be underlined. |
| 25 | Value | The value of each radiobutton is assigned to the control variable when it is turned on by the user. |
| 26 | variable | It is the control variable which is used to keep track of the user's choices. It is shared among all the radiobuttons. |
| 27 | Width | The horizontal dimension of the widget. It is represented as the number of characters. |
| 28 | wraplength | We can wrap the text to the number of lines by setting this option to the desired number so that each line contains only that number of characters. |

## **Methods**

The radiobutton widget provides the following methods.

5.9M

146

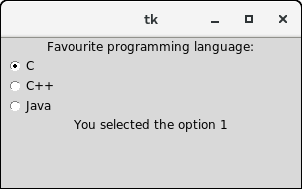
How to find Nth Highest Salary in SQL

| **SN** | **Method** | **Description** |
| --- | --- | --- |
| 1 | deselect() | It is used to turn of the radiobutton. |
| 2 | flash() | It is used to flash the radiobutton between its active and normal colors few times. |
| 3 | invoke() | It is used to call any procedure associated when the state of a Radiobutton is changed. |
| 4 | select() | It is used to select the radiobutton. |

### Example

1. **from** tkinter **import** \*
3. **def** selection():
4. selection = "You selected the option " + str(radio.get())
5. label.config(text = selection)
7. top = Tk()
8. top.geometry("300x150")
9. radio = IntVar()
10. lbl = Label(text = "Favourite programming language:")
11. lbl.pack()
12. R1 = Radiobutton(top, text="C", variable=radio, value=1,
13. command=selection)
14. R1.pack( anchor = W )
16. R2 = Radiobutton(top, text="C++", variable=radio, value=2,
17. command=selection)
18. R2.pack( anchor = W )
20. R3 = Radiobutton(top, text="Java", variable=radio, value=3,
21. command=selection)
22. R3.pack( anchor = W)
24. label = Label(top)
25. label.pack()
26. top.mainloop()

**Output:**



# **Python Tkinter Menubutton**

The Menubutton widget can be defined as the drop-down menu that is shown to the user all the time. It is used to provide the user a option to select the appropriate choice exist within the application.

The Menubutton is used to implement various types of menus in the python application. A Menu is associated with the Menubutton that can display the choices of the Menubutton when clicked by the user.

The syntax to use the python tkinter Menubutton is given below.

### Syntax

1. w = Menubutton(Top, options)

A list of various options is given below.

| **SN** | **Option** | **Description** |
| --- | --- | --- |
| 1 | activebackground | The background color of the widget when the widget is under focus. |
| 2 | activeforeground | The font color of the widget text when the widget is under focus. |
| 3 | anchor | It specifies the exact position of the widget content when the widget is assigned more space than needed. |
| 4 | Bg | It specifies the background color of the widget. |
| 5 | bitmap | It is set to the graphical content which is to be displayed to the widget. |
| 6 | Bd | It represents the size of the border. The default value is 2 pixels. |
| 7 | Cursor | The mouse pointer will be changed to the cursor type specified when the widget is under the focus. The possible value of the cursor type is arrow, or dot etc. |
| 8 | direction | It direction can be specified so that menu can be displayed to the specified direction of the button. Use LEFT, RIGHT, or ABOVE to place the widget accordingly. |
| 9 | disabledforeground | The text color of the widget when the widget is disabled. |
| 10 | Fg | The normal foreground color of the widget. |
| 11 | Height | The vertical dimension of the Menubutton. It is specified as the number of lines. |
| 12 | highlightcolor | The highlight color shown to the widget under focus. |
| 13 | Image | The image displayed on the widget. |
| 14 | Justify | This specified the exact position of the text under the widget when the text is unable to fill the width of the widget. We can use the LEFT for the left justification, RIGHT for the right justification, CENTER for the centre justification. |
| 15 | Menu | It represents the menu specified with the Menubutton. |
| 16 | Padx | The horizontal padding of the widget. |
| 17 | Pady | The vertical padding of the widget. |
| 18 | Relief | This option specifies the type of the border. The default value is RAISED. |
| 19 | State | The normal state of the Mousebutton is enabled. We can set it to DISABLED to make it unresponsive. |
| 20 | Text | The text shown with the widget. |
| 21 | textvariable | We can set the control variable of string type to the text variable so that we can control the text of the widget at runtime. |
| 22 | underline | The text of the widget is not underlined by default but we can set this option to make the text of the widget underlined. |
| 23 | Width | It represents the width of the widget in characters. The default value is 20. |
| 24 | wraplength | We can break the text of the widget in the number of lines so that the text contains the number of lines not greater than the specified value. |

### Example

1. # !/usr/bin/python3
3. **from** tkinter **import** \*
5. top = Tk()
7. top.geometry("200x250")
9. menubutton = Menubutton(top, text = "Language", relief = FLAT)
11. menubutton.grid()
13. menubutton.menu = Menu(menubutton)
15. menubutton["menu"]=menubutton.menu
17. menubutton.menu.add\_checkbutton(label = "Hindi", variable=IntVar())
19. menubutton.menu.add\_checkbutton(label = "English", variable = IntVar())
21. menubutton.pack()
23. top.mainloop()

**Output:**

