Python Data Types

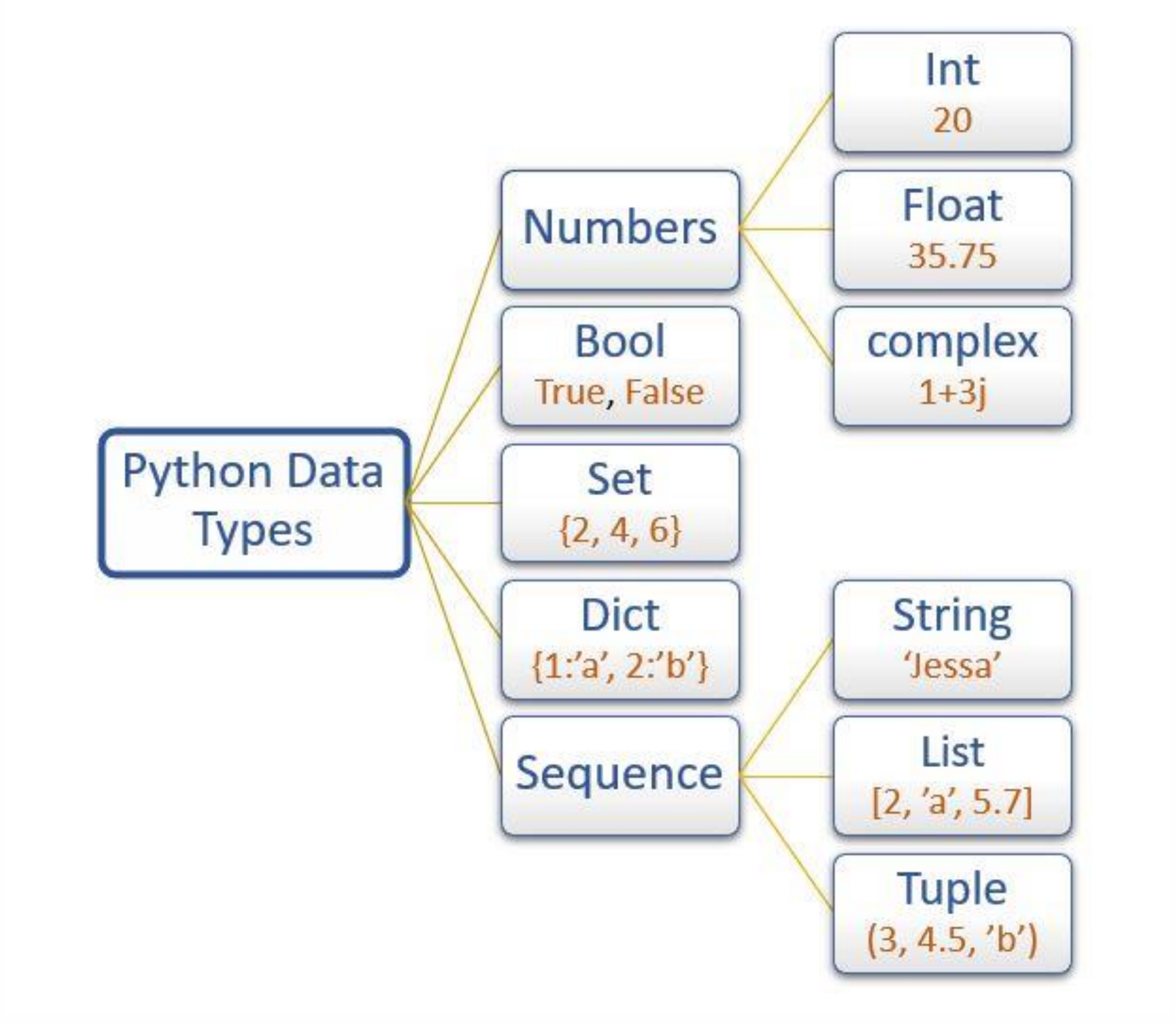
Data types specify the different sizes and values that can be stored in the variable. For example, Python stores numbers, strings, and a list of values using different data types.

Python is a dynamically typed language; therefore, we do not need to specify the variable's type while declaring it. Whatever value we assign to the variable based on that data type will be automatically assigned. For example, `name = 'Jessa'` here Python will store the name variable as a `str` data type.

No matter what value is stored in a variable (object), a variable can be any type like int, float, str, list, set, tuple, dict, bool, etc.

There are mainly four types of basic/primitive data types available in Python

- **Numeric**: int, float, and complex
- **Sequence**: String, list, and tuple
- **Set**
- **Dictionary** (dict)



To check the data type of variable use the built-in function `type()` and `isinstance()`.

- The `type()` function returns the data type of the variable
- The isinstance() function checks whether an object belongs to a particular class.

In this article, we will learn the following Python data types in detail.

Data type	Description	Example
int	To store integer values	n = 20
float	To store decimal values	n = 20.75

Data type	Description	Example
complex	To store complex numbers (real and imaginary part)	n = 10+20j
str	To store textual/string data	name = 'Jessa'
bool	To store boolean values	flag = True
list	To store a sequence of mutable data	l = [3, 'a', 2.5]
tuple	To store sequence immutable data	t =(2, 'b', 6.4)
dict	To store key: value pair	d = {1:'J', 2:'E'}
set	To store unorder and unindexed values	s = {1, 3, 5}
frozenset	To store immutable version of the set	f_set=frozenset({5,7
range	To generate a sequence of number	numbers = range(10
bytes	To store bytes values	b=bytes([5,10,15,11

Python Data Types

In Python, A string is a **sequence of characters enclosed within a single quote or double quote**. These characters could be anything like letters, numbers, or special symbols enclosed within double quotation marks. For example, `"PYnative"` is a string.

The string type in Python is represented using a `str` class.

To work with text or character data in Python, we use Strings. Once a string is created, we can do many operations on it, such as searching inside it, creating a substring from it, and splitting it.

```
Example:
platform = "PYnative"
print(type(platform))  # <class 'str'>

# display string
print(platform)  # 'PYnative'

# accessing 2nd character of a string
print(platform[1])  # Y
```

Note: The string is immutable, i.e., it cannot be changed once defined. You need to create a copy of it if you want to modify it. This non-changeable behaviour is called immutability.

Int data type

Python uses the int data type to **represent whole integer values**. For example, we can use the int data type to store the roll number of a student. The Integer type in Python is represented using a `int` class.

You can store positive and negative integer numbers of any length such as 235, -758, 235689741.

We can create an integer variable using the two ways

1. Directly assigning an integer value to a variable
2. Using a `int()` class.

Example

```
# store int value
roll_no = 33
# display roll no
print("Roll number is:", roll_no)
# output 33
print(type(roll_no))
# output class 'int'

# store integer using int() class
id = int(25)
print(id)  # 25
print(type(id))  # class 'int'
```

You can also store integer values other than base 10 such as

- Binary (base 2)
- Octal (base 8)
- Hexadecimal numbers (base 16)

Example

```
# decimal int 16 with base 8
# Prefix with zero + letter o
octal_num = 0o20
print(octal_num)  # 16
print(type(octal_num))  # class 'int'

# decimal int 16 with base 16
# Prefix with zero + letter x
hexadecimal_num = 0x10  # decimal equivalent of 21
print(hexadecimal_num)  # 16
print(type(hexadecimal_num))  # class 'int'

# decimal int 16 with base 2
# Prefix with zero + letter b
binary_num = 0b10000  # decimal equivalent of 6
print(binary_num)  # 16
print(type(binary_num))  # class 'int'
```

Float data type

To represent **floating-point values or decimal value**s, we can use the float data type. For example, if we want to store the salary, we can use the float type.

The float type in Python is represented using a `float` class.

We can create a float variable using the two ways

1. Directly assigning a float value to a variable
2. Using a `float()` class.

Example

```python
# store a floating-point value
salary = 8000.456
print("Salary is :", salary)  # 8000.456
print(type(salary))  # class 'float'

# store a floating-point value using float() class
num = float(54.75)
print(num)  # 54.75
print(type(num))  # class 'float'
```

Floating-point values can be represented using the exponential form, also called **scientific notation.** The benefit of using the exponential form to represent floating-point values is we can represent large values using less memory.

Example

```python
# exponential float
num1 = 1.22e4
print(num1)  # 12200.0
print(type(num1))  # class 'float'
```

Complex data type

A complex number is a **number with a real and an imaginary component** represented as `a+bj` where `a` and `b` contain integers or floating-point values.

The complex type is generally used in scientific applications and electrical engineering applications. If we want to declare a complex value, then we can use the `a+bj` form. See the following example.

Example

```python
x = 9 + 8j  # both value are int type
y = 10 + 4.5j  # one int and one float
```

```python
z = 11.2 + 1.2j  # both value are float type
print(type(x))  # class 'complex'>

print(x)  # (9+8j)
print(y)  # (10+4.5j)
print(z)  # (11.2+1.2j)
```

The real part of the complex number is represented using an integer value. The integer value can be in the form of either decimal, float, binary, or hexadecimal. But **the imaginary part** should be represented using **the decimal** form only. If we are trying to represent an imaginary part as binary, hex, or octal, we will get an error.

List data type

The Python List is an **ordered collection (also known as a sequence) of elements**. List elements can be accessed, iterated, and removed according to the order they inserted at the creation time.

We use the list data type to represent groups of the element as a single entity. For example: If we want to store all student's names, we can use `list` type.

1. The list can contain data of all data types such as `int`, `float`, `string`
2. Duplicates elements are allowed in the list
3. The list is mutable which means we can modify the value of list elements

We can create a list using the two ways

1. By enclosing elements in the **square brackets** `[]`.
2. Using a `list()` class.

Example list creation and manipulation

```python
my_list = ["Jessa", "Kelly", 20, 35.75]
# display list
print(my_list)  # ['Jessa', 'Kelly', 20, 35.75]
print(type(my_list))  # class 'list'

# Accessing first element of list
print(my_list[0])  # 'Jessa'

# slicing list elements
print(my_list[1:5])  # ['Kelly', 20, 35.75]

# modify 2nd element of a list
my_list[1] = "Emma"
print(my_list[1])  # 'Emma'

# create list using a list class
my_list2 = list(["Jessa", "Kelly", 20, 35.75])
print(my_list2)  # ['Jessa', 'Kelly', 20, 35.75]
```

Tuple data type

Tuples are **ordered collections of elements that are unchangeab**le.
The `tuple` is the same as the `list`, except the tuple is immutable means we can't modify the tuple once created.

In other words, we can say a tuple is a read-only version of the list.

For example: If you want to store the roll numbers of students that you don't change, you can use the `tuple` data type.

Note: Tuple maintains the insertion order and also, allows us to store duplicate elements.

We can create a tuple using the two ways

1. By enclosing elements in the parenthesis ()
2. Using a `tuple()` class.

Example Tuple creation and manipulation

```python
# create a tuple
my_tuple = (11, 24, 56, 88, 78)
print(my_tuple)  # (11, 24, 56, 88, 78)
print(type(my_tuple))  # class 'tuple'

# Accessing 3rd element of a tuple
```

```python
print(my_tuple[2])  # 56

# slice a tuple
print(my_tuple[2:7])  # (56, 88, 78)

# create a tuple using a tuple() class
my_tuple2 = tuple((10, 20, 30, 40))
print(my_tuple2)  # (10, 20, 30, 40)
```

Tuple is immutable

A tuple is immutable means once we create a tuple, we can’t modify it

```python
# create a tuple
my_tuple = (11, 24, 56, 88, 78)

# modify 2nd element of tuple
my_tuple[1] = 35
print(my_tuple)
# TypeError: 'tuple' object does not support item assignment
```

Dict data type

In Python, dictionaries are **unordered collections of unique values stored in (Key-Value) pairs**. Use a dictionary data type to store data as a key-value pair.

The dictionary type is represented using a `dict` class. For example, If you want to store the name and roll number of all students, then you can use the `dict` type.

In a dictionary, duplicate keys are not allowed, but the value can be duplicated. If we try to insert a value with a duplicate key, the old value will be replaced with the new value.

Dictionary has some characteristics which are listed below:

1. A heterogeneous (i.e., `str`, `list`, `tuple`) elements are allowed for both key and value in a dictionary. But An object can be a key in a dictionary if it is hashable.
2. The dictionary is mutable which means we can modify its items
3. Dictionary is unordered so we can’t perform indexing and slicing

Example dictionary creation and manipulation

```
# create a dictionary
my_dict = {1: "Smith", 2: "Emma", 3: "Jessa"}

# display dictionary
print(my_dict)  # {1: 'Smith', 2: 'Emma', 3: 'Jessa'}
print(type(my_dict))  # class 'dict'

# create a dictionary using a dit class
my_dict = dict({1: "Smith", 2: "Emma", 3: "Jessa"})

# display dictionary
print(my_dict)  # {1: 'Smith', 2: 'Emma', 3: 'Jessa'}
print(type(my_dict))  # class 'dict'

# access value using a key name
print(my_dict[1])  # Smith

# change the value of a key
my_dict[1] = "Kelly"
print(my_dict[1])  # Kelly
```

Set data type

In Python, a set is an **unordered collection of data items that are unique**. In other words, Python Set is a collection of elements (Or objects) that contains no duplicate elements.

In Python, the Set data type used to represent a group of unique elements as a single entity. For example, If we want to store student ID numbers, we can use the set data type.

The Set data type in Python is represented using a `set` class.

We can create a Set using the two ways

1. By enclosing values in the curly brackets `{}`
2. Using a `set()` class.

The set data type has the following characteristics.

1. It is mutable which means we can change set items
2. Duplicate elements are not allowed
3. Heterogeneous (values of all data types) elements are allowed
4. Insertion order of elements is not preserved, so we can’t perform indexing on a Set

Example Set creation and manipulation

```
# create a set using curly brackets{,}
my_set = {100, 25.75, "Jessa"}
print(my_set)  # {25.75, 100, 'Jessa'}
print(type(my_set))  # class 'set'

# create a set using set class
my_set = set({100, 25.75, "Jessa"})
print(my_set)  # {25.75, 100, 'Jessa'}
print(type(my_set))  # class 'set'

# add element to set
my_set.add(300)
print(my_set)  # {25.75, 100, 'Jessa', 300}

# remove element from set
my_set.remove(100)
print(my_set)  # {25.75, 'Jessa', 300}
```

Bool data type

In Python, to **represent boolean values (`True` and `False`)** we use the `bool` data type. Boolean values are used to evaluate the value of the expression. For example, when we compare two values, the expression is evaluated, and Python returns the boolean `True` or `False`.

Example

```
x = 25
y = 20

z = x > y
print(z)  # True
print(type(z))  # class 'bool
```

Bytes data type

The `bytes` data type represents a group of byte numbers just like an array. We use the `bytes()` constructor to create bytes type, which also returns a bytes object. Bytes are **immutable** (Cannot be changed).

Use bytes data type if we want to handle binary data like images, videos, and audio files.

Example

```
a = [9, 14, 17, 11, 78]
b = bytes(a)
print(type(b))  # class 'bytes'
print(b[0])  # 9
print(b[-1])  # 78
```