Python - Data Types

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.

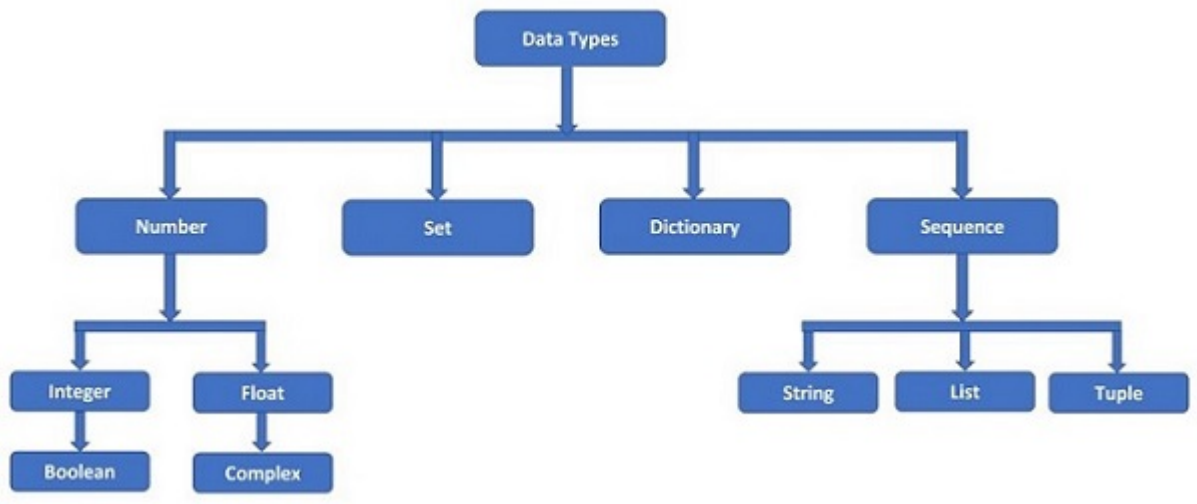
Each programming language has its own classification of data items. With these datatypes, we can store different types of data values.

Types of Data Types in Python

Python supports the following built-in data types –

- Numeric Data Types
  - int
  - flot
  - complex
- String Data Types
- Sequence Data Types
  - list
  - tuple
  - range
- Binary Data Types
  - bytes
  - bytearray
  - memoryview
- Dictionary Data Type
- Set Data Type
  - set
  - frozenset

- Boolean Data Type
- None Type



Learn **Python** in-depth with real-world projects through our **Python certification course**. Enroll and become a certified expert to boost your career.

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 ($\sqrt{-1}$), 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'>
```

Here are some examples of numbers –

int	float	complex
10	0.0	3.14j
0O777	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3+e18	.876j
0x17	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

Example of Numeric Data Types

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

</> Open Compiler

```
# 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 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** and **concatenation** 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

</> Open Compiler

```
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

</> Open Compiler

```
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 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

</> Open Compiler

```
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 –

</> Open Compiler

```
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 –

(a) 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 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 –

</> Open Compiler

```
# 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.

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() constructor, slicing bytes or bytearray objects, extracting from array objects, or using built-in functions like open() 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 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

</> Open Compiler

```
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 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 –

</> Open Compiler

```
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 –

</> Open Compiler

```
# 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** –

</> Open Compiler

```
# 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()** 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))
```

Open Compiler

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.

Read: Python Type Casting

Example

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

</> Open Compiler

```
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 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** Converts x to an integer. base specifies the base if x is a string.
2	**Python long() function** Converts x to a long integer. base specifies the base if x is a string. This function has been deprecated.
3	**Python float() function** Converts x to a floating-point number.
4	**Python complex() function** Creates a complex number.
5	**Python str() function** Converts object x to a string representation.
6	**Python repr() function** Converts object x to an expression string.
7	**Python eval() function** Evaluates a string and returns an object.
8	**Python tuple() function** Converts s to a tuple.
9	**Python list() function** Converts s to a list.
10	**Python set() function** Converts s to a set.
11	**Python dict() function** Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**Python frozenset() function** Converts s to a frozen set.
13	**Python chr() function** Converts an integer to a character.
14	**Python unichr() function** Converts an integer to a Unicode character.
15	**Python ord() function** Converts a single character to its integer value.
16	**Python hex() function** Converts an integer to a hexadecimal string.

17	**Python oct() function** Converts an integer to an octal string.