Python Data Types

Variables can hold values, and every value has a data-type. Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

```python
a = 5
```

The variable **a** holds integer value five and we did not define its type. Python interpreter will automatically interpret variables **a** as an integer type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function, which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

```python
a=10
b="Hi Python"
c = 10.5
print(type(a))
print(type(b))
print(type(c))
```

Output:

```
<type 'int'>
<type 'str'>
<type 'float'>
```

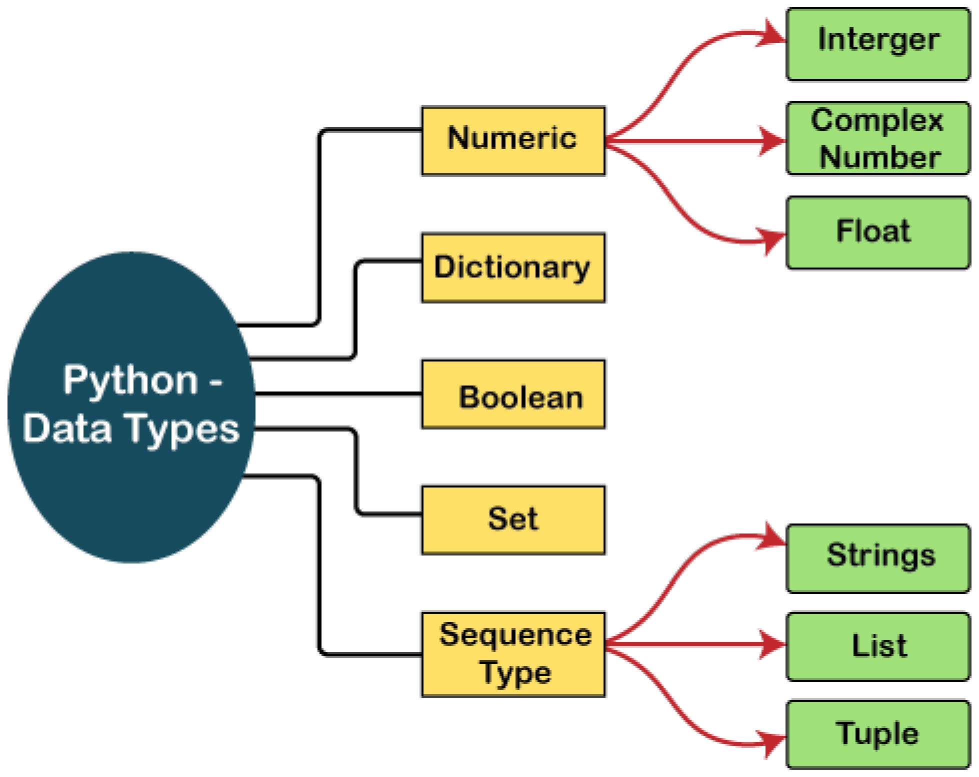
Standard data types

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. Numbers
2. Sequence Type
3. Boolean
4. Set

5. Dictionary



In this section of the tutorial, we will give a brief introduction of the above data-types. We will discuss each one of them in detail later in this tutorial.

Numbers

Number stores numeric values. The integer, float, and complex values belong to a Python Numbers data-type. Python provides the **type()** function to know the data-type of the variable. Similarly, the **isinstance()** function is used to check an object belongs to a particular class.

Python creates Number objects when a number is assigned to a variable. For example;

```
a = 5
print("The type of a", type(a))


b = 40.5
print("The type of b", type(b))


c = 1+3j
```

```
print("The type of c", type(c))
```

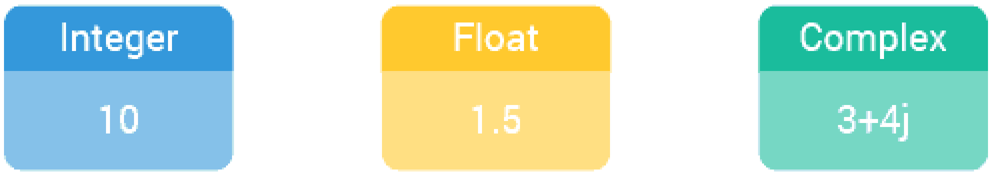
1. `print(" c is a complex number", isinstance(1+3j,complex))`

   Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

Python supports three types of numeric data.

1. **Int -** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
2. **Float -** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
3. **complex -** A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.



Integers

An integer is a whole number that can be positive or negative.

```
# Following numbers are integers
x = 10
y = -10
z = 123456789
```

In Python 3, there is no limit to how long an integer value can be. It can grow to have as many digits as your computer’s memory space allows.

```
# Integers have unlimited precision
x =
999999999999999999999999999999999999999999999999999999999999999999999999999
99999
```

We normally write integers in base 10. However, Python allows us to write integers in **Hexadecimal** (base 16), **Octal** (base 8), and **Binary** (base 2) formats. You can do that by adding one of the following prefixes to the integer.

Prefix	**Interpretation**	**Base**
‘0b’ or ‘0B’	Binary	2
‘0o’ or ‘0O’	Octal	8
‘0x’ or ‘0X’	Hexadecimal	16

```
# Integers in binary, octal and hexadecimal formats

# binary
print(0b10111011)
# Prints 187

# octal
print(0o10)
# Prints 8

# hex
print(0xFF)
# Prints 255
```

In addition, **Boolean** is a sub-type of integers.

```
# Boolean in python
x = True
x = False
```

Floating-Point Numbers

Floating-point number or Float is a positive or negative number with a fractional part.

```
# Following numbers are floats
x = 10.1

y = -10.5
z = 1.123456
```

You can append the character `e` or `E` followed by a positive or negative integer to specify **scientific notation**.

```
# Scientific notation
print(42e3)

# Prints 42000.0


print(4.2e-3)
# Prints 0.0042
```

The maximum value a float can have is approximately **$1.8×10^{308}$** . Any number greater than that is indicated by the string inf (infinity)

```
# Maximum value of a float
print(1.79e308)

# Prints 1.79e+308


print(1.8e308)
# Prints inf
```

However, the minimum value a float can have is approximately **$5.0×10^{-324}$** . Any number, less than that is considered zero.

```
# Minimum value of a float
print(5e-324)

# Prints 4.94065645841e-324


print(5e-325)
# Prints 0.0
```

Complex Numbers

A complex number is specified as `real_part + imaginary_part`, where the imaginary_part is written with a `j` or `J`.

```
# Following numbers are complex numbers
x = 2j
y = 3+4j
```

To extract real and imaginary parts from a complex number x, use `x.real` and `x.imag`

```
x = 3+4j


# real part
print(x.real)
# Prints 3.0


# imaginary part
print(x.imag)
# Prints 4.0
```

Sequence Type

String

The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.

String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.

In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*.

The operator * is known as a repetition operator as the operation "Python" *2 returns 'Python Python'.

The following example illustrates the string in Python.

Example - 1

```
str = "string using double quotes"
print(str)
s = '''A multiline
string'''
print(s)
```

Output:

```
string using double quotes
A multiline
string
```

Consider the following example of string handling.

Example - 2

```
str1 = 'hello hellopython' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

List

Python Lists are similar to arrays in C. However, the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].

We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Consider the following example.

```
list1  = [1, "hi", "Python", 2]
#Checking type of given list
print(type(list1))
```

```python

#Printing the list1
print (list1)

# List slicing
print (list1[3:])

# List slicing
print (list1[0:2])

# List Concatenation using + operator
print (list1 + list1)

# List repetation using * operator
print (list1 * 3)
```

Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

Tuple

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

```python
tup  = ("hi", "Python", 2)
# Checking type of tup
print (type(tup))

#Printing the tuple
```

```
print (tup)

# Tuple slicing
print (tup[1:])
print (tup[0:1])

# Tuple concatenation using + operator
print (tup + tup)

# Tuple repatation using * operator
print (tup * 3)

# Adding value to tup. It will throw an error.
t[2] = "hi"
```

Output:

```
<class 'tuple'>
('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)

Traceback (most recent call last):
  File "main.py", line 14, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary

Dictionary is an unordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type, whereas value is an arbitrary Python object.

The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Consider the following example.

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
```

```
# Printing dictionary
print (d)

# Accesing value using keys
print("1st name is "+d[1])
print("2nd name is "+ d[4])
  print (d.keys())
print (d.values())
```

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Boolean

Boolean type provides two built-in values, True and False. These values are used to determine the given statement true or false. It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'. Consider the following example.

```
# Python program to check the boolean type
A = 5>3
print(A)

print(type(A))
print(type(True))
print(type(False))
print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

Set

Python Set is the unordered collection of the data type. It is iterable, mutable(can modify after creation), and has unique elements. In set, the order of the elements is undefined; it may return the changed sequence of the element. The set is created by using a built-in function **set(),** or a sequence of elements is passed in the curly braces and separated by the comma. It can contain various types of values. Consider the following example.

```python
# Creating Empty set
set1 = set()

set2 = {'James', 2, 3,'Python'}

#Printing Set value
print(set2)

# Adding element to the set

set2.add(10)
print(set2)

#Removing element from the set
set2.remove(2)
print(set2)
```

Output:

```
{3, 'Python', 'James', 2}
{'Python', 'James', 3, 2, 10}
{'Python', 'James', 3, 10}
```

Python Literals

Python Literals can be defined as data that is given in a variable or constant.

Python supports the following literals:

1. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

Example:

```
Name = "Aman" , '12345'
Num = 20
Fnum = 25.5
```