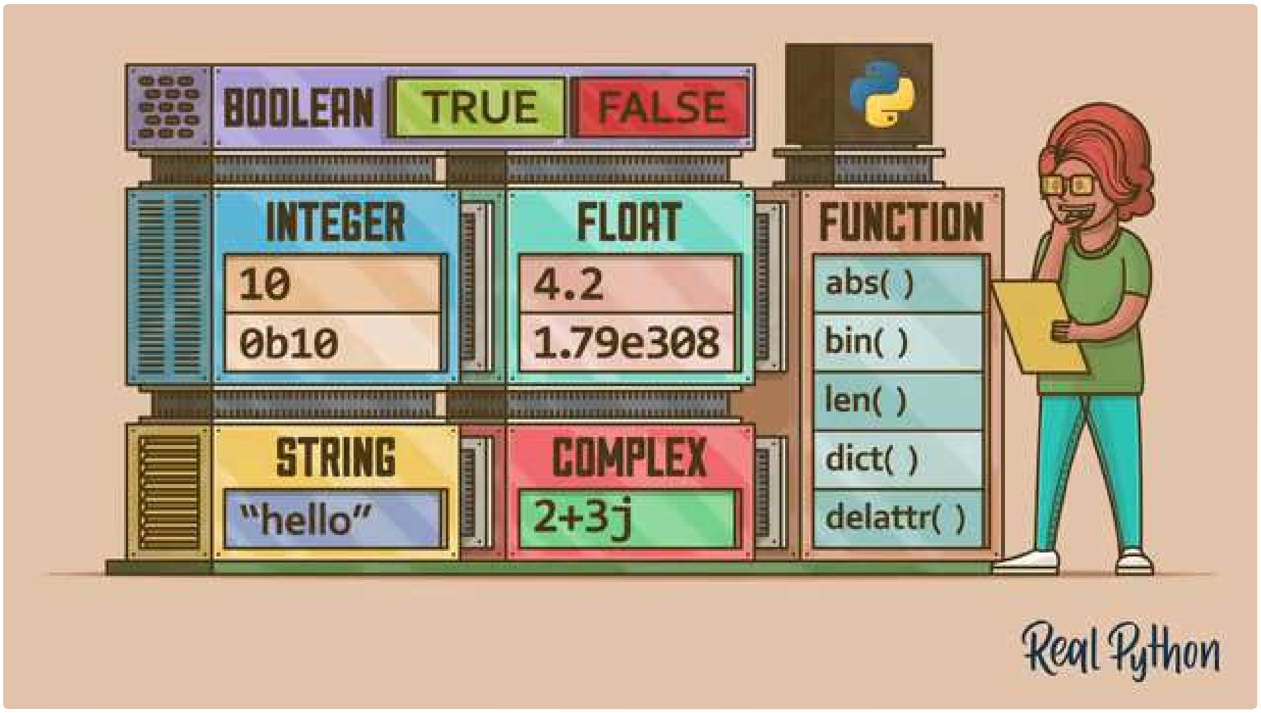


Basic Data Types in Python

by John Sturtz Jun 05, 2018 15 Comments basics python

Mark as Completed

Share Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Basic Data Types in Python**

Now you know how to interact with the Python interpreter and execute Python code. It's time to dig into the Python language. First up is a discussion of the basic data types that are built into Python.

Here's what you'll learn in this tutorial:

- You'll learn about several basic **numeric, string**, and **Boolean** types that are built into Python. By the end of this tutorial, you'll be familiar with what objects of these types look like, and how to represent them.
- You'll also get an overview of Python's built-in **functions.** These are pre-written chunks of code you can call to do useful things. You have already seen the built-in `print()` function, but there are many others.

Free PDF Download: Python 3 Cheat Sheet

Take the Quiz: Test your knowledge with our interactive "Basic Data Types in Python" quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

Integers

In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

Python

```
>>> print(123123123123123123123123123123123123123123123123 + 1)
123123123123123123123123123123123123123123123124
```

Python interprets a sequence of decimal digits without any prefix to be a decimal number:

Python

```
>>> print(10)
10
```

The following strings can be prepended to an integer value to indicate a base other than 10:

Prefix	**Interpretation**	**Base**
`0b` (zero + lowercase letter `'b'`) `0B` (zero + uppercase letter `'B'`)	Binary	2
`0o` (zero + lowercase letter `'o'`) `0O` (zero + uppercase letter `'O'`)	Octal	8
`0x` (zero + lowercase letter `'x'`) `0X` (zero + uppercase letter `'X'`)	Hexadecimal	16

For example:

Python

```
>>> print(0o10)
8

>>> print(0x10)
16

>>> print(0b10)
2
```

For more information on integer values with non-decimal bases, see the following Wikipedia sites: Binary, Octal, and Hexadecimal.

The underlying type of a Python integer, irrespective of the base used to specify it, is called `int`:

Python

```
>>> type(10)
<class 'int'>
>>> type(0o10)
<class 'int'>
>>> type(0x10)
<class 'int'>
```

Note: This is a good time to mention that if you want to display a value while in a REPL session, you don't need to use the `print()` function. Just typing the value at the `>>>` prompt and hitting `Enter ↩` will display it:

Python

```
>>> 10
10
>>> 0x10
16
>>> 0b10
2
```

Many of the examples in this tutorial series will use this feature.

Note that this does not work inside a script file. A value appearing on a line by itself in a script file will not do anything.

Floating-Point Numbers

The `float` type in Python designates a floating-point number. `float` values are specified with a decimal point. Optionally, the character `e` or `E` followed by a positive or negative integer may be appended to specify scientific notation:

Python

```
>>> 4.2
4.2
>>> type(4.2)
<class 'float'>
>>> 4.
4.0
>>> .2
0.2

>>> .4e7
4000000.0
>>> type(.4e7)
<class 'float'>
>>> 4.2e-4
0.00042
```

Deep Dive: Floating-Point Representation

The following is a bit more in-depth information on how Python represents floating-point numbers internally. You can readily use floating-point numbers in Python without understanding them to this level, so don't worry if this seems overly complicated. The information is presented here in case you are curious.

Almost all platforms represent Python `float` values as 64-bit "double-precision" values, according to the IEEE 754 standard. In that case, the maximum value a floating-point number can have is approximately 1.8×10^{308}. Python will indicate a number greater than that by the string `inf`:

Python

```
>>> 1.79e308
1.79e+308
>>> 1.8e308
inf
```

The closest a nonzero number can be to zero is approximately 5.0×10^{-324}. Anything closer to zero than that is effectively zero:

Python

```
>>> 5e-324
5e-324
>>> 1e-325
0.0
```

Floating point numbers are represented internally as binary (base-2) fractions. Most decimal fractions cannot be represented exactly as binary fractions, so in most cases the internal representation of a floating-point number is an approximation of the actual value. In practice, the difference between the actual value and the represented value is very small and should not usually cause significant problems.

Further Reading: For additional information on floating-point representation in Python and the potential pitfalls involved, see Floating Point Arithmetic: Issues and Limitations in the Python documentation.

Complex Numbers

Complex numbers are specified as `<real part>+<imaginary part>j`. For example:

Python

```
>>> 2+3j
(2+3j)
>>> type(2+3j)
<class 'complex'>
```

Strings

Strings are sequences of character data. The string type in Python is called `str`.

String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

Python

```
>>> print("I am a string.")
I am a string.
>>> type("I am a string.")
<class 'str'>

>>> print('I am too.')
I am too.
>>> type('I am too.')
<class 'str'>
```

A string in Python can contain as many characters as you wish. The only limit is your machine's memory resources. A string can also be empty:

Python

```
>>> ''
''
```

What if you want to include a quote character as part of the string itself? Your first impulse might be to try something like this:

Python

```
>>> print('This string contains a single quote (') character.')
SyntaxError: invalid syntax
```

As you can see, that doesn't work so well. The string in this example opens with a single quote, so Python assumes the next single quote, the one in parentheses which was intended to be part of the string, is the closing delimiter. The final single quote is then a stray and causes the syntax error shown.

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

Python

```
>>> print("This string contains a single quote (') character.")
This string contains a single quote (') character.

>>> print('This string contains a double quote (") character.')
This string contains a double quote (") character.
```

Escape Sequences in Strings

Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways:

- You may want to suppress the special interpretation that certain characters are usually given within a string.
- You may want to apply special interpretation to characters in a string which would normally be taken literally.

You can accomplish this using a backslash (\) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially. (This is referred to as an escape sequence, because the backslash causes the subsequent character sequence to "escape" its usual meaning.)

Let's see how this works.

Suppressing Special Character Meaning

You have already seen the problems you can come up against when you try to include quote characters in a string. If a string is delimited by single quotes, you can't directly specify a single quote character as part of the string because, for that string, the single quote has special meaning—it terminates the string:

Python

```
>>> print('This string contains a single quote (') character.')
SyntaxError: invalid syntax
```

Specifying a backslash in front of the quote character in a string "escapes" it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

Python

```
>>> print('This string contains a single quote (\') character.')
This string contains a single quote (') character.
```

The same works in a string delimited by double quotes as well:

Python

```
>>> print("This string contains a double quote (\") character.")
This string contains a double quote (") character.
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

Escape Sequence	Usual Interpretation of Character(s) After Backslash	"Escaped" Interpretation
`\'`	Terminates string with single quote opening delimiter	Literal single quote (`'`) character
`\"`	Terminates string with double quote opening delimiter	Literal double quote (`"`) character
`\<newline>`	Terminates input line	Newline is ignored
`\\`	Introduces escape sequence	Literal backslash (`\`) character

Ordinarily, a newline character terminates line input. So pressing `Enter ↩` in the middle of a string will cause Python to think it is incomplete:

Python

```
>>> print('a

SyntaxError: EOL while scanning string literal
```

To break up a string over more than one line, include a backslash before each newline, and the newlines will be ignored:

Python

```
>>> print('a\
... b\
... c')
abc
```

To include a literal backslash in a string, escape it with a backslash:

Python

```
>>> print('foo\\bar')
foo\bar
```

Applying Special Meaning to Characters

Next, suppose you need to create a string that contains a tab character in it. Some text editors may allow you to insert a tab character directly into your code. But many programmers consider that poor practice, for several reasons:

- The computer can distinguish between a tab character and a sequence of space characters, but you can't. To a human reading the code, tab and space characters are visually indistinguishable.
- Some text editors are configured to automatically eliminate tab characters by expanding them to the appropriate number of spaces.
- Some Python REPL environments will not insert tabs into code.

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence `\t`:

Python

```
>>> print('foo\tbar')
foo     bar
```

The escape sequence `\t` causes the `t` character to lose its usual meaning, that of a literal `t`. Instead, the combination is interpreted as a tab character.

Here is a list of escape sequences that cause Python to apply special meaning instead of interpreting literally:

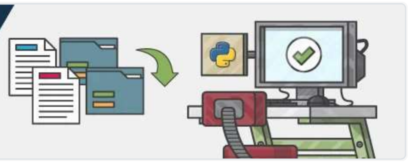
Escape Sequence	**"Escaped" Interpretation**
`\a`	ASCII Bell (`BEL`) character
`\b`	ASCII Backspace (`BS`) character
`\f`	ASCII Formfeed (`FF`) character
`\n`	ASCII Linefeed (`LF`) character
`\N{<name>}`	Character from Unicode database with given `<name>`
`\r`	ASCII Carriage Return (`CR`) character
`\t`	ASCII Horizontal Tab (`TAB`) character
`\uxxxx`	Unicode character with 16-bit hex value `xxxx`
`\Uxxxxxxxx`	Unicode character with 32-bit hex value `xxxxxxxx`
`\v`	ASCII Vertical Tab (`VT`) character
`\ooo`	Character with octal value `ooo`
`\xhh`	Character with hex value `hh`

Examples:

Python

```
>>> print("a\tb")
a    b
>>> print("a\141\x61")
aaa
>>> print("a\nb")
a
b
>>> print('\u2192 \N{rightwards arrow}')
→ →
```

This type of escape sequence is typically used to insert characters that are not readily generated from the keyboard or are not easily readable or printable.



Raw Strings

A raw string literal is preceded by `r` or `R`, which specifies that escape sequences in the associated string are not translated. The backslash character is left in the string:

Python
```
>>> print('foo\nbar')
foo
bar
>>> print(r'foo\nbar')
foo\nbar

>>> print('foo\\bar')
foo\bar
>>> print(R'foo\\bar')
foo\\bar
```

Triple-Quoted Strings

There is yet another way of delimiting strings in Python. Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes. Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them. This provides a convenient way to create a string with both single and double quotes in it:

Python
```
>>> print('''This string has a single (') and a double (") quote.''')
This string has a single (') and a double (") quote.
```

Because newlines can be included without escaping them, this also allows for multiline strings:

Python
```
>>> print("""This is a
string that spans
across several lines""")
This is a
string that spans
across several lines
```

You will see in the upcoming tutorial on Python Program Structure how triple-quoted strings can be used to add an explanatory comment to Python code.

Boolean Type, Boolean Context, and “Truthiness”

Python 3 provides a Boolean data type. Objects of Boolean type may have one of two values, `True` or `False`:

Python
```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

As you will see in upcoming tutorials, expressions in Python are often evaluated in Boolean context, meaning they are interpreted to represent truth or falsehood. A value that is true in Boolean context is sometimes said to be "truthy," and one that is false in Boolean context is said to be "falsy." (You may also see "falsy" spelled "falsey.")

The "truthiness" of an object of Boolean type is self-evident: Boolean objects that are equal to `True` are truthy (true), and those equal to `False` are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false.

You will learn more about evaluation of objects in Boolean context when you encounter logical operators in the upcoming tutorial on operators and expressions in Python.

Built-In Functions

The Python interpreter supports many functions that are built-in: sixty-eight, as of Python 3.6. You will cover many of these in the following discussions, as they come up in context.

For now, a brief overview follows, just to give a feel for what is available. See the Python documentation on built-in functions for more detail. Many of the following descriptions refer to topics and concepts that will be discussed in future tutorials.

Math

Function	Description
`abs()`	Returns absolute value of a number
`divmod()`	Returns quotient and remainder of integer division
`max()`	Returns the largest of the given arguments or items in an iterable
`min()`	Returns the smallest of the given arguments or items in an iterable
`pow()`	Raises a number to a power
`round()`	Rounds a floating-point value
`sum()`	Sums the items of an iterable

Type Conversion

Function	Description
`ascii()`	Returns a string containing a printable representation of an object
`bin()`	Converts an integer to a binary string
`bool()`	Converts an argument to a Boolean value
`chr()`	Returns string representation of character given by integer argument
`complex()`	Returns a complex number constructed from arguments

Function	Description
`float()`	Returns a floating-point object constructed from a number or string
`hex()`	Converts an integer to a hexadecimal string
`int()`	Returns an integer object constructed from a number or string
`oct()`	Converts an integer to an octal string
`ord()`	Returns integer representation of a character
`repr()`	Returns a string containing a printable representation of an object
`str()`	Returns a string version of an object
`type()`	Returns the type of an object or creates a new type object

Iterables and Iterators

Function	Description
`all()`	Returns `True` if all elements of an iterable are true
`any()`	Returns `True` if any elements of an iterable are true
`enumerate()`	Returns a list of tuples containing indices and values from an iterable
`filter()`	Filters elements from an iterable
`iter()`	Returns an iterator object
`len()`	Returns the length of an object
`map()`	Applies a function to every item of an iterable
`next()`	Retrieves the next item from an iterator
`range()`	Generates a range of integer values
`reversed()`	Returns a reverse iterator
`slice()`	Returns a `slice` object
`sorted()`	Returns a sorted list from an iterable
`zip()`	Creates an iterator that aggregates elements from iterables

Composite Data Type

Function	Description
`bytearray()`	Creates and returns an object of the `bytearray` class

Function	Description
`bytes()`	Creates and returns a `bytes` object (similar to `bytearray`, but immutable)
`dict()`	Creates a `dict` object
`frozenset()`	Creates a `frozenset` object
`list()`	Creates a `list` object
`object()`	Creates a new featureless object
`set()`	Creates a `set` object
`tuple()`	Creates a `tuple` object

Classes, Attributes, and Inheritance

Function	Description
`classmethod()`	Returns a class method for a function
`delattr()`	Deletes an attribute from an object
`getattr()`	Returns the value of a named attribute of an object
`hasattr()`	Returns `True` if an object has a given attribute
`isinstance()`	Determines whether an object is an instance of a given class
`issubclass()`	Determines whether a class is a subclass of a given class
`property()`	Returns a property value of a class
`setattr()`	Sets the value of a named attribute of an object
`super()`	Returns a proxy object that delegates method calls to a parent or sibling class

Input/Output

Function	Description
`format()`	Converts a value to a formatted representation
`input()`	Reads input from the console
`open()`	Opens a file and returns a file object
`print()`	Prints to a text stream or the console

Variables, References, and Scope

Function	Description
`dir()`	Returns a list of names in current local scope or a list of object attributes
`globals()`	Returns a dictionary representing the current global symbol table
`id()`	Returns the identity of an object
`locals()`	Updates and returns a dictionary representing current local symbol table
`vars()`	Returns `__dict__` attribute for a module, class, or object

Miscellaneous

Function	Description
`callable()`	Returns `True` if object appears callable
`compile()`	Compiles source into a code or AST object
`eval()`	Evaluates a Python expression
`exec()`	Implements dynamic execution of Python code
`hash()`	Returns the hash value of an object
`help()`	Invokes the built-in help system
`memoryview()`	Returns a memory view object
`staticmethod()`	Returns a static method for a function
`__import__()`	Invoked by the `import` statement

Conclusion

In this tutorial, you learned about the built-in **data types** and **functions** Python provides.

The examples given so far have all manipulated and displayed only constant values. In most programs, you are usually going to want to create objects that change in value as the program executes.

Head to the next tutorial to learn about Python **variables.**

Take the Quiz: Test your knowledge with our interactive "Basic Data Types in Python" quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

« Interacting with Python

Basic Data Types in Python

Variables in Python »