**Learn Python Programming**

Python is a powerful general-purpose programming language. It is used in web development, data science, creating software prototypes, and so on. Fortunately for beginners, Python has simple easy-to-use syntax. This makes Python an excellent language to learn to program for beginners.

Python is a cross-platform programming language, which means that it can run on multiple platforms like Windows, macOS, Linux, and has even been ported to the Java and .NET virtual machines. It is free and open-source.

Even though most of today's Linux and Mac have Python pre-installed in it, the version might be out-of-date. So, it is always a good idea to install the most current version.

# Python Keywords and Identifiers

#### **In this tutorial, you will learn about keywords (reserved words in Python) and identifiers (names given to variables, functions, etc.).**

## Python Keywords

Keywords are the reserved words in Python.

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

In Python, keywords are case sensitive.

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

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

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

## Python Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another.

### Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase **(a to z)** or uppercase **(A to Z)** or digits **(0 to 9)** or an underscore \_. Names like myClass, var\_1 and print\_this\_to\_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is a valid name.
3. Keywords cannot be used as identifiers.

global = 1

**Output**

File "<interactive input>", line 1

global = 1

^

SyntaxError: invalid syntax

1. We cannot use special symbols like **!**, **@**, **#**, **$**, **%** etc. in our identifier.

a@ = 0

**Output**

File "<interactive input>", line 1

a@ = 0

^

SyntaxError: invalid syntax

1. An identifier can be of any length.

Things to Remember

Python is a case-sensitive language. This means, Variable and variable are not the same.

Always give the identifiers a name that makes sense. While c = 10 is a valid name, writing count = 10 would make more sense, and it would be easier to figure out what it represents when you look at your code after a long gap.

Multiple words can be separated using an underscore, like this\_is\_a\_long\_variable.

# Python Statement, Indentation and Comments

#### **In this tutorial, you will learn about Python statements, why indentation is important and use of comments in programming.**

## Python Statement

Instructions that a Python interpreter can execute are called statements. For example, a = 1 is an assignment statement. if statement, for statement, while statement, etc. are other kinds of statements which will be discussed later.

### Multi-line statement

In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

a = 1 + 2 + 3 + \

4 + 5 + 6 + \

7 + 8 + 9

This is an explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ], and braces { }. For instance, we can implement the above multi-line statement as:

a = (1 + 2 + 3 +

4 + 5 + 6 +

7 + 8 + 9)

Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

colors = ['red',

'blue',

'green']

We can also put multiple statements in a single line using semicolons, as follows:

a = 1; b = 2; c = 3

## Python Indentation

Most of the programming languages like C, C++, and Java use braces { } to define a block of code. Python, however, uses indentation.

A code block (body of a [function](https://www.programiz.com/python-programming/function), [loop](https://www.programiz.com/python-programming/for-loop), etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally, four whitespaces are used for indentation and are preferred over tabs. Here is an example.

for i in range(1,11):

print(i)

if i == 5:

break

The enforcement of indentation in Python makes the code look neat and clean. This results in Python programs that look similar and consistent.

Indentation can be ignored in line continuation, but it's always a good idea to indent. It makes the code more readable. For example:

if True:

print('Hello')

a = 5

and

if True: print('Hello'); a = 5

both are valid and do the same thing, but the former style is clearer.

Incorrect indentation will result in IndentationError.

## Python Comments

Comments are very important while writing a program. They describe what is going on inside a program, so that a person looking at the source code does not have a hard time figuring it out.

You might forget the key details of the program you just wrote in a month's time. So taking the time to explain these concepts in the form of comments is always fruitful.

In Python, we use the hash (**#**) symbol to start writing a comment.

It extends up to the newline character. Comments are for programmers to better understand a program. Python Interpreter ignores comments.

#This is a comment

#print out Hello

print('Hello')

### Multi-line comments

We can have comments that extend up to multiple lines. One way is to use the hash(**#**) symbol at the beginning of each line. For example:

#This is a long comment

#and it extends

#to multiple lines

Another way of doing this is to use triple quotes, either ''' or """.

These triple quotes are generally used for multi-line strings. But they can be used as a multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

"""This is also a

perfect example of

multi-line comments"""

# Python Variables, Constants and Literals

#### **In this tutorial, you will learn about Python variables, constants, literals and their use cases.**

## Python Variables

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

number = 10

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

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

number = 10

number = 1.1

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

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

## Assigning values to Variables in Python

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

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

website = "apple.com"

print(website)

#another example

print('pratik','geeksforgeeks', sep='@')

**Output**

apple.com

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

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

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

website = "apple.com"

print(website)

# assigning a new variable to website

website = "programiz.com"

print(website)

**Output**

apple.com

programiz.com

In the above program, we have assigned apple.com to the website variable initially. Then, the value is changed to programiz.com.

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

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

print (a)

print (b)

print (c)

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

x = y = z = "same"

print (x)

print (y)

print (z)

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

## Constants

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

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

## Assigning value to constant in Python

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

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

Create a **constant.py**:

PI = 3.14

GRAVITY = 9.8

Create a **main.py**:

import constant

print(constant.PI)

print(constant.GRAVITY)

**Output**

3.14

9.8

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

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

Rules and Naming Convention for Variables and constants

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

CapWords

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

current\_salary

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

TEMP

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

Literals

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

## Numeric Literals

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

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

a = 0b1010 #Binary Literals

b = 100 #Decimal Literal

c = 0o310 #Octal Literal

d = 0x12c #Hexadecimal Literal

#Float Literal

float\_1 = 10.5

float\_2 = 1.5e2

#Complex Literal

x = 3.14j

print(a, b, c, d)

print(float\_1, float\_2)

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

**Output**

10 100 200 300

10.5 150.0

3.14j 3.14 0.0

In the above program,

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

## String literals

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

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

strings = "This is Python"

char = "C"

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

unicode = u"\u00dcnic\u00f6de"

raw\_str = r"raw \n string"

print(strings)

print(char)

print(multiline\_str)

print(unicode)

print(raw\_str)

**Output**

This is Python

C

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

Ünicöde

raw \n string

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

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

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

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

## Boolean literals

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

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

x = (1 == True)

y = (1 == False)

a = True + 4

b = False + 10

print("x is", x)

print("y is", y)

print("a:", a)

print("b:", b)

**Output**

x is True

y is False

a: 5

b: 10

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

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

## Special literals

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

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

drink = "Available"

food = None

def menu(x):

if x == drink:

print(drink)

else:

print(food)

menu(drink)

menu(food)

**Output**

Available

None

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

## Literal Collections

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

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

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

numbers = (1, 2, 3) #tuple

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

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

print(fruits)

print(numbers)

print(alphabets)

print(vowels)

**Output**

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

(1, 2, 3)

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

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

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

# Python Data Types

#### **In this tutorial, you will learn about different data types you can use in Python.**

## Data types in Python

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

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

## Python Numbers

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

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

a = 5

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

a = 2.0

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

a = 1+2j

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

**Output**

5 is of type <class 'int'>

2.0 is of type <class 'float'>

(1+2j) is complex number? True

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

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

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

>>> a = 1234567890123456789

>>> a

1234567890123456789

>>> b = 0.1234567890123456789

>>> b

0.12345678901234568

>>> c = 1+2j

>>> c

(1+2j)

Notice that the float variable b got truncated.

## Python List

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

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

a = [1, 2.2, 'python']

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

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

# a[2] = 15

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

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

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

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

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

**Output**

a[2] = 15

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

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

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

a = [1, 2, 3]

a[2] = 4

print(a)

**Output**

[1, 2, 4]

# Python List

#### **In this article, we'll learn everything about Python lists, how they are created, slicing of a list, adding or removing elements from them and so on.**

Python offers a range of compound data types often referred to as sequences. List is one of the most frequently used and very versatile data types used in Python.

## How to create a list?

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

# empty list

my\_list = []

# list of integers

my\_list = [1, 2, 3]

# list with mixed data types

my\_list = [1, "Hello", 3.4]

A list can also have another list as an item. This is called a nested list.

# nested list

my\_list = ["mouse", [8, 4, 6], ['a']]

## How to access elements from a list?

There are various ways in which we can access the elements of a list.

### List Index

We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.

Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

Nested lists are accessed using nested indexing.

# List indexing

my\_list = ['p', 'r', 'o', 'b', 'e']

# Output: p

print(my\_list[0])

# Output: o

print(my\_list[2])

# Output: e

print(my\_list[4])

# Nested List

n\_list = ["Happy", [2, 0, 1, 5]]

# Nested indexing

print(n\_list[0][1])

print(n\_list[1][3])

# Error! Only integer can be used for indexing

print(my\_list[4.0])

**Output**

p

o

e

a

5

Traceback (most recent call last):

File "<string>", line 21, in <module>

TypeError: list indices must be integers or slices, not float

### Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

# Negative indexing in lists

my\_list = ['p','r','o','b','e']

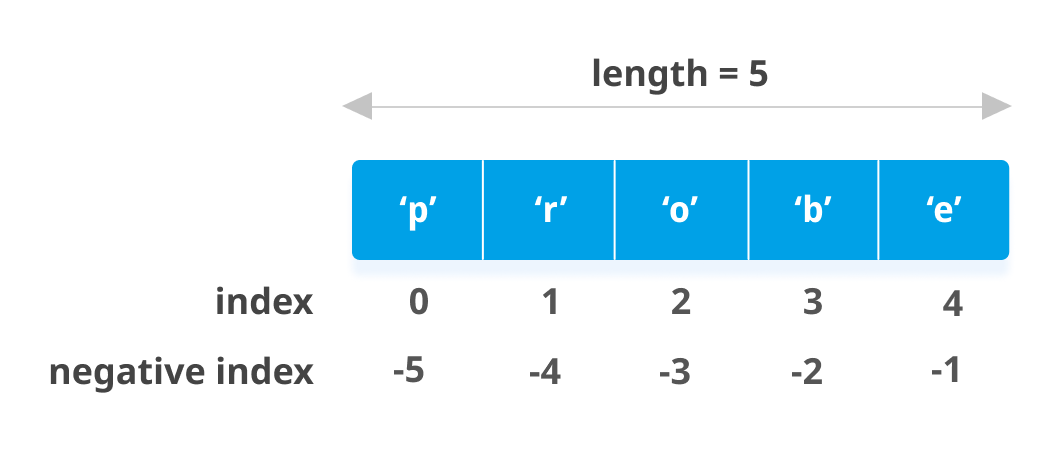
print(my\_list[-1])

print(my\_list[-5])

When we run the above program, we will get the following output:

e

p

List indexing in Python

## How to slice lists in Python?

We can access a range of items in a list by using the slicing operator :(colon).

# List slicing in Python

my\_list = ['p','r','o','g','r','a','m','i','z']

# elements 3rd to 5th

print(my\_list[2:5])

# elements beginning to 4th

print(my\_list[:-5])

# elements 6th to end

print(my\_list[5:])

# elements beginning to end

print(my\_list[:])

**Output**

['o', 'g', 'r']

['p', 'r', 'o', 'g']

['a', 'm', 'i', 'z']

['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need two indices that will slice that portion from the list.

Element Slicing from a list in Python

## How to change or add elements to a list?

Lists are mutable, meaning their elements can be changed unlike [string](https://www.programiz.com/python-programming/string) or [tuple](https://www.programiz.com/python-programming/tuple).

We can use the assignment operator (=) to change an item or a range of items.

# Correcting mistake values in a list

odd = [2, 4, 6, 8]

# change the 1st item

odd[0] = 1

print(odd)

# change 2nd to 4th items

odd[1:4] = [3, 5, 7]

print(odd)

**Output**

[1, 4, 6, 8]

[1, 3, 5, 7]

We can add one item to a list using the append() method or add several items using extend() method.

# Appending and Extending lists in Python

odd = [1, 3, 5]

odd.append(7)

print(odd)

odd.extend([9, 11, 13])

print(odd)

**Output**

[1, 3, 5, 7]

[1, 3, 5, 7, 9, 11, 13]

We can also use + operator to combine two lists. This is also called concatenation.

The \* operator repeats a list for the given number of times.

# Concatenating and repeating lists

odd = [1, 3, 5]

print(odd + [9, 7, 5])

print(["re"] \* 3)

**Output**

[1, 3, 5, 9, 7, 5]

['re', 're', 're']

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

# Demonstration of list insert() method

odd = [1, 9]

odd.insert(1,3)

print(odd)

odd[2:2] = [5, 7]

print(odd)

**Output**

[1, 3, 9]

[1, 3, 5, 7, 9]

## How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword del. It can even delete the list entirely.

# Deleting list items

my\_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# delete one item

del my\_list[2]

print(my\_list)

# delete multiple items

del my\_list[1:5]

print(my\_list)

# delete entire list

del my\_list

# Error: List not defined

print(my\_list)

**Output**

['p', 'r', 'b', 'l', 'e', 'm']

['p', 'm']

Traceback (most recent call last):

File "<string>", line 18, in <module>

NameError: name 'my\_list' is not defined

We can use remove() method to remove the given item or pop() method to remove an item at the given index.

The pop() method removes and returns the last item if the index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the clear() method to empty a list.

my\_list = ['p','r','o','b','l','e','m']

my\_list.remove('p')

# Output: ['r', 'o', 'b', 'l', 'e', 'm']

print(my\_list)

# Output: 'o'

print(my\_list.pop(1))

# Output: ['r', 'b', 'l', 'e', 'm']

print(my\_list)

# Output: 'm'

print(my\_list.pop())

# Output: ['r', 'b', 'l', 'e']

print(my\_list)

my\_list.clear()

# Output: []

print(my\_list)

**Output**

['r', 'o', 'b', 'l', 'e', 'm']

o

['r', 'b', 'l', 'e', 'm']

m

['r', 'b', 'l', 'e']

[]

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

>>> my\_list = ['p','r','o','b','l','e','m']

>>> my\_list[2:3] = []

>>> my\_list

['p', 'r', 'b', 'l', 'e', 'm']

>>> my\_list[2:5] = []

>>> my\_list

['p', 'r', 'm']

## Python List Methods

Methods that are available with list objects in Python programming are tabulated below.

They are accessed as list.method(). Some of the methods have already been used above.

|  |
| --- |
| [Python List Methods](https://www.programiz.com/python-programming/methods/list) |
| [**append() -**](https://www.programiz.com/python-programming/methods/list/append) [Add an element to the end of the list](https://www.programiz.com/python-programming/methods/list/append) |
| [**extend()**](https://www.programiz.com/python-programming/methods/list/extend) [-](https://www.programiz.com/python-programming/methods/list/extend) [Add all elements of a list to the another list](https://www.programiz.com/python-programming/methods/list/extend) |
| [**insert()**](https://www.programiz.com/python-programming/methods/list/insert) [-](https://www.programiz.com/python-programming/methods/list/insert) [Insert an item at the defined index](https://www.programiz.com/python-programming/methods/list/insert) |
| [**remove()**](https://www.programiz.com/python-programming/methods/list/remove) [-](https://www.programiz.com/python-programming/methods/list/remove) [Removes an item from the list](https://www.programiz.com/python-programming/methods/list/remove) |
| [**pop()**](https://www.programiz.com/python-programming/methods/list/pop) [-](https://www.programiz.com/python-programming/methods/list/pop) [Removes and returns an element at the given index](https://www.programiz.com/python-programming/methods/list/pop) |
| [**clear()**](https://www.programiz.com/python-programming/methods/list/clear) [- Removes all items from the list](https://www.programiz.com/python-programming/methods/list/clear) |
| [**index()**](https://www.programiz.com/python-programming/methods/list/index) [- Returns the index of the first matched item](https://www.programiz.com/python-programming/methods/list/index) |
| [**count()**](https://www.programiz.com/python-programming/methods/list/count) [- Returns the count of the number of items passed as an argument](https://www.programiz.com/python-programming/methods/list/count) |
| [**sort()**](https://www.programiz.com/python-programming/methods/list/sort) [- Sort items in a list in ascending order](https://www.programiz.com/python-programming/methods/list/sort) |
| [**reverse()**](https://www.programiz.com/python-programming/methods/list/reverse) [- Reverse the order of items in the list](https://www.programiz.com/python-programming/methods/list/reverse) |
| [**copy()**](https://www.programiz.com/python-programming/methods/list/copy) [- Returns a shallow copy of the list](https://www.programiz.com/python-programming/methods/list/copy) |

Some examples of Python list methods:

# Python list methods

my\_list = [3, 8, 1, 6, 0, 8, 4]

# Output: 1

print(my\_list.index(8))

# Output: 2

print(my\_list.count(8))

my\_list.sort()

# Output: [0, 1, 3, 4, 6, 8, 8]

print(my\_list)

my\_list.reverse()

# Output: [8, 8, 6, 4, 3, 1, 0]

print(my\_list)

**Output**

1

2

[0, 1, 3, 4, 6, 8, 8]

[8, 8, 6, 4, 3, 1, 0]

## List Comprehension: Elegant way to create new List

List comprehension is an elegant and concise way to create a new list from an existing list in Python.

A list comprehension consists of an expression followed by [for statement](https://www.programiz.com/python-programming/for-loop) inside square brackets.

Here is an example to make a list with each item being increasing power of 2.

pow2 = [2 \*\* x for x in range(10)]

print(pow2)

**Output**

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

This code is equivalent to:

pow2 = []

for x in range(10):

pow2.append(2 \*\* x)

A list comprehension can optionally contain more for or [if statements](https://www.programiz.com/python-programming/if-elif-else). An optional if statement can filter out items for the new list. Here are some examples.

>>> pow2 = [2 \*\* x for x in range(10) if x > 5]

>>> pow2

[64, 128, 256, 512]

>>> odd = [x for x in range(20) if x % 2 == 1]

>>> odd

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

>>> [x+y for x in ['Python ','C '] for y in ['Language','Programming']]

['Python Language', 'Python Programming', 'C Language', 'C Programming']

## Other List Operations in Python

### List Membership Test

We can test if an item exists in a list or not, using the keyword in.

my\_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# Output: True

print('p' in my\_list)

# Output: False

print('a' in my\_list)

# Output: True

print('c' not in my\_list)

**Output**

True

False

True

### Iterating Through a List

Using a for loop we can iterate through each item in a list.

for fruit in ['apple','banana','mango']:

print("I like",fruit)

**Output**

I like apple

I like banana

I like mango

# Python List append()

#### **The append() method adds an item to the end of the list.**

The syntax of the append() method is:

list.append(item)

## append() Parameters

The method takes a single argument

* item - an item to be added at the end of the list

The item can be numbers, strings, dictionaries, another list, and so on.

## Return Value from append()

The method doesn't return any value (returns None).

## Example 1: Adding Element to a List

# animals list

animals = ['cat', 'dog', 'rabbit']

# 'guinea pig' is appended to the animals list

animals.append('guinea pig')

# Updated animals list

print('Updated animals list: ', animals)

**Output**

Updated animals list: ['cat', 'dog', 'rabbit', 'guinea pig']

## Example 2: Adding List to a List

# animals list

animals = ['cat', 'dog', 'rabbit']

# list of wild animals

wild\_animals = ['tiger', 'fox']

# appending wild\_animals list to the animals list

animals.append(wild\_animals)

print('Updated animals list: ', animals)

**Output**

Updated animals list: ['cat', 'dog', 'rabbit', ['tiger', 'fox']]

It is important to notice that, a single item (wild\_animals list) is added to the animals list in the above program.

If you need to add items of a list to another list (rather than the list itself), use the [extend() method](https://www.programiz.com/python-programming/methods/list/extend).

# Python List extend()

#### **The extend() method adds all the elements of an iterable (list, tuple, string etc.) to the end of the list.**

The syntax of the extend() method is:

list1.extend(iterable)

Here, all the elements of iterable are added to the end of list1.

## extend() Parameters

As mentioned, the extend() method takes an iterable such as list, tuple, string etc.

## Return Value from extend()

The extend() method modifies the original list. It doesn't return any value.

## Example 1: Using extend() Method

# language list

language = ['French', 'English']

# another list of language

language1 = ['Spanish', 'Portuguese']

# appending language1 elements to language

language.extend(language1)

print('Language List:', language)

**Output**

Language List: ['French', 'English', 'Spanish', 'Portuguese']

## Example 2: Add Elements of Tuple and Set to List

# language list

language = ['French']

# language tuple

language\_tuple = ('Spanish', 'Portuguese')

# language set

language\_set = {'Chinese', 'Japanese'}

# appending language\_tuple elements to language

language.extend(language\_tuple)

print('New Language List:', language)

# appending language\_set elements to language

language.extend(language\_set)

print('Newer Language List:', language)

**Output**

New Language List: ['French', 'Spanish', 'Portuguese']

Newer Language List: ['French', 'Spanish', 'Portuguese', 'Japanese', 'Chinese']

### Other Ways to Extend a List

You can also append all elements of an iterable to the list using:

**1. the + operator**

a = [1, 2]

b = [3, 4]

a += b # a = a + b

# Output: [1, 2, 3, 4]

print('a =', a)

**Output**

a = [1, 2, 3, 4]

**2. the list slicing syntax**

a = [1, 2]

b = [3, 4]

a[len(a):] = b

# Output: [1, 2, 3, 4]

print('a =', a)

**Output**

a = [1, 2, 3, 4]

### Python extend() Vs append()

If you need to add an element to the end of a list, you can use the append() method.

a1 = [1, 2]

a2 = [1, 2]

b = (3, 4)

# a1 = [1, 2, 3, 4]

a1.extend(b)

print(a1)

# a2 = [1, 2, (3, 4)]

a2.append(b)

print(a2)

**Output**

[1, 2, 3, 4]

[1, 2, (3, 4)]

# Python List insert()

#### **The list insert() method inserts an element to the list at the specified index.**

The syntax of the insert() method is

list.insert(i, elem)

Here, elem is inserted to the list at the ith index. All the elements after elem are shifted to the right.

## insert() Parameters

The insert() method takes two parameters:

* **index** - the index where the element needs to be inserted
* **element** - this is the element to be inserted in the list

**Notes:**

* If index is 0, the element is inserted at the beginning of the list.
* If index is 3, the element is inserted after the 3rd element. Its position will be 4th.

## Return Value from insert()

The insert() method doesn't return anything; returns None. It only updates the current list.

## Example 1: Inserting an Element to the List

# vowel list

vowel = ['a', 'e', 'i', 'u']

# 'o' is inserted at index 3

# the position of 'o' will be 4th

vowel.insert(3, 'o')

print('Updated List:', vowel)

**Output**

Updated List: ['a', 'e', 'i', 'o', 'u']

## Example 2: Inserting a Tuple (as an Element) to the List

mixed\_list = [{1, 2}, [5, 6, 7]]

# number tuple

number\_tuple = (3, 4)

# inserting a tuple to the list

mixed\_list.insert(1, number\_tuple)

print('Updated List:', mixed\_list)

**Output**

Updated List: [{1, 2}, (3, 4), [5, 6, 7]]

# Python List remove()

#### **The remove() method removes the first matching element (which is passed as an argument) from the list.**

The syntax of the remove() method is:

list.remove(element)

## remove() Parameters

* The remove() method takes a single element as an argument and removes it from the list.
* If the element doesn't exist, it throws **ValueError: list.remove(x): x not in list** exception.

## Return Value from remove()

The remove() doesn't return any value (returns None).

## Example 1: Remove element from the list

# animals list

animals = ['cat', 'dog', 'rabbit', 'guinea pig']

# 'rabbit' is removed

animals.remove('rabbit')

# Updated animals List

print('Updated animals list: ', animals)

**Output**

Updated animals list: ['cat', 'dog', 'guinea pig']

## Example 2: remove() method on a list having duplicate elements

If a list contains duplicate elements, the remove() method only removes the first matching element.

# animals list

animals = ['cat', 'dog', 'dog', 'guinea pig', 'dog']

# 'dog' is removed

animals.remove('dog')

# Updated animals list

print('Updated animals list: ', animals)

**Output**

Updated animals list: ['cat', 'dog', 'guinea pig', 'dog']

Here, only the first occurrence of element 'dog' is removed from the list.

## Example 3: Deleting element that doesn't exist

# animals list

animals = ['cat', 'dog', 'rabbit', 'guinea pig']

# Deleting 'fish' element

animals.remove('fish')

# Updated animals List

print('Updated animals list: ', animals)

**Output**

Traceback (most recent call last):

File ".. .. ..", line 5, in <module>

animal.remove('fish')

ValueError: list.remove(x): x not in list

Here, we are getting an error because the animals list doesn't contain 'fish'.

* If you need to delete elements based on the index (like the fourth element), you can use the [pop() method](https://www.programiz.com/python-programming/methods/list/pop).
* Also, you can use the [Python del statement](https://www.programiz.com/python-programming/del) to remove items from the list.

# Python List pop()

#### **The pop() method removes the item at the given index from the list and returns the removed item.**

The syntax of the pop() method is:

list.pop(index)

## pop() parameters

* The pop() method takes a single argument (index).
* The argument passed to the method is optional. If not passed, the default index **-1** is passed as an argument (index of the last item).
* If the index passed to the method is not in range, it throws **IndexError: pop index out of range**exception.

## Return Value from pop()

The pop() method returns the item present at the given index. This item is also removed from the list.

## Example 1: Pop item at the given index from the list

# programming languages list

languages = ['Python', 'Java', 'C++', 'French', 'C']

# remove and return the 4th item

return\_value = languages.pop(3)

print('Return Value:', return\_value)

# Updated List

print('Updated List:', languages)

**Output**

Return Value: French

Updated List: ['Python', 'Java', 'C++', 'C']

**Note:** Index in Python starts from 0, not 1.

If you need to pop the 4th element, you need to pass **3** to the pop() method.

## Example 2: pop() without an index, and for negative indices

# programming languages list

languages = ['Python', 'Java', 'C++', 'Ruby', 'C']

# remove and return the last item

print('When index is not passed:')

print('Return Value:', languages.pop())

print('Updated List:', languages)

# remove and return the last item

print('\nWhen -1 is passed:')

print('Return Value:', languages.pop(-1))

print('Updated List:', languages)

# remove and return the third last item

print('\nWhen -3 is passed:')

print('Return Value:', languages.pop(-3))

print('Updated List:', languages)

**Output**

When index is not passed:

Return Value: C

Updated List: ['Python', 'Java', 'C++', 'Ruby']

When -1 is passed:

Return Value: Ruby

Updated List: ['Python', 'Java', 'C++']

When -3 is passed:

Return Value: Python

Updated List: ['Java', 'C++']

If you need to remove the given item from the list, you can to use the [remove() method](https://www.programiz.com/python-programming/methods/list/remove).

And, you can use the del statement to [remove an item or slices from the list](https://www.programiz.com/python-programming/del#items-list).

# Python List clear()

#### **The clear() method removes all items from the list.**

The syntax of clear() method is:

list.clear()

## clear() Parameters

The clear() method doesn't take any parameters.

## Return Value from clear()

The clear() method only empties the given [list](https://www.programiz.com/python-programming/list). It doesn't return any value.

## Example 1: Working of clear() method

# Defining a list

list = [{1, 2}, ('a'), ['1.1', '2.2']]

# clearing the list

list.clear()

print('List:', list)

**Output**

List: []

**Note:** If you are using Python 2 or Python 3.2 and below, you cannot use the clear() method. You can use the del operator instead.

## Example 2: Emptying the List Using del

# Defining a list

list = [{1, 2}, ('a'), ['1.1', '2.2']]

# clearing the list

del list[:]

print('List:', list)

**Output**

List: []

# Python List index()

#### **The index() method returns the index of the specified element in the list.**

The syntax of the list index() method is:

list.index(element, start, end)

## list index() parameters

The list index() method can take a maximum of three arguments:

* **element** - the element to be searched
* **start** (optional) - start searching from this index
* **end** (optional) - search the element up to this index

## Return Value from List index()

* The index() method returns the index of the given element in the list.
* If the element is not found, a ValueError exception is raised.

**Note:** The index() method only returns the first occurrence of the matching element.

## Example 1: Find the index of the element

# vowels list

vowels = ['a', 'e', 'i', 'o', 'i', 'u']

# index of 'e' in vowels

index = vowels.index('e')

print('The index of e:', index)

# element 'i' is searched

# index of the first 'i' is returned

index = vowels.index('i')

print('The index of i:', index)

**Output**

The index of e: 1

The index of i: 2

## Example 2: Index of the Element not Present in the List

# vowels list

vowels = ['a', 'e', 'i', 'o', 'u']

# index of'p' is vowels

index = vowels.index('p')

print('The index of p:', index)

**Output**

ValueError: 'p' is not in list

## Example 3: Working of index() With Start and End Parameters

# alphabets list

alphabets = ['a', 'e', 'i', 'o', 'g', 'l', 'i', 'u']

# index of 'i' in alphabets

index = alphabets.index('e') # 2

print('The index of e:', index)

# 'i' after the 4th index is searched

index = alphabets.index('i', 4) # 6

print('The index of i:', index)

# 'i' between 3rd and 5th index is searched

index = alphabets.index('i', 3, 5) # Error!

print('The index of i:', index)

**Output**

The index of e: 1

The index of i: 6

Traceback (most recent call last):

File "\*lt;string>", line 13, in

ValueError: 'i' is not in list

# Python List count()

#### **The count() method returns the number of times the specified element appears in the list.**

The syntax of the count() method is:

list.count(element)

## count() Parameters

The count() method takes a single argument:

* **element** - the element to be counted

## Return value from count()

The count() method returns the number of times element appears in the list.

## Example 1: Use of count()

# vowels list

vowels = ['a', 'e', 'i', 'o', 'i', 'u']

# count element 'i'

count = vowels.count('i')

# print count

print('The count of i is:', count)

# count element 'p'

count = vowels.count('p')

# print count

print('The count of p is:', count)

**Output**

The count of i is: 2

The count of p is: 0

## Example 2: Count Tuple and List Elements Inside List

# random list

random = ['a', ('a', 'b'), ('a', 'b'), [3, 4]]

# count element ('a', 'b')

count = random.count(('a', 'b'))

# print count

print("The count of ('a', 'b') is:", count)

# count element [3, 4]

count = random.count([3, 4])

# print count

print("The count of [3, 4] is:", count)

**Output**

The count of ('a', 'b') is: 2

The count of [3, 4] is: 1

# Python List sort()

#### **The sort() method sorts the elements of a given list in a specific ascending or descending order.**

The syntax of the sort() method is:

list.sort(key=..., reverse=...)

Alternatively, you can also use Python's built-in [sorted()](https://www.programiz.com/python-programming/methods/built-in/sorted) function for the same purpose.

sorted(list, key=..., reverse=...)

**Note:** The simplest difference between sort() and sorted() is: sort() changes the list directly and doesn't return any value, while sorted() doesn't change the list and returns the sorted list.

## sort() Parameters

By default, sort() doesn't require any extra parameters. However, it has two optional parameters:

* **reverse** - If True, the sorted list is reversed (or sorted in Descending order)
* **key** - function that serves as a key for the sort comparison

## Return value from sort()

The sort() method doesn't return any value. Rather, it changes the original list.

If you want a function to return the sorted list rather than change the original list, use sorted().

## Example 1: Sort a given list

# vowels list

vowels = ['e', 'a', 'u', 'o', 'i']

# sort the vowels

vowels.sort()

# print vowels

print('Sorted list:', vowels)

**Output**

Sorted list: ['a', 'e', 'i', 'o', 'u']

## Sort in Descending order

The sort() method accepts a reverse parameter as an optional argument.

Setting reverse = True sorts the list in the descending order.

list.sort(reverse=True)

Alternately for sorted(), you can use the following code.

sorted(list, reverse=True)

## Example 2: Sort the list in Descending order

# vowels list

vowels = ['e', 'a', 'u', 'o', 'i']

# sort the vowels

vowels.sort(reverse=True)

# print vowels

print('Sorted list (in Descending):', vowels)

**Output**

Sorted list (in Descending): ['u', 'o', 'i', 'e', 'a']

## Sort with custom function using key

If you want your own implementation for sorting, the sort() method also accepts a key function as an optional parameter.

Based on the results of the key function, you can sort the given list.

list.sort(key=len)

Alternatively for sorted:

sorted(list, key=len)

Here, len is the Python's in-built function to count the length of an element.

The list is sorted based on the length of each element, from lowest count to highest.

We know that a tuple is sorted using its first parameter by default. Let's look at how to customize the sort() method to sort using the second element.

## Example 3: Sort the list using key

# take second element for sort

def takeSecond(elem):

return elem[1]

# random list

random = [(2, 2), (3, 4), (4, 1), (1, 3)]

# sort list with key

random.sort(key=takeSecond)

# print list

print('Sorted list:', random)

**Output**

Sorted list: [(4, 1), (2, 2), (1, 3), (3, 4)]

Let's take another example. Suppose we have a list of information about the employees of an office where each element is a dictionary.

We can sort the list in the following way:

# sorting using custom key

employees = [

{'Name': 'Alan Turing', 'age': 25, 'salary': 10000},

{'Name': 'Sharon Lin', 'age': 30, 'salary': 8000},

{'Name': 'John Hopkins', 'age': 18, 'salary': 1000},

{'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000},

]

# custom functions to get employee info

def get\_name(employee):

return employee.get('Name')

def get\_age(employee):

return employee.get('age')

def get\_salary(employee):

return employee.get('salary')

# sort by name (Ascending order)

employees.sort(key=get\_name)

print(employees, end='\n\n')

# sort by Age (Ascending order)

employees.sort(key=get\_age)

print(employees, end='\n\n')

# sort by salary (Descending order)

employees.sort(key=get\_salary, reverse=True)

print(employees, end='\n\n')

**Output**

[{'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'John Hopkins', 'age': 18, 'salary': 1000}, {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}]

[{'Name': 'John Hopkins', 'age': 18, 'salary': 1000}, {'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}, {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}]

[{'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, {'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}, {'Name': 'John Hopkins', 'age': 18, 'salary': 1000}]

Here, for the first case, our custom function returns the name of each employee. Since the name is a string, Python by default sorts it using the alphabetical order.

For the second case, age (int) is returned and is sorted in ascending order.

For the third case, the function returns the salary (int), and is sorted in the descending order using reverse = True.

It is a good practice to use the lambda function when the function can be summarized in one line. So, we can also write the above program as:

# sorting using custom key

employees = [

{'Name': 'Alan Turing', 'age': 25, 'salary': 10000},

{'Name': 'Sharon Lin', 'age': 30, 'salary': 8000},

{'Name': 'John Hopkins', 'age': 18, 'salary': 1000},

{'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000},

]

# sort by name (Ascending order)

employees.sort(key=lambda x: x.get('Name'))

print(employees, end='\n\n')

# sort by Age (Ascending order)

employees.sort(key=lambda x: x.get('age'))

print(employees, end='\n\n')

# sort by salary (Descending order)

employees.sort(key=lambda x: x.get('salary'), reverse=True)

print(employees, end='\n\n')

**Output**

[{'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'John Hopkins', 'age': 18, 'salary': 1000}, {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}]

[{'Name': 'John Hopkins', 'age': 18, 'salary': 1000}, {'Name': 'Alan Turing', 'age': 25, 'salary': 10000}, {'Name': 'Sharon Lin', 'age': 30, 'salary': 8000}, {'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}]

[{'Name': 'Mikhail Tal', 'age': 40, 'salary': 15000}, {'Name': 'Alan Turing', 'age': 25, 'sal

# Python List reverse()

#### **The reverse() method reverses the elements of the list.**

The syntax of the reverse() method is:

list.reverse()

## reverse() parameter

The reverse() method doesn't take any arguments.

## Return Value from reverse()

The reverse() method doesn't return any value. It updates the existing list.

## Example 1: Reverse a List

# Operating System List

systems = ['Windows', 'macOS', 'Linux']

print('Original List:', systems)

# List Reverse

systems.reverse()

# updated list

print('Updated List:', systems)

**Output**

Original List: ['Windows', 'macOS', 'Linux']

Updated List: ['Linux', 'macOS', 'Windows']

There are other several ways to reverse a list.

## Example 2: Reverse a List Using Slicing Operator

# Operating System List

systems = ['Windows', 'macOS', 'Linux']

print('Original List:', systems)

# Reversing a list

#Syntax: reversed\_list = systems[start:stop:step]

reversed\_list = systems[::-1]

# updated list

print('Updated List:', reversed\_list)

**Output**

Original List: ['Windows', 'macOS', 'Linux']

Updated List: ['Linux', 'macOS', 'Windows']

## Example 3: Accessing Elements in Reversed Order

If you need to access individual elements of a list in the reverse order, it's better to use reversed() function.

# Operating System List

systems = ['Windows', 'macOS', 'Linux']

# Printing Elements in Reversed Order

for o in reversed(systems):

print(o)

**Output**

Linux

macOS

Windows

# Python List copy()

#### **The copy() method returns a shallow copy of the list.**

A [list](https://www.programiz.com/python-programming/list) can be copied using the **=** operator. For example,

old\_list = [1, 2, 3]

​new\_list = old\_list

The problem with copying lists in this way is that if you modify new\_list, old\_list is also modified. It is because the new list is referencing or pointing to the same old\_list object.

old\_list = [1, 2, 3]

new\_list = old\_list

# add an element to list

new\_list.append('a')

print('New List:', new\_list)

print('Old List:', old\_list)

**Output**

Old List: [1, 2, 3, 'a']

New List: [1, 2, 3, 'a']

However, if you need the original list unchanged when the new list is modified, you can use the copy() method.  
  
**Related tutorial:** [Python Shallow Copy Vs Deep Copy](https://www.programiz.com/python-programming/shallow-deep-copy)

The syntax of the copy() method is:

new\_list = list.copy()

## copy() parameters

The copy() method doesn't take any parameters.

## Return Value from copy()

The copy() method returns a new list. It doesn't modify the original list.

## Example 1: Copying a List

# mixed list

my\_list = ['cat', 0, 6.7]

# copying a list

new\_list = my\_list.copy()

print('Copied List:', new\_list)

**Output**

Copied List: ['cat', 0, 6.7]

If you modify the new\_list in the above example, my\_list will not be modified.

## Example 2: Copy List Using Slicing Syntax

# shallow copy using the slicing syntax

# mixed list

list = ['cat', 0, 6.7]

# copying a list using slicing

new\_list = list[:]

# Adding an element to the new list

new\_list.append('dog')

# Printing new and old list

print('Old List:', list)

print('New List:', new\_list)

**Output**

Old List: ['cat', 0, 6.7]

New List: ['cat', 0, 6.7, 'dog']

## Python Tuple

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

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

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

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

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

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

# t[1] = 'program'

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

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

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

# Generates error

# Tuples are immutable

t[0] = 10

**Output**

t[1] = program

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

Traceback (most recent call last):

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

t[0] = 10

TypeError: 'tuple' object does not support item assignment

## Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, [string](https://www.programiz.com/python-programming/string), etc.).

# Different types of tuples

# Empty tuple

my\_tuple = ()

print(my\_tuple)

# Tuple having integers

my\_tuple = (1, 2, 3)

print(my\_tuple)

# tuple with mixed datatypes

my\_tuple = (1, "Hello", 3.4)

print(my\_tuple)

# nested tuple

my\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

print(my\_tuple)

**Output**

()

(1, 2, 3)

(1, 'Hello', 3.4)

('mouse', [8, 4, 6], (1, 2, 3))

A tuple can also be created without using parentheses. This is known as tuple packing.

my\_tuple = 3, 4.6, "dog"

print(my\_tuple)

# tuple unpacking is also possible

a, b, c = my\_tuple

print(a) # 3

print(b) # 4.6

print(c) # dog

**Output**

(3, 4.6, 'dog')

3

4.6

dog

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

my\_tuple = ("hello")

print(type(my\_tuple)) # <class 'str'>

# Creating a tuple having one element

my\_tuple = ("hello",)

print(type(my\_tuple)) # <class 'tuple'>

# Parentheses is optional

my\_tuple = "hello",

print(type(my\_tuple)) # <class 'tuple'>

**Output**

<class 'str'>

<class 'tuple'>

<class 'tuple'>

## Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

### 1. Indexing

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

# Accessing tuple elements using indexing

my\_tuple = ('p','e','r','m','i','t')

print(my\_tuple[0]) # 'p'

print(my\_tuple[5]) # 't'

# IndexError: list index out of range

# print(my\_tuple[6])

# Index must be an integer

# TypeError: list indices must be integers, not float

# my\_tuple[2.0]

# nested tuple

n\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index

print(n\_tuple[0][3]) # 's'

print(n\_tuple[1][1]) # 4

**Output**

p

t

s

4

### 2. Negative Indexing

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

# Negative indexing for accessing tuple elements

my\_tuple = ('p', 'e', 'r', 'm', 'i', 't')

# Output: 't'

print(my\_tuple[-1])

# Output: 'p'

print(my\_tuple[-6])

**Output**

t

p

### 3. Slicing

We can access a range of items in a tuple by using the slicing operator colon :.

# Accessing tuple elements using slicing

my\_tuple = ('p','r','o','g','r','a','m','i','z')

# elements 2nd to 4th

# Output: ('r', 'o', 'g')

print(my\_tuple[1:4])

# elements beginning to 2nd

# Output: ('p', 'r')

print(my\_tuple[:-7])

# elements 8th to end

# Output: ('i', 'z')

print(my\_tuple[7:])

# elements beginning to end

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

print(my\_tuple[:])

**Output**

('r', 'o', 'g')

('p', 'r')

('i', 'z')

('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.

Element Slicing in Python

## Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

# Changing tuple values

my\_tuple = (4, 2, 3, [6, 5])

# TypeError: 'tuple' object does not support item assignment

# my\_tuple[1] = 9

# However, item of mutable element can be changed

my\_tuple[3][0] = 9 # Output: (4, 2, 3, [9, 5])

print(my\_tuple)

# Tuples can be reassigned

my\_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

print(my\_tuple)

**Output**

(4, 2, 3, [9, 5])

('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

We can use + operator to combine two tuples. This is called **concatenation**.

We can also **repeat** the elements in a tuple for a given number of times using the \* operator.

Both + and \* operations result in a new tuple.

# Concatenation

# Output: (1, 2, 3, 4, 5, 6)

print((1, 2, 3) + (4, 5, 6))

# Repeat

# Output: ('Repeat', 'Repeat', 'Repeat')

print(("Repeat",) \* 3)

**Output**

(1, 2, 3, 4, 5, 6)

('Repeat', 'Repeat', 'Repeat')

## Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.

Deleting a tuple entirely, however, is possible using the keyword [del](https://www.programiz.com/python-programming/keyword-list#del).

# Deleting tuples

my\_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# can't delete items

# TypeError: 'tuple' object doesn't support item deletion

# del my\_tuple[3]

# Can delete an entire tuple

del my\_tuple

# NameError: name 'my\_tuple' is not defined

print(my\_tuple)

**Output**

Traceback (most recent call last):

File "<string>", line 12, in <module>

NameError: name 'my\_tuple' is not defined

## Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Some examples of Python tuple methods:

my\_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my\_tuple.count('p')) # Output: 2

print(my\_tuple.index('l')) # Output: 3

**Output**

2

3

## Other Tuple Operations

### 1. Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

# Membership test in tuple

my\_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation

print('a' in my\_tuple)

print('b' in my\_tuple)

# Not in operation

print('g' not in my\_tuple)

**Output**

True

False

True

### 2. Iterating Through a Tuple

We can use a for loop to iterate through each item in a tuple.

# Using a for loop to iterate through a tuple

for name in ('John', 'Kate'):

print("Hello", name)

**Output**

Hello John

Hello Kate

### Advantages of Tuple over List

Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

* We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
* Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
* Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
* If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

## Python Strings

#### [String](https://www.programiz.com/python-programming/string) is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """. **you will learn to create, format, modify and delete strings in Python. Also, you will be introduced to various string operations and functions.**

s = "This is a string"

print(s)

s = '''A multiline

string'''

print(s)

**Output**

This is a string

A multiline

string

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

s = 'Hello world!'

# s[4] = 'o'

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

# s[6:11] = 'world'

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

# Generates error

# Strings are immutable in Python

s[5] ='d'

**Output**

s[4] = o

s[6:11] = world

Traceback (most recent call last):

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

TypeError: 'str' object does not support item assignment

## What is String in Python?

A string is a sequence of characters.

A character is simply a symbol. For example, the English language has 26 characters.

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.

## How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

# defining strings in Python

# all of the following are equivalent

my\_string = 'Hello'

print(my\_string)

my\_string = "Hello"

print(my\_string)

my\_string = '''Hello'''

print(my\_string)

# triple quotes string can extend multiple lines

my\_string = """Hello, welcome to

the world of Python"""

print(my\_string)

When you run the program, the output will be:

Hello

Hello

Hello

Hello, welcome to

the world of Python

## How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an IndexError. The index must be an integer. We can't use floats or other types, this will result into TypeError.

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator :(colon).

#Accessing string characters in Python

str = 'programiz'

print('str = ', str)

#first character

print('str[0] = ', str[0])

#last character

print('str[-1] = ', str[-1])

#slicing 2nd to 5th character

print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character

print('str[5:-2] = ', str[5:-2])

When we run the above program, we get the following output:

str = programiz

str[0] = p

str[-1] = z

str[1:5] = rogr

str[5:-2] = am

If we try to access an index out of the range or use numbers other than an integer, we will get errors.

# index must be in range

>>> my\_string[15]

...

IndexError: string index out of range

# index must be an integer

>>> my\_string[1.5]

...

TypeError: string indices must be integers

Slicing can be best visualized by considering the index to be between the elements as shown below.

If we want to access a range, we need the index that will slice the portion from the string.

String Slicing in Python

## How to change or delete a string?

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. We can simply reassign different strings to the same name.

>>> my\_string = 'programiz'

>>> my\_string[5] = 'a'

...

TypeError: 'str' object does not support item assignment

>>> my\_string = 'Python'

>>> my\_string

'Python'

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the del keyword.

>>> del my\_string[1]

...

TypeError: 'str' object doesn't support item deletion

>>> del my\_string

>>> my\_string

...

NameError: name 'my\_string' is not defined

## Python String Operations

There are many operations that can be performed with strings which makes it one of the most used data types in Python.

To learn more about the data types available in Python visit: [Python Data Types](https://www.programiz.com/python-programming/variables-datatypes)

### Concatenation of Two or More Strings

Joining of two or more strings into a single one is called concatenation.

The **+** operator does this in Python. Simply writing two string literals together also concatenates them.

The **\*** operator can be used to repeat the string for a given number of times.

# Python String Operations

str1 = 'Hello'

str2 ='World!'

# using +

print('str1 + str2 = ', str1 + str2)

# using \*

print('str1 \* 3 =', str1 \* 3)

When we run the above program, we get the following output:

str1 + str2 = HelloWorld!

str1 \* 3 = HelloHelloHello

Writing two string literals together also concatenates them like **+** operator.

If we want to concatenate strings in different lines, we can use parentheses.

>>> # two string literals together

>>> 'Hello ''World!'

'Hello World!'

>>> # using parentheses

>>> s = ('Hello '

... 'World')

>>> s

'Hello World'

### Iterating Through a string

We can iterate through a string using a [for loop](https://www.programiz.com/python-programming/for-loop). Here is an example to count the number of 'l's in a string.

# Iterating through a string

count = 0

for letter in 'Hello World':

if(letter == 'l'):

count += 1

print(count,'letters found')

When we run the above program, we get the following output:

3 letters found

### String Membership Test

We can test if a substring exists within a string or not, using the keyword in.

>>> 'a' in 'program'

True

>>> 'at' not in 'battle'

False

### Built-in functions to Work with Python

Various built-in functions that work with sequence work with strings as well.

Some of the commonly used ones are enumerate() and len(). The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, len() returns the length (number of characters) of the string.

str = 'cold'

# enumerate()

list\_enumerate = list(enumerate(str))

print('list(enumerate(str) = ', list\_enumerate)

#character count

print('len(str) = ', len(str))

When we run the above program, we get the following output:

list(enumerate(str) = [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]

len(str) = 4

## Python String Formatting

### Escape Sequence

If we want to print a text like He said, "What's there?", we can neither use single quotes nor double quotes. This will result in a SyntaxError as the text itself contains both single and double quotes.

>>> print("He said, "What's there?"")

...

SyntaxError: invalid syntax

>>> print('He said, "What's there?"')

...

SyntaxError: invalid syntax

One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences.

An escape sequence starts with a backslash and is interpreted differently. If we use a single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes. Here is how it can be done to represent the above text.

# using triple quotes

print('''He said, "What's there?"''')

# escaping single quotes

print('He said, "What\'s there?"')

# escaping double quotes

print("He said, \"What's there?\"")

When we run the above program, we get the following output:

He said, "What's there?"

He said, "What's there?"

He said, "What's there?"

Here is a list of all the escape sequences supported by Python.

|  |  |
| --- | --- |
| Escape Sequence | Description |
| \newline | Backslash and newline ignored |
| \\ | Backslash |
| \' | Single quote |
| \" | Double quote |
| \a | ASCII Bell |
| \b | ASCII Backspace |
| \f | ASCII Formfeed |
| \n | ASCII Linefeed |
| \r | ASCII Carriage Return |
| \t | ASCII Horizontal Tab |
| \v | ASCII Vertical Tab |
| \ooo | Character with octal value ooo |
| \xHH | Character with hexadecimal value HH |

Here are some examples

>>> print("C:\\Python32\\Lib")

C:\Python32\Lib

>>> print("This is printed\nin two lines")

This is printed

in two lines

>>> print("This is \x48\x45\x58 representation")

This is HEX representation

### Raw String to ignore escape sequence

Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place r or R in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

>>> print("This is \x61 \ngood example")

This is a

good example

>>> print(r"This is \x61 \ngood example")

This is \x61 \ngood example

### The format() Method for Formatting Strings

The format() method that is available with the string object is very versatile and powerful in formatting strings. Format strings contain curly braces {} as placeholders or replacement fields which get replaced.

We can use positional arguments or keyword arguments to specify the order.

# Python string format() method

# default(implicit) order

default\_order = "{}, {} and {}".format('John','Bill','Sean')

print('\n--- Default Order ---')

print(default\_order)

# order using positional argument

positional\_order = "{1}, {0} and {2}".format('John','Bill','Sean')

print('\n--- Positional Order ---')

print(positional\_order)

# order using keyword argument

keyword\_order = "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')

print('\n--- Keyword Order ---')

print(keyword\_order)

When we run the above program, we get the following output:

--- Default Order ---

John, Bill and Sean

--- Positional Order ---

Bill, John and Sean

--- Keyword Order ---

Sean, Bill and John

The format() method can have optional format specifications. They are separated from the field name using colon. For example, we can left-justify <, right-justify > or center ^ a string in the given space.

We can also format integers as binary, hexadecimal, etc. and floats can be rounded or displayed in the exponent format. There are tons of formatting you can use. Visit here for all the [string formatting available with the](https://www.programiz.com/python-programming/methods/string/format) [format()](https://www.programiz.com/python-programming/methods/string/format) method.

>>> # formatting integers

>>> "Binary representation of {0} is {0:b}".format(12)

'Binary representation of 12 is 1100'

>>> # formatting floats

>>> "Exponent representation: {0:e}".format(1566.345)

'Exponent representation: 1.566345e+03'

>>> # round off

>>> "One third is: {0:.3f}".format(1/3)

'One third is: 0.333'

>>> # string alignment

>>> "|{:<10}|{:^10}|{:>10}|".format('butter','bread','ham')

'|butter | bread | ham|'

### Old style formatting

We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

>>> x = 12.3456789

>>> print('The value of x is %3.2f' %x)

The value of x is 12.35

>>> print('The value of x is %3.4f' %x)

The value of x is 12.3457

## Common Python String Methods

There are numerous methods available with the string object. The format() method that we mentioned above is one of them. Some of the commonly used methods are lower(), upper(), join(), split(), find(), replace() etc. Here is a complete list of all the [built-in methods to work with strings in Python](https://www.programiz.com/python-programming/methods/string).

>>> "PrOgRaMiZ".lower()

'programiz'

>>> "PrOgRaMiZ".upper()

'PROGRAMIZ'

>>> "This will split all words into a list".split()

['This', 'will', 'split', 'all', 'words', 'into', 'a', 'list']

>>> ' '.join(['This', 'will', 'join', 'all', 'words', 'into', 'a', 'string'])

'This will join all words into a string'

>>> 'Happy New Year'.find('ew')

7

>>> 'Happy New Year'.replace('Happy','Brilliant')

'Brilliant New Year

## Python Set

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

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

# printing set variable

print("a = ", a)

# data type of variable a

print(type(a))

**Output**

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

<class 'set'>

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

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

print(a)

**Output**

{1, 2, 3}

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

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

>>> a[1]

Traceback (most recent call last):

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

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

TypeError: 'set' object does not support indexing

## Creating Python Sets

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like [lists](https://www.programiz.com/python-programming/list), sets or [dictionaries](https://www.programiz.com/python-programming/dictionary) as its elements.

# Different types of sets in Python

# set of integers

my\_set = {1, 2, 3}

print(my\_set)

# set of mixed datatypes

my\_set = {1.0, "Hello", (1, 2, 3)}

print(my\_set)

**Output**

{1, 2, 3}

{1.0, (1, 2, 3), 'Hello'}

Try the following examples as well.

# set cannot have duplicates

# Output: {1, 2, 3, 4}

my\_set = {1, 2, 3, 4, 3, 2}

print(my\_set)

# we can make set from a list

# Output: {1, 2, 3}

my\_set = set([1, 2, 3, 2])

print(my\_set)

# set cannot have mutable items

# here [3, 4] is a mutable list

# this will cause an error.

my\_set = {1, 2, [3, 4]}

**Output**

{1, 2, 3, 4}

{1, 2, 3}

Traceback (most recent call last):

File "<string>", line 15, in <module>

my\_set = {1, 2, [3, 4]}

TypeError: unhashable type: 'list'

Creating an empty set is a bit tricky.

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

# Distinguish set and dictionary while creating empty set

# initialize a with {}

a = {}

# check data type of a

print(type(a))

# initialize a with set()

a = set()

# check data type of a

print(type(a))

**Output**

<class 'dict'>

<class 'set'>

## Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take [tuples](https://www.programiz.com/python-programming/tuple), lists, [strings](https://www.programiz.com/python-programming/string) or other sets as its argument. In all cases, duplicates are avoided.

# initialize my\_set

my\_set = {1, 3}

print(my\_set)

# if you uncomment line 9,

# you will get an error

# TypeError: 'set' object does not support indexing

# my\_set[0]

# add an element

# Output: {1, 2, 3}

my\_set.add(2)

print(my\_set)

# add multiple elements

# Output: {1, 2, 3, 4}

my\_set.update([2, 3, 4])

print(my\_set)

# add list and set

# Output: {1, 2, 3, 4, 5, 6, 8}

my\_set.update([4, 5], {1, 6, 8})

print(my\_set)

**Output**

{1, 3}

{1, 2, 3}

{1, 2, 3, 4}

{1, 2, 3, 4, 5, 6, 8}

## Removing elements from a set

A particular item can be removed from a set using the methods discard() and remove().

The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).

The following example will illustrate this.

# Difference between discard() and remove()

# initialize my\_set

my\_set = {1, 3, 4, 5, 6}

print(my\_set)

# discard an element

# Output: {1, 3, 5, 6}

my\_set.discard(4)

print(my\_set)

# remove an element

# Output: {1, 3, 5}

my\_set.remove(6)

print(my\_set)

# discard an element

# not present in my\_set

# Output: {1, 3, 5}

my\_set.discard(2)

print(my\_set)

# remove an element

# not present in my\_set

# you will get an error.

# Output: KeyError

my\_set.remove(2)

**Output**

{1, 3, 4, 5, 6}

{1, 3, 5, 6}

{1, 3, 5}

{1, 3, 5}

Traceback (most recent call last):

File "<string>", line 28, in <module>

KeyError: 2

Similarly, we can remove and return an item using the pop() method.

Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all the items from a set using the clear() method.

# initialize my\_set

# Output: set of unique elements

my\_set = set("HelloWorld")

print(my\_set)

# pop an element

# Output: random element

print(my\_set.pop())

# pop another element

my\_set.pop()

print(my\_set)

# clear my\_set

# Output: set()

my\_set.clear()

print(my\_set)

print(my\_set)

**Output**

{'H', 'l', 'r', 'W', 'o', 'd', 'e'}

H

{'r', 'W', 'o', 'd', 'e'}

set()

## Python Set Operations

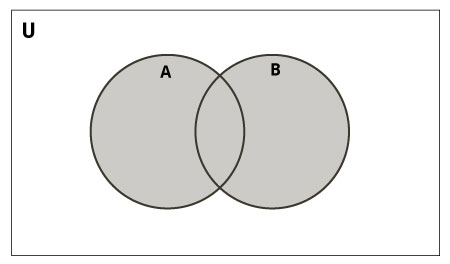
Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

>>> A = {1, 2, 3, 4, 5}

>>> B = {4, 5, 6, 7, 8}

### Set Union

Set Union in Python

Union of A and B is a set of all elements from both sets.

Union is performed using | operator. Same can be accomplished using the union() method.

# Set union method

# initialize A and B

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

B = {4, 5, 6, 7, 8}

# use | operator

# Output: {1, 2, 3, 4, 5, 6, 7, 8}

print(A | B)

**Output**

{1, 2, 3, 4, 5, 6, 7, 8}

Try the following examples on Python shell.

# use union function

>>> A.union(B)

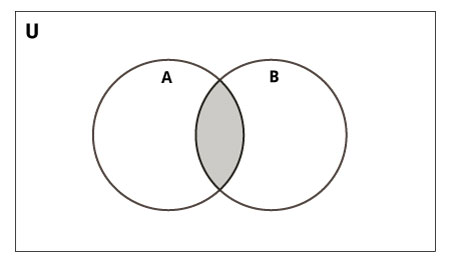
{1, 2, 3, 4, 5, 6, 7, 8}

# use union function on B

>>> B.union(A)

{1, 2, 3, 4, 5, 6, 7, 8}

### Set Intersection

Set Intersection in Python

Intersection of A and B is a set of elements that are common in both the sets.

Intersection is performed using & operator. Same can be accomplished using the intersection() method.

# Intersection of sets

# initialize A and B

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

B = {4, 5, 6, 7, 8}

# use & operator

# Output: {4, 5}

print(A & B)

**Output**

{4, 5}

Try the following examples on Python shell.

# use intersection function on A

>>> A.intersection(B)

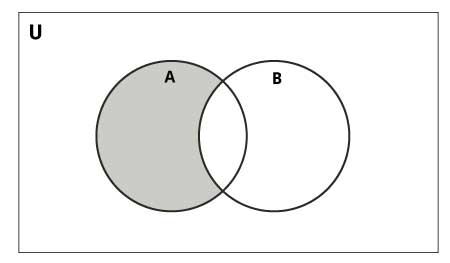
{4, 5}

# use intersection function on B

>>> B.intersection(A)

{4, 5}

### Set Difference

Set Difference in Python

Difference of the set B from set A(A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of elements in B but not in A.

Difference is performed using - operator. Same can be accomplished using the difference() method.

# Difference of two sets

# initialize A and B

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

B = {4, 5, 6, 7, 8}

# use - operator on A

# Output: {1, 2, 3}

print(A - B)

**Output**

{1, 2, 3}

Try the following examples on Python shell.

# use difference function on A

>>> A.difference(B)

{1, 2, 3}

# use - operator on B

>>> B - A

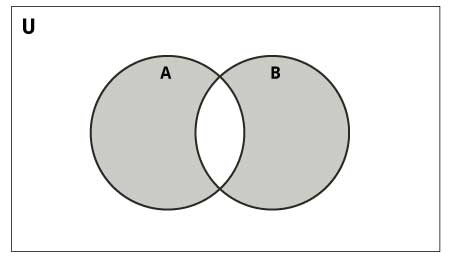
{8, 6, 7}

# use difference function on B

>>> B.difference(A)

{8, 6, 7}

### Set Symmetric Difference

Set Symmetric Difference in Python

Symmetric Difference of A and B is a set of elements in A and B but not in both (excluding the intersection).

Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric\_difference().

# Symmetric difference of two sets

# initialize A and B

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

B = {4, 5, 6, 7, 8}

# use ^ operator

# Output: {1, 2, 3, 6, 7, 8}

print(A ^ B)

**Output**

{1, 2, 3, 6, 7, 8}

Try the following examples on Python shell.

# use symmetric\_difference function on A

>>> A.symmetric\_difference(B)

{1, 2, 3, 6, 7, 8}

# use symmetric\_difference function on B

>>> B.symmetric\_difference(A)

{1, 2, 3, 6, 7, 8}

## Other Python Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with the set objects:

|  |  |
| --- | --- |
| Method | Description |
| [add()](https://www.programiz.com/python-programming/methods/set/add) | Adds an element to the set |
| [clear()](https://www.programiz.com/python-programming/methods/set/clear) | Removes all elements from the set |
| [copy()](https://www.programiz.com/python-programming/methods/set/copy) | Returns a copy of the set |
| [difference()](https://www.programiz.com/python-programming/methods/set/difference) | Returns the difference of two or more sets as a new set |
| [difference\_update()](https://www.programiz.com/python-programming/methods/set/difference_update) | Removes all elements of another set from this set |
| [discard()](https://www.programiz.com/python-programming/methods/set/discard) | Removes an element from the set if it is a member. (Do nothing if the element is not in set) |
| [intersection()](https://www.programiz.com/python-programming/methods/set/intersection) | Returns the intersection of two sets as a new set |
| [intersection\_update()](https://www.programiz.com/python-programming/methods/set/intersection_update) | Updates the set with the intersection of itself and another |
| [isdisjoint()](https://www.programiz.com/python-programming/methods/set/isdisjoint) | Returns True if two sets have a null intersection |
| [issubset()](https://www.programiz.com/python-programming/methods/set/issubset) | Returns True if another set contains this set |
| [issuperset()](https://www.programiz.com/python-programming/methods/set/issuperset) | Returns True if this set contains another set |
| [pop()](https://www.programiz.com/python-programming/methods/set/pop) | Removes and returns an arbitrary set element. Raises KeyError if the set is empty |
| [remove()](https://www.programiz.com/python-programming/methods/set/remove) | Removes an element from the set. If the element is not a member, raises a KeyError |
| [symmetric\_difference()](https://www.programiz.com/python-programming/methods/set/symmetric_difference) | Returns the symmetric difference of two sets as a new set |
| [symmetric\_difference\_update()](https://www.programiz.com/python-programming/methods/set/symmetric_difference_update) | Updates a set with the symmetric difference of itself and another |
| [union()](https://www.programiz.com/python-programming/methods/set/union) | Returns the union of sets in a new set |
| [update()](https://www.programiz.com/python-programming/methods/set/update) | Updates the set with the union of itself and others |

## Other Set Operations

### Set Membership Test

We can test if an item exists in a set or not, using the in keyword.

# in keyword in a set

# initialize my\_set

my\_set = set("apple")

# check if 'a' is present

# Output: True

print('a' in my\_set)

# check if 'p' is present

# Output: False

print('p' not in my\_set)

**Output**

True

False

### Iterating Through a Set

We can iterate through each item in a set using a for loop.

>>> for letter in set("apple"):

... print(letter)

...

a

p

e

l

### Built-in Functions with Set

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with sets to perform different tasks.

|  |  |
| --- | --- |
| Function | Description |
| [all()](https://www.programiz.com/python-programming/methods/built-in/all) | Returns True if all elements of the set are true (or if the set is empty). |
| [any()](https://www.programiz.com/python-programming/methods/built-in/any) | Returns True if any element of the set is true. If the set is empty, returns False. |
| [enumerate()](https://www.programiz.com/python-programming/methods/built-in/enumerate) | Returns an enumerate object. It contains the index and value for all the items of the set as a pair. |
| [len()](https://www.programiz.com/python-programming/methods/built-in/len) | Returns the length (the number of items) in the set. |
| [max()](https://www.programiz.com/python-programming/methods/built-in/max) | Returns the largest item in the set. |
| [min()](https://www.programiz.com/python-programming/methods/built-in/min) | Returns the smallest item in the set. |
| [sorted()](https://www.programiz.com/python-programming/methods/built-in/sorted) | Returns a new sorted list from elements in the set(does not sort the set itself). |
| [sum()](https://www.programiz.com/python-programming/methods/built-in/sum) | Returns the sum of all elements in the set. |

## Python Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the [frozenset()](https://www.programiz.com/python-programming/methods/built-in/frozenset) function.

This data type supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric\_difference() and union(). Being immutable, it does not have methods that add or remove elements.

# Frozensets

# initialize A and B

A = frozenset([1, 2, 3, 4])

B = frozenset([3, 4, 5, 6])

Try these examples on Python shell.

>>> A.isdisjoint(B)

False

>>> A.difference(B)

frozenset({1, 2})

>>> A | B

frozenset({1, 2, 3, 4, 5, 6})

>>> A.add(3)

...

AttributeError: 'frozenset' object has no attribute 'add'

## Python Dictionary

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

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

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

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

>>> type(d)

<class 'dict'>

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

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

print(type(d))

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

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

# Generates error

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

**Output**

<class 'dict'>

d[1] = value

d['key'] = 2

Traceback (most recent call last):

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

KeyError: 2

## Conversion between data types

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

>>> float(5)

5.0

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

>>> int(10.6)

10

>>> int(-10.6)

-10

Conversion to and from string must contain compatible values.

>>> float('2.5')

2.5

>>> str(25)

'25'

>>> int('1p')

Traceback (most recent call last):

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

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

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

We can even convert one sequence to another.

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

{1, 2, 3}

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

(5, 6, 7)

>>> list('hello')

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

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

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

{1: 2, 3: 4}

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

{3: 26, 4: 44}

## Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.

An item has a key and a corresponding value that is expressed as a pair (**key: value**).

While the values can be of any data type and can repeat, keys must be of immutable type ([string](https://www.programiz.com/python-programming/string), [number](https://www.programiz.com/python-programming/numbers) or [tuple](https://www.programiz.com/python-programming/tuple) with immutable elements) and must be unique.

# empty dictionary

my\_dict = {}

# dictionary with integer keys

my\_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys

my\_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()

my\_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair

my\_dict = dict([(1,'apple'), (2,'ball')])

As you can see from above, we can also create a dictionary using the built-in dict() function.

## Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method.

If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

# get vs [] for retrieving elements

my\_dict = {'name': 'Jack', 'age': 26}

# Output: Jack

print(my\_dict['name'])

# Output: 26

print(my\_dict.get('age'))

# Trying to access keys which doesn't exist throws error

# Output None

print(my\_dict.get('address'))

# KeyError

print(my\_dict['address'])

**Output**

Jack

26

None

Traceback (most recent call last):

File "<string>", line 15, in <module>

print(my\_dict['address'])

KeyError: 'address'

## Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

# Changing and adding Dictionary Elements

my\_dict = {'name': 'Jack', 'age': 26}

# update value

my\_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}

print(my\_dict)

# add item

my\_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}

print(my\_dict)

**Output**

{'name': 'Jack', 'age': 27}

{'name': 'Jack', 'age': 27, 'address': 'Downtown'}

## Removing elements from Dictionary

We can remove a particular item in a dictionary by using the pop() method. This method removes an item with the provided key and returns the value.

The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary. All the items can be removed at once, using the clear() method.

We can also use the del keyword to remove individual items or the entire dictionary itself.

# Removing elements from a dictionary

# create a dictionary

squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value

# Output: 16

print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}

print(squares)

# remove an arbitrary item, return (key,value)

# Output: (5, 25)

print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}

print(squares)

# remove all items

squares.clear()

# Output: {}

print(squares)

# delete the dictionary itself

del squares

# Throws Error

print(squares)

**Output**

16

{1: 1, 2: 4, 3: 9, 5: 25}

(5, 25)

{1: 1, 2: 4, 3: 9}

{}

Traceback (most recent call last):

File "<string>", line 30, in <module>

print(squares)

NameError: name 'squares' is not defined

## Python Dictionary Methods

Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.

|  |  |
| --- | --- |
| Method | Description |
| [clear()](https://www.programiz.com/python-programming/methods/dictionary/clear) | Removes all items from the dictionary. |
| [copy()](https://www.programiz.com/python-programming/methods/dictionary/copy) | Returns a shallow copy of the dictionary. |
| [fromkeys(seq[, v])](https://www.programiz.com/python-programming/methods/dictionary/fromkeys) | Returns a new dictionary with keys from seq and value equal to v (defaults to None). |
| [get(key[,d])](https://www.programiz.com/python-programming/methods/dictionary/get) | Returns the value of the key. If the key does not exist, returns d (defaults to None). |
| [items()](https://www.programiz.com/python-programming/methods/dictionary/items) | Return a new object of the dictionary's items in (key, value) format. |
| [keys()](https://www.programiz.com/python-programming/methods/dictionary/keys) | Returns a new object of the dictionary's keys. |
| [pop(key[,d])](https://www.programiz.com/python-programming/methods/dictionary/pop) | Removes the item with the key and returns its value or d if key is not found. If d is not provided and the key is not found, it raises KeyError. |
| [popitem()](https://www.programiz.com/python-programming/methods/dictionary/popitem) | Removes and returns an arbitrary item (**key, value**). Raises KeyError if the dictionary is empty. |
| [setdefault(key[,d])](https://www.programiz.com/python-programming/methods/dictionary/setdefault) | Returns the corresponding value if the key is in the dictionary. If not, inserts the key with a value of d and returns d (defaults to None). |
| [update([other])](https://www.programiz.com/python-programming/methods/dictionary/update) | Updates the dictionary with the key/value pairs from other, overwriting existing keys. |
| [values()](https://www.programiz.com/python-programming/methods/dictionary/values) | Returns a new object of the dictionary's values |

Here are a few example use cases of these methods.

# Dictionary Methods

marks = {}.fromkeys(['Math', 'English', 'Science'], 0)

# Output: {'English': 0, 'Math': 0, 'Science': 0}

print(marks)

for item in marks.items():

print(item)

# Output: ['English', 'Math', 'Science']

print(list(sorted(marks.keys())))

**Output**

{'Math': 0, 'English': 0, 'Science': 0}

('Math', 0)

('English', 0)

('Science', 0)

['English', 'Math', 'Science']

## Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (**key: value**) followed by a for statement inside curly braces {}.

Here is an example to make a dictionary with each item being a pair of a number and its square.

# Dictionary Comprehension

squares = {x: x\*x for x in range(6)}

print(squares)

**Output**

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

This code is equivalent to

squares = {}

for x in range(6):

squares[x] = x\*x

print(squares)

**Output**

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

A dictionary comprehension can optionally contain more [for](https://www.programiz.com/python-programming/for-loop) or [if](https://www.programiz.com/python-programming/if-elif-else) statements.

An optional if statement can filter out items to form the new dictionary.

Here are some examples to make a dictionary with only odd items.

# Dictionary Comprehension with if conditional

odd\_squares = {x: x\*x for x in range(11) if x % 2 == 1}

print(odd\_squares)

**Output**

{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

To learn more dictionary comprehensions, visit [Python Dictionary Comprehension](https://www.programiz.com/python-programming/dictionary-comprehension).

## Other Dictionary Operations

### Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword in. Notice that the membership test is only for the keys and not for the values.

# Membership Test for Dictionary Keys

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True

print(1 in squares)

# Output: True

print(2 not in squares)

# membership tests for key only not value

# Output: False

print(49 in squares)

**Output**

True

True

False

### Iterating Through a Dictionary

We can iterate through each key in a dictionary using a for loop.

# Iterating through a Dictionary

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

for i in squares:

print(squares[i])

**Output**

1

9

25

49

81

### Dictionary Built-in Functions

Built-in functions like all(), any(), len(), cmp(), sorted(), etc. are commonly used with dictionaries to perform different tasks.

|  |  |
| --- | --- |
| Function | Description |
| [all()](https://www.programiz.com/python-programming/methods/built-in/all) | Return True if all keys of the dictionary are True (or if the dictionary is empty). |
| [any()](https://www.programiz.com/python-programming/methods/built-in/any) | Return True if any key of the dictionary is true. If the dictionary is empty, return False. |
| [len()](https://www.programiz.com/python-programming/methods/built-in/len) | Return the length (the number of items) in the dictionary. |
| cmp() | Compares items of two dictionaries. (Not available in Python 3) |
| [sorted()](https://www.programiz.com/python-programming/methods/built-in/sorted) | Return a new sorted list of keys in the dictionary. |

Here are some examples that use built-in functions to work with a dictionary.

# Dictionary Built-in Functions

squares = {0: 0, 1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: False

print(all(squares))

# Output: True

print(any(squares))

# Output: 6

print(len(squares))

# Output: [0, 1, 3, 5, 7, 9]

print(sorted(squares))

**Output**

False

True

6

[0, 1, 3, 5, 7, 9

# Python Type Conversion and Type Casting

#### **In this article, you will learn about the Type conversion and uses of type conversion.**

## Type Conversion

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

1. Implicit Type Conversion
2. Explicit Type Conversion

## Implicit Type Conversion

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

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

### Example 1: Converting integer to float

num\_int = 123

num\_flo = 1.23

num\_new = num\_int + num\_flo

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

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

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

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

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

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

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

Value of num\_new: 124.23

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

In the above program,

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

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

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

num\_int = 123

num\_str = "456"

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

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

print(num\_int+num\_str)

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

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

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

Traceback (most recent call last):

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

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

In the above program,

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

## Explicit Type Conversion

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

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

Syntax :

<required\_datatype>(expression)

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

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

num\_int = 123

num\_str = "456"

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

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

num\_str = int(num\_str)

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

num\_sum = num\_int + num\_str

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

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

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

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

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

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

Sum of num\_int and num\_str: 579

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

In the above program,

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

## Key Points to Remember

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

# Python Input, Output and Import

#### **This tutorial focuses on two built-in functions print() and input() to perform I/O task in Python. Also, you will learn to import modules and use them in your program.**

Python provides numerous [built-in functions](https://www.programiz.com/python-programming/built-in-function) that are readily available to us at the Python prompt.

Some of the functions like input() and print() are widely used for standard input and output operations respectively. Let us see the output section first.

## Python Output Using print() function

We use the print() function to output data to the standard output device (screen). We can also [output data to a file](https://www.programiz.com/python-programming/file-operation), but this will be discussed later.

An example of its use is given below.

print('This sentence is output to the screen')

**Output**

This sentence is output to the screen

Another example is given below:

a = 5

print('The value of a is', a)

**Output**

The value of a is 5

In the second print() statement, we can notice that space was added between the [string](https://www.programiz.com/python-programming/string) and the value of variable a. This is by default, but we can change it.

The actual syntax of the print() function is:

print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Here, objects is the value(s) to be printed.

The sep separator is used between the values. It defaults into a space character.

After all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout (screen). Here is an example to illustrate this.

print(1, 2, 3, 4)

print(1, 2, 3, 4, sep='\*')

print(1, 2, 3, 4, sep='#', end='&')

**Output**

1 2 3 4

1\*2\*3\*4

1#2#3#4&

## Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

>>> x = 5; y = 10

>>> print('The value of x is {} and y is {}'.format(x,y))

The value of x is 5 and y is 10

Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

print('I love {0} and {1}'.format('bread','butter'))

print('I love {1} and {0}'.format('bread','butter'))

**Output**

I love bread and butter

I love butter and bread

We can even use keyword arguments to format the string.

>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))

Hello John, Goodmorning

We can also format strings like the old sprintf() style used in [C programming language](https://www.programiz.com/c-programming). We use the % operator to accomplish this.

>>> x = 12.3456789

>>> print('The value of x is %3.2f' %x)

The value of x is 12.35

>>> print('The value of x is %3.4f' %x)

The value of x is 12.3457

## Python Input

Up until now, our programs were static. The value of variables was defined or hard coded into the source code.

To allow flexibility, we might want to take the input from the user. In Python, we have the input() function to allow this. The syntax for input() is:

input([prompt])

where prompt is the string we wish to display on the screen. It is optional.

>>> num = input('Enter a number: ')

Enter a number: 10

>>> num

'10'

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

>>> int('10')

10

>>> float('10')

10.0

This same operation can be performed using the eval() function. But eval takes it further. It can evaluate even expressions, provided the input is a string

>>> int('2+3')

Traceback (most recent call last):

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

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

ValueError: invalid literal for int() with base 10: '2+3'

>>> eval('2+3')

5

## Python Import

When our program grows bigger, it is a good idea to break it into different modules.

A module is a file containing Python definitions and statements. [Python modules](https://www.programiz.com/python-programming/modules) have a filename and end with the extension .py.

Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.

For example, we can import the math module by typing the following line:

import math

We can use the module in the following ways:

import math

print(math.pi)

**Output**

3.141592653589793

Now all the definitions inside math module are available in our scope. We can also import some specific attributes and functions only, using the from keyword. For example:

>>> from math import pi

>>> pi

3.141592653589793

While importing a module, Python looks at several places defined in sys.path. It is a list of directory locations.

>>> import sys

>>> sys.path

['',

'C:\\Python33\\Lib\\idlelib',

'C:\\Windows\\system32\\python33.zip',

'C:\\Python33\\DLLs',

'C:\\Python33\\lib',

'C:\\Python33',

'C:\\Python33\\lib\\site-packages']

We can also add our own location to this list.

# Python Operators

#### **In this tutorial, you'll learn everything about different types of operators in Python, their syntax and how to use them with examples.**

## What are operators in python?

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

For example:

>>> 2+3

5

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

## Arithmetic operators

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

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

### Example 1: Arithmetic operators in Python

x = 15 154 =15\*15\*15\*15=50625

y = 4

# Output: x + y = 19

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

# Output: x - y = 11

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

# Output: x \* y = 60

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

# Output: x / y = 3.75

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

# Output: x // y = 3

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

# Output: x \*\* y = 50625

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

**Output**

x + y = 19

x - y = 11

x \* y = 60

x / y = 3.75

x // y = 3

x \*\* y = 50625

## Comparison operators

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

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

### Example 2: Comparison operators in Python

x = 10

y = 12

# Output: x > y is False

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

# Output: x < y is True

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

# Output: x == y is False

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

# Output: x != y is True

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

# Output: x >= y is False

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

# Output: x <= y is True

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

**Output**

x > y is False

x < y is True

x == y is False

x != y is True

x >= y is False

x <= y is True

## Logical operators

Logical operators are the and, or, not operators.

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

### Example 3: Logical Operators in Python

x = True

y = False

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

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

print('not x is',not x)

**Output**

x and y is False

x or y is True

not x is False

## Bitwise operators

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

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

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

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

## Assignment operators =

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

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

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

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

Special operators

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

### Identity operators

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

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

### Example 4: Identity operators in Python

x1 = 5

y1 = 5

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

# Output: False

print(x1 is not y1)

# Output: True

print(x2 is y2)

# Output: False

print(x3 is y3)

**Output**

False

True

False

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

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

### Membership operators

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

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

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

### Example #5: Membership operators in Python

x = 'Hello world'

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

# Output: True

print('H' in x)

# Output: True

print('hello' not in x)

# Output: True

print(1 in y)

# Output: False

print('a' in y)

**Output**

True

True

True

False

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

# Python Namespace and Scope

#### **In this tutorial, you will learn about namespace, mapping from names to objects, and scope of a variable.**

## What is Name in Python?

If you have ever read 'The Zen of Python' (type import this in the Python interpreter), the last line states, **Namespaces are one honking great idea -- let's do more of those!** So what are these mysterious namespaces? Let us first look at what name is.

Name (also called identifier) is simply a name given to objects. Everything in Python is an [object](https://www.programiz.com/python-programming/class). Name is a way to access the underlying object.

For example, when we do the assignment a = 2, 2 is an object stored in memory and a is the name we associate it with. We can get the address (in RAM) of some object through the [built-in function](https://www.programiz.com/python-programming/built-in-function) id(). Let's look at how to use it.

# Note: You may get different values for the id

a = 2

print('id(2) =', id(2))

print('id(a) =', id(a))

**Output**

id(2) = 9302208

id(a) = 9302208

Here, both refer to the same object 2, so they have the same id(). Let's make things a little more interesting.

# Note: You may get different values for the id

a = 2

print('id(a) =', id(a))

a = a+1

print('id(a) =', id(a))

print('id(3) =', id(3))

b = 2

print('id(b) =', id(b))

print('id(2) =', id(2))

**Output**

id(a) = 9302208

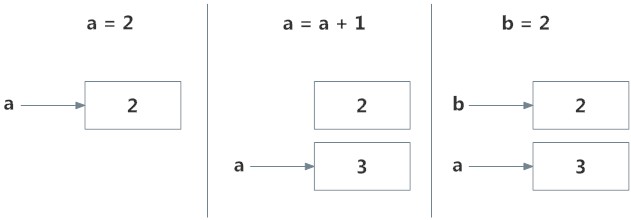
id(a) = 9302240

id(3) = 9302240

id(b) = 9302208

id(2) = 9302208

What is happening in the above sequence of steps? Let's use a diagram to explain this:

Memory diagram of variables in Python

Initially, an object 2 is created and the name a is associated with it, when we do a = a+1, a new object 3 is created and now a is associated with this object.

Note that id(a) and id(3) have the same values.

Furthermore, when b = 2 is executed, the new name b gets associated with the previous object 2.

This is efficient as Python does not have to create a new duplicate object. This dynamic nature of name binding makes Python powerful; a name could refer to any type of object.

>>> a = 5

>>> a = 'Hello World!'

>>> a = [1,2,3]

All these are valid and a will refer to three different types of objects in different instances. [Functions](https://www.programiz.com/python-programming/function) are objects too, so a name can refer to them as well.

def printHello():

print("Hello")

a = printHello

a()

**Output**

Hello

The same name a can refer to a function and we can call the function using this name.

## What is a Namespace in Python?

Now that we understand what names are, we can move on to the concept of namespaces.

To simply put it, a namespace is a collection of names.

In Python, you can imagine a namespace as a mapping of every name you have defined to corresponding objects.

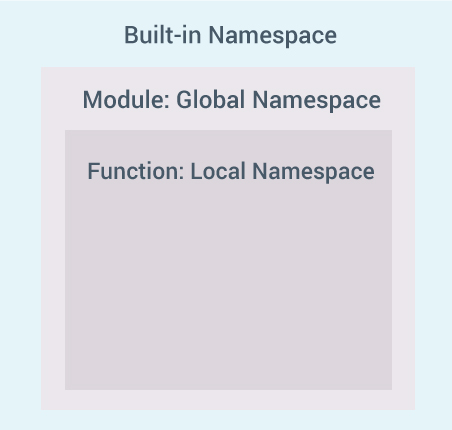
Different namespaces can co-exist at a given time but are completely isolated.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long as the interpreter runs.

This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program. Each [module](https://www.programiz.com/python-programming/modules) creates its own global namespace.

These different namespaces are isolated. Hence, the same name that may exist in different modules do not collide.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar, is the case with class. Following diagram may help to clarify this concept.

A diagram of different namespaces in Python

## Python Variable Scope

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.

A scope is the portion of a program from where a namespace can be accessed directly without any prefix.

At any given moment, there are at least three nested scopes.

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.

If there is a function inside another function, a new scope is nested inside the local scope.

## Example of Scope and Namespace in Python

def outer\_function():

b = 20

def inner\_func():

c = 30

a = 10

Here, the variable a is in the global namespace. Variable b is in the local namespace of outer\_function() and c is in the nested local namespace of inner\_function().

When we are in inner\_function(), c is local to us, b is nonlocal and a is global. We can read as well as assign new values to c but can only read b and a from inner\_function().

If we try to assign as a value to b, a new variable b is created in the local namespace which is different than the nonlocal b. The same thing happens when we assign a value to a.

However, if we declare a as global, all the reference and assignment go to the global a. Similarly, if we want to rebind the variable b, it must be declared as nonlocal. The following example will further clarify this.

def outer\_function():

a = 20

def inner\_function():

a = 30

print('a =', a)

inner\_function()

print('a =', a)

a = 10

outer\_function()

print('a =', a)

As you can see, the output of this program is

a = 30

a = 20

a = 10

In this program, three different variables a are defined in separate namespaces and accessed accordingly. While in the following program,

def outer\_function():

global a

a = 20

def inner\_function():

global a

a = 30

print('a =', a)

inner\_function()

print('a =', a)

a = 10

outer\_function()

print('a =', a)

The output of the program is.

a = 30

a = 30

a = 30

Here, all references and assignments are to the global a due to the use of keyword global.

# Python if...else Statement

#### **In this article, you will learn to create decisions in a Python program using different forms of if..else statement.**

## What is if...else statement in Python?

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

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

### Python if Statement Syntax

if test expression:

statement(s)

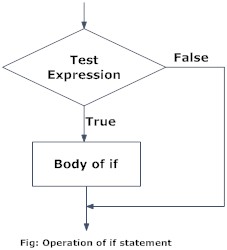
Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True.

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

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

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

### Python if Statement Flowchart

Flowchart of if statement in Python programming

### Example: Python if Statement

# If the number is positive, we print an appropriate message

num = 3

if num > 0:

print(num, "is a positive number.")

print("This is always printed.")

num = -1

if num > 0:

print(num, "is a positive number.")

print("This is also always printed.")

When you run the program, the output will be:

3 is a positive number

This is always printed

This is also always printed.

In the above example, num > 0 is the test expression.

The body of if is executed only if this evaluates to True.

When the variable num is equal to 3, test expression is true and statements inside the body of if are executed.

If the variable num is equal to -1, test expression is false and statements inside the body of if are skipped.

The print() statement falls outside of the if block (unindented). Hence, it is executed regardless of the test expression.

## Python if...else Statement

### Syntax of if...else

if test expression:

Body of if

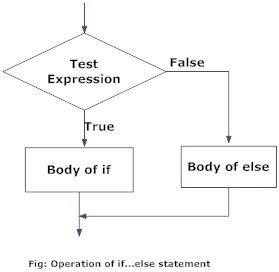
else:

Body of else

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

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

### Python if..else Flowchart

Flowchart of if...else statement in Python

### Example of if...else

# Program checks if the number is positive or negative

# And displays an appropriate message

num = 3

# Try these two variations as well.

# num = -5

# num = 0

if num >= 0:

print("Positive or Zero")

else:

print("Negative number")

**Output**

Positive or Zero

In the above example, when num is equal to 3, the test expression is true and the body of if is executed and the body of else is skipped.

If num is equal to -5, the test expression is false and the body of else is executed and the body of if is skipped.

If num is equal to 0, the test expression is true and body of if is executed and body of else is skipped.

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

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

if test expression:

Body of if

elif test expression:

Body of elif

else:

Body of else

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

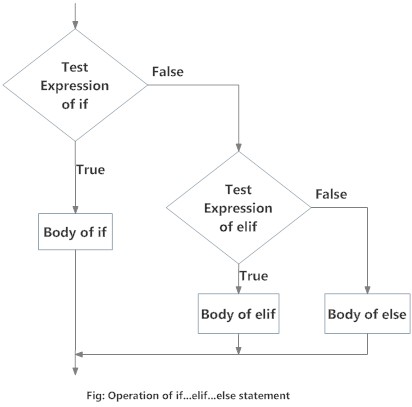
If the condition for if is False, it checks the condition of the next elif block and so on.

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

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

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

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

Flowchart of if...elif....else statement in Python

### Example of if...elif...else

'''In this program,

we check if the number is positive or

negative or zero and

display an appropriate message'''

num = 3.4

# Try these two variations as well:

# num = 0

# num = -4.5

if num > 0:

print("Positive number")

elif num == 0:

print("Zero")

else:

print("Negative number")

When variable num is positive, Positive number is printed.

If num is equal to 0, Zero is printed.

If num is negative, Negative number is printed.

## Python Nested if statements

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

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

### Python Nested if Example

'''In this program, we input a number

check if the number is positive or

negative or zero and display

an appropriate message

This time we use nested if statement'''

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

if num >= 0:

if num == 0:

print("Zero")

else:

print("Positive number")

else:

print("Negative number")

**Output 1**

Enter a number: 5

Positive number

**Output 2**

Enter a number: -1

Negative number

**Output 3**

Enter a number: 0

Zero

# Python for Loop

#### **In this article, you'll learn to iterate over a sequence of elements using the different variations of for loop.**

What is for loop in Python?

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

### Syntax of for Loop

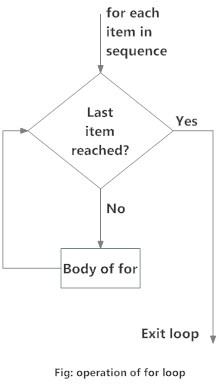
for val in sequence:

Body of for

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

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

### Flowchart of for Loop

Flowchart of for Loop in Python

### Example: Python for Loop

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

# List of numbers

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

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

sum = sum+val

print("The sum is", sum)

When you run the program, the output will be:

The sum is 48

## The range() function

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

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

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

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

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

The following example will clarify this.

print(range(10))

print(list(range(10)))

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

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

**Output**

range(0, 10)

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

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

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

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

# Program to iterate through a list using indexing

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

# iterate over the list using index

for i in range(len(genre)):

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

**Output**

I like pop

I like rock

​I like jazz

## for loop with else

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

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

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

Here is an example to illustrate this.

digits = [0, 1, 5]

for i in digits:

print(i)

else:

print("No items left.")

When you run the program, the output will be:

0

1

5

No items left.

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

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

# program to display student's marks from record

student\_name = 'Soyuj'

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

for student in marks:

if student == student\_name:

print(marks[student])

break

else:

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

**Output**

No entry with that name found.

# Python while Loop

#### **Loops are used in programming to repeat a specific block of code. In this article, you will learn to create a while loop in Python.**

What is while loop in Python?

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

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

### Syntax of while Loop in Python

while test\_expression:

Body of while

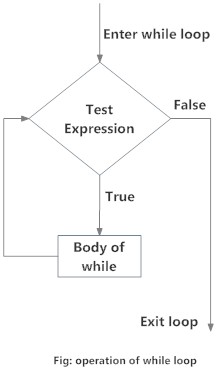
In the while loop, test expression is checked first. The body of the loop is entered only if the test\_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test\_expression evaluates to False.

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

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

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

### Flowchart of while Loop

Flowchart for while loop in Python

### Example: Python while Loop

# Program to add natural

# numbers up to

# sum = 1+2+3+...+n

# To take input from the user,

# n = int(input("Enter n: "))

n = 10

# initialize sum and counter

sum = 0

i = 1

while i <= n:

sum = sum + i

i = i+1 # update counter

# print the sum

print("The sum is", sum)

When you run the program, the output will be:

Enter n: 10

The sum is 55

In the above program, the test expression will be True as long as our counter variable i is less than or equal to n (10 in our program).

We need to increase the value of the counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never-ending loop).

Finally, the result is displayed.

## While loop with else

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

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

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

Here is an example to illustrate this.

'''Example to illustrate

the use of else statement

with the while loop'''

counter = 0

while counter < 3:

print("Inside loop")

counter = counter + 1

else:

print("Inside else")

**Output**

Inside loop

Inside loop

Inside loop

Inside else

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

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

# Python break and continue

#### **In this article, you will learn to use break and continue statements to alter the flow of a loop.**

## What is the use of break and continue in Python?

In Python, break and continue statements can alter the flow of a normal loop.

Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.

The break and continue statements are used in these cases.

## Python break statement

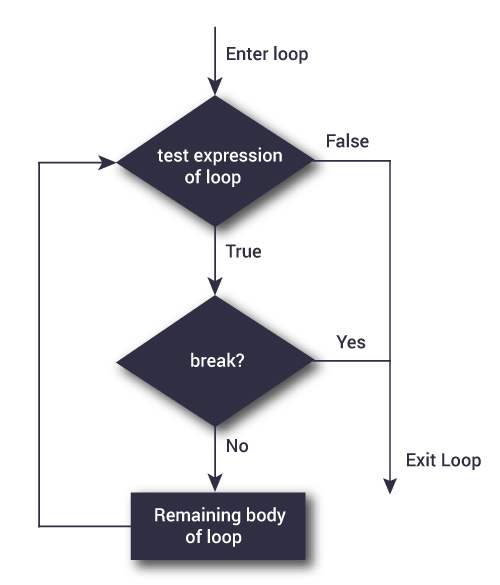
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

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

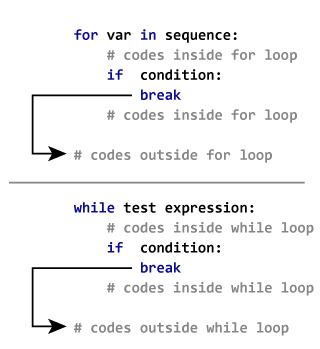
### Syntax of break

break

### Flowchart of break

Flowchart of break statement in Python

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

Working of the break statement

### Example: Python break

# Use of break statement inside the loop

for val in "string":

if val == "i":

break

print(val)

print("The end")

**Output**

s

t

r

The end

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

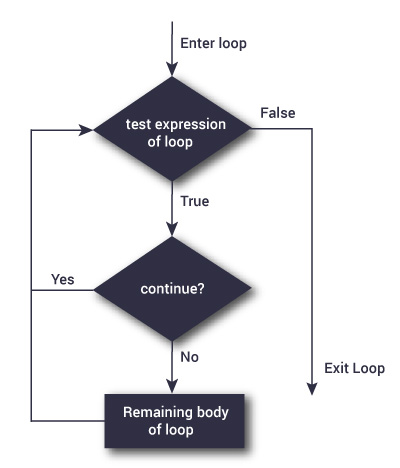
## Python continue statement

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

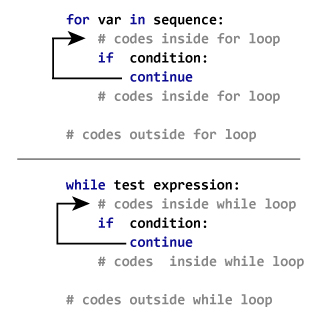
### Syntax of Continue

continue

### Flowchart of continue

Flowchart of continue statement in Python

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

How continue statement works in python

### Example: Python continue

# Program to show the use of continue statement inside loops

for val in "string":

if val == "i":

continue

print(val)

print("The end")

**Output**

s

t

r

n

g

The end

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

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

# Python pass statement

#### **In this article, you'll learn about pass statement. It is used as a placeholder for future implementation of functions, loops, etc.**

## What is pass statement in Python?

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

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

### Syntax of pass

pass

We generally use it as a placeholder.

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

### Example: pass Statement

'''pass is just a placeholder for

functionality to be added later.'''

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

for val in sequence:

pass

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

def function(args):

pass

class Example:

pass

# Python Functions

#### **In this article, you'll learn about functions, what a function is, the syntax, components, and types of functions. Also, you'll learn to create a function in Python.**

## What is a function in Python?

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

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

Furthermore, it avoids repetition and makes the code reusable.

### Syntax of Function

def function\_name(parameters):

"""docstring"""

statement(s)

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

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

### Example of a function

def greet(name):

"""

This function greets to

the person passed in as

a parameter

"""

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

### How to call a function in python?

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

>>> greet('Paul')

Hello, Paul. Good morning!

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

def greet(name):

"""

This function greets to

the person passed in as

a parameter

"""

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

greet('Paul')

## Docstrings

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

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

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

**For example**:

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

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

This function greets to

the person passed in as

a parameter

## The return statement

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

### Syntax of return

return [expression\_list]

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

**For example:**

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

Hello, May. Good morning!

None

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

### Example of return

def absolute\_value(num):

"""This function returns the absolute

value of the entered number"""

if num >= 0:

return num

else:

return -num

print(absolute\_value(2))

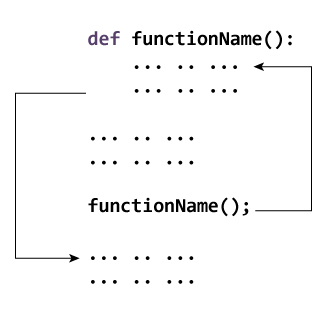
print(absolute\_value(-4))

**Output**

2

4

## How Function works in Python?

Working of functions in Python

## Scope and Lifetime of variables

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

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

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

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

def my\_func():

x = 10

print("Value inside function:",x)

x = 20

my\_func()

print("Value outside function:",x)

**Output**

Value inside function: 10

Value outside function: 20

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

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

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

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

## Types of Functions

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

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

# Python Function Arguments

#### **In Python, you can define a function that takes variable number of arguments. In this article, you will learn to define such functions using default, keyword and arbitrary arguments.**

## Arguments

In the [user-defined function](https://www.programiz.com/python-programming/user-defined-function) topic, we learned about defining a function and calling it. Otherwise, the function call will result in an error. Here is an example.

def greet(name, msg):

"""This function greets to

the person with the provided message"""

print("Hello", name + ', ' + msg)

greet("Monica", "Good morning!")

**Output**

Hello Monica, Good morning!

Here, the function greet() has two parameters.

Since we have called this function with two arguments, it runs smoothly and we do not get any error.

If we call it with a different number of arguments, the interpreter will show an error message. Below is a call to this function with one and no arguments along with their respective error messages.

>>> greet("Monica") # only one argument

TypeError: greet() missing 1 required positional argument: 'msg'

>>> greet() # no arguments

TypeError: greet() missing 2 required positional arguments: 'name' and 'msg'

## Variable Function Arguments

Up until now, functions had a fixed number of arguments. In Python, there are other ways to define a function that can take variable number of arguments.

Three different forms of this type are described below.

### Python Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=). Here is an example.

def greet(name, msg="Good morning!"):

"""

This function greets to

the person with the

provided message.

If the message is not provided,

it defaults to "Good

morning!"

"""

print("Hello", name + ', ' + msg)

greet("Kate")

greet("Bruce", "How do you do?")

**Output**

Hello Kate, Good morning!

Hello Bruce, How do you do?

In this function, the parameter name does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter msg has a default value of "Good morning!". So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

def greet(msg = "Good morning!", name):

We would get an error as:

SyntaxError: non-default argument follows default argument

### Python Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

For example, in the above function greet(), when we called it as greet("Bruce", "How do you do?"), the value "Bruce" gets assigned to the argument name and similarly "How do you do?" to msg.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. Following calls to the above function are all valid and produce the same result.

# 2 keyword arguments

greet(name = "Bruce",msg = "How do you do?")

# 2 keyword arguments (out of order)

greet(msg = "How do you do?",name = "Bruce")

1 positional, 1 keyword argument

greet("Bruce", msg = "How do you do?")

As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.

Having a positional argument after keyword arguments will result in errors. For example, the function call as follows:

greet(name="Bruce","How do you do?")

Will result in an error:

SyntaxError: non-keyword arg after keyword arg

Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

In the function definition, we use an asterisk (\*) before the parameter name to denote this kind of argument. Here is an example.

def greet(\*names):

"""This function greets all

the person in the names tuple."""

# names is a tuple with arguments

for name in names:

print("Hello", name)

greet("Monica", "Luke", "Steve", "John")

**Output**

Hello Monica

Hello Luke

Hello Steve

Hello John

Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a for loop to retrieve all the arguments back.

# Python Recursion

#### **In this tutorial, you will learn to create a recursive function (a function that calls itself).**

## What is recursion?

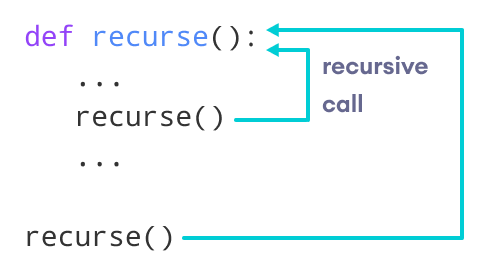
Recursion is the process of defining something in terms of itself.

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

## Python Recursive Function

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

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

Recursive Function in Python

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

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

### Example of a recursive function

def factorial(x):

"""This is a recursive function

to find the factorial of an integer"""

if x == 1:

return 1

else:

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

num = 3

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

**Output**

The factorial of 3 is 6

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

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

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

factorial(3) # 1st call with 3

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

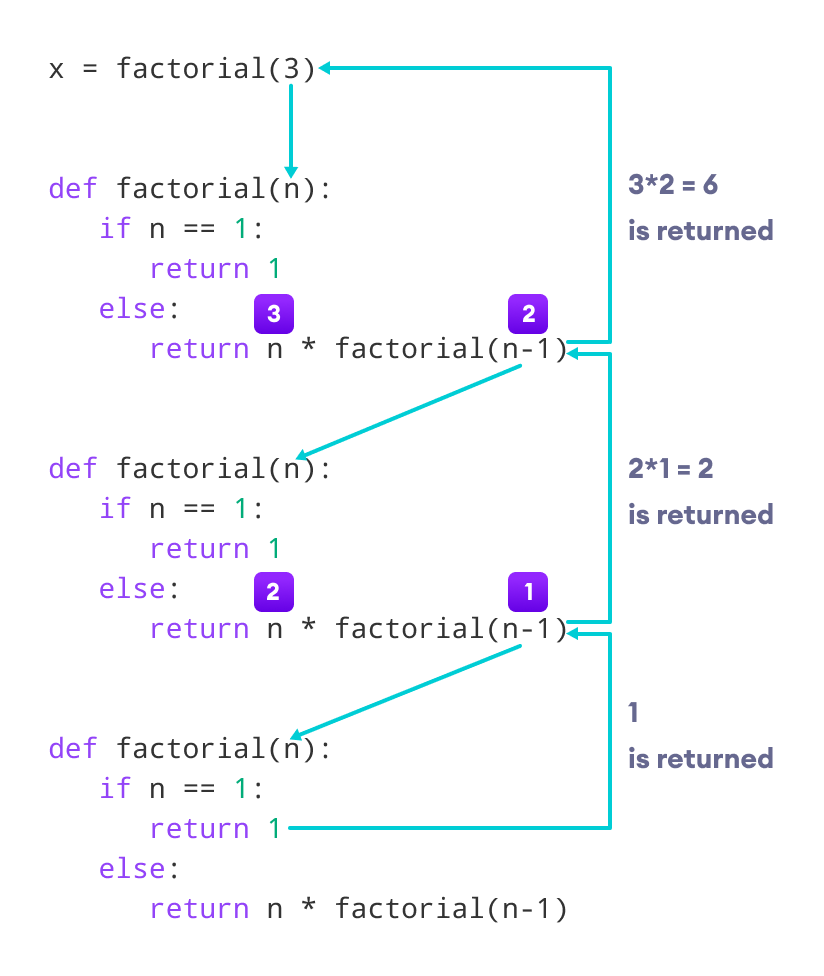
3 \* 2 \* factorial(1) # 3rd call with 1

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

3 \* 2 # return from 2nd call

6 # return from 1st call

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

Working of a recursive factorial function

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

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

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

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

def recursor():

recursor()

recursor()

**Output**

Traceback (most recent call last):

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

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

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

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

[Previous line repeated 996 more times]

RecursionError: maximum recursion depth exceeded

## Advantages of Recursion

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

Disadvantages of Recursion

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

# Python Anonymous/Lambda Function

#### **In this article, you'll learn about the anonymous function, also known as lambda functions. You'll learn what they are, their syntax and how to use them (with examples).**

## What are lambda functions in Python?

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

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

Hence, anonymous functions are also called lambda functions.

## How to use lambda Functions in Python?

A lambda function in python has the following syntax.

### Syntax of Lambda Function in python

lambda arguments: expression

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

### Example of Lambda Function in python

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

# Program to show the use of lambda functions

double = lambda x: x \* 2

print(double(5))

**Output**

10

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

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

double = lambda x: x \* 2

is nearly the same as:

def double(x):

return x \* 2

## Use of Lambda Function in python

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

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

### Example use with filter()

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

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

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

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

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

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

print(new\_list)

**Output**

[4, 6, 8, 12]

### Example use with map()

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

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

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

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

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

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

print(new\_list)

**Output**

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

# Python Global, Local and Nonlocal variables

#### **In this tutorial, you’ll learn about Python Global variables, Local variables, Nonlocal variables and where to use them.**

## Global Variables

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

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

### Example 1: Create a Global Variable

x = "global"

def foo():

print("x inside:", x)

foo()

print("x outside:", x)

**Output**

x inside: global

x outside: global

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

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

x = "global"

def foo():

x = x \* 2

print(x)

foo()

**Output**

UnboundLocalError: local variable 'x' referenced before assignment

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

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

## Local Variables

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

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

def foo():

y = "local"

foo()

print(y)

**Output**

NameError: name 'y' is not defined

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

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

### Example 3: Create a Local Variable

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

def foo():

y = "local"

print(y)

foo()

**Output**

local

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

## Global and local variables

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

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

x = "global "

def foo():

global x

y = "local"

x = x \* 2

print(x)

print(y)

foo()

**Output**

global global

local

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

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

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

x = 5

def foo():

x = 10

print("local x:", x)

foo()

print("global x:", x)

**Output**

local x: 10

global x: 5

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

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

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

## Nonlocal Variables

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

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

We use nonlocal keywords to create nonlocal variables.

### Example 6: Create a nonlocal variable

def outer():

x = "local"

def inner():

nonlocal x

x = "nonlocal"

print("inner:", x)

inner()

print("outer:", x)

outer()

**Output**

inner: nonlocal

outer: nonlocal

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

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

# Python Global Keyword

#### **In this article, you’ll learn about the global keyword, global variable and when to use global keywords.**

## What is the global keyword

In Python, global keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

## Rules of global Keyword

The basic rules for global keyword in Python are:

* When we create a variable inside a function, it is local by default.
* When we define a variable outside of a function, it is global by default. You don't have to use global keyword.
* We use global keyword to read and write a global variable inside a function.
* Use of global keyword outside a function has no effect.

## Use of global Keyword

Let's take an example.

#### **Example 1: Accessing global Variable From Inside a Function**

c = 1 # global variable

def add():

print(c)

add()

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

1

However, we may have some scenarios where we need to modify the global variable from inside a function.

#### **Example 2: Modifying Global Variable From Inside the Function**

c = 1 # global variable

def add():

c = c + 2 # increment c by 2

print(c)

add()

When we run the above program, the output shows an error:

UnboundLocalError: local variable 'c' referenced before assignment

This is because we can only access the global variable but cannot modify it from inside the function.

The solution for this is to use the global keyword.

#### **Example 3: Changing Global Variable From Inside a Function using global**

c = 0 # global variable

def add():

global c

c = c + 2 # increment by 2

print("Inside add():", c)

add()

print("In main:", c)

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

Inside add(): 2

In main: 2

In the above program, we define c as a global keyword inside the add() function.

Then, we increment the variable c by 1, i.e c = c + 2. After that, we call the add() function. Finally, we print the global variable c.

As we can see, change also occurred on the global variable outside the function, c = 2.

## Global Variables Across Python Modules

# Python Modules

#### **In this article, you will learn to create and import custom modules in Python. Also, you will find different techniques to import and use custom and built-in modules in Python.**

## What are modules in Python?

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for example: example.py, is called a module, and its module name would be example.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as example.py.

# Python Module example

def add(a, b):

"""This program adds two

numbers and return the result"""

result = a + b

return result

Here, we have defined a [function](https://www.programiz.com/python-programming/function) add() inside a module named example. The function takes in two numbers and returns their sum.

## How to import modules in Python?

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.

>>> import example

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there.

Using the module name we can access the function using the dot . operator. For example:

>>> example.add(4,5.5)

9.5

### Python import statement

We can import a module using the import statement and access the definitions inside it using the dot operator as described above. Here is an example.

# import statement example

# to import standard module math

import math

print("The value of pi is", math.pi)

When you run the program, the output will be:

The value of pi is 3.141592653589793

### Import with renaming

We can import a module by renaming it as follows:

# import module by renaming it

import math as m

print("The value of pi is", m.pi)

We have renamed the math module as m. This can save us typing time in some cases.

Note that the name math is not recognized in our scope. Hence, math.pi is invalid, and m.pi is the correct implementation.

### Python from...import statement

We can import specific names from a module without importing the module as a whole. Here is an example.

# import only pi from math module

from math import pi

print("The value of pi is", pi)

Here, we imported only the pi attribute from the math module.

In such cases, we don't use the dot operator. We can also import multiple attributes as follows:

>>> from math import pi, e

>>> pi

3.141592653589793

>>> e

2.718281828459045

### Import all names

We can import all names(definitions) from a module using the following construct:

# import all names from the standard module math

from math import \*

print("The value of pi is", pi)

Here, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

## Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in sys.path. The search is in this order.

* The current directory.
* PYTHONPATH (an environment variable with a list of directories).
* The installation-dependent default directory.

>>> import sys

>>> sys.path

['',

'C:\\Python33\\Lib\\idlelib',

'C:\\Windows\\system32\\python33.zip',

'C:\\Python33\\DLLs',

'C:\\Python33\\lib',

'C:\\Python33',

'C:\\Python33\\lib\\site-packages']

We can add and modify this list to add our own path.

## Reloading a module

The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.

Suppose we have the following code in a module named my\_module.

# This module shows the effect of

# multiple imports and reload

print("This code got executed")

Now we see the effect of multiple imports.

>>> import my\_module

This code got executed

>>> import my\_module

>>> import my\_module

We can see that our code got executed only once. This goes to say that our module was imported only once.

Now if our module changed during the course of the program, we would have to reload it.One way to do this is to restart the interpreter. But this does not help much.

Python provides a more efficient way of doing this. We can use the reload() function inside the imp module to reload a module. We can do it in the following ways:

>>> import imp

>>> import my\_module

This code got executed

>>> import my\_module

>>> imp.reload(my\_module)

This code got executed

<module 'my\_module' from '.\\my\_module.py'>

## The dir() built-in function

We can use the dir() function to find out names that are defined inside a module.

For example, we have defined a function add() in the module example that we had in the beginning.

We can use dir in example module in the following way:

>>> dir(example)

['\_\_builtins\_\_',

'\_\_cached\_\_',

'\_\_doc\_\_',

'\_\_file\_\_',

'\_\_initializing\_\_',

'\_\_loader\_\_',

'\_\_name\_\_',

'\_\_package\_\_',

'add']

Here, we can see a sorted list of names (along with add). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

For example, the \_\_name\_\_ attribute contains the name of the module.

>>> import example

>>> example.\_\_name\_\_

'example'

All the names defined in our current namespace can be found out using the dir() function without any arguments.

>>> a = 1

>>> b = "hello"

>>> import math

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'a', 'b', 'math', 'pyscripter']

# Python Package

#### **In this article, you'll learn to divide your code base into clean, efficient modules using Python packages. Also, you'll learn to import and use your own or third party packages in your Python program.**

## What are packages?

We don't usually store all of our files on our computer in the same location. We use a well-organized hierarchy of directories for easier access.

Similar files are kept in the same directory, for example, we may keep all the songs in the "**music**" directory. Analogous to this, Python has packages for directories and [modules](https://www.programiz.com/python-programming/modules) for files.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.

A directory must contain a file named \_\_init\_\_.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

Here is an example. Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.

Package Module Structure in Python Programming

## Importing module from a package

We can import modules from packages using the dot (.) operator.

For example, if we want to import the start module in the above example, it can be done as follows:

import Game.Level.start

Now, if this module contains a [function](https://www.programiz.com/python-programming/function) named select\_difficulty(), we must use the full name to reference it.

Game.Level.start.select\_difficulty(2)

If this construct seems lengthy, we can import the module without the package prefix as follows:

from Game.Level import start

We can now call the function simply as follows:

start.select\_difficulty(2)

Another way of importing just the required function (or class or variable) from a module within a package would be as follows:

from Game.Level.start import select\_difficulty

Now we can directly call this function.

select\_difficulty(2)

Although easier, this method is not recommended. Using the full [namespace](https://www.programiz.com/python-programming/namespace) avoids confusion and prevents two same identifier names from colliding.

While importing packages, Python looks in the list of directories defined in sys.path, similar as for [module search path](https://www.programiz.com/python-programming/modules#search).

# Python File I/O

#### **In this tutorial, you'll learn about Python file operations. More specifically, opening a file, reading from it, writing into it, closing it, and various file methods that you should be aware of.**

## Files

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).

Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (perform operation)
3. Close the file

## Opening Files in Python

Python has a built-in open() function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

>>> f = open("test.txt") # open file in current directory

>>> f = open("C:/Python38/README.txt") # specifying full path

We can specify the mode while opening a file. In mode, we specify whether we want to read r, write w or append a to the file. We can also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

|  |  |
| --- | --- |
| Mode | Description |
| r | Opens a file for reading. (default) |
| w | Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists. |
| x | Opens a file for exclusive creation. If the file already exists, the operation fails. |
| a | Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist. |
| t | Opens in text mode. (default) |
| b | Opens in binary mode. |
| + | Opens a file for updating (reading and writing) |

f = open("test.txt") # equivalent to 'r' or 'rt'

f = open("test.txt",'w') # write in text mode

f = open("img.bmp",'r+b') # read and write in binary mode

Unlike other languages, the character a does not imply the number 97 until it is encoded using ASCII (or other equivalent encodings).

Moreover, the default encoding is platform dependent. In windows, it is cp1252 but utf-8 in Linux.

So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

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

## Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file. It is done using the close() method available in Python.

Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

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

# perform file operations

f.close()

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a [try...finally](https://www.programiz.com/python-programming/exception-handling) block.

try:

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

# perform file operations

finally:

f.close()

This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.

The best way to close a file is by using the with statement. This ensures that the file is closed when the block inside the with statement is exited.

We don't need to explicitly call the close() method. It is done internally.

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

# perform file operations

## 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.

## Reading Files in Python

To read a file in Python, we must open the file in reading r mode.

There are various methods available for this purpose. We can use the read(size) method to read in the size number of data. If the size parameter is not specified, it reads and returns up to the end of the file.

We can read the text.txt file we wrote in the above section in the following way:

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

>>> f.read(4) # read the first 4 data

'This'

>>> f.read(4) # read the next 4 data

' is '

>>> f.read() # read in the rest till end of file

'my first file\nThis file\ncontains three lines\n'

>>> f.read() # further reading returns empty sting

''

We can see that the read() method returns a newline as '\n'. Once the end of the file is reached, we get an empty string on further reading.

We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).

>>> f.tell() # get the current file position

56

>>> f.seek(0) # bring file cursor to initial position

0

>>> print(f.read()) # read the entire file

This is my first file

This file

contains three lines

We can read a file line-by-line using a [for loop](https://www.programiz.com/python-programming/for-loop). This is both efficient and fast.

>>> for line in f:

... print(line, end = '')

...

This is my first file

This file

contains three lines

In this program, the lines in the file itself include a newline character \n. So, we use the end parameter of the print() function to avoid two newlines when printing.

Alternatively, we can use the readline() method to read individual lines of a file. This method reads a file till the newline, including the newline character.

>>> f.readline()

'This is my first file\n'

>>> f.readline()

'This file\n'

>>> f.readline()

'contains three lines\n'

>>> f.readline()

''

Lastly, the readlines() method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

>>> f.readlines()

['This is my first file\n', 'This file\n', 'contains three lines\n']

## Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

Here is the complete list of methods in text mode with a brief description:

|  |  |
| --- | --- |
| Method | Description |
| close() | Closes an opened file. It has no effect if the file is already closed. |
| detach() | Separates the underlying binary buffer from the TextIOBase and returns it. |
| fileno() | Returns an integer number (file descriptor) of the file. |
| flush() | Flushes the write buffer of the file stream. |
| isatty() | Returns True if the file stream is interactive. |
| read(n) | Reads at most n characters from the file. Reads till end of file if it is negative or None. |
| readable() | Returns True if the file stream can be read from. |
| readline(n=-1) | Reads and returns one line from the file. Reads in at most n bytes if specified. |
| readlines(n=-1) | Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified. |
| seek(offset,from=SEEK\_SET) | Changes the file position to offset bytes, in reference to from (start, current, end). |
| seekable() | Returns True if the file stream supports random access. |
| tell() | Returns the current file location. |
| truncate(size=None) | Resizes the file stream to size bytes. If size is not specified, resizes to current location. |
| writable() | Returns True if the file stream can be written to. |
| write(s) | Writes the string s to the file and returns the number of characters written. |
| writelines(lines) | Writes a list of lines to the file. |

## Python Directory

If there are a large number of [files](https://www.programiz.com/python-programming/file-operation) to handle in our Python program, we can arrange our code within different directories to make things more manageable.

A directory or folder is a collection of files and subdirectories. Python has the os [module](https://www.programiz.com/python-programming/modules) that provides us with many useful methods to work with directories (and files as well).

## Get Current Directory

We can get the present working directory using the getcwd() method of the os module.

This method returns the current working directory in the form of a string. We can also use the getcwdb() method to get it as bytes object.

>>> import os

>>> os.getcwd()

'C:\\Program Files\\PyScripter'

>>> os.getcwdb()

b'C:\\Program Files\\PyScripter'

The extra backslash implies an escape sequence. The print() function will render this properly.

>>> print(os.getcwd())

C:\Program Files\PyScripter

## Changing Directory

We can change the current working directory by using the chdir() method.

The new path that we want to change into must be supplied as a string to this method. We can use both the forward-slash / or the backward-slash \ to separate the path elements.

It is safer to use an escape sequence when using the backward slash.

>>> os.chdir('C:\\Python33')

>>> print(os.getcwd())

C:\Python33

## List Directories and Files

All files and sub-directories inside a directory can be retrieved using the listdir() method.

## Making a New Directory

We can make a new directory using the mkdir() method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

>>> os.mkdir('test')

>>> os.listdir()

['test']

## Renaming a Directory or a File

The rename() method can rename a directory or a file.

For renaming any directory or file, the rename() method takes in two basic arguments: the old name as the first argument and the new name as the second argument.

>>> os.listdir()

['test']

>>> os.rename('test','new\_one')

>>> os.listdir()

['new\_one']

## Removing Directory or File

A file can be removed (deleted) using the remove() method.

Similarly, the rmdir() method removes an empty directory.

>>> os.listdir()

['new\_one', 'old.txt']

>>> os.remove('old.txt')

>>> os.listdir()

['new\_one']

>>> os.rmdir('new\_one')

>>> os.listdir()

[]

**Note**: The rmdir() method can only remove empty directories.

In order to remove a non-empty directory, we can use the rmtree() method inside the shutil module.

>>> os.listdir()

['test']

>>> os.rmdir('test')

Traceback (most recent call last):

...

OSError: [WinError 145] The directory is not empty: 'test'

>>> import shutil

>>> shutil.rmtree('test')

>>> os.listdir()

[]

# Python Object Oriented Programming

#### **In this tutorial, you’ll learn about Object-Oriented Programming (OOP) in Python and its fundamental concept with the help of examples.**

## Object Oriented Programming

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

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

An object has two characteristics:

* attributes
* behavior

Let's take an example:

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

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

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

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

## Class

A class is a blueprint for the object.

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

The example for class of parrot can be :

class Parrot:

pass

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

## Object

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

The example for object of parrot class can be:

obj = Parrot()

Here, obj is an object of class Parrot.

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

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

class Parrot:

# class attribute

species = "bird"

# instance attribute

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

self.name = name

self.age = age

# instantiate the Parrot class

blu = Parrot("Blu", 10)

woo = Parrot("Woo", 15)

# access the class attributes

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

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

# access the instance attributes

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

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

**Output**

Blu is a bird

Woo is also a bird

Blu is 10 years old

Woo is 15 years old

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

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

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

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

## Methods

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

### Example 2 : Creating Methods in Python

class Parrot:

# instance attributes

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

self.name = name

self.age = age

# instance method

def sing(self, song):

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

def dance(self):

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

# instantiate the object

blu = Parrot("Blu", 10)

# call our instance methods

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

print(blu.dance())

**Output**

Blu sings 'Happy'

Blu is now dancing

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

## Inheritance

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

### Example 3: Use of Inheritance in Python

# parent class

class Bird:

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

print("Bird is ready")

def whoisThis(self):

print("Bird")

def swim(self):

print("Swim faster")

# child class

class Penguin(Bird):

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

# call super() function

super().\_\_init\_\_()

print("Penguin is ready")

def whoisThis(self):

print("Penguin")

def run(self):

print("Run faster")

peggy = Penguin() #instantiate the object of child class

peggy.whoisThis()

peggy.swim()

peggy.run()

**Output**

Bird is ready

Penguin is ready

Penguin

Swim faster

Run faster

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the swim() method.

Again, the child class modified the behavior of the parent class. We can see this from the whoisThis() method. Furthermore, we extend the functions of the parent class, by creating a new run() method.

Additionally, we use the super() function inside the \_\_init\_\_() method. This allows us to run the \_\_init\_\_() method of the parent class inside the child class.

## Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single \_ or double \_\_.

### Example 4: Data Encapsulation in Python

class Computer:

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

self.\_\_maxprice = 900

def sell(self):

print("Selling Price: {}".format(self.\_\_maxprice))

def setMaxPrice(self, price):

self.\_\_maxprice = price

c = Computer()

c.sell()

# change the price

c.\_\_maxprice = 1000

c.sell()

# using setter function

c.setMaxPrice(1000)

c.sell()

**Output**

Selling Price: 900

Selling Price: 900

Selling Price: 1000

In the above program, we defined a Computer class.

We used \_\_init\_\_() method to store the maximum selling price of Computer. We tried to modify the price. However, we can't change it because Python treats the \_\_maxprice as private attributes.

As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

## Polymorphism

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

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

### Example 5: Using Polymorphism in Python

class Parrot:

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin:

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

flying\_test(peggy)

**Output**

Parrot can fly

Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have a common fly() method. However, their functions are different.

To use polymorphism, we created a common interface i.e flying\_test() function that takes any object and calls the object's fly() method. Thus, when we passed the blu and peggy objects in the flying\_test() function, it ran effectively.

## Key Points to Remember:

* Object-Oriented Programming makes the program easy to understand as well as efficient.
* Since the class is sharable, the code can be reused.
* Data is safe and secure with data abstraction.
* Polymorphism allows the same interface for different objects, so programmers can write efficient code.

## Python Objects and Classes

Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stresses on objects.

An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

As many houses can be made from a house's blueprint, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

## Defining a Class in Python

Like function definitions begin with the [def](https://www.programiz.com/python-programming/keyword-list#def) keyword in Python, class definitions begin with a [class](https://www.programiz.com/python-programming/keyword-list#class) keyword.

The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

class MyNewClass:

'''This is a docstring. I have created a new class'''

pass

A class creates a new local [namespace](https://www.programiz.com/python-programming/namespace) where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores \_\_. For example, \_\_doc\_\_ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

class Person:

"This is a person class"

age = 10

def greet(self):

print('Hello')

# Output: 10

print(Person.age)

# Output: <function Person.greet>

print(Person.greet)

# Output: 'This is my second class'

print(Person.\_\_doc\_\_)

**Output**

10

<function Person.greet at 0x7fc78c6e8160>

This is a person class

## Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a [function](https://www.programiz.com/python-programming/function) call.

>>> harry = Person()

This will create a new object instance named harry. We can access the attributes of objects using the object name prefix.

Attributes may be data or method. Methods of an object are corresponding functions of that class.

This means to say, since Person.greet is a function object (attribute of class), Person.greet will be a method object.

class Person:

"This is a person class"

age = 10

def greet(self):

print('Hello')

# create a new object of Person class

harry = Person()

# Output: <function Person.greet>

print(Person.greet)

# Output: <bound method Person.greet of <\_\_main\_\_.Person object>>

print(harry.greet)

# Calling object's greet() method

# Output: Hello

harry.greet()

**Output**

<function Person.greet at 0x7fd288e4e160>

<bound method Person.greet of <\_\_main\_\_.Person object at 0x7fd288e9fa30>>

Hello

You may have noticed the self parameter in function definition inside the class but we called the method simply as harry.greet() without any [arguments](https://www.programiz.com/python-programming/function-argument). It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, harry.greet() translates into Person.greet(harry).

In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

For these reasons, the first argument of the function in class must be the object itself. This is conventionally called self. It can be named otherwise but we highly recommend to follow the convention.

Now you must be familiar with class object, instance object, function object, method object and their differences.

## Constructors in Python

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

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

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

class ComplexNumber:

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

self.real = r

self.imag = i

def get\_data(self):

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

# Create a new ComplexNumber object

num1 = ComplexNumber(2, 3)

# Call get\_data() method

# Output: 2+3j

num1.get\_data()

# Create another ComplexNumber object

# and create a new attribute 'attr'

num2 = ComplexNumber(5)

num2.attr = 10

# Output: (5, 0, 10)

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

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

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

print(num1.attr)

**Output**

2+3j

(5, 0, 10)

Traceback (most recent call last):

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

print(num1.attr)

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

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

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

## Deleting Attributes and Objects

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

>>> num1 = ComplexNumber(2,3)

>>> del num1.imag

>>> num1.get\_data()

Traceback (most recent call last):

...

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

>>> del ComplexNumber.get\_data

>>> num1.get\_data()

Traceback (most recent call last):

...

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

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

>>> c1 = ComplexNumber(1,3)

>>> del c1

>>> c1

Traceback (most recent call last):

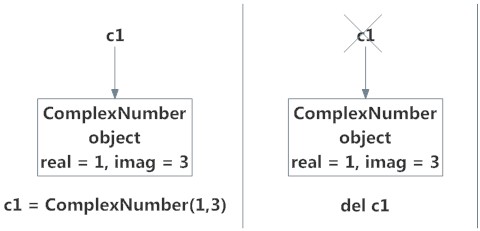
...

NameError: name 'c1' is not defined

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

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

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



Deleting objects in Python removes the name binding

# Python Inheritance

#### **Inheritance enables us to define a class that takes all the functionality from a parent class and allows us to add more. In this tutorial, you will learn to use inheritance in Python.**

## Inheritance in Python

Inheritance is a powerful feature in object oriented programming.

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

### Python Inheritance Syntax

class BaseClass:

Body of base class

class DerivedClass(BaseClass):

Body of derived class

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

### Example of Inheritance in Python

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

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

class Polygon:

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

self.n = no\_of\_sides

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

def inputSides(self):

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

def dispSides(self):

for i in range(self.n):

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

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

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

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

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

class Triangle(Polygon):

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

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

def findArea(self):

a, b, c = self.sides

# calculate the semi-perimeter

s = (a + b + c) / 2

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

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

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

>>> t = Triangle()

>>> t.inputSides()

Enter side 1 : 3

Enter side 2 : 5

Enter side 3 : 4

>>> t.dispSides()

Side 1 is 3.0

Side 2 is 5.0

Side 3 is 4.0

>>> t.findArea()

The area of the triangle is 6.00

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

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

## Method Overriding in Python

In the above example, notice that \_\_init\_\_() method was defined in both classes, Triangle as well Polygon. When this happens, the method in the derived class overrides that in the base class. This is to say, \_\_init\_\_() in Triangle gets preference over the \_\_init\_\_ in Polygon.

Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling Polygon.\_\_init\_\_() from \_\_init\_\_() in Triangle).

A better option would be to use the built-in function super(). So, super().\_\_init\_\_(3) is equivalent to Polygon.\_\_init\_\_(self,3) and is preferred. To learn more about the super() function in Python, visit [Python super() function](http://rhettinger.wordpress.com/2011/05/26/super-considered-super/).

Two built-in functions isinstance() and issubclass() are used to check inheritances.

The function isinstance() returns True if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

>>> isinstance(t,Triangle)

True

>>> isinstance(t,Polygon)

True

>>> isinstance(t,int)

False

>>> isinstance(t,object)

True

Similarly, issubclass() is used to check for class inheritance.

>>> issubclass(Polygon,Triangle)

False

>>> issubclass(Triangle,Polygon)

True

>>> issubclass(bool,int)

True

## Python Multiple Inheritance

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

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

### Example

class Base1:

pass

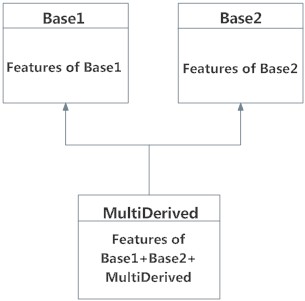
class Base2:

pass

class MultiDerived(Base1, Base2):

pass

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

Multiple Inheritance in Python

The MultiDerived class inherits from both Base1 and Base2 classes.

## Python Multilevel Inheritance

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

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

An example with corresponding visualization is given below.

class Base:

pass

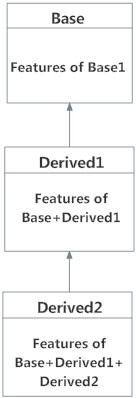
class Derived1(Base):

pass

class Derived2(Derived1):

pass

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

Multilevel Inheritance in Python

## Python Operator Overloading

[Python operators](https://www.programiz.com/python-programming/operators) work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

class Point:

def \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

p1 = Point(1, 2)

p2 = Point(2, 3)

print(p1+p2)

**Output**

Traceback (most recent call last):

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

print(p1+p2)

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

Here, we can see that a TypeError was raised, since Python didn't know how to add two Point objects together.

However, we can achieve this task in Python through operator overloading. But first, let's get a notion about special functions.

## Python Special Functions

Class functions that begin with double underscore \_\_ are called special functions in Python.

These functions are not the typical functions that we define for a class. The \_\_init\_\_() function we defined above is one of them. It gets called every time we create a new object of that class.

There are numerous other special functions in Python. Visit [Python Special Functions](http://docs.python.org/3/reference/datamodel.html#special-method-names) to learn more about them.

Using special functions, we can make our class compatible with built-in functions.

>>> p1 = Point(2,3)

>>> print(p1)

<\_\_main\_\_.Point object at 0x00000000031F8CC0>

Suppose we want the print() function to print the coordinates of the Point object instead of what we got. We can define a \_\_str\_\_() method in our class that controls how the object gets printed. Let's look at how we can achieve this:

class Point:

def \_\_init\_\_(self, x = 0, y = 0):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "({0},{1})".format(self.x,self.y)

p1 = Point(2, 3)

print(p1)

**Output**

(2, 3)

That's better. Turns out, that this same method is invoked when we use the built-in function str() or format().

>>> str(p1)

'(2,3)'

>>> format(p1)

'(2,3)'

So, when you use str(p1) or format(p1), Python internally calls the p1.\_\_str\_\_() method. Hence the name, special functions.

Now let's go back to operator overloading.

## Overloading the + Operator

To overload the + operator, we will need to implement \_\_add\_\_() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is more sensible to return a Point object of the coordinate sum.

class Point:

def \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "({0},{1})".format(self.x, self.y)

def \_\_add\_\_(self, other):

x = self.x + other.x

y = self.y + other.y

return Point(x, y)

Now let's try the addition operation again:

class Point:

def \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "({0},{1})".format(self.x, self.y)

def \_\_add\_\_(self, other):

x = self.x + other.x

y = self.y + other.y

return Point(x, y)

p1 = Point(1, 2)

p2 = Point(2, 3)

print(p1+p2)

**Output**

(3,5)

What actually happens is that, when you use p1 + p2, Python calls p1.\_\_add\_\_(p2) which in turn is Point.\_\_add\_\_(p1,p2). After this, the addition operation is carried out the way we specified.

Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

|  |  |  |
| --- | --- | --- |
| Operator | Expression | Internally |
| Addition | p1 + p2 | p1.\_\_add\_\_(p2) |
| Subtraction | p1 - p2 | p1.\_\_sub\_\_(p2) |
| Multiplication | p1 \* p2 | p1.\_\_mul\_\_(p2) |
| Power | p1 \*\* p2 | p1.\_\_pow\_\_(p2) |
| Division | p1 / p2 | p1.\_\_truediv\_\_(p2) |
| Floor Division | p1 // p2 | p1.\_\_floordiv\_\_(p2) |
| Remainder (modulo) | p1 % p2 | p1.\_\_mod\_\_(p2) |
| Bitwise Left Shift | p1 << p2 | p1.\_\_lshift\_\_(p2) |
| Bitwise Right Shift | p1 >> p2 | p1.\_\_rshift\_\_(p2) |
| Bitwise AND | p1 & p2 | p1.\_\_and\_\_(p2) |
| Bitwise OR | p1 | p2 | p1.\_\_or\_\_(p2) |
| Bitwise XOR | p1 ^ p2 | p1.\_\_xor\_\_(p2) |
| Bitwise NOT | ~p1 | p1.\_\_invert\_\_() |

## Overloading Comparison Operators

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.

Suppose we wanted to implement the less than symbol < symbol in our Point class.

Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

# overloading the less than operator

class Point:

def \_\_init\_\_(self, x=0, y=0):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "({0},{1})".format(self.x, self.y)

def \_\_lt\_\_(self, other):

self\_mag = (self.x \*\* 2) + (self.y \*\* 2)

other\_mag = (other.x \*\* 2) + (other.y \*\* 2)

return self\_mag < other\_mag

p1 = Point(1,1)

p2 = Point(-2,-3)

p3 = Point(1,-1)

# use less than

print(p1<p2)

print(p2<p3)

print(p1<p3)

**Output**

True

False

False

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

|  |  |  |
| --- | --- | --- |
| Operator | Expression | Internally |
| Less than | p1 < p2 | p1.\_\_lt\_\_(p2) |
| Less than or equal to | p1 <= p2 | p1.\_\_le\_\_(p2) |
| Equal to | p1 == p2 | p1.\_\_eq\_\_(p2) |
| Not equal to | p1 != p2 | p1.\_\_ne\_\_(p2) |
| Greater than | p1 > p2 | p1.\_\_gt\_\_(p2) |
| Greater than or equal to | p1 >= p2 | p1.\_\_ge\_\_(p2) |