Python Variables

Variable is a name that is used to **refer to memory location**. Python variable is also known as an **identifier** and used to hold value.

In Python, we don't need to specify the **type of variable** because Python is a infer language and **smart enough** to get variable type.

Variable names can be a group of both the **letters and digits**, but they have to begin with a **letter or an underscore**.

It is recommended to use lowercase letters for the variable name. **R**ahul and **r**ahul both are **two different variables**.

Identifier Naming

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

- The first character of the variable must be an alphabet or underscore (_).
- All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
- Identifier name must not contain any **white-space, or special character** (!, @, #, %, ^, &, *).
- Identifier name must not be similar to any keyword defined in the language.
- Identifier names are case sensitive; for example, my name, and MyName is not the same.
- Examples of valid identifiers: a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

Declaring Variable and Assigning Values

Python does not bind us **to declare a variable before using it** in the application. It allows us to **create a variable at the required time**.

We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

Object References

It is necessary to understand how the Python interpreter works when we declare a variable. The process of treating variables is somewhat different from many other programming languages.

Python is the highly object-oriented programming language; that's why every data item belongs to a specific type of class. Consider the following example.

```
1. print("John")
```

Output:

```
John
```

The Python object creates an integer object and displays it to the console. In the above print statement, we have created a string object. Let's check the type of it using the Python built-in **type()** function.

```
1. type("John")
```

Output:

```
<class 'str'>
```

In Python, variables are a symbolic name that is a reference or pointer to an object. The variables are used to denote objects by that name.

Let's understand the following example

```
1. a = 50
```

In the above image, the variable **a** refers to an integer object.

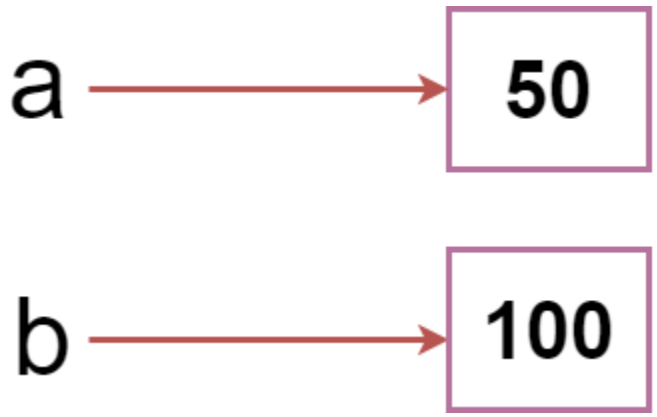
Suppose we assign the integer value 50 to a new variable b.

```
a = 50

b = a
```

The variable b refers to the same object that a points to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

```
a = 50

b =100
```



Python manages memory efficiently if we assign the same variable to two different values.

Object Identity

In Python, every created object identifies uniquely in Python. Python provides the guaranteed that no two objects will have the same identifier. The built-in **id()** function, is used to identify the object identifier. Consider the following example.

```
1. a = 50
2. b = a
3. print(id(a))
4. print(id(b))
5. # Reassigned variable a
6. a = 500
7. print(id(a))
```

Output:

```
140734982691168
140734982691168
2822056960944
```

We assigned the **b = a, a** and **b** both point to the same object. When we checked by the **id()** function it returned the same number. We reassign **a** to 500; then it referred to the new object identifier.

Variable Names

We have already discussed how to declare the valid variable. Variable names can be any length can have uppercase, lowercase (A to Z, a to z), the digit (0-9), and underscore character(_). Consider the following example of valid variables names.

```
name = "Devansh"
age = 20
marks = 80.50

print(name)
print(age)
print(marks)
```

Output:

```
Devansh
20
80.5
```

Consider the following valid variables name.

```
name = "A"
Name = "B"
naMe = "C"
NAME = "D"
n_a_m_e = "E"
_name = "F"
name_ = "G"
_name_ = "H"
na56me = "I"

print(name,Name,naMe,NAME,n_a_m_e, NAME, n_a_m_e, _name, name_,_name, na56me)
```

Output:

```
A B C D E D E F G F I
```

In the above example, we have declared a few valid variable names such as name, _name_ , etc. But it is not recommended because when we try to read code, it may create confusion. The variable name should be descriptive to make code more readable.

The multi-word keywords can be created by the following method.

- **Camel Case -** In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.
- **Pascal Case -** It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.
- **Snake Case -** In the snake case, Words are separated by the underscore. For example - name_of_student, etc.

Multiple Assignment

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Consider the following example.

1. Assigning single value to multiple variables

Eg:

```
x=y=z=50
print(x)
print(y)
print(z)
```

Output:

```
50
50
50
```

2. Assigning multiple values to multiple variables:

Eg:

```
a,b,c=5,10,15
print(a)
print(b)
print(c)
```

Output:

```
5
10
15
```

The values will be assigned in the order in which variables appear.

Python Variable Types

There are two types of variables in Python - Local variable and Global variable. Let's understand the following variables.

Local Variable

Local variables are the variables that declared inside the function and have scope within the function. Let's understand the following example.

Example -

```
# Declaring a function
def add():
    # Defining local variables. They has scope only within a function
    a = 20
    b = 30
    c = a + b
    print("The sum is:", c)

# Calling a function
add()
```

Output:

```
The sum is: 50
```

Explanation:

In the above code, we declared a function named **add()** and assigned a few variables within the function. These variables will be referred to as the **local variables** which have scope only inside the function. If we try to use them outside the function, we get a following error.

```
1. add()
2. # Accessing local variable outside the function
3. print(a)
```

Output:

```
The sum is: 50
    print(a)
NameError: name 'a' is not defined
```

We tried to use local variable outside their scope; it threw the **NameError.**

Global Variables

Global variables can be used throughout the program, and its scope is in the entire program. We can use global variables inside or outside the function.

A variable declared outside the function is the global variable by default. Python provides the **global** keyword to use global variable inside the function. If we don't use the **global** keyword, the function treats it as a local variable. Let's understand the following example.

Example -

```
# Declare a variable and initialize it
x = 101

# Global variable in function
def mainFunction():
    # printing a global variable
    global x
    print(x)
    # modifying a global variable
    x = 'Welcome To UVPCE'
    print(x)

mainFunction()
print(x)
```

Output:

```
101
Welcome To Javatpoint
Welcome To Javatpoint
```

Explanation:

In the above code, we declare a global variable **x** and assign a value to it. Next, we defined a function and accessed the declared variable using the **global** keyword inside the function. Now we can modify its value. Then, we assigned a new string value to the variable x.

Now, we called the function and proceeded to print **x**. It printed the as newly assigned value of x.

Delete a variable

We can delete the variable using the **del** keyword. The syntax is given below.

Syntax -

1. **del** <variable_name>

In the following example, we create a variable x and assign value to it. We deleted variable x, and print it, we get the error **"variable x is not defined"**. The variable x will no longer use in future.

Example -

```
# Assigning a value to x
x = 6
print(x)
# deleting a variable.
del x
print(x)
```

Output:

```
6
Traceback (most recent call last):
  File   "C:/Users/DEVANSH   SHARMA/PycharmProjects/Hello/multiprocessing.py",
line 389, in
    print(x)
NameError: name 'x' is not defined
```

Maximum Possible Value of an Integer in Python

Unlike the other programming languages, Python doesn't have long int or float data types. It treats all integer values as an **int** data type. Here, the question arises. What is

the maximum possible value can hold by the variable in Python? Consider the following example.

Example -

```
# A Python program to display that we can store
# large numbers in Python

a = 10000000000000000000000000000000000000000000000
a = a + 1
print(type(a))
print (a)
```

Output:

```
<class 'int'>
100000000000000000000000000000000000000000000001
```

As we can see in the above example, we assigned a large integer value to variable **x** and checked its type. It printed **class <int>** not long int. Hence, there is no limitation number by bits and we can expand to the limit of our memory.

Python doesn't have any special data type to store larger numbers.

Print Single and Multiple Variables in Python

We can print multiple variables within the single print statement. Below are the example of single and multiple printing values.

Example - 1 (Printing Single Variable)

```
# printing single value
a = 5
print(a)
print((a))
```

Output:

```
5
5
```

Example - 2 (Printing Multiple Variables)

```
a = 5
b = 6
# printing multiple variables
print(a,b)
# separate the variables by the comma
Print(1, 2, 3, 4, 5, 6, 7, 8)
```

Output:

```
5 6
1 2 3 4 5 6 7 8
```

Basic Fundamentals:

This section contains the fundamentals of Python, such as:

i)Tokens and their types.

ii) Comments

a)Tokens:

- The tokens can be defined as a punctuator mark, reserved words, and each word in a statement.
- The token is the smallest unit inside the given program.

There are following tokens in Python:

- Keywords.
- Identifiers.
- Literals.
- Operators.

We will discuss above the tokens in detail next tutorials.

Operators are the constructs which can manipulate the value of operands.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look on all operators one by one.

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the

		power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) −	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.

<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then −

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a

/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in the binary format their values will be 0011 1100 and 0000 1101 respectively. Following table lists out the bitwise operators supported by Python language with an example each in those, we use the above two variables (a and b) as operands −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Identity operators compare the memory locations of two objects. There are two Identity operators explained below −

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Sr.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Complement, unary plus and minus (method names for the last two are +@ and -@)
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -**

	Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'
7	**^ |** Bitwise exclusive `OR' and regular `OR'
8	**<= < > >=** Comparison operators
9	**<> == !=** Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is is not** Identity operators
12	**in not in** Membership operators
13	**not or and** Logical operators

Python Data Types

Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	`str`
Numeric Types:	`int`, `float`, `complex`
	`list`, `tuple`, `range`
Mapping Type:	`dict`
Set Types:	`set`, `frozenset`
Boolean Type:	`bool`
Binary Types:	`bytes`, `bytearray`, `memoryview`

Getting the Data Type

You can get the data type of any object by using the `type()` function:

Example

Print the data type of the variable x:

```
x = 5
print(type(x))
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
`x = "Hello World"`	str
`x = 20`	int
`x = 20.5`	float
`x = 1j`	complex
`x = ["apple", "banana", "cherry"]`	list
`x = ("apple", "banana", "cherry")`	tuple

`x = range(6)`	range
`x = {"name" : "John", "age" : 36}`	dict
`x = {"apple", "banana", "cherry"}`	set
`x = frozenset({"apple", "banana", "cherry"})`	frozenset
`x = True`	bool
`x = b"Hello"`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview

Setting the Specific Data Type

If you want to specify the data type, you can use the following constructor functions:

Example	Data Type
`x = str("Hello World")`	str
`x = int(20)`	int
`x = float(20.5)`	float
`x = complex(1j)`	complex
`x = list(("apple", "banana", "cherry"))`	list
`x = tuple(("apple", "banana", "cherry"))`	tuple
`x = range(6)`	range
`x = dict(name="John", age=36)`	dict

`x = set(("apple", "banana", "cherry"))`	set
`x = frozenset(("apple", "banana", "cherry"))`	frozenset
`x = bool(5)`	bool
`x = bytes(5)`	bytes
`x = bytearray(5)`	bytearray
`x = memoryview(bytes(5))`	memoryview

Python Indentation

Indentation refers to the spaces at the beginning of a code line.

Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.

Python uses indentation to indicate a block of code.

Example

```
if 5 > 2:
  print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

Example

Syntax Error:

```
if 5 > 2:
print("Five is greater than two!")
```

The number of spaces is up to you as a programmer, but it has to be at least one.

Example

```
if 5 > 2:
 print("Five is greater than two!")
if 5 > 2:
        print("Five is greater than two!")
```

You have to use the same number of spaces in the same block of code, otherwise Python will give you an error:

Example

Syntax Error:

```
if 5 > 2:
 print("Five is greater than two!")
        print("Five is greater than two!")
```

Comments

Python has commenting capability for the purpose of in-code documentation.

Comments start with a #, and Python will render the rest of the line as a comment:

Example

Comments in Python:

```
#This is a comment.
print("Hello, World!")
```

Multi Line Comments

Python does not really have a syntax for multi line comments.

To add a multiline comment you could insert a # for each line:

Example

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

r, not quite as intended, you can use a multiline string.

Since Python will ignore string literals that are not assigned to a variable, you can add a multiline string (triple quotes) in your code, and place your comment inside it:

Example

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with

except	lambda	yield

Lines and Indentation

Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example −

```
if True:
   print "True"
else:
   print "False"
```

However, the following block generates an error −

```
if True:
print "Answer"
print "True"
else:
print "Answer"
print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would form a block. The following example has various statement blocks −

Note − Do not try to understand the logic at this point of time. Just make sure you understood various blocks even if they are without braces.

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Or three single quotes:

Example

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Everything in Python is an object including classes. All classes are instances of a class called "type".The type object is also an instance of type class. You can inspect the inheritance hierarchy of class by examining the __bases__ attribute of a class object. type() method returns class type of the argument(object) passed as parameter.If single argument type(obj) is passed to type method, it returns the type of given object. If three arguments type(name, bases, dict) is passed, it returns a new type object.

Using type()

Let's look at the classes for the most used data types. In the below program we initialize some variables and then use the type() to ascertain their class.

Example

```
# Some variables
a = 5
b = 5.2
c = 'hello'
A = [1,4,7]
B = {'k1':'Sun','K2':"Mon",'K3':'Tue'}
C = ('Sky','Blue','Vast')
print(type(a))
print(type(b))
print(type(c))

print(type(A))
print(type(B))
print(type(C))
```

Output

Running the above code gives us the following result −

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'list'>
<class 'dict'>
<class 'tuple'>
```

Type of the classes

If we go deeper to look at the type of classes above, we will see that they are all belong to class named 'type'.

Example

```
print(type(int))
print(type(dict))
print(type(list))

print(type(type))
```

Output

Running the above code gives us the following result −

```
<class 'type'>
<class 'type'>
<class 'type'>
<class 'type'>
```

Creating a new Object Type

We can also use a similar approach as above to create new objects. Here we pass three parameters to create the new type object.

Example

```
Object1 = type('A', (object,), dict(a='Hello', b=5))
print(type(Object1))
print(vars(Object1))
class NewCalss:
   a = 'Good day!'
   b = 7
Object2 = type('B', (NewCalss,), dict(a='Hello', b=5))
print(type(Object2))
```

```
print(vars(Object2))
```

Output

Running the above code gives us the following result −

```
<class 'type'>
{'a': 'Hello', 'b': 5, '__module__': '__main__', '__dict__':
<attribute '__dict__' of 'A' objects>, '__weakref__': <attribute
'__weakref__' of 'A' objects>, '__doc__': None}
<class 'type'>
{'a': 'Hello', 'b': 5, '__module__': '__main__', '__doc__': None}
```

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

To verify the type of any object in Python, use the `type()` function:

Example

```
print(type(x))
print(type(y))
print(type(z))
```

Int

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example

Integers:

```
x = 1
y = 35656222554887711
z = -3255522

print(type(x))
print(type(y))
print(type(z))
```

Float

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Example

Floats:

```
x = 1.10
y = 1.0
z = -35.59

print(type(x))
print(type(y))
print(type(z))
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

Example

Floats:

```
x = 35e3
y = 12E4
z = -87.7e100

print(type(x))
```

```
print(type(y))
print(type(z))
```

Complex

Complex numbers are written with a "j" as the imaginary part:

Example

Complex:

```
x = 3+5j
y = 5j
z = -5j

print(type(x))
print(type(y))
print(type(z))
```

Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

Example

Convert from one type to another:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
```

```
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

Note: You cannot convert complex numbers into another number type.

Random Number

Python does not have a `random()` function to make a random number, but Python has a built-in module called `random` that can be used to make random numbers:

Example

Import the random module, and display a random number between 1 and 9:

```
import random

print(random.randrange(1, 10))
```

Python has a set of built-in methods that you can use on strings.

Note: All string methods returns new values. They do not change the original string.

Method	Description
capitalize()	Converts the first character to upper case

casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric

isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case

join()	Converts the elements of an iterable into a string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string

rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case

zfill()	Fills the string with a specified number of 0 values at the beginning

Note: All string methods returns new values. They do not change the original string.

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

`'hello'` is the same as `"hello"`.

You can display a string literal with the `print()` function:

Example

```
print("Hello")
print('Hello')
```

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

Example

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Or three single quotes:

Example

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

Example

Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

Looping Through a String

Since strings are arrays, we can loop through the characters in a string, with a `for` loop.

Example

Loop through the letters in the word "banana":

```
for x in "banana":
  print(x)
```

String Length

To get the length of a string, use the `len()` function.

Example

The `len()` function returns the length of a string:

```
a = "Hello, World!"
print(len(a))
```

Check String

To check if a certain phrase or character is present in a string, we can use the keyword `in`.

Example

Check if "free" is present in the following text:

```
txt = "The best things in life are free!"
print("free" in txt)
```

Use it in an `if` statement:

Example

Print only if "free" is present:

```
txt = "The best things in life are free!"
if "free" in txt:
  print("Yes, 'free' is present.")
```

Check if NOT

To check if a certain phrase or character is NOT present in a string, we can use the keyword `not in`.

Example

Check if "expensive" is NOT present in the following text:

```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

Use it in an `if` statement:

Example

print only if "expensive" is NOT present:

```
txt = "The best things in life are free!"
if "expensive" not in txt:
  print("No, 'expensive' is NOT present.")
```

Python RegEx

A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.

RegEx can be used to check if a string contains the specified search pattern.

RegEx Module

Python has a built-in package called `re`, which can be used to work with Regular Expressions.

Import the `re` module:

```
import re
```

RegEx in Python

When you have imported the `re` module, you can start using regular expressions:

Example

Search the string to see if it starts with "The" and ends with "Spain":

```
import re

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
```

RegEx Functions

The `re` module offers a set of functions that allows us to search a string for a match:

Function	Description
findall	Returns a list containing all matches

search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Metacharacters

Metacharacters are characters with a special meaning:

Character	Description	Example
[]	A set of characters	"[a-m]"
\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"

$	Ends with	"planet$"
*	Zero or more occurrences	"he.*o"
+	One or more occurrences	"he.+o"
?	Zero or one occurrences	"he.?o"
{}	Exactly the specified number of occurrences	"he.{2}o"
|	Either or	"falls|stays"
()	Capture and group	

Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"
\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"

\S	Returns a match where the string DOES NOT contain a white space character	"\S"
\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Sets

A set is a set of characters inside a pair of square brackets `[]` with a special meaning:

Set	**Description**
[arn]	Returns a match where one of the specified characters (`a`, `r`, or `n`) are present
[a-n]	Returns a match for any lower case character, alphabetically

	between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9
[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

The findall() Function

The `findall()` function returns a list containing all matches.

Example

Print a list of all matches:

```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

The list contains the matches in the order they are found.

If no matches are found, an empty list is returned:

Example

Return an empty list if no match was found:

```
import re

txt = "The rain in Spain"
x = re.findall("Portugal", txt)
print(x)
```

The search() Function

The `search()` function searches the string for a match, and returns a Match object if there is a match.

If there is more than one match, only the first occurrence of the match will be returned:

Example

Search for the first white-space character in the string:

```
import re

txt = "The rain in Spain"
x = re.search("\s", txt)
```

```
print("The first white-space character is located in position:",
x.start())
```

If no matches are found, the value `None` is returned:

Example

Make a search that returns no match:

```
import re

txt = "The rain in Spain"
x = re.search("Portugal", txt)
print(x)
```

The split() Function

The `split()` function returns a list where the string has been split at each match:

Example

Split at each white-space character:

```
import re

txt = "The rain in Spain"
x = re.split("\s", txt)
print(x)
```

You can control the number of occurrences by specifying the `maxsplit` parameter:

Example

Split the string only at the first occurrence:

```
import re

txt = "The rain in Spain"
```

```
x = re.split("\s", txt, 1)
print(x)
```

The sub() Function

The `sub()` function replaces the matches with the text of your choice:

Example

Replace every white-space character with the number 9:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt)
print(x)
```

You can control the number of replacements by specifying the `count` parameter:

Example

Replace the first 2 occurrences:

```
import re

txt = "The rain in Spain"
x = re.sub("\s", "9", txt, 2)
print(x)
```

Match Object

A Match Object is an object containing information about the search and the result.

Note: If there is no match, the value `None` will be returned, instead of the Match Object.

Example

Do a search that will return a Match Object:

```
import re

txt = "The rain in Spain"
x = re.search("ai", txt)
print(x) #this will print an object
```

The Match object has properties and methods used to retrieve information about the search, and the result:

`.span()` returns a tuple containing the start-, and end positions of the match.
`.string` returns the string passed into the function
`.group()` returns the part of the string where there was a match

Example

Print the position (start- and end-position) of the first match occurrence.

The regular expression looks for any words that starts with an upper case "S":

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.span())
```

Example

Print the string passed into the function:

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.string)
```

Example

Print the part of the string where there was a match.

The regular expression looks for any words that starts with an upper case "S":

```
import re

txt = "The rain in Spain"
x = re.search(r"\bS\w+", txt)
print(x.group())
```

Note: If there is no match, the value `None` will be returned, instead of the Match Object.

Python Exception

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error

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

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

Common Exceptions

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

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.

4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

The problem without handling exceptions

As we have already discussed, the exception is an abnormal condition that halts the execution of the program.

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

Example

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print("a/b = %d" %c)

#other code:
print("Hi I am other part of the program")
```

Output:

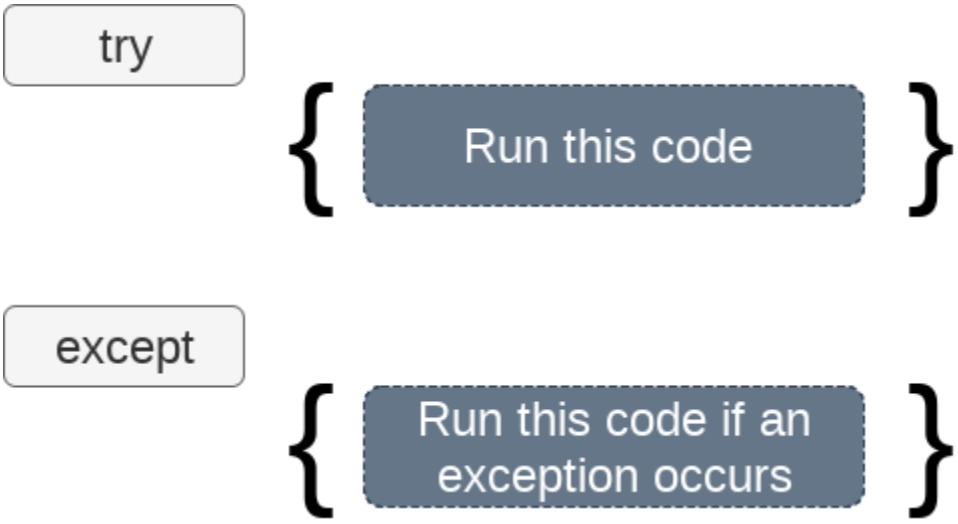
```
Enter a:10
Enter b:0
Traceback (most recent call last):
  File "exception-test.py", line 3, in <module>
    c = a/b;
ZeroDivisionError: division by zero
```

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

Exception handling in python

The try-expect statement

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



Syntax

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```

Consider the following example.

Example 1

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```

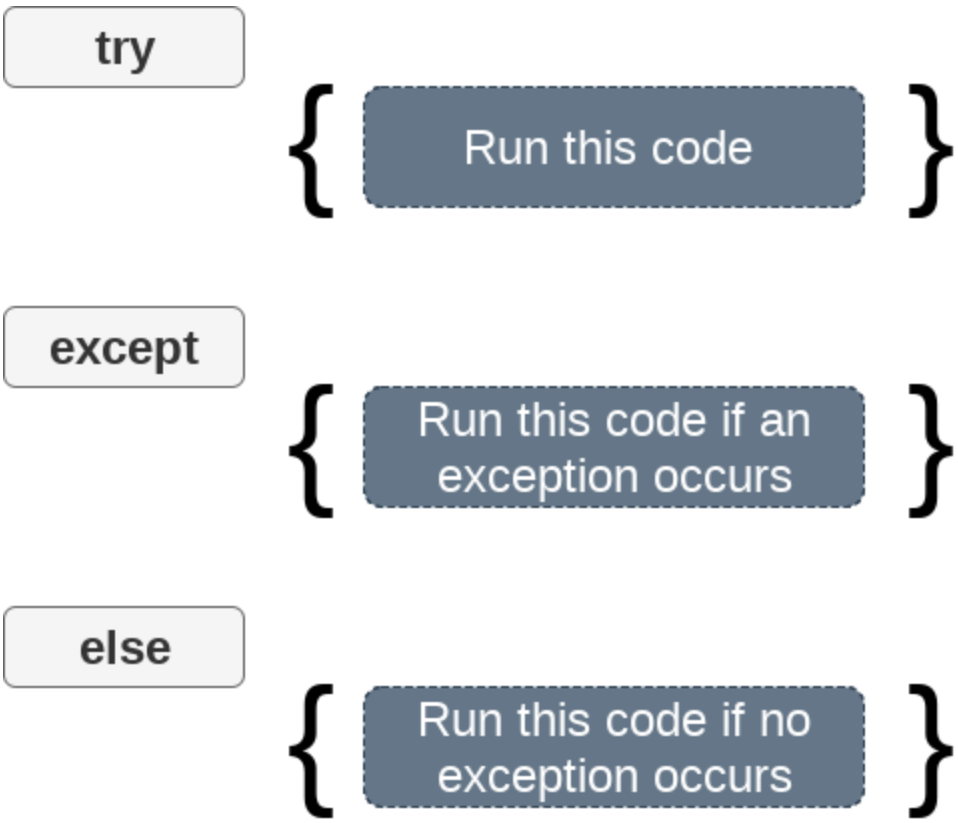
Output:

```
Enter a:10
Enter b:0
Can't divide with zero
```

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

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

```
1. try:
2.     #block of code
3.
4. except Exception1:
5.     #block of code
6.
7. else:
8.     #this code executes if no except block is executed
```



Consider the following program.

Example 2

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
# Using Exception with except statement. If we print(Exception) it will return exception class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
<class 'Exception'>
```

The except statement with no exception

Python provides the flexibility not to specify the name of exception with the exception statement.

Consider the following example.

Example

```
1. try:
2.     a = int(input("Enter a:"))
3.     b = int(input("Enter b:"))
4.     c = a/b;
5.     print("a/b = %d"%c)
6. except:
7.     print("can't divide by zero")
8. else:
9.     print("Hi I am else block")
```

The except statement using with exception variable

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
    # Using exception object with the except statement
except Exception as e:
    print("can't divide by zero")
    print(e)
```

```
else:
    print("Hi I am else block")
```

Output:

```
Enter a:10
Enter b:0
can't divide by zero
division by zero
```

Points to remember

1. Python facilitates us to not specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.

Example

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    fileptr.close()
```

Output:

```
File not found
```

Declaring Multiple Exceptions

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. The syntax is given below.

Syntax

```
try:
    #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code

else:
    #block of code
```

Consider the following example.

```
try:
   a=10/0;
except(ArithmeticError, IOError):
   print("Arithmetic Exception")
else:
   print("Successfully Done")
```

Output:

```
Arithmetic Exception
```

The try...finally block

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

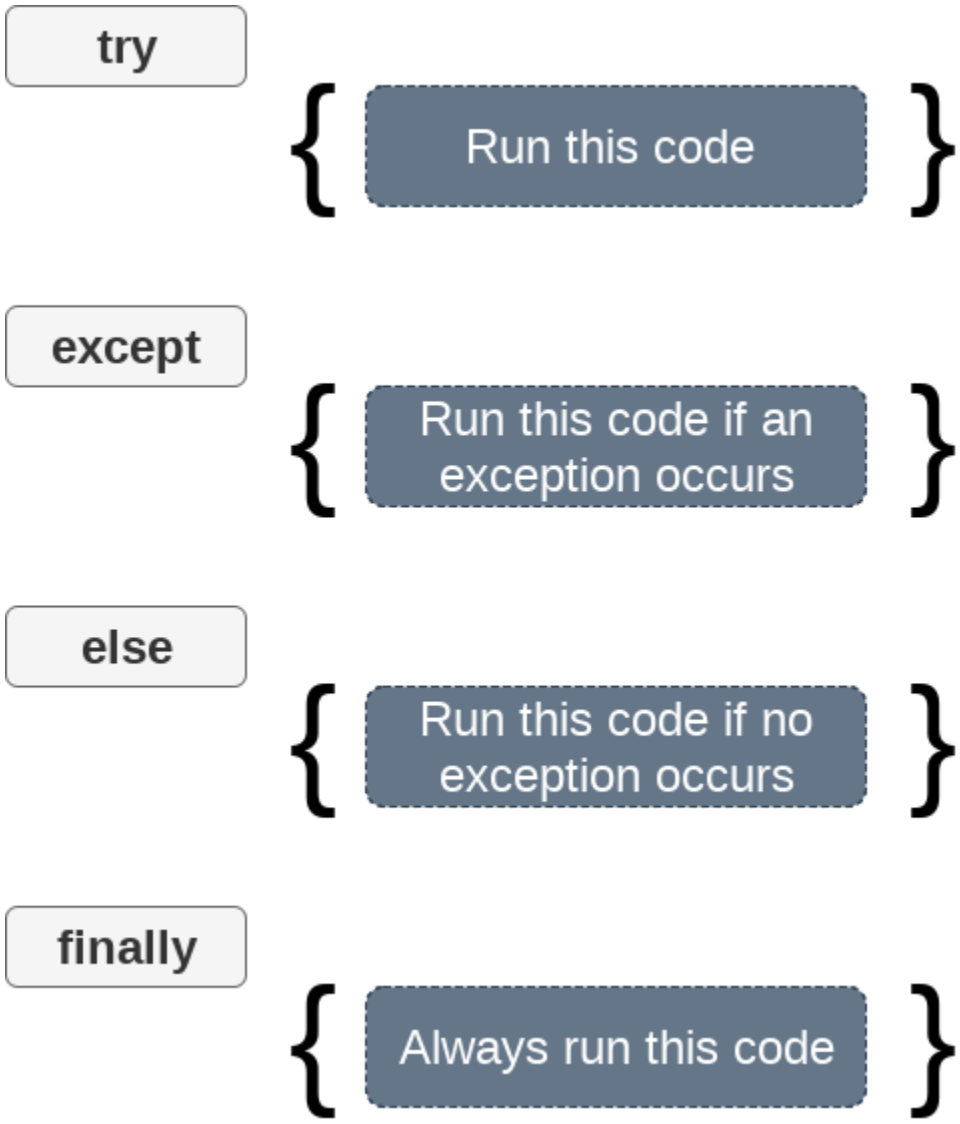
We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

Syntax

```
try:
    # block of code
```

```
    # this may throw an exception
finally:
    # block of code
    # this will always be executed
```



Example

```
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
```

```
    fileptr.close()
    print("file closed")
except:
  print("Error")
```

Output:

```
file closed
Error
```

Raising exceptions

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

For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

The syntax to use the raise statement is given below.

Syntax

1. `raise Exception_class,<value>`

Points to remember

1. To raise an exception, the raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.
3. To access the value "**as**" keyword is used. "**e**" is used as a reference variable which stores the value of the exception.
4. We can pass the value to an exception to specify the exception type.

Example

```
try:
  age = int(input("Enter the age:"))
  if(age<18):
    raise ValueError
```

```
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Output:

```
Enter the age:17
The age is not valid
```

Example 2 Raise the exception with message

```
try:
    num = int(input("Enter a positive integer: "))
    if(num <= 0):
# we can pass the message in the raise statement
        raise ValueError("That is  a negative number!")
except ValueError as e:
    print(e)
```

Output:

```
Enter a positive integer: -5
That is a negative number!
```

Example 3

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    if b is 0:
        raise ArithmeticError
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

Output:

```
Enter a:10
Enter b:0
The value of b can't be 0
```

Custom Exception

The Python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

Consider the following example.

Example

```
class ErrorInCode(Exception):
    def __init__(self, data):
        self.data = data
    def __str__(self):
        return repr(self.data)

try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```

Output:

```
Received error: 2000
```

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

File Handling

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in `open()` function.

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

Example

```
f = open("demofile.txt", "r")
print(f.read())
```

If the file is located in a different location, you will have to specify the file path, like this:

Example

Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Example

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Note: the "w" method will overwrite the entire file.

Create a New File

To create a new file in Python, use the `open()` method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Delete a File

To delete a file, you must import the OS module, and run its `os.remove()` function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

Example

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

Python seek() Method

Python seek() method is used for changing the current location of the file handle. The file handle is like a cursor, which is used for defining the location of the data which has to be read or written in the file.

Syntax:

fi.seek(offset, from_where), where fi is the file pointer

Parameters:

Offset: This is used for defining the number of positions to move forward.

from_where: This is used for defining the point of reference.

NOTE: The seek() method in Python does not return any value.

The from_where argument is used for selecting the reference point. It can accept three values:

- **0:** The 0 value is used for setting the whence argument at the beginning of the file.
- **1:** The 1 value is used for setting the whence argument at the current position of the file.
- **2:** The 2 value is used for setting the whence argument at the end of the file.

The from_where argument is set to 0 by default. Therefore, the reference point cannot be set to the current position or end position, in the text mode, except when the offset is equal to 0.

Example1: (The user has to read the text file, which contains the following text:

"This is the sentence I am Writing to show the example of the seek() method working in Python."

```
fi = open("text.txt", "r")
# the second parameter of the seek method is by default 0
fi.seek(30)
# now, we will print the current position
print(fi.tell())

print(fi.readline())
fi.close()
```

Output:

```
30
ing to show the example of the seek() method working in Python.
```

Sample Text File -

```
# let's assuming the text file has following 5 lines
# line 1: This is the sentence I am Writing to show the example of the seek()
# method working in Python.
# line 2: The line no. 2 for showing the second method of writing the seek()
# method is python
# Line 3: the line no. 3 for showing the second method
# Line 4: the line no. 4 for showing the second method
# Line 5: the line no. 5 for showing the second method
```

Example 2: (method 2)

```
fi = open ("text.txt", "r+")
print ("Name of the file: ", fi.name)

line_1 = fi.readline()
print ("Read Line: %s" % (line_1))

# Again, set the pointer to the beginning
fi.seek (0, 1)
line_2 = fi.readline()
print ("Read Line: %s" % (line_2))
```

```
# Close opend file
fi.close()
```

Output:

```
Name of the file:  text.txt
Read Line: This is the sentence I am Writing to show the example of the seek()
method working in Python.

Read Line: The line no. 2 for showing the second method of writing the seek()
method is python
```

The seek() method can also work with the negative offset, but only when the file is opened in binary mode.

Example 3: (negative offset)

```
fi = open ("text.txt", "rb")

# the user is setting the reference point to thirtieth position to the left from
# end
fi.seek (-70, 2)

# now print the current position
print (fi.tell())

# now the user will Convert the binary to string
print (fi.readline().decode('utf-8'))

fi.close()
```

Output:

```
25
 Writing to show the example of the seek() method working in Python.
```

Python File writelines() Method

```
f = open("demofile3.txt", "a")
f.writelines(["See you soon!", "Over and out."])
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

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

- **Tkinter** − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python. We would look this option in this chapter.
- **wxPython** − This is an open-source Python interface for wxWindows http://wxpython.org.
- **JPython** − JPython is a Python port for Java which gives Python scripts seamless access to Java class libraries on the local machine http://www.jython.org.

There are many other interfaces available, which you can find them on the net.

Python Tkinter

The `tkinter` package (“Tk interface”) is the standard Python interface to the Tcl/Tk GUI toolkit. Both Tk and `tkinter` are available on most Unix platforms, including macOS, as well as on Windows systems.

Tk

> Tk is a Tcl package implemented in C that adds custom commands to create and manipulate GUI widgets. Each `Tk` object embeds its own Tcl interpreter instance with Tk loaded into it. Tk’s widgets are very customizable, though at the cost of a dated appearance. Tk uses Tcl’s event queue to generate and process GUI events.

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.

Call the main event loop so that the actions can take place on the user's computer screen.

To verify that Tkinter is properly installed, enter:

```
python -m tkinter
```

How To Download Tkinter For Python 3

The best way to get the latest version of Tkinter is to install Python 3.7 or later. But Tkinter can also be downloaded and installed as part of any standard Python 3 installation.

Example

```
# !/usr/bin/python3
from tkinter import *
```

```
#creating the application main window.
top = Tk()
#Entering the event main loop
top.mainloop()
```

Output:

mainloop() tells Python to run the Tkinter event loop. This method listens for events, such as button clicks or keypresses, and blocks any code that comes after it from running until the window it's called on is closed. Go ahead and close the window you've created, and you'll see a new prompt displayed in the shell.

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	The Button is used to add various kinds of buttons to the python application.
2	Canvas	The canvas widget is used to draw the canvas on the window.
3	Checkbutton	The Checkbutton is used to display the CheckButton on the window.

4	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	It can be defined as a container to which, another widget can be added and organized.
6	Label	A label is a text used to display some message or information about the other widgets.
7	ListBox	The ListBox widget is used to display a list of options to the user.
8	Menubutton	The Menubutton is used to display the menu items to the user.
9	Menu	It is used to add menu items to the user.
10	Message	The Message widget is used to display the message-box to the user.
11	Radiobutton	The button 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	It is used to provide the slider to the user.
13	Scrollbar	It provides the scrollbar to the user so that the user can scroll the window up and down.
14	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	It is used to create a separate window container.
15	Spinbox	It is an entry widget used to select from options of values.

16	PanedWindow	It is like a container widget that contains horizontal or vertical panes.
17	LabelFrame	A LabelFrame is a container widget that acts as the container
18	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

```
# !/usr/bin/python3
from tkinter import *
parent = Tk()
redbutton = Button(parent, text = "Red", fg = "red")
redbutton.pack( side = LEFT)
greenbutton = Button(parent, text = "Black", fg = "black")
greenbutton.pack( side = RIGHT )
bluebutton = Button(parent, text = "Blue", fg = "blue")
bluebutton.pack( side = TOP )
blackbutton = Button(parent, text = "Green", fg = "red")
blackbutton.pack( side = BOTTOM)
parent.mainloop()
```

Output:

Python Tkinter 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

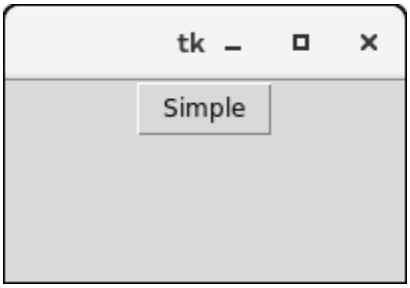
		CENTER for the center.
13	Padx	Additional padding to the button in the horizontal direction.
14	pady	Additional padding to the button in the vertical direction.
15	Relief	It represents the type of the border. It can be SUNKEN, RAISED, GROOVE, and RIDGE.
17	State	This option is set to DISABLED to make 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

```
#python application to create a simple button

from tkinter import *


top = Tk()
```

```
top.geometry("200x100")

b = Button(top,text = "Simple")

b.pack()

top.mainloop()
```

Output:

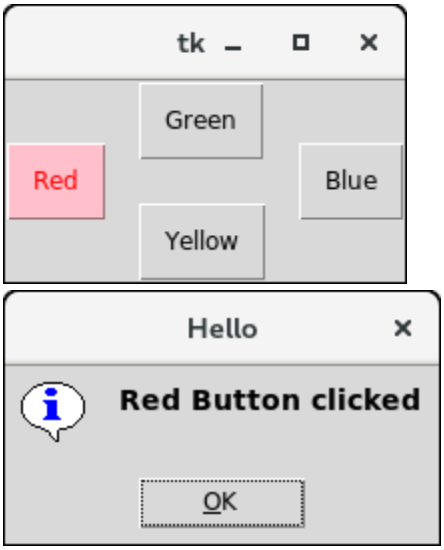


Example

```
from tkinter import *

top = Tk()

top.geometry("200x100")

def fun():
    messagebox.showinfo("Hello", "Red Button clicked")
```

```
b1 = Button(top,text = "Red",command = fun,activeforeground = "red",activebackground = "pin
k",pady=10)

b2 = Button(top, text = "Blue",activeforeground = "blue",activebackground = "pink",pady=10)

b3 = Button(top, text = "Green",activeforeground = "green",activebackground = "pink",pady = 1
0)

b4 = Button(top, text = "Yellow",activeforeground = "yellow",activebackground = "pink",pady =
10)

b1.pack(side = LEFT)

b2.pack(side = RIGHT)

b3.pack(side = TOP)

b4.pack(side = BOTTOM)

top.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.

```
from tkinter import *
gui = Tk()
Label(gui, text="Firstname").grid(row=0)
Label(gui, text="Lastname").grid(row=1)
e1 = Entry(gui)
e2 = Entry(gui)
e1.grid(row=0, column=1)
e2.grid(row=1, column=3)
gui.mainloop()
```

- **Columnspan**
  The width of the widget. It represents the number of columns up to which, the column is expanded.
- **ipadx,ipady**
  It represents the number of pixels to pad the widget inside the widget's border.

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

Example

```
from tkinter import *
root = Tk()
btn_column = Button(root, text="I'm in column 3")
btn_column.grid(column=3)

btn_columnspan = Button(root, text="I have a columnspan
of 3")
btn_columnspan.grid(columnspan=3)

btn_ipadx = Button(root, text="ipadx of 4")
btn_ipadx.grid(ipadx=4)

btn_ipady = Button(root, text="ipady of 4")
btn_ipady.grid(ipady=16)

btn_padx = Button(root, text="padx of 4")
btn_padx.grid(padx=180)

btn_pady = Button(root, text="pady of 4")
```

```
btn_pady.grid(pady=18)

btn_row = Button(root, text="I'm in row 2")
btn_row.grid(row=2)

btn_rowspan = Button(root, text="Rowspan of 2")
btn_rowspan.grid(rowspan=2)

btn_sticky = Button(root, text="I'm stuck to north-
east")
btn_sticky.grid(sticky=NE)

root.mainloop()
```

Python tkinter place (layout)

«Tkinter « Grid Layout « Pack

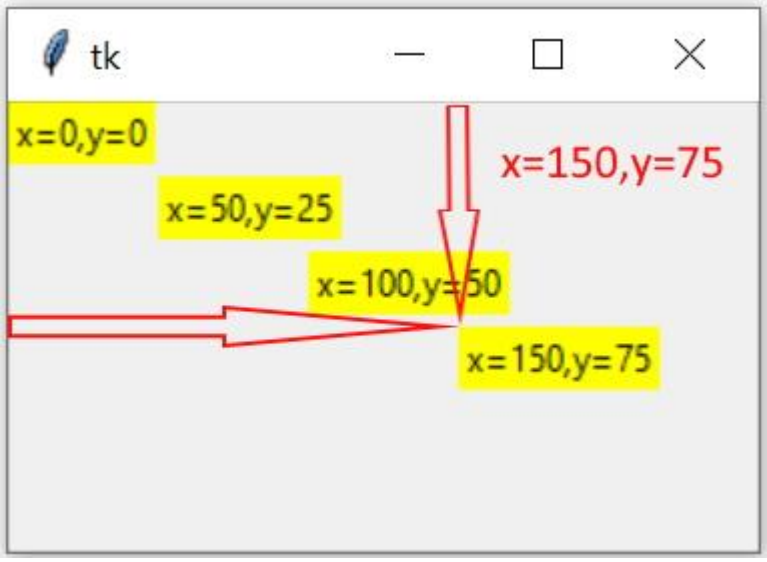
Positioning of widgets based on absolute and relative coordinates with respect to parent window.

Tkinter layout management using place for absolute & relative height width and positioning of widget.

Here is an example of layout of the page to place the components using x , y coordinates.

We will keep Labels showing x (horizontal) and y (vertical) positions

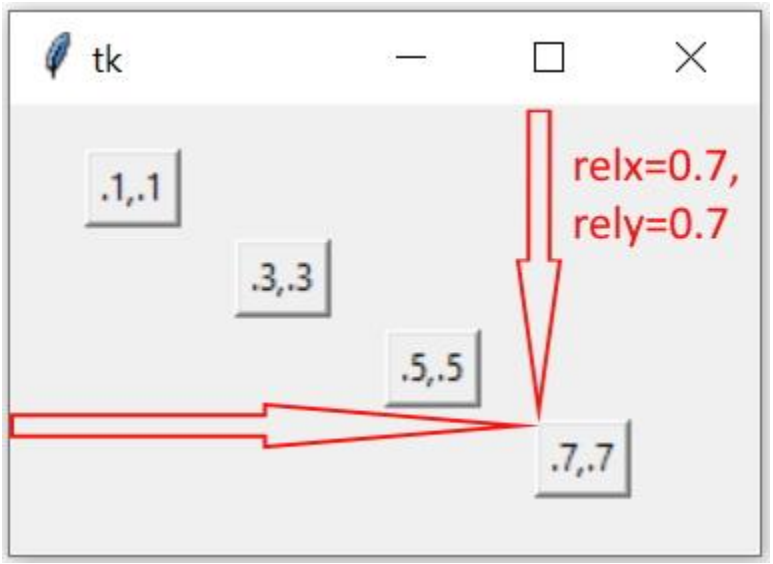
Here `x=0` is left edge and `y=0` is top edge of the window.




```
import tkinter as tk

my_w = tk.Tk()

my_w.geometry("250x150")  # Size of the window



l1=tk.Label(my_w,text='x=0,y=0',bg='yellow')

l1.place(x=0,y=0)

l2=tk.Label(my_w,text='x=50,y=25',bg='yellow')

l2.place(x=50,y=25)

l3=tk.Label(my_w,text='x=100,y=50',bg='yellow')

l3.place(x=100,y=50)
```

```
l4=tk.Label(my_w,text='x=150,y=75',bg='yellow')

l4.place(x=150,y=75)



my_w.mainloop()  # Keep the window open
```

relx & rely



We can use relative x (horizontal) and y (vertical) position with respect to width and height of the window by using *relx* and *rely* options. *relx* and *rely* can take values from 0 to 1

`relx=0` is left edge of the window `relx=1` is right edge of the window.

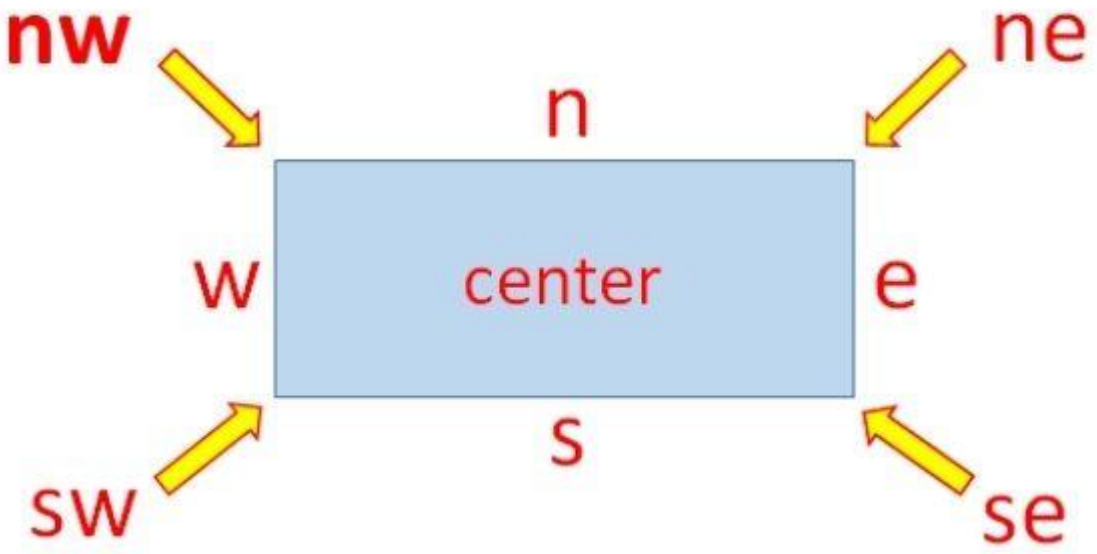
`rely=0` is top of the window and `rely=1` is the bottom edge of the window.

Here default value for `anchor='nw'` is used.

```
l1=tk.Label(my_w,text='relx=.1,rely=.1 ',bg='yellow')
```

```
l1.place(relx=.1,rely=.1)

l2=tk.Label(my_w,text='x=.3,y=.3',bg='yellow')

l2.place(relx=.3,rely=.3)

l3=tk.Label(my_w,text='x=.5,y=.5',bg='yellow')

l3.place(relx=.5,rely=.5)

l4=tk.Label(my_w,text='x=.7,y=.7',bg='yellow')

l4.place(relx=.7,rely=.7)
```

Understanding anchor



The option *anchor* says which side or the corner of the widget to be considered for positing the widget in layout. Values of *anchor* option is here

```
anchor=nw(default),ne,se,sw,n,s,
e,w,center
```

Using relx & rely with anchor at four corners

We will place four Labels at four corners of a window. We will

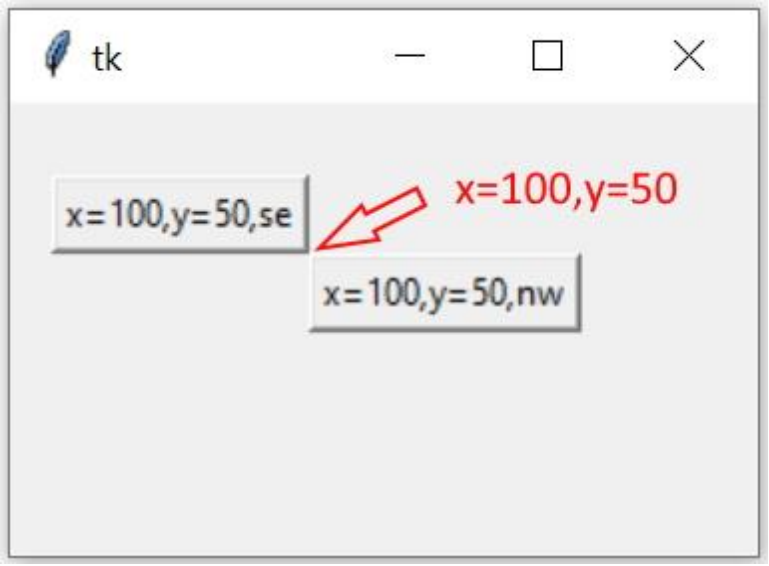
use *relx* and *rely* with values either 0 or 1 to assign relative position to widgets.

We have to change the anchor values to position the widget at corners. Note that we have not used any fraction values for *relx* or *rely*.

```
l1=tk.Label(my_w,text='x=0, y=0, nw',bg='yellow')

l1.place(relx=0,rely=0,anchor='nw') # top left

l2=tk.Label(my_w,text='x=1, y=0, ne',bg='yellow')

l2.place(relx=1,rely=0,anchor='ne') # top right



l3=tk.Label(my_w,text='x=1, y=1, se',bg='yellow')

l3.place(relx=1,rely=1,anchor='se') # bottom right



l4=tk.Label(my_w,text='x=0, y=1, sw',bg='yellow')

l4.place(relx=0,rely=1,anchor='sw') # bottom left
```

anchor and x,y coordinates

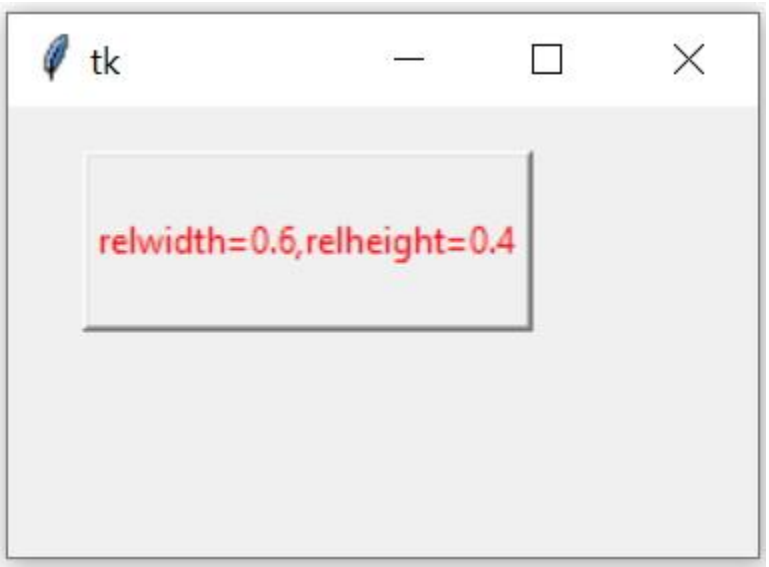
Here both the buttons has same x and y coordinate value but are placed at different positions due to their respective *anchor* values. Note that the bottom right (`anchor='se'`) corner of first widget is touching the top left (`anchor='nw'`) of second widget.




```
b1=tk.Button(my_w,text='x=100,y=50,se')

b1.place(x=100,y=50,anchor='se') # top left



b2=tk.Button(my_w,text='x=100,y=50,nw')

b2.place(x=100,y=50,anchor='nw')
```

relwidth & relheight

The dimension of the widget can be fixed based on the size of the parent window by using *relwidth* and *relheigh*



t options.

Here `relwidth=1` will occupy the full width of the parent window, similarly `relheight=1` will occupy the full height of the window.

```
import tkinter as tk

my_w = tk.Tk()

my_w.geometry("250x150")  # Size of the window

b1=tk.Button(my_w,text='relwidth=0.6,relheight=0.4',fg='red')

b1.place(relx=0.1,rely=0.1,relwidth=0.6,relheight=0.4)

my_w.mainloop()  # Keep the window open
```

Menu in Tkinter()

```
from tkinter import *

top = Tk()

def hello():
    print("hello!")

# create a toplevel menu
menubar = Menu(top)
menubar.add_command(label="Hello!", command=hello)
menubar.add_command(label="Quit!", command=top.quit)
```

```
# display the menu
top.config(menu=menubar)

top.mainloop()
```

```
    import tkinter as tk
    from tkinter import Menu

    # root window
    root = tk.Tk()
    root.title('Menu Demo')

    # create a menubar
    menubar = Menu(root)
    root.config(menu=menubar)

    # create a menu
    file_menu = Menu(menubar)

    # add a menu item to the menu
    file_menu.add_command(
        label='Exit',
        command=root.destroy
    )


    # add the File menu to the menubar
    menubar.add_cascade(
        label="File",
        menu=file_menu
```

```
    )

    root.mainloop()
```

creating a simple menu

First, create a root window and set its title to 'Menu Demo':

```
root = tk.Tk()
root.title('Menu Demo')
```
Code language: Python (python)

Second, create a menu bar and assign it to the menu option of the root window:

```
menubar = Menu(root)
root.config(menu=menubar)
```
Code language: Python (python)

Note that each top-level window can only have only one menu bar.

Third, create a **File** menu whose container is the menubar:

```
file_menu = Menu(menubar)
```
Code language: Python (python)

Fourth, add a menu item to the file_menu:

```
file_menu.add_command(
    label='Exit',
    command=root.destroy,
)
```
Code language: Python (python)

In this example, the label of the menu item is Exit.

When you click the Exit menu item, Python will call the root.destroy() method automatically to close the root window.

Finally, add the File menu to the menubar:

```
menubar.add_cascade(
    label="File",
    menu=file_menu,
```

```
    underline=0
)
```

Code language: Python (python)

The underline option allows you to create a keyboard shortcut. It specifies the character position that should be underlined.

Note that the position starts from zero. In this example, we specify it as the first character which is F. And you can select it by using the Alt+F keyboard shortcut.

To remove the dashed line, you can set the tearoff property of the menu to False:

```
file_menu = Menu(menubar, tearoff=False)
```

```
from tkinter import *

# root window
root = Tk()
root.geometry('320x150')
root.title('Menu Demo')


# create a menubar
menubar = Menu(root)
root.config(menu=menubar)

# create the file_menu
file_menu = Menu(
    menubar,
    tearoff=0
)

# add menu items to the File menu
file_menu.add_command(label='New')
file_menu.add_command(label='Open...')
```

```
file_menu.add_command(label='Close')
file_menu.add_separator()

# add Exit menu item
file_menu.add_command(
    label='Exit',
    command=root.destroy
)

# add the File menu to the menubar
menubar.add_cascade(
    label="File",
    menu=file_menu
)
# create the Help menu
help_menu = Menu(
    menubar,
    tearoff=0
)

help_menu.add_command(label='Welcome')
help_menu.add_command(label='About...')

# add the Help menu to the menubar
menubar.add_cascade(
    label="Help",
    menu=help_menu
)

root.mainloop()
```

The only new statement in this program is to use the add_separator() method to add a separator to the menu.

```
from tkinter import *

# root window
root = Tk()
root.geometry('320x150')
root.title('Menu Demo')


# create a menubar
menubar = Menu(root)
root.config(menu=menubar)

# create the file_menu
file_menu = Menu(
    menubar,
    tearoff=0
)

# add menu items to the File menu
file_menu.add_command(label='New')
file_menu.add_command(label='Open...')
file_menu.add_command(label='Close')
file_menu.add_separator()

# add a submenu
sub_menu = Menu(file_menu, tearoff=0)
sub_menu.add_command(label='Keyboard Shortcuts')
sub_menu.add_command(label='Color Themes')

# add the File menu to the menubar
```

```
file_menu.add_cascade(
    label="Preferences",
    menu=sub_menu
)

# add Exit menu item
file_menu.add_separator()
file_menu.add_command(
    label='Exit',
    command=root.destroy
)


menubar.add_cascade(
    label="File",
    menu=file_menu,
    underline=0
)
# create the Help menu
help_menu = Menu(
    menubar,
    tearoff=0
)

help_menu.add_command(label='Welcome')
help_menu.add_command(label='About...')

# add the Help menu to the menubar
menubar.add_cascade(
    label="Help",
    menu=help_menu,
    underline=0
```

```
)

root.mainloop()
```

```
from tkinter import *
from tkinter import messagebox

ws =Tk()
ws.title("Python Guides")
ws.geometry("300x250")

def about():
    messagebox.showinfo('PythonGuides', 'Python Guides
aims at providing best practical tutorials')

menubar = Menu(ws, background='#ff8000',
foreground='black', activebackground='white',
activeforeground='black')
file = Menu(menubar, tearoff=1, background='#ffcc99',
foreground='black')
file.add_command(label="New")
file.add_command(label="Open")
file.add_command(label="Save")
file.add_command(label="Save as")
file.add_separator()
file.add_command(label="Exit", command=ws.quit)
menubar.add_cascade(label="File", menu=file)

edit = Menu(menubar, tearoff=0)
edit.add_command(label="Undo")
edit.add_separator()
```

```
edit.add_command(label="Cut")
edit.add_command(label="Copy")
edit.add_command(label="Paste")
menubar.add_cascade(label="Edit", menu=edit)

help = Menu(menubar, tearoff=0)
help.add_command(label="About", command=about)
menubar.add_cascade(label="Help", menu=help)

ws.config(menu=menubar)
ws.mainloop()
```

```
from tkinter import *

from tkinter import Menu

window = Tk()

window.title("Welcome to Ganpat University ")

menu = Menu(window)

new_item = Menu(menu)

new_item.add_command(label='New')

new_item.add_separator()

new_item.add_command(label='Edit')

menu.add_cascade(label='File', menu=new_item)

window.config(menu=menu)

window.mainloop()


from tkinter import *

from tkinter import ttk
```

```
window = Tk()

window.title("Welcome to UVPCE app")

tab_control = ttk.Notebook(window)

tab1 = ttk.Frame(tab_control)

tab2 = ttk.Frame(tab_control)

tab_control.add(tab1, text='First')

tab_control.add(tab2, text='Second')

lbl1 = Label(tab1, text= 'label1')

lbl1.grid(column=0, row=0)

lbl2 = Label(tab2, text= 'label2')

lbl2.grid(column=0, row=0)

tab_control.pack(expand=1, fill='both')

window.mainloop()
```

The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets.

It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets.

Syntax

Here is the simple syntax to create this widget −

```
w = Frame ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

```
from Tkinter import *

root = Tk()
frame = Frame(root)
frame.pack()
```

```
bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )

redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)

greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )

bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )

blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)

root.mainloop()
```

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


```
import tkinter as tk
#from tkinter import ttk
from tkinter.messagebox import showerror

# root window
root = tk.Tk()
root.title('Temperature Converter')
root.geometry('300x70')
root.resizable(False, False)


def fahrenheit_to_celsius(f):
    """ Convert fahrenheit to celsius
    """
    return (f - 32) * 5/9
```

```python
# frame
frame = tk.Frame(root)


# field options
options = {'padx': 5, 'pady': 5}

# temperature label
temperature_label = tk.Label(frame, text='Fahrenheit')
temperature_label.grid(column=0, row=0, sticky='W',
**options)

# temperature entry
temperature = tk.StringVar()
temperature_entry = tk.Entry(frame,
textvariable=temperature)
temperature_entry.grid(column=1, row=0, **options)
temperature_entry.focus()

# convert button


def convert_button_clicked():
    """  Handle convert button click event
    """
    try:
        f = float(temperature.get())
        c = fahrenheit_to_celsius(f)
        result = f'{f} Fahrenheit = {c:.2f} Celsius'
        result_label.config(text=result)
```

```
    except ValueError as error:
        showerror(title='Error', message=error)


convert_button = tk.Button(frame, text='Convert')
convert_button.grid(column=2, row=0, sticky='W',
**options)
convert_button.configure(command=convert_button_clicked)

# result label
result_label = tk.Label(frame)
result_label.grid(row=1, columnspan=3, **options)

# add padding to the frame and show it
frame.grid(padx=10, pady=10)

# start the app
root.mainloop()
```

```
import tkinter as tk
from tkinter import ttk
from tkinter.messagebox import showinfo
from calendar import month_name

root = tk.Tk()

# config the root window
root.geometry('300x200')
root.resizable(False, False)
```

```
root.title('Combobox Widget')

# label
label = ttk.Label(text="Please select a month:")
label.pack(fill=tk.X, padx=5, pady=5)

# create a combobox
selected_month = tk.StringVar()
month_cb = ttk.Combobox(root,
textvariable=selected_month)

# get first 3 letters of every month name
month_cb['values'] = [month_name[m][0:3] for m in
range(1, 13)]

# prevent typing a value
month_cb['state'] = 'readonly'

# place the widget
month_cb.pack(fill=tk.X, padx=5, pady=5)


# bind the selected value changes
def month_changed(event):
    """ handle the month changed event """
    showinfo(
        title='Result',
        message=f'You selected {selected_month.get()}!'
    )

month_cb.bind('<<ComboboxSelected>>', month_changed)
```

```
root.mainloop()
```

Python tkinter Label

This code we will add one label.

```
import tkinter as tk


my_w=tk.Tk()


my_w.geometry('300x100')


my_w.title('GANPAT UNIVERSITY')




l3 = tk.Label(my_w,  text='Welcome', width=15 )


l3.grid(row=1,column=1)


my_w.mainloop()
```

Tkinter Label with font styles color & background using fg bg text & relief with borderwidth

Adding font style and font color.

We will add one tuple with font style using *font* option and add font color by using *fg* option. Change these lines.

```
my_font1=('times', 18, 'bold') # font style declaring


l3 = tk.Label(my_w,  text='Welcome', width=15,font=my_font1,fg='red' )
```

Adding background color

add bg='yellow' to add background color to the Label

```
l3 = tk.Label(my_w,  text='Welcome', width=15,font=my_font1,


              fg='red',bg='yellow' )
```

To change or manage the text of a label we can use StringVar()

```
import tkinter  as tk


my_w = tk.Tk()


my_w.geometry("500x500")




my_str = tk.StringVar()


l1 = tk.Label(my_w,  textvariable=my_str, width=10 )


l1.grid(row=1,column=2)


my_str.set("Hi Welcome")




my_w.mainloop()
```

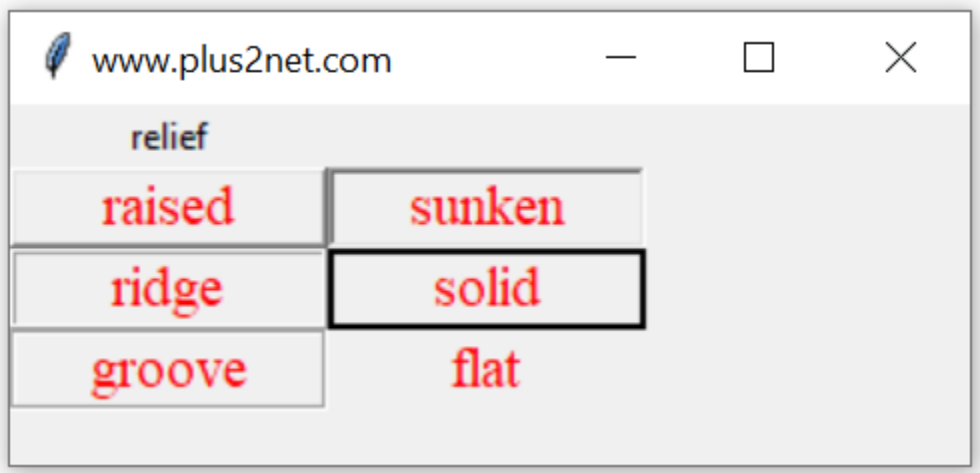
Using relief option

Option *relief* can take *raised*, *sunken*, *ridge*, *solid*,*flat*, *groove*

```
import tkinter as tk

my_w=tk.Tk()

my_w.geometry('320x120')

my_w.title('www.plus2net.com')

my_font1=('times', 14) # font style declaring

l0 = tk.Label(my_w,  text='relief', width=10)

l0.grid(row=1,column=2)



l1 = tk.Label(my_w,  text='raised', width=10,font=my_font1,

              fg='red',borderwidth=2, relief='raised' )

l1.grid(row=2,column=2)



l2 = tk.Label(my_w,  text='sunken', width=10,font=my_font1,

              fg='red',borderwidth=2, relief='sunken' )
```

```
l2.grid(row=2,column=3)

l3 = tk.Label(my_w,  text='ridge', width=10,font=my_font1,

              fg='red',borderwidth=2, relief='ridge' )

l3.grid(row=3,column=2)



l4 = tk.Label(my_w,  text='solid', width=10,font=my_font1,

              fg='red',borderwidth=2, relief='solid' )

l4.grid(row=3,column=3)



l5 = tk.Label(my_w,  text='groove', width=10,font=my_font1,

              fg='red',borderwidth=2, relief='groove' )

l5.grid(row=4,column=2)



l6 = tk.Label(my_w,  text='flat', width=10,font=my_font1,

              fg='red',borderwidth=2, relief='flat' )

l6.grid(row=4,column=3)
```

```
my_w.mainloop()
```



Positioning Labels using GRID

We will change the font size, font colour, font family and place them at different locations using GRID row and column layouts.

```
import tkinter  as tk
my_w = tk.Tk()
my_w.geometry("300x200")

my_font1=('times', 18, 'bold')
```

```
my_str1 = tk.StringVar()
l1 = tk.Label(my_w,  textvariable=my_str1,
fg='red',font=my_font1 )
l1.grid(row=1,column=2,columnspan=3,sticky='E')
my_str1.set("GANPAT UNIVERSITY")

my_font2=('Verdana', 10, 'normal')
my_str2 = tk.StringVar()
l2 = tk.Label(my_w,  textvariable=my_str2,
fg='black',font=my_font2 )
l2.grid(row=2,column=1)
my_str2.set("python tutorials")

my_font3=('MS Sans Serif', 10, 'italic')
my_str3 = tk.StringVar()
l3 = tk.Label(my_w,  textvariable=my_str3,
fg='green',font=my_font3 )
l3.grid(row=3,column=3)
my_str3.set("Tkinter")

my_str4 = tk.StringVar()
l4 = tk.Label(my_w,  textvariable=my_str4, font=my_font2
)
l4.config(fg='blue')
l4.grid(row=4,column=4)
my_str4.set("Pandas")


my_str5 = tk.StringVar()
l5 = tk.Label(my_w,  textvariable=my_str5,
fg='yellow',font=my_font2 )
l5.grid(row=5,column=5)
```

```
my_str5.set("Numpy")

my_w.mainloop()
```

There are many optional options we can add to Label, the list is available below.

Tkinter Label managing text by StringVar to update using user input by using get() & set() methods

Now let us add the click event of a button button, to change the text of the label.

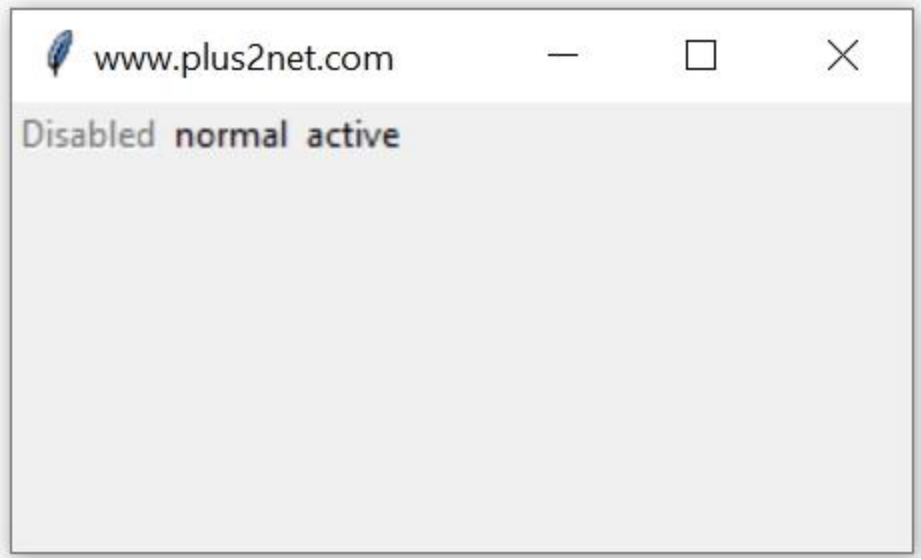
```
# Change the text of a label
import tkinter  as tk
my_w = tk.Tk()
my_w.geometry("500x500")

def my_upd():
     my_str.set("GANPAT UNIVERSITY")

my_str = tk.StringVar()
l1 = tk.Label(my_w,  textvariable=my_str, width=20 )
l1.grid(row=1,column=2)
my_str.set("Hi Welcome")

b1 = tk.Button(my_w, text='Change text',
width=15,bg='yellow',command=lambda: my_upd())
b1.grid(row=2,column=2)
```

```
my_w.mainloop()
```

Adding Image


```
import tkinter  as tk
my_w = tk.Tk()
my_w.geometry("400x200")

l1 = tk.Label(my_w,  width=15 )
l1.grid(row=1,column=1)

my_img = tk.PhotoImage(file = "E:\download.png")
l2 = tk.Label(my_w,  image=my_img )
l2.grid(row=1,column=2)


my_w.mainloop()
```

Enable disable Label



The *state* option can have three values : *disabled*, *normal* , *active*.

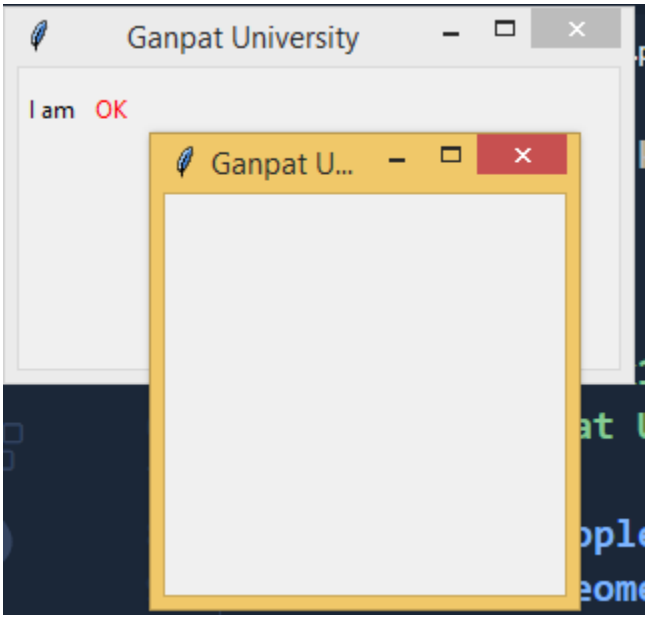
```
l1 = tk.Label(my_w, text='Disabled',state='disabled')

l1.grid(row=1,column=1)

l2 = tk.Label(my_w, text='normal',state='normal')

l2.grid(row=1,column=2)

l3 = tk.Label(my_w, text='active',state='active')

l3.grid(row=1,column=3)
```

By using *config* the state can be managed.

```
l1.config(state='normal')
```

What is the difference between normal and active ?
With active state appearance can be changed to add mouse over effects like sunken ,raised etc..

Click event of Label



Tkinter bind click event of a Label to open Toplevel child window on bind

There is no *command* option for label. We can use bind option to trigger function on click of a Lable. Here one child window is opened on click of the Label saying OK.

```
import tkinter as tk
from tkinter import *

my_w = tk.Tk()
my_w.geometry("300x150")  # Size of the window
my_w.title("Ganpat University")  # Adding a title
def my_fun(*ff):
```

```
    my_w_child=Toplevel(my_w) # Child window
    my_w_child.geometry("200x200")  # Size of the window

l1=tk.Label(my_w,text='I am')
l1.grid(row=0,column=0,padx=2,pady=10)
l2=tk.Label(my_w,text='OK',fg='red')
l2.grid(row=0,column=1,padx=2,pady=10)
l2.bind("<Button-1>",my_fun) # mouse click
my_w.mainloop()
```

A Tkinter application runs most of its time inside an event loop, which is entered via the mainloop method. It waiting for events to happen. Events can be key presses or mouse operations by the user.

Tkinter provides a mechanism to let the programmer deal with events. For each widget, it's possible to bind Python functions and methods to an event.

widget.bind(event, handler)

If the defined event occurs in the widget, the "handler" function is called with an event object. describing the event.

```
#!/usr/bin/python3
# write tkinter as Tkinter to be Python 2.x compatible
from tkinter import *
def hello(event):
    print("Single Click, Button-l")
def quit(event):
    print("Double Click, so let's stop")
    import sys; sys.exit()

widget = Button(None, text='Mouse Clicks')
widget.pack()
widget.bind('<Button-1>', hello)
widget.bind('<Double-1>', quit)
widget.mainloop()
```

Events

Tkinter uses so-called event sequences for allowing the user to define which events, both specific and general, he or she wants to bind to handlers. It is the first argument "event" of the bind method. The event sequence is given as a string, using the following syntax:

```
<modifier-type-detail>
```

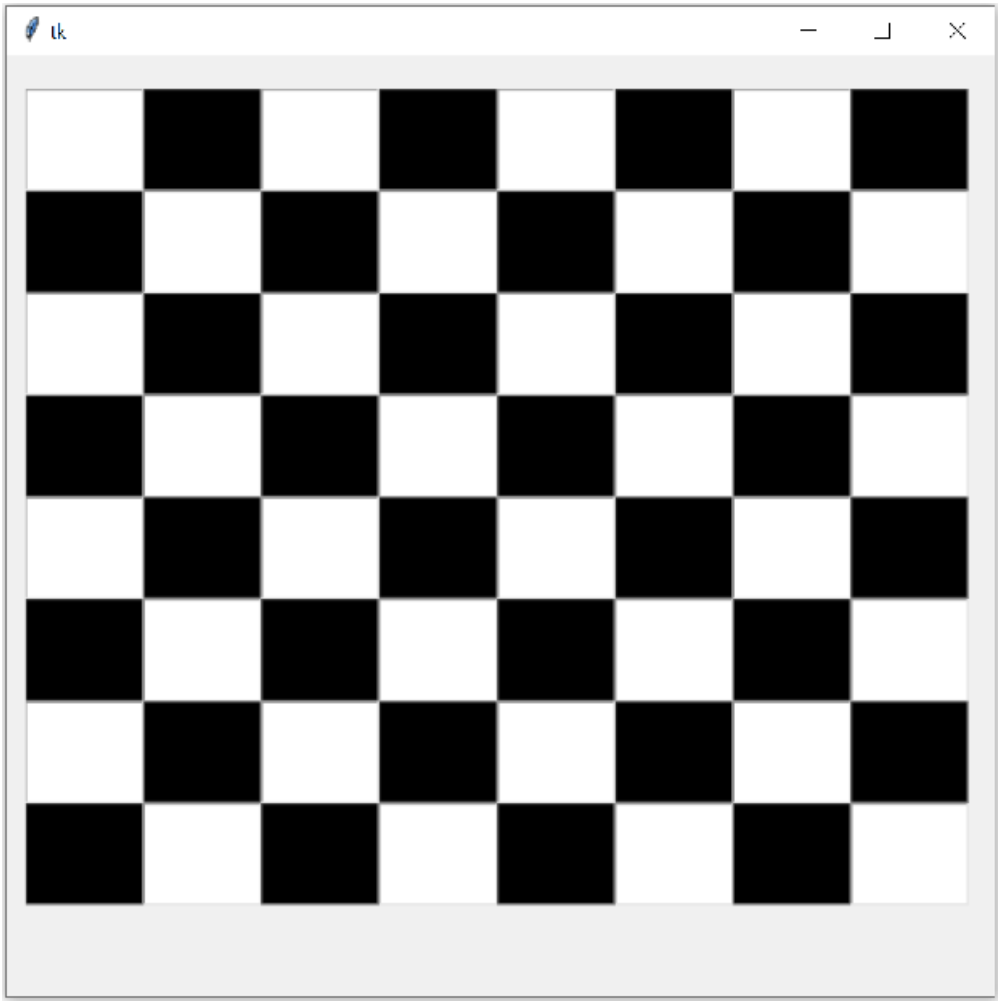
The type field is the essential part of an event specifier, whereas the "modifier" and "detail" fields are not obligatory and are left out in many cases. They are used to provide additional information for the chosen "type". The event "type" describes the kind of event to be bound, e.g. actions like mouse clicks, key presses or the widget got the input focus.

Event	Description
<Button>	A mouse button is pressed with the mouse pointer over the widget. The detail part specifies which button, e.g. The left mouse button is defined by the event <Button-1>, the middle button by <Button-2>, and the rightmost mouse button by <Button-3>. <Button-4> defines the scroll up event on mice with wheel support and and <Button-5> the scroll down. If you press down a mouse button over a widget and keep it pressed, Tkinter will automatically "grab" the mouse pointer. Further mouse events like Motion and Release events will be sent to the current widget, even if the mouse is moved outside the current widget. The current position, relative to the widget, of the mouse pointer is provided in the x and y members of the event object passed to the callback. You can use ButtonPress instead of Button, or even leave it out completely: , , and <1> are all synonyms.
<Motion>	The mouse is moved with a mouse button being held down. To specify the left, middle or right mouse button use <B1-Motion>, <B2-Motion> and <B3-Motion> respectively. The current position of the mouse pointer is provided in the x

	and y members of the event object passed to the callback, i.e. event.x, event.y
<ButtonRelease>	Event, if a button is released. To specify the left, middle or right mouse button use <ButtonRelease-1>, <ButtonRelease-2>, and <ButtonRelease-3> respectively. The current position of the mouse pointer is provided in the x and y members of the event object passed to the callback, i.e. event.x, event.y
<Double-Button>	Similar to the Button event, see above, but the button is double clicked instead of a single click. To specify the left, middle or right mouse button use <Double-Button-1>, <Double-Button-2>, and <Double-Button-3> respectively. You can use Double or Triple as prefixes. Note that if you bind to both a single click (<Button-1>) and a double click (<Double-Button-1>), both bindings will be called.
<Enter>	The mouse pointer entered the widget. Attention: This doesn't mean that the user pressed the Enter key!. <Return> is used for this purpose.
<Leave>	The mouse pointer left the widget.
<FocusIn>	Keyboard focus was moved to this widget, or to a child of this widget.
<FocusOut>	Keyboard focus was moved from this widget to another widget.
<Return>	The user pressed the Enter key. You can bind to virtually all keys on the keyboard: The special keys are Cancel (the Break key), BackSpace, Tab, Return(the Enter key), Shift_L (any Shift key), Control_L (any Control key), Alt_L (any Alt key), Pause, Caps_Lock, Escape, Prior (Page Up),

	Next (Page Down), End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num_Lock, and Scroll_Lock.
<Key>	The user pressed any key. The key is provided in the char member of the event object passed to the callback (this is an empty string for special keys).
a	The user typed an "a" key. Most printable characters can be used as is. The exceptions are space (<space>) and less than (<less>). Note that 1 is a keyboard binding, while <1> is a button binding.
<Shift-Up>	The user pressed the Up arrow, while holding the Shift key pressed. You can use prefixes like Alt, Shift, and Control.
<Configure>	The size of the widget changed. The new size is provided in the width and height attributes of the event object passed to the callback. On some platforms, it can mean that the location changed.

Creating Chess board using Label




```
import tkinter  as tk

my_w = tk.Tk()

my_w.geometry("620x590")

bg='#ffffff'

l0=tk.Label(my_w,text='  ')
```

```
l0.grid(row=0,column=0)

height=4

width=10 # change this to match your display

for i in range (1,9):

    if bg=='#ffffff':

        bg='#000000'

    else:

        bg='#ffffff'

    for j in range(1,9):

        if bg=='#ffffff':

            bg='#000000'

        else:

            bg='#ffffff'

        l1=tk.Label(my_w,bg=bg,height=height,width=width,

                borderwidth=1,relief='groove')

        l1.grid(row=i,column=j)
```

```
my_w.mainloop()
```

By preseing the + button we can increase the button width and by pressing the - we can decrease the button width.

```
import tkinter as tk
my_w=tk.Tk()
my_w.geometry('500x500')

def my_fun(todo):
    w=b2.cget('width')
    if(todo=='increase'):
       w=w+5
    else:
       w=w-5
    b2.config(width=w)

b1=tk.Button(my_w,text='+',width=20,bg='yellow',command=
lambda: my_fun('increase'))
b1.grid(row=1,column=1)
b2=tk.Button(my_w,text='Close',width=10,bg='Green',comma
nd=my_w.destroy)
b2.grid(row=1,column=3)
b3=tk.Button(my_w,text='-
',width=20,bg='yellow',command=lambda:
my_fun('decrease'))
b3.grid(row=2,column=1)
my_w.mainloop()
```

The command specifies a callback function that will be called automatically when the button clicked.

Lamda Functions (also referred to as Anonymous Function in Python) are very useful in building Tkinter GUI applications. They allow us to send multiple data through the callback function. Lambda can be inside any function that works as an anonymous function for expressions. In Button Command, lambda is used to pass the data to a callback function.

Example

In this example, we will create an application that will have some buttons in it. The button command is defined with the lambda function to pass the specific value to a callback function.

```
#Import the library

from tkinter import *

from tkinter import ttk



#Create an instance of Tkinter frame

win= Tk()



#Set the window geometry

win.geometry("750x250")



#Display a Label

def print_text(text):

   Label(win, text=text,font=('Helvetica 13 bold')).pack()
```

```
btn1= ttk.Button(win, text="Button1" ,command= lambda:

print_text("Button 1"))

btn1.pack(pady=10)

btn2= ttk.Button(win, text="Button2" ,command= lambda:

print_text("Button 2"))

btn2.pack(pady=10)

btn3= ttk.Button(win, text="Button3" ,command= lambda:

print_text("Button 3"))

btn3.pack(pady=10)


win.mainloop()
```

Output

Running the above code will display a window that contains some buttons. Each button passing a text message as the argument to a common callback function using lambda function.

Change the colour of Close button by clicking different buttons.

```
import tkinter as tk
my_w = tk.Tk()
my_w.geometry("600x200")
```

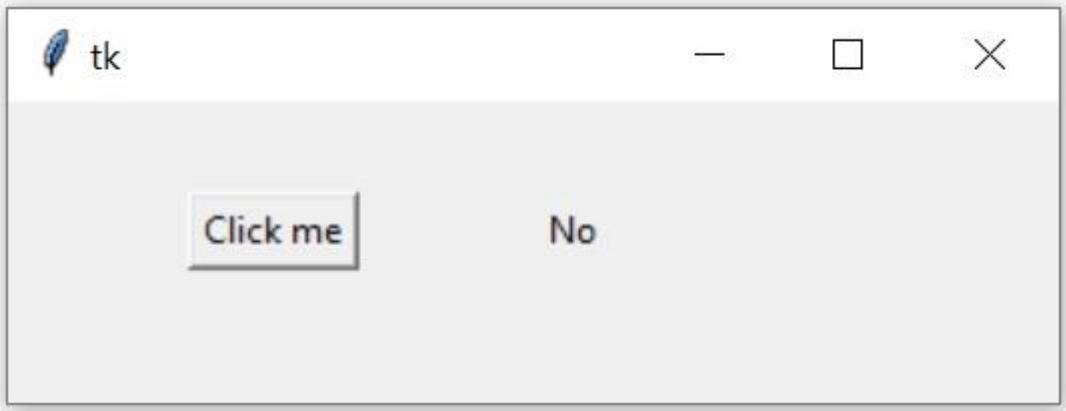
```
def my_upd(c_type):
    b_c.config(bg=c_type)

b_c=tk.Button(my_w,text='Close',width=10,command=my_w.de
stroy)
b_c.grid(row=1,column=2)

b1 = tk.Button(my_w, text='Yellow',
width=10,bg='yellow',command=lambda: my_upd('yellow'))
b1.grid(row=1,column=1)

b2 = tk.Button(my_w, text='Blue',
width=10,bg='blue',command=lambda: my_upd('blue'))
b2.grid(row=2,column=1)

b3 = tk.Button(my_w, text='Green',
width=10,bg='green',command=lambda: my_upd('green'))
b3.grid(row=3,column=1)

b4 = tk.Button(my_w, text='Red',
width=10,bg='red',command=lambda: my_upd('red'))
b4.grid(row=4,column=1)

my_w.mainloop()
```

Changing mouseover Cursor Style over a button.

```
import tkinter as tk
my_w = tk.Tk()
my_w.geometry("600x200")

def my_upd(c_type):
    b6.config(cursor=c_type)

b1 = tk.Button(my_w, text='Circle',
width=10,bg='yellow',command=lambda: my_upd('circle'))
b1.grid(row=1,column=1)

b2 = tk.Button(my_w, text='Arrow',
width=10,bg='blue',command=lambda: my_upd('arrow'))
b2.grid(row=2,column=1)

b3 = tk.Button(my_w, text='sailboat',
width=10,bg='green',command=lambda: my_upd('sailboat'))
b3.grid(row=3,column=1)

b4 = tk.Button(my_w, text='hand1',
width=10,bg='yellow',command=lambda: my_upd('hand1'))
b4.grid(row=4,column=1)

b5 = tk.Button(my_w, text='hand2',
width=10,bg='blue',command=lambda: my_upd('hand2'))
b5.grid(row=5,column=1)

b6 = tk.Button(my_w, text='Close',
width=10,bg='red',command=my_w.destroy)
b6.grid(row=1,column=2)
```

```
my_w.mainloop()
```

Every week has a number of the year (00,01,02,03 ... 53). On Click of a button, if the week number is odd then output should be *No* and it is *Yes* if week number is even.
We used int() to convert string to number.
Check the formatted output when we use strftime(). Here we used *%U* to get the week number as string.



Week number of the year (Sunday as the first day of the week) as a zero padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0.

```
import tkinter  as tk
from datetime import date
my_w = tk.Tk()
my_w.geometry("350x300") # change width height

def my_upd():
```

```
    dt=date.today().strftime('%U') # week number as
string
    print(dt)
    if(int(dt)%2==0):
        str1="Yes" # for even week number
    else:
        str1="No"  # for odd week number
    l1.config(text=str1) # update the text on label
b1=tk.Button(my_w,text='Click
me',command=lambda:my_upd())
b1.grid(row=1,column=1,padx=60,pady=30)
l1=tk.Label(my_w,text='Data')
l1.grid(row=1,column=2)
my_w.mainloop()
```

The Listbox widget is used to display a list of items from which a user can select a number of items.

Syntax

Here is the simple syntax to create this widget −

```
w = Listbox ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**bg** The normal background color displayed behind the label and indicator.

2	**bd** The size of the border around the indicator. Default is 2 pixels.
3	**cursor** The cursor that appears when the mouse is over the listbox.
4	**font** The font used for the text in the listbox.
5	**fg** The color used for the text in the listbox.
6	**height** Number of lines (not pixels!) shown in the listbox. Default is 10.
7	**highlightcolor** Color shown in the focus highlight when the widget has the focus.
8	**highlightthickness** Thickness of the focus highlight.
9	**relief** Selects three-dimensional border shading effects. The default is SUNKEN.
10	**selectbackground** The background color to use displaying selected text.
11	**selectmode** Determines how many items can be selected, and how mouse drags affect the selection − • **BROWSE** − Normally, you can only select one line out of a listbox. If you click on

	an item and then drag to a different line, the selection will follow the mouse. This is the default. • **SINGLE** − You can only select one line, and you can't drag the mouse.wherever you click button 1, that line is selected. • **MULTIPLE** − You can select any number of lines at once. Clicking on any line toggles whether or not it is selected. • **EXTENDED** − You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.
12	**width** The width of the widget in characters. The default is 20.
13	**xscrollcommand** If you want to allow the user to scroll the listbox horizontally, you can link your listbox widget to a horizontal scrollbar.
14	**yscrollcommand** If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar.

Methods

Methods on listbox objects include −

Sr.No.	Option & Description
1	**activate (index)** Selects the line specifies by the given index.
2	**curselection()** 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)** Deletes the lines whose indices are in the range [first, last]. If the second

	argument is omitted, the single line with index first is deleted.
4	**get (first, last=None)** Returns a tuple containing the text of the lines with indices from first to last, inclusive. If the second argument is omitted, returns the text of the line closest to first.
5	**index (i)** If possible, positions the visible part of the listbox so that the line containing index i is at the top of the widget.
6	**insert (index, *elements)** Insert one or more new lines into the listbox before the line specified by index. Use END as the first argument if you want to add new lines to the end of the listbox.
7	**nearest (y)** Return the index of the visible line closest to the y-coordinate y relative to the listbox widget.
8	**see (index)** Adjust the position of the listbox so that the line referred to by index is visible.
9	**size()** Returns the number of lines in the listbox.
10	**xview()** To make the listbox horizontally scrollable, set the command option of the associated horizontal scrollbar to this method.
11	**xview_moveto (fraction)** Scroll the listbox so that the leftmost fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].

12	**xview_scroll (number, what)** Scrolls the listbox horizontally. For the what argument, use either UNITS to scroll by characters, or PAGES to scroll by pages, that is, by the width of the listbox. The number argument tells how many to scroll.
13	**yview()** To make the listbox vertically scrollable, set the command option of the associated vertical scrollbar to this method.
14	**yview_moveto (fraction)** Scroll the listbox so that the top fraction of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].
15	**yview_scroll (number, what)** Scrolls the listbox vertically. For the what argument, use either UNITS to scroll by lines, or PAGES to scroll by pages, that is, by the height of the listbox. The number argument tells how many to scroll.

Example

Try the following example yourself −

```
from Tkinter import *
import tkMessageBox
import Tkinter

top = Tk()

Lb1 = Listbox(top)
Lb1.insert(1, "Python")
Lb1.insert(2, "Perl")
Lb1.insert(3, "C")
Lb1.insert(4, "PHP")
Lb1.insert(5, "JSP")
Lb1.insert(6, "Ruby")

Lb1.pack()
top.mainloop()
```

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

A menubutton is the part of a drop-down menu that stays on the screen all the time. Every menubutton is associated with a Menu widget that can display the choices for that menubutton when the user clicks on it.

Syntax

Here is the simple syntax to create this widget −

```
w = Menubutton ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**activebackground** The background color when the mouse is over the menubutton.
2	**activeforeground** The foreground color when the mouse is over the menubutton.
3	**anchor** This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text.
4	**bg**

	The normal background color displayed behind the label and indicator.
5	**bitmap** To display a bitmap on the menubutton, set this option to a bitmap name.
6	**bd** The size of the border around the indicator. Default is 2 pixels.
7	**cursor** The cursor that appears when the mouse is over this menubutton.
8	**direction** Set direction=LEFT to display the menu to the left of the button; use direction=RIGHT to display the menu to the right of the button; or use direction='above' to place the menu above the button.
9	**disabledforeground** The foreground color shown on this menubutton when it is disabled.
10	**fg** The foreground color when the mouse is not over the menubutton.
11	**height** The height of the menubutton in lines of text (not pixels!). The default is to fit the menubutton's size to its contents.
12	**highlightcolor** Color shown in the focus highlight when the widget has the focus.
13	**image** To display an image on this menubutton,

14	**justify** This option controls where the text is located when the text doesn't fill the menubutton: use justify=LEFT to left-justify the text (this is the default); use justify=CENTER to center it, or justify=RIGHT to right-justify.
15	**menu** To associate the menubutton with a set of choices, set this option to the Menu object containing those choices. That menu object must have been created by passing the associated menubutton to the constructor as its first argument.
16	**padx** How much space to leave to the left and right of the text of the menubutton. Default is 1.
17	**pady** How much space to leave above and below the text of the menubutton. Default is 1.
18	**relief** Selects three-dimensional border shading effects. The default is RAISED.
19	**state** Normally, menubuttons respond to the mouse. Set state=DISABLED to gray out the menubutton and make it unresponsive.
20	**text** To display text on the menubutton, set this option to the string containing the desired text. Newlines ("\n") within the string will cause line breaks.
21	**textvariable** You can associate a control variable of class StringVar with this menubutton. Setting that control variable will change the displayed text.

22	**underline** Normally, no underline appears under the text on the menubutton. To underline one of the characters, set this option to the index of that character.
23	**width** The width of the widget in characters. The default is 20.
24	**wraplength** Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Example

Try the following example yourself −

```
from tkinter import *

top = Tk()

mb=  Menubutton ( top, text="condiments", relief=RAISED )
mb.grid()
mb.menu =  Menu ( mb, tearoff = 0 )
mb["menu"] =  mb.menu

mayoVar = IntVar()
ketchVar = IntVar()

mb.menu.add_checkbutton ( label="mayo",
                          variable=mayoVar )
mb.menu.add_checkbutton ( label="ketchup",
                          variable=ketchVar )

mb.pack()
top.mainloop()
```

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

This widget provides a multiline and noneditable object that displays texts, automatically breaking lines and justifying their contents.

Its functionality is very similar to the one provided by the Label widget, except that it can also automatically wrap the text, maintaining a given width or aspect ratio.

Syntax

Here is the simple syntax to create this widget −

```
w = Message ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**anchor** This options controls where the text is positioned if the widget has more space than the text needs. The default is anchor=CENTER, which centers the text in the available space.
2	**bg** The normal background color displayed behind the label and indicator.
3	**bitmap** Set this option equal to a bitmap or image object and the label will display that graphic.

4	**bd** The size of the border around the indicator. Default is 2 pixels.
5	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
6	**font** If you are displaying text in this label (with the text or textvariable option, the font option specifies in what font that text will be displayed.
7	**fg** If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap.
8	**height** The vertical dimension of the new frame.
9	**image** To display a static image in the label widget, set this option to an image object.
10	**justify** Specifies how multiple lines of text will be aligned with respect to each other: LEFT for flush left, CENTER for centered (the default), or RIGHT for right-justified.
11	**padx** Extra space added to the left and right of the text within the widget. Default is 1.
12	**pady** Extra space added above and below the text within the widget. Default is 1.

13	**relief** Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
14	**text** To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ("\n") will force a line break.
15	**textvariable** To slave the text displayed in a label widget to a control variable of class *StringVar*, set this option to that variable.
16	**underline** You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
17	**width** Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
18	**wraplength** You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Example

Try the following example yourself −

```
from Tkinter import *

root = Tk()
var = StringVar()
label = Message( root, textvariable=var, relief=RAISED )

var.set("Hey!? How are you doing?")
label.pack()
```

```
root.mainloop()
```

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

This widget implements a multiple-choice button, which is a way to offer many possible selections to the user and lets user choose only one of them.

In order to implement this functionality, each group of radiobuttons must be associated to the same variable and each one of the buttons must symbolize a single value. You can use the Tab key to switch from one radionbutton to another.

Syntax

Here is the simple syntax to create this widget −

```
w = Radiobutton ( master, option, ...  )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**activebackground** The background color when the mouse is over the radiobutton.
2	**activeforeground** The foreground color when the mouse is over the radiobutton.
3	**anchor** If the widget inhabits a space larger than it needs, this option specifies where the radiobutton will sit in that space. The default is anchor=CENTER.
4	**bg**

	The normal background color behind the indicator and label.
5	**bitmap** To display a monochrome image on a radiobutton, set this option to a bitmap.
6	**borderwidth** The size of the border around the indicator part itself. Default is 2 pixels.
7	**command** A procedure to be called every time the user changes the state of this radiobutton.
8	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the radiobutton.
9	**font** The font used for the text.
10	**fg** The color used to render the text.
11	**height** The number of lines (not pixels) of text on the radiobutton. Default is 1.
12	**highlightbackground** The color of the focus highlight when the radiobutton does not have focus.
13	**highlightcolor** The color of the focus highlight when the radiobutton has the focus.
14	**image**

	To display a graphic image instead of text for this radiobutton, set this option to an image object.
15	**justify** If the text contains multiple lines, this option controls how the text is justified: CENTER (the default), LEFT, or RIGHT.
16	**padx** How much space to leave to the left and right of the radiobutton and text. Default is 1.
17	**pady** How much space to leave above and below the radiobutton and text. Default is 1.
18	**relief** Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
19	**selectcolor** The color of the radiobutton when it is set. Default is red.
20	**selectimage** If you are using the image option to display a graphic instead of text when the radiobutton is cleared, you can set the selectimage option to a different image that will be displayed when the radiobutton is set.
21	**state** The default is state=NORMAL, but you can set state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is ACTIVE.
22	**text** The label displayed next to the radiobutton. Use newlines ("\n") to display multiple lines of text.

23	**textvariable** To slave the text displayed in a label widget to a control variable of class *StringVar*, set this option to that variable.
24	**underline** You can display an underline (_) below the nth letter of the text, counting from 0, by setting this option to n. The default is underline=-1, which means no underlining.
25	**value** When a radiobutton is turned on by the user, its control variable is set to its current value option. If the control variable is an *IntVar*, give each radiobutton in the group a different integer value option. If the control variable is a *StringVar*, give each radiobutton a different string value option.
26	**variable** The control variable that this radiobutton shares with the other radiobuttons in the group. This can be either an IntVar or a StringVar.
27	**width** Width of the label in characters (not pixels!). If this option is not set, the label will be sized to fit its contents.
28	**wraplength** You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

Methods

Sr.No.	Method & Description
1	**deselect()** Clears (turns off) the radiobutton.

2	**flash()** Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.
3	**invoke()** You can call this method to get the same actions that would occur if the user clicked on the radiobutton to change its state.
4	**select()** Sets (turns on) the radiobutton.

Example

Try the following example yourself −

```
from Tkinter import *

def sel():
   selection = "You selected the option " + str(var.get())
   label.config(text = selection)

root = Tk()
var = IntVar()
R1 = Radiobutton(root, text="Option 1", variable=var, value=1,
                 command=sel)
R1.pack( anchor = W )

R2 = Radiobutton(root, text="Option 2", variable=var, value=2,
                 command=sel)
R2.pack( anchor = W )

R3 = Radiobutton(root, text="Option 3", variable=var, value=3,
                 command=sel)
R3.pack( anchor = W)

label = Label(root)
label.pack()
root.mainloop()
```

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

The Scale widget provides a graphical slider object that allows you to select values from a specific scale.

Syntax

Here is the simple syntax to create this widget −

```
w = Scale ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**activebackground** The background color when the mouse is over the scale.
2	**bg** The background color of the parts of the widget that are outside the trough.
3	**bd** Width of the 3-d border around the trough and slider. Default is 2 pixels.
4	**command** A procedure to be called every time the slider is moved. This procedure will be passed one argument, the new scale value. If the slider is moved rapidly, you may not get a callback for every possible position, but you'll certainly get a callback when it settles.

5	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the scale.
6	**digits** The way your program reads the current value shown in a scale widget is through a control variable. The control variable for a scale can be an IntVar, a DoubleVar (float), or a StringVar. If it is a string variable, the digits option controls how many digits to use when the numeric scale value is converted to a string.
7	**font** The font used for the label and annotations.
8	**fg** The color of the text used for the label and annotations.
9	**from_** A float or integer value that defines one end of the scale's range.
10	**highlightbackground** The color of the focus highlight when the scale does not have focus.
11	**highlightcolor** The color of the focus highlight when the scale has the focus.
12	**label** You can display a label within the scale widget by setting this option to the label's text. The label appears in the top left corner if the scale is horizontal, or the top right corner if vertical. The default is no label.
13	**length** The length of the scale widget. This is the x dimension if the scale is horizontal, or the y dimension if vertical. The default is 100 pixels.

14	**orient** Set orient=HORIZONTAL if you want the scale to run along the x dimension, or orient=VERTICAL to run parallel to the y-axis. Default is horizontal.
15	**relief** Specifies the appearance of a decorative border around the label. The default is FLAT; for other values.
16	**repeatdelay** This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
17	**resolution** Normally, the user will only be able to change the scale in whole units. Set this option to some other value to change the smallest increment of the scale's value. For example, if from_=-1.0 and to=1.0, and you set resolution=0.5, the scale will have 5 possible values: -1.0, -0.5, 0.0, +0.5, and +1.0.
18	**showvalue** Normally, the current value of the scale is displayed in text form by the slider (above it for horizontal scales, to the left for vertical scales). Set this option to 0 to suppress that label.
19	**sliderlength** Normally the slider is 30 pixels along the length of the scale. You can change that length by setting the sliderlength option to your desired length.
20	**state** Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set state=DISABLED to make the widget unresponsive.
21	**takefocus** Normally, the focus will cycle through scale widgets. Set this option to 0 if you

	don't want this behavior.
22	**tickinterval** To display periodic scale values, set this option to a number, and ticks will be displayed on multiples of that value. For example, if from_=0.0, to=1.0, and tickinterval=0.25, labels will be displayed along the scale at values 0.0, 0.25, 0.50, 0.75, and 1.00. These labels appear below the scale if horizontal, to its left if vertical. Default is 0, which suppresses display of ticks.
23	**to** A float or integer value that defines one end of the scale's range; the other end is defined by the from_ option, discussed above. The to value can be either greater than or less than the from_ value. For vertical scales, the to value defines the bottom of the scale; for horizontal scales, the right end.
24	**troughcolor** The color of the trough.
25	**variable** The control variable for this scale, if any. Control variables may be from class IntVar, DoubleVar (float), or StringVar. In the latter case, the numerical value will be converted to a string.
26	**width** The width of the trough part of the widget. This is the x dimension for vertical scales and the y dimension if the scale has orient=HORIZONTAL. Default is 15 pixels.

Methods

Scale objects have these methods −

Sr.No.	Method & Description
1	**get()**

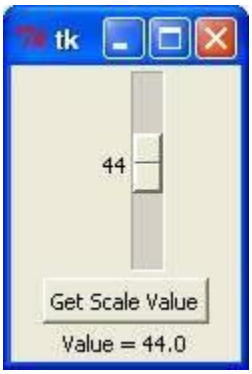
	This method returns the current value of the scale.
2	**set (value)** Sets the scale's value.

Example

Try the following example yourself −

```
from Tkinter import *

def sel():
   selection = "Value = " + str(var.get())
   label.config(text = selection)

root = Tk()
var = DoubleVar()
scale = Scale( root, variable = var )
scale.pack(anchor=CENTER)

button = Button(root, text="Get Scale Value", command=sel)
button.pack(anchor=CENTER)

label = Label(root)
label.pack()

root.mainloop()
```

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



This widget provides a slide controller that is used to implement vertical scrolled widgets, such as Listbox, Text and Canvas. Note that you can also create horizontal scrollbars on Entry widgets.

Syntax

Here is the simple syntax to create this widget −

```
w = Scrollbar ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**activebackground** The color of the slider and arrowheads when the mouse is over them.
2	**bg** The color of the slider and arrowheads when the mouse is not over them.
3	**bd** The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a 2-pixel border around the arrowheads and slider.
4	**command** A procedure to be called whenever the scrollbar is moved.
5	**cursor** The cursor that appears when the mouse is over the scrollbar.
6	**elementborderwidth** The width of the borders around the arrowheads and slider. The default is elementborderwidth=-1, which means to use the value of the borderwidth option.
7	**highlightbackground**

	The color of the focus highlight when the scrollbar does not have focus.
8	**highlightcolor** The color of the focus highlight when the scrollbar has the focus.
9	**highlightthickness** The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
10	**jump** This option controls what happens when a user drags the slider. Normally (jump=0), every small drag of the slider causes the command callback to be called. If you set this option to 1, the callback isn't called until the user releases the mouse button.
11	**orient** Set orient=HORIZONTAL for a horizontal scrollbar, orient=VERTICAL for a vertical one.
12	**repeatdelay** This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is repeatdelay=300, and the units are milliseconds.
13	**repeatinterval** repeatinterval
14	**takefocus** Normally, you can tab the focus through a scrollbar widget. Set takefocus=0 if you don't want this behavior.
15	**troughcolor** The color of the trough.

16	**width** Width of the scrollbar (its y dimension if horizontal, and its x dimension if vertical). Default is 16.

Methods

Scrollbar objects have these methods −

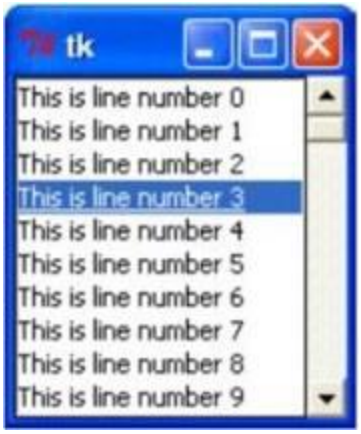
Sr.No.	Method & Description
1	**get()** Returns two numbers (a, b) describing the current position of the slider. The a value gives the position of the left or top edge of the slider, for horizontal and vertical scrollbars respectively; the b value gives the position of the right or bottom edge.
2	**set (first, last)** To connect a scrollbar to another widget w, set w's xscrollcommand or yscrollcommand to the scrollbar's set() method. The arguments have the same meaning as the values returned by the get() method.

Example

Try the following example yourself −

```
from Tkinter import *

root = Tk()
scrollbar = Scrollbar(root)
scrollbar.pack( side = RIGHT, fill = Y )

mylist = Listbox(root, yscrollcommand = scrollbar.set )
for line in range(100):
   mylist.insert(END, "This is line number " + str(line))

mylist.pack( side = LEFT, fill = BOTH )
scrollbar.config( command = mylist.yview )

mainloop()
```

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



Text widgets provide advanced capabilities that allow you to edit a multiline text and format the way it has to be displayed, such as changing its color and font.

You can also use elegant structures like tabs and marks to locate specific sections of the text, and apply changes to those areas. Moreover, you can embed windows and images in the text because this widget was designed to handle both plain and formatted text.

Syntax

Here is the simple syntax to create this widget −

```
w = Text ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**bg** The default background color of the text widget.
2	**bd** The width of the border around the text widget. Default is 2 pixels.
3	**cursor**

	The cursor that will appear when the mouse is over the text widget.
4	**exportselection** Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
5	**font** The default font for text inserted into the widget.
6	**fg** The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
7	**height** The height of the widget in lines (not pixels!), measured according to the current font size.
8	**highlightbackground** The color of the focus highlight when the text widget does not have focus.
9	**highlightcolor** The color of the focus highlight when the text widget has the focus.
10	**highlightthickness** The thickness of the focus highlight. Default is 1. Set highlightthickness=0 to suppress display of the focus highlight.
11	**insertbackground** The color of the insertion cursor. Default is black.
12	**insertborderwidth** Size of the 3-D border around the insertion cursor. Default is 0.

13	**insertofftime** The number of milliseconds the insertion cursor is off during its blink cycle. Set this option to zero to suppress blinking. Default is 300.
14	**insertontime** The number of milliseconds the insertion cursor is on during its blink cycle. Default is 600.
15	**insertwidth** Width of the insertion cursor (its height is determined by the tallest item in its line). Default is 2 pixels.
16	**padx** The size of the internal padding added to the left and right of the text area. Default is one pixel.
17	**pady** The size of the internal padding added above and below the text area. Default is one pixel.
18	**relief** The 3-D appearance of the text widget. Default is relief=SUNKEN.
19	**selectbackground** The background color to use displaying selected text.
20	**selectborderwidth** The width of the border to use around selected text.
21	**spacing1** This option specifies how much extra vertical space is put above each line of text. If a line wraps, this space is added only before the first line it occupies on the display. Default is 0.

22	**spacing2** This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. Default is 0.
23	**spacing3** This option specifies how much extra vertical space is added below each line of text. If a line wraps, this space is added only after the last line it occupies on the display. Default is 0.
24	**state** Normally, text widgets respond to keyboard and mouse events; set state=NORMAL to get this behavior. If you set state=DISABLED, the text widget will not respond, and you won't be able to modify its contents programmatically either.
25	**tabs** This option controls how tab characters position text.
26	**width** The width of the widget in characters (not pixels!), measured according to the current font size.
27	**wrap** This option controls the display of lines that are too wide. Set wrap=WORD and it will break the line after the last word that will fit. With the default behavior, wrap=CHAR, any line that gets too long will be broken at any character.
28	**xscrollcommand** To make the text widget horizontally scrollable, set this option to the set() method of the horizontal scrollbar.
29	**yscrollcommand** To make the text widget vertically scrollable, set this option to the set() method of the vertical scrollbar.

Methods

Text objects have these methods −

Sr.No.	Methods & Description
1	**delete(startindex [,endindex])** This method deletes a specific character or a range of text.
2	**get(startindex [,endindex])** This method returns a specific character or a range of text.
3	**index(index)** Returns the absolute value of an index based on the given index.
4	**insert(index [,string]...)** This method inserts strings at the specified index location.
5	**see(index)** This method returns true if the text located at the index position is visible.

Text widgets support three distinct helper structures: Marks, Tabs, and Indexes −

Marks are used to bookmark positions between two characters within a given text. We have the following methods available when handling marks −

Sr.No.	Methods & Description
1	**index(mark)** Returns the line and column location of a specific mark.
2	**mark_gravity(mark [,gravity])** Returns the gravity of the given mark. If the second argument is provided, the gravity is set for the given mark.

3	**mark_names()** Returns all marks from the Text widget.
4	**mark_set(mark, index)** Informs a new position to the given mark.
5	**mark_unset(mark)** Removes the given mark from the Text widget.

Tags are used to associate names to regions of text which makes easy the task of modifying the display settings of specific text areas. Tags are also used to bind event callbacks to specific ranges of text.

Following are the available methods for handling tabs −

Sr.No.	Methods & Description
1	**tag_add(tagname, startindex[,endindex] ...)** This method tags either the position defined by startindex, or a range delimited by the positions startindex and endindex.
2	**tag_config** You can use this method to configure the tag properties, which include, justify(center, left, or right), tabs(this property has the same functionality of the Text widget tabs's property), and underline(used to underline the tagged text).
3	**tag_delete(tagname)** This method is used to delete and remove a given tag.
4	**tag_remove(tagname [,startindex[.endindex]] ...)** After applying this method, the given tag is removed from the provided area without deleting the actual tag definition.

Example

Try the following example yourself −

```
from Tkinter import *

def onclick():
   pass

root = Tk()
text = Text(root)
text.insert(INSERT, "Hello.....")
text.insert(END, "Bye Bye.....")
text.pack()

text.tag_add("here", "1.0", "1.4")
text.tag_add("start", "1.8", "1.13")
text.tag_config("here", background="yellow", foreground="blue")
text.tag_config("start", background="black", foreground="green")
root.mainloop()
```

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

The Entry widget is used to accept single-line text strings from a user.

- If you want to display multiple lines of text that can be edited, then you should use the *Text* widget.
- If you want to display one or more lines of text that cannot be modified by the user, then you should use the *Label* widget.

Syntax

Here is the simple syntax to create this widget −

```
w = Entry( master, option, ... )
```

Parameters

- **master** − This represents the parent window.

- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**bg** The normal background color displayed behind the label and indicator.
2	**bd** The size of the border around the indicator. Default is 2 pixels.
3	**command** A procedure to be called every time the user changes the state of this checkbutton.
4	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
5	**font** The font used for the text.
6	**exportselection** By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0.
7	**fg** The color used to render the text.
8	**highlightcolor** The color of the focus highlight when the checkbutton has the focus.
9	**justify**

	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
10	**relief** With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
11	**selectbackground** The background color to use displaying selected text.
12	**selectborderwidth** The width of the border to use around selected text. The default is one pixel.
13	**selectforeground** The foreground (text) color of selected text.
14	**show** Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
15	**state** The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
16	**textvariable** In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
17	**width** The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.

18	**xscrollcommand** If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar.

Methods

Following are commonly used methods for this widget −

Sr.No.	Method & Description
1	**delete (first, last=None)** Deletes characters from the widget, starting with the one at index first, up to but not including the character at position last. If the second argument is omitted, only the single character at position first is deleted.
2	**get()** Returns the entry's current text as a string.
3	**icursor (index)** Set the insertion cursor just before the character at the given index.
4	**index (index)** Shift the contents of the entry so that the character at the given index is the leftmost visible character. Has no effect if the text fits entirely within the entry.
5	**insert (index, s)** Inserts string s before the character at the given index.
6	**select_adjust (index)** This method is used to make sure that the selection includes the character at the specified index.
7	**select_clear()**

	Clears the selection. If there isn't currently a selection, has no effect.
8	**select_from (index)** Sets the ANCHOR index position to the character selected by index, and selects that character.
9	**select_present()** If there is a selection, returns true, else returns false.
10	**select_range (start, end)** Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position.
11	**select_to (index)** Selects all the text from the ANCHOR position up to but not including the character at the given index.
12	**xview (index)** This method is useful in linking the Entry widget to a horizontal scrollbar.
13	**xview_scroll (number, what)** Used to scroll the entry horizontally. The what argument must be either UNITS, to scroll by character widths, or PAGES, to scroll by chunks the size of the entry widget. The number is positive to scroll left to right, negative to scroll right to left.

Example

Try the following example yourself −

```
from Tkinter import *

top = Tk()
L1 = Label(top, text="User Name")
L1.pack( side = LEFT)
E1 = Entry(top, bd =5)
E1.pack(side = RIGHT)
```

```
top.mainloop()
```

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

The Canvas is a rectangular area intended for drawing pictures or other complex layouts. You can place graphics, text, widgets or frames on a Canvas.

Syntax

Here is the simple syntax to create this widget −

```
w = Canvas ( master, option=value, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**bd** Border width in pixels. Default is 2.
2	**bg** Normal background color.
3	**confine** If true (the default), the canvas cannot be scrolled outside of the scrollregion.
4	**cursor** Cursor used in the canvas like *arrow, circle, dot etc.*
5	**height**

	Size of the canvas in the Y dimension.
6	**highlightcolor** Color shown in the focus highlight.
7	**relief** Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
8	**scrollregion** A tuple (w, n, e, s) that defines over how large an area the canvas can be scrolled, where w is the left side, n the top, e the right side, and s the bottom.
9	**width** Size of the canvas in the X dimension.
10	**xscrollincrement** If you set this option to some positive dimension, the canvas can be positioned only on multiples of that distance, and the value will be used for scrolling by scrolling units, such as when the user clicks on the arrows at the ends of a scrollbar.
11	**xscrollcommand** If the canvas is scrollable, this attribute should be the .set() method of the horizontal scrollbar.
12	**yscrollincrement** Works like xscrollincrement, but governs vertical movement.
13	**yscrollcommand** If the canvas is scrollable, this attribute should be the .set() method of the vertical scrollbar.

The Canvas widget can support the following standard items −

arc − Creates an arc item, which can be a chord, a pieslice or a simple arc.

```
coord = 10, 50, 240, 210
arc = canvas.create_arc(coord, start=0, extent=150, fill="blue")
```

image − Creates an image item, which can be an instance of either the BitmapImage or the PhotoImage classes.

```
filename = PhotoImage(file = "sunshine.gif")
image = canvas.create_image(50, 50, anchor=NE, image=filename)
```

line − Creates a line item.

```
line = canvas.create_line(x0, y0, x1, y1, ..., xn, yn, options)
```

oval − Creates a circle or an ellipse at the given coordinates. It takes two pairs of coordinates; the top left and bottom right corners of the bounding rectangle for the oval.

```
oval = canvas.create_oval(x0, y0, x1, y1, options)
```

polygon − Creates a polygon item that must have at least three vertices.

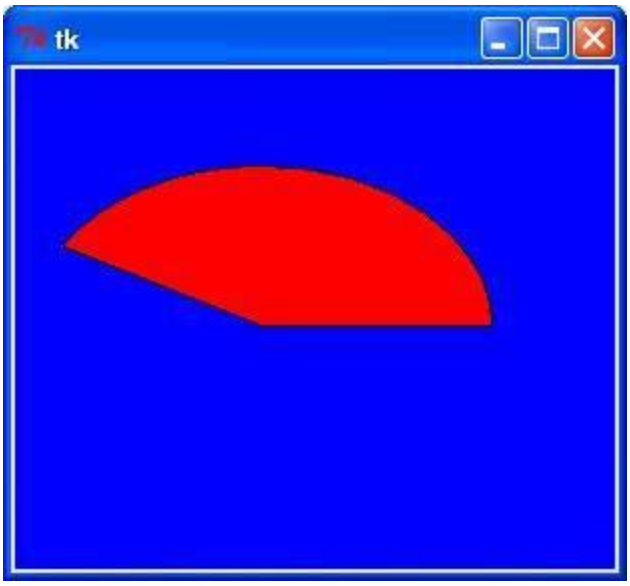
```
oval = canvas.create_polygon(x0, y0, x1, y1,...xn, yn, options)
```

Example

Try the following example yourself −

```
from tkinter import *

top = Tk()

top.geometry("200x200")

#creating a simple canvas
c = Canvas(top,bg = "pink",height = "200",width = 200)

arc = c.create_arc((5,10,150,200),start = 0,extent = 150, fill= "white")

c.pack()

top.mainloop()
```

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



The Checkbutton widget is used to display a number of options to a user as toggle buttons. The user can then select one or more options by clicking the button corresponding to each option.

You can also display images in place of text.

Syntax

Here is the simple syntax to create this widget −

```
w = Checkbutton ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**activebackground** Background color when the checkbutton is under the cursor.
2	**activeforeground** Foreground color when the checkbutton is under the cursor.

3	**bg** The normal background color displayed behind the label and indicator.
4	**bitmap** To display a monochrome image on a button.
5	**bd** The size of the border around the indicator. Default is 2 pixels.
6	**command** A procedure to be called every time the user changes the state of this checkbutton.
7	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
8	**disabledforeground** The foreground color used to render the text of a disabled checkbutton. The default is a stippled version of the default foreground color.
9	**font** The font used for the text.
10	**fg** The color used to render the text.
11	**height** The number of lines of text on the checkbutton. Default is 1.
12	**highlightcolor**

	The color of the focus highlight when the checkbutton has the focus.
13	**image** To display a graphic image on the button.
14	**justify** If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
15	**offvalue** Normally, a checkbutton's associated control variable will be set to 0 when it is cleared (off). You can supply an alternate value for the off state by setting offvalue to that value.
16	**onvalue** Normally, a checkbutton's associated control variable will be set to 1 when it is set (on). You can supply an alternate value for the on state by setting onvalue to that value.
17	**padx** How much space to leave to the left and right of the checkbutton and text. Default is 1 pixel.
18	**pady** How much space to leave above and below the checkbutton and text. Default is 1 pixel.
19	**relief** With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
20	**selectcolor** The color of the checkbutton when it is set. Default is selectcolor="red".

21	**selectimage** If you set this option to an image, that image will appear in the checkbutton when it is set.
22	**state** The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is ACTIVE.
23	**text** The label displayed next to the checkbutton. Use newlines ("\n") to display multiple lines of text.
24	**underline** With the default value of -1, none of the characters of the text label are underlined. Set this option to the index of a character in the text (counting from zero) to underline that character.
25	**variable** The control variable that tracks the current state of the checkbutton. Normally this variable is an *IntVar*, and 0 means cleared and 1 means set, but see the offvalue and onvalue options above.
26	**width** The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
27	**wraplength** Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Methods

Following are commonly used methods for this widget −

Sr.No.	Method & Description
1	**deselect()** Clears (turns off) the checkbutton.
2	**flash()** Flashes the checkbutton a few times between its active and normal colors, but leaves it the way it started.
3	**invoke()** You can call this method to get the same actions that would occur if the user clicked on the checkbutton to change its state.
4	**select()** Sets (turns on) the checkbutton.
5	**toggle()** Clears the checkbutton if set, sets it if cleared.

Example

Try the following example yourself −

```
from Tkinter import *
import tkMessageBox
import Tkinter

top = Tkinter.Tk()
CheckVar1 = IntVar()
CheckVar2 = IntVar()
C1 = Checkbutton(top, text = "Music", variable = CheckVar1, \
                 onvalue = 1, offvalue = 0, height=5, \
                 width = 20)
C2 = Checkbutton(top, text = "Video", variable = CheckVar2, \
                 onvalue = 1, offvalue = 0, height=5, \
                 width = 20)
C1.pack()
C2.pack()
```

```
top.mainloop()
```

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

The Frame widget is very important for the process of grouping and organizing other widgets in a somehow friendly way. It works like a container, which is responsible for arranging the position of other widgets.

It uses rectangular areas in the screen to organize the layout and to provide padding of these widgets. A frame can also be used as a foundation class to implement complex widgets.

Syntax

Here is the simple syntax to create this widget −

```
w = Frame ( master, option, ... )
```

Parameters

- **master** − This represents the parent window.
- **options** − Here is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas.

Sr.No.	Option & Description
1	**bg** The normal background color displayed behind the label and indicator.
2	**bd** The size of the border around the indicator. Default is 2 pixels.

3	**cursor** If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the checkbutton.
4	**height** The vertical dimension of the new frame.
5	**highlightbackground** Color of the focus highlight when the frame does not have focus.
6	**highlightcolor** Color shown in the focus highlight when the frame has the focus.
7	**highlightthickness** Thickness of the focus highlight.
8	**relief** With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles
9	**width** The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.

Example

Try the following example yourself −

```
from Tkinter import *

root = Tk()
frame = Frame(root)
frame.pack()

bottomframe = Frame(root)
```

```
bottomframe.pack( side = BOTTOM )

redbutton = Button(frame, text="Red", fg="red")
redbutton.pack( side = LEFT)

greenbutton = Button(frame, text="Brown", fg="brown")
greenbutton.pack( side = LEFT )

bluebutton = Button(frame, text="Blue", fg="blue")
bluebutton.pack( side = LEFT )

blackbutton = Button(bottomframe, text="Black", fg="black")
blackbutton.pack( side = BOTTOM)

root.mainloop()
```

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

Writing to Files in Python

In order to write into a file in Python, we need to open it in write `w`, append `a` or exclusive creation `x` mode.

We need to be careful with the `w` mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.

Writing a string or sequence of bytes (for binary files) is done using the `write()` method. This method returns the number of characters written to the file.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

This program will create a new file named `test.txt` in the current directory if it does not exist. If it does exist, it is overwritten.
We must include the newline characters ourselves to distinguish the different lines.

```
with open("welcome.txt") as file: # Use file to refer to
the file object

   data = file.read()

   do something with data
```

Opens output.txt in write mode

```
with open('output.txt', 'w') as file:  # Use file to
refer to the file object

    file.write('Hi there!')
```

What is Pandas?

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

Pandas allows us to analyze big data and make conclusions based on statistical theories.

Pandas can clean messy data sets, and make them readable and relevant.

Relevant data is very important in data science.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?

Pandas are also able to delete rows that are not relevant, or contain wrong values, like empty or NULL values. This is called *cleaning* the data.

Installation of Pandas

If you have Python and PIP already installed on a system, then installation of Pandas is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install pandas
```

What is PIP?

PIP is a package manager for Python packages, or modules if you like.

Note: If you have Python version 3.4 or later, PIP is included by default.

What is a Package?

A package contains all the files you need for a module.

Modules are Python code libraries you can include in your project.

Check if PIP is Installed

Navigate your command line to the location of Python's script directory, and type the following:

Example

Check PIP version:

```
C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip --version
```

```
import pandas as pd

df = pd.read_csv('Quiz-2-FCS.csv')
#print(df.head())

specified_column=df[['Enrollment_No','Username','Mobile Number']]

print(specified_column)



Laptop_dict = {'Product':
['Laptop','Printer','Monitor','Tablet'],
        'Price': [1200,100,300,150]}
#print(Laptop_dict)
Laptop_df=pd.DataFrame(Laptop_dict)
#print(Laptop_df)
```

```
#print(df.to_string())

#print(df['Batch'].unique())
#df1=df[df['score']>5]
#print(df1)

#df1.to_csv('new_csv.csv')
```

Exercise 1: From the given dataset print the first and last five rows

```
import pandas as pd
df = pd.read_csv("D:\\Python\\Articles\\pandas\\automobile-
dataset\\Automobile_data.csv")
df.head(n=2)
```

```
import pandas as pd

df = pd.read_csv("D:\\Python\\Articles\\pandas\\automobile-
dataset\\Automobile_data.csv")

df.tail(5)
```

Find the most expensive car company name

```
import pandas as pd
df = pd.read_csv("Automobile_data.csv")
df = df [['company','price']][df.price==df['price'].max()]
print(df)
```

Print All Toyota Cars details

```
import pandas as pd
df = pd.read_csv("Automobile_data.csv")
car_Manufacturers = df.groupby('company')
toyotaDf = car_Manufacturers.get_group('toyota')
print(toyotaDf.to_string())
```

Count total cars per company

```
import pandas as pd
df = pd.read_csv("Automobile_data.csv")
print(df['company'].value_counts())
```

The Common Gateway Interface, or CGI, is a set of standards that define how information is exchanged between the web server and a custom script. The CGI specs are currently maintained by the NCSA.

What is CGI?

- The Common Gateway Interface, or CGI, is a standard for external gateway programs to interface with information servers such as HTTP servers.
- The current version is CGI/1.1 and CGI/1.2 is under progress.

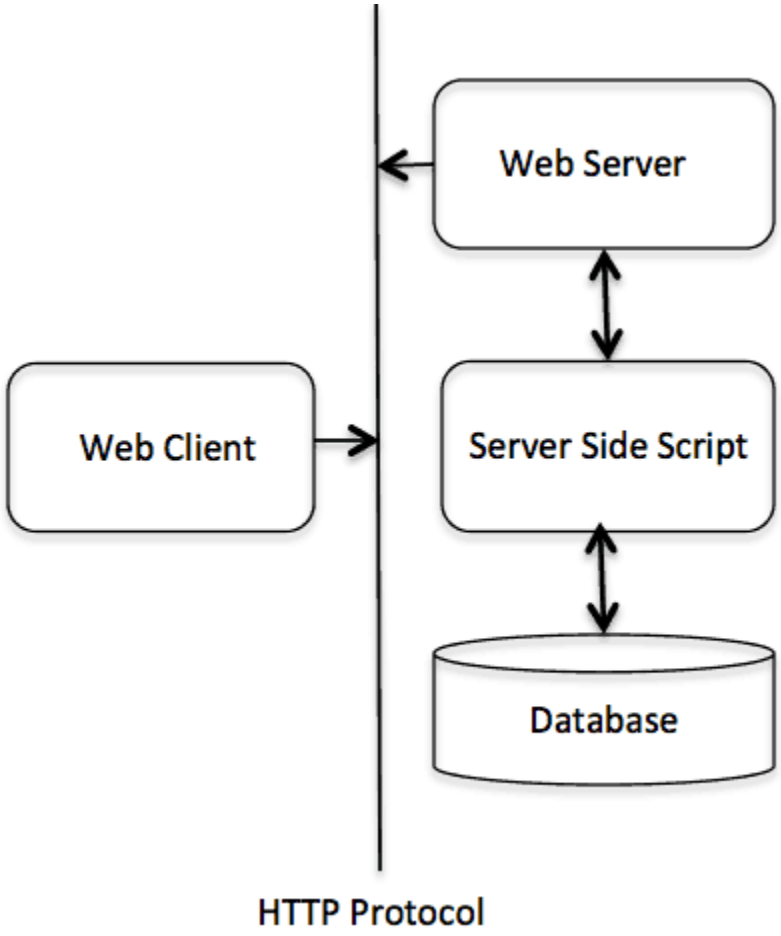
Web Browsing

To understand the concept of CGI, let us see what happens when we click a hyper link to browse a particular web page or URL.

- Your browser contacts the HTTP web server and demands for the URL, i.e., filename.
- Web Server parses the URL and looks for the filename. If it finds that file then sends it back to the browser, otherwise sends an error message indicating that you requested a wrong file.
- Web browser takes response from web server and displays either the received file or error message.

However, it is possible to set up the HTTP server so that whenever a file in a certain directory is requested that file is not sent back; instead it is executed as a program, and whatever that program outputs is sent back for your browser to display. This function is called the Common Gateway Interface or CGI and the programs are called CGI scripts. These CGI programs can be a Python Script, PERL Script, Shell Script, C or C++ program, etc.

CGI Architecture Diagram



CGI is a set of standards that defines a standard way of passing information or web-user requests to an application program and getting data back to forward it to users. It is the exchange of information between the web server and a custom script. When the users requested the web-page, the server sends the requested web-page. The web server usually passes the information to all application programs that process data and sends back an acknowledged message; this technique of passing data back-and-forth between server and application is the Common Gateway Interface. The current version of CGI is CGI/1.1 & CGI/1.2 is under process.

Browsing

What happens when a user clicks a hyperlink to browse a particular web-page or URL (Uniform Resource Locator).

The steps are:

- Browser contacts the HTTP web server for demanding the URL

- Parsing the URL
- Look for the filename
- If it finds that file, a request is sent back
- Web browser takes a response from the webserver
- As the server response, it either shows the received file or an error message.

It may become possible to set-up an HTTP server because when a certain directory is requested, that file is not sent back; instead, it is executed as a program, and that program's output is displayed back to your browser.

Configuring CGI

The steps are:

1. Find out which user is running the Web-server
2. Check for server configuration to see if you can run the scripts in a particular directory
3. Check for file's permission
4. Make a clear assurance that scripts you made are readable and executable by the webserver user
5. Make sure the Python-Script's first line refers to a webserver that the interpreter can run

What are cookies on websites?

Cookies are small files of information that a web server generates and sends to a web browser. Web browsers store the cookies they receive for a predetermined period of time, or for the length of a user's session on a website. They attach the relevant cookies to any future requests the user makes of the web server.

Cookies help inform websites about the user, enabling the websites to personalize the user experience. For example, ecommerce websites use cookies to know what merchandise users have placed in their shopping carts. In addition, some cookies are necessary for security purposes, such as authentication cookies (see below).

The cookies that are used on the Internet are also called "HTTP cookies." Like much of the web, cookies are sent using the HTTP protocol.

Where are cookies stored?

Web browsers store cookies in a designated file on users' devices. The Google Chrome web browser, for instance, stores all cookies in a file labeled "Cookies." Chrome users can view the cookies stored by the browser by opening developer tools, clicking the "Application" tab, and clicking on "Cookies" in the left side menu.

What are cookies used for?

User sessions: Cookies help associate website activity with a specific user. A session cookie contains a unique string (a combination of letters and numbers) that matches a user session with relevant data and content for that user.

Suppose Alice has an account on a shopping website. She logs into her account from the website's homepage. When she logs in, the website's server generates a session cookie and sends the cookie to Alice's browser. This cookie tells the website to load Alice's account content, so that the homepage now reads, "Welcome, Alice."

Alice then clicks to a product page displaying a pair of jeans. When Alice's web browser sends an HTTP request to the website for the jeans product page, it includes Alice's session cookie with the request. Because the website has this cookie, it recognizes the user as Alice, and she does not have to log in again when the new page loads.

Personalization: Cookies help a website "remember" user actions or user preferences, enabling the website to customize the user's experience.

If Alice logs out of the shopping website, her username can be stored in a cookie and sent to her web browser. Next time she loads that website, the web browser sends this cookie to the web server, which then prompts Alice to log in with the username she used last time.

Tracking: Some cookies record what websites users visit. This information is sent to the server that originated the cookie the next time the browser has to load content from

that server. With third-party tracking cookies, this process takes place anytime the browser loads a website that uses that tracking service.

If Alice has previously visited a website that sent her browser a tracking cookie, this cookie may record that Alice is now viewing a product page for jeans. The next time Alice loads a website that uses this tracking service, she may see ads for jeans.

However, advertising is not the only use for tracking cookies. Many analytics services also use tracking cookies to anonymously record user activity. (Cloudflare Web Analytics is one of the few services that does not use cookies to provide analytics, helping to protect user privacy.)

What are the different types of cookies?

Some of the most important types of cookies to know include:

Session cookies

A session cookie helps a website track a user's session. Session cookies are deleted after a user's session ends — once they log out of their account on a website or exit the website. Session cookies have no expiration date, which signifies to the browser that they should be deleted once the session is over.

Persistent cookies

Unlike session cookies, persistent cookies remain in a user's browser for a predetermined length of time, which could be a day, a week, several months, or even years. Persistent cookies always contain an expiration date.

Authentication cookies

Authentication cookies help manage user sessions; they are generated when a user logs into an account via their browser. They ensure that sensitive information is delivered to the correct user sessions by associating user account information with a cookie identifier string.

Tracking cookies

Tracking cookies are generated by tracking services. They record user activity, and browsers send this record to the associated tracking service the next time they load a website that uses that tracking service.

Zombie cookies

Like the "zombies" of popular fiction, zombie cookies regenerate after they are deleted. Zombie cookies create backup versions of themselves outside of a browser's typical cookie storage location. They use these backups to reappear within a browser after they are deleted. Zombie cookies are sometimes used by unscrupulous ad networks, and even by cyber attackers.

What is a third-party cookie?

A third-party cookie is a cookie that belongs to a domain other than the one displayed in the browser. Third-party cookies are most often used for tracking purposes. They contrast with first-party cookies, which are associated with the same domain that appears in the user's browser.

When Alice does her shopping at jeans.example.com, the jeans.example.com origin server uses a session cookie to remember that she has logged into her account. This is an example of a first-party cookie. However, Alice may not be aware that a cookie from example.ad-network.com is also stored in her browser and is tracking her activity on jeans.example.com, even though she is not currently accessing example.ad-network.com. This is an example of a third-party cookie.

How do cookies affect user privacy?

As described above, cookies can be used to record browsing activity, including for advertising purposes. However, many users do not want their online behavior to be tracked. Users also lack visibility or control over what tracking services do with the data they collect.

Even when cookie-based tracking is not tied to a specific user's name or device, with some types of tracking it could still be possible to link a record of a user's browsing activity with their real identity. This information could be used in any number of ways, from unwanted advertising to the monitoring, stalking, or harassment of users. (This is not the case with all cookie usage.)

Some privacy laws, like the EU's ePrivacy Directive, address and govern the use of cookies. Under this directive, users have to provide "informed consent" — they have to be notified of how the website uses cookies and agree to this usage — before the website can use cookies. (The exception to this is cookies that are "strictly necessary" for the website to function.) The EU's General Data Protection Regulation (GDPR) considers cookie identifiers to be personal data, so its rules apply to cookie usage in the EU as well. Also, any personal data collected by cookies falls under the GDPR's jurisdiction.

Largely because of these laws, many websites now display cookie banners that allow users to review and control the cookies those websites use.

What is Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

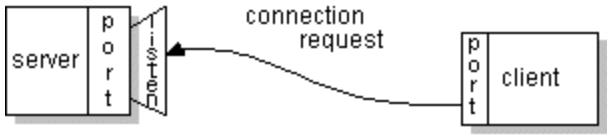
Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

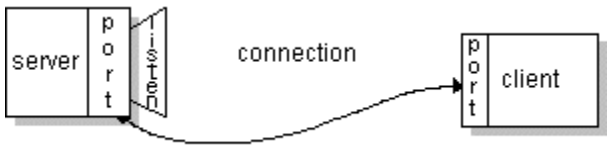
What Is a Socket?

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the .server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

Sockets have their own vocabulary −

Sr.No.	Term & Description
1	**Domain** The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.
2	**Type** The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
3	**Protocol** Typically zero, this may be used to identify a variant of a protocol within a domain and type.
4	**Hostname** The identifier of a network interface − • A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation • A string "<broadcast>", which specifies an INADDR_BROADCAST address. • A zero-length string, which specifies INADDR_ANY, or • An Integer, interpreted as a binary address in host byte order.

5	**Port** Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

The *socket* Module

To create a socket, you must use the *socket.socket()* function available in *socket* module, which has the general syntax −

```
s = socket.socket (socket_family, socket_type, protocol=0)
```

Here is the description of the parameters −

- **socket_family** − This is either AF_UNIX or AF_INET, as explained earlier.
- **socket_type** − This is either SOCK_STREAM or SOCK_DGRAM.
- **protocol** − This is usually left out, defaulting to 0.

Once you have *socket* object, then you can use required functions to create your client or server program. Following is the list of functions required −

Server Socket Methods

Sr.No.	Method & Description
1	**s.bind()** This method binds address (hostname, port number pair) to socket.
2	**s.listen()** This method sets up and start TCP listener.
3	**s.accept()** This passively accept TCP client connection, waiting until connection arrives (blocking).

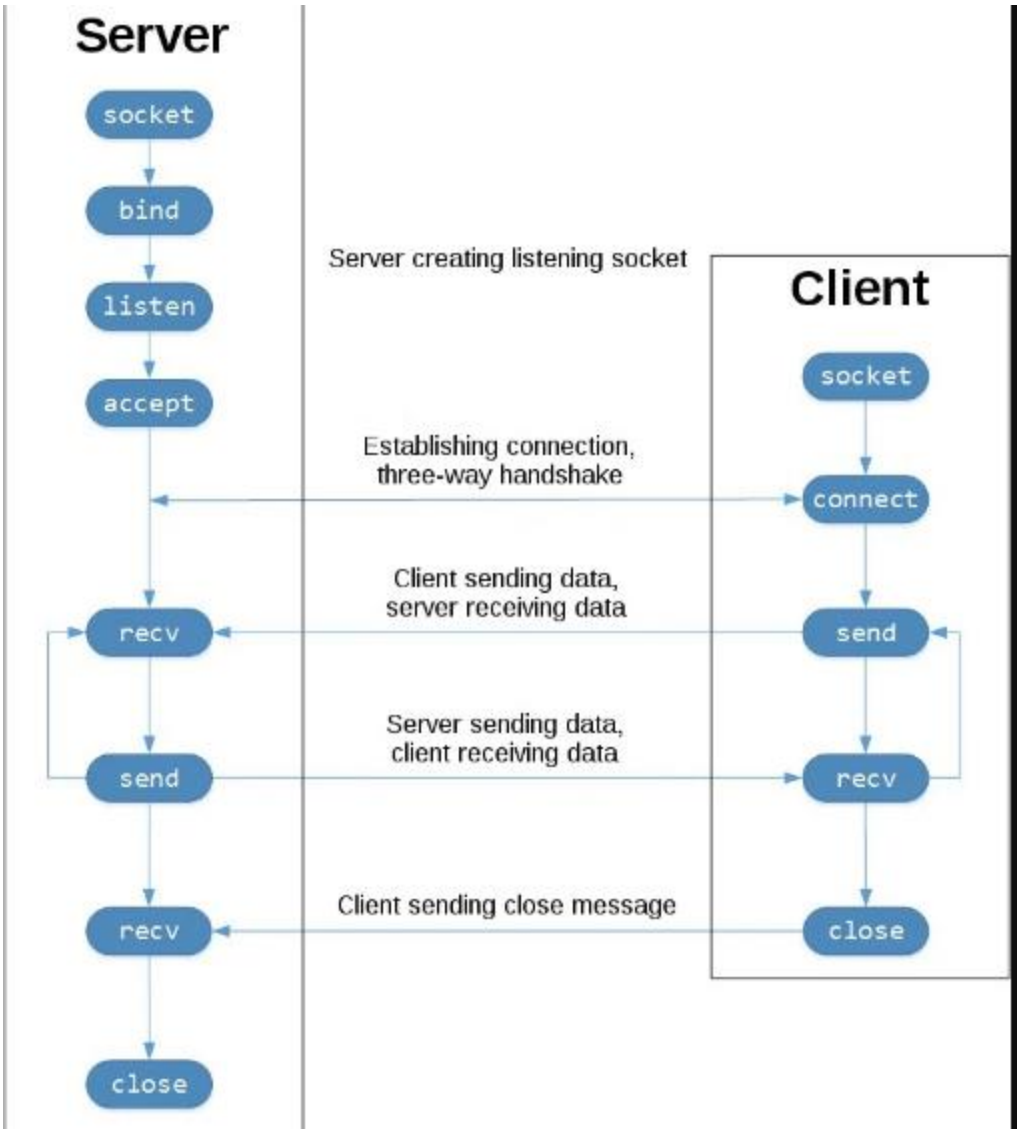
Client Socket Methods

Sr.No.	Method & Description

1	**s.connect()** This method actively initiates TCP server connection.

General Socket Methods

Sr.No.	Method & Description
1	**s.recv()** This method receives TCP message
2	**s.send()** This method transmits TCP message
3	**s.recvfrom()** This method receives UDP message
4	**s.sendto()** This method transmits UDP message
5	**s.close()** This method closes socket
6	**socket.gethostname()** Returns the hostname.



A Simple Server

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call **bind(hostname, port)** function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

```
#!/usr/bin/python           # This is server.py file

import socket               # Import socket module
```

```
s = socket.socket()          # Create a socket object
host = socket.gethostname()  # Get local machine name
port = 12345                 # Reserve a port for your service.
s.bind((host, port))         # Bind to the port

s.listen(5)                  # Now wait for client connection.
while True:
   c, addr = s.accept()      # Establish connection with client.
   print 'Got connection from', addr
   c.send('Thank you for connecting')
   c.close()                 # Close the connection
```

A Simple Client

Let us write a very simple client program which opens a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's *socket* module function.

The **socket.connect(hosname, port)** opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits −

```
#!/usr/bin/python            # This is client.py file

import socket                # Import socket module

s = socket.socket()          # Create a socket object
host = socket.gethostname()  # Get local machine name
port = 12345                 # Reserve a port for your service.

s.connect((host, port))
print s.recv(1024)
s.close()                     # Close the socket when done
```

Now run this server.py in background and then run above client.py to see the result.

```
# Following would start a server in background.
$ python server.py &

# Once server is started run client as follows:
$ python client.py
```

This would produce following result −

```
Got connection from ('127.0.0.1', 48437)
Thank you for connecting
```

Python Internet modules

A list of some important modules in Python Network/Internet programming.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

Please check all the libraries mentioned above to work with FTP, SMTP, POP, and IMAP protocols.

Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an `AF_UNIX` socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error

handler (see **PEP 383**). An address in Linux's abstract namespace is returned as a bytes-like object with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or bytes-like object can be used for either type of address when passing it as an argument.

Changed in version 3.3: Previously, `AF_UNIX` socket paths were assumed to use UTF-8 encoding.

Changed in version 3.5: Writable bytes-like object is now accepted.

- A pair `(host, port)` is used for the `AF_INET` address family, where *host* is a string representing either a hostname in internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.
  - For IPv4 addresses, two special forms are accepted instead of a host address: `''` represents `INADDR_ANY`, which is used to bind to all interfaces, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.
- For `AF_INET6` address family, a four-tuple `(host, port, flowinfo, scope_id)` is used, where *flowinfo* and *scope_id* represent the `sin6_flowinfo` and `sin6_scope_id` members in `struct sockaddr_in6` in C. For `socket` module methods, *flowinfo* and *scope_id* can be omitted just for backward compatibility. Note, however, omission of *scope_id* can cause problems in manipulating scoped IPv6 addresses.

  Changed in version 3.7: For multicast addresses (with *scope_id* meaningful) *address* may not contain `%scope_id` (or `zone id`) part. This information is superfluous and may be safely omitted (recommended).

- `AF_NETLINK` sockets are represented as pairs `(pid, groups)`.