**Basic Data Types in Python**

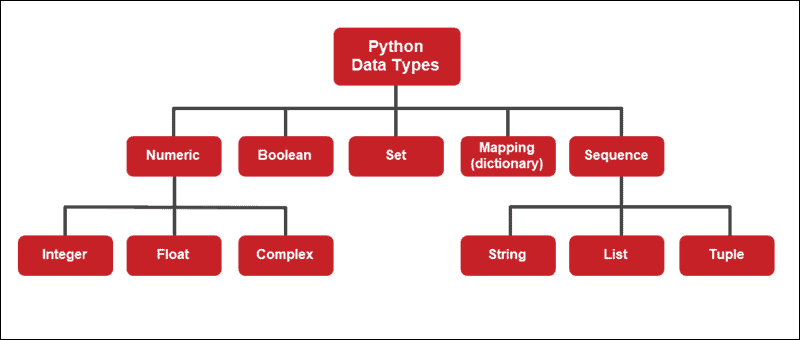
A **data type** is a characteristic that tells the compiler (or interpreter) how a programmer intends to use the data. There are two general categories of data types, differing whether the data is changeable after definition:

1. **Immutable**. Data types that are not changeable after assignment.

2. **Mutable**. Data types that are changeable after assignment.

**Note**: Variable IDs change when changing data for immutable types, whereas mutable types retain the same variable ID after the change. Check the variable ID with the built-in function **id(<variable>)**.

Variables store different types of data. Creating a variable of a particular data type creates an object of a data type class. The Python interpreter automatically assumes a type when creating a variable.



The data type of any object is found using the built-in **type()** function. The output shows the name of the class for the given object.

**Numeric Type**

Numeric type objects represent number values. They are divided into three subgroups:

* Integers
* Floating Point Numbers
* Complex Numbers

**Integers**

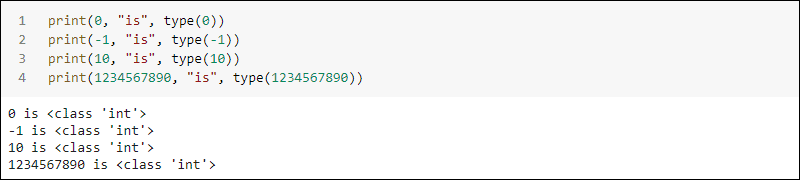
Integer values belong to the **int** class. Specifically, integers represent positive or negative whole numbers without a decimal. Some examples of integers include:

print(0, "is", type(0))

print(-1, "is", type(-1))

print(10, "is", type(10))

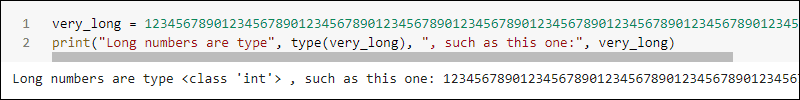
print(1234567890, "is", type(1234567890))



Integers have unlimited precision. There is no distinction between long and short numbers. The length depends on the computer memory:

very\_long = 1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890

print("Long numbers are type", type(very\_long), ", such as this one:", very\_long)

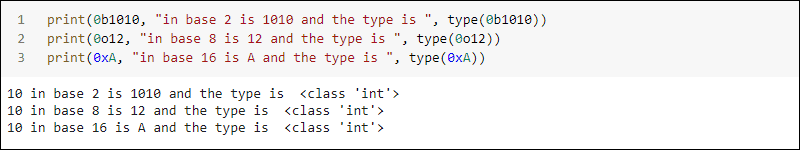


Binary, octal and hexadecimal number bases also evaluate to integers:

print(0b1010, "in base 2 is 1010 and the type is ", type(0b1010))

print(0o12, "in base 8 is 12 and the type is ", type(0o12))

print(0xA, "in base 16 is A and the type is ", type(0xA))



Printing the values converts to their base ten forms automatically.

**Note**: One key skill in writing code with Python is the ability to use comments. Comments are usually used to leave notes about the function of the code. However, they can also be used to disable parts of the code. Learn [how to comment in Python](https://phoenixnap.com/kb/how-to-use-comments-in-python) effectively.

**Floating-Point Numbers**

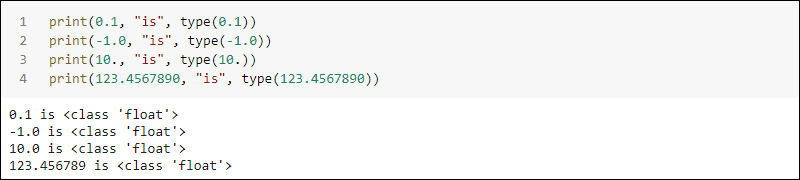
Floating-point numbers in Python are numbers defined with a decimal point. The class type is **float**. For instance:

print(0.1, "is", type(0.1))

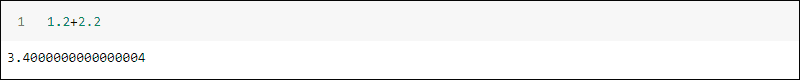
print(-1.0, "is", type(-1.0))

print(10., "is", type(10.))

print(123.4567890, "is", type(123.4567890))



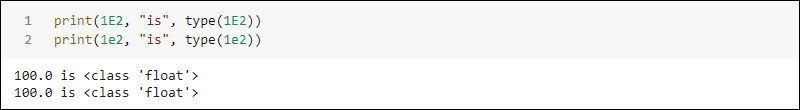
Internally, they are binary fractions, which means the number approximates the decimal counterpart. The difference between the real and the approximated value is often unnoticeable. Rounding the value yields the exact number. For example:



Alternatively, a number with the character *E* followed by a number indicates scientific notation:

print(1E2, "is", type(1E2))

print(1e2, "is", type(1e2))



Scientific notation belongs to the floating-point class of numbers as well. The syntax accepts both lowercase *e* as well as uppercase *E*.

Floating-point numbers after 1.79×10308 evaluate to infinity. The smallest non-zero number is 5.0×10-324. Smaller numbers evaluate to zero:



The values correspond to 64-bit double-precision values.

**Complex Numbers**

Complex numbers are often used in mathematical sciences. Python provides a class for complex numbers called **complex**. To write complex numbers, use:

<real part> + <complex part>j

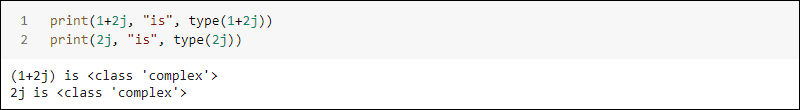
Alternatively, you can omit the real part:

<complex part>j

For example:

print(1+2j, "is", type(1 + 2j))

print(2j, "is", type(2j))



The output shows the data belongs to the **complex** class.

**Note**: Check out our [SciPy tutorial](https://phoenixnap.com/kb/scipy-tutorial" \t "_blank) to learn about scientific computing in Python using the SciPy library.

**Sequence Type**

Sequence types help represent sequential data stored into a single variable. There are three types of sequences used in Python:

* String
* List
* Tuple

Individual sequence types differ from one another when it comes to changeability and order.

**Strings**

Strings are a sequence of bytes representing Unicode characters. The string type in Python is called **str**.

**Create a String**

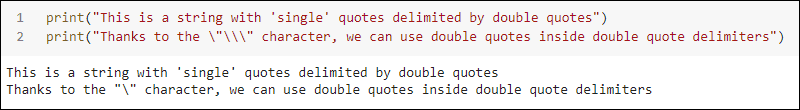
Depending on the use case and characters needed, there are four different ways to create a string. They differ based on the delimiters and whether a string is single or multiline.

1. Create a string by using **double quote delimiters**:

print("This is a string with 'single' quotes delimited by double quotes")

A string delimited with double quotes is helpful for strings containing single quotes or apostrophes. Alternatively, you can use escape characters:

print("Thanks to the \"\\\" character, we can use double quotes inside double quote delimiters")



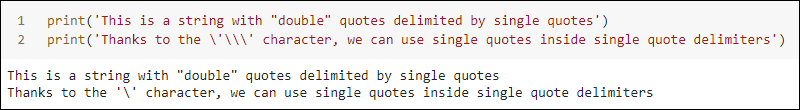
The example additionally shows how to display the backslash character in a string.

2. Create a string by using **single quote delimiters**:

print('This is a string with "double" quotes delimited by single quotes')

Python strings delimited by single quotes are valid when a string contains a double quote. Another way is to use escape characters. For instance:

print('Thanks to the \'\\\' character, we can use single quotes inside single quote delimiters')



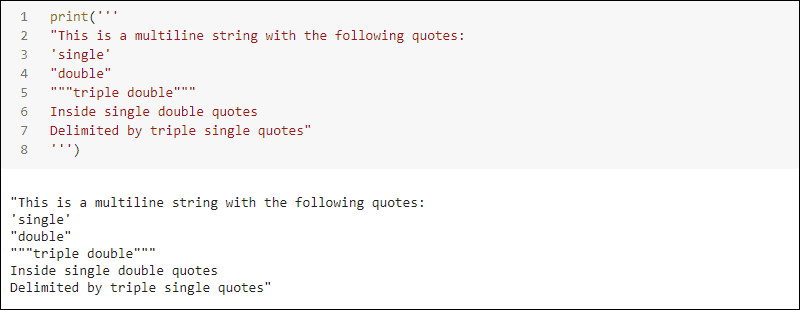
There are two ways to delimit a multiline string.

a) Create a multiline string by using **triple single quote delimiters**:

print('''This is a multiline string

with 'single', "double" and """triple-double""" quotes

delimited with triple single quotes''')



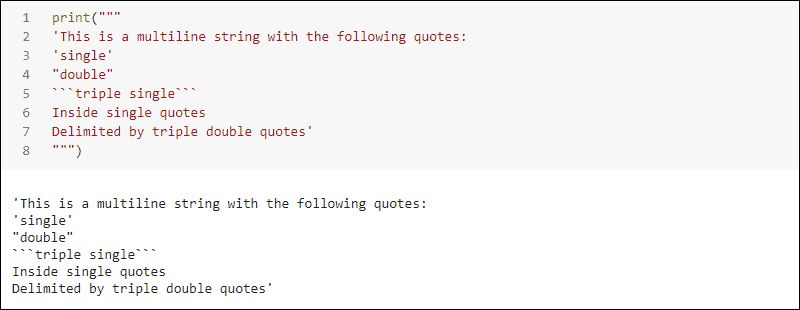
Use the triple single quotes to delimit a string if it contains both single, double, triple double quotes or ends in a double quote.

b) Create a string by using **triple double quote delimiters**:

print("""'This is a multiline string

with 'single', "double" and ```triple single``` quotes

delimited with triple double quotes'""")



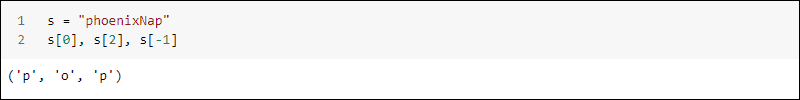
Use the triple double quote delimiter for strings containing single, double, triple single quotes or strings ending in a single quote.

**Access Elements of a String**

Strings in Python are arrays of characters. To access individual elements, use indexing:

s = "phoenixNap"

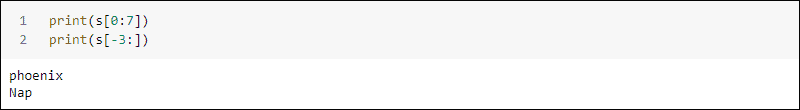
s[0], s[2], s[-1]



To access parts of a string, use slicing:

print(s[0:7])

print(s[-3:])

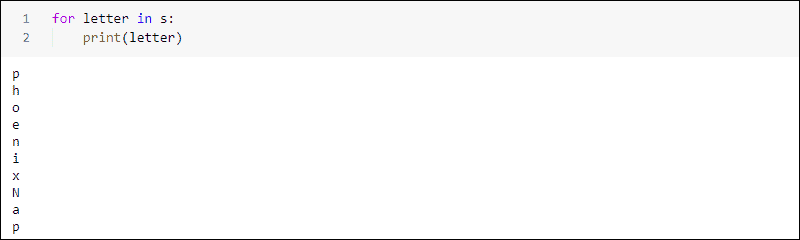


Access the first element at index 0. Counting backward starting from -1 accesses the end of a sequence.

Since strings are arrays, you can loop through all the characters by using a **for** loop:

for letter in s:

print(letter)



The code prints all letters of a string one by one.

**Lists**

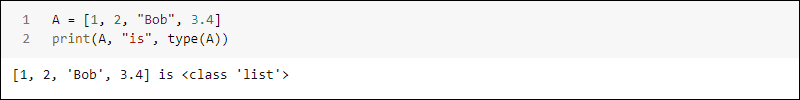
A Python list is an ordered changeable array. Lists allow duplicate elements regardless of their type. Adding or removing members from a list allows changes after creation.

**Create a List**

Create a list in Python by using *square brackets*, separating individual elements with a comma:

A = [1, 2, "Bob", 3.4]

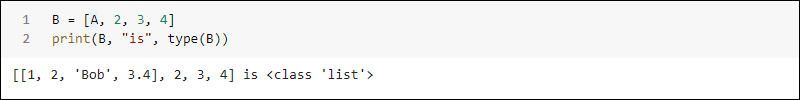
print(A, "is", type(A))



Make a *nested list* by adding a list to a list:

B = [A, 2, 3, 4]

print(B, "is", type(B))

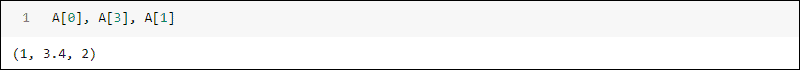


Since Python lists are changeable, they allow creating empty lists and appending elements later, as well as adding or removing members to an existing list.

**Access Elements of a List**

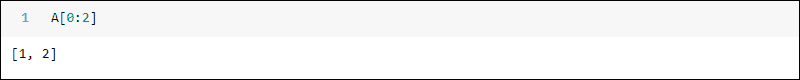
Lists are a sequence of elements. Access members by using indexing notation, where the first element is at index 0:

A[0], A[3], A[1]



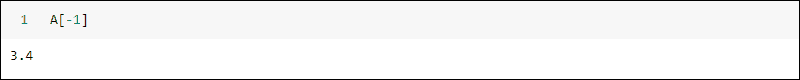
Slicing a list returns all the elements between two indexes:

A[0:2]



Negative indexes are also possible:

A[-1]



The **-1** index prints the last element in the list. Negative indexing is especially useful for navigating to the end of a long list of members.

**Tuples**

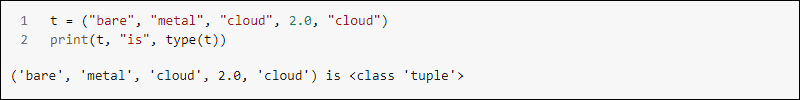
Python Tuples are an array of unchangeable ordered elements. Once a tuple is stored into a variable, members cannot be added or removed. A tuple allows duplicate members of any type.

**Create a Tuple**

To create a tuple, use the standard round brackets, separating individual elements with a comma:

t = ("bare", "metal", "cloud", 2.0, "cloud")

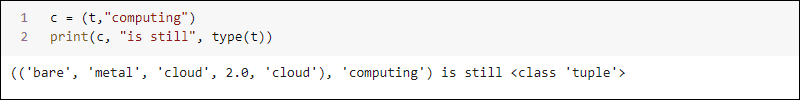
print(t, "is", type(t))



Make a nested tuple by adding a tuple to a tuple:

c = (t, "computing")

print(c, "is still", type(t))

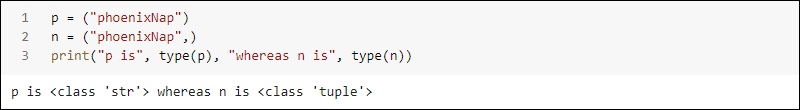


To create a tuple with a single element, use a comma after the first element:

p = ("phoenixNap")

n = ("phoenixNap",)

print("p is", type(p), "whereas n is", type(n))



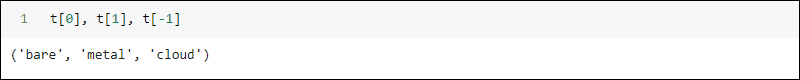
Without a comma, the variable is a string.

Create an empty tuple using the round brackets without elements. Although it seems redundant since tuples are unchangeable, an empty tuple helps indicate a lack of data in certain use cases.

**Access Elements of a Tuple**

Tuples support indexing notation. Access individual elements by using square brackets and the index of the element:

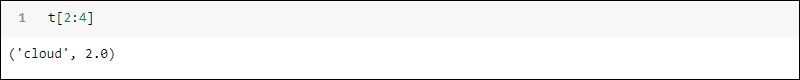
t[0], t[1], t[-1]



Negative indexing allows accessing elements at the end of the list.

To access parts of a tuple, use slicing notation:

t[2:4]



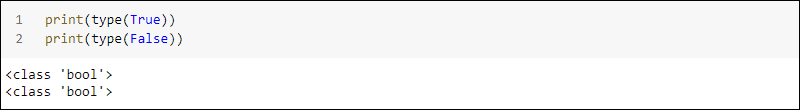
The output shows the third and fourth elements of the tuple.

**Boolean Type**

Boolean data types belong to the **bool** class and determine the truth value of expressions. Objects of the Boolean type evaluate either to **True** or **False**:

print(type(True))

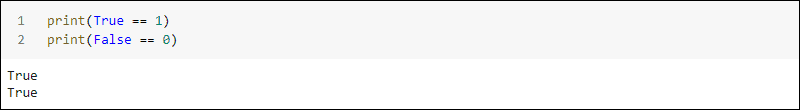
print(type(False))



Booleans are a subtype of integer values. Checking the truth value of integers 1 and 0 with True and False returns true in both cases:

print(True == 1)

print(False == 0)



The data type of the values is different. True and False are both Boolean types, whereas 1 and 0 are integer types.

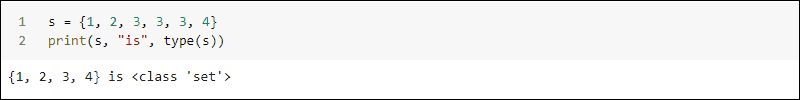
**Set Type**

The Set data type is part of the **set** class. It stores data collections into a single variable. Sets are unordered and do not allow access to individual elements through indexing. Any duplicate values are ignored.

To create a set, use the curly brackets notation and separate individual elements with a comma:

s = {1, 2, 3, 3, 3, 4}

print(s, "is", type(s))



Notice the multiple instances of data disappear.

**Mapping Type**

Mapping data type is represented by a Python dictionary. A dictionary is a collection of data with key and value pairs belonging to the **dict** class.

To create a dictionary, use the curly bracket notation and define the key value pairs. For example:

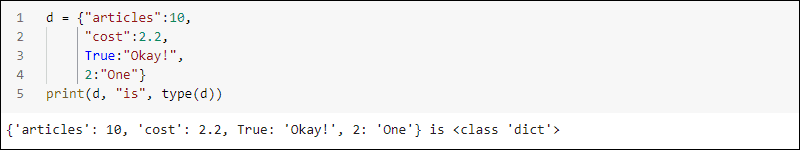
d = {"articles":10,

"cost":2.2,

True:"Okay!",

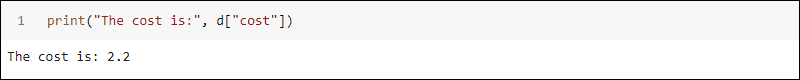
2:"One"}

print(d, "is", type(d))



The key and value pairs accept any data type. To access a value in a dictionary, use the key as index:

print("The cost is:", d["cost"])



Dictionaries come in handy when storing linked data pairs.

**Managing Data Types in Python**

When writing lengthy and complex programs, managing different data types becomes a crucial aspect of tackling programming problems. It is impossible to predict the needed data type flawlessly. However, knowing how to check or change a data type is essential to programming.

**Check Data Type**

Every variable has a data type, which is checked with the built-in function **type()**:

print(type(1))

print(type(1.))

print(type(1+0j))

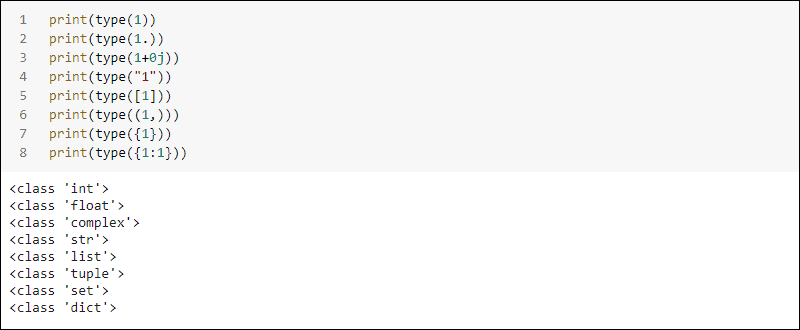
print(type("1"))

print(type([1]))

print(type((1,)))

print(type({1}))

print(type({1:1}))



The example above shows several ways to write the number 1 using various data types. The function **type()** also works on variables.

**Set Data Type**

The data type in Python is set automatically when writing a value to a variable. The class constructor for each data type allows setting the specific data type of a variable as well:

|  |  |
| --- | --- |
| **Data Type** | **Constructor** |
| String | **str(<value>)** |
| Integer | **int(<value>)** |
| Floating-point | **float(<value>)** |
| Complex | **complex(<value>)** |
| List | **list((<value>, <value>))** |
| Tuple | **tuple((<value>, <value>))** |
| Boolean | **bool(<value)** |
| Set | **set((<value>, <value>))** |
| Dictionary | **dict((<key>=<value>, <key>=<value>))** |

**Convert Data Type**

Some Python data types are convertible to other data types. There are two ways to convert a variable type:

* **Explicitly**. The class constructor for data types converts data types as well.
* **Implicitly**. When possible, the Python interpreter automatically converts a data type - for example, adding an integer to a float yields a float result.

**Note:** Python supports various methods for analyzing and resolving unaccounted data. Learn more in our guide [Handling Missing Data in Python: Causes and Solutions](https://phoenixnap.com/kb/handling-missing-data-in-python).

**Conclusion**

Python provides many built-in data types depending on the use and function of the data stored. Since data types are classes, creating new data types is not a complicated process.

When programming in Python, both [IDEs and Code Editors](https://phoenixnap.com/kb/best-python-ide-code-editor) are useful. Knowing the difference between all the available options and when to use them is crucial for efficient development.

For further reading, learn about the [MySQL data types](https://phoenixnap.com/kb/mysql-data-types).