**What is Python?**

Python is a very popular general-purpose interpreted, interactive, object-oriented, and high-level programming language. Python is dynamically-typed and garbage-collected programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

Python supports multiple programming paradigms, including Procedural, Object Oriented and Functional programming language. Python design philosophy emphasizes code readability with the use of significant indentation.

This Python tutorial gives a complete understanding of Python programming language, starting from basic concepts to advanced concepts. This tutorial will take you through simple and practical approaches while learning Python Programming language.

**Python "Hello, World!"**

To start with Python programming, the very basic program is to print "Hello, World!" You can use the print() function. Below is an example of Python code to print

Hello, World!" −

Open Compiler

# Python code to print "Hello, World!"

print ("Hello, World!")

Characteristics of Python

Following are important characteristics of **Python Programming** −

* It supports functional and structured programming methods as well as [OOP](https://www.tutorialspoint.com/python/python_oops_concepts.htm).
* It can be used as a scripting language or can be compiled to byte-code for building large applications.
* It provides very high-level dynamic data types and supports dynamic type checking.
* It supports automatic garbage collection.
* It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

**Applications of Python**

[Python](https://www.tutorialspoint.com/python/python_overview.htm) is a general-purpose programming language. It is suitable for the development of a wide range of software applications. Over the last few years Python has been the preferred language of choice for developers in the following application areas –

**Data Science**

Python's recent meteoric rise in the popularity charts is largely due to its Data science libraries. Python has become an essential skill for data scientists. Today, real time web applications, mobile applications and other devices generate huge amount of data. Python's data science libraries help companies generate business insights from this data.

Libraries like [NumPy](https://www.tutorialspoint.com/numpy/index.htm), [Pandas](https://www.tutorialspoint.com/python_pandas/index.htm), and [Matplotlib](https://www.tutorialspoint.com/matplotlib/index.htm) are extensively used to apply mathematical algorithms to the data and generate [visualizations](https://www.tutorialspoint.com/python_pandas/python_pandas_visualization.htm). Commercial and community Python distributions like Anaconda and Active State bundle all the essential libraries required for data science.

**Machine Learning**

Python libraries such as [Scikit-learn](https://www.tutorialspoint.com/scikit_learn/index.htm) and [TensorFlow](https://www.tutorialspoint.com/tensorflow/index.htm) help in building models for prediction of trends like customer satisfaction, projected values of stocks etc. based upon the past data. [Machine learning](https://www.tutorialspoint.com/machine_learning/index.htm) applications include (but not restricted to) medical diagnosis, statistical arbitrage, basket analysis, sales prediction etc.

**Web Development**

Python's web frameworks facilitate rapid web application development. [Django](https://www.tutorialspoint.com/django/index.htm), [Pyramid](https://www.tutorialspoint.com/python_pyramid/index.htm), [Flask](https://www.tutorialspoint.com/flask/index.htm) are very popular among the web developer community. etc. make it very easy to develop and deploy simple as well as complex web applications.

Latest versions of Python provide asynchronous programming support. Modern web frameworks leverage this feature to develop fast and high performance web apps and APIs.

**Computer Vision and Image processing**

[OpenCV](https://www.tutorialspoint.com/opencv_python/index.htm) is a widely popular library for capturing and processing images. Image processing algorithms extract information from images, reconstruct image and video data. Computer Vision uses image processing for face detection and pattern recognition. OpenCV is a C++ library. Its Python port is extensively used because of its rapid development feature.

Some of the application areas of computer vision are [robotics](https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_robotics.htm), industrial surveillance, automation, and [biometrics](https://www.tutorialspoint.com/biometrics/index.htm) etc.

**Embedded Systems and IoT**

Micropython (<https://micropython.org/>), a lightweight version especially for microcontrollers like [Arduino](https://www.tutorialspoint.com/arduino/index.htm). Many automation products, robotics, [IoT](https://www.tutorialspoint.com/internet_of_things/index.htm), and kiosk applications are built around Arduino and programmed with Micropython. [Raspberry Pi](https://www.tutorialspoint.com/raspberry_pi/index.htm) is also very popular alow cost single board computer used for these type of applications.

**Job Scheduling and Automation**

Python found one of its first applications in automating CRON (Command Run ON) jobs. Certain tasks like periodic data backups, can be written in Python scripts scheduled to be invoked automatically by operating system scheduler.

Many software products like Maya embed Python API for writing automation scripts (something similar to Excel micros).

**Desktop GUI Applications**

Python is a great option for building ergonomic, attractive, and user-friendly desktop GUI applications. Several graphics libraries, though built in C/C++, have been ported to Python. The popular Qt graphics toolkit is available as a [PyQt](https://www.tutorialspoint.com/pyqt/index.htm" \t "_blank) package in Python. Similarly, WxWidgets has been ported to Python as [WxPython](https://www.tutorialspoint.com/wxpython/index.htm" \t "_blank). Python's built-in GUI package, TKinter is a Python interface to the Tk Graphics toolkit.

Here is a select list of Python GUI libraries:

* Tkinter − Tkinter is the Python interface to the Tk GUI toolkit shipped with Python's standard library.
* wxPython − This is the Python interface for the wxWidgets GUI toolkit. BitTorrent Client application has been built with wxPython functionality.
* PyQt – Qt is one of the most popular GUI toolkits. It has been ported to Python as a PyQt5 package. Notable desktop GUI apps that use PyQt include QGIS, Spyder IDE, Calibre Ebook Manager, etc.
* PyGTK − PyGTK is a set of wrappers written in Python and C for GTK + GUI library. The complete PyGTK tutorial is available here.
* [PySimpleGUI](https://www.tutorialspoint.com/pysimplegui/index.htm) − PySimpleGui is an open-source, cross-platform GUI library for Python. It aims to provide a uniform API for creating desktop GUIs based on Python's Tkinter, PySide, and WxPython toolkits.
* Jython − Jython is a Python port for Java, which gives Python scripts seamless access to the Java GUI libraries on the local machine.

**Console-based Applications**

Python is often employed to build CLI (command-line interface) applications. Such scripts can be used to run scheduled CRON jobs such as taking database backups etc. There are many Python libraries that parse the command line arguments. The argparse library comes bundled with Pythons standard library. You can use Click (part of Flask framework) and Typer (included in FastAPI framework) to build console interfaces to the web-based applications built by the respective frameworks. Textual is a rapid development framework to build apps that run inside a terminal as well as browsers.

**CAD Applications**

CAD engineers can take advantage of Python's versatility to automate repetitive tasks such as drawing shapes and generating reports.

Autodesk Fusion 360 is a popular CAD software, which has a Python API that allows users to automate tasks and create custom tools. Similarly, SolidWorks has a built-in Python shell that allows users to run Python scripts inside the software.

CATIA is another very popular CAD software. Along with a VBScript, certain third-party Python libraries that can be used to control CATIA.

**Game Development**

Some popular gaming apps have been built with Python. Examples include BattleField2, The Sims 4, World of Tanks, Pirates of the Caribbean, and more. These apps are built with one of the following Python libraries.

Pygame is one of the most popular Python libraries used to build engaging computer games. Pygame is an open-source Python library for making multimedia applications like games built on top of the excellent SDL library. It is a cross-platform library, which means you can build a game that can run on any operating system platform.

Another library Kivy is also widely used to build desktop as well as mobile-based games. Kivy has a multi-touch interface. It is an open-source and cross-platform Python library for rapid development of game applications. Kivy runs on Linux, Windows, OS X, Android, iOS, and Raspberry Pi.

PyKyra library is based on both SDL (Software and Documentation Localisation) and the Kyra engine. It is one of the fastest game development frameworks. PyKyra supports MPEG , MP3, Ogg Vorbis, Wav, etc., multimedia formats.

**Features of Python**

The latest release of Python is 3.x. As mentioned before, Python is one of the most widely used languages on the web. I'm going to list a few of them here:

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

**Python Overview**

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

* Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
* Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
* Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
* Python is a Beginner's Language − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python is an open-source and cross-platform programming language. It is available for use under Python Software Foundation License (compatible to GNU General Public License) on all the major operating system platforms Linux, Windows and Mac OS.

To facilitate new features and to maintain that readability, the Python Enhancement Proposal (PEP) process was developed. This process allows anyone to submit a PEP for a new feature, library, or other addition.

The design philosophy of Python emphasizes on simplicity, readability and unambiguity. Python is known for its batteries included approach as Python software is distributed with a comprehensive standard library of functions and modules.

Python's design philosophy is documented in the Zen of Python. It consists of nineteen aphorisms such as −

* Beautiful is better than ugly
* Explicit is better than implicit
* Simple is better than complex
* Complex is better than complicated

**Pythonic Code Style**

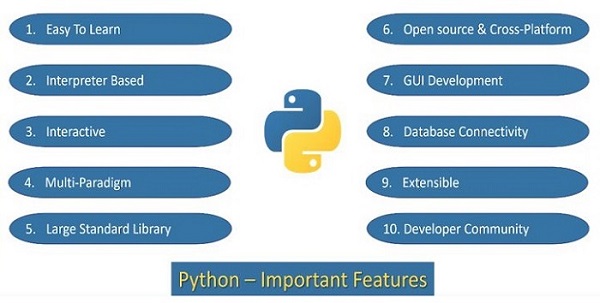
Python leaves you free to choose to program in an object-oriented, procedural, functional, aspect-oriented, or even logic-oriented way. These freedoms make Python a great language to write clean and beautiful code.

Pythonic Code Style is actually more of a design philosophy and suggests to write a code which is:

* Clean
* Simple
* Beautiful
* Explicit
* Readable

**Python – Features**

Python is a feature-rich, high-level, interpreted, interactive, and object-oriented scripting language. Python is a versatile and very popular programming language due to its features such as readability, simplicity, extensive libraries, and many more. In this tutorial, we will learn about the various features of Python that make it a powerful and versatile programming language.



Python's most important features are as follows:

**Easy to Learn**

This is one of the most important reasons for the popularity of Python. Python has a limited set of keywords. Its features such as simple [syntax](https://www.tutorialspoint.com/python/python_basic_syntax.htm), usage of indentation to avoid clutter of curly brackets and dynamic typing that doesn't necessitate prior declaration of variable help a beginner to learn Python quickly and easily.

**Dynamically Typed**

Python is a dynamically typed programming language. In Python, you don't need to specify the variable time at the time of the variable declaration. The types are specified at the runtime based on the assigned value due to its dynamically typed feature.

**Interpreter Based**

Instructions in any programming languages must be translated into machine code for the processor to execute them. Programming languages are either compiler based or interpreter based.

In case of a compiler, a [machine language](https://www.tutorialspoint.com/machine_learning/index.htm) version of the entire source program is generated. The conversion fails even if there is a single erroneous statement. Hence, the development process is tedious for the beginners. The C family languages (including [C](https://www.tutorialspoint.com/cprogramming/index.htm), [C++](https://www.tutorialspoint.com/cplusplus/index.htm), [Java](https://www.tutorialspoint.com/java/index.htm), [C#](https://www.tutorialspoint.com/csharp/index.htm) etc) are compiler based.

Python is an interpreter based language. The interpreter takes one instruction from the source code at a time, translates it into machine code and executes it. Instructions before the first occurrence of error are executed. With this feature, it is easier to debug the program and thus proves useful for the beginner level programmer to gain confidence gradually. Python therefore is a beginner-friendly language.

**Interactive**

Standard Python distribution comes with an interactive shell that works on the principle of REPL (Read Evaluate Print Loop). The shell presents a Python prompt >>>. You can type any valid Python expression and press Enter. Python interpreter immediately returns the response and the prompt comes back to read the next expression.

>>> 2\*3+1

7

>>> print ("Hello World")

Hello World

The interactive mode is especially useful to get familiar with a library and test out its functionality. You can try out small code snippets in interactive mode before writing a program.

**Multi-paradigm**

Python is a completely [object-oriented](https://www.tutorialspoint.com/python/python_oops_concepts.htm) language. Everything in a Python program is an [object](https://www.tutorialspoint.com/python/python_object_classes.htm). However, Python conveniently encapsulates its object orientation to be used as an imperative or procedural language such as C. Python also provides certain functionality that resembles functional programming. Moreover, certain third-party tools have been developed to support other programming paradigms such as aspect-oriented and logic programming.

Standard Library

Even though it has a very few keywords (only Thirty Five), Python software is distributed with a standard library made of large number of modules and packages. Thus Python has out of box support for programming needs such as serialization, data compression, internet data handling, and many more. Python is known for its batteries included approach.

Some of the Python's popular modules are:

* [NumPy](https://www.tutorialspoint.com/numpy/index.htm)
* [Pandas](https://www.tutorialspoint.com/python_pandas/index.htm)
* [Matplotlib](https://www.tutorialspoint.com/matplotlib/index.htm)
* Tkinter
* [Math](https://www.tutorialspoint.com/python/python_maths.htm)

**Open Source and Cross Platform**

Python's standard distribution can be downloaded from  <https://www.python.org/downloads/> without any restrictions. You can download pre-compiled binaries for various operating system platforms. In addition, the source code is also freely available, which is why it comes under open-source category.

Python software (along with the documentation) is distributed under Python Software Foundation License. It is a BSD style permissive software license and compatible to GNU GPL (General Public License).

Python is a cross-platform language. Pre-compiled binaries are available for use on various operating system platforms such as [Windows](https://www.tutorialspoint.com/windows10/index.htm), [Linux](https://www.tutorialspoint.com/unix/index.htm), Mac OS, [Android OS](https://www.tutorialspoint.com/android/index.htm). The reference implementation of Python is called CPython and is written in C. You can download the source code and compile it for your OS platform.

A Python program is first compiled to an intermediate platform independent byte code. The virtual machine inside the interpreter then executes the byte code. This behaviour makes Python a cross-platform language, and thus a Python program can be easily ported from one OS platform to other.

**GUI Applications**

Python's standard distribution has an excellent graphics library called TKinter. It is a Python port for the vastly popular GUI toolkit called TCL/Tk. You can build attractive user-friendly GUI applications in Python. GUI toolkits are generally written in C/C++. Many of them have been ported to Python. Examples are [PyQt](https://www.tutorialspoint.com/pyqt/index.htm" \t "_blank), [WxWidgets](https://www.tutorialspoint.com/wxpython/index.htm" \t "_blank), [PySimpleGUI](https://www.tutorialspoint.com/pysimplegui/index.htm" \t "_blank) etc.

**Database Connectivity**

Almost any type of database can be used as a backend with the Python application. DB-API is a set of specifications for database driver software to let Python communicate with a relational database. With many third-party libraries, Python can also work with NoSQL databases such as [MongoDB](https://www.tutorialspoint.com/mongodb/index.htm).

**Extensible**

The term extensibility implies the ability to add new features or modify existing features. As stated earlier, CPython (which is Python's reference implementation) is written in C. Hence one can easily write modules/libraries in C and incorporate them in the standard library. There are other implementations of Python such as Jython (written in Java) and [IPython](https://www.tutorialspoint.com/jupyter/ipython_introduction.htm" \t "_blank) (written in C#). Hence, it is possible to write and merge new functionality in these implementations with Java and C# respectively.

**Active Developer Community**

As a result of Python's popularity and open-source nature, a large number of Python developers often interact with online forums and conferences. Python Software Foundation also has a significant member base, involved in the organization's mission to "**Promote, Protect, and Advance the Python Programming Language**"

Python also enjoys a significant institutional support. Major IT companies Google, Microsoft, and Meta contribute immensely by preparing documentation and other resources.

Apart from the above-mentioned features, Python has another big list of good features, few are listed below −

* It supports functional and structured programming methods as well as OOP.
* It can be used as a scripting language or can be compiled to byte-code for building large applications.
* It provides very high-level dynamic [data types](https://www.tutorialspoint.com/python/python_data_types.htm) and supports dynamic type checking.
* It supports automatic garbage collection.
* It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

**Python - Syntax**

The Python syntax defines a set of rules that are used to create a Python Program. The Python Programming Language Syntax has many similarities to Perl, C, and Java Programming Languages. However, there are some definite differences between the languages.

**First Python Program**

Let us execute a [Python program to print "Hello, World!"](https://www.tutorialspoint.com/python/python_hello_world.htm) in two different modes of Python Programming. (a) Interactive Mode Programming (b) Script Mode Programming.

Python - Interactive Mode Programming

We can invoke a [Python interpreter](https://www.tutorialspoint.com/online_python_compiler.php) from command line by typing **python** at the command prompt as following –

$ python3

Python 3.10.6 (main, Mar 10 2023, 10:55:28) [GCC 11.3.0] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>>

Here **>>>** denotes a Python Command Prompt where you can type your commands. Let's type the following text at the Python prompt and press the Enter −

>>> print ("Hello, World!")

If you are running older version of Python, like Python 2.4.x, then you would need to use print statement without parenthesis as in **print "Hello, World!"**. However in Python version 3.x, this produces the following result −

Hello, World!

**Python Identifiers**

A Python identifier is a name used to identify a [variable](https://www.tutorialspoint.com/python/python_variables.htm), [function](https://www.tutorialspoint.com/python/python_functions.htm), [class](https://www.tutorialspoint.com/python/python_object_classes.htm), [module](https://www.tutorialspoint.com/python/python_modules.htm) or other object. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as &commat;, $, and % within identifiers.

*Python is a case sensitive programming language. Thus,****Manpower****and****manpower****are two different identifiers in Python.*

Here are naming conventions for Python identifiers −

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

**Python Reserved Words**

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

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

**Python Lines and Indentation**

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

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

if True:

print ("True"

else:

print ("False")

**Python Multi-Line Statements**

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example −

total = item\_one + \

item\_two + \

item\_three

Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example, following statement works well in Python –

days = ['Monday', 'Tuesday', 'Wednesday',

'Thursday', 'Friday']

**Quotations in Python**

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

The triple quotes are used to span the string across multiple lines. For example, all the following are legal –

word = 'word'

print (word)

sentence = "This is a sentence."

print (sentence)

paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""

print (paragraph)

**Comments in Python**

A comment is a programmer-readable explanation or annotation in the Python source code. They are added with the purpose of making the source code easier for humans to understand, and are ignored by Python interpreter

Just like most modern languages, Python supports single-line (or end-of-line) and multi-line (block) comments. Python comments are very much similar to the comments available in PHP, BASH and Perl Programming languages.

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

# First comment

print ("Hello, World!") # Second comment

This produces the following result −

Hello, World!

You can type a comment on the same line after a statement or expression −

name = "Madisetti" # This is again comment

You can comment multiple lines as follows −

# This is a comment.

# This is a comment, too.

# This is a comment, too.

# I said that already.

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comment:

''' This is a multiline

comment.

'''

**Using Blank Lines in Python Programs**

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

In an interactive interpreter session, you must enter an empty physical line to terminate a multiline statement.

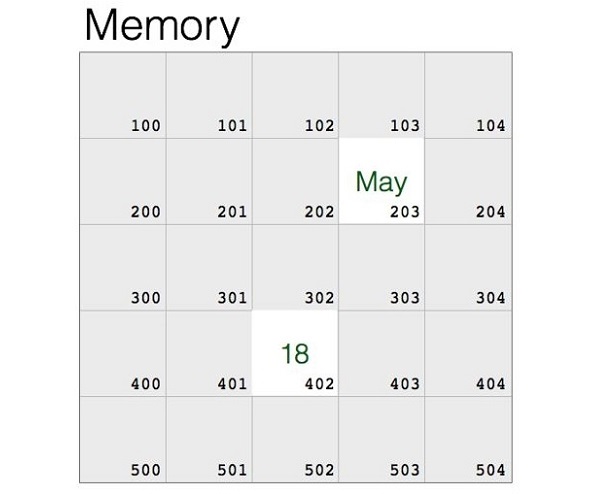
**Python Variables**

Python variables are the reserved memory locations used to store values with in a Python Program. This means that when you create a variable you reserve some space in the memory.

Based on the data type of a variable, memory space is allocated to it. Therefore, by assigning different data types to Python variables, you can store integers, decimals or characters in these variables.

**Memory Addresses**

Data items belonging to different data types are stored in computer's memory. Computer's memory locations are having a number or address, internally represented in binary form. Data is also stored in binary form as the computer works on the principle of binary representation. In the following diagram, a string **May** and a number **18** is shown as stored in memory locations.



If you know the assembly language, you will covert these data items and the memory address, and give a machine language instruction. However, it is not easy for everybody. Language translator such as Python interpreter performs this type of conversion. It stores the object in a randomly chosen memory location. Python's built-in **id()** function returns the address where the object is stored.

>>> "May"

May

>>> id("May")

2167264641264

>>> 18

18

>>> id(18)

140714055169352

Once the data is stored in the memory, it should be accessed repeatedly for performing a certain process. Obviously, fetching the data from its ID is cumbersome. High level languages like Python make it possible to give a suitable alias or a label to refer to the memory location.

In the above example, let us label the location of May as month, and location in which 18 is stored as age. Python uses the assignment operator (=) to bind an object with the label.

>>> month="May"

>>> age=18

The data object (May) and its name (month) have the same id(). The id() of 18 and age are also same.

The label is an identifier. It is usually called as a variable. A Python variable is a symbolic name that is a reference or pointer to an object.

The label is an identifier. It is usually called as a variable. A Python variable is a symbolic name that is a reference or pointer to an object.

**Creating Python Variables**

Python variables do not need explicit declaration to reserve memory space or you can say to create a variable. A Python variable is created automatically when you assign a value to it. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

**Example to Create Python Variables**

This example creates different types (an integer, a float, and a string) of variables.

counter = 100 # Creates an integer variable

miles = 1000.0 # Creates a floating point variable

name = "Zara Ali" # Creates a string variable

Printing Python Variables

Once we create a Python variable and assign a value to it, we can print it using **print()** function. Following is the extension of previous example and shows how to print different variables in Python:

Example to Print Python Variables

This example prints variables.

counter = 100 # Creates an integer variable

miles = 1000.0 # Creates a floating point variable

name = "Zara Ali" # Creates a string variable

print (counter)

print (miles)

print (name)

Here, 100, 1000.0 and "Zara Ali" are the values assigned to *counter*, *miles*, and *name* variables, respectively. When running the above Python program, this produces the

following result –

100

1000.0

Zara Ali

**Deleting Python Variables**

You can delete the reference to a number object by using the del statement.

**Example**

Following examples shows how we can delete a variable and if we try to use a deleted variable then Python interpreter will throw an error:

counter = 100

print (counter)

del counter

print (counter)

This will produce the following result:

100

Traceback (most recent call last):

File "main.py", line 7, in <module>

print (counter)

NameError: name 'counter' is not defined

**Getting Type of a Variable**

You can get the data type of a Python variable using the python built-in function type() as follows.

Example: Printing Variables Type

x = "Zara"

y = 10

z = 10.10

print(type(x)

print(type(y))

print(type(z))

This will produce the following result:

<class 'str'>

<class 'int'>

<class 'float'>

**Casting Python Variables**

You can specify the data type of a variable with the help of casting as follows:

**Example**

This example demonstrates case sensitivity of variables.

x = str(10) # x will be '10'

y = int(10) # y will be 10

z = float(10) # z will be 10.0

print( "x =", x )

print( "y =", y )

print( "z =", z )

This will produce the following result:

x = 10

y = 10

z = 10.0

**Case-Sensitivity of Python Variables**

Python variables are case sensitive which means **Age** and **age** are two different variables:

age = 20

Age = 30

print( "age =", age )

print( "Age =", Age )

This will produce the following result:

age = 20

Age = 30

**Python Variables - Multiple Assignment**

Python allows to initialize more than one variable in a single statement. In the following case, three variables have same value.

>>> a=10

>>> b=10

>>> c=10

Instead of separate assignments, you can do it in a single assignment statement as follows −

>>> a=b=c=10

>>> print (a,b,c)

10 10 10

In the following case, we have three variables with different values.

>>> a=10

>>> b=20

>>> c=30

These separate assignment statements can be combined in one. You need to give comma separated variable names on left, and comma separated values on the right of = operator.

>>> a,b,c = 10,20,30

>>> print (a,b,c)

10 20 30

Let's try few examples in script mode: −

a = b = c =100

print (a)

print (b)

print (c)

This produces the following result:

100

100

100

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example

a,b,c = 1,2,"Zara Ali"

print (a)

print (b)

print (c)

This produces the following result:

1

2

Zara Ali

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "Zara Ali" is assigned to the variable c.

**Python Variables - Naming Convention**

Every Python variable should have a unique name like a, b, c. A variable name can be meaningful like color, age, name etc. There are certain rules which should be taken care while naming a Python variable:

* A variable name must start with a letter or the underscore character
* A variable name cannot start with a number or any special character like $, (, \* % etc.
* A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ )
* Python variable names are case-sensitive which means Name and NAME are two different variables in Python.
* Python reserved keywords cannot be used naming the variable.

If the name of variable contains multiple words, we should use these naming patterns −

* Camel case − First letter is a lowercase, but first letter of each subsequent word is in uppercase. For example: kmPerHour, pricePerLitre
* Pascal case − First letter of each word is in uppercase. For example: KmPerHour, PricePerLitre
* Snake case − Use single underscore (\_) character to separate words. For example: km\_per\_hour, price\_per\_litre

**Example**

Following are valid Python variable names:

counter = 100

\_count = 100

name1 = "Zara"

name2 = "Nuha"

Age = 20

zara\_salary = 100000

#NOW PRINTING THE VALUES

print (counter)

print (\_count)

print (name1)

print (name2)

print (Age)

print (zara\_salary)

This will produce the following result:

100

100

Zara

Nuha

20

100000

Example

Following are invalid Python variable names:

1counter = 100

$\_count = 100

zara-salary = 100000

print (1counter)

print ($count)

print (zara-salary)

This will produce the following result:

File "main.py", line 3

1counter = 100

^

SyntaxError: invalid syntax

**Python Local Variables**

Python Local Variables are defined inside a function. We cannot access variable outside the function.

**Example**

Following is an example to show the usage of local variables:

def sum(x,y):

sum = x + y

return sum

print(sum(5, 10))

This will produce the following result –

15

**Python Global Variables**

Any variable created outside a function can be accessed within any function and so they have global scope.

**Example**

Following is an example of global variables –

x = 5

y = 10

def sum():

sum = x + y

return sum

print(sum())

This will produce the following result −

15

**Constants in Python**

Python doesn't have any formally defined constants, however you can indicate a variable to be treated as a constant by using all-caps names with underscores. For example, the name PI\_VALUE indicates that you don't want the variable redefined or changed in any way.

**Python Data Types**

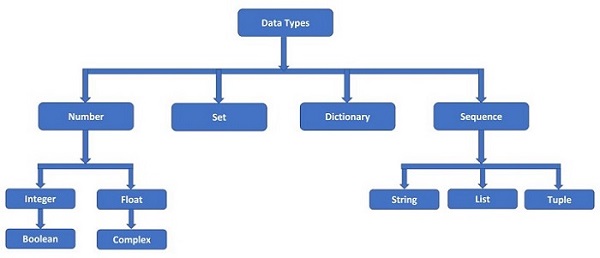
Python data types are actually classes, and the defined variables are their instances or objects. Since Python is dynamically typed, the data type of a variable is determined at runtime based on the assigned value.

In general, the data types are used to define the type of a variable. It represents the type of data we are going to store in a variable and determines what operations can be done on it.

Each programming language has its own classification of data items. With these datatypes, we can store different types of data values.

**Types of Data Types in Python**

* [Numeric Data Types](https://www.tutorialspoint.com/python/python_data_types.htm#numeric_data_types)
  + int
  + float
  + complex
* [String Data Types](https://www.tutorialspoint.com/python/python_data_types.htm#string_data_types)
* [Sequence Data Types](https://www.tutorialspoint.com/python/python_data_types.htm#sequence_data_types)
  + list
  + tuple
  + range
* [Binary Data Types](https://www.tutorialspoint.com/python/python_data_types.htm#binary_data_types)
  + bytes
  + bytearray
  + memoryview
* [Dictionary Data Type](https://www.tutorialspoint.com/python/python_data_types.htm#dictionary_data_type)
* [Set Data Type](https://www.tutorialspoint.com/python/python_data_types.htm#set_data_type)
  + set
  + frozenset
* [Boolean Data Type](https://www.tutorialspoint.com/python/python_data_types.htm#boolean_data_type)
* [None Type](https://www.tutorialspoint.com/python/python_data_types.htm#none_type)



**1. Python Numeric Data Types**

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

var1 = 1 # int data type

var2 = True # bool data type

var3 = 10.023 # float data type

var4 = 10+3j # complex data type

Python supports four different numerical types and each of them have built-in classes in Python library, called **int, bool, float** and **complex** respectively −

* int (signed integers)
* float (floating point real values)
* complex (complex numbers)

A complex number is made up of two parts - real and imaginary. They are separated by '+' or '-' signs. The imaginary part is suffixed by 'j' which is the imaginary number. The square root of -1 (&bsol;sqrt{-1}), is defined as imaginary number. Complex number in Python is represented as x+yj, where x is the real part, and y is the imaginary part. So, 5+6j is a complex number.

>>> type(5+6j)

<class 'complex'>

Example of Numeric Data Types

Following is an example to show the usage of Integer, Float and Complex numbers:

# integer variable.

a=100

print("The type of variable having value", a, " is ", type(a))

# float variable.

c=20.345

print("The type of variable having value", c, " is ", type(c))

# complex variable.

d=10+3j

print("The type of variable having value", d, " is ",

type(d))

**2. Python String Data Type**

[Python string](https://www.tutorialspoint.com/python/python_strings.htm) is a sequence of one or more Unicode characters, enclosed in single, double or triple quotation marks (also called inverted commas). Python strings are immutable which means when you perform an operation on strings, you always produce a new string object of the same type, rather than mutating an existing string.

As long as the same sequence of characters is enclosed, single or double or triple quotes don't matter. Hence, following string representations are equivalent.

>>> 'TutorialsPoint'

'TutorialsPoint'

>>> "TutorialsPoint"

'TutorialsPoint'

>>> '''TutorialsPoint'''

'TutorialsPoint'

A string in Python is an object of **str** class. It can be verified with **type()** function.

>>> type("Welcome To TutorialsPoint")

<class 'str'>

A string is a non-numeric data type. Obviously, we cannot perform arithmetic operations on it. However, operations such as [**slicing**](https://www.tutorialspoint.com/python/python_slicing_strings.htm) and [**concatenation**](https://www.tutorialspoint.com/python/python_string_concatenation.htm) can be done. Python's str class defines a number of useful methods for string processing. Subsets of strings can be taken using the slice operator ([ ] and [:] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator in Python.

**Example of String Data Type**

str = 'Hello World!'

print (str) # Prints complete string

print (str[0]) # Prints first character of the string

print (str[2:5]) # Prints characters starting from 3rd to 5th

print (str[2:]) # Prints string starting from 3rd character

print (str \* 2) # Prints string two times

print (str + "TEST") # Prints concatenated string

This will produce the following result –

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

**3. Python Sequence Data Types**

Sequence is a collection data type. It is an ordered collection of items. Items in the sequence have a positional index starting with 0. It is conceptually similar to an array in C or C++. There are following three sequence data types defined in Python.

* List Data Type
* Tuple Data Type
* Range Data Type

Python sequences are bounded and iterable - Whenever we say an iterable in Python, it means a sequence data type (for example, a list).

**(a) Python List Data Type**

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

>>> [2023, "Python", 3.11, 5+6j, 1.23E-4]

A list in Python is an object of **list** class. We can check it with type() function.

>>> type([2023, "Python", 3.11, 5+6j, 1.23E-4])

<class 'list'>

As mentioned, an item in the list may be of any data type. It means that a list object can also be an item in another list. In that case, it becomes a nested list.

>>> [['One', 'Two', 'Three'], [1,2,3], [1.0, 2.0, 3.0]]

A list can have items which are simple numbers, strings, tuple, dictionary, set or object of user defined class also.

The values stored in a Python list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (\*) is the repetition operator.

**Example of List Data Type**

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']

print (list) # Prints complete list

print (list[0]) # Prints first element of the list

print (list[1:3]) # Prints elements starting from 2nd till 3rd

print (list[2:]) # Prints elements starting from 3rd element

print (tinylist \* 2) # Prints list two times

print (list + tinylist) # Prints concatenated lists

This produce the following result –

['abcd', 786, 2.23, 'john', 70.2]

abcd

[786, 2.23]

[2.23, 'john', 70.2]

[123, 'john', 123, 'john']

['abcd', 786, 2.23, 'john', 70.2, 123, 'john']

**(b) Python Tuple Data Type**

[Python tuple](https://www.tutorialspoint.com/python/python_tuples.htm) is another sequence data type that is similar to a list. A Python tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses (...).

A tuple is also a sequence, hence each item in the tuple has an index referring to its position in the collection. The index starts from 0.

>>> (2023, "Python", 3.11, 5+6j, 1.23E-4)

In Python, a tuple is an object of **tuple** class. We can check it with the type() function.

>>> type((2023, "Python", 3.11, 5+6j, 1.23E-4))

<class 'tuple'>

As in case of a list, an item in the tuple may also be a list, a tuple itself or an object of any other Python class.

>>> (['One', 'Two', 'Three'], 1,2.0,3, (1.0, 2.0, 3.0))

To form a tuple, use of parentheses is optional. Data items separated by comma without any enclosing symbols are treated as a tuple by default.

>>> 2023, "Python", 3.11, 5+6j, 1.23E-4

(2023, 'Python', 3.11, (5+6j), 0.000123)

**Example of Tuple data Type**

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

tinytuple = (123, 'john')

print (tuple) # Prints the complete tuple

print (tuple[0]) # Prints first element of the tuple

print (tuple[1:3]) # Prints elements of the tuple starting from 2nd till 3rd

print (tuple[2:]) # Prints elements of the tuple starting from 3rd element

print (tinytuple \* 2) # Prints the contents of the tuple twice

print (tuple + tinytuple) # Prints concatenated tuples

This produce the following result –

('abcd', 786, 2.23, 'john', 70.2)

abcd

(786, 2.23)

(2.23, 'john', 70.2)

(123, 'john', 123, 'john')

('abcd', 786, 2.23, 'john', 70.2, 123, 'john')

The main differences between lists and tuples are: Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed i.e. lists are mutable, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated (immutable). Tuples can be thought of as **read-only** lists.

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tuple[2] = 1000 # Invalid syntax with tuple

list[2] = 1000 # Valid syntax with list

**(c) Python Range Data Type**

A Python range is an immutable sequence of numbers which is typically used to iterate through a specific number of items.

It is represented by the **Range** class. The constructor of this class accepts a sequence of numbers starting from 0 and increments to 1 until it reaches a specified number. Following is the syntax of the function –

range(start, stop, step)

Here is the description of the parameters used −

* **start**: Integer number to specify starting position, (Its optional, Default: 0)
* **stop**: Integer number to specify ending position (It's mandatory)
* **step**: Integer number to specify increment, (Its optional, Default: 1)

**Example of Range Data Type**

Following is a program which uses for loop to print number from 0 to 4 –

for i in range(5):

print(i)

This produce the following result –

0

1

2

3

4

Now let's modify above program to print the number starting from 2 instead of 0 –

for i in range(2, 5):

print(i)

This produce the following result –

2

3

4

Again, let's modify the program to print the number starting from 1 but with an increment of 2 instead of 1:

for i in range(1, 5, 2):

print(i)

This produce the following result –

1

3

**4. Python Binary Data Types**

A binary data type in Python is a way to represent data as a series of binary digits, which are 0's and 1's. It is like a special language computers understand to store and process information efficiently.

This type of data is commonly used when dealing with things like files, images, or anything that can be represented using just two possible values. So, instead of using regular numbers or letters, binary sequence data types use a combination of 0s and 1s to represent information.

Python provides three different ways to represent binary data. They are as follows −

* bytes
* bytearray
* memoryview

Let us discuss each of these data types individually −

1. **Python Bytes Data Type**

The byte data type in Python represents a sequence of bytes. Each byte is an integer value between 0 and 255. It is commonly used to store binary data, such as images, files, or network packets.

We can create bytes in Python using the built-in [**bytes()** function](https://www.tutorialspoint.com/python/python_bytes_function.htm) or by prefixing a sequence of numbers with **b**.

**Example of Bytes Data Type**

In the following example, we are using the built-in bytes() function to explicitly specify a sequence of numbers representing ASCII values –

# Using bytes() function to create bytes

b1 = bytes([65, 66, 67, 68, 69])

print(b1)

The result obtained is as follows –

b'ABCDE'

In here, we are using the "b" prefix before a string to automatically create a bytes object –

# Using prefix 'b' to create bytes

b2 = b'Hello'

print(b2)

Following is the output of the above code –

b'Hello'

**(b) Python Bytearray Data Type**

The bytearray data type in Python is quite similar to the bytes data type, but with one key difference: it is mutable, meaning you can modify the values stored in it after it is created.

You can create a bytearray using various methods, including by passing an iterable of integers representing byte values, by encoding a string, or by converting an existing bytes or bytearray object. For this, we use [**bytearray()** function](https://www.tutorialspoint.com/python/python_bytearray_function.htm).

**Example of Bytearray Data Type**

In the example below, we are creating a bytearray by passing an iterable of integers representing byte values –

# Creating a bytearray from an iterable of integers

value = bytearray([72, 101, 108, 108, 111])

print(value)

The output obtained is as shown below –

bytearray(b'Hello')

Now, we are creating a bytearray by encoding a string using a "UTF-8" encoding –

# Creating a bytearray by encoding a string

val = bytearray("Hello", 'utf-8')

print(val)

The result produced is as follows –

bytearray(b'Hello')

**(c) Python Memoryview Data Type**

In Python, a memoryview is a built-in object that provides a view into the memory of the original object, generally objects that support the buffer protocol, such as byte arrays (bytearray) and bytes (bytes). It allows you to access the underlying data of the original object without copying it, providing efficient memory access for large datasets.

You can create a memoryview using various methods. These methods include using the [memoryview()](https://www.tutorialspoint.com/python/python_memoryview_function.htm) constructor, slicing bytes or bytearray objects, extracting from array objects, or using built-in functions like [open()](https://www.tutorialspoint.com/python/python_open_function.htm) when reading from files.

**Example of Memoryview Data Type**

In the given example, we are creating a memoryview object directly by passing a supported object to the memoryview() constructor. The supported objects generally include byte arrays (bytearray), bytes (bytes), and other objects that support the buffer protocol –

data = bytearray(b'Hello, world!')

view = memoryview(data)

print(view)

Following is the output of the above code –

<memory at 0x00000186FFAA3580>

If you have an array object, you can create a memoryview using the buffer interface as shown below –

import array

arr = array.array('i', [1, 2, 3, 4, 5])

view = memoryview(arr)

print(view)

The output obtained is as shown below –

<memory at 0x0000017963CD3580>

You can also create a memoryview by slicing a bytes or bytearray object –

data = b'Hello, world!'

# Creating a view of the last part of the data

view = memoryview(data[7:])

print(view)

The result obtained is as follows –

<memory at 0x00000200D9AA3580>

**5. Python Dictionary Data Type**

[Python dictionaries](https://www.tutorialspoint.com/python/python_dictionaries.htm) are kind of hash table type. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Python dictionary is like associative arrays or hashes found in Perl and consist of **key:value** pairs. The pairs are separated by comma and put inside curly brackets {}. To establish mapping between key and value, the semicolon':' symbol is put between the two.

>>> {1:'one', 2:'two', 3:'three'}

In Python, dictionary is an object of the built-in **dict** class. We can check it with the type() function.

>>> type({1:'one', 2:'two', 3:'three'})

<class 'dict'>

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

**Example of Dictionary Data Type**

dict = {}

dict['one'] = "This is one"

dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print(dict['one']) # Prints value for 'one' key

print(dict[2]) # Prints value for 2 key

print(tinydict) # Prints complete dictionary

print (tinydict.keys()) # Prints all the keys

print (tinydict.values()) # Prints all the values

This produce the following result –

This is one

This is two

{'dept': 'sales', 'code': 6734, 'name': 'john'}

['dept', 'code', 'name']

['sales', 6734, 'john']

Python's dictionary is not a sequence. It is a collection of items but each item (key:value pair) is not identified by positional index as in string, list or tuple. Hence, slicing operation cannot be done on a dictionary. Dictionary is a mutable object, so it is possible to perform add, modify or delete actions with corresponding functionality defined in dict class. These operations will be explained in a subsequent chapter.

**6. Python Set Data Type**

[Set](https://www.tutorialspoint.com/python/python_sets.htm) is a Python implementation of set as defined in Mathematics. A set in Python is a collection, but is not an indexed or ordered collection as string, list or tuple. An object cannot appear more than once in a set, whereas in List and Tuple, same object can appear more than once.

Comma separated items in a set are put inside curly brackets or braces {}. Items in the set collection can be of different data types.

>>> {2023, "Python", 3.11, 5+6j, 1.23E-4}

{(5+6j), 3.11, 0.000123, 'Python', 2023}

Note that items in the set collection may not follow the same order in which they are entered. The position of items is optimized by Python to perform operations over set as defined in mathematics.

Python's Set is an object of built-in **set** class, as can be checked with the type() function.

>>> type({2023, "Python", 3.11, 5+6j, 1.23E-4})

<class 'set'>

A set can store only immutable objects such as number (int, float, complex or bool), string or tuple. If you try to put a list or a dictionary in the set collection, Python raises a TypeError.

>>> {['One', 'Two', 'Three'], 1,2,3, (1.0, 2.0, 3.0)}

Traceback (most recent call last):

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

TypeError: unhashable type: 'list'

**Hashing** is a mechanism in computer science which enables quicker searching of objects in computer's memory. **Only immutable objects are hashable**.

Even if a set doesn't allow mutable items, the set itself is mutable. Hence, add/delete/update operations are permitted on a set object, using the methods in built-in set class. Python also has a set of operators to perform set manipulation. The methods and operators are explained in latter chapters

**Example of Set**

set1 = {123, 452, 5, 6}

set2 = {'Java', 'Python', 'JavaScript'}

print(set1)

print(set2)

This will generate the following output –

{123, 452, 5, 6}

{'Python', 'JavaScript', 'Java'}

**7. Python Boolean Data Type**

Python **boolean** type is one of built-in data types which represents one of the two values either **True** or **False**. Python **bool()** function allows you to evaluate the value of any expression and returns either True or False based on the expression.

A Boolean number has only two possible values, as represented by the keywords, **True** and **False**. They correspond to integer 1 and 0 respectively.

>>> type (True)

<class 'bool'> >>>

type(False)

<class 'bool'>

**Example of Boolean Data Type**

Following is a program which prints the value of boolean variables a and b –

a = True\

# display the value of a

print(a)

# display the data type of a

print(type(a))

This will produce the following result –

true

<class 'bool'>

Following is another program which evaluates the expressions and prints the return values –

# Returns false as a is not equal to b

a = 2

b = 4

print(bool(a==b))

# Following also prints the same

print(a==b)

# Returns False as a is None

a = None

print(bool(a))

# Returns false as a is an empty sequence

a = ()

print(bool(a))

# Returns false as a is 0

a = 0.0

print(bool(a))

# Returns false as a is 10

a = 10

print(bool(a))

This produce the following result –False

False

False

False

False

True

**8. Python None Type**

Python's none type is represented by the "**nonetype**." It is an object of its own data type. The **nonetype** represents the null type of values or absence of a value.

**Example of None Type**

In the following example, we are assigning **None** to a variable **x** and printing its type, which will be **nonetyoe** –

# Declaring a variable

# And, assigning a Null value (None)

x = None

# Printing its value and type

print("x = ", x)

print("type of x = ", type(x))

This produce the following result –

x = None

type of x = <class 'NoneType'>

**Getting Data Type**

To get the data types in Python, you can use the **type()** function. The [**type()**](https://www.tutorialspoint.com/python/python_type_function.htm) is a built-in function that returns the class of the given object.

Example

In the following example, we are getting the type of the values and variables −  
  
# Getting type of values

print(type(123))

print(type(9.99))

# Getting type of variables

a = 10

b = 2.12

c = "Hello"

d = (10, 20, 30)

e = [10, 20, 30]

print(type(a))

print(type(b))

print(type(c))

print(type(d))

print(type(e))

This produce the following result –

<class 'int'>

<class 'float'>

<class 'int'>

<class 'float'>

<class 'str'>

<class 'tuple'>

<class 'list'>

**Setting Data Type**

In Python, during declaring a variable or an object, you don't need to set the data types. Data type is set automatically based on the assigned value.

Example

The following example, demonstrating how a variable's data type is set based on the given value −

# Declaring a variable

# And, assigning an integer value

x = 10

# Printing its value and type

print("x = ", x)

print("type of x = ", type(x))

# Now, assigning string value to

# the same variable

x = "Hello World!"

# Printing its value and type

print("x = ", x)

print("type of x = ", type(x))

This produce the following result −

x = 10

type of x = <class 'int'>

x = Hello World!

type of x = <class 'str'>

**Primitive and Non-primitive Data Types**

The above-explained data types can also be categorized as primitive and non-primitive.

**1. Primitive Types**

The primitive data types are the fundamental data types that are used to create complex data types (sometimes called complex data structures). There are mainly four primitive data types, which are −

* Integers
* Floats
* Booleans, and
* Strings

**2. Non-primitive Types**

The non-primitive data types store values or collections of values. There are mainly four types of non-primitive types, which are −

* Lists
* Tuples
* Dictionaries, and
* Sets

**Python Data Type Conversion**

Sometimes, you may need to perform conversions between the built-in data types. To convert data between different Python data types, you simply use the type name as a function.

Example:

Following is an example which converts different values to integer, floating point and string values respectively −

print("Conversion to integer data type")

a = int(1) # a will be 1

b = int(2.2) # b will be 2

c = int("3.3") # c will be 3

print (a)

print (b)

print (c)

print("Conversion to floating point number")

a = float(1) # a will be 1.0

b = float(2.2) # b will be 2.2

c = float("3.3") # c will be 3.3

print (a)

print (b)

print (c)

print("Conversion to string")

a = str(1) # a will be "1"

b = str(2.2) # b will be "2.2"

c = str("3.3") # c will be "3.3"

print (a)

print (b)

print (c)

This produce the following result –

Conversion to integer data type

1

2

3

Conversion to floating point number

1.0

2.2

3.3

Conversion to string

1

2.2

3.3

**Data Type Conversion Functions**

There are several [built-in functions](https://www.tutorialspoint.com/python/python_built_in_functions.htm) to perform conversion from one data type to another. These functions return a new object representing the converted value.

|  |  |
| --- | --- |
| **Sr. No** | **Function & Description** |
| 1 | [**Python int() function**](https://www.tutorialspoint.com/python/python-int-function.htm)  Converts x to an integer. base specifies the base if x is a string. |
| 2 | [**Python long() function**](https://www.tutorialspoint.com/python/python-long-function.htm)  Converts x to a long integer. base specifies the base if x is a string. *This function has been deprecated.* |
| 3 | [**Python float() function**](https://www.tutorialspoint.com/python/python-float-function.htm)  Converts x to a floating-point number. |
| 4 | [**Python complex() function**](https://www.tutorialspoint.com/python/python-complex-function.htm)  Creates a complex number. |
| 5 | [**Python str() function**](https://www.tutorialspoint.com/python/python-str-function.htm)  Converts object x to a string representation. |
| 6 | [**Python repr() function**](https://www.tutorialspoint.com/python/python-repr-function.htm)  Converts object x to an expression string. |
| 7 | [**Python eval() function**](https://www.tutorialspoint.com/python/python-eval-function.htm)  Evaluates a string and returns an object. |
| 8 | [**Python tuple() function**](https://www.tutorialspoint.com/python/python-tuple-function.htm)  Converts s to a tuple. |
| 9 | [**Python list() function**](https://www.tutorialspoint.com/python/python-list-function.htm)  Converts s to a list. |
| 10 | [**Python set() function**](https://www.tutorialspoint.com/python/python-set-function.htm)  Converts s to a set. |
| 11 | [**Python dict() function**](https://www.tutorialspoint.com/python/python-dict-function.htm)  Creates a dictionary. d must be a sequence of (key,value) tuples. |
| 12 | [**Python frozenset() function**](https://www.tutorialspoint.com/python/python-frozenset-function.htm)  Converts s to a frozen set. |
| 13 | [**Python chr() function**](https://www.tutorialspoint.com/python/python-chr-function.htm)  Converts an integer to a character. |
| 14 | [**Python unichr() function**](https://www.tutorialspoint.com/python/python-unichr-function.htm)  Converts an integer to a Unicode character. |
| 15 | [**Python ord() function**](https://www.tutorialspoint.com/python/python-ord-function.htm)  Converts a single character to its integer value. |
| 16 | [**Python hex() function**](https://www.tutorialspoint.com/python/python-hex-function.htm)  Converts an integer to a hexadecimal string. |
| 17 | [**Python oct() function**](https://www.tutorialspoint.com/python/python-oct-function.htm)  Converts an integer to an octal string. |

**Python Type Casting**

From a programming point of view, a type casting refers to converting an object of one type into another. Here, we shall learn about type casting in Python Programming.

*{ Python Type Casting is a process in which we convert a literal of one data type to another data type. Python supports two types of casting −****implicit****and****explicit****. }*

In Python there are different data types, such as numbers, sequences, mappings etc. There may be a situation where, you have the available data of one type but you want to use it in another form. For example, the user has input a string but you want to use it as a number. Python's type casting mechanism let you do that.

**Python Implicit Casting**

When any language compiler/interpreter automatically converts object of one type into other, it is called automatic or **implicit casting**. Python is a strongly typed language. It doesn't allow automatic type conversion between unrelated data types. For example, a string cannot be converted to any number type. However, an integer can be cast into a float. Other languages such as JavaScript is a weakly typed language, where an integer is coerced into a string for concatenation.

Note that memory requirement of each data type is different. For example, an **integer** object in Python occupies 4 bytes of memory, while a **float** object needs 8 bytes because of its fractional part. Hence, Python interpreter doesn't automatically convert a **float** to **int**, because it will result in loss of data. On the other hand, **int** can be easily converted into **float** by setting its fractional part to 0.

Implicit **int** to **float** casting takes place when any arithmetic operation on **int** and **float** operands is done.

Consider we have an , **int** and one **float** variable

<<< a=10 # int object

<<< b=10.5 # float object

To perform their addition, 10 − the integer object is upgraded to 10.0. It is a float, but equivalent to its earlier numeric value. Now we can perform addition of two floats.

<<< c=a+b

<<< print (c) 20.5

In implicit type casting, a Python object with lesser byte size is upgraded to match the bigger byte size of other object in the operation. For example, a Boolean object is first upgraded to int and then to float, before the addition with a floating point object. In the following example, we try to add a Boolean object in a float, please note that True is equal to 1, and False is equal to 0.

a=True;

b=10.5;

c=a+b;

print (c);

This will produce the following result:

11.5

**Python Explicit Casting**

Although automatic or implicit casting is limited to **int** to **float** conversion, you can use Python's built-in functions int(), float() and str() to perform the explicit conversions such as string to integer.

**Python int() Function**

Python's built-in **int()** function converts an integer literal to an integer object, a float to integer, and a string to integer if the string itself has a valid integer literal representation. Using **int()** with an int object as argument is equivalent to declaring an **int** object directly.

<<< a = int(10)

<<< a 10

Is same as –

<<< a = 10

<<< a 10

<<< type(a)

<class 'int>

If the argument to **int()** function is a float object or floating point expression, it returns an int object. For example −

<<< a = int(10.5) #converts a float object to int

<<< a

10

<<< a = int(2\*3.14) #expression results float, is converted to int

<<< a

6 <<<

type(a)

<class 'int'>

The **int()** function also returns integer 1 if a Boolean object is given as argument.

><<< a=int(True)

<<< a 1

<<< type(a)

<class 'int'

**String to Integer**

The **int()** function returns an integer from a string object, only if it contains a valid integer representation.

<<< a = int("100")

<<< a

100

<<< type(a)

<class 'int'>

<<< a = ("10"+"01")

<<< a = int("10"+"01")

<<< a

1001

<<< type(a)

<class 'int'>

However, if the string contains a non-integer representation, Python raises ValueError.

<<< a = int("10.5")

Traceback (most recent call last):

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

ValueError: invalid literal for int() with base 10: '10.5'

<<< a = int("Hello World")

Traceback (most recent call last):

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

ValueError: invalid literal for int() with base 10: 'Hello World'

The **int()** function also returns integer from binary, octal and hexa-decimal string. For this, the function needs a base parameter which must be 2, 8 or 16 respectively. The string should have a valid binary/octal/Hexa-decimal representation.

**Binary String to Integer**

The string should be made up of 1 and 0 only, and the base should be 2.

<<< a = int("110011", 2)

<<< a

51

The Decimal equivalent of binary number 110011 is 51.

**Octal String to Integer**

The string should only contain 0 to 7 digits, and the base should be 8.

<<< a = int("20", 8)

<<< a

16

The Decimal equivalent of octal 20 is 16.

**Hexa-Decimal String to Integer**

The string should contain only the Hexadecimal symbols i.e., 0-9 and A, B, C, D, E or F. Base should be 16.

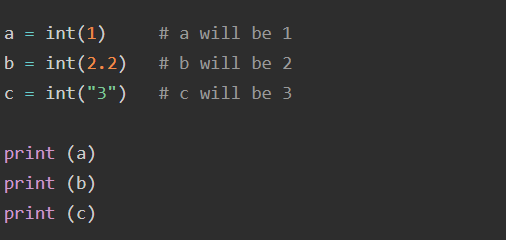
<<< a = int("2A9", 16)

<<< a

681

Decimal equivalent of Hexadecimal 2A9 is 681. You can easily verify these conversions with calculator app in Windows, Ubuntu or Smartphones.

Following is an example to convert number, float and string into integer data type:



This produce the following result −

1

2

3

**Python float() Function**

The **float()** is a built-in function in Python. It returns a float object if the argument is a float literal, integer or a string with valid floating point representation.

Using float() with an float object as argument is equivalent to declaring a float object directly

<<< a = float(9.99)

<<< a

9.99

<<< type(a)

<class 'float'>

is same as −

<<< a = 9.99

<<< a

9.99

<<< type(a)

<class 'float'>

If the argument to **float()** function is an integer, the returned value is a floating point with fractional part set to 0.

<<< a = float(100)

<<< a

100.0

<<< type(a)

<class 'float'>

The **float()** function returns float object from a string, if the string contains a valid floating point number, otherwise ValueError is raised.

<<< a = float("9.99")

<<< a

9.99

<<< type(a)

<class 'float'>

<<< a = float("1,234.50")

Traceback (most recent call last):

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

ValueError: could not convert string to float: '1,234.50'

The reason of ValueError here is the presence of comma in the string. For the purpose of string to float conversion, the sceientific notation of floating point is also considered valid.

<<< a = float("1.00E4")

<<< a

10000.0

<<< type(a)

<class 'float'>

<<< a = float("1.00E-4")

<<< a

0.0001

<<< type(a)

<class 'float'>

Following is an example to convert number, float and string into float data type:

a = float(1) # a will be 1.0

b = float(2.2) # b will be 2.2

c = float("3.3") # c will be 3.3

print (a)

print (b)

print (c)

This produce the following result −

1.0

2.2

3.3

**Python str() Function**

We saw how a Python obtains integer or float number from corresponding string representation. The **str()** function works the opposite. It surrounds an integer or a float object with quotes (') to return a str object. The **str()** function returns the string representation of any Python object. In this section, we shall see different examples of **str()** function in Python.

The str() function has three parameters. First required parameter (or argument) is the object whose string representation we want. Other two operators, encoding and errors, are optional.

We shall execute str() function in Python console to easily verify that the returned object is a string, with the enclosing quotation marks (').

**Integer to string**

You can convert any integer number into a string as follows:

<<< a = str(10)

<<< a

'10'

<<< type(a)

<class 'str'>

**Float to String**

str() function converts floating point objects with both the notations of floating point, standard notation with a decimal point separating integer and fractional part, and the scientific notation to string object.

<<< a=str(11.10)

<<< a

'11.1'

<<< type(a)

<class 'str'>

<<< a = str(2/5)

<<< a

'0.4'

<<< type(a)

<class 'str'>

In the second case, a division expression is given as argument to str() function. Note that the expression is evaluated first and then result is converted to string.

Floating points in scientific notations using E or e and with positive or negative power are converted to string with str() function.

<<< a=str(10E4)

<<< a

'100000.0'

<<< type(a)

<class 'str'>

<<< a=str(1.23e-4)

<<< a

'0.000123'

<<< type(a)

<class 'str'>

When Boolean constant is entered as argument, it is surrounded by (') so that True becomes 'True'. List and Tuple objects can also be given argument to str() function. The resultant string is the list/tuple surrounded by (').

<<< a=str('True')

<<< a

'True'

<<< a=str([1,2,3])

<<< a

'[1, 2, 3]'

<<< a=str((1,2,3))

<<< a

'(1, 2, 3)'

<<< a=str({1:100, 2:200, 3:300})

<<< a

'{1: 100, 2: 200, 3: 300}'

Following is an example to convert number, float and string into string data type:

a = str(1) # a will be "1"

b = str(2.2) # b will be "2.2"

c = str("3.3") # c will be "3.3"

print (a)

print (b)

print (c)

This produce the following result −

1

2.2

3.3

**Conversion of Sequence Types**

List, Tuple and String are Python's sequence types. They are ordered or indexed collection of items.

A string and tuple can be converted into a list object by using the **list()** function. Similarly, the **tuple()** function converts a string or list to a tuple.

We shall take an object each of these three sequence types and study their inter-conversion.

<<< a=[1,2,3,4,5]   # List Object

<<< b=(1,2,3,4,5)   # Tupple Object

<<< c="Hello"       # String Object

### list() separates each character in the string and builds the list

<<< obj=list(c)

<<< obj

['H', 'e', 'l', 'l', 'o']

### The parentheses of tuple are replaced by square brackets

<<< obj=list(b)

<<< obj

[1, 2, 3, 4, 5]

### tuple() separates each character from string and builds a tuple of characters

<<< obj=tuple(c)

<<< obj

('H', 'e', 'l', 'l', 'o')

### square brackets of list are replaced by parentheses.

<<< obj=tuple(a)

<<< obj

(1, 2, 3, 4, 5)

### str() function puts the list and tuple inside the quote symbols.

<<< obj=str(a)

<<< obj

'[1, 2, 3, 4, 5]'

<<< obj=str(b)

<<< obj

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

Thus Python's explicit type casting feature allows conversion of one data type to other with the help of its built-in functions.

**Data Type Conversion Functions**

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

|  |  |
| --- | --- |
| **Sr.No** | **Function & Description** |
| 1 | [**Python int() function**](https://www.tutorialspoint.com/python/python-int-function.htm)  Converts x to an integer. base specifies the base if x is a string. |
| 2 | [**Python long() function**](https://www.tutorialspoint.com/python/python-long-function.htm)  Converts x to a long integer. base specifies the base if x is a string. |
| 3 | [**Python float() function**](https://www.tutorialspoint.com/python/python-float-function.htm)  Converts x to a floating-point number. |
| 4 | [**Python complex() function**](https://www.tutorialspoint.com/python/python-complex-function.htm)  Creates a complex number. |
| 5 | [**Python str() function**](https://www.tutorialspoint.com/python/python-str-function.htm)  Converts object x to a string representation. |
| 6 | [**Python repr() function**](https://www.tutorialspoint.com/python/python-repr-function.htm)  Converts object x to an expression string. |
| 7 | [**Python eval() function**](https://www.tutorialspoint.com/python/python-eval-function.htm)  Evaluates a string and returns an object. |
| 8 | [**Python tuple() function**](https://www.tutorialspoint.com/python/python-tuple-function.htm)  Converts s to a tuple. |
| 9 | [**Python list() function**](https://www.tutorialspoint.com/python/python-list-function.htm)  Converts s to a list. |
| 10 | [**Python set() function**](https://www.tutorialspoint.com/python/python-set-function.htm)  Converts s to a set. |
| 11 | [**Python dict() function**](https://www.tutorialspoint.com/python/python-dict-function.htm)  Creates a dictionary. d must be a sequence of (key,value) tuples. |
| 12 | [**Python frozenset() function**](https://www.tutorialspoint.com/python/python-frozenset-function.htm)  Converts s to a frozen set. |
| 13 | [**Python chr() function**](https://www.tutorialspoint.com/python/python-chr-function.htm)  Converts an integer to a character. |
| 14 | [**Python unichr() function**](https://www.tutorialspoint.com/python/python-unichr-function.htm)  Converts an integer to a Unicode character. |
| 15 | [**Python ord() function**](https://www.tutorialspoint.com/python/python-ord-function.htm)  Converts a single character to its integer value. |
| 16 | [**Python hex() function**](https://www.tutorialspoint.com/python/python-hex-function.htm)  Converts an integer to a hexadecimal string. |
| 17 | [**Python oct() function**](https://www.tutorialspoint.com/python/python-oct-function.htm)  Converts an integer to an octal string. |

**Python Operators**

**Python operators** are special symbols used to perform specific operations on one or more operands. The [variables](https://www.tutorialspoint.com/python/python_variables.htm), values, or expressions can be used as operands. For example, Python's addition operator (**+**) is used to perform addition operations on two variables, values, or expressions.

The following are some of the terms related to **Python operators**:

* **Unary operators**: Python operators that require one operand to perform a specific operation are known as unary operators.
* **Binary operators**: Python operators that require two operands to perform a specific operation are known as binary operators.
* **Operands**: Variables, values, or expressions that are used with the operator to perform a specific operation.

Types of Python Operators

Python operators are categorized in the following categories −

* [Arithmetic Operators](https://www.tutorialspoint.com/python/python_operators.htm#arithmetic-operators)
* [Comparison (Relational) Operators](https://www.tutorialspoint.com/python/python_operators.htm#comparison-operators)
* [Assignment Operators](https://www.tutorialspoint.com/python/python_operators.htm#assignment-operators)
* [Logical Operators](https://www.tutorialspoint.com/python/python_operators.htm#logical-operators)
* [Bitwise Operators](https://www.tutorialspoint.com/python/python_operators.htm#bitwise-operators)
* [Membership Operators](https://www.tutorialspoint.com/python/python_operators.htm#membership-operators)
* [Identity Operators](https://www.tutorialspoint.com/python/python_operators.htm#identity-operators)

Let us have a look at all the operators one by one.

**Python Arithmetic Operators**

[Python Arithmetic operators](https://www.tutorialspoint.com/python/python_arithmetic_operators.htm) are used to perform basic mathematical operations such as addition, subtraction, multiplication, etc.

The following table contains all arithmetic operators with their symbols, names, and examples (assume that the values of **a** and **b** are 10 and 20, respectively) −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| + | Addition | a + b = 30 |
| - | Subtraction | a b = -10 |
| \* | Multiplication | a \* b = 200 |
| / | Division | b / a = 2 |
| % | Modulus | b % a = 0 |
| \*\* | Exponent | a\*\*b =10\*\*20 |
| // | Floor Division | 9//2 = 4 |

Example of Python Arithmetic Operators

a = 21

b = 10

c = 0

c = a + b

print ("a: {} b: {} a+b: {}".format(a,b,c))

c = a - b

print ("a: {} b: {} a-b: {}".format(a,b,c) )

c = a \* b

print ("a: {} b: {} a\*b: {}".format(a,b,c))

c = a / b

print ("a: {} b: {} a/b: {}".format(a,b,c))

c = a % b

print ("a: {} b: {} a%b: {}".format(a,b,c))

a = 2

b = 3

c = a\*\*b

print ("a: {} b: {} a\*\*b: {}".format(a,b,c))

a = 10

b = 5

c = a//b

print ("a: {} b: {} a//b: {}".format(a,b,c))

**Output**

Line 1 - a is not equal to b

Line 2 - a is not equal to b

Line 3 - a is not less than b

Line 4 - a is greater than b

Line 5 - a is either less than or equal to  b

Line 6 - b is either greater than  or equal to b

**Python Assignment Operators**

[Python Assignment operators](https://www.tutorialspoint.com/python/python_assignment_operators.htm) are used to assign values to variables. Following is a table which shows all Python assignment operators.

The following table contains all assignment operators with their symbols, names, and examples −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Example** | **Same As** |
| = | a = 10 | a = 10 |
| += | a += 30 | a = a + 30 |
| -= | a -= 15 | a = a - 15 |
| \*= | a \*= 10 | a = a \* 10 |
| /= | a /= 5 | a = a / 5 |
| %= | a %= 5 | a = a % 5 |
| \*\*= | a \*\*= 4 | a = a \*\* 4 |
| //= | a //= 5 | a = a // 5 |
| &= | a &= 5 | a = a & 5 |
| |= | a |= 5 | a = a | 5 |
| ^= | a ^= 5 | a = a ^ 5 |
| >>= | a >>= 5 | a = a >> 5 |
| <<= | a <<= 5 | a = a << 5 |

Example of Python Assignment Operators

a = 21

b = 10

c = 0

print ("a: {} b: {} c : {}".format(a,b,c))

c = a + b

print ("a: {}  c = a + b: {}".format(a,c))

c += a

print ("a: {} c += a: {}".format(a,c))

c \*= a

print ("a: {} c \*= a: {}".format(a,c))

c /= a

print ("a: {} c /= a : {}".format(a,c))

c  = 2

print ("a: {} b: {} c : {}".format(a,b,c))

c %= a

print ("a: {} c %= a: {}".format(a,c))

c \*\*= a

print ("a: {} c \*\*= a: {}".format(a,c))

c //= a

print ("a: {} c //= a: {}".format(a,c))

**Output**

a: 21 b: 10 c : 0

a: 21  c = a + b: 31

a: 21 c += a: 52

a: 21 c \*= a: 1092

a: 21 c /= a : 52.0

a: 21 b: 10 c : 2

a: 21 c %= a: 2

a: 21 c \*\*= a: 2097152

a: 21 c //= a: 99864

**Python Bitwise Operators**

[Python Bitwise operator](https://www.tutorialspoint.com/python/python_bitwise_operators.htm) works on bits and performs bit by bit operation. These operators are used to compare binary numbers.

The following table contains all bitwise operators with their symbols, names, and examples –

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| & | AND | a & b |
| | | OR | a | b |
| ^ | XOR | a ^ b |
| ~ | NOT | ~a |
| << | Zero fill left shift | a << 3 |
| >> | Signed right shift | a >> 3 |

a = 20

b = 10

print ('a=',a,':',bin(a),'b=',b,':',bin(b))

c = 0

c = a & b;

print ("result of AND is ", c,':',bin(c))

c = a | b;

print ("result of OR is ", c,':',bin(c))

c = a ^ b;

print ("result of EXOR is ", c,':',bin(c))

c = ~a;

print ("result of COMPLEMENT is ", c,':',bin(c))

c = a << 2;

print ("result of LEFT SHIFT is ", c,':',bin(c))

c = a >> 2;

print ("result of RIGHT SHIFT is ", c,':',bin(c))

**Output**

a= 20 : 0b10100 b= 10 : 0b1010

result of AND is  0 : 0b0

result of OR is  30 : 0b11110

result of EXOR is  30 : 0b11110

result of COMPLEMENT is  -21 : -0b10101

result of LEFT SHIFT is  80 : 0b1010000

result of RIGHT SHIFT is  5 : 0b101

**Python Logical Operators**

[Python logical operators](https://www.tutorialspoint.com/python/python_logical_operators.htm) are used to combile two or more conditions and check the final result. There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

The following table contains all logical operators with their symbols, names, and examples −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| and | AND | a and b |
| or | OR | a or b |
| not | NOT | not(a) |

Example of Python Logical Operators

var = 5

print(var > 3 and var < 10)

print(var > 3 or var < 4)

print(not (var a> 3 and var < 10))

**Output**

True

True

False

Python Membership Operators

Python's [membership operators](https://www.tutorialspoint.com/python/python_membership_operators.htm) test for membership in a sequence, such as strings, lists, or tuples.

There are two membership operators as explained below −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| in | Returns True if it finds a variable in the specified sequence, false otherwise. | a in b |
| not in | returns True if it does not finds a variable in the specified sequence and false otherwise. | a not in b |

a = 10

b = 20

list = [1, 2, 3, 4, 5 ]

print ("a:", a, "b:", b, "list:", list)

if ( a in list ):

   print ("a is present in the given list")

else:

   print ("a is not present in the given list")

if ( b not in list ):

   print ("b is not present in the given list")

else:

   print ("b is present in the given list")

c=b/a

print ("c:", c, "list:", list)

if ( c in list ):

   print ("c is available in the given list")

else:

    print ("c is not available in the given list")

**Output**

a: 10 b: 20 list: [1, 2, 3, 4, 5]

a is not present in the given list

b is not present in the given list

c: 2.0 list: [1, 2, 3, 4, 5]

c is available in the given list

**Python Identity Operators**

[Python identity operators](https://www.tutorialspoint.com/python/python_identity_operators.htm) compare the memory locations of two objects.

There are two Identity operators explained below −

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| is | Returns True if both variables are the same object and false otherwise. | a is b |
| is not | Returns True if both variables are not the same object and false otherwise. | a is not b |

Example of Python Identity Operators

a = [1, 2, 3, 4, 5]

b = [1, 2, 3, 4, 5]

c = a

print(a is c)

print(a is b)

print(a is not c)

print(a is not b)

**Output**

True

False

False

True

**Python Operators Precedence**

Operators precedence decides the order of the evaluation in which an operator is evaluated. Python operators have different levels of precedence. The following table contains the list of operators having highest to lowest precedence −

The following table lists all operators from highest [precedence](https://www.tutorialspoint.com/python/operators_precedence_example.htm) to lowest.

|  |  |
| --- | --- |
| **Sr.No.** | **Operator & Description** |
| 1 | **\*\***  Exponentiation (raise to the power) |
| 2 | **~ + -**  Complement, unary plus and minus (method names for the last two are +@ and -@) |
| 3 | **\* / % //**  Multiply, divide, modulo and floor division |
| 4 | **+ -**  Addition and subtraction |
| 5 | **>> <<**  Right and left bitwise shift |
| 6 | **&**  Bitwise 'AND' |
| 7 | **^ |**  Bitwise exclusive `OR' and regular `OR' |
| 8 | **<= < > >=**  Comparison operators |
| 9 | **<> == !=**  Equality operators |
| 10 | **= %= /= //= -= += \*= \*\*=**  Assignment operators |
| 11 | **is is not**  Identity operators |
| 12 | **in not in**  Membership operators |
| 13 | **not or and**  Logical operators |

**Python Arithmetic Operators**

Python arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, division, and more on numbers. Arithmetic operators are binary operators in the sense they operate on two operands. Python fully supports mixed arithmetic. That is, the two operands can be of two different number types. In such a situation.

Types of Arithmetic Operators

Following is the table which lists down all the arithmetic operators available in Python:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Name** | **Example** |
| + | Addition | a + b = 30 |
| - | Subtraction | a b = -10 |
| \* | Multiplication | a \* b = 200 |
| / | Division | b / a = 2 |
| % | Modulus | b % a = 0 |
| \*\* | Exponent | a\*\*b =10\*\*20 |
| // | Floor Division | 9//2 = 4 |

Let us study these operators with examples.

**Addition Operator**

The addition operator represents by + symbol. It is a basic arithmetic operator. It adds the two numeric operands on the either side and returns the addition result.

Example to add two integer numbers

In the following example, the two integer [variables](https://www.tutorialspoint.com/python/python_variables.htm) are the operands for the "+" operator.

a=10

b=20

print ("Addition of two integers")

print ("a =",a,"b =",b,"addition =",a+b)

It will produce the following **output** −

Addition of two integers

a = 10 b = 20 addition = 30

Example to add integer and float numbers

Addition of integer and float results in a float.

a=10

b=20.5

print ("Addition of integer and float")

print ("a =",a,"b =",b,"addition =",a+b)

It will produce the following **output** −

Addition of integer and float

a = 10 b = 20.5 addition = 30.5

Example to add two complex numbers

The result of adding float to complex is a complex number.

a=10+5j

b=20.5

print ("Addition of complex and float")

print ("a=",a,"b=",b,"addition=",a+b)

It will produce the following **output** −

Addition of complex and float

a= (10+5j) b= 20.5 addition= (30.5+5j)

**Subtraction Operator**

The subtraction operator represents by - symbol. It subtracts the second operand from the first. The resultant number is negative if the second operand is larger.

Example to subtract two integer numbers

First example shows subtraction of two integers.

a=10

b=20

print ("Subtraction of two integers:")

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

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

Result −

Subtraction of two integers

a = 10 b = 20 a-b = -10

a = 10 b = 20 b-a = 10

Example to subtract integer and float numbers

Subtraction of an integer and a float follows the same principle.

a=10

b=20.5

print ("subtraction of integer and float")

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

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

It will produce the following **output** −

subtraction of integer and float

a= 10 b= 20.5 a-b= -10.5

a= 10 b= 20.5 b-a= 10.5

Example to subtract complex numbers

In the subtraction involving a complex and a float, real component is involved in the operation.

a=10+5j

b=20.5

print ("subtraction of complex and float")

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

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

It will produce the following **output** −

subtraction of complex and float

a= (10+5j) b= 20.5 a-b= (-10.5+5j)

a= (10+5j) b= 20.5 b-a= (10.5-5j)

**Multiplication Operator**

The \* (asterisk) symbol is defined as a multiplication operator in Python (as in many languages). It returns the product of the two operands on its either side. If any of the operands negative, the result is also negative. If both are negative, the result is positive. Changing the order of operands doesn't change the result

Example to multiply two integers

a=10

b=20

print ("Multiplication of two integers")

print ("a =",a,"b =",b,"a\*b =",a\*b)

It will produce the following **output** −

Multiplication of two integers

a = 10 b = 20 a\*b = 200

Example to multiply integer and float numbers

In multiplication, a float operand may have a standard decimal point notation, or a scientific notation.

a=10

b=20.5

print ("Multiplication of integer and float")

print ("a=",a,"b=",b,"a\*b=",a\*b)

a=-5.55

b=6.75E-3

print ("Multiplication of float and float")

print ("a =",a,"b =",b,"a\*b =",a\*b)

It will produce the following **output** −

Multiplication of integer and float

a = 10 b = 20.5 a-b = -10.5

Multiplication of float and float

a = -5.55 b = 0.00675 a\*b = -0.037462499999999996

Example to multiply complex numbers

For the multiplication operation involving one complex operand, the other operand multiplies both the real part and imaginary part.

a=10+5j

b=20.5

print ("Multiplication of complex and float")

print ("a =",a,"b =",b,"a\*b =",a\*b)

It will produce the following **output** −

Multiplication of complex and float

a = (10+5j) b = 20.5 a\*b = (205+102.5j)

**Division Operator**

The "/" symbol is usually called as forward slash. The result of division operator is numerator (left operand) divided by denominator (right operand). The resultant number is negative if any of the operands is negative. Since infinity cannot be stored in the memory, Python raises ZeroDivisionError if the denominator is 0.

The result of division operator in Python is always a float, even if both operands are integers. Example to divide two numbers

a=10

b=20

print ("Division of two integers")

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

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

It will produce the following **output** −

Division of two integers

a= 10 b= 20 a/b= 0.5

a= 10 b= 20 b/a= 2.0

Example to divide two float numbers

In Division, a float operand may have a standard decimal point notation, or a scientific notation.

a=10

b=-20.5

print ("Division of integer and float")

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

a=-2.50

b=1.25E2

print ("Division of float and float")

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

It will produce the following **output** −

Division of integer and float

a= 10 b= -20.5 a/b= -0.4878048780487805

Division of float and float

a= -2.5 b= 125.0 a/b= -0.02

Example to divide complex numbers

When one of the operands is a complex number, division between the other operand and both parts of complex number (real and imaginary) object takes place.

a=7.5+7.5j

b=2.5

print ("Division of complex and float")

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

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

It will produce the following **output** −

Division of complex and float

a = (7.5+7.5j) b = 2.5 a/b = (3+3j)

a = (7.5+7.5j) b = 2.5 b/a = (0.16666666666666666-0.16666666666666666j)

If the numerator is 0, the result of division is always 0 except when denominator is 0, in which case, Python raises ZeroDivisionError wirh Division by Zero error message

a=0

b=2.5

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

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

It will produce the following **output** −

a= 0 b= 2.5 a/b= 0.0

Traceback (most recent call last):

  File "C:\Users\mlath\examples\example.py", line 20, in <module>

     print ("a=",a,"b=",b,"b/a=",b/a)

                                 ~^~

ZeroDivisionError: float division by zero

**Modulus Operator**

Python defines the "%" symbol, which is known aa Percent symbol, as Modulus (or modulo) operator. It returns the remainder after the denominator divides the numerator. It can also be called Remainder operator. The result of the modulus operator is the number that remains after the integer quotient. To give an example, when 10 is divided by 3, the quotient is 3 and remainder is 1. Hence, 10%3 (normally pronounced as 10 mod 3) results in 1.

**Example for modulus operation on integers**

If both the operands are integer, the modulus value is an integer. If numerator is completely divisible, remainder is 0. If numerator is smaller than denominator, modulus is equal to the numerator. If denominator is 0, Python raises ZeroDivisionError.

a=10

b=2

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

a=10

b=4

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

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

a=0

b=10

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

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

It will produce the following **output** −

a= 10 b= 2 a%b= 0

a= 10 b= 4 a%b= 2

a= 10 b= 4 b%a= 4

a= 0 b= 10 a%b= 0

Traceback (most recent call last):

  File "C:\Users\mlath\examples\example.py", line 13, in <module>

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

                                    ~^~

ZeroDivisionError: integer modulo by zero

Example for modulus operation on floats

If any of the operands is a float, the mod value is always float.

a=10

b=2.5

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

a=10

b=1.5

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

a=7.7

b=2.5

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

a=12.4

b=3

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

It will produce the following **output** −

a= 10 b= 2.5 a%b= 0.0

a= 10 b= 1.5 a%b= 1.0

a= 7.7 b= 2.5 a%b= 0.20000000000000018

a= 12.4 b= 3 a%b= 0.40000000000000036

Python doesn't accept complex numbers to be used as operand in modulus operation. It throws TypeError: unsupported operand type(s) for %.

**Exponent Operator**

Python uses \*\* (double asterisk) as the exponent operator (sometimes called raised to operator). So, for a\*\*b, you say a raised to b, or even bth power of a.

If in the exponentiation expression, both operands are integer, result is also an integer. In case either one is a float, the result is float. Similarly, if either one operand is complex number, exponent operator returns a complex number.

If the base is 0, the result is 0, and if the index is 0 then the result is always 1.

Example of exponent operator

a=10

b=2

print ("a=",a, "b=",b, "a\*\*b=", a\*\*b)

a=10

b=1.5

print ("a=",a, "b=",b, "a\*\*b=", a\*\*b)

a=7.7

b=2

print ("a=",a, "b=",b, "a\*\*b=", a\*\*b)

a=1+2j

b=4

print ("a=",a, "b=",b, "a\*\*b=", a\*\*b)

a=12.4

b=0

print ("a=",a, "b=",b, "a\*\*b=", a\*\*b)

print ("a=",a, "b=",b, "b\*\*a=", b\*\*a)

It will produce the following **output** −

a= 10 b= 2 a\*\*b= 100

a= 10 b= 1.5 a\*\*b= 31.622776601683793

a= 7.7 b= 2 a\*\*b= 59.290000000000006

a= (1+2j) b= 4 a\*\*b= (-7-24j)

a= 12.4 b= 0 a\*\*b= 1.0

a= 12.4 b= 0 b\*\*a= 0.0

**Floor Division Operator**

Floor division is also called as integer division. Python uses // (double forward slash) symbol for the purpose. Unlike the modulus or modulo which returns the remainder, the floor division gives the quotient of the division of operands involved.

If both operands are positive, floor operator returns a number with fractional part removed from it. For example, the floor division of 9.8 by 2 returns 4 (pure division is 4.9, strip the fractional part, result is 4).

But if one of the operands is negative, the result is rounded away from zero (towards negative infinity). Floor division of -9.8 by 2 returns 5 (pure division is -4.9, rounded away from 0).

Example of floor division operator

a=9

b=2

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

a=9

b=-2

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

a=10

b=1.5

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

a=-10

b=1.5

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

It will produce the following **output** −

a= 9 b= 2 a//b= 4

a= 9 b= -2 a//b= -5

a= 10 b= 1.5 a//b= 6.0

a= -10 b= 1.5 a//b= -7.0

Precedence and Associativity of Arithmetic Operators

|  |  |  |
| --- | --- | --- |
| **Operator(s)** | **Description** | **Associativity** |
| \*\* | Exponent Operator | Associativity of Exponent operator is from **Right to Left**. |
| %, \*, /, // | Modulus, Multiplication, Division, and Floor Division | Associativity of Modulus, Multiplication, Division, and Floor Division operators are from **Left to Right**. |
| +, | Addition and Subtraction Operators | Associativity of Addition and Subtraction operators are from **Left to Right**. |

The following table shows the precedence and associativity of the arithmetic operators in Python.

**Arithmetic Operators with Complex Numbers**

Arithmetic operators behave slightly differently when the both operands are complex number objects.

**Addition and subtraction of complex numbers**

Addition and subtraction of complex numbers is a simple addition/subtraction of respective real and imaginary components.

a=2.5+3.4j

b=-3+1.0j

print ("Addition of complex numbers - a=",a, "b=",b, "a+b=", a+b)

print ("Subtraction of complex numbers - a=",a, "b=",b, "a-b=", a-b)

It will produce the following **output** −

Addition of complex numbers - a= (2.5+3.4j) b= (-3+1j) a+b= (-0.5+4.4j)

Subtraction of complex numbers - a= (2.5+3.4j) b= (-3+1j) a-b= (5.5+2.4j)

**Multiplication of complex numbers**

Multiplication of complex numbers is similar to multiplication of two binomials in algebra. If "a+bj" and "x+yj" are two complex numbers, then their multiplication is given by this formula −

(a+bj)\*(x+yj) = ax+ayj+xbj+byj2 = (ax-by)+(ay+xb)j

For example,

a=6+4j

b=3+2j

c=a\*b

c=(18-8)+(12+12)j

c=10+24j

The following program confirms the result −

a=6+4j

b=3+2j

print ("Multplication of complex numbers - a=",a, "b=",b, "a\*b=", a\*b)

To understand the how the division of two complex numbers takes place, we should use the conjugate of a complex number. Python's complex object has a conjugate() method that returns a complex number with the sign of imaginary part reversed.

>>> a=5+6j

>>> a.conjugate()

(5-6j)

**Division of complex numbers**

To divide two complex numbers, divide and multiply the numerator as well as the denominator with the conjugate of denominator.

a=6+4j

b=3+2j

c=a/b

c=(6+4j)/(3+2j)

c=(6+4j)\*(3-2j)/3+2j)\*(3-2j)

c=(18-12j+12j+8)/(9-6j+6j+4)

c=26/13

c=2+0j

To verify, run the following code −

a=6+4j

b=3+2j

print ("Division of complex numbers - a=",a, "b=",b, "a/b=", a/b)

Python Comparison Operators

**Comparison operators** in Python are very important in [Python's conditional statements](https://www.tutorialspoint.com/python/python_decision_making.htm) ([**if, else** and **elif**](https://www.tutorialspoint.com/python/python_if_else.htm)) and [looping statements](https://www.tutorialspoint.com/python/python_loops.htm) ([while](https://www.tutorialspoint.com/python/python_while_loops.htm) and [for loops](https://www.tutorialspoint.com/python/python_for_loops.htm)). The **comparison operators** also called **relational operators**. Some of the well known operators are "<" stands for less than, and ">" stands for greater than operator.

Python uses two more operators, combining "=" symbol with these two. The "<=" symbol is for less than or equal to operator and the ">=" symbol is for greater than or equal to operator.

Different Comparison Operators in Python

Python has two more comparison operators in the form of "==" and "!=". They are for **is equal to** and **is not equal to** operators. Hence, there are six comparison operators in Python and they are listed below in this table:

|  |  |  |
| --- | --- | --- |
| < | Less than | a<b |
| > | Greater than | a>b |
| <= | Less than or equal to | a<=b |
| >= | Greater than or equal to | a>=b |
| == | Is equal to | a==b |
| != | Is not equal to | a!=b |

Comparison operators are binary in nature, requiring two operands. An expression involving a comparison operator is called a Boolean expression, and always returns either True or False.

Example

a=5

b=7

print (a>b)

print (a<b)

It will produce the following **output** −

False

True

Both the operands may be [Python literals](https://www.tutorialspoint.com/python/python_literals.htm), [variables](https://www.tutorialspoint.com/python/python_variables.htm) or expressions. Since Python supports mixed arithmetic, you can have any number type operands.

Example

The following code demonstrates the use of Python's **comparison operators with integer numbers** −

print ("Both operands are integer")

a=5

b=7

print ("a=",a, "b=",b, "a>b is", a>b)

print ("a=",a, "b=",b,"a<b is",a<b)

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

print ("a=",a, "b=",b,"a!=b is",a!=b)

It will produce the following **output** −

Both operands are integer

a= 5 b= 7 a>b is False

a= 5 b= 7 a<b is True

a= 5 b= 7 a==b is False

a= 5 b= 7 a!=b is True

Comparison of Float Number

In the following example, an integer and a float operand are compared.

Example

print ("comparison of int and float")

a=10

b=10.0

print ("a=",a, "b=",b, "a>b is", a>b)

print ("a=",a, "b=",b,"a<b is",a<b)

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

print ("a=",a, "b=",b,"a!=b is",a!=b)

It will produce the following **output** −

comparison of int and float

a= 10 b= 10.0 a>b is False

a= 10 b= 10.0 a<b is False

a= 10 b= 10.0 a==b is True

a= 10 b= 10.0 a!=b is False

Comparison of Complex umbers

Although complex object is a number [data type in Python](https://www.tutorialspoint.com/python/python_data_types.htm), its behavior is different from others. Python doesn't support < and > operators, however it does support equality (==) and inequality (!=) operators.

Example

print ("comparison of complex numbers")

a=10+1j

b=10.-1j

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

print ("a=",a, "b=",b,"a!=b is",a!=b)

It will produce the following **output** −

comparison of complex numbers

a= (10+1j) b= (10-1j) a==b is False

a= (10+1j) b= (10-1j) a!=b is True

You get a TypeError with less than or greater than operators.

Example

print ("comparison of complex numbers")

a=10+1j

b=10.-1j

print ("a=",a, "b=",b,"a<b is",a<b)

print ("a=",a, "b=",b,"a>b is",a>b)

It will produce the following **output** −

comparison of complex numbers

Traceback (most recent call last):

  File "C:\Users\mlath\examples\example.py", line 5, in <module>

    print ("a=",a, "b=",b,"a<b is",a<b)

                                      ^^^

TypeError: '<' not supported between instances of 'complex' and

'complex

Comparison of Booleans

Boolean objects in Python are really integers: True is 1 and False is 0. In fact, Python treats any non-zero number as True. In Python, comparison of Boolean objects is possible. "False < True" is True!

Example

print ("comparison of Booleans")

a=True

b=False

print ("a=",a, "b=",b,"a<b is",a<b)

print ("a=",a, "b=",b,"a>b is",a>b)

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

print ("a=",a, "b=",b,"a!=b is",a!=b)

It will produce the following **output** −

comparison of Booleans

a= True b= False a<b is False

a= True b= False a>b is True

a= True b= False a==b is False

a= True b= False a!=b is True

Comparison of Sequence Types

In Python, comparison of only similar sequence objects can be performed. A string object is comparable with another string only. A [list](https://www.tutorialspoint.com/python/python_lists.htm) cannot be compared with a [tuple](https://www.tutorialspoint.com/python/python_tuples.htm), even if both have same items.

Example

print ("comparison of different sequence types")

a=(1,2,3)

b=[1,2,3]

print ("a=",a, "b=",b,"a<b is",a<b)

It will produce the following **output** −

comparison of different sequence types

Traceback (most recent call last):

  File "C:\Users\mlath\examples\example.py", line 5, in <module>

    print ("a=",a, "b=",b,"a<b is",a<b)

                                       ^^^

TypeError: '<' not supported between instances of 'tuple' and 'list'

Sequence objects are compared by lexicographical ordering mechanism. The comparison starts from item at 0th index. If they are equal, comparison moves to next index till the items at certain index happen to be not equal, or one of the sequences is exhausted. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one.

Which of the operands is greater depends on the difference in values of items at the index where they are unequal. For example, 'BAT'>'BAR' is True, as T comes after R in Unicode order.

If all items of two sequences compare equal, the sequences are considered equal.

Example

print ("comparison of strings")

a='BAT'

b='BALL'

print ("a=",a, "b=",b,"a<b is",a<b)

print ("a=",a, "b=",b,"a>b is",a>b)

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

print ("a=",a, "b=",b,"a!=b is",a!=b)

It will produce the following **output** −

comparison of strings

a= BAT b= BALL a<b is False

a= BAT b= BALL a>b is True

a= BAT b= BALL a==b is False

a= BAT b= BALL a!=b is True

In the following example, two tuple objects are compared −

Example

print ("comparison of tuples")

a=(1,2,4)

b=(1,2,3)

print ("a=",a, "b=",b,"a<b is",a<b)

print ("a=",a, "b=",b,"a>b is",a>b)

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

print ("a=",a, "b=",b,"a!=b is",a!=b)

It will produce the following **output** −

a= (1, 2, 4) b= (1, 2, 3) a<b is False

a= (1, 2, 4) b= (1, 2, 3) a>b is True

a= (1, 2, 4) b= (1, 2, 3) a==b is False

a= (1, 2, 4) b= (1, 2, 3) a!=b is True

Comparison of Dictionary Objects

The use of "<" and ">" operators for Python's dictionary is not defined. In case of these operands, TypeError: '<' not supported between instances of 'dict' and 'dict' is reported.

Equality comparison checks if the length of both the dict items is same. Length of dictionary is the number of key-value pairs in it.

Python dictionaries are simply compared by length. The dictionary with fewer elements is considered less than a dictionary with more elements.

Example

print ("comparison of dictionary objects")

a={1:1,2:2}

b={2:2, 1:1, 3:3}

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

print ("a=",a, "b=",b,"a!=b is",a!=b)

It will produce the following **output** −

comparison of dictionary objects

a= {1: 1, 2: 2} b= {2: 2, 1: 1, 3: 3} a==b is False

a= {1: 1, 2: 2} b= {2: 2, 1: 1, 3: 3} a!=b is True

Python Assignment Operator

The = (equal to) symbol is defined as assignment operator in Python. The value of Python expression on its right is assigned to a single [variable](https://www.tutorialspoint.com/python/python_variables.htm) on its left. The = symbol as in programming in general (and Python in particular) should not be confused with its usage in Mathematics, where it states that the expressions on the either side of the symbol are equal.

Example of Assignment Operator in Python

Consider following Python statements −

a = 10

b = 5

a = a + b

print (a)

At the first instance, at least for somebody new to programming but who knows maths, the statement "a=a+b" looks strange. How could a be equal to "a+b"? However, it needs to be reemphasized that the = symbol is an assignment operator here and not used to show the equality of LHS and RHS.

Because it is an assignment, the expression on right evaluates to 15, the value is assigned to a.

In the statement "a+=b", the two operators "+" and "=" can be combined in a "+=" operator. It is called as add and assign operator. In a single statement, it performs addition of two operands "a" and "b", and result is assigned to operand on left, i.e., "a".

Augmented Assignment Operators in Python

In addition to the simple assignment operator, Python provides few more assignment operators for advanced use. They are called cumulative or augmented assignment operators. In this chapter, we shall learn to use augmented assignment operators defined in Python.

Python has the augmented assignment operators for all [arithmetic](https://www.tutorialspoint.com/python/python_arithmetic_operators.htm) and comparison operators.

Python augmented assignment operators combines addition and assignment in one statement. Since Python supports mixed arithmetic, the two operands may be of different types. However, the type of left operand changes to the operand of on right, if it is wider.

Example

The += operator is an augmented operator. It is also called cumulative addition operator, as it adds "b" in "a" and assigns the result back to a variable.

The following are the augmented assignment operators in Python:

* Augmented Addition Operator
* Augmented Subtraction Operator
* Augmented Multiplication Operator
* Augmented Division Operator
* Augmented Modulus Operator
* Augmented Exponent Operator
* Augmented Floor division Operator

Augmented Addition Operator (+=)

Following examples will help in understanding how the "+=" operator works −

a=10

b=5

print ("Augmented addition of int and int")

a+=b # equivalent to a=a+b

print ("a=",a, "type(a):", type(a))

a=10

b=5.5

print ("Augmented addition of int and float")

a+=b  # equivalent to a=a+b

print ("a=",a, "type(a):", type(a))

a=10.50

b=5+6j

print ("Augmented addition of float and complex")

a+=b #equivalent to a=a+b

print ("a=",a, "type(a):", type(a))

It will produce the following **output** −

Augmented addition of int and int

a= 15 type(a): <class 'int'>

Augmented addition of int and float

a= 15.5 type(a): <class 'float'>

Augmented addition of float and complex

a= (15.5+6j) type(a): <class 'complex'>

Augmented Subtraction Operator (-=)

Use -= symbol to perform subtract and assign operations in a single statement. The "a-=b" statement performs "a=a-b" assignment. Operands may be of any number type. Python performs implicit type casting on the [object](https://www.tutorialspoint.com/python/python_object_classes.htm) which is narrower in size.

a=10

b=5

print ("Augmented subtraction of int and int")

a-=b #equivalent to a=a-b

print ("a=",a, "type(a):", type(a))

a=10

b=5.5

print ("Augmented subtraction of int and float")

a-=b #equivalent to a=a-b

print ("a=",a, "type(a):", type(a))

a=10.50

b=5+6j

print ("Augmented subtraction of float and complex")

a-=b #equivalent to a=a-b

print ("a=",a, "type(a):", type(a))

It will produce the following **output** −

Augmented subtraction of int and int

a= 5 type(a): <class 'int'>

Augmented subtraction of int and float

a= 4.5 type(a): <class 'float'>

Augmented subtraction of float and complex

a= (5.5-6j) type(a): <class 'complex'>

Augmented Multiplication Operator (\*=)

The "\*=" operator works on similar principle. "a\*=b" performs multiply and assign operations, and is equivalent to "a=a\*b". In case of augmented multiplication of two complex numbers, the rule of multiplication as discussed in the previous chapter is applicable.

a=10

b=5

print ("Augmented multiplication of int and int")

a\*=b #equivalent to a=a\*b

print ("a=",a, "type(a):", type(a))

a=10

b=5.5

print ("Augmented multiplication of int and float")

a\*=b #equivalent to a=a\*b

print ("a=",a, "type(a):", type(a))

a=6+4j

b=3+2j

print ("Augmented multiplication of complex and complex")

a\*=b #equivalent to a=a\*b

print ("a=",a, "type(a):", type(a))

It will produce the following **output** −

Augmented multiplication of int and int

a= 50 type(a): <class 'int'>

Augmented multiplication of int and float

a= 55.0 type(a): <class 'float'>

Augmented multiplication of complex and complex

a= (10+24j) type(a): <class 'complex'>

Augmented Division Operator (/=)

The combination symbol "/=" acts as divide and assignment operator, hence "a/=b" is equivalent to "a=a/b". The division operation of int or float operands is float. Division of two complex numbers returns a complex number. Given below are examples of augmented division operator.

a=10

b=5

print ("Augmented division of int and int")

a/=b #equivalent to a=a/b

print ("a=",a, "type(a):", type(a))

a=10

b=5.5

print ("Augmented division of int and float")

a/=b #equivalent to a=a/b

print ("a=",a, "type(a):", type(a))

a=6+4j

b=3+2j

print ("Augmented division of complex and complex")

a/=b #equivalent to a=a/b

print ("a=",a, "type(a):", type(a))

It will produce the following **output** −

Augmented division of int and int

a= 2.0 type(a): <class 'float'>

Augmented division of int and float

a= 1.8181818181818181 type(a): <class 'float'>

Augmented division of complex and complex

a= (2+0j) type(a): <class 'complex'>

Augmented Modulus Operator (%=)

To perform modulus and assignment operation in a single statement, use the %= operator. Like the mod operator, its augmented version also is not supported for complex number.

a=10

b=5

print ("Augmented modulus operator with int and int")

a%=b #equivalent to a=a%b

print ("a=",a, "type(a):", type(a))

a=10

b=5.5

print ("Augmented modulus operator with int and float")

a%=b #equivalent to a=a%b

print ("a=",a, "type(a):", type(a))

It will produce the following **output** −

Augmented modulus operator with int and int

a= 0 type(a): <class 'int'>

Augmented modulus operator with int and float

a= 4.5 type(a): <class 'float'>

Augmented Exponent Operator (\*\*=)

The "\*\*=" operator results in computation of "a" raised to "b", and assigning the value back to "a". Given below are some examples −

a=10

b=5

print ("Augmented exponent operator with int and int")

a\*\*=b #equivalent to a=a\*\*b

print ("a=",a, "type(a):", type(a))

a=10

b=5.5

print ("Augmented exponent operator with int and float")

a\*\*=b #equivalent to a=a\*\*b

print ("a=",a, "type(a):", type(a))

a=6+4j

b=3+2j

print ("Augmented exponent operator with complex and complex")

a\*\*=b #equivalent to a=a\*\*b

print ("a=",a, "type(a):", type(a))

It will produce the following **output** −

Augmented exponent operator with int and int

a= 100000 type(a): <class 'int'>

Augmented exponent operator with int and float

a= 316227.7660168379 type(a): <class 'float'>

Augmented exponent operator with complex and complex

a= (97.52306038414744-62.22529992036203j) type(a): <class 'complex'>

Augmented Floor division Operator (//=)

For performing floor division and assignment in a single statement, use the "//=" operator. "a//=b" is equivalent to "a=a//b". This operator cannot be used with complex numbers.

a=10

b=5

print ("Augmented floor division operator with int and int")

a//=b #equivalent to a=a//b

print ("a=",a, "type(a):", type(a))

a=10

b=5.5

print ("Augmented floor division operator with int and float")

a//=b #equivalent to a=a//b

print ("a=",a, "type(a):", type(a))

It will produce the following **output** −

Augmented floor division operator with int and int

a= 2 type(a): <class 'int'>

Augmented floor division operator with int and float

a= 1.0 type(a): <class 'float'>