Introduction

Getting started

Python Keywords and identifiers

Python Statements & Comments

Python variables

Python Data types

Python type Conversion

Python I/O and import

Python Operators

Python Name space

Keywords and identifiers

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

Python Keywords

Keywords are predefined, reserved words used in Python programming that have special meanings to the compiler.

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.

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.

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

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

If you want to have an overview, here is the complete [list of all the keywords](https://www.programiz.com/python-programming/keyword-list) with examples.

Python Identifiers

Identifiers are the name given to variables, classes, methods, etc. For example,

language = 'Python'

Here, language is a variable (an identifier) which holds the value 'Python'.

We cannot use keywords as variable names as they are reserved names that are built-in to Python. For example,

continue = 'Python'

The above code is wrong because we have used continue as a variable name. To learn more about variables, visit [Python Variables](https://www.programiz.com/python-programming/variables-constants-literals).

Rules for Naming an Identifier

Identifiers cannot be a keyword.

Identifiers are case-sensitive.

It can have a sequence of letters and digits. However, it must begin with a letter or \_. The first letter of an identifier cannot be a digit.

It's a convention to start an identifier with a letter rather \_.

Whitespaces are not allowed.

We cannot use special symbols like !, @, #, $, and so on.

Some Valid and Invalid Identifiers in Python

|  |  |
| --- | --- |
| Valid Identifiers | Invalid Identifiers |
| score | @core |
| return\_value | return |
| highest\_score | highest score |
| name1 | 1name |
| convert\_to\_string | convert to\_string |

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 Statements & Comments

In this tutorial, we will learn about Python statements, why we use them, and how to use comments in the right way.

In computer programming, comments are hints that we use to make our code more understandable.

Comments are completely ignored by the interpreter. They are meant for fellow programmers. For example,

# declare and initialize two variables

num1 = 6

num2 = 9# print the output

print('This is output')

Here, we have used the following comments,

declare and initialize two variables

print the output

Types of Comments in Python

In Python, there are two types of comments:

single-line comment

multi-line comment

Single-line Comment in Python

A single-line comment starts and ends in the same line. We use the # symbol to write a single-line comment. For example,

# create a variable

name = 'Eric Cartman'

# print the value

print(name)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

Eric Cartman

Here, we have created two single-line comments:

# create a variable

# print the value

We can also use the single-line comment along with the code.

name = 'Eric Cartman' # name is a string

Here, code before # are executed and code after # are ignored by the interpreter.

Multi-line Comment in Python

Python doesn't offer a separate way to write multiline comments. However, there are other ways to get around this issue.

We can use # at the beginning of each line of comment on multiple lines. For example,

''' This is also a

perfect example of

multi-line comments '''

# This is a long comment

# and it extends

# to multiple lines

Here, each line is treated as a single comment, and all of them are ignored.

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

These triple quotes are generally used for multi-line strings. But if we do not assign it to any variable or function, we can use it as a comment.

The interpreter ignores the string that is not assigned to any variable or function.

Let's see an example,

Here, the multiline string isn't assigned to any variable, so it is ignored by the interpreter. Even though it is not technically a multiline comment, it can be used as one.

Use of Python Comment

1. Make Code Easier to Understand

If we write comments in our code, it will be easier for future reference.

Also, it will be easier for other developers to understand the code.

2. Using Comments for Debugging

If we get an error while running the program, we can comment the line of code that causes the error instead of removing it. For example,

print('Python')

# print('Error Line )

print('Django')

Here, print('Error Line) was causing an error so we have changed it to a comment. Now, the program runs without any errors.

This is how comments can be a valuable debugging tool.

Note: Always use comments to explain why we did something rather than how we did something. Comments shouldn't be a substitute to explain poorly written code.

Python variable, constants and literals

In this tutorial, we will learn about Python variables, constants, literals with the help of examples.

Python Variables

In programming, a variable is a container (storage area) to hold data. For example,

number = 10

Here, number is the variable storing the value 10.

Assigning values to Variables in Python

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

[Run Code](https://www.programiz.com/python-programming/online-compiler)

# assign value to site\_name variable

site\_name = 'programiz.pro'

print(site\_name)

# Output: programiz.pro

In the above example, we assigned the value 'programiz.pro' to the site\_name variable. Then, we printed out the value assigned to site\_name.

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 programiz.pro is a string and declares the site\_name variable as a string.

Changing the Value of a Variable in Python

site\_name = 'programiz.pro'

print(site\_name)

# assigning a new value to site\_name

site\_name = 'apple.com'

print(site\_name)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Output

programiz.pro

apple.com

Here, the value of site\_name is changed from 'programiz.pro' to 'apple.com'.

Example: Assigning multiple values to multiple variables

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

print(a) # prints 5

print(b) # prints 3.2

print(c) # prints Hello

Run Code

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

site1 = site2 = 'programiz.com'

print(site1) # prints programiz.com

print(site2) # prints programiz.com

Here, we have assigned the same string value 'programiz.com' to both the variables site1 and site2.

Rules for Naming Python Variables

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:

snake\_case

MACRO\_CASE

camelCase

CapWords

Create a name that makes sense. For example, vowel makes more sense than v.

If you want to create a variable name having two words, use underscore to separate them. For example:

my\_name

current\_salary

Python is case-sensitive. So num and Num are different variables. For example,

var num = 5

var Num = 55

print(num) # 5

print(Num) # 55

Avoid using [keywords](https://www.programiz.com/python-programming/keywords-identifier) like if, True, class, etc. as variable names.

Python Constants

A constant is a special type of variable whose value cannot be changed.

In Python, constants are usually declared and assigned in a [module](https://www.programiz.com/python-programming/modules) (a new file containing variables, functions, etc which is imported to the main file).

Let's see how we declare constants in separate file and use it in the main file,

Create a constant.py:

# declare constants

PI = 3.14

GRAVITY = 9.8

Create a main.py:

# import constant file we created above

import constant

print(constant.PI) # prints 3.14

print(constant.GRAVITY) # prints 9.8

In the above example, we created the constant.py module file. Then, we assigned the constant value to PI and GRAVITY.

After that, we create the main.py file and import the constant module. Finally, we printed 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.

Python Literals

Literals are representations of fixed values in a program. They can be numbers, characters, or strings, etc. For example, 'Hello, World!', 12, 23.0, 'C', etc.

Literals are often used to assign values to variables or constants. For example,

some\_string = 'Python is fun'

some\_character = 'S'

result1 = True

site\_name = 'programiz.com'

In the above expression, site\_name is a variable, and 'programiz.com' is a literal.

Python Numeric Literals

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

|  |  |  |
| --- | --- | --- |
| **Type** | **Example** | **Remarks** |
| **Decimal** | 5, 10, -68 | Regular numbers. |
| **Binary** | 0b101, 0b11 | Start with 0b. |
| **Octal** | 0o13 | Start with 0o. |
| **Hexadecimal** | 0x13 | Start with 0x. |
| **Floating-point Literal** | 10.5, 3.14 | Containing floating decimal points. |
| **Complex Literal** | 6 + 9j | Numerals in the form a + bj, where a is real and b is imaginary part |

Python Boolean Literals

There are two boolean literals: True and False.

For example,

Here, True is a boolean literal assigned to result1.

String and Character Literals in Python

Character literals are unicode characters enclosed in a quote. For example,

Here, S is a character literal assigned to some\_character.

Similarly, String literals are sequences of Characters enclosed in quotation marks.

For example,

Here, 'Python is fun' is a string literal assigned to some\_string.

Special Literal in Python

Python contains one special literal None. We use it to specify a null variable. For example,

value = None

print(value)

# Output: None

Here, we get None as an output as the value variable has no value assigned to it.

Literal Collections

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

# list literal

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

print(fruits)

# tuple literal

numbers = (1, 2, 3)

print(numbers)

# dictionary literal

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

print(alphabets)

# set literal

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

print(vowels)

Run Code

Output

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

(1, 2, 3)

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

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

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

To learn more about literal collections, refer to [Python Data Types](https://www.programiz.com/python-programming/variables-datatypes).

**Python data types**

In this tutorial, you will learn about different data types we can use in Python with the help of examples.

In computer programming, data types specify the type of data that can be stored inside a variable. For example,

num = 24

Here, **24** (an integer) is assigned to the num variable. So the data type of num is of the int class.

Python Data Types

|  |  |  |
| --- | --- | --- |
| Data Types | Classes | Description |
| Numeric | int, float, complex | holds numeric values |
| String | str | holds sequence of characters |
| Sequence | list, tuple, range | holds collection of items |
| Mapping | dict | holds data in key-value pair form |
| Boolean | bool | holds either True or False |
| Set | set, frozeenset | hold collection of unique items |

Since everything is an object in Python programming, data types are actually classes and variables are instances(object) of these classes.

Python Numeric Data type

In Python, numeric data type is used to hold numeric values.

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.

int - holds signed integers of non-limited length.

float - holds floating decimal points and it's accurate up to **15** decimal places.

complex - holds complex numbers.

We can use the type() function to know which class a variable or a value belongs to.

Let's see an example,

num1 = 5

print(num1, 'is of type', type(num1))

num2 = 2.0

print(num2, 'is of type', type(num2))

num3 = 1+2j

print(num3,'is of type', type(num3))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

5 is of type <class 'int'>

2.0 is of type <class 'float'>

(1+2j) is of type <class 'complex'>

In the above example, we have created three variables named num1, num2 and num3 with values **5**, **5.0**, and 1+2j respectively.

We have also used the type() function to know which class a certain variable belongs to.

Since,

**5** is an integer value, type() returns int as the class of num1 i.e <class 'int'>

**2.0** is a floating value, type() returns float as the class of num2 i.e <class 'float'>

1 + 2j is a complex number, type() returns complex as the class of num3 i.e <class 'complex'>

Python List Data Type

List is an ordered collection of similar or different types of items separated by commas and enclosed within brackets [ ]. For example,

languages = ["Swift", "Java", "Python"]

Here, we have created a list named languages with **3** string values inside it.

Access List Items

To access items from a list, we use the index number **(0, 1, 2 ...)**. For example,

languages = ["Swift", "Java", "Python"]

# access element at index 0

print(languages[0]) # Swift

# access element at index 2

print(languages[2]) # Python

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have used the index values to access items from the languages list.

languages[0] - access first item from languages i.e. Swift

languages[2] - access third item from languages i.e. Python

To learn more about lists, visit [Python List](https://www.programiz.com/python-programming/list).

Python Tuple Data Type

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.

In Python, we use the parentheses () to store items of a tuple. For example,

product = ('Xbox', 499.99)

Here, product is a tuple with a string value Xbox and integer value **499.99**.

Access Tuple Items

Similar to lists, we use the index number to access tuple items in Python . For example,

# create a tuple

product = ('Microsoft', 'Xbox', 499.99)

# access element at index 0

print(product[0]) # Microsoft

# access element at index 1

print(product[1]) # Xbox

[Run Code](https://www.programiz.com/python-programming/online-compiler)

To learn more about tuples, visit [Python Tuples](https://www.programiz.com/python-programming/tuple).

Python String Data

TypeString is a sequence of characters represented by either single or double quotes. For example,

name = 'Python'

print(name)

message = 'Python for beginners'

print(message)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Python

Python for beginners

In the above example, we have created string-type variables: name and message with values 'Python' and 'Python for beginners' respectively.

To learn more about strings, visit [Python Strings](https://www.programiz.com/python-programming/string).

Python Set Data Type

Set is an unordered collection of unique items. Set is defined by values separated by commas inside braces { }. For example,

# create a set named student\_id

student\_id = {112, 114, 116, 118, 115}

# display student\_id elements

print(student\_id)

# display type of student\_id

print(type(student\_id))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

{112, 114, 115, 116, 118}

<class 'set'>

Here, we have created a set named student\_info with **5** integer values.

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

To learn more about sets, visit [Python Sets](https://www.programiz.com/python-programming/set).

Python Dictionary Data Type

Python dictionary is an ordered collection of items. It stores elements in key/value pairs.

Here, keys are unique identifiers that are associated with each value.

Let's see an example,

# create a dictionary named capital\_city

capital\_city = {'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

print(capital\_city)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

{'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

In the above example, we have created a dictionary named capital\_city. Here,

**Keys** are 'Nepal', 'Italy', 'England'

**Values** are 'Kathmandu', 'Rome', 'London'

Access Dictionary Values Using Keys

We use keys to retrieve the respective value. But not the other way around. For example,

# create a dictionary named capital\_city

capital\_city = {'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

print(capital\_city['Nepal']) # prints Kathmandu

print(capital\_city['Kathmandu']) # throws error message

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we have accessed values using keys from the capital\_city dictionary.

Since 'Nepal' is key, capital\_city['Nepal'] accesses its respective value i.e. Kathmandu

However, 'Kathmandu' is the value for the 'Nepal' key, so capital\_city['Kathmandu'] throws an error message.

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

# Python Type Conversion

In this tutorial, we will learn about the Python Type conversion with the help of examples.

In programming, type conversion is the process of converting data of one type to another. For example: converting int data to str.

There are two types of type conversion in Python.

* Implicit Conversion - automatic type conversion
* Explicit Conversion - manual type conversion

## Python Implicit Type Conversion

In certain situations, Python automatically converts one data type to another. This is known as implicit type conversion.

## Example 1: Converting integer to float

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.

integer\_number = 123

float\_number = 1.23

new\_number = integer\_number + float\_number

# display new value and resulting data type

print("Value:",new\_number)

print("Data Type:",type(new\_number))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Value: 124.23

Data Type: <class 'float'>

In the above example, we have created two variables: integer\_number and float\_number of int and float type respectively.

Then we added these two variables and stored the result in new\_number.

As we can see new\_number has value **124.23** and is of the float data type.

It is because Python always converts smaller data types to larger data types to avoid the loss of data.

**Note:**

* We get TypeError, if we try to add str and int. For example, '12' + 23. Python is not able to use Implicit Conversion in such conditions.
* 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 built-in 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.

## Example 2: Addition of string and integer Using Explicit Conversion

num\_string = '12'

num\_integer = 23

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

# explicit type conversion

num\_string = int(num\_string)

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

num\_sum = num\_integer + num\_string

print("Sum:",num\_sum)

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

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

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

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

Sum: 35

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

In the above example, we have created two variables: num\_string and num\_integer with str and int type values respectively. Notice the code,

num\_string = int(num\_string)

Here, we have used int() to perform explicit type conversion of num\_string to integer type.

After converting num\_string to an integer value, Python is able to add these two variables.

Finally, we got the num\_sum value i.e **35** and data type to be int.

## Key Points to Remember

1. Type Conversion is the conversion of an 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 Basic Input and Output

In this tutorial, we will learn simple ways to display output to users and take input from users in Python with the help of examples.

## Python Output

In Python, we can simply use the print() function to print output. For example,

print('Python is powerful')

# Output: Python is powerful

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, the print() function displays the string enclosed inside the single quotation.

**Syntax of print()**

In the above code, the print() function is taking a single parameter.

However, the actual syntax of the print function accepts **5** parameters

print(object= separator= end= file= flush=)

Here,

* **object** - value(s) to be printed
* **sep** (optional) - allows us to separate multiple **objects** inside print().
* **end** (optional) - allows us to add add specific values like new line "\n", tab "\t"
* **file** (optional) - where the values are printed. It's default value is sys.stdout (screen)
* **flush** (optional) - boolean specifying if the output is flushed or buffered. Default: False

## Example 1: Python Print Statement

print('Good Morning!')

print('It is rainy today')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Good Morning!

It is rainy today

In the above example, the print() statement only includes the **object** to be printed. Here, the value for **end** is not used. Hence, it takes the default value '\n'.

So we get the output in two different lines.

## Example 2: Python print() with end Parameter

# print with end whitespace

print('Good Morning!', end= ' ')

print('It is rainy today')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Good Morning! It is rainy today

Notice that we have included the end= ' ' after the end of the first print() statement.

Hence, we get the output in a single line separated by space.

## Example 3: Python print() with sep parameter

print('New Year', 2023, 'See you soon!', sep= '. ')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

New Year. 2023. See you soon!

In the above example, the print() statement includes multiple **items** separated by a comma.

Notice that we have used the optional parameter sep= ". " inside the print() statement.

Hence, the output includes items separated by . not comma.

## Example: Print Python Variables and Literals

We can also use the print() function to print [Python variables](https://www.programiz.com/python-programming/variables-constants-literals). For example,

number = -10.6

name = "Programiz"

# print literals

print(5)

# print variables

print(number)

print(name)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

5

-10.6

Programiz

## Example: Print Concatenated Strings

We can also join two strings together inside the print() statement. For example,

print('Programiz is ' + 'awesome.')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Programiz is awesome.

Here,

* the + operator joins two strings 'Programiz is ' and 'awesome.'
* the print() function prints the joined string

## 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. For example,

x = 5

y = 10

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

[Run Code](https://www.programiz.com/python-programming/online-compiler)

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

## Python Input

While programming, we might want to take the input from the user. In Python, we can use the input() function.

**Syntax of input()**

input(prompt)

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

## Example: Python User Input

# using input() to take user input

num = input('Enter a number: ')

print('You Entered:', num)

print('Data type of num:', type(num))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Enter a number: 10

You Entered: 10

Data type of num: <class 'str'>

In the above example, we have used the input() function to take input from the user and stored the user input in the num variable.

It is important to note that the entered value **10** is a string, not a number. So, type(num) returns <class 'str'>.

To convert user input into a number we can use int() or float() functions as:

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

Here, the data type of the user input is converted from string to integer .

# Python Namespace and Scope

In this tutorial, you will learn about namespace, mapping from names to objects, and scope of a variable with the help of examples.

To simply put it, a namespace is a collection of names.

In Python, we can imagine a namespace as a mapping of every name we have defined to corresponding objects.

It is used to store the values of variables and other objects in the program, and to associate them with a specific name.

This allows us to use the same name for different variables or objects in different parts of your code, without causing any conflicts or confusion.

## Types of Python namespace

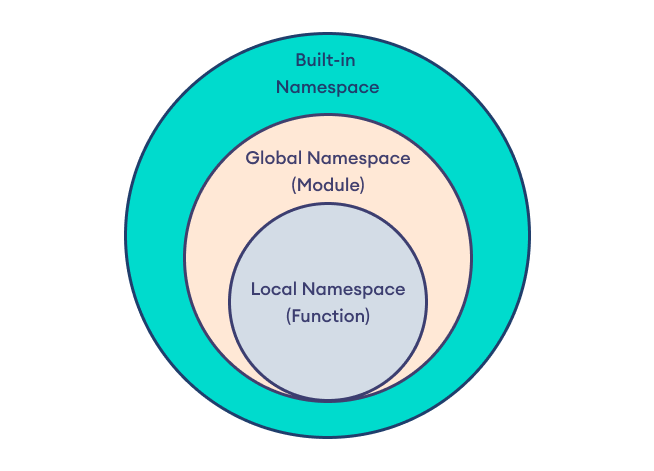
A namespace containing all the **built-in names** is created when we start the Python interpreter and exists as long as the interpreter runs.

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 does 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. The following diagram may help to clarify this concept.

Python Namespaces

## 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 1: Scope and Namespace in Python

# global\_var is in the global namespace

global\_var = 10

def outer\_function():

# outer\_var is in the local namespace

outer\_var = 20

def inner\_function():

# inner\_var is in the nested local namespace

inner\_var = 30

print(inner\_var)

print(outer\_var)

inner\_function()

# print the value of the global variable

print(global\_var)

# call the outer function and print local and nested local variables

outer\_function()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

10

20

30

In the above example, there are three separate namespaces: the global namespace, the local namespace within the outer function, and the local namespace within the inner function.

Here,

* global\_var - is in the global namespace with value **10**
* outer\_val - is in the local namespace of outer\_function() with value **20**
* inner\_val - is in the nested local namespace of inner\_function() with value **30**

When the code is executed, the global\_var global variable is printed first, followed by the local variable: outer\_var and inner\_var when the outer and inner functions are called.

## Example 2: Use of global Keyword in Python

# define global variable

global\_var = 10

def my\_function():

# define local variable

local\_var = 20

# modify global variable value

global global\_var

global\_var = 30

# print global variable value

print(global\_var)

# call the function and modify the global variable

my\_function()

# print the modified value of the global variable

print(global\_var)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

10

30

Here, when the function is called, the global keyword is used to indicate that global\_var is a global variable, and its value is modified to **30**.

So, when the code is executed, global\_var is printed first with a value of **10**, then the function is called and the global variable is modified to **30** from the inside of the function.

And finally the modified value of global\_var is printed again.

**Python flow control**

Python if… else

Python for loop

Python while loop

Python break and continue

Python pass

Python if...else Statement

In this tutorial, you will learn about the Python if...else statement with the help of examples to create decision-making programs.

In computer programming, we use the if statement to run a block code only when a certain condition is met.

For example, assigning grades **(A, B, C)** based on marks obtained by a student.

1. if the percentage is above **90**, assign grade **A**
2. if the percentage is above **75**, assign grade **B**
3. if the percentage is above **65**, assign grade **C**

In Python, there are three forms of the if...else statement.

1. if statement
2. if...else statement
3. if...elif...else statement

## 1. Python if statement

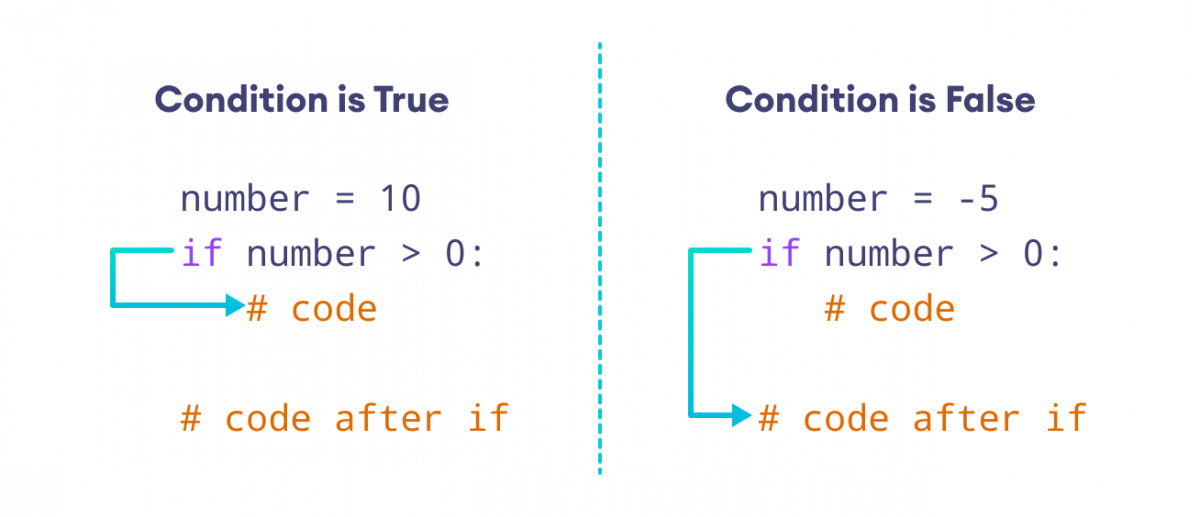
The syntax of if statement in Python is:

if condition:

# body of if statement

The if statement evaluates condition.

1. If condition is evaluated to True, the code inside the body of if is executed.
2. If condition is evaluated to False, the code inside the body of if is skipped.

Working of if Statement

### Example 1: Python if Statement

number = 10

# check if number is greater than 0

if number > 0:

print('Number is positive.')

print('The if statement is easy')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Number is positive.

The if statement is easy

In the above example, we have created a variable named number. Notice the test condition,

number > 0

Here, since number is greater than **0**, the condition evaluates True.

If we change the value of variable to a negative integer. Let's say **-5**.

number = -5

Now, when we run the program, the output will be:

The if statement is easy

This is because the value of number is less than **0**. Hence, the condition evaluates to False. And, the body of if block is skipped.

## 2. Python if...else Statement

An if statement can have an optional else clause.

The syntax of if...else statement is:

if condition:

# block of code if condition is True

else:

# block of code if condition is False

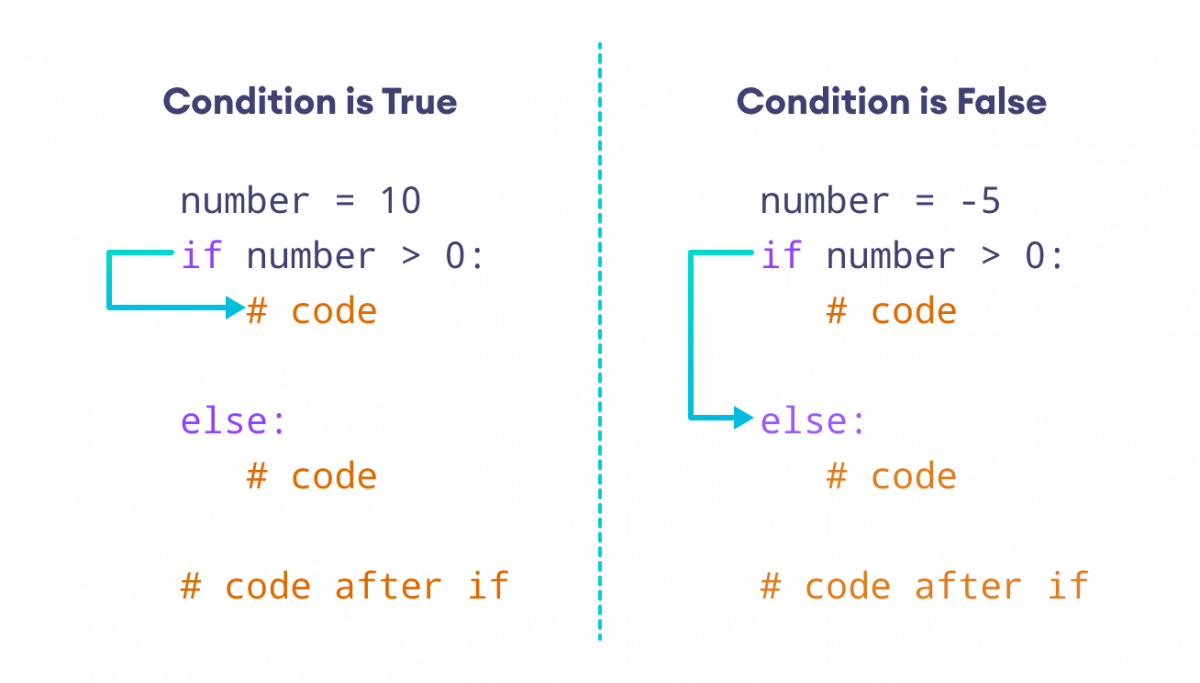
The if...else statement evaluates the given condition:

**If the condition evaluates to** **True,**

* the code inside if is executed
* the code inside else is skipped

**If the condition evaluates to** **False,**

* the code inside else is executed
* the code inside if is skipped

Working of if...else Statement

### Example 2. Python if...else Statement

number = 10

if number > 0:

print('Positive number')

else:

print('Negative number')

print('This statement is always executed')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Positive number

This statement is always executed

In the above example, we have created a variable named number. Notice the test condition,

number > 0

Since the value of number is **10**, the test condition evaluates to True. Hence code inside the body of if is executed.

If we change the value of variable to a negative integer. Let's say **-5**.

number = -5

Now if we run the program, the output will be:

Number is negative.

This statement is always executed.

Here, the test condition evaluates to False. Hence code inside the body of else is executed.

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

The if...else statement is used to execute a block of code among two alternatives.

However, if we need to make a choice between more than two alternatives, then we use the if...elif...else statement.

The syntax of the if...elif...else statement is:

if condition1:

# code block 1

elif condition2:

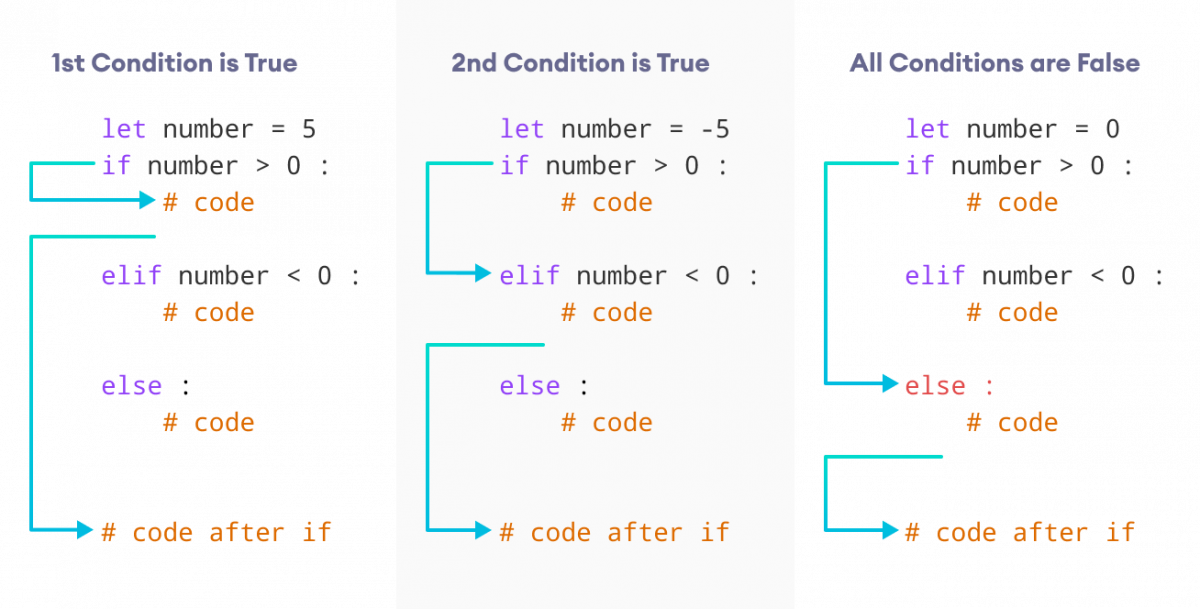
# code block 2

else:

# code block 3

Here,

1. If **condition1** evaluates to true, **code block 1** is executed.
2. If **condition1** evaluates to false, then **condition2** is evaluated.
   1. If **condition2** is true, **code block 2** is executed.
   2. If **condition2** is false, **code block 3** is executed.

Working of if...elif Statement

### Example 3: Python if...elif...else Statement

number = 0

if number > 0:

print("Positive number")

elif number == 0:

print('Zero')

else:

print('Negative number')

print('This statement is always executed')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Zero

This statement is always executed

In the above example, we have created a variable named number with the value **0**. Here, we have two condition expressions:

Here, both the conditions evaluate to False. Hence the statement inside the body of else is executed.

## Python Nested if statements

We can also use an if statement inside of an if statement. This is known as a **nested if** statement.

The syntax of **nested if** statement is:

# outer if statement

if condition1:

# statement(s)

# inner if statement

if condition2:

# statement(s)

**Notes**:

* We can add else and elif statements to the inner if statement as required.
* We can also insert inner if statement inside the outer else or elif statements(if they exist)
* We can nest multiple layers of if statements.

### Example 4: Python Nested if Statement

number = 5

# outer if statement

if (number >= 0):

# inner if statement

if number == 0:

print('Number is 0')

# inner else statement

else:

print('Number is positive')

# outer else statement

else:

print('Number is negative')

# Output: Number is positive

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have used a **nested if** statement to check whether the given number is **positive, negative, or 0**.

Python for Loop

In this tutorial, we'll learn how to use a for loop in Python with the help of examples.

In computer programming, loops are used to repeat a block of code.

For example, if we want to show a message **100** times, then we can use a loop. It's just a simple example; you can achieve much more with loops.

There are 2 types of loops in Python:

* [for loop](https://www.programiz.com/python-programming/for-loop)
* [while loop](https://www.programiz.com/python-programming/while-loop)

## Python for Loop

In Python, a for loop is used to iterate over sequences such as [lists](https://www.programiz.com/python-programming/list), [tuples](https://www.programiz.com/python-programming/tuple), [string](https://www.programiz.com/python-programming/strings), etc. For example,

languages = ['Swift', 'Python', 'Go', 'JavaScript']

# run a loop for each item of the list

for language in languages:

print(language)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Swift

Python

Go

JavaScript

In the above example, we have created a list called languages.

Initially, the value of language is set to the first element of the array,i.e. Swift, so the print statement inside the loop is executed.

language is updated with the next element of the list, and the print statement is executed again. This way, the loop runs until the last element of the list is accessed.

## for Loop Syntax

The syntax of a for loop is:

for val in sequence:

# statement(s)

Here, val accesses each item of sequence on each iteration. The loop continues until we reach the last item in the sequence.

## Flowchart of Python for Loop

Working of Python for loop

## Example: Loop Through a String

for x in 'Python':

print(x)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

P

y

t

h

o

n

## Python for Loop with Python range()

A [range](https://www.programiz.com/python-programming/methods/built-in/range) is a series of values between two numeric intervals.

We use Python's built-in function range() to define a range of values. For example,

values = range(4)

Here, **4** inside range() defines a range containing values **0, 1, 2, 3.**

In Python, we can use for loop to iterate over a range. For example,

# use of range() to define a range of values

values = range(4)

# iterate from i = 0 to i = 3

for i in values:

print(i)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

0

1

2

3

In the above example, we have used the for loop to iterate over a range from **0** to **3**.

The value of i is set to **0** and it is updated to the next number of the range on each iteration. This process continues until **3** is reached.

|  |  |  |
| --- | --- | --- |
| Iteration | Condition | Action |
| 1st | True | 0 is printed. i is increased to **1**. |
| 2nd | True | 1 is printed. i is increased to **2**. |
| 3rd | True | 2 is printed. i is increased to **3**. |
| 4th | True | 3 is printed. i is increased to **4**. |
| 5th | False | The loop is terminated |

**Note**: To learn more about the use of for loop with range, visit [Python range()](https://www.programiz.com/python-programming/methods/built-in/range).

## Using a for Loop Without Accessing Items

It is not mandatory to use items of a sequence within a for loop. For example,

languages = ['Swift', 'Python', 'Go']

for language in languages:

print('Hello')

print('Hi')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Hello

Hi

Hello

Hi

Hello

Hi

Here, the loop runs three times because our list has three items. In each iteration, the loop body prints 'Hello' and 'Hi'. The items of the list are not used within the loop.

If we do not intend to use items of a sequence within the loop, we can write the loop like this:

languages = ['Swift', 'Python', 'Go']

for \_ in languages:

print('Hello')

print('Hi')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

The \_ symbol is used to denote that the elements of a sequence will not be used within the loop body.

## Python for loop with else

A for loop can have an optional else block. The else part is executed when the loop is exhausted (after the loop iterates through every item of a sequence). For example,

digits = [0, 1, 5]

for i in digits:

print(i)

else:

print("No items left.")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

0

1

5

No items left.

Here, the for loop prints all the items of the digits list. When the loop finishes, it executes the else block and prints No items left.

**Note**: The else block will not execute if the for loop is stopped by a [break](https://www.programiz.com/python-programming/break-continue) statement.

# Python while Loop

In this tutorial, we will learn about the while loop in Python programming with the help of examples.

In programming, loops are used to repeat a block of code. For example, if we want to show a message **100** times, then we can use a loop. It's just a simple example, we can achieve much more with loops.

In the previous tutorial, we learned about [Python for loop](https://www.programiz.com/python-programming/for-loop). Now we will learn about the while loop.

## Python while Loop

Python while loop is used to run a block code until a certain condition is met.

The syntax of while loop is:

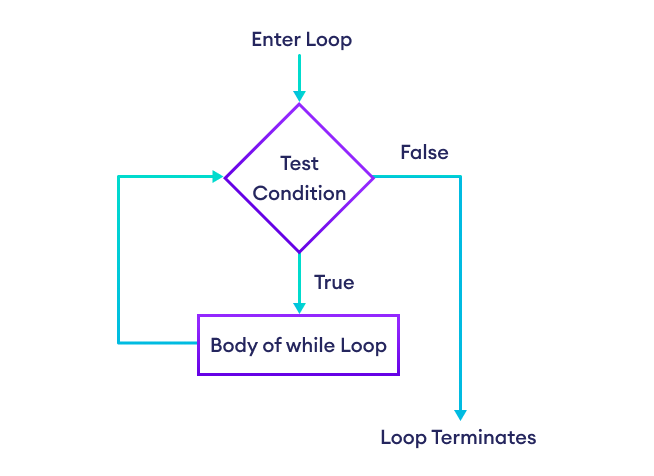
while condition:

# body of while loop

Here,

1. A while loop evaluates the condition
2. If the condition evaluates to True, the code inside the while loop is executed.
3. condition is evaluated again.
4. This process continues until the condition is False.
5. When condition evaluates to False, the loop stops.

## Flowchart of Python while Loop

Flowchart of while Loop

## Example: Python while Loop

# program to display numbers from 1 to 5

# initialize the variable

i = 1

n = 5

# while loop from i = 1 to 5

while i <= n:

print(i)

i = i + 1

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

1

2

3

4

5

Here's how the program works:

|  |  |  |
| --- | --- | --- |
| Variable | Condition: i <= n | Action |
| i = 1 n = 5 | True | 1 is printed. i is increased to **2**. |
| i = 2 n = 5 | True | 2 is printed. i is increased to **3**. |
| i = 3 n = 5 | True | 3 is printed. i is increased to **4**. |
| i = 4 n = 5 | True | 4 is printed. i is increased to **5**. |
| i = 5 n = 5 | True | 5 is printed. i is increased to **6**. |
| i = 6 n = 5 | False | The loop is terminated. |

### Example 2: Python while Loop

# program to calculate the sum of numbers

# until the user enters zero

total = 0

number = int(input('Enter a number: '))

# add numbers until number is zero

while number != 0:

total += number # total = total + number

# take integer input again

number = int(input('Enter a number: '))

print('total =', total)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Enter a number: 12

Enter a number: 4

Enter a number: -5

Enter a number: 0

total = 11

In the above example, the while iterates until the user enters zero. When the user enters zero, the test condition evaluates to False and the loop ends.

## Infinite while Loop in Python

If the condition of a loop is always True, the loop runs for infinite times (until the memory is full). For example,

age = 32

# the test condition is always True

while age > 18:

print('You can vote')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, the condition always evaluates to True. Hence, the loop body will run for infinite times.

## Python While loop with else

In Python, a while loop may have an optional else block.

Here, the else part is executed after the condition of the loop evaluates to False.

counter = 0

while counter < 3:

print('Inside loop')

counter = counter + 1

else:

print('Inside else')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Inside loop

Inside loop

Inside loop

Inside else

**Note**: The else block will not execute if the while loop is terminated by a [break](https://www.programiz.com/python-programming/break-continue) statement.

counter = 0

while counter < 3:

# loop ends because of break

# the else part is not executed

if counter == 1:

break

print('Inside loop')

counter = counter + 1

else:

print('Inside else')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Inside loop

Inside else

## Python for Vs while loops

The for loop is usually used when the number of iterations is **known**. For example,

# this loop is iterated 4 times (0 to 3)

for i in range(4):

print(i)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

The while loop is usually used when the number of iterations is **unknown**. For example,

while condition:

# run code until the condition evaluates to False

# Python break and continue

In this tutorial, we will learn to use break and continue statements to alter the flow of a loop.

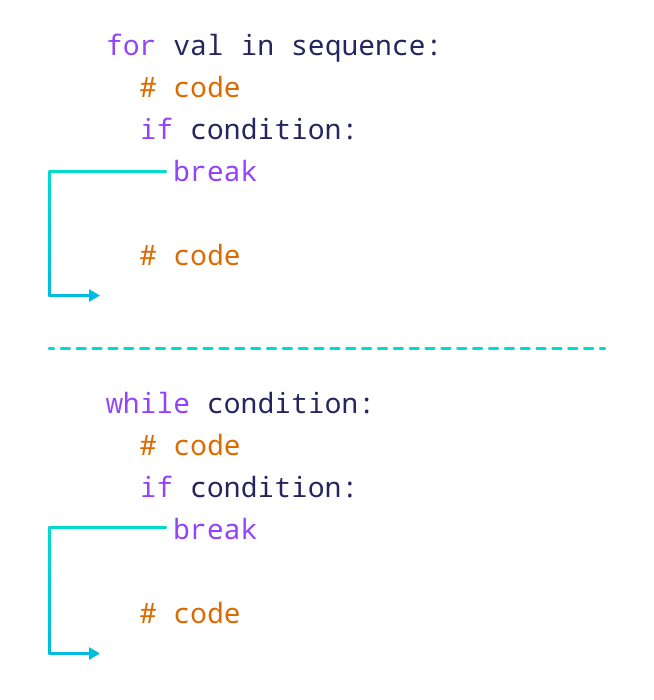
## Python break Statement

The break statement is used to terminate the loop immediately when it is encountered.

The syntax of the break statement is:

break

## Working of Python break Statement

Working of the break statement

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 above.

## Python break Statement with for Loop

We can use the break statement with the for loop to terminate the loop when a certain condition is met. For example,

for i in range(5):

if i == 3:

break

print(i)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

0

1

2

In the above example, we have used the for loop to print the value of i. Notice the use of the break statement,

if i == 3:

break

Here, when i is equal to **3**, the break statement terminates the loop. Hence, the output doesn't include values after **2**.

**Note**: The break statement is almost always used with decision-making statements.

## Python break Statement with while Loop

We can also terminate the while loop using the break statement. For example,

# program to find first 5 multiples of 6

i = 1

while i <= 10:

print('6 \* ',(i), '=',6 \* i)

if i >= 5:

break

i = i + 1

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

6 \* 1 = 6

6 \* 2 = 12

6 \* 3 = 18

6 \* 4 = 24

6 \* 5 = 30

In the above example, we have used the while loop to find the first **5** multiples of **6**. Here notice the line,

if i >= 5:

break

This means when i is greater than or equal to **5**, the while loop is terminated.

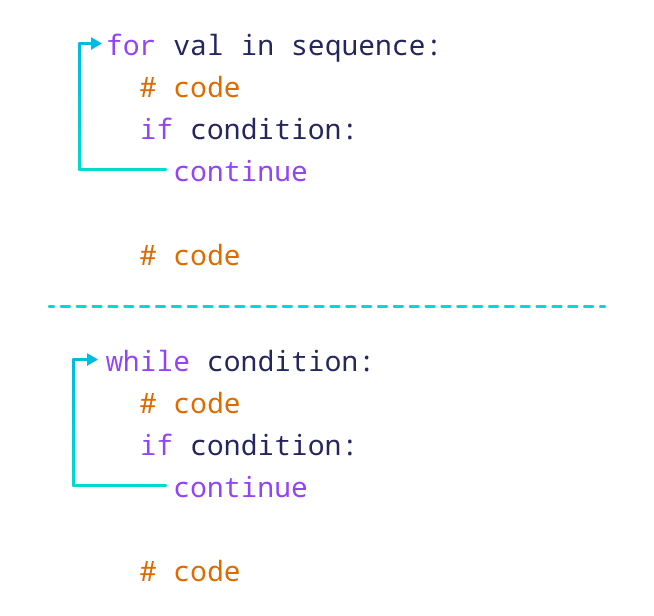
## Python continue Statement

The continue statement is used to skip the current iteration of the loop and the control flow of the program goes to the next iteration.

The syntax of the continue statement is:

continue

## Working of Python continue Statement

How continue statement works in python

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

## Python continue Statement with for Loop

We can use the continue statement with the for loop to skip the current iteration of the loop. Then the control of the program jumps to the next iteration. For example,

for i in range(5):

if i == 3:

continue

print(i)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

0

1

2

4

In the above example, we have used the for loop to print the value of i. Notice the use of the continue statement,

if i == 3:

continue

Here, when i is equal to **3**, the continue statement is executed. Hence, the value **3** is not printed to the output.

## Python continue Statement with while Loop

In Python, we can also skip the current iteration of the while loop using the continue statement. For example,

# program to print odd numbers from 1 to 10

num = 0

while num < 10:

num += 1

if (num % 2) == 0:

continue

print(num)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

1

3

5

7

9

In the above example, we have used the while loop to print the odd numbers between **1** to **10**. Notice the line,

if (num % 2) == 0:

continue

Here, when the number is even, the continue statement skips the current iteration and starts the next iteration.

# Python pass Statement

In this tutorial, we'll learn about the pass statement in Python programming with the help of examples.

In Python programming, the pass statement is a null statement which can be used as a placeholder for future code.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. In such cases, we can use the pass statement.

The syntax of the pass statement is:

pass

## Using pass With Conditional Statement

n = 10

# use pass inside if statement

if n > 10:

pass

print('Hello')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, notice that we have used the pass statement inside the if statement.

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

Suppose we didn't use pass or just put a comment as:

n = 10

if n > 10:

# write code later

print('Hello')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we will get an error message: IndentationError: expected an indented block

**Note**: 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.

## Use of pass Statement inside Function or Class

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. For example,

def function(args):

pass

class Example:

pass

**Python functions**

Python function

Function argument

Python recursion

Anonyms functions

Global, local and non local

Python global keyword

Python modules

Python package

# Python Functions

In this tutorial, we will learn about the Python function and function expressions with the help of examples.

A function is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

* create a circle function
* create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

## Types of function

There are two types of function in Python programming:

* **Standard library functions** - These are built-in functions in Python that are available to use.
* **User-defined functions** - We can create our own functions based on our requirements.

## Python Function Declaration

The syntax to declare a function is:

def function\_name(arguments):

# function body

return

Here,

* def - keyword used to declare a function
* function\_name - any name given to the function
* arguments - any value passed to function
* return (optional) - returns value from a function

Let's see an example,

def greet():

print('Hello World!')

Here, we have created a function named greet(). It simply prints the text Hello World!.

This function doesn't have any arguments and doesn't return any values. We will learn about arguments and return statements later in this tutorial.

## Calling a Function in Python

In the above example, we have declared a function named greet().

def greet():

print('Hello World!')

Now, to use this function, we need to call it.

Here's how we can call the greet() function in Python.

# call the function

greet()

## Example: Python Function

def greet():

print('Hello World!')

# call the function

greet()

print('Outside function')

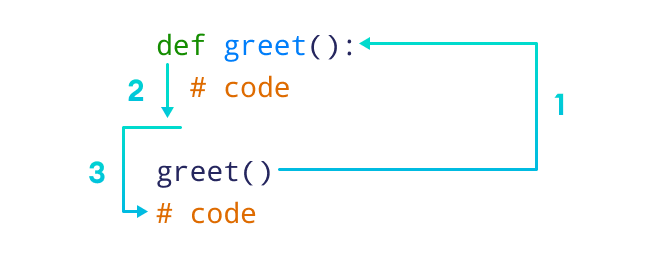
[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Hello World!

Outside function

In the above example, we have created a function named greet(). Here's how the program works:

Working of Python Function

Here,

* When the function is called, the control of the program goes to the function definition.
* All codes inside the function are executed.
* The control of the program jumps to the next statement after the function call.

## Python Function Arguments

As mentioned earlier, a function can also have arguments. An argument is a value that is accepted by a function. For example,

# function with two arguments

def add\_numbers(num1, num2):

sum = num1 + num2

print('Sum: ',sum)

# function with no argument

def add\_numbers():

# code

If we create a function with arguments, we need to pass the corresponding values while calling them. For example,

# function call with two values

add\_numbers(5, 4)

# function call with no value

add\_numbers()

Here, add\_numbers(5, 4) specifies that arguments num1 and num2 will get values **5** and **4** respectively.

## Example 1: Python Function Arguments

# function with two arguments

def add\_numbers(num1, num2):

sum = num1 + num2

print("Sum: ",sum)

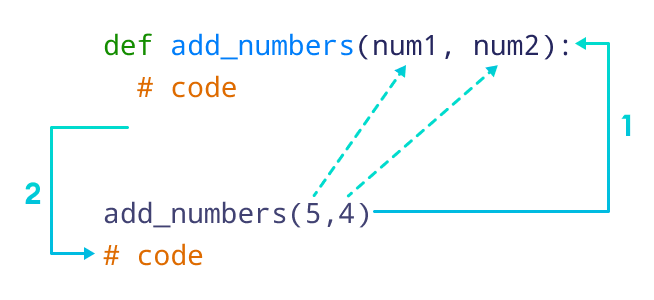
# function call with two values

add\_numbers(5, 4)

# Output: Sum: 9

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have created a function named add\_numbers() with arguments: num1 and num2.

Python Function with Arguments

We can also call the function by mentioning the argument name as:

add\_numbers(num1 = 5, num2 = 4)

In Python, we call it Keyword Argument (or named argument). The code above is equivalent to

add\_numbers(5, 4)

## The return Statement in Python

A Python function may or may not return a value. If we want our function to return some value to a function call, we use the return statement. For example,

def add\_numbers():

...

return sum

Here, we are returning the variable sum to the function call.

**Note:** The return statement also denotes that the function has ended. Any code after return is not executed.

## Example 2: Function return Type

# function definition

def find\_square(num):

result = num \* num

return result

# function call

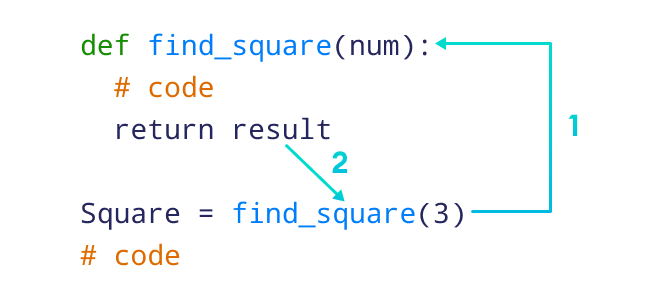
square = find\_square(3)

print('Square:',square)

# Output: Square: 9

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have created a function named find\_square(). The function accepts a number and returns the square of the number.

Working of functions in Python

### Example 3: Add Two Numbers

# function that adds two numbers

def add\_numbers(num1, num2):

sum = num1 + num2

return sum

# calling function with two values

result = add\_numbers(5, 4)

print('Sum: ', result)

# Output: Sum: 9

[Run Code](https://www.programiz.com/python-programming/online-compiler)

## Python Library Functions

In Python, standard library functions are the built-in functions that can be used directly in our program. For example,

* print() - prints the string inside the quotation marks
* sqrt() - returns the square root of a number
* pow() - returns the power of a number

These library functions are defined inside the module. And, to use them we must include the module inside our program.

For example, sqrt() is defined inside the math module.

## Example 4: Python Library Function

import math

# sqrt computes the square root

square\_root = math.sqrt(4)

print("Square Root of 4 is",square\_root)

# pow() comptes the power

power = pow(2, 3)

print("2 to the power 3 is",power)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Square Root of 4 is 2.0

2 to the power 3 is 8

In the above example, we have used

* math.sqrt(4) - to compute the square root of **4**
* pow(2, 3) - computes the power of a number i.e. **23**

Here, notice the statement,

import math

Since sqrt() is defined inside the math module, we need to include it in our program.

## Benefits of Using Functions

**1. Code Reusable** - We can use the same function multiple times in our program which makes our code reusable. For example,

# function definition

def get\_square(num):

return num \* num

for i in [1,2,3]:

# function call

result = get\_square(i)

print('Square of',i, '=',result)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Square of 1 = 1

Square of 2 = 4

Square of 3 = 9

In the above example, we have created the function named get\_square() to calculate the square of a number. Here, the function is used to calculate the square of numbers from **1** to **3**.

Hence, the same method is used again and again.

**2. Code Readability** - Functions help us break our code into chunks to make our program readable and easy to understand.

# Python Function Arguments

In this tutorial, we will learn about function arguments in Python with the help of examples.

In computer programming, an argument is a value that is accepted by a function.

Before we learn about function arguments, make sure to know about [Python Functions](https://www.programiz.com/python-programming/function).

## Example 1: Python Function Arguments

def add\_numbers(a, b):

sum = a + b

print('Sum:', sum)

add\_numbers(2, 3)

# Output: Sum: 5

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, the function add\_numbers() takes two parameters: a and b. Notice the line,

add\_numbers(2, 3)

Here, add\_numbers(2, 3) specifies that parameters a and b will get values **2** and **3** respectively.

## Function Argument with Default Values

In Python, we can provide default values to function arguments.

We use the = operator to provide default values. For example,

def add\_numbers( a = 7, b = 8):

sum = a + b

print('Sum:', sum)

# function call with two arguments

add\_numbers(2, 3)

# function call with one argument

add\_numbers(a = 2)

# function call with no arguments

add\_numbers()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Sum: 5

Sum: 10

Sum: 15

In the above example, notice the function definition

def add\_numbers(a = 7, b = 8):

...

Here, we have provided default values **7** and **8** for parameters a and b respectively. Here's how this program works

**1. add\_number(2, 3)**

Both values are passed during the function call. Hence, these values are used instead of the default values.

**2. add\_number(2)**

Only one value is passed during the function call. So, according to the positional argument **2** is assigned to argument a, and the default value is used for parameter b.

**3. add\_number()**

No value is passed during the function call. Hence, default value is used for both parameters a and b.

## Python Keyword Argument

In keyword arguments, arguments are assigned based on the name of arguments. For example,

def display\_info(first\_name, last\_name):

print('First Name:', first\_name)

print('Last Name:', last\_name)

display\_info(last\_name = 'Cartman', first\_name = 'Eric')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

First Name: Eric

Last Name: Cartman

Here, notice the function call,

display\_info(last\_name = 'Cartman', first\_name = 'Eric')

Here, we have assigned names to arguments during the function call.

Hence, first\_name in the function call is assigned to first\_name in the function definition. Similarly, last\_name in the function call is assigned to last\_name in the function definition.

In such scenarios, the position of arguments doesn't matter.

## Python Function With Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. To handle this kind of situation, we can use arbitrary arguments in Python.

Arbitrary arguments allow us to pass a varying number of values during a function call.

We use an asterisk (\*) before the parameter name to denote this kind of argument. For example,

# program to find sum of multiple numbers

def find\_sum(\*numbers):

result = 0

for num in numbers:

result = result + num

print("Sum = ", result)

# function call with 3 arguments

find\_sum(1, 2, 3)

# function call with 2 arguments

find\_sum(4, 9)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Sum = 6

Sum = 13

In the above example, we have created the function find\_sum() that accepts arbitrary arguments. Notice the lines,

find\_sum(1, 2, 3)

find\_sum(4, 9)

Here, we are able to call the same function with different arguments.

**Note**: After getting multiple values, numbers behave as an array so we are able to use the for loop to access each value.

# Python Recursion

In this tutorial, you will learn to create a recursive function (a function that calls itself).

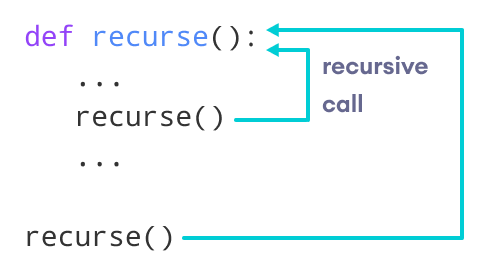
Recursion is the process of defining something in terms of itself.

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.



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))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**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

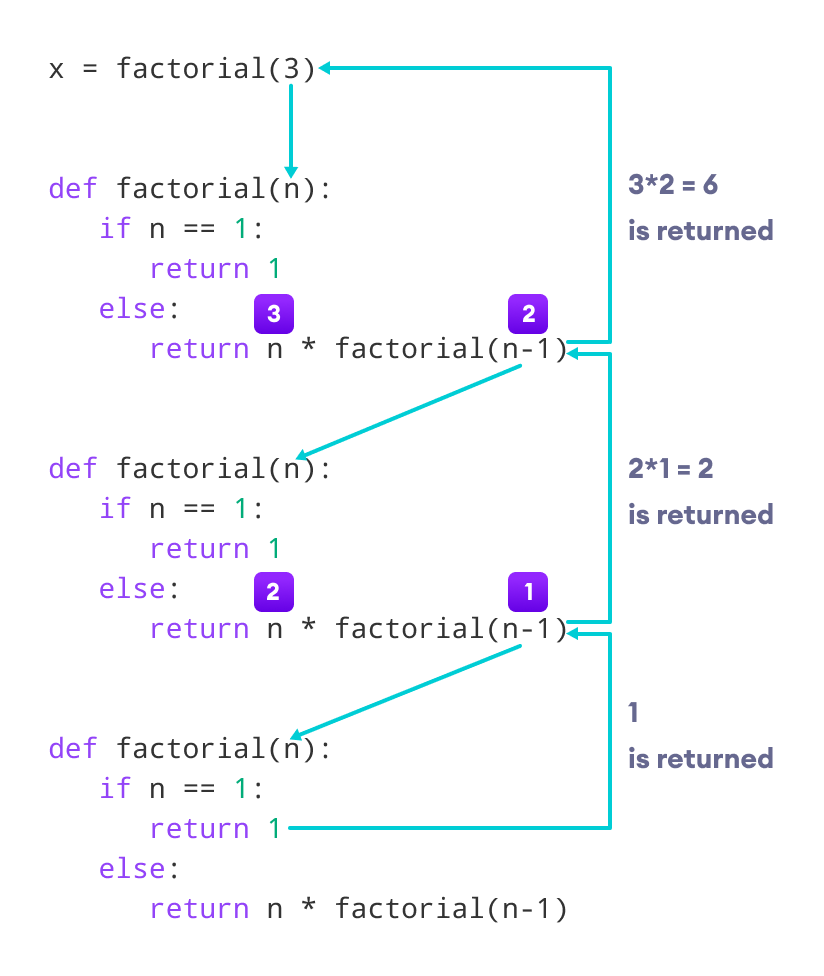
3 \* 2 \* factorial(1) # 3rd call with 1

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:



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 Lambda/Anonymous Function

In this tutorial, we'll learn about Python lambda functions with the help of examples.

In Python, a lambda function is a special type of function without the function name. For example,

lambda : print('Hello World')

Here, we have created a lambda function that prints 'Hello World'.

Before you learn about lambdas, make sure to know about [Python Functions](https://www.programiz.com/python-programming/function).

## Python lambda Function Declaration

We use the lambda keyword instead of def to create a lambda function. Here's the syntax to declare the lambda function:

lambda argument(s) : expression

Here,

* argument(s) - any value passed to the lambda function
* expression - expression is executed and returned

Let's see an example,

greet = lambda : print('Hello World')

Here, we have defined a lambda function and assigned it to the variable named greet.

To execute this lambda function, we need to call it. Here's how we can call the lambda function

# call the lambda

greet()

The lambda function above simply prints the text 'Hello World'.

**Note**: This lambda function doesn't have any arguments.

## Example: Python lambda Function

# declare a lambda function

greet = lambda : print('Hello World')

# call lambda function

greet()

# Output: Hello World

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have defined a lambda function and assigned it to the greet variable.

When we call the lambda function, the print() statement inside the lambda function is executed.

## Python lambda Function with an Argument

Similar to normal functions, the lambda function can also accept arguments. For example,

# lambda that accepts one argument

greet\_user = lambda name : print('Hey there,', name)

# lambda call

greet\_user('Delilah')

# Output: Hey there, Delilah

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have assigned a lambda function to the greet\_user variable.

Here, name after the lambda keyword specifies that the lambda function accepts the argument named name.

Notice the call of lambda function,

greet\_user('Delilah')

Here, we have passed a string value 'Delilah' to our lambda function.

And finally, the statement inside the lambda function is executed.

Python Variable Scope

In this tutorial, we'll learn about Python Global variables, Local variables, and Nonlocal variables with the help of examples.

In Python, we can declare variables in three different scopes: local scope, global, and nonlocal scope.

A variable scope specifies the region where we can access a variable. For example,

def add\_numbers():

sum = 5 + 4

Here, the sum variable is created inside the function, so it can only be accessed within it (local scope). This type of variable is called a local variable.

Based on the scope, we can classify Python variables into three types:

1. Local Variables
2. Global Variables
3. Nonlocal Variables

## Python Local Variables

When we declare variables inside a function, these variables will have a local scope (within the function). We cannot access them outside the function.

These types of variables are called local variables. For example,

def greet():

# local variable

message = 'Hello'

print('Local', message)

greet()

# try to access message variable

# outside greet() function

print(message)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Local Hello

NameError: name 'message' is not defined

Here, the message variable is local to the greet() function, so it can only be accessed within the function.

That's why we get an error when we try to access it outside the greet() function.

To fix this issue, we can make the variable named message global.

## Python 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.

# declare global variable

message = 'Hello'

def greet():

# declare local variable

print('Local', message)

greet()

print('Global', message)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Local Hello

Global Hello

This time we can access the message variable from outside of the greet() function. This is because we have created the message variable as the global variable.

# declare global variable

message = 'Hello'

Now, message will be accessible from any scope (region) of the program.

## Python Nonlocal Variables

In Python, 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.

We use the nonlocal keyword to create nonlocal variables.For example,

# outside function

def outer():

message = 'local'

# nested function

def inner():

# declare nonlocal variable

nonlocal message

message = 'nonlocal'

print("inner:", message)

inner()

print("outer:", message)

outer()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

inner: nonlocal

outer: nonlocal

In the above example, there is a nested inner() function. We have used the 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 tutorial, we'll learn about the global keyword with the help of examples.

In Python, the global keyword allows us 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.

Before we learn about the global keyword, make sure you have got some basics of [Python Variable Scope](https://www.programiz.com/python-programming/global-local-nonlocal-variables).

## Access and Modify Python Global Variable

First let's try to access a global variable from the inside of a function,

c = 1 # global variable

def add():

print(c)

add()

# Output: 1

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we can see that we have accessed a global variable from the inside of a function.

However, if we try to modify the global variable from inside a function as:

# global variable

c = 1

def add():

# increment c by 2

c = c + 2

print(c)

add()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

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: Changing Global Variable From Inside a Function using global

# global variable

c = 1

def add():

# use of global keyword

global c

# increment c by 2

c = c + 2

print(c)

add()

# Output: 3

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have defined c as the global keyword inside add().

Then, we have incremented the variable c by **2**, i.e c = c + 2.

As we can see while calling add(), the value of global variable c is modified from **1** to **3**.

## Global in Nested Functions

In Python, we can also use the global keyword in a nested function. For example,

def outer\_function():

num = 20

def inner\_function():

global num

num = 25

print("Before calling inner\_function(): ", num)

inner\_function()

print("After calling inner\_function(): ", num)

outer\_function()

print("Outside both function: ", num)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Before calling inner\_function(): 20

After calling inner\_function(): 20

Outside both function: 25

In the above example, we declared a global variable inside the nested function inner\_function().

Inside outer\_function(), num has no effect of the global keyword.

Before and after calling inner\_function(), num takes the value of the local variable i.e num = 20.

Outside of the outer\_function() function, num will take the value defined in the inner\_function() function i.e x = 25.

This is because we have used the global keyword in num to create a global variable inside the inner\_function() function (local scope).

So, if we make any changes inside the inner\_function() function, the changes appear outside the local scope, i.e. outer\_function().

## 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 the global keyword.
* We use the global keyword to read and write a global variable inside a function.
* Use of the global keyword outside a function has no effect.

# Python Modules

In this tutorial, 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.

As our program grows bigger, it may contain many lines of code. Instead of putting everything in a single file, we can use modules to separate codes in separate files as per their functionality. This makes our code organized and easier to maintain.

Module is a file that contains code to perform a specific task. A module may contain variables, functions, classes etc. Let's see an example,

Let us create a module. Type the following and save it as example.py.

# Python Module addition

def add(a, b):

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.

## 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:

addition.add(4,5) # returns 9

**Note**:

* Python has tons of standard modules. You can check out the full list of [Python standard modules](http://docs.python.org/3/py-modindex.html) and their use cases.
* Standard modules can be imported the same way as we import our user-defined modules.

## Import Python Standard Library Modules

The Python standard library contains well over **200** modules. We can import a module according to our needs.

Suppose we want to get the value of pi, first we import the math module and use math.pi. For example,

# import standard math module

import math

# use math.pi to get value of pi

print("The value of pi is", math.pi)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

The value of pi is 3.141592653589793

## Python import with Renaming

In Python, we can also import a module by renaming it. For example,

# import module by renaming it

import math as m

print(m.pi)

# Output: 3.141592653589793

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, 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. For example,

# import only pi from math module

from math import pi

print(pi)

# Output: 3.141592653589793

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we imported only the pi attribute from the math module.

## Import all names

In Python, 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)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

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.

## The dir() built-in function

In Python, we can use the dir() function to list all the function names in a module.

For example, earlier we have defined a function add() in the module example.

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\_\_

# Output: '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 tutorial, we'll learn to create, import, and use Python packages in a program with the help of examples.

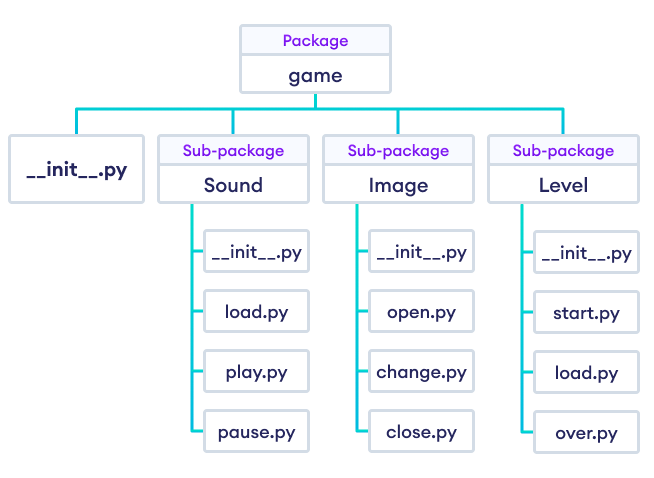
A package is a container that contains various functions to perform specific tasks. For example, the math package includes the sqrt() function to perform the square root of a number.

While working on big projects, we have to deal with a large amount of code, and writing everything together in the same file will make our code look messy. Instead, we can separate our code into multiple files by keeping the related code together in packages.

Now, we can use the package whenever we need it in our projects. This way we can also reuse our code.

## Package Model Structure in Python Programming

Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.

Game Package Model Structure

**Note**: 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.

## Importing module from a package

In Python, 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)

#### Import Without Package Prefix

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)

#### Import Required Functionality Only

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 data types**

Python numbers

Python list

Python tuple

Python string

Python set

Python dictionary

# Python Numbers, Type Conversion and Mathematics

In this tutorial, you will learn about Python Number, Type Conversion, and Mathematics with the help of examples.

The number [data types](https://www.programiz.com/python-programming/variables-datatypes) are used to store the numeric values.

Python supports integers, floating-point numbers and complex numbers. They are defined as int, float, and complex classes in Python.

* int - holds signed integers of non-limited length.
* float - holds floating decimal points and it's accurate up to **15** decimal places.
* complex - holds complex numbers.

## Python Numeric Data Type

Integers and floating points are separated by the presence or absence of a decimal point. For instance,

* **5** is an integer
* **5.42** 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.

We can use the type() function to know which class a variable or a value belongs to.

Let's see an example,

num1 = 5

print(num1, 'is of type', type(num1))

num2 = 5.42

print(num2, 'is of type', type(num2))

num3 = 8+2j

print(num3, 'is of type', type(num3))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

5 is of type <class 'int'>

5.42 is of type <class 'float'>

(8+2j) is of type <class 'complex'>

In the above example, we have created three variables named num1, num2 and num3 with values **5**, **5.42**, and 8+2j respectively.

We have also used the type() function to know which class a certain variable belongs to. Since,

* **5** is an integer value, type() returns int as the class of num1 i.e <class 'int'>
* **5.42** is a floating value, type() returns float as the class of num2 i.e <class 'float'>
* 1 + 2j is a complex number, type() returns complex as the class of num3 i.e <class 'complex'>

## Number Systems

The numbers we deal with every day are of the decimal **(base 10)** number system.

But computer programmers need to work with binary **(base 2)**, hexadecimal **(base 16)** and octal **(base 8)** number systems.

In Python, we can represent these numbers by appropriately placing a prefix before that number. The following table lists these prefixes.

|  |  |
| --- | --- |
| Number System | Prefix |
| Binary | 0b or 0B |
| Octal | 0o or 0O |
| Hexadecimal | 0x or 0X |

Here are some examples

print(0b1101011) # prints 107

print(0xFB + 0b10) # prints 253

print(0o15) # prints 13

[Run Code](https://www.programiz.com/python-programming/online-compiler)

## Type Conversion in Python

In programming, type conversion is the process of converting one type of number into another.

Operations like addition, subtraction convert integers to float implicitly (automatically), if one of the operands is float. For example,

print(1 + 2.0) # prints 3.0

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we can see above that **1** (integer) is converted into **1.0** (float) for addition and the result is also a floating point number.

#### Explicit Type Conversion

We can also use built-in functions like int(), float() and complex() to convert between types explicitly. These functions can even convert from [strings](https://www.programiz.com/python-programming/string).

num1 = int(2.3)

print(num1) # prints 2

num2 = int(-2.8)

print(num2) # prints -2

num3 = float(5)

print(num3) # prints 5.0

num4 = complex('3+5j')

print(num4) # prints (3 + 5j)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, when converting from float to integer, the number gets truncated (decimal parts are removed).

Similarly when converting from integer to float, .0 is postfixed to the number.

## Python Random Module

Python offers the random module to generate random numbers or to pick a random item from an iterator.

First we need to import the random module. For example,

import random

print(random.randrange(10, 20))

list1 = ['a', 'b', 'c', 'd', 'e']

# get random item from list1

print(random.choice(list1))

# Shuffle list1

random.shuffle(list1)

# Print the shuffled list1

print(list1)

# Print random element

print(random.random())

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

15

a

['d', 'b', 'c', 'e', 'a']

0.6716121217631744

To learn more about the random module, visit Python [Random Module](https://www.programiz.com/python-programming/modules/random).

## Python Mathematics

Python offers the math module to carry out different mathematics like trigonometry, logarithms, probability and statistics, etc. For example,

import math

print(math.pi)

print(math.cos(math.pi))

print(math.exp(10))

print(math.log10(1000))

print(math.sinh(1))

print(math.factorial(6))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

3.141592653589793

-1.0

22026.465794806718

3.0

1.1752011936438014

720

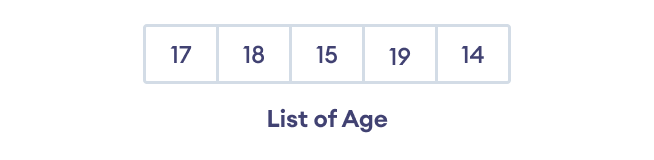
Here is the full list of functions and attributes available in the [Python math module](https://www.programiz.com/python-programming/modules/math).

# Python List

In this tutorial, we will learn about Python lists (creating lists, changing list items, removing items and other list operations) with the help of examples.

In Python, lists are used to store multiple data at once. For example,

Suppose we need to record the ages of **5** students. Instead of creating **5** separate variables, we can simply create a list:

Lists Elements

## Create a Python List

A list is created in Python by placing items inside [], separated by commas . For example,

# A list with 3 integers

numbers = [1, 2, 5]

print(numbers)

# Output: [1, 2, 5]

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we have created a list named numbers with **3** integer items.

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

# empty list

my\_list = []

# list with mixed data types

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

## Access Python List Elements

In Python, each item in a list is associated with a number. The number is known as a list index.

We can access elements of an array using the index number **(0, 1, 2 …)**. For example,

languages = ["Python", "Swift", "C++"]

# access item at index 0

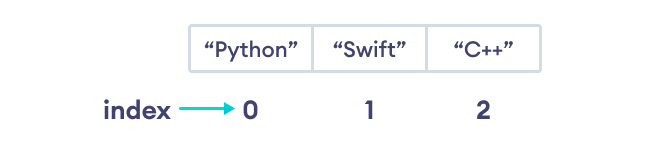
print(languages[0]) # Python

# access item at index 2

print(languages[2]) # C++

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have created a list named languages.

List Indexing in Python

Here, we can see each list item is associated with the index number. And, we have used the index number to access the items.

**Note:** The list index always starts with **0**. Hence, the first element of a list is present at index **0**, not **1**.

## Negative Indexing in Python

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.

Let's see an example,

languages = ["Python", "Swift", "C++"]

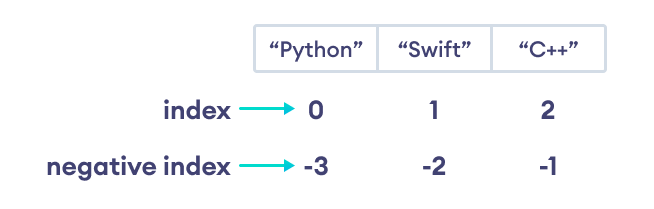
# access item at index 0

print(languages[-1]) # C++

# access item at index 2

print(languages[-3]) # Python

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Python Negative Indexing

**Note**: If the specified index does not exist in the list, Python throws the IndexError exception.

## Slicing of a Python List

In Python it is possible to access a section of items from the list using the slicing operator :, not just a single item. For example,

# List slicing in Python

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

# items from index 2 to index 4

print(my\_list[2:5])

# items from index 5 to end

print(my\_list[5:])

# items beginning to end

print(my\_list[:])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

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

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

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

Here,

* my\_list[2:5] returns a list with items from index **2** to index **4**.
* my\_list[5:] returns a list with items from index **5** to the end.
* my\_list[:] returns all list items

**Note**: When we slice lists, the start index is inclusive but the end index is exclusive.

## Add Elements to a Python List

Python List provides different methods to add items to a list.

**1. Using append()**

The [append()](https://www.programiz.com/python-programming/methods/list/append) method adds an item at the end of the list. For example,

numbers = [21, 34, 54, 12]

print("Before Append:", numbers)

# using append method

numbers.append(32)

print("After Append:", numbers)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Before Append: [21, 34, 54, 12]

After Append: [21, 34, 54, 12, 32]

In the above example, we have created a list named numbers. Notice the line,

numbers.append(32)

Here, append() adds **32** at the end of the array.

**2. Using extend()**

We use the [extend()](https://www.programiz.com/python-programming/methods/list/extend) method to add all items of one list to another. For example,

prime\_numbers = [2, 3, 5]

print("List1:", prime\_numbers)

even\_numbers = [4, 6, 8]

print("List2:", even\_numbers)

# join two lists

prime\_numbers.extend(even\_numbers)

print("List after append:", prime\_numbers)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

List1: [2, 3, 5]

List2: [4, 6, 8]

List after append: [2, 3, 5, 4, 6, 8]

In the above example, we have two lists named prime\_numbers and even\_numbers. Notice the statement,

prime\_numbers.extend(even\_numbers)

Here, we are adding all elements of even\_numbers to prime\_numbers.

## Change List Items

Python lists are mutable. Meaning lists are changeable. And, we can change items of a list by assigning new values using = operator. For example,

languages = ['Python', 'Swift', 'C++']

# changing the third item to 'C'

languages[2] = 'C'

print(languages) # ['Python', 'Swift', 'C']

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, initially the value at index **3** is 'C++'. We then changed the value to 'C' using

languages[2] = 'C'

## Remove an Item From a List

**1. Using del()**

In Python we can use [the del statement](https://www.programiz.com/python-programming/del) to remove one or more items from a list. For example,

languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']

# deleting the second item

del languages[1]

print(languages) # ['Python', 'C++', 'C', 'Java', 'Rust', 'R']

# deleting the last item

del languages[-1]

print(languages) # ['Python', 'C++', 'C', 'Java', 'Rust']

# delete first two items

del languages[0 : 2] # ['C', 'Java', 'Rust']

print(languages)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**2. Using remove()**

We can also use the [remove()](https://www.programiz.com/python-programming/methods/list/remove) method to delete a list item. For example,

languages = ['Python', 'Swift', 'C++', 'C', 'Java', 'Rust', 'R']

# remove 'Python' from the list

languages.remove('Python')

print(languages) # ['Swift', 'C++', 'C', 'Java', 'Rust', 'R']

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, languages.remove('Python') removes 'Python' from the languages list.

## Python List Methods

Python has many useful [list methods](https://www.programiz.com/python-programming/methods/list) that makes it really easy to work with lists.

|  |  |
| --- | --- |
| Method | Description |
| [append()](https://www.programiz.com/python-programming/methods/list/append) | add an item to the end of the list |
| [extend()](https://www.programiz.com/python-programming/methods/list/extend) | add items of lists and other iterables to the end of the list |
| [insert()](https://www.programiz.com/python-programming/methods/list/insert) | inserts an item at the specified index |
| [remove()](https://www.programiz.com/python-programming/methods/list/remove) | removes item present at the given index |
| [pop()](https://www.programiz.com/python-programming/methods/list/pop) | returns and removes item present at the given index |
| [clear()](https://www.programiz.com/python-programming/methods/list/clear) | removes all items from the list |
| [index()](https://www.programiz.com/python-programming/methods/list/index) | returns the index of the first matched item |
| [count()](https://www.programiz.com/python-programming/methods/list/count) | returns the count of the specified item in the list |
| [sort()](https://www.programiz.com/python-programming/methods/list/sort) | sort the list in ascending/descending order |
| [reverse()](https://www.programiz.com/python-programming/methods/list/reverse) | reverses the item of the list |
| [copy()](https://www.programiz.com/python-programming/methods/list/copy) | returns the shallow copy of the list |

## Iterating through a List

We can use the [for loop](https://www.programiz.com/python-programming/for-loop) to iterate over the elements of a list. For example,

languages = ['Python', 'Swift', 'C++']

# iterating through the list

for language in languages:

print(language)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Python

Swift

C++

## Check if an Item Exists in the Python List

We use the in keyword to check if an item exists in the list or not. For example,

languages = ['Python', 'Swift', 'C++']

print('C' in languages) # False

print('Python' in languages) # True

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here,

* 'C' is not present in languages, 'C' in languages evaluates to False.
* 'Python' is present in languages, 'Python' in languages evaluates to True.

## Python List Length

In Python, we use the len() function to find the number of elements present in a list. For example,

languages = ['Python', 'Swift', 'C++']

print("List: ", languages)

print("Total Elements: ", len(languages)) # 3

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

List: ['Python', 'Swift', 'C++']

Total Elements: 3

## Python List Comprehension

[List comprehension](https://www.programiz.com/python-programming/list-comprehension) is a concise and elegant way to create lists.

A list comprehension consists of an expression followed by [the 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 by power of **2**.

numbers = [number\*number for number in range(1, 6)]

print(numbers)

# Output: [1, 4, 9, 16, 25]

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have used the list comprehension to make a list with each item being increased by power of **2.** Notice the code,

[number\*x for x in range(1, 6)]

The code above means to create a list of number\*number where number takes values from **1 to 5**

The code above,

numbers = [x\*x for x in range(1, 6)]

is equivalent to

numbers = []

for x in range(1, 6):

numbers.append(x \* x)

# Python Tuple

In this tutorial, we'll learn about Python Tuples with the help of examples.

A tuple in Python is similar to a [list](https://www.programiz.com/python-programming/list). The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

## 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)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

()

(1, 2, 3)

(1, 'Hello', 3.4)

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

In the above example, we have created different types of tuples and stored different data items inside them.

As mentioned earlier, we can also create tuples without using parentheses:

my\_tuple = 1, 2, 3

my\_tuple = 1, "Hello", 3.4

## Create a Python Tuple With one Element

In Python, 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 a tuple,

var1 = ("Hello") # string

var2 = ("Hello",) # tuple

We can use the type() function to know which class a variable or a value belongs to.

var1 = ("hello")

print(type(var1)) # <class 'str'>

# Creating a tuple having one element

var2 = ("hello",)

print(type(var2)) # <class 'tuple'>

# Parentheses is optional

var3 = "hello",

print(type(var3)) # <class 'tuple'>

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here,

* ("hello") is a string so type() returns str as class of var1 i.e. <class 'str'>
* ("hello",) and "hello", both are tuples so type() returns tuple as class of var1 i.e. <class 'tuple'>

## Access Python Tuple Elements

Like a [list](https://www.programiz.com/python-programming/list), each element of a tuple is represented by index numbers **(0, 1, ...)** where the first element is at index **0**.

We use the index number to access tuple elements. For example,

### 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

letters = ("p", "r", "o", "g", "r", "a", "m", "i", "z")

print(letters[0]) # prints "p"

print(letters[5]) # prints "a"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example,

* letters[0] - accesses the first element
* letters[5] - accesses the sixth element

### 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. For example,

# accessing tuple elements using negative indexing

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

print(letters[-1]) # prints 'z'

print(letters[-3]) # prints 'm'

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example,

* letters[-1] - accesses last element
* letters[-3] - accesses third last element

### 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 index

print(my\_tuple[1:4]) # prints ('r', 'o', 'g')

# elements beginning to 2nd

print(my\_tuple[:-7]) # prints ('p', 'r')

# elements 8th to end

print(my\_tuple[7:]) # prints ('i', 'z')

# elements beginning to end

print(my\_tuple[:]) # Prints ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

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

('p', 'r')

('i', 'z')

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

Here,

* my\_tuple[1:4] returns a tuple with elements from index **1** to index **3**.
* my\_tuple[:-7] returns a tuple with elements from beginning to index **2**.
* my\_tuple[7:] returns a tuple with elements from index **7** to the end.
* my\_tuple[:] returns all tuple items.

**Note**: When we slice lists, the start index is inclusive but the end index is exclusive.

## Python Tuple Methods

In Python ,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')) # prints 2

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

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here,

* my\_tuple.count('p') - counts total number of 'p' in my\_tuple
* my\_tuple.index('l') - returns the first occurrence of 'l' in my\_tuple

## Iterating through a Tuple in Python

We can use the [for loop](https://www.programiz.com/python-programming/for-loop) to iterate over the elements of a tuple. For example,

languages = ('Python', 'Swift', 'C++')

# iterating through the tuple

for language in languages:

print(language)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Python

Swift

C++

## Check if an Item Exists in the Python Tuple

We use the in keyword to check if an item exists in the tuple or not. For example,

languages = ('Python', 'Swift', 'C++')

print('C' in languages) # False

print('Python' in languages) # True

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here,

* 'C' is not present in languages, 'C' in languages evaluates to False.
* 'Python' is present in languages, 'Python' in languages evaluates to True.

## Advantages of Tuple over List in Python

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:

* 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 a 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

In this article, we will learn about the Python Strings with the help of examples.

In computer programming, a string is a sequence of characters. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

We use single quotes or double quotes to represent a string in Python. For example,

# create a string using double quotes

string1 = "Python programming"

# create a string using single quotes

string1 = 'Python programming'

Here, we have created a string variable named string1. The variable is initialized with the string Python Programming.

## Example: Python String

# create string type variables

name = "Python"

print(name)

message = "I love Python."

print(message)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Python

I love Python.

In the above example, we have created string-type variables: name and message with values "Python" and "I love Python" respectively.

Here, we have used double quotes to represent strings but we can use single quotes too.

## Access String Characters in Python

We can access the characters in a string in three ways.

* **Indexing:** One way is to treat strings as a [list](https://www.programiz.com/python-programming/list) and use index values. For example,

greet = 'hello'

# access 1st index element

print(greet[1]) # "e"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

* **Negative Indexing:** Similar to a list, Python allows [negative indexing](https://www.programiz.com/python-programming/list#negative-indexing) for its strings. For example,

greet = 'hello'

# access 4th last element

print(greet[-4]) # "e"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

* **Slicing:** Access a range of characters in a string by using the slicing operator colon :. For example,

greet = 'Hello'

# access character from 1st index to 3rd index

print(greet[1:4]) # "ell"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

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

## Python Strings are immutable

In Python, strings are immutable. That means the characters of a string cannot be changed. For example,

message = 'Hola Amigos'

message[0] = 'H'

print(message)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

TypeError: 'str' object does not support item assignment

However, we can assign the variable name to a new string. For example,

message = 'Hola Amigos'

# assign new string to message variable

message = 'Hello Friends'

prints(message); # prints "Hello Friends"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

## Python Multiline String

We can also create a multiline string in Python. For this, we use triple double quotes """ or triple single quotes '''. For example,

# multiline string

message = """

Never gonna give you up

Never gonna let you down

"""

print(message)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Never gonna give you up

Never gonna let you down

In the above example, anything inside the enclosing triple-quotes is one multiline string.

## Python String Operations

There are many operations that can be performed with strings which makes it one of the most used [data types](https://www.programiz.com/python-programming/variables-datatypes) in Python.

### 1. Compare Two Strings

We use the == operator to compare two strings. If two strings are equal, the operator returns True. Otherwise, it returns False. For example,

str1 = "Hello, world!"

str2 = "I love Python."

str3 = "Hello, world!"

# compare str1 and str2

print(str1 == str2)

# compare str1 and str3

print(str1 == str3)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

False

True

In the above example,

* str1 and str2 are not equal. Hence, the result is False.
* str1 and str3 are equal. Hence, the result is True.

### 2. Join Two or More Strings

In Python, we can join (concatenate) two or more strings using the + operator.

greet = "Hello, "

name = "Jack"

# using + operator

result = greet + name

print(result)

# Output: Hello, Jack

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have used the + operator to join two strings: greet and name.

## Iterate Through a Python String

We can iterate through a string using a [for loop](https://www.programiz.com/python-programming/for-loop). For example,

greet = 'Hello'

# iterating through greet string

for letter in greet:

print(letter)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

H

e

l

l

o

## Python String Length

In Python, we use the len() method to find the length of a string. For example,

greet = 'Hello'

# count length of greet string

print(len(greet))

# Output: 5

[Run Code](https://www.programiz.com/python-programming/online-compiler)

## String Membership Test

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

print('a' in 'program') # True

print('at' not in 'battle') False

[Run Code](https://www.programiz.com/python-programming/online-compiler)

## Methods of Python String

Besides those mentioned above, there are various [string methods](https://www.programiz.com/python-programming/methods/string) present in Python. Here are some of those methods:

|  |  |
| --- | --- |
| Methods | Description |
| [upper()](https://www.programiz.com/python-programming/methods/string/upper) | converts the string to uppercase |
| [lower()](https://www.programiz.com/python-programming/methods/string/lower) | converts the string to lowercase |
| [partition()](https://www.programiz.com/python-programming/methods/string/partition) | returns a tuple |
| [replace()](https://www.programiz.com/python-programming/methods/string/replace) | replaces substring inside |
| [find()](https://www.programiz.com/python-programming/methods/string/find) | returns the index of first occurrence of substring |
| [rstrip()](https://www.programiz.com/python-programming/methods/string/rstrip) | removes trailing characters |
| [split()](https://www.programiz.com/python-programming/methods/string/split) | splits string from left |
| [startswith()](https://www.programiz.com/python-programming/methods/string/startswith) | checks if string starts with the specified string |
| [isnumeric()](https://www.programiz.com/python-programming/methods/string/isnumeric) | checks numeric characters |
| [index()](https://www.programiz.com/python-programming/methods/string/index) | returns index of substring |

## Escape Sequences in Python

The escape sequence is used to escape some of the characters present inside a string.

Suppose we need to include both double quote and single quote inside a string,

example = "He said, "What's there?""

print(example) # throws error

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Since strings are represented by single or double quotes, the compiler will treat "He said, " as the string. Hence, the above code will cause an error.

To solve this issue, we use the escape character \ in Python.

# escape double quotes

example = "He said, \"What's there?\""

# escape single quotes

example = 'He said, "What\'s there?"'

print(example)

# Output: He said, "What's there?"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

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

|  |  |
| --- | --- |
| Escape Sequence | Description |
| \\ | 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 |

## Python String Formatting (f-Strings)

Python **f-Strings** make it really easy to print values and variables. For example,

name = 'Cathy'

country = 'UK'

print(f'{name} is from {country}')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Cathy is from UK

Here, f'{name} is from {country}' is an **f-string**.

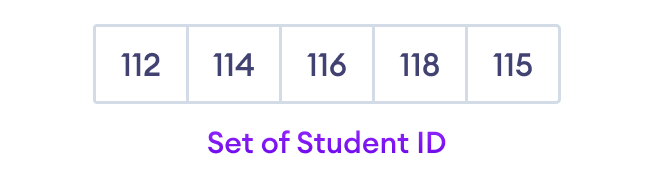
This new formatting syntax is powerful and easy to use. From now on, we will use f-Strings to print strings and variables.

# Python Sets

In this tutorial, we will learn Set and its various operations in Python with the help of examples.

A set is a collection of unique data. That is, elements of a set cannot be duplicate. For example,

Suppose we want to store information about **student IDs**. Since **student IDs** cannot be duplicate, we can use a set.

Python Set Elements

## Create a Set in Python

In Python, we create sets by placing all the elements inside curly braces {}, separated by comma.

A set 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.

Let's see an example,

# create a set of integer type

student\_id = {112, 114, 116, 118, 115}

print('Student ID:', student\_id)

# create a set of string type

vowel\_letters = {'a', 'e', 'i', 'o', 'u'}

print('Vowel Letters:', vowel\_letters)

# create a set of mixed data types

mixed\_set = {'Hello', 101, -2, 'Bye'}

print('Set of mixed data types:', mixed\_set)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Student ID: {112, 114, 115, 116, 118}

Vowel Letters: {'u', 'a', 'e', 'i', 'o'}

Set of mixed data types: {'Hello', 'Bye', 101, -2}

In the above example, we have created different types of sets by placing all the elements inside the curly braces {}.

**Note:** When you run this code, you might get output in a different order. This is because the set has no particular order.

## Create an Empty Set in Python

Creating an empty set is a bit tricky. Empty curly braces {} will make an empty [dictionary](https://www.programiz.com/python-programming/dictionary) in Python.

To make a set without any elements, we use the [set()](https://www.programiz.com/python-programming/methods/built-in/set) function without any argument. For example,

# create an empty set

empty\_set = set()

# create an empty dictionary

empty\_dictionary = { }

# check data type of empty\_set

print('Data type of empty\_set:', type(empty\_set))

# check data type of dictionary\_set

print('Data type of empty\_dictionary', type(empty\_dictionary))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Data type of empty\_set: <class 'set'>

Data type of empty\_dictionary <class 'dict'>

Here,

* empty\_set - an empty set created using set()
* empty\_dictionary - an empty dictionary created using {}

Finally we have used the type() function to know which class empty\_set and empty\_dictionary belong to.

## Duplicate Items in a Set

Let's see what will happen if we try to include duplicate items in a set.

numbers = {2, 4, 6, 6, 2, 8}

print(numbers) # {8, 2, 4, 6}

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we can see there are no duplicate items in the set as a set cannot contain duplicates.

## Add and Update Set Items 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.

### Add Items to a Set in Python

In Python, we use the add() method to add an item to a set. For example,

numbers = {21, 34, 54, 12}

print('Initial Set:',numbers)

# using add() method

numbers.add(32)

print('Updated Set:', numbers)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Initial Set: {34, 12, 21, 54}

Updated Set: {32, 34, 12, 21, 54}

In the above example, we have created a set named numbers. Notice the line,

numbers.add(32)

Here, add() adds **32** to our set.

### Update Python Set

The update() method is used to update the set with items other collection types (lists, tuples, sets, etc). For example,

companies = {'Lacoste', 'Ralph Lauren'}

tech\_companies = ['apple', 'google', 'apple']

companies.update(tech\_companies)

print(companies)

# Output: {'google', 'apple', 'Lacoste', 'Ralph Lauren'}

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, all the unique elements of tech\_companies are added to the companies set.

## Remove an Element from a Set

We use the discard() method to remove the specified element from a set. For example,

languages = {'Swift', 'Java', 'Python'}

print('Initial Set:',languages)

# remove 'Java' from a set

removedValue = languages.discard('Java')

print('Set after remove():', languages)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Initial Set: {'Python', 'Swift', 'Java'}

Set after remove(): {'Python', 'Swift'}

Here, we have used the discard() method to remove 'Java' from the languages set.

## 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. |

## Iterate Over a Set in Python

fruits = {"Apple", "Peach", "Mango"}

# for loop to access each fruits

for fruit in fruits:

print(fruit)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Mango

Peach

Apple

## Find Number of Set Elements

We can use the len() method to find the number of elements present in a Set. For example,

even\_numbers = {2,4,6,8}

print('Set:',even\_numbers)

# find number of elements

print('Total Elements:', len(even\_numbers))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Set: {8, 2, 4, 6}

Total Elements: 4

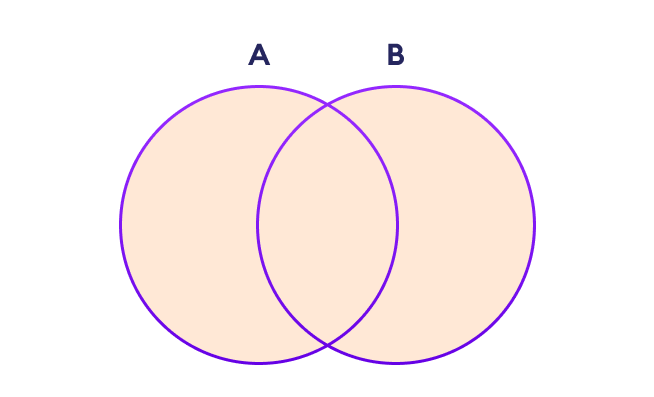
Here, we have used the len() method to find the number of elements present in a Set.

## Python Set Operations

Python Set provides different built-in methods to perform mathematical set operations like union, intersection, subtraction, and symmetric difference.

### Union of Two Sets

The union of two sets **A** and **B** include all the elements of set **A** and **B**.

Set Union in Python

We use the | operator or the union() method to perform the set union operation. For example,

# first set

A = {1, 3, 5}

# second set

B = {0, 2, 4}

# perform union operation using |

print('Union using |:', A | B)

# perform union operation using union()

print('Union using union():', A.union(B))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

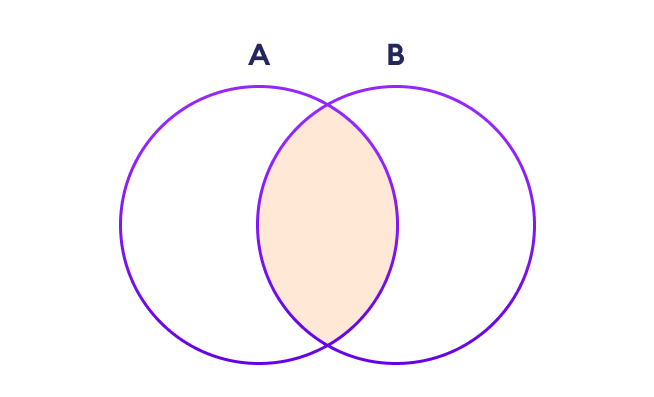
Union using |: {0, 1, 2, 3, 4, 5}

Union using union(): {0, 1, 2, 3, 4, 5}

**Note**: A|B and union() is equivalent to A ⋃ B set operation.

### Set Intersection

The intersection of two sets **A** and **B** include the common elements between set **A** and **B**.

Set Intersection in Python

In Python, we use the & operator or the intersection() method to perform the set intersection operation. For example,

# first set

A = {1, 3, 5}

# second set

B = {1, 2, 3}

# perform intersection operation using &

print('Intersection using &:', A & B)

# perform intersection operation using intersection()

print('Intersection using intersection():', A.intersection(B))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

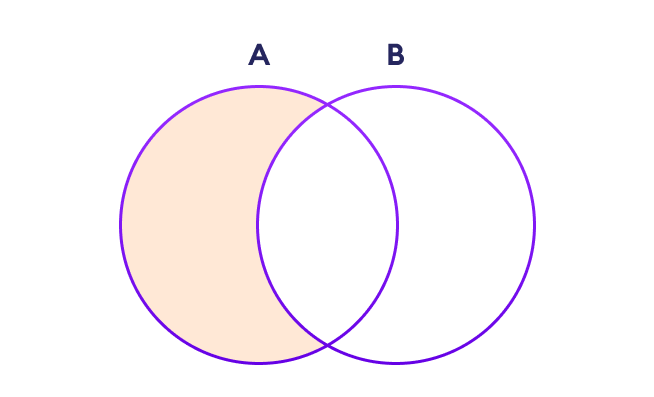
Intersection using &: {1, 3}

Intersection using intersection(): {1, 3}

**Note**: A&B and intersection() is equivalent to A ⋂ B set operation.

### Difference between Two Sets

The difference between two sets **A** and **B** include elements of set **A** that are not present on set **B**.

Set Difference in Python

We use the - operator or the difference() method to perform the difference between two sets. For example,

# first set

A = {2, 3, 5}

# second set

B = {1, 2, 6}

# perform difference operation using &

print('Difference using &:', A - B)

# perform difference operation using difference()

print('Difference using difference():', A.difference(B))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

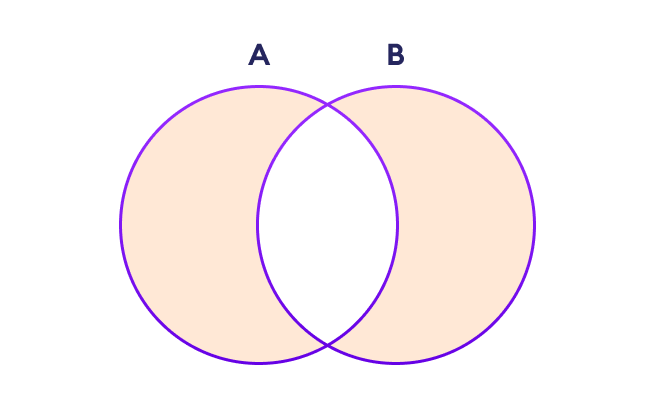
Difference using &: {3, 5}

Difference using difference(): {3, 5}

**Note**: A - B and A.difference(B) is equivalent to A - B set operation.

### Set Symmetric Difference

The symmetric difference between two sets **A** and **B** includes all elements of **A** and **B** without the common elements.

Set Symmetric Difference in Python

In Python, we use the ^ operator or the symmetric\_difference() method to perform symmetric difference between two sets. For example,

# first set

A = {2, 3, 5}

# second set

B = {1, 2, 6}

# perform difference operation using &

print('using ^:', A ^ B)

# using symmetric\_difference()

print('using symmetric\_difference():', A.symmetric\_difference(B))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

using ^: {1, 3, 5, 6}

using symmetric\_difference(): {1, 3, 5, 6}

## Check if two sets are equal

We can use the == operator to check whether two sets are equal or not. For example,

# first set

A = {1, 3, 5}

# second set

B = {3, 5, 1}

# perform difference operation using &

if A == B:

print('Set A and Set B are equal')

else:

print('Set A and Set B are not equal')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Set A and Set B are equal

In the above example, A and B have the same elements, so the condition

if A == B

evaluates to True. Hence, the statement print('Set A and Set B are equal') inside the if is executed.

## 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 |

# Python Dictionary

In this tutorial, you'll learn about Python dictionaries; how they are created, accessing, adding, removing elements from them and various built-in methods.

Python dictionary is an ordered collection (starting from **Python 3.7**) of items. It stores elements in **key/value** pairs. Here, **keys** are unique identifiers that are associated with each **value**.

Let's see an example,

If we want to store information about countries and their capitals, we can create a dictionary with country names as **keys** and capitals as **values**.

|  |  |
| --- | --- |
| Keys | Values |
| Nepal | Kathmandu |
| Italy | Rome |
| England | London |

## Create a dictionary in Python

Here's how we can create a dictionary in Python.

capital\_city = {"Nepal": "Kathmandu", "Italy": "Rome", "England": "London"}

print(capital\_city)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

{'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}

In the above example, we have created a dictionary named capital\_city. Here,

1. **Keys** are "Nepal", "Italy", "England"
2. **Values** are "Kathmandu", "Rome", "London"

**Note**: Here, **keys** and **values** both are of string type. We can also have **keys** and **values** of different data types.

## Example 1: Python Dictionary

# dictionary with keys and values of different data types

numbers = {1: "One", 2: "Two", 3: "Three"}

print(numbers)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

[3: "Three", 1: "One", 2: "Two"]

In the above example, we have created a dictionary named numbers. Here, **keys** are of integer type and **values** are of string type.

## Add Elements to a Python Dictionary

We can add elements to a dictionary using the name of the dictionary with []. For example,

capital\_city = {"Nepal": "Kathmandu", "England": "London"}

print("Initial Dictionary: ",capital\_city)

capital\_city["Japan"] = "Tokyo"

print("Updated Dictionary: ",capital\_city)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Initial Dictionary: {'Nepal': 'Kathmandu', 'England': 'London'}

Updated Dictionary: {'Nepal': 'Kathmandu', 'England': 'London', 'Japan': 'Tokyo'}

In the above example, we have created a dictionary named capital\_city. Notice the line,

capital\_city["Japan"] = "Tokyo"

Here, we have added a new element to capital\_city with **key**: Japan and **value**: Tokyo.

## Change Value of Dictionary

We can also use [] to change the value associated with a particular key. For example,

student\_id = {111: "Eric", 112: "Kyle", 113: "Butters"}

print("Initial Dictionary: ", student\_id)

student\_id[112] = "Stan"

print("Updated Dictionary: ", student\_id)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Initial Dictionary: {111: 'Eric', 112: 'Kyle', 113: 'Butters'}

Updated Dictionary: {111: 'Eric', 112: 'Stan', 113: 'Butters'}

In the above example, we have created a dictionary named student\_id. Initially, the value associated with the key 112 is "Kyle". Now, notice the line,

student\_id[112] = "Stan"

Here, we have changed the value associated with the key 112 to "Stan".

## Accessing Elements from Dictionary

In Python, we use the keys to access their corresponding values. For example,

student\_id = {111: "Eric", 112: "Kyle", 113: "Butters"}

print(student\_id[111]) # prints Eric

print(student\_id[113]) # prints Butters

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we have used the keys to access their corresponding values.

If we try to access the value of a key that doesn't exist, we'll get an error. For example,

student\_id = {111: "Eric", 112: "Kyle", 113: "Butters"}

print(student\_id[211])

# Output: KeyError: 211

[Run Code](https://www.programiz.com/python-programming/online-compiler)

## Removing elements from Dictionary

We use the del statement to remove an element from the dictionary. For example,

student\_id = {111: "Eric", 112: "Kyle", 113: "Butters"}

print("Initial Dictionary: ", student\_id)

del student\_id[111]

print("Updated Dictionary ", student\_id)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Initial Dictionary: {111: 'Eric', 112: 'Kyle', 113: 'Butters'}

Updated Dictionary {112: 'Kyle', 113: 'Butters'}

Here, we have created a dictionary named student\_id. Notice the code,

del student\_id[111]

The del statement removes the element associated with the key 111.

We can also delete the whole dictionary using the del statement,

student\_id = {111: "Eric", 112: "Kyle", 113: "Butters"}

# delete student\_id dictionary

del student\_id

print(student\_id)

# Output: NameError: name 'student\_id' is not defined

[Run Code](https://www.programiz.com/python-programming/online-compiler)

We are getting an error message because we have deleted the student\_id dictionary and student\_id doesn't exist anymore.

## Python Dictionary Methods

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

|  |  |
| --- | --- |
| 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. |
| [sorted()](https://www.programiz.com/python-programming/methods/built-in/sorted) | Return a new sorted list of keys in the dictionary. |
| [clear()](https://www.programiz.com/python-programming/methods/dictionary/clear) | Removes all items from the dictionary. |
| [keys()](https://www.programiz.com/python-programming/methods/dictionary/keys) | Returns a new object of the dictionary's keys. |
| [values()](https://www.programiz.com/python-programming/methods/dictionary/values) | Returns a new object of the dictionary's values |

## 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) # prints True

print(2 not in squares) # prints True

# membership tests for key only not value

print(49 in squares) # prints false

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**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])

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

1

9

25

49

81

Here, we have iterated through each **key** in the squares dictionary using the for loop.

**Python files**

Python file operation

Python directory

Python exception

Exception handling

User – defined exception

# Python File I/O Python File Operation

In this tutorial, we will learn about Python File and its various operations with the help of examples.

A file is a container in computer storage devices used for storing data.

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.

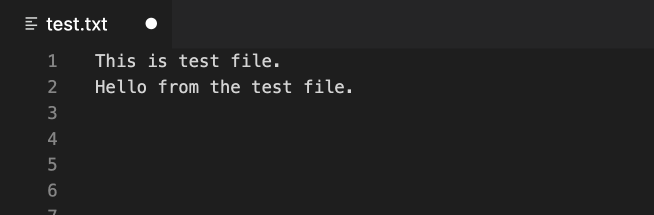
Hence, in Python, a file operation takes place in the following order:

1. Open a file
2. Read or write (perform operation)
3. Close the file

## Opening Files in Python

In Python, we use the open() method to open files.

To demonstrate how we open files in Python, let's suppose we have a file named test.txt with the following content.

Opening Files in Python

Now, let's try to open data from this file using the open() function.

# open file in current directory

file1 = open("test.txt")

Here, we have created a file object named file1. This object can be used to work with files and directories.

By default, the files are open in **read mode** (cannot be modified). The code above is equivalent to

file1 = open("test.txt", "r")

Here, we have explicitly specified the mode by passing the "r" argument which means file is opened for reading.

### Different Modes to Open a File in Python

|  |  |
| --- | --- |
| Mode | Description |
| r | Open a file for reading. (default) |
| w | Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists. |
| x | Open a file for exclusive creation. If the file already exists, the operation fails. |
| a | Open a file for appending at the end of the file without truncating it. Creates a new file if it does not exist. |
| t | Open in text mode. (default) |
| b | Open in binary mode. |
| + | Open a file for updating (reading and writing) |

Here's few simple examples of how to open a file in different modes,

file1 = open("test.txt") # equivalent to 'r' or 'rt'

file1 = open("test.txt",'w') # write in text mode

file1 = open("img.bmp",'r+b') # read and write in binary mode

## Reading Files in Python

After we open a file, we use the read() method to read its contents. For example,

# open a file

file1 = open("test.txt", "r")

# read the file

read\_content = file1.read()

print(read\_content)

**Output**

This is a test file.

Hello from the test file.

In the above example, we have read the test.txt file that is available in our current directory. Notice the code,

read\_content = file1.read

Here, file1.read() reads the test.txt file and is stored in the read\_content variable.

## 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 in Python. For example,

# open a file

file1 = open("test.txt", "r")

# read the file

read\_content = file1.read()

print(read\_content)

# close the file

file1.close()

**Output**

This is a test file.

Hello from the test file.

Here, we have used the close() method to close the file.

After we perform file operation, we should always close the file; it's a good programming practice.

## Exception Handling in Files

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.

Let's see an example,

try:

file1 = open("test.txt", "r")

read\_content = file1.read()

print(read\_content)

finally:

# close the file

file1.close()

Here, we have closed the file in the finally block as finally always executes, and the file will be closed even if an exception occurs.

## Use of with...open Syntax

In Python, we can use the with...open syntax to automatically close the file. For example,

with open("test.txt", "r") as file1:

read\_content = file1.read()

print(read\_content)

**Note**: Since we don't have to worry about closing the file, make a habit of using the with...open syntax.

## Writing to Files in Python

There are two things we need to remember while writing to a file.

* If we try to open a file that doesn't exist, a new file is created.
* If a file already exists, its content is erased, and new content is added to the file.

In order to write into a file in Python, we need to open it in write mode by passing "w" inside open() as a second argument.

Suppose, we don't have a file named **test2.txt**. Let's see what happens if we write contents to the test2.txt file.

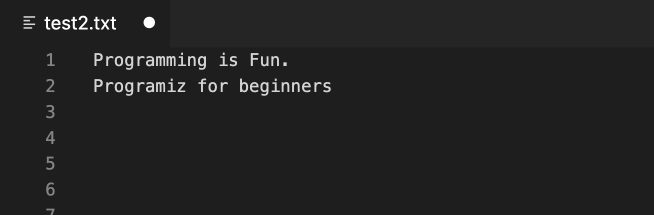
with open(test2.txt', 'w') as file2:

# write contents to the test2.txt file

file2.write('Programming is Fun.')

fil2.write('Programiz for beginners')

Here, a new test2.txt file is created and this file will have contents specified inside the write() method.

Writing to Python Files

## 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 an integer that represents the current position of the file's object. |
| 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 and Files Management

In this tutorial, we'll learn about file and directory management in Python with the help of examples.

A directory is a collection of [files](https://www.programiz.com/python-programming/file-operation) and subdirectories. A directory inside a directory is known as a subdirectory.

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 in Python

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. For example,

import os

print(os.getcwd())

# Output: C:\Program Files\PyScripter

Here, getcwd() returns the current directory in the form of a string.

## Changing Directory in Python

In Python, 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. And we can use both the forward-slash / or the backward-slash \ to separate the path elements.

Let's see an example,

import os

# change directory

os.chdir('C:\\Python33')

print(os.getcwd())

Output: C:\Python33

Here, we have used the chdir() method to change the current working directory and passed a new path as a string to chdir().

## List Directories and Files in Python

All files and sub-directories inside a directory can be retrieved using the listdir() method.

This method takes in a path and returns a list of subdirectories and files in that path.

If no path is specified, it returns the list of subdirectories and files from the current working directory.

import os

print(os.getcwd())

C:\Python33

# list all sub-directories

os.listdir()

['DLLs',

'Doc',

'include',

'Lib',

'libs',

'LICENSE.txt',

'NEWS.txt',

'python.exe',

'pythonw.exe',

'README.txt',

'Scripts',

'tcl',

'Tools']

os.listdir('G:\\')

['$RECYCLE.BIN',

'Movies',

'Music',

'Photos',

'Series',

'System Volume Information']

## Making a New Directory in Python

In Python, 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, rename() takes in two basic arguments:

* the old name as the first argument
* the new name as the second argument.

Let's see an example,

import os

os.listdir()

['test']

# rename a directory

os.rename('test','new\_one')

os.listdir()

['new\_one']

Here, 'test' directory is renamed to 'new\_one' using the rename() method.

## Removing Directory or File in Python

In Python, we can use the remove() method or the rmdir() method to remove a file or directory.

First let's use remove() to delete a file,

import os

# delete "myfile.txt" file

os.remove("myfile.txt")

Here, we have used the remove() method to remove the "myfile.txt" file.

Now let's use rmdir() to delete an empty directory,

import os

# delete the empty directory "mydir"

os.rmdir("mydir")

In order to remove a non-empty directory, we can use the rmtree() method inside the shutil module. For example,

import shutil

# delete "mydir" directory and all of its contents

shutil.rmtree("mydir")

It's important to note that these functions permanently delete the files or directories, so we need to careful when using them.

# Python Exceptions

In this tutorial, we will learn about exceptions in Python. We will cover exceptions and different types of exceptions in Python

An exception is an unexpected event that occurs during program execution. For example,

divide\_by\_zero = 7 / 0

The above code causes an exception as it is not possible to divide a number by **0**.

Let's learn about Python Exceptions in detail.

## Python Logical Errors (Exceptions)

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

For instance, they occur when we

* try to open a file(for reading) that does not exist (FileNotFoundError)
* try to divide a number by zero (ZeroDivisionError)
* try to import a module that does not exist (ImportError) and so on.

Whenever these types of runtime errors occur, Python creates an exception object.

If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Let's look at how Python treats these errors:

divide\_numbers = 7 / 0

prit(divide\_numbers)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Traceback (most recent call last):

File "<string>", line 1, in <module>

ZeroDivisionError: division by zero

Here, while trying to divide 7 / 0, the program throws a system exception ZeroDivisionError

## Python Built-in Exceptions

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur.

We can view all the built-in exceptions using the built-in local() function as follows:

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

[Run Code](https://www.programiz.com/python-programming/online-compiler)

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

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

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

If required, we can also define our own exceptions in Python. To learn more about them, visit [Python User-defined Exceptions](https://www.programiz.com/python-programming/user-defined-exception).

We can handle these built-in and user-defined exceptions in Python using try, except and finally statements. To learn more about them, visit [Python try, except and finally statements](https://www.programiz.com/python-programming/exception-handling).

## Python Error and Exception

**Errors** represent conditions such as compilation error, syntax error, error in the logical part of the code, library incompatibility, infinite recursion, etc.

Errors are usually beyond the control of the programmer and we should not try to handle errors.

**Exceptions** can be caught and handled by the program.

Now we know about exceptions, we will learn about handling exceptions in the next tutorial.

# Python Exception Handling

In the tutorial, we will learn about different approaches of exception handling in Python with the help of examples.

In the last tutorial, we learned about [Python exceptions](https://www.programiz.com/python-programming/exceptions). We know that exceptions abnormally terminate the execution of a program.

This is why it is important to handle exceptions. In Python, we use the try...except block

## Python try...except Block

The try...except block is used to handle exceptions in Python. Here's the syntax of try...except block:

try:

# code that may cause exception

except:

# code to run when exception occurs

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by an except block.

When an exception occurs, it is caught by the except block. The except block cannot be used without the try block.

### Example: Exception Handling Using try...except

try:

numerator = 10

denominator = 0

result = numerator/denominator

print(result)

except:

print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the example, we are trying to divide a number by **0**. Here, this code generates an exception.

To handle the exception, we have put the code, result = numerator/denominator inside the try block. Now when an exception occurs, the rest of the code inside the try block is skipped.

The except block catches the exception and statements inside the except block are executed.

If none of the statements in the try block generates an exception, the except block is skipped.

## Catching Specific Exceptions in Python

For each try block, there can be zero or more except blocks. Multiple except blocks allow us to handle each exception differently.

The argument type of each except block indicates the type of exception that can be handled by it. For example,

try:

even\_numbers = [2,4,6,8]

print(even\_numbers[5])

except ZeroDivisionError:

print("Denominator cannot be 0.")

except IndexError:

print("Index Out of Bound.")

# Output: Index Out of Bound

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In this example, we have created a list named even\_numbers.

Since the list index starts from **0**, the last element of the list is at index **3**. Notice the statement,

print(even\_numbers[5])

Here, we are trying to access a value to the index **5**. Hence, IndexError exception occurs.

When the IndexError exception occurs in the try block,

* The ZeroDivisionError exception is skipped.
* The set of code inside the IndexError exception is executed.

## Python try with else clause

In some situations, we might want to run a certain block of code if the code block inside try runs without any errors.

For these cases, you can use the optional else keyword with the try statement.

Let's look at an example:

# program to print the reciprocal of even numbers

try:

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

assert num % 2 == 0

except:

print("Not an even number!")

else:

reciprocal = 1/num

print(reciprocal)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

If we pass an odd number:

Enter a number: 1

Not an even number!

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

Enter a number: 4

0.25

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

Enter a number: 0

Traceback (most recent call last):

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

reciprocal = 1/num

ZeroDivisionError: division by zero

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

## Python try...finally

In Python, the finally block is always executed no matter whether there is an exception or not.

The finally block is optional. And, for each try block, there can be only one finally block.

Let's see an example,

try:

numerator = 10

denominator = 0

result = numerator/denominator

print(result)

except:

print("Error: Denominator cannot be 0.")

finally:

print("This is finally block.")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Error: Denominator cannot be 0.

This is finally block.

In the above example, we are dividing a number by **0** inside the try block. Here, this code generates an exception.

The exception is caught by the except block. And, then the finally block is executed.

# Python Custom Exceptions

In this tutorial, we will learn how to define custom exceptions depending upon our requirements with the help of examples.

In the previous tutorial, we learned about different [built-in exceptions](https://www.programiz.com/python-programming/exceptions) in Python and why it is important to handle exceptions. .

However, sometimes we may need to create our own custom exceptions that serve our purpose.

## Defining Custom Exceptions

In Python, we can define custom exceptions by creating a new class that is derived from the built-in Exception class.

Here's the syntax to define custom exceptions,

class CustomError(Exception):

...

pass

try:

...

except CustomError:

...

Here, CustomError is a user-defined error which inherits from the Exception class.

Note:

* When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file.
* Many standard modules define their exceptions separately as exceptions.py or errors.py (generally but not always).

## Example: Python User-Defined Exception

# define Python user-defined exceptions

class InvalidAgeException(Exception):

"Raised when the input value is less than 18"

pass

# you need to guess this number

number = 18

try:

input\_num = int(input("Enter a number: "))

if input\_num < number:

raise InvalidAgeException

else:

print("Eligible to Vote")

except InvalidAgeException:

print("Exception occurred: Invalid Age")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

If the user input input\_num is greater than **18**,

Enter a number: 45

Eligible to Vote

If the user input input\_num is smaller than **18**,

Enter a number: 14

Exception occurred: Invalid Age

In the above example, we have defined the custom exception InvalidAgeException by creating a new class that is derived from the built-in Exception class.

Here, when input\_num is smaller than **18**, this code generates an exception.

When an exception occurs, the rest of the code inside the try block is skipped.

The except block catches the user-defined InvalidAgeException exception and statements inside the except block are executed.

## Customizing Exception Classes

We can further customize this class to accept other arguments as per our needs.

To learn about customizing the Exception classes, you need to have the basic knowledge of Object-Oriented programming.

Visit [Python Object Oriented Programming](https://www.programiz.com/python-programming/object-oriented-programming) to learn about Object-Oriented programming in Python.

Let's see an example,

class SalaryNotInRangeError(Exception):

"""Exception raised for errors in the input salary.

Attributes:

salary -- input salary which caused the error

message -- explanation of the error

"""

def \_\_init\_\_(self, salary, message="Salary is not in (5000, 15000) range"):

self.salary = salary

self.message = message

super().\_\_init\_\_(self.message)

salary = int(input("Enter salary amount: "))

if not 5000 < salary < 15000:

raise SalaryNotInRangeError(salary)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Enter salary amount: 2000

Traceback (most recent call last):

File "<string>", line 17, in <module>

raise SalaryNotInRangeError(salary)

\_\_main\_\_.SalaryNotInRangeError: Salary is not in (5000, 15000) range

Here, we have overridden the constructor of the Exception class to accept our own custom arguments salary and message.

Then, the constructor of the parent Exception class is called manually with the self.message argument using super().

The custom self.salary attribute is defined to be used later.

The inherited \_\_str\_\_ method of the Exception class is then used to display the corresponding message when SalaryNotInRangeError is raised.

**Python object & class**

Python OOP

Classes & Object

Python inheritance

Multiple inheritance

Operator overloading

# Python Object Oriented Programming

In this tutorial, we’ll learn about Object-Oriented Programming (OOP) in Python with the help of examples.

Python is a versatile programming language that supports various programming styles, including object-oriented programming (OOP) through the use of **objects** and **classes**.

An object is any entity that has **attributes** and **behaviors**. For example, a parrot is an object. It has

* **attributes** - name, age, color, etc.
* **behavior** - dancing, singing, etc.

Similarly, a class is a blueprint for that object.

## Python Class and Object

class Parrot:

# class attribute

name = ""

age = 0

# create parrot1 object

parrot1 = Parrot()

parrot1.name = "Blu"

parrot1.age = 10

# create another object parrot2

parrot2 = Parrot()

parrot2.name = "Woo"

parrot2.age = 15

# access attributes

print(f"{parrot1.name} is {parrot1.age} years old")

print(f"{parrot2.name} is {parrot2.age} years old")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Blu is 10 years old

Woo is 15 years old

In the above example, we created a class with the name Parrot with two attributes: name and age.

Then, we create instances of the Parrot class. Here, parrot1 and parrot2 are references (value) to our new objects.

We then accessed and assigned different values to the instance attributes using the objects name and the . notation.

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

## Python 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 2: Use of Inheritance in Python

# base class

class Animal:

def eat(self):

print( "I can eat!")

def sleep(self):

print("I can sleep!")

# derived class

class Dog(Animal):

def bark(self):

print("I can bark! Woof woof!!")

# Create object of the Dog class

dog1 = Dog()

# Calling members of the base class

dog1.eat()

dog1.sleep()

# Calling member of the derived class

dog1.bark();

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I can eat!

I can sleep!

I can bark! Woof woof!!

Here, dog1 (the object of derived class Dog) can access members of the base class Animal. It's because Dog is inherited from Animal.

# Calling members of the Animal class

dog1.eat()

dog1.sleep()

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

## Python Encapsulation

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of attributes and methods inside a single class.

It prevents outer classes from accessing and changing attributes and methods of a class. This also helps to achieve **data hiding**.

In Python, we denote private attributes using underscore as the prefix i.e single \_ or double \_\_. For example,

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()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**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. Here, notice the code

c.\_\_maxprice = 1000

Here, we have tried to modify the value of \_\_maxprice outside of the class. However, since \_\_maxprice is a private variable, this modification is not seen on the output.

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 another important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

Let's see an example,

class Polygon:

# method to render a shape

def render(self):

print("Rendering Polygon...")

class Square(Polygon):

# renders Square

def render(self):

print("Rendering Square...")

class Circle(Polygon):

# renders circle

def render(self):

print("Rendering Circle...")

# create an object of Square

s1 = Square()

s1.render()

# create an object of Circle

c1 = Circle()

c1.render()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Rendering Square...

Rendering Circle...

In the above example, we have created a superclass: Polygon and two subclasses: Square and Circle. Notice the use of the render() method.

The main purpose of the render() method is to render the shape. However, the process of rendering a square is different from the process of rendering a circle.

Hence, the render() method behaves differently in different classes. Or, we can say render() is polymorphic.

## 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

In this tutorial, we will learn about Python classes and objects with the help of examples.

In the last tutorial, we learned about [Python OOP](https://www.programiz.com/python-programming/object-oriented-programming). We know that python also supports the concept of objects and classes.

An object is simply a collection of data (variables) and methods (functions). Similarly, a class is a blueprint for that object.

Before we learn about objects, let's first know about classes in Python.

## Python Classes

A class is considered as a blueprint of objects. We can think of the 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.

Since many houses can be made from the same description, we can create many objects from a class.

## Define Python Class

We use the class keyword to create a class in Python. For example,

class ClassName:

# class definition

Here, we have created a class named ClassName.

Let's see an example,

class Bike:

name = ""

gear = 0

Here,

* Bike - the name of the class
* name/gear - variables inside the class with default values "" and **0** respectively.

**Note**: The variables inside a class are called attributes.

## Python Objects

An object is called an instance of a class. For example, suppose Bike is a class then we can create objects like bike1, bike2, etc from the class.

Here's the syntax to create an object.

objectName = ClassName()

Let's see an example,

# create class

class Bike:

name = ""

gear = 0

# create objects of class

bike1 = Bike()

Here, bike1 is the object of the class. Now, we can use this object to access the class attributes.

## Access Class Attributes Using Objects

We use the . notation to access the attributes of a class. For example,

# modify the name attribute

bike1.name = "Mountain Bike"

# access the gear attribute

bike1.gear

Here, we have used bike1.name and bike1.gear to change and access the value of name and gear attribute respectively.

## Example 1: Python Class and Objects

# define a class

class Bike:

name = ""

gear = 0

# create object of class

bike1 = Bike()

# access attributes and assign new values

bike1.gear = 11

bike1.name = "Mountain Bike"

print(f"Name: {bike1.name}, Gears: {bike1.gear} ")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Name: Mountain Bike, Gears: 11

In the above example, we have defined the class named Bike with two attributes: name and gear.

We have also created an object bike1 of the class Bike.

Finally, we have accessed and modified the attributes of an object using the . notation.

## Create Multiple Objects of Python Class

We can also create multiple objects from a single class. For example,

# define a class

class Employee:

# define an attribute

employee\_id = 0

# create two objects of the Employee class

employee1 = Employee()

employee2 = Employee()

# access attributes using employee1

employee1.employeeID = 1001

print(f"Employee ID: {employee1.employeeID}")

# access attributes using employee2

employee2.employeeID = 1002

print(f"Employee ID: {employee2.employeeID}")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Employee ID: 1001

Employee ID: 1002

In the above example, we have created two objects employee1 and employee2 of the Employee class.

## Python Methods

We can also define a function inside a Python class. A [Python Function](https://www.programiz.com/python-programming/function) defined inside a class is called a method.

Let's see an example,

# create a class

class Room:

length = 0.0

breadth = 0.0

# method to calculate area

def calculate\_area(self):

print("Area of Room =", self.length \* self.breadth)

# create object of Room class

study\_room = Room()

# assign values to all the attributes

study\_room.length = 42.5

study\_room.breadth = 30.8

# access method inside class

study\_room.calculate\_area()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Area of Room = 1309.0

In the above example, we have created a class named Room with:

* **Attributes**: length and breadth
* **Method**: calculate\_area()

Here, we have created an object named study\_room from the Room class. We then used the object to assign values to attributes: length and breadth.

Notice that we have also used the object to call the method inside the class,

study\_room.calculate\_area()

Here, we have used the . notation to call the method. Finally, the statement inside the method is executed.

## Python Constructors

Earlier we assigned a default value to a class attribute,

class Bike:

name = ""

...

# create object

bike1 = Bike()

However, we can also initialize values using the constructors. For example,

class Bike:

# constructor function

def \_\_init\_\_(self, name = ""):

self.name = name

bike1 = Bike()

Here, \_\_init\_\_() is the constructor function that is called whenever a new object of that class is instantiated.

The constructor above initializes the value of the name attribute. We have used the self.name to refer to the name attribute of the bike1 object.

If we use a constructor to initialize values inside a class, we need to pass the corresponding value during the object creation of the class.

bike1 = Bike("Mountain Bike")

Here, "Mountain Bike" is passed to the name parameter of \_\_init\_\_().

# Python Inheritance

In this tutorial, we will learn about Python inheritance and its types with the help of examples.

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

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

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

## Python Inheritance Syntax

Here's the syntax of the inheritance in Python,

# define a superclass

class super\_class:

# attributes and method definition

# inheritance

class sub\_class(super\_class):

# attributes and method of super\_class

# attributes and method of sub\_class

Here, we are inheriting the sub\_class class from the super\_class class.

## Example 1: Python Inheritance

class Animal:

# attribute and method of the parent class

name = ""

def eat(self):

print("I can eat")

# inherit from Animal

class Dog(Animal):

# new method in subclass

def display(self):

# access name attribute of superclass using self

print("My name is ", self.name)

# create an object of the subclass

labrador = Dog()

# access superclass attribute and method

labrador.name = "Rohu"

labrador.eat()

# call subclass method

labrador.display()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I can eat

My name is Rohu

In the above example, we have derived a subclass Dog from a superclass Animal. Notice the statements,

labrador.name = "Rohu"

labrador.eat()

Here, we are using labrador (object of Dog) to access name and eat() of the Animal class. This is possible because the subclass inherits all attributes and methods of the superclass.

Also, we have accessed the name attribute inside the method of the Dog class using self.

## is-a relationship

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

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

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

### Example 2: Inheritance in Python

Let's take a look at another example of inheritance in Python,

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
* The dispSides() method 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, the Triangle class has a new method findArea() to find and print the area of the triangle.

Now let's see the complete working code of the example above including creating an object,

class Polygon:

# Initializing the number of sides

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)]

# method to display the length of each side of the polygon

def dispSides(self):

for i in range(self.n):

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

class Triangle(Polygon):

# Initializing the number of sides of the triangle to 3 by

# calling the \_\_init\_\_ method of the Polygon class

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

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

def findArea(self):

a, b, c = self.sides

# calculate the semi-perimeter

s = (a + b + c) / 2

# Using Heron's formula to calculate the area of the triangle

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

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

# Creating an instance of the Triangle class

t = Triangle()

# Prompting the user to enter the sides of the triangle

t.inputSides()

# Displaying the sides of the triangle

t.dispSides()

# Calculating and printing the area of the triangle

t.findArea()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Enter side 1 : 3

Enter side 2 : 5

Enter side 3 : 4

Side 1 is 3.0

Side 2 is 5.0

Side 3 is 4.0

The area of the triangle is 6.00

Here, 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 Inheritance

In the previous example, we see the object of the subclass can access the method of the superclass.

**However, what if the same method is present in both the superclass and subclass?**

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

### Example: Method Overriding

class Animal:

# attributes and method of the parent class

name = ""

def eat(self):

print("I can eat")

# inherit from Animal

class Dog(Animal):

# override eat() method

def eat(self):

print("I like to eat bones")

# create an object of the subclass

labrador = Dog()

# call the eat() method on the labrador object

labrador.eat()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I like to eat bones

In the above example, the same method eat() is present in both the Dog class and the Animal class.

Now, when we call the eat() method using the object of the Dog subclass, the method of the Dog class is called.

This is because the eat() method of the Dog subclass overrides the same method of the Animal superclass.

## The super() Method in Python Inheritance

Previously we saw that the same method in the subclass overrides the method in the superclass.

However, if we need to access the superclass method from the subclass, we use the super() method. For example,

class Animal:

name = ""

def eat(self):

print("I can eat")

# inherit from Animal

class Dog(Animal):

# override eat() method

def eat(self):

# call the eat() method of the superclass using super()

super().eat()

print("I like to eat bones")

# create an object of the subclass

labrador = Dog()

labrador.eat()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I can eat

I like to eat bones

In the above example, the eat() method of the Dog subclass overrides the same method of the Animal superclass.

Inside the Dog class, we have used

# call method of superclass

super().eat()

to call the eat() method of the Animal superclass from the Dog subclass.

So, when we call the eat() method using the labrador object

# call the eat() method

labrador.eat()

Both the overridden and the superclass version of the eat() method is executed.

## Uses of Inheritance

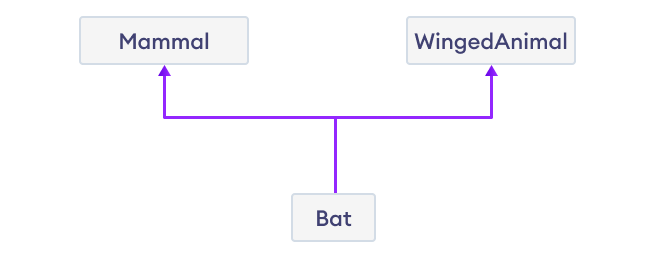
1. Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
2. Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
3. Since you can also add your own functionalities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

# Python Multiple Inheritance

In this tutorial, we'll learn about multiple inheritance in Python with the help of examples.

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

For example, A class Bat is derived from superclasses Mammal and WingedAnimal. It makes sense because bat is a mammal as well as a winged animal.

Multiple Inheritance

## Python Multiple Inheritance Syntax

class SuperClass1:

# features of SuperClass1

class SuperClass2:

# features of SuperClass2

class MultiDerived(SuperClass1, SuperClass2):

# features of SuperClass1 + SuperClass2 + MultiDerived class

Here, the MultiDerived class is derived from SuperClass1 and SuperClass2 classes.

## Example: Python Multiple Inheritance

class Mammal:

def mammal\_info(self):

print("Mammals can give direct birth.")

class WingedAnimal:

def winged\_animal\_info(self):

print("Winged animals can flap.")

class Bat(Mammal, WingedAnimal):

pass

# create an object of Bat class

b1 = Bat()

b1.mammal\_info()

b1.winged\_animal\_info()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Mammals can give direct birth.

Winged animals can flap.

In the above example, the Bat class is derived from two super classes: Mammal and WingedAnimal. Notice the statements,

b1 = Bat()

b1.mammal\_info()

b1.winged\_animal\_info()

Here, we are using b1 (object of Bat) to access mammal\_info() and winged\_animal\_info() methods of the Mammal and the WingedAnimal class respectively.

## Python Multilevel Inheritance

In Python, not only can we derive a class from the superclass but you can also derive a class from the derived class. This form of inheritance is known as **multilevel inheritance**.

Here's the syntax of the multilevel inheritance,

class SuperClass:

# Super class code here

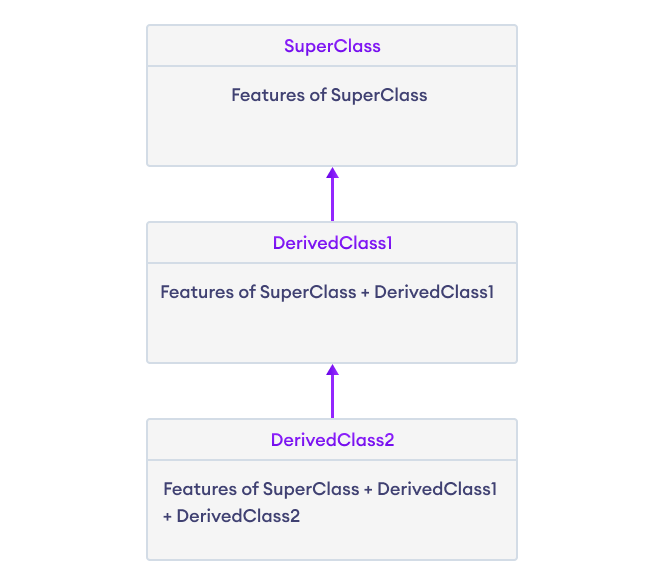
class DerivedClass1(SuperClass):

# Derived class 1 code here

class DerivedClass2(DerivedClass1):

# Derived class 2 code here

Here, the DerivedClass1 class is derived from the SuperClass class, and the DerivedClass2 class is derived from the DerivedClass1 class.

Multilevel Inheritance in Python

## Example: Python Multilevel Inheritance

class SuperClass:

def super\_method(self):

print("Super Class method called")

# define class that derive from SuperClass

class DerivedClass1(SuperClass):

def derived1\_method(self):

print("Derived class 1 method called")

# define class that derive from DerivedClass1

class DerivedClass2(DerivedClass1):

def derived2\_method(self):

print("Derived class 2 method called")

# create an object of DerivedClass2

d2 = DerivedClass2()

d2.super\_method() # Output: "Super Class method called"

d2.derived1\_method() # Output: "Derived class 1 method called"

d2.derived2\_method() # Output: "Derived class 2 method called"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Super Class method called

Derived class 1 method called

Derived class 2 method called

In the above example, DerivedClass2 is derived from DerivedClass1, which is derived from SuperClass.

It means that DerivedClass2 inherits all the attributes and methods of both DerivedClass1 and SuperClass.

Hence, we are using d2 (object of DerivedClass2) to call methods from SuperClass, DerivedClass1, and DerivedClass2.

## Method Resolution Order (MRO) in Python

If two superclasses have the same method name and the derived class calls that method, Python uses the MRO to search for the right method to call. For example,

class SuperClass1:

def info(self):

print("Super Class 1 method called")

class SuperClass2:

def info(self):

print("Super Class 2 method called")

class Derived(SuperClass1, SuperClass2):

pass

d1 = Derived()

d1.info()

# Output: "Super Class 1 method called"

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, SuperClass1 and SuperClass2 both of these classes define a method info().

So when info() is called using the d1 object of the Derived class, Python uses the **MRO** to determine which method to call.

In this case, the **MRO** specifies that methods should be inherited from the leftmost superclass first, so info() of SuperClass1 is called rather than that of SuperClass2.

# Python Operator Overloading

In this tutorial, we will learn about operator overloading in Python with the help of examples.

In Python, we can change the way [operators](https://www.programiz.com/python-programming/operators) work for user-defined 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**.

## Python Special Functions

Class functions that begin with double underscore \_\_ are called special functions in Python.

The special functions are defined by the Python interpreter and used to implement certain features or behaviors.

They are called **"double underscore"** functions because they have a double underscore prefix and suffix, such as \_\_init\_\_() or \_\_add\_\_().

Here are some of the special functions available in Python,

|  |  |
| --- | --- |
| Function | Description |
| \_\_init\_\_() | initialize the attributes of the object |
| \_\_str\_\_() | returns a string representation of the object |
| \_\_len\_\_() | returns the length of the object |
| \_\_add\_\_() | adds two objects |
| \_\_call\_\_() | call objects of the class like a normal function |

## Example: + Operator Overloading in Python

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 the Point object of the coordinate sum.

Let's see an example,

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)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, what actually happens is that, when we 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.

Here's an example of how we can overload the < operator to compare two objects the Person class based on their age:

class Person:

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

self.name = name

self.age = age

# overload < operator

def \_\_lt\_\_(self, other):

return self.age < other.age

p1 = Person("Alice", 20)

p2 = Person("Bob", 30)

print(p1 < p2) # prints True

print(p2 < p1) # prints False

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

True

False

Here, \_\_lt\_\_() overloads the < operator to compare the age attribute of two objects.

The \_\_lt\_\_() method returns,

* True - if the first object's age is less than the second object's age
* False - if the first object's age is greater than the second object's age

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) |

## Advantages of Operator Overloading

Here are some advantages of operator overloading,

* Improves code readability by allowing the use of familiar operators.
* Ensures that objects of a class behave consistently with built-in types and other user-defined types.
* Makes it simpler to write code, especially for complex data types.
* Allows for code reuse by implementing one operator method and using it for other operators.

**Python advanced topics**

Python iterator

Python generator

Python closure

Python decorators

Python property

Python Ex

Python examples

# Python Iterators

In this tutorial, you will learn about the Python Iterators with the help of examples.

Iterators are methods that iterate collections like [lists](https://www.programiz.com/python-programming/list), [tuples](https://www.programiz.com/python-programming/tuple), etc. Using an iterator method, we can loop through an object and return its elements.

Technically, a Python **iterator object** must implement two special methods, \_\_iter\_\_() and \_\_next\_\_(), collectively called the **iterator protocol**.

## Iterating Through an Iterator

In Python, we can use the next() function to return the next item in the sequence.

Let's see an example,

# define a list

my\_list = [4, 7, 0]

# create an iterator from the list

iterator = iter(my\_list)

# get the first element of the iterator

print(next(iterator)) # prints 4

# get the second element of the iterator

print(next(iterator)) # prints 7

# get the third element of the iterator

print(next(iterator)) # prints 0

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

4

7

0

Here, first we created an iterator from the list using the iter() method. And then used the next() function to retrieve the elements of the iterator in sequential order.

When we reach the end and there is no more data to be returned, we will get the StopIteration Exception.

### Using for Loop

A more elegant way of automatically iterating is by using the [for loop](https://www.programiz.com/python-programming/for-loop). For example,

# define a list

my\_list = [4, 7, 0]

for element in my\_list:

print(element)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

4

7

0

## Working of for loop for Iterators

The for loop in Python is used to iterate over a sequence of elements, such as a list, tuple, or string.

When we use the for loop with an iterator, the loop will automatically iterate over the elements of the iterator until it is exhausted.

Here's an example of how a for loop works with an iterator,

# create a list of integers

my\_list = [1, 2, 3, 4, 5]

# create an iterator from the list

iterator = iter(my\_list)

# iterate through the elements of the iterator

for element in iterator:

# Print each element

print(element)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In this example, the for loop iterates over the elements of the iterator object.

On each iteration, the loop assigns the value of the next element to the variable element, and then executes the indented code block.

This process continues until the iterator is exhausted, at which point the for loop terminates.

## Building Custom Iterators

Building an iterator from scratch is easy in Python. We just have to implement the \_\_iter\_\_() and the \_\_next\_\_() methods,

* \_\_iter\_\_() returns the iterator object itself. If required, some initialization can be performed.
* \_\_next\_\_() must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.

Let's see an example that will give us the next power of **2** in each iteration. Power exponent starts from zero up to a user set number,

class PowTwo:

"""Class to implement an iterator

of powers of two"""

def \_\_init\_\_(self, max=0):

self.max = max

def \_\_iter\_\_(self):

self.n = 0

return self

def \_\_next\_\_(self):

if self.n <= self.max:

result = 2 \*\* self.n

self.n += 1

return result

else:

raise StopIteration

# create an object

numbers = PowTwo(3)

# create an iterable from the object

i = iter(numbers)

# Using next to get to the next iterator element

print(next(i)) # prints 1

print(next(i)) # prints 2

print(next(i)) # prints 4

print(next(i)) # prints 8

print(next(i)) # raises StopIteration exception

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

1

2

4

8

Traceback (most recent call last):

File "<string>", line 32, in <module>

File "<string>", line 18, in \_\_next\_\_

StopIteration

We can also use a for loop to iterate over our iterator class.

for i in PowTwo(3):

print(i)

**Output**

1

2

4

8

To learn more about object-oriented programming, visit [Python OOP](https://www.programiz.com/python-programming/object-oriented-programming).

## Python Infinite Iterators

An infinite iterator is an iterator that never ends, meaning that it will continue to produce elements indefinitely.

Here is an example of how to create an infinite iterator in Python using the count() function from the itertools module,

from itertools import count

# create an infinite iterator that starts at 1 and increments by 1 each time

infinite\_iterator = count(1)

# print the first 5 elements of the infinite iterator

for i in range(5):

print(next(infinite\_iterator))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

1

2

3

4

5

Here, we have created an infinite iterator that starts at **1** and increments by **1** each time.

And then we printed the first **5** elements of the infinite iterator using the for loop and the next() method.

# Python Generators

In this tutorial, you'll learn how to create iterations easily using Python generators, how it is different from iterators and normal functions, and why you should use it.

In Python, a generator is a [function](https://www.programiz.com/python-programming/function) that returns an [iterator](https://www.programiz.com/python-programming/iterator) that produces a sequence of values when iterated over.

Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

## Create Python Generator

In Python, similar to defining a [normal function](https://www.programiz.com/python-programming/function), we can define a generator function using the def keyword, but instead of the return statement we use the yield statement.

def generator\_name(arg):

# statements

yield something

Here, the yield keyword is used to produce a value from the generator.

When the generator function is called, it does not execute the function body immediately. Instead, it returns a generator object that can be iterated over to produce the values.

## Example: Python Generator

Here's an example of a generator function that produces a sequence of numbers,

def my\_generator(n):

# initialize counter

value = 0

# loop until counter is less than n

while value < n:

# produce the current value of the counter

yield value

# increment the counter

value += 1

# iterate over the generator object produced by my\_generator

for value in my\_generator(3):

# print each value produced by generator

print(value)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

0

1

2

In the above example, the my\_generator() generator function takes an integer n as an argument and produces a sequence of numbers from **0** to n-1.

The yield keyword is used to produce a value from the generator and pause the generator function's execution until the next value is requested.

The for loop iterates over the generator object produced by my\_generator(), and the print statement prints each value produced by the generator.

We can also create a generator object from the generator function by calling the function like we would any other function as,

generator = my\_range(3)

print(next(generator)) # 0

print(next(generator)) # 1

print(next(generator)) # 2

## Python Generator Expression

In Python, a generator expression is a concise way to create a generator object.

It is similar to a [list comprehension](https://www.programiz.com/python-programming/list-comprehension), but instead of creating a list, it creates a generator object that can be iterated over to produce the values in the generator.

### Generator Expression Syntax

A generator expression has the following syntax,

(expression for item in iterable)

Here, expression is a value that will be returned for each item in the iterable.

The generator expression creates a generator object that produces the values of expression for each item in the iterable, one at a time, when iterated over.

## Example 2: Python Generator Expression

# create the generator object

squares\_generator = (i \* i for i in range(5))

# iterate over the generator and print the values

for i in squares\_generator:

print(i)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

0

1

4

9

16

Here, we have created the generator object that will produce the squares of the numbers **0** through **4** when iterated over.

And then, to iterate over the generator and get the values, we have used the for loop.

## Use of Python Generators

There are several reasons that make generators a powerful implementation.

### 1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of **2** using an iterator class.

class PowTwo:

def \_\_init\_\_(self, max=0):

self.n = 0

self.max = max

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

if self.n > self.max:

raise StopIteration

result = 2 \*\* self.n

self.n += 1

return result

The above program was lengthy and confusing. Now, let's do the same using a generator function.

def PowTwoGen(max=0):

n = 0

while n < max:

yield 2 \*\* n

n += 1

Since generators keep track of details automatically, the implementation was concise and much cleaner.

### 2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

### 3. Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

def all\_even():

n = 0

while True:

yield n

n += 2

### 4. Pipelining Generators

Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

def fibonacci\_numbers(nums):

x, y = 0, 1

for \_ in range(nums):

x, y = y, x+y

yield x

def square(nums):

for num in nums:

yield num\*\*2

print(sum(square(fibonacci\_numbers(10))))

# Output: 4895

[Run Code](https://www.programiz.com/python-programming/online-compiler)

This pipelining is efficient and easy to read (and yes, a lot cooler!).

# Python Closures

In this tutorial, you'll learn about Python closure with the help of examples.

Python closure is a nested function that allows us to access variables of the outer function even after the outer function is closed.

Before we learn about closure, let's first revise the concept of nested functions in Python.

## Nested function in Python

In Python, we can create a function inside another function. This is known as a nested function. For example,

def greet(name):

# inner function

def display\_name():

print("Hi", name)

# call inner function

display\_name()

# call outer function

greet("John")

# Output: Hi John

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have defined the display\_name() function inside the greet() function.

Here, display\_name() is a nested function. The nested function works similar to the normal function. It executes when display\_name() is called inside the function greet().

## Python Closures

As we have already discussed, closure is a nested function that helps us access the outer function's variables even after the outer function is closed. For example,

def greet():

# variable defined outside the inner function

name = "John"

# return a nested anonymous function

return lambda: "Hi " + name

# call the outer function

message = greet()

# call the inner function

print(message())

# Output: Hi John

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have created a function named greet() that returns a nested [anonymous function](https://www.programiz.com/python-programming/anonymous-function).

Here, when we call the outer function,

message = greet()

The returned function is now assigned to the message variable.

At this point, the execution of the outer function is completed, so the name variable should be destroyed. However, when we call the anonymous function using

print(message())

we are able to access the name variable of the outer function.

It's possible because the nested function now acts as a closure that closes the outer scope variable within its scope even after the outer function is executed.

Let's see one more example to make this concept clear.

## Example: Print Odd Numbers using Python Closure

def calculate():

num = 1

def inner\_func():

nonlocal num

num += 2

return num

return inner\_func

# call the outer function

odd = calculate()

# call the inner function

print(odd())

print(odd())

print(odd())

# call the outer function again

odd2 = calculate()

print(odd2())

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

3

5

7

3

In the above example,

odd = calculate()

This code executes the outer function calculate() and returns a closure to the odd number. T

That's why we can access the num variable of calculate() even after completing the outer function.

Again, when we call the outer function using

odd2 = calculate()

a new closure is returned. Hence, we get **3** again when we call odd2().

## When to use closures?

So what are closures good for?

Closures can be used to avoid global values and provide data hiding, and can be an elegant solution for simple cases with one or few methods.

However, for larger cases with multiple attributes and methods, a class implementation may be more appropriate.

def make\_multiplier\_of(n):

def multiplier(x):

return x \* n

return multiplier

# Multiplier of 3

times3 = make\_multiplier\_of(3)

# Multiplier of 5

times5 = make\_multiplier\_of(5)

# Output: 27

print(times3(9))

# Output: 15

print(times5(3))

# Output: 30

print(times5(times3(2)))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

[Python Decorators](https://www.programiz.com/python-programming/decorator) make extensive use of closures as well.

On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out.

All function objects have a \_\_closure\_\_ attribute that returns a tuple of cell objects if it is a closure function.

Referring to the example above, we know times3 and times5 are closure functions.

# Python Decorators

In this tutorial, we will learn about Python Decorators with the help of examples.

In Python, a decorator is a design pattern that allows you to modify the functionality of a function by wrapping it in another function.

The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it.

## Prerequisites for learning decorators

Before we learn about decorators, we need to understand a few important concepts related to Python functions. Also, remember that everything in Python is an object, even functions are objects.

### Nested Function

We can include one function inside another, known as a nested function. For example,

def outer(x):

def inner(y):

return x + y

return inner

add\_five = outer(5)

result = add\_five(6)

print(result) # prints 11

# Output: 11

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we have created the inner() function inside the outer() function.

### Pass Function as Argument

We can pass a function as an argument to another function in Python. For Example,

def add(x, y):

return x + y

def calculate(func, x, y):

return func(x, y)

result = calculate(add, 4, 6)

print(result) # prints 10

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

10

In the above example, the calculate() function takes a function as its argument. While calling calculate(), we are passing the add() function as the argument.

In the calculate() function, arguments: func, x, y become add, 4, and 6 respectively.

And hence, func(x, y) becomes add(4, 6) which returns **10**.

### Return a Function as a Value

In Python, we can also return a function as a return value. For example,

def greeting(name):

def hello():

return "Hello, " + name + "!"

return hello

greet = greeting("Atlantis")

print(greet()) # prints "Hello, Atlantis!"

# Output: Hello, Atlantis!

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, the return hello statement returns the inner hello() function. This function is now assigned to the greet variable.

That's why, when we call greet() as a function, we get the output.

## Python Decorators

As mentioned earlier, A Python decorator is a function that takes in a function and returns it by adding some functionality.

In fact, any object which implements the special \_\_call\_\_() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.

Basically, a decorator takes in a function, adds some functionality and returns it.

def make\_pretty(func):

def inner():

print("I got decorated")

func()

return inner

def ordinary():

print("I am ordinary")

# Output: I am ordinary

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we have created two functions:

* ordinary() that prints "I am ordinary"
* make\_pretty() that takes a function as its argument and has a nested function named inner(), and returns the inner function.

We are calling the ordinary() function normally, so we get the output "I am ordinary". Now, let's call it using the decorator function.

def make\_pretty(func):

# define the inner function

def inner():

# add some additional behavior to decorated function

print("I got decorated")

# call original function

func()

# return the inner function

return inner

# define ordinary function

def ordinary():

print("I am ordinary")

# decorate the ordinary function

decorated\_func = make\_pretty(ordinary)

# call the decorated function

decorated\_func()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I got decorated

I am ordinary

In the example shown above, make\_pretty() is a decorator. Notice the code,

decorated\_func = make\_pretty(ordinary)

* We are now passing the ordinary() function as the argument to the make\_pretty().
* The make\_pretty() function returns the inner function, and it is now assigned to the decorated\_func variable.

decorated\_func()

Here, we are actually calling the inner() function, where we are printing

### @ Symbol With Decorator

Instead of assigning the function call to a variable, Python provides a much more elegant way to achieve this functionality using the @ symbol. For example,

def make\_pretty(func):

def inner():

print("I got decorated")

func()

return inner

@make\_pretty

def ordinary():

print("I am ordinary")

ordinary()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I got decorated

I am ordinary

Here, the ordinary() function is decorated with the make\_pretty() decorator using the @make\_pretty syntax, which is equivalent to calling ordinary = make\_pretty(ordinary).

## Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like:

def divide(a, b):

return a/b

This function has two parameters, a and b. We know it will give an error if we pass in b as **0**.

Now let's make a decorator to check for this case that will cause the error.

def smart\_divide(func):

def inner(a, b):

print("I am going to divide", a, "and", b)

if b == 0:

print("Whoops! cannot divide")

return

return func(a, b)

return inner

@smart\_divide

def divide(a, b):

print(a/b)

divide(2,5)

divide(2,0)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

I am going to divide 2 and 5

0.4

I am going to divide 2 and 0

Whoops! cannot divide

Here, when we call the divide() function with the arguments **(2,5)**, the inner() function defined in the smart\_divide() decorator is called instead.

This inner() function calls the original divide() function with the arguments **2** and **5** and returns the result, which is **0.4**.

Similarly, When we call the divide() function with the arguments (**2,0)**, the inner() function checks that b is equal to **0** and prints an error message before returning None.

## Chaining Decorators in Python

Multiple decorators can be chained in Python.

To chain decorators in Python, we can apply multiple decorators to a single function by placing them one after the other, with the most inner decorator being applied first.

def star(func):

def inner(\*args, \*\*kwargs):

print("\*" \* 15)

func(\*args, \*\*kwargs)

print("\*" \* 15)

return inner

def percent(func):

def inner(\*args, \*\*kwargs):

print("%" \* 15)

func(\*args, \*\*kwargs)

print("%" \* 15)

return inner

@star

@percent

def printer(msg):

print(msg)

printer("Hello")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

%%%%%%%%%%%%%%%

Hello

%%%%%%%%%%%%%%%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

The above syntax of,

@star

@percent

def printer(msg):

print(msg)

is equivalent to

def printer(msg):

print(msg)

printer = star(percent(printer))

The order in which we chain decorators matter. If we had reversed the order as,

@percent

@star

def printer(msg):

print(msg)

The output would be:

%%%%%%%%%%%%%%%

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Hello

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

%%%%%%%%%%%%%%%

# Python @property decorator

In this tutorial, you will learn about Python @property decorator; a pythonic way to use getters and setters in object-oriented programming.

Python programming provides us with a built-in @property decorator which makes usage of getter and setters much easier in Object-Oriented Programming.

Before going into details on what @property decorator is, let us first build an intuition on why it would be needed in the first place.

## Class Without Getters and Setters

Let us assume that we decide to make a [class](https://www.programiz.com/python-programming/class) that stores the temperature in degrees Celsius. And, it would also implement a method to convert the temperature into degrees Fahrenheit.

One way of doing this is as follows:

class Celsius:

def \_\_init\_\_(self, temperature = 0):

self.temperature = temperature

def to\_fahrenheit(self):

return (self.temperature \* 1.8) + 32

We can make objects out of this class and manipulate the temperature attribute as we wish:

# Basic method of setting and getting attributes in Python

class Celsius:

def \_\_init\_\_(self, temperature=0):

self.temperature = temperature

def to\_fahrenheit(self):

return (self.temperature \* 1.8) + 32

# Create a new object

human = Celsius()

# Set the temperature

human.temperature = 37

# Get the temperature attribute

print(human.temperature)

# Get the to\_fahrenheit method

print(human.to\_fahrenheit())

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

37

98.60000000000001

Here, the extra decimal places when converting into Fahrenheit is due to the [Floating Point Arithmetic Error](https://www.programiz.com/python-programming/numbers#dec).

So, whenever we assign or retrieve any object attribute like temperature as shown above, Python searches it in the object's built-in \_\_dict\_\_ dictionary attribute as

print(human.\_\_dict\_\_)

# Output: {'temperature': 37}

Therefore, human.temperature internally becomes human.\_\_dict\_\_['temperature'].

## Using Getters and Setters

Suppose we want to extend the usability of the Celsius class defined above. We know that the temperature of any object cannot reach below **-273.15** degrees Celsius.

Let's update our code to implement this value constraint.

An obvious solution to the above restriction will be to hide the attribute temperature (make it private) and define new getter and setter methods to manipulate it.

This can be done as follows:

# Making Getters and Setter methods

class Celsius:

def \_\_init\_\_(self, temperature=0):

self.set\_temperature(temperature)

def to\_fahrenheit(self):

return (self.get\_temperature() \* 1.8) + 32

# getter method

def get\_temperature(self):

return self.\_temperature

# setter method

def set\_temperature(self, value):

if value < -273.15:

raise ValueError("Temperature below -273.15 is not possible.")

self.\_temperature = value

As we can see, the above method introduces two new get\_temperature() and set\_temperature() methods.

Furthermore, temperature was replaced with \_temperature. An underscore \_ at the beginning is used to denote private variables in Python.

Now, let's use this implementation:

# Making Getters and Setter methods

class Celsius:

def \_\_init\_\_(self, temperature=0):

self.set\_temperature(temperature)

def to\_fahrenheit(self):

return (self.get\_temperature() \* 1.8) + 32

# getter method

def get\_temperature(self):

return self.\_temperature

# setter method

def set\_temperature(self, value):

if value < -273.15:

raise ValueError("Temperature below -273.15 is not possible.")

self.\_temperature = value

# Create a new object, set\_temperature() internally called by \_\_init\_\_

human = Celsius(37)

# Get the temperature attribute via a getter

print(human.get\_temperature())

# Get the to\_fahrenheit method, get\_temperature() called by the method itself

print(human.to\_fahrenheit())

# new constraint implementation

human.set\_temperature(-300)

# Get the to\_fahreheit method

print(human.to\_fahrenheit())

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

37

98.60000000000001

Traceback (most recent call last):

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

File "<string>", line 16, in set\_temperature

ValueError: Temperature below -273.15 is not possible.

This update successfully implemented the new restriction. We are no longer allowed to set the temperature below **-273.15** degrees Celsius.

**Note**: The private variables don't actually exist in Python. There are simply norms to be followed. The language itself doesn't apply any restrictions.

However, the bigger problem with the above update is that all the programs that implemented our previous class have to modify their code from obj.temperature to obj.get\_temperature() and all expressions like obj.temperature = val to obj.set\_temperature(val).

This refactoring can cause problems while dealing with hundreds of thousands of lines of codes.

All in all, our new update was not backwards compatible. This is where @property comes to rescue.

## The property Class

A pythonic way to deal with the above problem is to use the property class. Here is how we can update our code:

# using property class

class Celsius:

def \_\_init\_\_(self, temperature=0):

self.temperature = temperature

def to\_fahrenheit(self):

return (self.temperature \* 1.8) + 32

# getter

def get\_temperature(self):

print("Getting value...")

return self.\_temperature

# setter

def set\_temperature(self, value):

print("Setting value...")

if value < -273.15:

raise ValueError("Temperature below -273.15 is not possible")

self.\_temperature = value

# creating a property object

temperature = property(get\_temperature, set\_temperature)

We added the print() function inside get\_temperature() and set\_temperature() to clearly observe that they are being executed.

The last line of the code makes a property object temperature. Simply put, property attaches some code (get\_temperature and set\_temperature) to the member attribute accesses (temperature).

Let's use this update code:

# using property class

class Celsius:

def \_\_init\_\_(self, temperature=0):

self.temperature = temperature

def to\_fahrenheit(self):

return (self.temperature \* 1.8) + 32

# getter

def get\_temperature(self):

print("Getting value...")

return self.\_temperature

# setter

def set\_temperature(self, value):

print("Setting value...")

if value < -273.15:

raise ValueError("Temperature below -273.15 is not possible")

self.\_temperature = value

# creating a property object

temperature = property(get\_temperature, set\_temperature)

human = Celsius(37)

print(human.temperature)

print(human.to\_fahrenheit())

human.temperature = -300

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Setting value...

Getting value...

37

Getting value...

98.60000000000001

Setting value...

Traceback (most recent call last):

File "<string>", line 31, in <module>

File "<string>", line 18, in set\_temperature

ValueError: Temperature below -273 is not possible

As we can see, any code that retrieves the value of temperature will automatically call get\_temperature() instead of a dictionary (\_\_dict\_\_) look-up.

Similarly, any code that assigns a value to temperature will automatically call set\_temperature().

We can even see above that set\_temperature() was called even when we created an object.

human = Celsius(37) # prints Setting value...

**Can you guess why?**

The reason is that when an object is created, the \_\_init\_\_() method gets called. This method has the line self.temperature = temperature. This expression automatically calls set\_temperature().

Similarly, any access like c.temperature automatically calls get\_temperature(). This is what property does.

By using property, we can see that no modification is required in the implementation of the value constraint. Thus, our implementation is backward compatible.

**Note**: The actual temperature value is stored in the private \_temperature variable. The temperature attribute is a property object which provides an interface to this private variable.

## The @property Decorator

In Python, property() is a built-in function that creates and returns a property object. The syntax of this function is:

property(fget=None, fset=None, fdel=None, doc=None)

Here,

* fget is function to get value of the attribute
* fset is function to set value of the attribute
* fdel is function to delete the attribute
* doc is a string (like a comment)

As seen from the implementation, these function arguments are optional.

A property object has three methods, getter(), setter(), and deleter() to specify fget, fset and fdel at a later point. This means, the line:

temperature = property(get\_temperature,set\_temperature)

can be broken down as:

# make empty property

temperature = property()

# assign fget

temperature = temperature.getter(get\_temperature)

# assign fset

temperature = temperature.setter(set\_temperature)

These two pieces of code are equivalent.

Programmers familiar with [Python Decorators](https://www.programiz.com/python-programming/decorator) can recognize that the above construct can be implemented as decorators.

We can even not define the names get\_temperature and set\_temperature as they are unnecessary and pollute the class namespace.

For this, we reuse the temperature name while defining our getter and setter functions. Let's look at how to implement this as a decorator:

# Using @property decorator

class Celsius:

def \_\_init\_\_(self, temperature=0):

self.temperature = temperature

def to\_fahrenheit(self):

return (self.temperature \* 1.8) + 32

@property

def temperature(self):

print("Getting value...")

return self.\_temperature

@temperature.setter

def temperature(self, value):

print("Setting value...")

if value < -273.15:

raise ValueError("Temperature below -273 is not possible")

self.\_temperature = value

# create an object

human = Celsius(37)

print(human.temperature)

print(human.to\_fahrenheit())

coldest\_thing = Celsius(-300)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Setting value...

Getting value...

37

Getting value...

98.60000000000001

Setting value...

Traceback (most recent call last):

File "", line 29, in

File "", line 4, in \_\_init\_\_

File "", line 18, in temperature

ValueError: Temperature below -273 is not possible

The above implementation is simple and efficient. It is the recommended way to use property.

# Python RegEx

In this tutorial, you will learn about regular expressions (RegEx), and use Python's re module to work with RegEx (with the help of examples).

A **Reg**ular **Ex**pression (RegEx) is a sequence of characters that defines a search pattern. For example,

^a...s$

The above code defines a RegEx pattern. The pattern is: **any five letter string starting with a and ending with s**.

A pattern defined using RegEx can be used to match against a string.

| Expression | String | Matched? |
| --- | --- | --- |
| ^a...s$ | abs | No match |
| alias | Match |
| abyss | Match |
| Alias | No match |
| An abacus | No match |

Python has a module named re to work with RegEx. Here's an example:

import re

pattern = '^a...s$'

test\_string = 'abyss'

result = re.match(pattern, test\_string)

if result:

print("Search successful.")

else:

print("Search unsuccessful.")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we used re.match() function to search pattern within the test\_string. The method returns a match object if the search is successful. If not, it returns None.

There are other several functions defined in the re module to work with RegEx. Before we explore that, let's learn about regular expressions themselves.

If you already know the basics of RegEx, jump to [Python RegEx](https://www.programiz.com/python-programming/regex#python-regex).

## Specify Pattern Using RegEx

To specify regular expressions, metacharacters are used. In the above example, ^ and $ are metacharacters.

### MetaCharacters

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

**[]** **.** **^** **$** **\*** **+** **?** **{}** **()** **\** **|**

**[] - Square brackets**

Square brackets specifies a set of characters you wish to match.

| Expression | String | Matched? |
| --- | --- | --- |
| [abc] | a | 1 match |
| ac | 2 matches |
| Hey Jude | No match |
| abc de ca | 5 matches |

Here, [abc] will match if the string you are trying to match contains any of the a, b or c.

You can also specify a range of characters using - inside square brackets.

* [a-e] is the same as [abcde].
* [1-4] is the same as [1234].
* [0-39] is the same as [01239].

You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.

* [^abc] means any character except a or b or c.
* [^0-9] means any non-digit character.

. - **Period**

A period matches any single character (except newline '\n').

| Expression | String | Matched? |
| --- | --- | --- |
| .. | a | No match |
| ac | 1 match |
| acd | 1 match |
| acde | 2 matches (contains 4 characters) |

^ - **Caret**

The caret symbol ^ is used to check if a string **starts with** a certain character.

| Expression | String | Matched? |
| --- | --- | --- |
| ^a | a | 1 match |
| abc | 1 match |
| bac | No match |
| ^ab | abc | 1 match |
| acb | No match (starts with a but not followed by b) |

$ - **Dollar**

The dollar symbol $ is used to check if a string **ends with** a certain character.

| Expression | String | Matched? |
| --- | --- | --- |
| a$ | a | 1 match |
| formula | 1 match |
| cab | No match |

\* - **Star**

The star symbol \* matches **zero or more occurrences** of the pattern left to it.

| Expression | String | Matched? |
| --- | --- | --- |
| ma\*n | mn | 1 match |
| man | 1 match |
| maaan | 1 match |
| main | No match (a is not followed by n) |
| woman | 1 match |

+ - **Plus**

The plus symbol + matches **one or more occurrences** of the pattern left to it.

| Expression | String | Matched? |
| --- | --- | --- |
| ma+n | mn | No match (no a character) |
| man | 1 match |
| maaan | 1 match |
| main | No match (a is not followed by n) |
| woman | 1 match |

? - **Question Mark**

The question mark symbol ? matches **zero or one occurrence** of the pattern left to it.

| Expression | String | Matched? |
| --- | --- | --- |
| ma?n | mn | 1 match |
| man | 1 match |
| maaan | No match (more than one a character) |
| main | No match (a is not followed by n) |
| woman | 1 match |

{} - **Braces**

Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.

| Expression | String | Matched? |
| --- | --- | --- |
| a{2,3} | abc dat | No match |
| abc daat | 1 match (at daat) |
| aabc daaat | 2 matches (at aabc and daaat) |
| aabc daaaat | 2 matches (at aabc and daaaat) |

Let's try one more example. This RegEx [0-9]{2, 4} matches at least 2 digits but not more than 4 digits

| Expression | String | Matched? |
| --- | --- | --- |
| [0-9]{2,4} | ab123csde | 1 match (match at ab123csde) |
| 12 and 345673 | 3 matches (12, 3456, 73) |
| 1 and 2 | No match |

| - **Alternation**

Vertical bar | is used for alternation (or operator).

| Expression | String | Matched? |
| --- | --- | --- |
| a|b | cde | No match |
| ade | 1 match (match at ade) |
| acdbea | 3 matches (at acdbea) |

Here, a|b match any string that contains either a or b

() - **Group**

Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz

| Expression | String | Matched? |
| --- | --- | --- |
| (a|b|c)xz | ab xz | No match |
| abxz | 1 match (match at abxz) |
| axz cabxz | 2 matches (at axzbc cabxz) |

\ - **Backslash**

Backlash \ is used to escape various characters including all metacharacters. For example,

\$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.

If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.

**Special Sequences**

Special sequences make commonly used patterns easier to write. Here's a list of special sequences:

\A - Matches if the specified characters are at the start of a string.

| Expression | String | Matched? |
| --- | --- | --- |
| \Athe | the sun | Match |
| In the sun | No match |

\b - Matches if the specified characters are at the beginning or end of a word.

| Expression | String | Matched? |
| --- | --- | --- |
| \bfoo | football | Match |
| a football | Match |
| afootball | No match |
| foo\b | the foo | Match |
| the afoo test | Match |
| the afootest | No match |

\B - Opposite of \b. Matches if the specified characters are **not** at the beginning or end of a word.

| Expression | String | Matched? |
| --- | --- | --- |
| \Bfoo | football | No match |
| a football | No match |
| afootball | Match |
| foo\B | the foo | No match |
| the afoo test | No match |
| the afootest | Match |

\d - Matches any decimal digit. Equivalent to [0-9]

| Expression | String | Matched? |
| --- | --- | --- |
| \d | 12abc3 | 3 matches (at 12abc3) |
| Python | No match |

\D - Matches any non-decimal digit. Equivalent to [^0-9]

| Expression | String | Matched? |
| --- | --- | --- |
| \D | 1ab34"50 | 3 matches (at 1ab34"50) |
| 1345 | No match |

\s - Matches where a string contains any whitespace character. Equivalent to [ \t\n\r\f\v].

| Expression | String | Matched? |
| --- | --- | --- |
| \s | Python RegEx | 1 match |
| PythonRegEx | No match |

\S - Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].

| Expression | String | Matched? |
| --- | --- | --- |
| \S | a b | 2 matches (at a b) |
|  | No match |

\w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9\_]. By the way, underscore \_ is also considered an alphanumeric character.

| Expression | String | Matched? |
| --- | --- | --- |
| \w | 12&": ;c | 3 matches (at 12&": ;c) |
| %"> ! | No match |

\W - Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9\_]

| Expression | String | Matched? |
| --- | --- | --- |
| \W | 1a2%c | 1 match (at 1a2%c) |
| Python | No match |

\Z - Matches if the specified characters are at the end of a string.

| Expression | String | Matched? |
| --- | --- | --- |
| Python\Z | I like Python | 1 match |
| I like Python Programming | No match |
| Python is fun. | No match |

**Tip:** To build and test regular expressions, you can use RegEx tester tools such as [regex101](https://regex101.com/). This tool not only helps you in creating regular expressions, but it also helps you learn it.

Now you understand the basics of RegEx, let's discuss how to use RegEx in your Python code.

## Python RegEx

Python has a module named re to work with regular expressions. To use it, we need to import the module.

import re

The module defines several functions and constants to work with RegEx.

## re.findall()

The re.findall() method returns a list of strings containing all matches.

### Example 1: re.findall()

# Program to extract numbers from a string

import re

string = 'hello 12 hi 89. Howdy 34'

pattern = '\d+'

result = re.findall(pattern, string)

print(result)

# Output: ['12', '89', '34']

If the pattern is not found, re.findall() returns an empty list.

## re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

### Example 2: re.split()

import re

string = 'Twelve:12 Eighty nine:89.'

pattern = '\d+'

result = re.split(pattern, string)

print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']

[Run Code](https://www.programiz.com/python-programming/online-compiler)

If the pattern is not found, re.split() returns a list containing the original string.

You can pass maxsplit argument to the re.split() method. It's the maximum number of splits that will occur.

import re

string = 'Twelve:12 Eighty nine:89 Nine:9.'

pattern = '\d+'

# maxsplit = 1

# split only at the first occurrence

result = re.split(pattern, string, 1)

print(result)

# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']

[Run Code](https://www.programiz.com/python-programming/online-compiler)

By the way, the default value of maxsplit is 0; meaning all possible splits.

## re.sub()

The syntax of re.sub() is:

re.sub(pattern, replace, string)

The method returns a string where matched occurrences are replaced with the content of replace variable.

### Example 3: re.sub()

# Program to remove all whitespaces

import re

# multiline string

string = 'abc 12\

de 23 \n f45 6'

# matches all whitespace characters

pattern = '\s+'

# empty string

replace = ''

new\_string = re.sub(pattern, replace, string)

print(new\_string)

# Output: abc12de23f456

[Run Code](https://www.programiz.com/python-programming/online-compiler)

If the pattern is not found, re.sub() returns the original string.

You can pass count as a fourth parameter to the re.sub() method. If omited, it results to 0. This will replace all occurrences.

import re

# multiline string

string = 'abc 12\

de 23 \n f45 6'

# matches all whitespace characters

pattern = '\s+'

replace = ''

new\_string = re.sub(r'\s+', replace, string, 1)

print(new\_string)

# Output:

# abc12de 23

# f45 6

## re.subn()

The re.subn() is similar to re.sub() except it returns a tuple of 2 items containing the new string and the number of substitutions made.

### Example 4: re.subn()

# Program to remove all whitespaces

import re

# multiline string

string = 'abc 12\

de 23 \n f45 6'

# matches all whitespace characters

pattern = '\s+'

# empty string

replace = ''

new\_string = re.subn(pattern, replace, string)

print(new\_string)

# Output: ('abc12de23f456', 4)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

## re.search()

The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, re.search() returns a match object; if not, it returns None.

match = re.search(pattern, str)

### Example 5: re.search()

import re

string = "Python is fun"

# check if 'Python' is at the beginning

match = re.search('\APython', string)

if match:

print("pattern found inside the string")

else:

print("pattern not found")

# Output: pattern found inside the string

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, match contains a match object.

## Match object

You can get methods and attributes of a match object using [dir()](https://www.programiz.com/python-programming/methods/built-in/dir) function.

Some of the commonly used methods and attributes of match objects are:

### match.group()

The group() method returns the part of the string where there is a match.

### Example 6: Match object

import re

string = '39801 356, 2102 1111'

# Three digit number followed by space followed by two digit number

pattern = '(\d{3}) (\d{2})'

# match variable contains a Match object.

match = re.search(pattern, string)

if match:

print(match.group())

else:

print("pattern not found")

# Output: 801 35

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, match variable contains a match object.

Our pattern (\d{3}) (\d{2}) has two subgroups (\d{3}) and (\d{2}). You can get the part of the string of these parenthesized subgroups. Here's how:

>>> match.group(1)

'801'

>>> match.group(2)

'35'

>>> match.group(1, 2)

('801', '35')

>>> match.groups()

('801', '35')

### match.start(), match.end() and match.span()

The start() function returns the index of the start of the matched substring. Similarly, end() returns the end index of the matched substring.

>>> match.start()

2

>>> match.end()

8

The span() function returns a tuple containing start and end index of the matched part.

>>> match.span()

(2, 8)

### match.re and match.string

The re attribute of a matched object returns a regular expression object. Similarly, string attribute returns the passed string.

>>> match.re

re.compile('(\\d{3}) (\\d{2})')

>>> match.string

'39801 356, 2102 1111'

We have covered all commonly used methods defined in the re module. If you want to learn more, visit [Python 3 re module](https://docs.python.org/3/library/re.html).

### Using r prefix before RegEx

When r or R prefix is used before a regular expression, it means raw string. For example, '\n' is a new line whereas r'\n' means two characters: a backslash \ followed by n.

Backlash \ is used to escape various characters including all metacharacters. However, using r prefix makes \ treat as a normal character.

### Example 7: Raw string using r prefix

import re

string = '\n and \r are escape sequences.'

result = re.findall(r'[\n\r]', string)

print(result)

# Output: ['\n', '\r']

**Python examples**

# Python Program to Print Hello world!

A simple program that displays “Hello, World!”. It's often used to illustrate the syntax of the language.

To understand this example, you should have the knowledge of the following [Python programming](https://www.programiz.com/python-programming) topics:

* [How to Get Started With Python?](https://www.programiz.com/python-programming/first-program)
* [Python Basic Input and Output](https://www.programiz.com/python-programming/input-output-import)

## Source Code

# This program prints Hello, world!

print('Hello, world!')

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Hello, world!

In this program, we have used the built-in print() function to print the string Hello, world! on our screen.

By the way, a [string](https://www.programiz.com/python-programming/string) is a sequence of characters. In Python, strings are enclosed inside single quotes, double quotes, or triple quotes.

**Python date and time**

Python date and time module

Python datetime.strftime()

Python datetime.strptime()

Current date and time

Get current time

Timestampt to datetime

Python time module

Python time.sleep()

Python has a module named **datetime** to work with dates and times.

It provides a variety of classes for representing and manipulating dates and times, as well as for formatting and parsing dates and times in a variety of formats.

### Example 1: Get Current Date and Time

import datetime

# get the current date and time

now = datetime.datetime.now()

print(now)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

2022-12-27 08:26:49.219717

Here, we have imported the datetime module using the import datetime statement.

One of the classes defined in the datetime module is the datetime class.

We then used the now() method to create a datetime object containing the current local date and time.

### Example 2: Get Current Date

import datetime

# get current date

current\_date = datetime.date.today()

print(current\_date)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

2022-12-27

In the above example, we have used the today() method defined in the date class to get a date object containing the current local date.

## Attributes of datetime Module

We can use the [dir()](https://www.programiz.com/python-programming/methods/built-in/dir) function to get a list containing all attributes of a module.

import datetime

print(dir(datetime))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

['MAXYEAR', 'MINYEAR', '\_\_builtins\_\_', '\_\_cached\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', '\_divide\_and\_round', 'date', 'datetime', 'datetime\_CAPI', 'time', 'timedelta', 'timezone', 'tzinfo']

Among all the attributes of datetime module, the most commonly used classes in the datetime module are:

* datetime.datetime - represents a single point in time, including a date and a time.
* datetime.date - represents a date (year, month, and day) without a time.
* datetime.time - represents a time (hour, minute, second, and microsecond) without a date.
* datetime.timedelta - represents a duration, which can be used to perform arithmetic with datetime objects.

## Python datetime.date Class

In Python, we can instantiate date objects from the date class. A date object represents a date (year, month and day).

### Example 3: Date object to represent a date

import datetime

d = datetime.date(2022, 12, 25)

print(d)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

2022-12-25

Here, date() in the above example is a constructor of the date class. The constructor takes three arguments: year, month and day.

### Import Only date Class

We can only import the date class from the datetime module. For example,

from datetime import date

d = date(2022, 12, 25)

print(d)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

2022-12-25

Here, from datetime import date only imports the date class from the datetime module.

### Example 4: Get current date using today()

We can create a date object containing the current date by using the class method named today(). For example,

from datetime import date

# today() to get current date

todays\_date = date.today()

print("Today's date =", todays\_date)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Today's date = 2022-12-27

### Example 5: Get date from a timestamp

We can also create date objects from a timestamp.

A Unix timestamp is the number of seconds between a particular date and **January 1, 1970** at UTC. You can convert a timestamp to date using the fromtimestamp() method.

from datetime import date

timestamp = date.fromtimestamp(1326244364)

print("Date =", timestamp)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Date = 2012-01-11

### Example 6: Print today's year, month and day

We can get year, month, day, day of the week etc. from the date object easily. For example,

from datetime import date

# date object of today's date

today = date.today()

print("Current year:", today.year)

print("Current month:", today.month)

print("Current day:", today.day)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Current year: 2022

Current month: 12

Current day: 27

## Python datetime.time Class

A time object instantiated from the time class represents the local time.

### Example 7: Time object to represent time

from datetime import time

# time(hour = 0, minute = 0, second = 0)

a = time()

print(a)

# time(hour, minute and second)

b = time(11, 34, 56)

print(b)

# time(hour, minute and second)

c = time(hour = 11, minute = 34, second = 56)

print(c)

# time(hour, minute, second, microsecond)

d = time(11, 34, 56, 234566)

print(d)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

a = 00:00:00

b = 11:34:56

c = 11:34:56

d = 11:34:56.234566

### Example 8: Print hour, minute, second and microsecond

Once we create the time object, we can easily print its attributes such as hour, minute etc. For example,

from datetime import time

a = time(11, 34, 56)

print("Hour =", a.hour)

print("Minute =", a.minute)

print("Second =", a.second)

print("Microsecond =", a.microsecond)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Hour = 11

Minute = 34

Second = 56

Microsecond = 0

Here, notice that we haven't passed the microsecond argument. Hence, its default value 0 is printed.

## The datetime.datetime Class

The datetime module has a class named datetime that can contain information from both **date** and **time** objects.

### Example 9: Python datetime object

from datetime import datetime

# datetime(year, month, day)

a = datetime(2022, 12, 28)

print(a)

# datetime(year, month, day, hour, minute, second, microsecond)

b = datetime(2022, 12, 28, 23, 55, 59, 342380)

print(b)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

2022-12-28 00:00:00

2022-12-28 23:55:59.342380

The first three arguments year, month and day in the datetime() constructor are mandatory.

### Example 10: Print year, month, hour, minute and timestamp

from datetime import datetime

a = datetime(2022, 12, 28, 23, 55, 59, 342380)

print("Year =", a.year)

print("Month =", a.month)

print("Hour =", a.hour)

print("Minute =", a.minute)

print("Timestamp =", a.timestamp())

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

year = 202

month = 12

day = 28

hour = 23

minute = 55

timestamp = 1511913359.34238

## Python datetime.timedelta Class

A timedelta object represents the difference between two dates or times. For example,

from datetime import datetime, date

# using date()

t1 = date(year = 2018, month = 7, day = 12)

t2 = date(year = 2017, month = 12, day = 23)

t3 = t1 - t2

print("t3 =", t3)

# using datetime()

t4 = datetime(year = 2018, month = 7, day = 12, hour = 7, minute = 9, second = 33)

t5 = datetime(year = 2019, month = 6, day = 10, hour = 5, minute = 55, second = 13)

t6 = t4 - t5

print("t6 =", t6)

print("Type of t3 =", type(t3))

print("Type of t6 =", type(t6))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

t3 = 201 days, 0:00:00

t6 = -333 days, 1:14:20

Type of t3 = <class 'datetime.timedelta'>

Type of t6 = <class 'datetime.timedelta'>

Notice, both t3 and t6 are of <class 'datetime.timedelta'> type.

### Example 12: Difference between two timedelta objects

from datetime import timedelta

t1 = timedelta(weeks = 2, days = 5, hours = 1, seconds = 33)

t2 = timedelta(days = 4, hours = 11, minutes = 4, seconds = 54)

t3 = t1 - t2

print("t3 =", t3)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

t3 = 14 days, 13:55:39

Here, we have created two timedelta objects t1 and t2, and their difference is printed on the screen.

### Example 14: Time duration in seconds

We can get the total number of seconds in a timedelta object using the total\_seconds() method.

from datetime import timedelta

t = timedelta(days = 5, hours = 1, seconds = 33, microseconds = 233423)

print("Total seconds =", t.total\_seconds())

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Total seconds = 435633.233423

## Python format datetime

The way date and time is represented may be different in different places, organizations etc. It's more common to use mm/dd/yyyy in the US, whereas dd/mm/yyyy is more common in the UK.

Python has strftime() and strptime() methods to handle this.

### Python strftime() Method

The strftime() method is defined under classes date, datetime and time. The method creates a formatted string from a given date, datetime or time object.

Let's see an example.

from datetime import datetime

# current date and time

now = datetime.now()

t = now.strftime("%H:%M:%S")

print("Time:", t)

s1 = now.strftime("%m/%d/%Y, %H:%M:%S")

# mm/dd/YY H:M:S format

print("s1:", s1)

s2 = now.strftime("%d/%m/%Y, %H:%M:%S")

# dd/mm/YY H:M:S format

print("s2:", s2)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

time: 04:34:52

s1: 12/26/2018, 04:34:52

s2: 26/12/2018, 04:34:52

Here, %Y, %m, %d, %H etc. are format codes. The strftime() method takes one or more format codes and returns a formatted string based on it.

In the above example, t, s1 and s2 are strings.

* %Y - year [0001,..., 2018, 2019,..., 9999]
* %m - month [01, 02, ..., 11, 12]
* %d - day [01, 02, ..., 30, 31]
* %H - hour [00, 01, ..., 22, 23
* %M - minute [00, 01, ..., 58, 59]
* %S - second [00, 01, ..., 58, 59]

To learn more about strftime() and format codes, visit: [Python strftime()](https://www.programiz.com/python-programming/datetime/strftime).

### Python strptime() Method

The strptime() method creates a datetime object from a given string (representing date and time). For example,

from datetime import datetime

date\_string = "25 December, 2022"

print("date\_string =", date\_string)

# use strptime() to create date object

date\_object = datetime.strptime(date\_string, "%d %B, %Y")

print("date\_object =", date\_object)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

date\_string = 25 december, 2022

date\_object = 2018-06-21 00:00:00

The strptime() method takes two arguments:

* a string representing date and time
* format code equivalent to the first argument

By the way, %d, %B and %Y format codes are used for day, month(full name) and year respectively.

Visit [Python strptime()](https://www.programiz.com/python-programming/datetime/strptime) to learn more.

## Handling timezone in Python

Suppose, we are working on a project and need to display date and time based on their timezone.

Rather than trying to handle timezone yourself, we suggest to use a third-party [pytZ module](http://pytz.sourceforge.net/).

from datetime import datetime

import pytz

local = datetime.now()

print("Local:", local.strftime("%m/%d/%Y, %H:%M:%S"))

tz\_NY = pytz.timezone('America/New\_York')

datetime\_NY = datetime.now(tz\_NY)

print("NY:", datetime\_NY.strftime("%m/%d/%Y, %H:%M:%S"))

tz\_London = pytz.timezone('Europe/London')

datetime\_London = datetime.now(tz\_London)

print("London:", datetime\_London.strftime("%m/%d/%Y, %H:%M:%S"))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Local: 12/27/2022, 09:40:19

NY: 12/27/2022, 04:40:19

London: 12/27/2022, 09:40:19

Here, datetime\_NY and datetime\_London are datetime objects containing the current date and time of their respective timezone.

# Python strftime()

In this tutorial, you will learn to convert date, time and datetime objects to its equivalent string (with the help of examples)

The strftime() method returns a string representing date and time using [date](https://www.programiz.com/python-programming/datetime#date), [time](https://www.programiz.com/python-programming/datetime#time) or [datetime](https://www.programiz.com/python-programming/datetime#datetime) object.

## Example 1: datetime to string using strftime()

The program below converts a datetime object containing current date and time to different string formats.

from datetime import datetime

now = datetime.now() # current date and time

year = now.strftime("%Y")

print("year:", year)

month = now.strftime("%m")

print("month:", month)

day = now.strftime("%d")

print("day:", day)

time = now.strftime("%H:%M:%S")

print("time:", time)

date\_time = now.strftime("%m/%d/%Y, %H:%M:%S")

print("date and time:",date\_time)

When you run the program, the output will something like be:

year: 2018

month: 12

day: 24

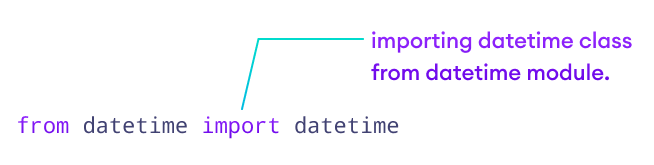
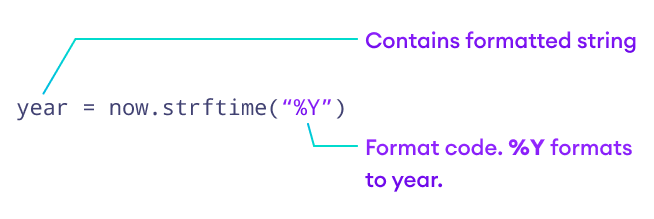
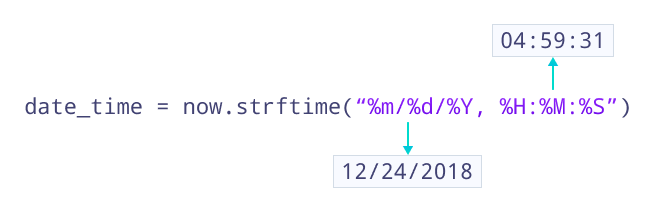
time: 04:59:31

date and time: 12/24/2018, 04:59:31

Here, year, day, time and date\_time are strings, whereas now is a datetime object.

## How strftime() works?

In the above program, %Y, %m, %d etc. are format codes. The strftime() method takes one or more format codes as an argument and returns a formatted string based on it.

1. We imported datetime class from the datetime module. It's because the object of datetime class can access strftime() method.  
    
2. The datetime object containing current date and time is stored in now variable.  
    
3. The strftime() method can be used to create formatted strings.
4. The string you pass to the strftime() method may contain more than one format codes.  
    

## Example 2: Creating string from a timestamp

from datetime import datetime

timestamp = 1528797322

date\_time = datetime.fromtimestamp(timestamp)

print("Date time object:", date\_time)

d = date\_time.strftime("%m/%d/%Y, %H:%M:%S")

print("Output 2:", d)

d = date\_time.strftime("%d %b, %Y")

print("Output 3:", d)

d = date\_time.strftime("%d %B, %Y")

print("Output 4:", d)

d = date\_time.strftime("%I%p")

print("Output 5:", d)

When you run the program, the output will be:

Date time object: 2018-06-12 09:55:22

Output 2: 06/12/2018, 09:55:22

Output 3: 12 Jun, 2018

Output 4: 12 June, 2018

Output 5: 09AM

## Format Code List

The table below shows all the codes that you can pass to the strftime() method.

| **Directive** | **Meaning** | **Example** |
| --- | --- | --- |
| %a | Abbreviated weekday name. | Sun, Mon, ... |
| %A | Full weekday name. | Sunday, Monday, ... |
| %w | Weekday as a decimal number. | 0, 1, ..., 6 |
| %d | Day of the month as a zero-padded decimal. | 01, 02, ..., 31 |
| %-d | Day of the month as a decimal number. | 1, 2, ..., 30 |
| %b | Abbreviated month name. | Jan, Feb, ..., Dec |
| %B | Full month name. | January, February, ... |
| %m | Month as a zero-padded decimal number. | 01, 02, ..., 12 |
| %-m | Month as a decimal number. | 1, 2, ..., 12 |
| %y | Year without century as a zero-padded decimal number. | 00, 01, ..., 99 |
| %-y | Year without century as a decimal number. | 0, 1, ..., 99 |
| %Y | Year with century as a decimal number. | 2013, 2019 etc. |
| %H | Hour (24-hour clock) as a zero-padded decimal number. | 00, 01, ..., 23 |
| %-H | Hour (24-hour clock) as a decimal number. | 0, 1, ..., 23 |
| %I | Hour (12-hour clock) as a zero-padded decimal number. | 01, 02, ..., 12 |
| %-I | Hour (12-hour clock) as a decimal number. | 1, 2, ... 12 |
| %p | Locale’s AM or PM. | AM, PM |
| %M | Minute as a zero-padded decimal number. | 00, 01, ..., 59 |
| %-M | Minute as a decimal number. | 0, 1, ..., 59 |
| %S | Second as a zero-padded decimal number. | 00, 01, ..., 59 |
| %-S | Second as a decimal number. | 0, 1, ..., 59 |
| %f | Microsecond as a decimal number, zero-padded on the left. | 000000 - 999999 |
| %z | UTC offset in the form +HHMM or -HHMM. |  |
| %Z | Time zone name. |  |
| %j | Day of the year as a zero-padded decimal number. | 001, 002, ..., 366 |
| %-j | Day of the year as a decimal number. | 1, 2, ..., 366 |
| %U | Week number of the year (Sunday as the first day of the week). All days in a new year preceding the first Sunday are considered to be in week 0. | 00, 01, ..., 53 |
| %W | Week number of the year (Monday as the first day of the week). All days in a new year preceding the first Monday are considered to be in week 0. | 00, 01, ..., 53 |
| %c | Locale’s appropriate date and time representation. | Mon Sep 30 07:06:05 2013 |
| %x | Locale’s appropriate date representation. | 09/30/13 |
| %X | Locale’s appropriate time representation. | 07:06:05 |
| %% | A literal '%' character. | % |

## Example 3: Locale's appropriate date and time

from datetime import datetime

timestamp = 1528797322

date\_time = datetime.fromtimestamp(timestamp)

d = date\_time.strftime("%c")

print("Output 1:", d)

d = date\_time.strftime("%x")

print("Output 2:", d)

d = date\_time.strftime("%X")

print("Output 3:", d)

When you run the program, the output will be:

Output 1: Tue Jun 12 09:55:22 2018

Output 2: 06/12/18

Output 3: 09:55:22

Format codes %c, %x and %X are used for locale's appropriate date and time representation.

We also recommend you to check [Python strptime()](https://www.programiz.com/python-programming/datetime/strptime). The strptime() method creates a datetime object from a string.

# Python strptime()

In this tutorial, you will learn to create a datetime object from a string (with the help of examples).

The strptime() method creates a [datetime](https://www.programiz.com/python-programming/datetime#datetime) object from the given string.

**Note:** You cannot create datetime object from every string. The string needs to be in a certain format.

## Example 1: string to datetime object

from datetime import datetime

date\_string = "21 June, 2018"

print("date\_string =", date\_string)

print("type of date\_string =", type(date\_string))

date\_object = datetime.strptime(date\_string, "%d %B, %Y")

print("date\_object =", date\_object)

print("type of date\_object =", type(date\_object))

When you run the program, the output will be:

date\_string = 21 June, 2018

type of date\_string = <class 'str'>

date\_object = 2018-06-21 00:00:00

type of date\_object = <class 'datetime.datetime'>

### How strptime() works?

The strptime() class method takes two arguments:

* string (that be converted to datetime)
* format code

Based on the string and format code used, the method returns its equivalent datetime object.

In the above example:



Here,

* %d - Represents the day of the month. **Example:** 01, 02, ..., 31
* %B - Month's name in full. **Example:** January, February etc.
* %Y - Year in four digits. **Example:** 2018, 2019 etc.

## Example 2: string to datetime object

from datetime import datetime

dt\_string = "12/11/2018 09:15:32"

# Considering date is in dd/mm/yyyy format

dt\_object1 = datetime.strptime(dt\_string, "%d/%m/%Y %H:%M:%S")

print("dt\_object1 =", dt\_object1)

# Considering date is in mm/dd/yyyy format

dt\_object2 = datetime.strptime(dt\_string, "%m/%d/%Y %H:%M:%S")

print("dt\_object2 =", dt\_object2)

When you run the program, the output will be:

dt\_object1 = 2018-11-12 09:15:32

dt\_object2 = 2018-12-11 09:15:32

## Format Code List

The table below shows all the format codes that you can use.

| **Directive** | **Meaning** | **Example** |
| --- | --- | --- |
| %a | Abbreviated weekday name. | Sun, Mon, ... |
| %A | Full weekday name. | Sunday, Monday, ... |
| %w | Weekday as a decimal number. | 0, 1, ..., 6 |
| %d | Day of the month as a zero-padded decimal. | 01, 02, ..., 31 |
| %-d | Day of the month as a decimal number. | 1, 2, ..., 30 |
| %b | Abbreviated month name. | Jan, Feb, ..., Dec |
| %B | Full month name. | January, February, ... |
| %m | Month as a zero-padded decimal number. | 01, 02, ..., 12 |
| %-m | Month as a decimal number. | 1, 2, ..., 12 |
| %y | Year without century as a zero-padded decimal number. | 00, 01, ..., 99 |
| %-y | Year without century as a decimal number. | 0, 1, ..., 99 |
| %Y | Year with century as a decimal number. | 2013, 2019 etc. |
| %H | Hour (24-hour clock) as a zero-padded decimal number. | 00, 01, ..., 23 |
| %-H | Hour (24-hour clock) as a decimal number. | 0, 1, ..., 23 |
| %I | Hour (12-hour clock) as a zero-padded decimal number. | 01, 02, ..., 12 |
| %-I | Hour (12-hour clock) as a decimal number. | 1, 2, ... 12 |
| %p | Locale’s AM or PM. | AM, PM |
| %M | Minute as a zero-padded decimal number. | 00, 01, ..., 59 |
| %-M | Minute as a decimal number. | 0, 1, ..., 59 |
| %S | Second as a zero-padded decimal number. | 00, 01, ..., 59 |
| %-S | Second as a decimal number. | 0, 1, ..., 59 |
| %f | Microsecond as a decimal number, zero-padded on the left. | 000000 - 999999 |
| %z | UTC offset in the form +HHMM or -HHMM. |  |
| %Z | Time zone name. |  |
| %j | Day of the year as a zero-padded decimal number. | 001, 002, ..., 366 |
| %-j | Day of the year as a decimal number. | 1, 2, ..., 366 |
| %U | Week number of the year (Sunday as the first day of the week). All days in a new year preceding the first Sunday are considered to be in week 0. | 00, 01, ..., 53 |
| %W | Week number of the year (Monday as the first day of the week). All days in a new year preceding the first Monday are considered to be in week 0. | 00, 01, ..., 53 |
| %c | Locale’s appropriate date and time representation. | Mon Sep 30 07:06:05 2013 |
| %x | Locale’s appropriate date representation. | 09/30/13 |
| %X | Locale’s appropriate time representation. | 07:06:05 |
| %% | A literal '%' character. | % |

### ValueError in strptime()

If the string (first argument) and the format code (second argument) passed to the strptime() doesn't match, you will get ValueError. For example:

from datetime import datetime

date\_string = "12/11/2018"

date\_object = datetime.strptime(date\_string, "%d %m %Y")

print("date\_object =", date\_object)

If you run this program, you will get an error.

ValueError: time data '12/11/2018' does not match format '%d %m %Y'

**Recommended Readings:** [Python strftime()](https://www.programiz.com/python-programming/datetime/strftime)

# How to get current date and time in Python?

In this tutorial, you will learn to get today's date and current date and time in Python with the help of examples.

There are a number of ways we can take to get the current date. We will use the date class of the [datetime](https://www.programiz.com/python-programming/datetime) module to accomplish this task.

## Example 1: Python get today's date

from datetime import date

today = date.today()

print("Today's date:", today)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Today's date: 2022-12-27

Here, we imported the date class from the datetime module. Then, we used the date.today() method to get the current local date.

## Example 2: Current date in different formats

from datetime import date

today = date.today()

# dd/mm/YY

d1 = today.strftime("%d/%m/%Y")

print("d1 =", d1)

# Textual month, day and year

d2 = today.strftime("%B %d, %Y")

print("d2 =", d2)

# mm/dd/y

d3 = today.strftime("%m/%d/%y")

print("d3 =", d3)

# Month abbreviation, day and year

d4 = today.strftime("%b-%d-%Y")

print("d4 =", d4)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

d1 = 27/12/2022

d2 = December 27, 2022

d3 = 12/27/22

d4 = Dec-27-2022

## Get the current date and time in Python

If we need to get the current date and time, you can use the datetime class of the datetime module.

from datetime import datetime

# datetime object containing current date and time

now = datetime.now()

print("now =", now)

# dd/mm/YY H:M:S

dt\_string = now.strftime("%d/%m/%Y %H:%M:%S")

print("date and time =", dt\_string)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

now = 2022-12-27 10:09:20.430322

date and time = 27/12/2022 10:09:20

Here, we have used datetime.now() to get the current date and time.

Then, we used strftime() to create a string representing date and time in another format.

# Python Get Current time

In this tutorial, we will learn to get the current time of your locale as well as different time zones in Python.

There are a number of ways we can take to get current time in Python.

* Using the datetime object
* Using the time module

## Current time using the datetime object

from datetime import datetime

now = datetime.now()

current\_time = now.strftime("%H:%M:%S")

print("Current Time =", current\_time)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Current Time = 07:41:19

In the above example, we have imported the datetime class from the [datetime](https://www.programiz.com/python-programming/datetime) module.

Then, we used the now() function to get a datetime object containing current date and time.

Using [datetime.strftime()](https://www.programiz.com/python-programming/datetime/strftime) function, we then created a string representing current time.

## Current time using time module

In Python, we can also get the current time using the time module.

import time

t = time.localtime()

current\_time = time.strftime("%H:%M:%S", t)

print(current\_time)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

07:46:58

## Current time of a Certain timezone

If we need to find the current time of a certain timezone, you can use [the pytZ module](http://pytz.sourceforge.net/).

from datetime import datetime

import pytz

# Get the timezone object for New York

tz\_NY = pytz.timezone('America/New\_York')

# Get the current time in New York

datetime\_NY = datetime.now(tz\_NY)

# Format the time as a string and print it

print("NY time:", datetime\_NY.strftime("%H:%M:%S"))

# Get the timezone object for London

tz\_London = pytz.timezone('Europe/London')

# Get the current time in London

datetime\_London = datetime.now(tz\_London)

# Format the time as a string and print it

print("London time:", datetime\_London.strftime("%H:%M:%S"))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

NY time: 03:45:16

London time: 08:45:16

Here, we have used the pytz module to find the current time of a certain time zone

# Python timestamp to datetime and vice-versa

In this tutorial, we will learn to convert timestamp to datetime object and datetime object to timestamp with the help of examples.

It's pretty common to store date and time as a timestamp in a database. A Unix timestamp is the number of seconds between a particular date and January 1, 1970 at UTC.

We can simply use the fromtimestamp() method from the datetime module to get a date from a UNIX timestamp.

## Python timestamp to datetime

from datetime import datetime

# timestamp is number of seconds since 1970-01-01

timestamp = 1545730073

# convert the timestamp to a datetime object in the local timezone

dt\_object = datetime.fromtimestamp(timestamp)

# print the datetime object and its type

print("dt\_object =", dt\_object)

print("type(dt\_object) =", type(dt\_object))

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

dt\_object = 2018-12-25 09:27:53

type(dt\_object) = <class 'datetime.datetime'>

Here, we have imported the datetime class from the [datetime](https://www.programiz.com/python-programming/datetime) module.

Then, we used the datetime.fromtimestamp() class method which returns the local date and time (datetime object). This object is stored in the dt\_object variable.

**Note:** We can easily create a string representing date and time from a datetime object using [strftime()](https://www.programiz.com/python-programming/datetime/strftime) method.

## Python datetime to timestamp

In Python, we can get timestamp from a datetime object using the datetime.timestamp() method. For example,

from datetime import datetime

# current date and time

now = datetime.now()

# convert from datetime to timestamp

ts = datetime.timestamp(now)

print("Timestamp =", ts)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Timestamp = 1672138646.118119

Here, the datetime.timestamp() method takes a datetime object as an argument and returns a Unix timestamp.

# Python time Module

In this tutorial, we will explore time module in detail. We will learn to use different time-related functions defined in the time module with the help of examples.

The time module in Python provides functions for handling time-related tasks.

The time-related tasks includes,

* reading the current time
* formatting time
* sleeping for a specified number of seconds and so on.

## Python time.time() Function

In Python, the time() function returns the number of seconds passed since epoch (the point where time begins).

For the Unix system, January 1, 1970, 00:00:00 at **UTC** is epoch.

Let's see an example,

# import the time module

import time

# get the current time in seconds since the epoch

seconds = time.time()

print("Seconds since epoch =", seconds)

# Output: Seconds since epoch = 1672214933.6804628

[Run Code](https://www.programiz.com/python-programming/online-compiler)

In the above example, we have used the time.time() function to get the current time in seconds since the epoch, and then printed the result.

## Python time.ctime() Function

The time.ctime() function in Python takes seconds passed since epoch as an argument and returns a string representing local time.

import time

# seconds passed since epoch

seconds = 1672215379.5045543

# convert the time in seconds since the epoch to a readable format

local\_time = time.ctime(seconds)

print("Local time:", local\_time)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Local time: Wed Dec 28 08:16:19 2022

Here, we have used the time.ctime() function to convert the time in seconds since the epoch to a readable format, and then printed the result.

## Python time.sleep() Function

The sleep() function suspends (delays) execution of the current thread for the given number of seconds.

import time

print("Printed immediately.")

time.sleep(2.4)

print("Printed after 2.4 seconds.")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Printed immediately.

Printed after 2.4 seconds.

Here's how this program works:

* "Printed immediately" is printed
* time.sleep(2.4) suspends execution for 2.4 seconds.
* "Printed after 2.4 seconds" is printed.

To learn more about sleep(), please visit: [Python sleep()](https://www.programiz.com/python-programming/time/sleep).

## Python time.localtime() Function

The localtime() function takes the number of seconds passed since epoch as an argument and returns struct\_time (a tuple containing **9** elements corresponding to struct\_time) in **local time**.

import time

result = time.localtime(1672214933)

print("result:", result)

print("\nyear:", result.tm\_year)

print("tm\_hour:", result.tm\_hour)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

result: time.struct\_time(tm\_year=2022, tm\_mon=12, tm\_mday=28, tm\_hour=8, tm\_min=8, tm\_sec=53, tm\_wday=2, tm\_yday=362, tm\_isdst=0)

year: 2022

tm\_hour: 8

Here, if no argument or None is passed to localtime(), the value returned by time() is used.

## Python time.gmtime() Function

The gmtime() function takes the number of seconds passed since epoch as an argument and returns struct\_time in **UTC**.

import time

result = time.gmtime(1672214933)

print("result:", result)

print("\nyear:", result.tm\_year)

print("tm\_hour:", result.tm\_hour)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

result: time.struct\_time(tm\_year=2022, tm\_mon=12, tm\_mday=28, tm\_hour=8, tm\_min=8, tm\_sec=53, tm\_wday=2, tm\_yday=362, tm\_isdst=0)

year: 2022

tm\_hour: 8

Here, if no argument or None is passed to gmtime(), the value returned by time() is used.

## Python time.mktime() Function

The mktime() function takes struct\_time (a tuple containing **9** elements corresponding to struct\_time) as an argument and returns the seconds passed since epoch in local time.

The struct\_time has the following structure:

(year, month, day, hour, minute, second, weekday, day of the year, daylight saving)

Let's see an example,

import time

time\_tuple = (2022, 12, 28, 8, 44, 4, 4, 362, 0)

# convert time\_tuple to seconds since epoch

seconds = time.mktime(time\_tuple)

print(seconds)

# Output: 1672217044.0

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we have converted the time\_tuple to seconds since the epoch.

## Python time.asctime() Function

In Python, the asctime() function takes struct\_time as an argument and returns a string representing it.

Similar to mktime(), the time\_tuple has the following structure:

(year, month, day, hour, minute, second, weekday, day of the year, daylight saving)

Let's see an example,

import time

t = (2022, 12, 28, 8, 44, 4, 4, 362, 0)

result = time.asctime(t)

print("Result:", result)

# Output: Result: Fri Dec 28 08:44:04 2022

[Run Code](https://www.programiz.com/python-programming/online-compiler)

Here, we can see time.asctime() converts the time tuple to a human-readable string.

## Python time.strftime() Function

The strftime() function takes struct\_time (or tuple corresponding to it) as an argument and returns a string representing it based on the format code used. For example,

import time

named\_tuple = time.localtime() # get struct\_time

time\_string = time.strftime("%m/%d/%Y, %H:%M:%S", named\_tuple)

print(time\_string)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

12/29/2022, 08:36:22

Here, %Y, %m, %d, %H etc. are format codes.

* %Y - year [0001,..., 2018, 2019,..., 9999]
* %m - month [01, 02, ..., 11, 12]
* %d - day [01, 02, ..., 30, 31]
* %H - hour [00, 01, ..., 22, 23
* %M - minutes [00, 01, ..., 58, 59]
* %S - second [00, 01, ..., 58, 61]

To learn more, visit: [time.strftime()](https://docs.python.org/3/library/time.html#time.strftime).

## Python time.strptime() Function

The strptime() function parses a string representing time and returns struct\_time.

import time

time\_string = "14 July, 2023"

result = time.strptime(time\_string, "%d %B, %Y")

print(result)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

time.struct\_time(tm\_year=2023, tm\_mon=7, tm\_mday=14, tm\_hour=0, tm\_min=0, tm\_sec=0, tm\_wday=4, tm\_yday=195, tm\_isdst=-1)

Here, strptime() parses a string and convert it to the struct\_time object.

# Python sleep() Function

In the tutorial, we will learn about the sleep() function with the help of examples.

Python has a module named [time](https://www.programiz.com/python-programming/time) which provides several useful methods to handle time-related tasks. One of the most popular methods among them is sleep().

The sleep() method suspends execution of the current thread for a given number of seconds.

## Example 1: Python sleep() Method

import time

print("Printed immediately.")

time.sleep(2.4)

print("Printed after 2.4 seconds.")

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Printed immediately.

Printed after 2.4 seconds.

Here's how this program works:

* "Printed immediately" is printed
* time.sleep(2.4) suspends execution for 2.4 seconds.
* "Printed after 2.4 seconds" is printed.

**Before Python 3.5**, the actual suspension time may be less than the argument specified to the time() function.

**Since Python 3.5**, the suspension time will be at least the seconds specified.

## Create a Digital Clock in Python

import time

while True:

localtime = time.localtime()

result = time.strftime("%I:%M:%S %p", localtime)

print(result)

time.sleep(1)

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

02:10:50 PM

02:10:51 PM

02:10:52 PM

02:10:53 PM

02:10:54 PM

... ... ...

In the above example, we computed and printed the current local time inside the infinite [while loop](https://www.programiz.com/python-programming/while-loop).

Then, the program waits for **1** second. Again, the current local time is computed and printed. This process goes on.

To learn more, visit [digital clock in Python shell](https://stackoverflow.com/questions/37515587/run-a-basic-digital-clock-in-the-python-shell).

## Multithreading in Python

Before talking about the sleep() method in multithreaded programs, let's talk about processes and threads.

A computer program is a collection of instructions. A process is the execution of those instructions.

A thread is a subset of the process. A process can have one or more threads.

### Example 3: Python multithreading

All the programs above in this article are single-threaded programs. Here's an example of a multithreaded Python program.

import threading

def print\_hello\_three\_times():

for i in range(3):

print("Hello")

def print\_hi\_three\_times():

for i in range(3):

print("Hi")

t1 = threading.Thread(target=print\_hello\_three\_times)

t2 = threading.Thread(target=print\_hi\_three\_times)

t1.start()

t2.start()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Hello

Hello

Hi

Hello

Hi

Hi

The above example has two threads t1 and t2. These threads are run using t1.start() and t2.start() statements.

Note that, t1 and t2 run concurrently and we might get different outputs.

To learn more about multithreading, visit [Multithreading in Python](https://stackoverflow.com/questions/2846653/how-to-use-threading-in-python).

## Python time.sleep() in multithreaded programs

In case of single-threaded programs, sleep() suspends execution of the thread and process.

However, the method suspends a thread rather than the whole process in multithreaded programs.

### Example 4: sleep() in a multithreaded program

import threading

import time

def print\_hello():

for i in range(4):

time.sleep(0.5)

print("Hello")

def print\_hi():

for i in range(4):

time.sleep(0.7)

print("Hi")

t1 = threading.Thread(target=print\_hello)

t2 = threading.Thread(target=print\_hi)

t1.start()

t2.start()

[Run Code](https://www.programiz.com/python-programming/online-compiler)

**Output**

Hello

Hi

Hello

Hi

Hello

Hello

Hi

Hi

The above program has two threads. We have used time.sleep(0.5) and time.sleep(0.75) to suspend execution of these two threads for **0.5** seconds and **0.7** seconds respectively.

**Recommended Reading:** [Python time.sleep() sleeps thread](https://stackoverflow.com/questions/92928/time-sleep-sleeps-thread-or-process)