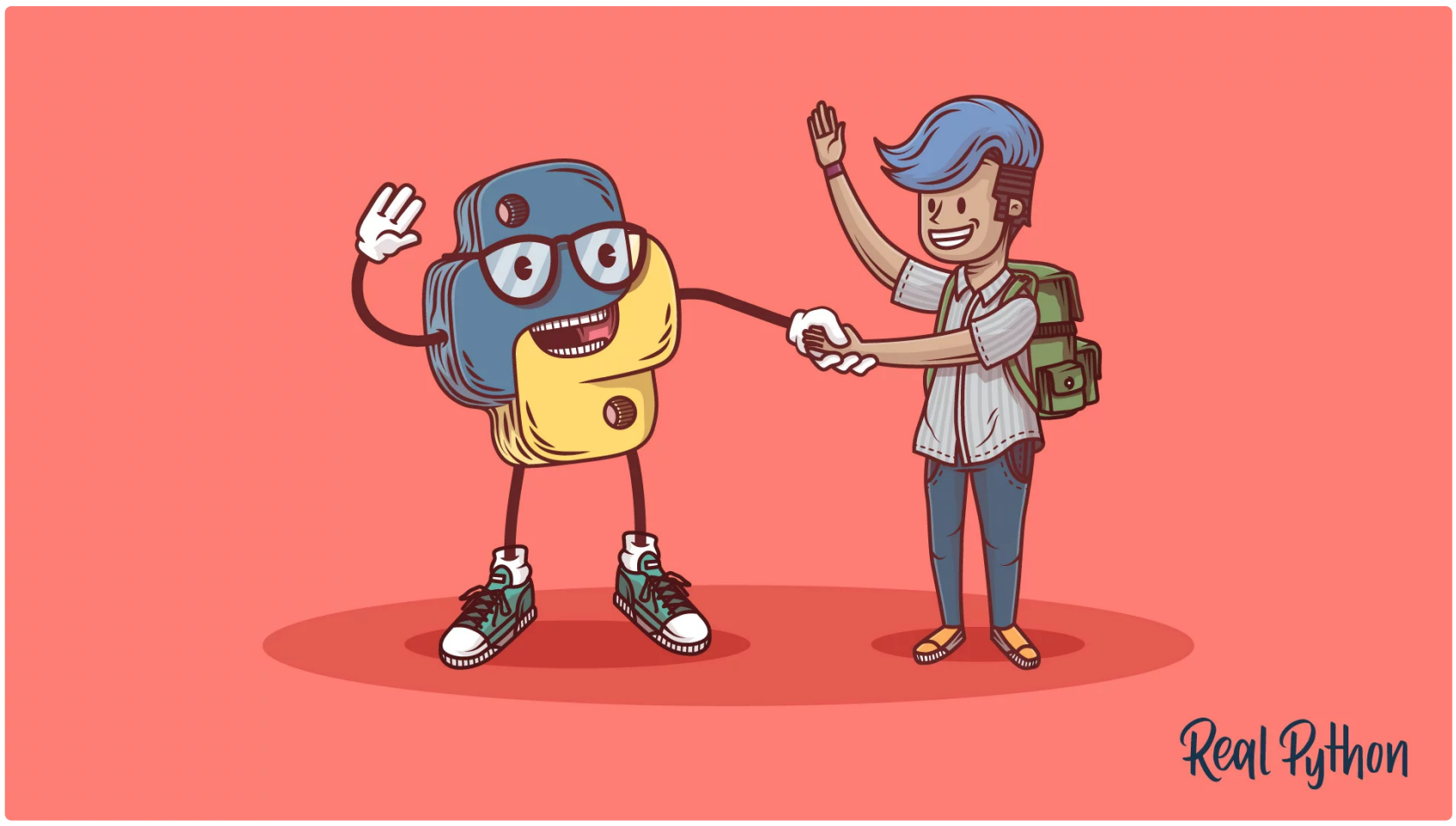

Basic Data Types in Python

by John Sturtz 9 Comments basics python

Tweet Share Email

Table of Contents

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

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.


```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Integers

In Python 3, there is effectively no limit t
of memory your system has, as are all th

Python

```
>>> print(123123123123123123123123123123123123123123123123 + 1)
123123123123123123123123123123123123123123123123124
```

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
to use the `print()` function. Just typi

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 floatin

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 represen
be represented exactly as binary frac
number is an approximation of the a
represented value is very small and s

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Further Reading: For additional info
pitfalls involved, see Floating Point A

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 w
string can also be empty:

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 si
SyntaxError: invalid syntax
```

As you can see, that doesn't work so wel
the next single quote, the one in parenth
final single quote is then a stray and cau

If you want to include either type of quo
other type. If a string is to contain a sing

Python

```
>>> print("This string contains a si
This string contains a single quote

>>> print('This string contains a dc
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 s	
`\"`	Terminates string with c	
`\newline`	Terminates input line	
`\\`	Introduces escape sequ	

Ordinarily, a newline character terminat to think it is incomplete:

Python

```
>>> print('a

SyntaxError: EOL while scanning stri
```

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
`\n`	ASCII
`\N{<name>}`	Char
`\r`	ASCII
`\t`	ASCII
`\uxxxx`	Unic
`\Uxxxxxxxx`	Unic
`\v`	ASCII Vertical Tab (VT) character
`\oxx`	Character with octal value `xx`
`\xhh`	Character with hex value `hh`

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


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 st
three single quotes or three double quot
double quotes, and newlines can be incl
string with both single and double quote

Python

```
>>> print('''This string has a singl
This string has a single (') and a c
```

Because newlines can be included withc

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 val
`divmod()`	Returns quotient an
`max()`	Returns the largest c
`min()`	Returns the smallest
`pow()`	Raises a number to a
`round()`	Rounds a floating-pc
`sum()`	Sums the items of ar

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


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
`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 tu
`filter()`	Filters elements fr
`iter()`	Returns an iterato
`len()`	Returns the lengtl
`map()`	Applies a function
`next()`	Retrieves the next
`range()`	Generates a range
`reversed()`	Returns a reverse
`slice()`	Returns a `slice` object
`sorted()`	Returns a sorted list from an iterable
`zip()`	Creates an iterator that aggregates elements from iterables

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Composite Data Type

Function	Description
`bytearray()`	Creates and returns an object of the `bytearray` class
`bytes()`	Creates and returns a `bytes` object (similar to `bytearray`, but immutable)
`dict()`	Creates a `dict` object
`frozenset()`	Creates a `frozenset` object
`list()`	Constructs a `list` object
`object()`	Returns 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 functio

Function	Description
`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 propert
`setattr()`	Sets the value of a
`super()`	Returns a proxy o

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Input/Output

Function	Description
`format()`	Converts a valu
`input()`	Reads input fro
`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 s

Function	Description
`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 bu

The examples given so far have all mani
usually going to want to create objects t

Head to the next tutorial to learn about

🎓 **Take the Quiz:** Test your knowled
completion you will receive a score sc

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


« Interacting with Python Basic Data Types in Python Variables in Python »

About **John Sturtz**

John is an avid Pythonista and a member of the Real Python tutorial team.

» More about John

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Dan

Jon

Joanna

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

Keep Learning

Related Tutorial Categories: basics python

All Tutorial Topics

advanced | api | basics | best-practices | community | databases | data-science | devops | django | docker | flask | front-end | intermediate | machine-learning | python | testing | tools | web-dev | web-scraping

Table of Contents