Python - Data Types

Python Data Types

Python Data Types are used to define the type of a variable. It defines what type of data we are going to store in a variable. The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python has various built-in data types which we will discuss with in this tutorial:

- Numeric - int, float, complex
- String - str
- Sequence - list, tuple, range
- Binary - bytes, bytearray, memoryview
- Mapping - dict
- Boolean - bool
- Set - set, frozenset
- None - NoneType

Python Numeric Data Type

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

```
var1 = 1
var2 = 10
var3 = 10.023
```

Python supports four different numerical types –

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Examples

Here are some examples of numbers –

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

- A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

Example

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

Edit & Run

```
# integer variable.
a=100
print("The type of variable having value", a, " is ", type(a))

# float variable.
b=20.345
print("The type of variable having value", b, " is ", type(b))

# complex variable.
c=10+3j
print("The type of variable having value", c, " is ", type(c))
```

Python String Data Type

Python Strings are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. 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. For example –

```
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
```

Edit & Run

This will produce the following result –

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

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.

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. For example –

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
```

Edit & Run

```
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']
```

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.

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

```
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 ti
print (tuple[2:])           # Prints elements of the tuple starting from 3rd ele
print (tinytuple * 2)       # Prints the contents of the tuple twice
print (tuple + tinytuple)   # Prints concatenated tuples
```

Edit & Run

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 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
```

Python Ranges

Python **range()** is an in-built function in Python which returns a sequence of numbers starting from 0 and increments to 1 until it reaches a specified number.

We use **range()** function with for and while loop to generate a sequence of numbers. 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 starting position (It's mandatory)
- **step**: Integer number to specify increment, (Its optional, Default: 1)

Examples

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

```
for i in range(5):
  print(i)
```

Edit & Run

This produce the following result –

```
0
1
2
3
4
```

Now let's modify above program to print the number starting from 1 instead of 0:

```
for i in range(1, 5):
  print(i)
```

Edit & Run

This produce the following result –

```
1
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)
```

Edit & Run

This produce the following result –

```
1
3
```

Python Dictionary

Python dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. 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.

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

```
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
```

Edit & Run

This produce the following result –

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Python dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Python Boolean Data Types

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.

Examples

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))
```

Edit & Run

This 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))
```

Edit & Run

```
# Returns false as a is 10
a = 10
print(bool(a))
```

This produce the following result –

```
False
False
False
False
False
True
```

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.

Conversion to int

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

Edit & Run

```
a = int(1)     # a will be 1
b = int(2.2)   # b will be 2
c = int("3")   # c will be 3

print (a)
print (b)
print (c)
```

This produce the following result –

```
1
2
3
```

Conversion to float

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

Edit & Run

```
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
```

Conversion to string

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)
```

Edit & Run

This produce the following result –

```
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	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string.
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)** Converts s to a list.
10	**set(s)** Converts s to a set.
11	**dict(d)**

	Creates a dictionary. d must be a sequence of (key,value) tuples.
12	**frozenset(s)** Converts s to a frozen set.
13	**chr(x)** Converts an integer to a character.
14	**unichr(x)** Converts an integer to a Unicode character.
15	**ord(x)** Converts a single character to its integer value.
16	**hex(x)** Converts an integer to a hexadecimal string.
17	**oct(x)** Converts an integer to an octal string.