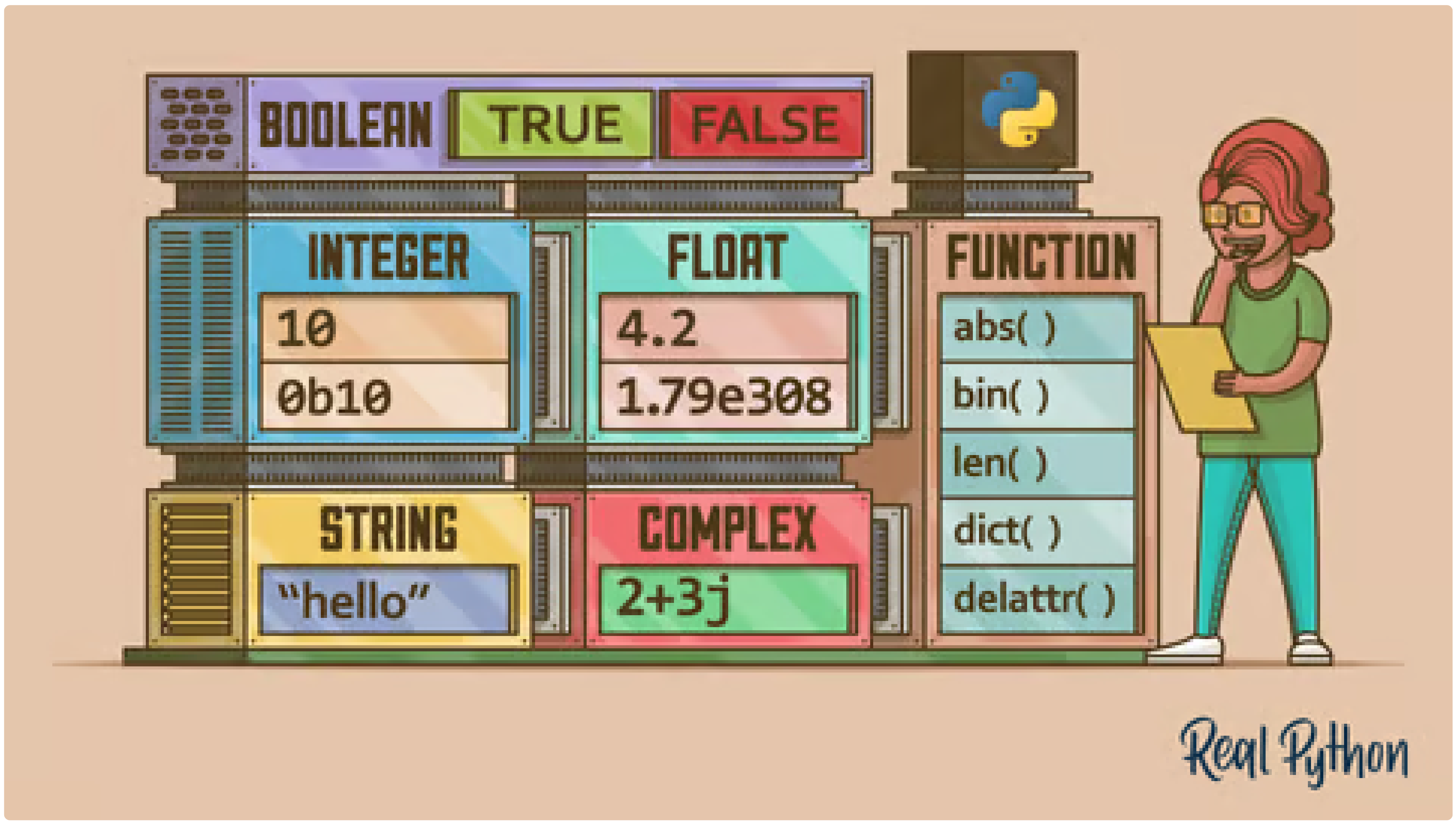


Basic Data Types in Python: A Quick Exploration

by Leodanis Pozo Ramos

Dec 21, 2024 52m 15 Comments basics python

Mark as Completed

Share

Table of Contents

Help

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: **Exploring Basic Data Types in Python**

Python data types are fundamental to the language, enabling you to represent various kinds of data. You use basic data types like **`int`**, **`float`**, and **`complex`** for numbers, **`str`** for text, **`bytes`** and **`bytearray`** for binary data, and **`bool`** for Boolean values. These data types form the core of most Python programs, allowing you to handle numeric, textual, and logical data efficiently.

Understanding Python data types involves recognizing their roles and how to work with them. You can create and manipulate these data types using built-in functions and methods, and convert between them when necessary.

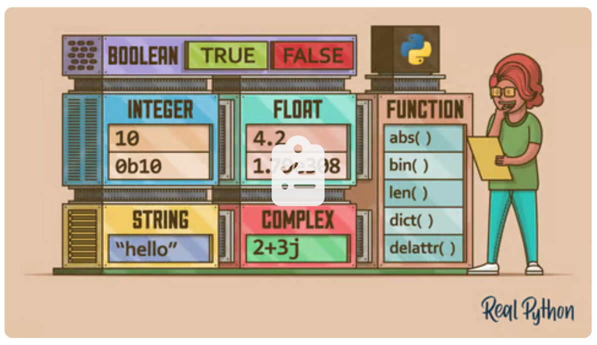
By the end of this tutorial, you'll understand that:

- Python's **basic data types** include `int`, `float`, `complex`, `str`, `bytes`, `bytearray`, and `bool`.
- You can **check a variable's type** using the **`type()` function** in Python.
- You can **convert data types** in Python using functions like `int()`, `float()`, `str()`, and others.
- Despite being **dynamically typed**, Python does have data types.
- The most essential data types in Python can be categorized as **numeric**, **sequence**, **binary**, and **Boolean**.

In this tutorial, you'll learn the basics of each data type. To learn more about a specific data type, you'll find useful resources in the corresponding section.

Get Your Code: Click here to download the free sample code that you'll use to learn about basic data types in Python.

Take the Quiz: Test your knowledge with our interactive "Basic Data Types in Python: A Quick Exploration" quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Basic Data Types in Python: A Quick Exploration

Take this quiz to test your understanding of the basic data types that are built into Python, like numbers, strings, bytes, and Booleans.

Python's Basic Data Types

Python has several **built-in data types** that you can use out of the box because they're built into the language. From all the built-in types available, you'll find that a few of them represent *basic* objects, such as numbers, strings and characters, bytes, and Boolean values.

Note that the term **basic** refers to objects that can represent data you typically find in real life, such as numbers and text. It doesn't include composite data types, such as lists, tuples, dictionaries, and others.

In Python, the built-in data types that you can consider basic are the following:

Class	Basic Type
`int`	Integer numbers
`float`	Floating-point numbers
`complex`	Complex numbers
`str`	Strings and characters
`bytes`, `bytearray`	Bytes
`bool`	Boolean values

In the following sections, you'll learn the basics of how to create, use, and work with all of these built-in data types in Python.

Integer Numbers

Integer numbers are whole numbers with no decimal places. They can be positive or negative numbers. For example, `0`, `1`, `2`, `3`, `-1`, `-2`, and `-3` are all integers. Usually, you'll use positive integer numbers to count things.

In Python, the integer data type is represented by the `int` class:

Python

```
>>> type(42)
<class 'int'>
```

In the following sections, you'll learn the basics of how to create and work with integer numbers in Python.

Integer Literals

When you need to use integer numbers in your code, you'll often use integer literals directly. Literals are constant values of built-in types spelled out literally, such as integers. Python provides a few different ways to create integer literals. The most common way is to use base-ten literals that look the same as integers look in math:

Python

```
>>> 42
42

>>> -84
-84

>>> 0
0
```

Here, you have three integer numbers: a positive one, a negative one, and zero. Note that to create negative integers, you need to prepend the minus sign (`-`) to the number.

Python has no limit to how long an integer value can be. The only constraint is the amount of memory your system has. Beyond that, an integer can be as long as you need:

Python

```
>>> 123123123123123123123123123123123123123123123123 + 1
123123123123123123123123123123123123123123123124
```

For a really, really long integer, you can get a `ValueError` when converting it to a string:

Python

```
>>> 123 ** 10000
Traceback (most recent call last):
  ...
ValueError: Exceeds the limit (4300 digits) for integer string conversion;
            use sys.set_int_max_str_digits() to increase the limit
```

If you need to print an integer number beyond the 4300-digit limit, then you can use the `sys.set_int_max_str_digits()` function to increase the limit and make your code work.

When you're working with long integers, you can use the underscore character to make the literals more readable:

Python

```
>>> 1_000_000
1000000
```

With the underscore as a thousands separator, you can make your integer literals more readable for fellow programmers reading your code.

You can also use other bases to represent integers. You can prepend the following characters to an integer value to indicate a base other than `10`:

Prefix	Representation	Base
`0b` or `0B` (Zero + b or B)	Binary	2
`0o` or `0O` (Zero + o or O)	Octal	8
`0x` or `0X` (Zero + x or X)	Hexadecimal	16

Using the above characters, you can create integer literals using binary, octal, and hexadecimal representations. For example:

Python

```
>>> 10  # Base 10
10
>>> type(10)
<class 'int'>

>>> 0b10  # Base 2
2
>>> type(0b10)
<class 'int'>

>>> 0o10  # Base 8
8
>>> type(0o10)
<class 'int'>

>>> 0x10  # Base 16
16
>>> type(0x10)
<class 'int'>
```

Note that the underlying type of a Python integer is always `int`. So, in all cases, the built-in `type()` function returns `int`, irrespective of the base you use to build the literal.

Integer Methods

The built-in `int` type has a few methods that you can use in some situations. Here's a quick summary of these methods:

Method	Description
`.as_integer_ratio()`	Returns a pair of integers whose ratio is equal to the original integer and has a positive denominator
`.bit_count()`	Returns the number of ones in the binary representation of the absolute value of the integer
`.bit_length()`	Returns the number of bits necessary to represent an integer in binary, excluding the sign and leading zeros
`.from_bytes()`	Returns the integer represented by the given array of bytes
`.to_bytes()`	Returns an array of bytes representing an integer
`.is_integer()`	Returns `True`

When you call the `.as_integer_ratio()` method on an integer value, you get the integer as the numerator and `1` as the denominator. As you'll see in a moment, this method is more useful in floating-point numbers.

Note that the `int` type also has a method called `.is_integer()`, which always returns `True`. This method exists for duck typing compatibility with floating-point numbers, which have the method as part of their public interface.

Note: To access an integer method on a literal, you need to wrap the literal in parentheses:

Python

```
>>> (42).as_integer_ratio()
(42, 1)

>>> 42.as_integer_ratio()
  File "<input>", line 1
    42.as_integer_ratio()
      ^
SyntaxError: invalid decimal literal
```

The parentheses are required because the dot character (`.`) also defines floating-point numbers, as you'll learn in a moment. If you don't use the parentheses, then you get a `SyntaxError`.

The `.bit_count()` and `.bit_length()` methods can help you when working on digital signal processing. For example, you may want every transmitted signal to have an even number of set bits:

Python

```
>>> signal = 0b11010110
>>> set_bits = signal.bit_count()

>>> if set_bits % 2 == 0:
...     print("Even parity")
... else:
...     print("Odd parity")
...
Odd parity
```

In this toy example, you use `.bit_count()` to ensure that the received signal has the correct parity. This way, you implement a basic error detection mechanism.

Finally, the `.from_bytes()` and `.to_bytes()` methods can be useful in network programming. Often, you need to send and receive data over the network in binary format. To do this, you can use `.to_bytes()` to convert the message for network transmission. Similarly, you can use `.from_bytes()` to convert the message back.

The Built-in `int()` Function

The built-in `int()` function provides another way to create integer values using different representations. With no arguments, the function returns `0`:

Python

```
>>> int()
0
```

This feature makes `int()` especially useful when you need a factory function for classes like `defaultdict` from the `collections` module.

Note: In Python, the built-in functions associated with data types, such as `int()`, `float()`, `str()`, and `bytes()`, are classes with a function-style name. The Python documentation calls them functions, so you'll follow that practice in this tutorial. However, keep in mind that something like `int()` is really a class constructor rather than a regular function.

The `int()` function is commonly used to convert other data types into integers, provided that they're valid numeric values:

Python

```
>>> int(42.0)
42

>>> int("42")
42

>>> int("one")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    int("one")
ValueError: invalid literal for int() with base 10: 'one'
```

In these examples, you first use `int()` to convert a floating-point number into an integer. Then, you convert a string into an integer. Note that when it comes to strings, you must ensure that the input string is a valid numeric value. Otherwise, you'll get a `ValueError` exception.

Note: When you use the `int()` function to convert floating-point numbers, you must be aware that the function just removes the decimal or fractional part.

This function can take an additional argument called `base`, which defaults to `10` for decimal integers. This argument allows you to convert strings that represent integer values, which are expressed using a different base:

Python

```
>>> int("0b10", base=2)
2

>>> int("10", base=8)
8

>>> int("10", base=16)
16
```

In this case, the first argument must be a string representing an integer value with or without a prefix. Then, you must provide the appropriate base in the second argument to run the conversion. Once you call the function, you get the resulting integer value.

Floating-Point Numbers

Floating-point numbers, or just **float**, are numbers with a decimal place. For example, `1.0` and `3.14` are floating-point numbers. You can also have negative float numbers, such as `-2.75`. In Python, the name of the `float` class represents floating-point numbers:

Python

```
>>> type(1.0)
<class 'float'>
```

In the following sections, you'll learn the basics of how to create and work with floating-point numbers in Python.

Floating-Point Literals

The `float` type in Python designates floating-point numbers. To create these types of numbers, you can also use literals, similar to what you use in math. However, in Python, the dot character (`.`) is what you must use to create floating-point literals:

Python

```
>>> 4.2
4.2

>>> 4.
4.0

>>> .2
0.2
```

In these quick examples, you create floating-point numbers in three different ways. First, you have a literal build using an integer part, the dot, and the decimal part. You can also create a literal using the dot without specifying the decimal part, which defaults to `0`. Finally, you make a literal without specifying the integer part, which also defaults to `0`.

You can also have negative float numbers:

Python

```
>>> -42.0
-42.0
```

To create a negative floating-point number using a literal, you need to prepend the minus sign (`-`) to the number.

Similar to integer numbers, if you're working with long floating-point numbers, you can use the underscore character as a thousands separator:

Python

```
>>> 1_000_000.0
1000000.0
```

By using an underscore, you can make your floating-point literals more readable for humans, which is great.

Optionally, you can use the characters `e` or `E` followed by a positive or negative integer to express the number using scientific notation:

Python

```
>>> .4e7
4000000.0

>>> 4.2E-4
0.00042
```

By using the `e` or `E` character, you can represent any floating-point number using scientific notation, as you did in the above examples.

Floating-Point Numbers Representation

Now, you can take a more in-depth look at how Python internally represents floating-point numbers. 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 in this section is only meant to satisfy your curiosity.

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

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

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 considered to be zero:

Python

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

Python internally represents floating-point numbers as binary (base-2) fractions. Most decimal fractions can't be represented exactly as binary fractions. So, in most cases, the internal representation of a floating-point number is an approximation of its actual value.

In practice, the difference between the *actual* and *represented* values is small and should be manageable. However, check out Make Python Lie to You for some challenges you should be aware of.

Floating-Point Methods

The built-in `float` type has a few methods and attributes which can be useful in some situations. Here's a quick summary of them:

Method	Description
`.as_integer_ratio()`	Returns a pair of integers whose ratio is exactly equal to the original `float`
`.is_integer()`	Returns `True` if the float instance is finite with integral value, and `False` otherwise
`.hex()`	Returns a representation of a floating-point number as a hexadecimal string
`.fromhex(string)`	Builds the `float` from a hexadecimal string

The `.as_integer_ratio()` method on a `float` value returns a pair of integers whose ratio equals the original number. You can use this method in scientific computations that require high precision. In these situations, you may need to avoid precision loss due to floating-point rounding errors.

For example, say that you need to perform computations with the gravitational constant:

Python

```
>>> G = 6.67430e-11

>>> G.as_integer_ratio()
(1290997375656627, 19342813113834066795298816)
```

With this exact ratio, you can perform calculations and prevent floating-point errors that may alter the results of your research.

The `.is_integer()` method allows you to check whether a given `float` value is an integer:

Python

```
>>> (42.0).is_integer()
True

>>> (42.42).is_integer()
False
```

When the number after the decimal point is `0`, the `.is_integer()` method returns `True`. Otherwise, it returns `False`.

Finally, the `.hex()` and `.fromhex()` methods allow you to work with floating-point values using a hexadecimal representation:

Python

```
>>> (42.0).hex()
'0x1.5000000000000p+5'

>>> float.fromhex("0x1.5000000000000p+5")
42.0
```

The `.hex()` method returns a string that represents the target float value as a hexadecimal value. Note that `.hex()` is an instance method. The `.fromhex()` method takes a string that represents a floating-point number as an argument and builds an actual float number from it.

In both methods, the hexadecimal string has the following format:

Python

```
[sign] ["0x"] integer ["." fraction] ["p" exponent]
```

In this template, apart from the `integer` identifier, the components are optional. Here's what they mean:

- **`sign`** defines whether the number is positive or negative. It may be either `+` or `-`. Only the `-` sign is required because `+` is the default.
- **`"0x"`** is the hexadecimal prefix.
- **`integer`** is a string of hexadecimal digits representing the whole part of the `float` number.
- **`"."`** is a dot that separates the whole and fractional parts.
- **`fraction`** is a string of hexadecimal digits representing the fractional part of the `float` number.
- **`"p"`** allows for adding an exponent value.
- **`exponent`** is a decimal integer with an optional leading sign.

With these components, you'll be able to create valid hexadecimal strings to process your floating-point numbers with the `.hex()` and `.fromhex()` methods.

The Built-in `float()` Function

The built-in `float()` function provides another way to create floating-point values. When you call `float()` with no argument, then you get `0.0`:

Python

```
>>> float()
0.0
```

Again, this feature of `float()` allows you to use it as a factory function.

The `float()` function also helps you convert other data types into `float`, provided that they're valid numeric values:

Python

```
>>> float(42)
42.0

>>> float("42")
42.0

>>> float("one")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    float("one")
ValueError: could not convert string to float: 'one'
```

In these examples, you first use `float()` to convert an integer number into a float. Then, you convert a string into a float. Again, with strings, you need to make sure that the input string is a valid numeric value. Otherwise, you get a `ValueError` exception.

Complex Numbers

Python has a built-in type for **complex numbers**. Complex numbers are composed of **real** and **imaginary** parts. They have the form *a* + *bi*, where *a* and *b* are real numbers, and *i* is the imaginary unit. In Python, you'll use a `j` instead of an `i`. For example:

Python

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

In this example, the argument to `type()` may look like an expression. However, it's a literal of a complex number in Python. If you pass the literal to the `type()` function, then you'll get the `complex` type back.

Note: To dive deeper into complex numbers, check out the Simplify Complex Numbers With Python tutorial.

In the following sections, you'll learn the basics of creating complex numbers in Python. You'll also explore the methods of this data type.

Complex Number Literals

In Python, you can define complex numbers using literals that look like `a + bj`, where `a` is the real part, and `bj` is the imaginary part:

Python

```
>>> 2 + 3j
(2+3j)

>>> 7j
7j

>>> 2.4 + 7.5j
(2.4+7.5j)

>>> 3j + 5
(5+3j)

>>> 5 - 3j
(5-3j)

>>> 1 + j
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    1 + j
        ^
NameError: name 'j' is not defined
```

As you can conclude from these examples, there are many ways to create complex numbers using literals. The key is that you need to use the `j` letter in one of the components. Note that the `j` can't be used alone. If you try to do so, you get a `NameError` exception because Python thinks that you're creating an expression. Instead, you need to write `1j`.

Complex Number Methods

In Python, the `complex` type has a single method called `.conjugate()`. When you call this method on a complex number, you get the conjugate:

Python

```
>>> number = 2 + 3j

>>> number.conjugate()
(2-3j)
```

The `conjugate()` method flips the sign of the imaginary part, returning the complex conjugate.

The Built-in `complex()` Function

You can also use the built-in `complex()` function to create complex numbers by providing the real and imaginary parts as arguments:

Python

```
>>> complex()
0j

>>> complex(1)
(1+0j)

>>> complex(0, 7)
7j

>>> complex(2, 3)
(2+3j)

>>> complex(2.4, 7.5)
(2.4+7.5j)

>>> complex(5, -3)
(5-3j)
```

When you call `complex()` with no argument, you get `0j`. If you call the function with a single argument, that argument is the real part, and the imaginary part will be `0j`. If you want only the imaginary part, you can pass `0` as the first argument. Note that you can also use negative numbers. In general, you can use integers and floating-point numbers as arguments to `complex()`.

You can also use `complex()` to convert strings to complex numbers:

Python

```
>>> complex("5-3j")
(5-3j)

>>> complex("5")
(5+0j)

>>> complex("5 - 3j")
Traceback (most recent call last):
    ...
ValueError: complex() arg is a malformed string

>>> complex("5", "3")
Traceback (most recent call last):
    ...
TypeError: complex() can't take second arg if first is a string
```

To convert strings into complex numbers, you must provide a string that follows the format of complex numbers. For example, you can't have spaces between the components. If you add spaces, then you get a `ValueError` exception.

Finally, note that you can't use strings to provide the imaginary part of complex numbers. If you do that, then you get a `TypeError` exception.

Strings and Characters

In Python, strings are sequences of character data that you can use to represent and store textual data. The string type in Python is called `str`:

Python

```
>>> type("Hello, World!")
<class 'str'>
```

In this example, the argument to `type()` is a string literal that you commonly create using double quotes to enclose some text.

Note: Check out the Strings and Character Data in Python tutorial to dive deeper into Python strings.

In the following sections, you'll learn the basics of how to create, use, format, and manipulate strings in Python.

Regular String Literals

You can also use literals to create strings. To build a single-line string literal, you can use double (`""`) or single quotes (`''`) and, optionally, a sequence of characters in between them. All the characters between the opening and closing quotes are part of the string:

Python

```
>>> print("I am a string")
I am a string

>>> print('I am a string too')
I am a string too
```

Python's strings can contain as many characters as you need. The only limit is your computer's memory.

You can define empty strings by using the quotes without placing characters between them:

Python

```
>>> ""
''

>>> ''
''

>>> len("")
0
```

An empty string doesn't contain any characters, so when you use the built-in `len()` function with an empty string as an argument, you get `0` as a result.

There is yet another way to delimit strings in Python. You can create triple-quoted string literals, which can be delimited using either three single quotes or three double quotes. Triple-quoted strings are commonly used to build multiline string literals. However, you can also use them to create single-line literals:

Python

```
>>> """A triple-quoted string in a single line"""
'A triple-quoted string in a single line'

>>> '''Another triple-quoted string in a single line'''
'Another triple-quoted string in a single line'

>>> """A triple-quoted string
... that spans across multiple
... lines"""
'A triple-quoted string\nthat spans across multiple\nlines'
```

Even though you can use triple-quoted strings to create single-line string literals, the main use case of them would be to create multiline strings. In Python code, probably the most common use case for these string literals is when you need to provide docstrings for your packages, modules, functions, classes, and methods.

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

```
>>> 'This string contains a single quote (') character'
  File "<input>", line 1
    'This string contains a single quote (') character'
                                            ^
SyntaxError: unmatched ')'
```

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—is the closing delimiter. The final single quote is then a stray, which causes the syntax error shown.

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

Python

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

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

In these examples, your first string includes a single quote as part of the text. To do this, you use double quotes to delimit the literal. In the second example, you do the opposite.

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:

1. Apply special meaning to characters
2. Suppress special character meaning

You can accomplish these goals by using a backslash (\) character to indicate that the characters following it should be interpreted specially. The combination of a backslash and a specific character is called an **escape sequence**. That's because the backslash causes the subsequent character to *escape* its usual meaning.

You already know that if you use single quotes to delimit a string, then you can't directly embed a single quote character as part of the string because, for that string, the single quote has a **special meaning**—it terminates the string. You can eliminate this limitation by using double quotes to delimit the string.

Alternatively, you can escape the quote character using a backslash:

Python

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

In this example, the backslash escapes the single quote character by suppressing its usual meaning. Now, Python knows that your intention isn't to terminate the string but to embed the single quote.

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

Character	Usual Interpretation	Escape Sequence	Escaped Interpretation
`'`	Delimit a string literal	`\'`	Literal single quote (`'`) character
`"`	Delimit a string literal	`\"`	Literal double quote (`"`) character
`<newline>`	Terminates the input line	`\<newline>`	Newline is ignored
`\`	Introduces an escape sequence	`\\`	Literal backslash (`\`) character

You already have an idea of how the first two escape sequences work. Now, how does the newline escape sequence work? Usually, a newline character terminates a physical line of input. So, pressing `Enter ↩` in the middle of a string will cause an error:

Python

```
>>> "Hello
  File "<input>", line 1
    "Hello
    ^
SyntaxError: incomplete input
```

When you press `Enter ↩` after typing `Hello`, you get a `SyntaxError`. If you need to break up a string over more than one line, then you can include a backslash before each new line:

Python

```
>>> "Hello\
... , World\
... !"
'Hello, World!'
```

By using a backslash before pressing enter, you make Python ignore the new line and interpret the whole construct as a single line.

Finally, sometimes you need to include a literal backslash character in a string. If that backslash doesn't precede a character with a special meaning, then you can insert it right away:

Python

```
>>> "This string contains a backslash (\) character"
'This string contains a backslash (\\) character'
```

In this example, the character after the backslash doesn't match any known escape sequence, so Python inserts the actual backslash for you. Note how the resulting string automatically doubles the backslash. Even though this example works, the best practice is to always double the backslash when you need this character in a string.

However, you may have the need to include a backslash right before a character that makes up an escape sequence:

Python

```
>>> "In this string, the backslash should be at the end \"
  File "<input>", line 1
    "In this string, the backslash should be at the end \"
    ^
SyntaxError: incomplete input
```

Because the sequence `\"` matches a known escape sequence, your string fails with a `SyntaxError`. To avoid this issue, you can double the backslash:

Python

```
>>> "In this string, the backslash should be at the end \\"
'In this string, the backslash should be at the end \\'
```

In this update, you double the backslash to escape the character and prevent Python from raising an error.

Note: When you use the built-in `print()` function to print a string that includes an escaped backslash, then you won't see the double backslash in the output:

Python

```
>>> print("In this string, the backslash should be at the end \\")
In this string, the backslash should be at the end \
```

In this example, the output only displays one backslash, producing the desired effect.

Up to this point, you've learned how to suppress the meaning of a given character by escaping it. Suppose you need to create a string containing a tab character. Some text editors may allow you to insert a tab character directly into your code. However, this is considered a poor practice for several reasons:

- Computers can distinguish between tabs and a sequence of spaces, but human beings can't because these characters are visually indistinguishable.
- Some text editors automatically eliminate tabs by expanding them to an appropriate number of spaces.
- Some Python REPL environments will not insert tabs into code.

In Python, you can specify a tab character by the `\t` escape sequence:

Python

```
>>> print("Before\tAfter")
Before     After
```

The `\t` escape sequence changes the usual meaning of the letter `t`. Instead, Python interprets the combination as a tab character.

Here is a list of escape sequences that cause Python to apply special meaning to some characters instead of interpreting them 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

Escape Sequence	Escaped Interpretation
`\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`

Of these escape sequences, the newline or linefeed character (`\n`) is probably the most popular. This sequence is commonly used to create nicely formatted text outputs.

Here are a few examples of the escape sequences in action:

Python

```
>>> # Tab
>>> print("a\tb")
a	b

>>> # Linefeed
>>> print("a\nb")
a
b

>>> # Octal
>>> print("\141")
a

>>> # Hex
>>> print("\x61")
a

>>> # Unicode by name
>>> print("\N{rightwards arrow}")
→
```

These escape sequences are typically useful when you need to insert characters that aren't readily generated from the keyboard or aren't easily readable or printable.

Raw String Literals

A raw string is a string that doesn't translate the escape sequences. Any backslash characters are left in the string.

Note: To learn more about raw strings, check out the What Are Python Raw Strings? tutorial.

To create a raw string, you can precede the literal with an `r` or `R`:

Python

```
>>> print("Before\tAfter")  # Regular string
Before	After

>>> print(r"Before\tAfter")  # Raw string
Before\tAfter
```

The raw string suppresses the meaning of the escape sequence and presents the characters as they are. This behavior comes in handy when you're creating regular expressions because it allows you to use several different characters that may have special meanings without restrictions.

F-String Literals

Python has another type of string literal called formatted strings or f-strings for short. F-strings allow you to interpolate values into your strings and format them as you need.

Note: To dive deeper into f-strings, check out the Python's F-String for String Interpolation and Formatting tutorial

To build f-string literals, you must prepend an `f` or `F` letter to the string literal. Because the idea behind f-strings is to interpolate values and format them into the final string, you need to use something called a **replacement field** in your string literal. You create these fields using curly brackets.

Here's a quick example of an f-string literal:

Python

```python
>>> name = "Jane"

>>> f"Hello, {name}!"
'Hello, Jane!'
```

In this example, you interpolate the variable name into your string using an f-string literal and a replacement field.

You can also use f-strings to format the interpolated values. To do that, you can use format specifiers that use the syntax defined in Python's string format mini-language. For example, here's how you can present numeric values using a currency format:

Python

```python
>>> income = 1234.1234

>>> f"Income: ${income:.2f}"
'Income: $1234.12'
```

Inside the replacement field, you have the variable you want to interpolate and the format specifier, which is the string that starts with a colon (`:`). In this example, the format specifier defines a floating-point number with two decimal places.

String Methods

Python's `str` data type is probably the built-in type with the most available methods. In fact, you'll find methods for most string processing operations. Here's a summary of the methods that perform some string processing and return a transformed string object:

Method	Description
`.capitalize()`	Converts the first character to uppercase and the rest to lowercase
`.casefold()`	Converts the string into lowercase
`.center(width[, fillchar])`	Centers the string between `width` using `fillchar`
`.encode(encoding, errors)`	Encodes the string using the specified `encoding`
`.expandtabs(tabsize)`	Replaces tab characters with spaces according to `tabsize`
`.format(*args, **kwargs)`	Interpolates and formats the specified values
`.format_map(mapping)`	Interpolates and formats the specified values using a dictionary

Method	Description
`.join(iterable)`	Joins the items in an iterable with the string as a separator
`.ljust(width[, fillchar])`	Returns a left-justified version of the string
`.rjust(width[, fillchar])`	Returns a right-justified version of the string
`.lower()`	Converts the string into lowercase
`.strip([chars])`	Trims the string by removing `chars` from the beginning and end
`.lstrip([chars])`	Trims the string by removing `chars` from the beginning
`.rstrip([chars])`	Trims the string by removing `chars` from the end
`.removeprefix(prefix, /)`	Removes `prefix` from the beginning of the string
`.removesuffix(suffix, /)`	Removes `suffix` from the end of the string
`.replace(old, new [, count])`	Returns a string where the `old` substring is replaced with `new`
`.swapcase()`	Converts lowercase letters to uppercase letters and vice versa
`.title()`	Converts the first character of each word to uppercase and the rest to lowercase
`.upper()`	Converts a string into uppercase
`.zfill(width)`	Fills the string with a specified number of zeroes at the beginning

All the above methods allow you to perform a specific transformation on an existing string. In all cases, you get a new string as a result:

Python

```python
>>> "beautiful is better than ugly".capitalize()
'Beautiful is better than ugly'

>>> name = "Jane"
>>> "Hello, {0}!".format(name)
'Hello, Jane!'

>>> " ".join(["Now", "is", "better", "than", "never"])
'Now is better than never'

>>> "====Header====".strip("=")
'Header'

>>> "---Tail---".removeprefix("---")
'Tail---'

>>> "---Head---".removesuffix("---")
'---Head'

>>> "Explicit is BETTER than implicit".title()
'Explicit Is Better Than Implicit'

>>> "Simple is better than complex".upper()
'SIMPLE IS BETTER THAN COMPLEX'
```

As you can see, the methods in these examples perform a specific transformation on the original string and return a new string object.

You'll also find that the `str` class has several Boolean-valued methods or predicate methods:

Method	Result
`.endswith(suffix[, start[, end]])`	`True` if the string ends with the specified suffix, `False` otherwise
`.startswith(prefix[, start[, end]])`	`True` if the string starts with the specified prefix, `False` otherwise
`.isalnum()`	`True` if all characters in the string are alphanumeric, `False` otherwise
`.isalpha()`	`True` if all characters in the string are letters, `False` otherwise
`.isascii()`	`True` if the string is empty or all characters in the string are ASCII, `False` otherwise
`.isdecimal()`	`True` if all characters in the string are decimals, `False` otherwise
`.isdigit()`	`True` if all characters in the string are digits, `False` otherwise
`.isidentifier()`	`True` if the string is a valid Python name, `False` otherwise
`.islower()`	`True` if all characters in the string are lowercase, `False` otherwise
`.isnumeric()`	`True` if all characters in the string are numeric, `False` otherwise
`.isprintable()`	`True` if all characters in the string are printable, `False` otherwise
`.isspace()`	`True` if all characters in the string are whitespaces, `False` otherwise
`.istitle()`	`True` if the string follows title case, `False` otherwise
`.isupper()`	`True` if all characters in the string are uppercase, `False` otherwise

All these methods allow you to check for various conditions in your strings. Here are a few demonstrative examples:

Python

```python
>>> filename = "main.py"
>>> if filename.endswith(".py"):
...     print("It's a Python file")
...
It's a Python file

>>> "123abc".isalnum()
True

>>> "123abc".isalpha()
False

>>> "123456".isdigit()
True

>>> "abcdf".islower()
True
```

In these examples, the methods check for specific conditions in the target string and return a Boolean value as a result.

Finally, you'll find a few other methods that allow you to run several other operations on your strings:

Method	Description
`.count(sub[, start[, end]])`	Returns the number of occurrences of a substring

Method	Description
`.find(sub[, start[, end]])`	Searches the string for a specified value and returns the position of where it was found
`.rfind(sub[, start[, end]])`	Searches the string for a specified value and returns the last position of where it was found
`.index(sub[, start[, end]])`	Searches the string for a specified value and returns the position of where it was found
`.rindex(sub[, start[, end]])`	Searches the string for a specified value and returns the last position of where it was found
`.split(sep=None, maxsplit=-1)`	Splits the string at the specified separator and returns a list
`.splitlines([keepends])`	Splits the string at line breaks and returns a list
`.partition(sep)`	Splits the string at the first occurance of `sep`
`.rpartition(sep)`	Splits the string at the last occurance of `sep`
`.split(sep=None, maxsplit=-1)`	Splits the string at the specified separator and returns a list
`.maketrans(x[, y[, z]])`	Returns a translation table to be used in translations
`.translate(table)`	Returns a translated string

The first method counts the number of repetitions of a substring in an existing string. Then, you have four methods that help you find substrings in a string.

The `.split()` method is especially useful when you need to split a string into a list of individual strings using a given character as a separator, which defaults to whitespaces. You can also use `.partition()` or `.rpartition()` if you need to divide the string in exactly two parts:

Python

```
>>> sentence = "Flat is better than nested"
>>> words = sentence.split()
>>> words
['Flat', 'is', 'better', 'than', 'nested']

>>> numbers = "1-2-3-4-5"
>>> head, sep, tail = numbers.partition("-")
>>> head
'1'
>>> sep
'-'
>>> tail
'2-3-4-5'

>>> numbers.rpartition("-")
('1-2-3-4', '-', '5')
```

In these toy examples, you've used the `.split()` method to build a list of words from a sentence. Note that by default, the method uses whitespace characters as separators. You also used `.partition()` and `.rpartition()` to separate out the first and last number from a string with numbers.

The `.maketrans()` and `.translate()` are nice tools for playing with strings. For example, say that you want to implement the Cesar cipher algorithm. This algorithm allows for basic text encryption by shifting the alphabet by a number of letters. For example, if you shift the letter `a` by three, then you get the letter `d`, and so on.

The following code implements `cipher()`, a function that takes a character and rotates it by three:

Python

```
>>> def cipher(text):
...     alphabet = "abcdefghijklmnopqrstuvwxyz"
...     shifted = "defghijklmnopqrstuvwxyzabc"
...     table = str.maketrans(alphabet, shifted)
...     return text.translate(table)
...

>>> cipher("python")
'sbwkrq'
```

In this example, you use `.maketrans()` to create a translation table that matches the lowercase alphabet to a shifted alphabet. Then, you apply the translation table to a string using the `.translate()` method.

Common Sequence Operations on Strings

Python's strings are **sequences** of characters. As other built-in sequences like lists and tuples, strings support a set of operations that are known as common sequence operations. The table below is a summary of all the operations that are common to most sequence types in Python:

Operation	Example	Result
Length	`len(s)`	The length of `s`
Indexing	`s[index]`	The item at index `i`
Slicing	`s[i:j]`	A slice of `s` from index `i` to `j`
Slicing	`s[i:j:k]`	A slice of `s` from index `i` to `j` with step `k`
Minimum	`min(s)`	The smallest item of `s`
Maximum	`max(s)`	The largest item of `s`
Membership	`x in s`	`True` if an item of `s` is equal to `x`, else `False`
Membership	`x not in s`	`False` if an item of `s` is equal to `x`, else `True`
Concatenation	`s + t`	The concatenation of `s` and `t`
Repetition	`s * n` or `n * s`	The repetition of `s` a number of times specified by `n`
Index	`s.index(x[, i[, j]])`	The index of the first occurrence of `x` in `s`
Count	`s.count(x)`	The total number of occurrences of `x` in `s`

Sometimes, you need to determine the number of characters in a string. In this situation, you can use the built-in `len()` function:

Python

```
>>> len("Pythonista")
10
```

When you call `len()` with a string as an argument, you get the number of characters in the string at hand.

Another common operation you'd run on strings is retrieving a single character or a substring from an existing string. In these situations, you can use indexing and slicing, respectively:

Python

```
>>> "Pythonista"[0]
'P'
>>> "Pythonista"[9]
'a'
>>> "Pythonista"[4]
'o'

>>> "Pythonista"[:6]
'Python'
```

To retrieve a character from an existing string, you use the indexing operator `[index]` with the index of the target character. Note that indices are zero-based, so the first character lives at index `0`.

To retrieve a slice or substring from an existing string, you use the slicing operator with the appropriate indices. In the example above, you don't provide the start index `i`, so Python assumes that you want to start from the beginning of the string. Then, you give the end index `j` to tell Python where to stop the slicing.

You can take a leap and try the rest of the operations by yourself. It will be a great learning exercise!

The Built-in `str()` and `repr()` Functions

When it comes to creating and working with strings, you have two functions that can help you out and make your life easier:

- `str()`
- `repr()`

The built-in `str()` function allows you to create new strings and also convert other data types into strings:

Python

```
>>> str()
''

>>> str(42)
'42'

>>> str(3.14)
'3.14'

>>> str([1, 2, 3])
'[1, 2, 3]'

>>> str({"one": 1, "two": 2, "three": 3})
"{'one': 1, 'two': 2, 'three': 3}"

>>> str({"A", "B", "C"})
"{'B', 'C', 'A'}"
```

In these examples, you use the `str()` function to convert objects from different built-in types into strings. In the first example, you use the function to create an empty string. In the other examples, you get strings consisting of the object's literals between quotes, which provide user-friendly representations of the objects.

At first glance, these results may not seem useful. However, there are use cases where you need to use `str()`.

For example, say that you have a list of numeric values and want to join them using the `str.join()` method. This method only accepts iterables of strings, so you need to convert the numbers:

Python

```
>>> "-".join([1, 2, 3, 4, 5])
Traceback (most recent call last):
    ...
TypeError: sequence item 0: expected str instance, int found

>>> "-".join(str(value) for value in [1, 2, 3, 4, 5])
'1-2-3-4-5'
```

If you try to pass a list of numeric values to `.join()`, then you get a `TypeError` exception because the function only joins strings. To work around this issue, you use a generator expression to convert each number to its string representation.

Behind the `str()` function, you'll have the `.__str__()` special method. In other words, when you call `str()`, Python automatically calls the `.__str__()` special method on the underlying object. You can use this special method to support `str()` in your own classes.

Consider the following `Person` class:

Python `person.py`

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"I'm {self.name}, and I'm {self.age} years old."
```

In this class, you have two instance attributes, `.name` and `.age`. Then, you have `.__str__()` special methods to provide user-friendly string representations for your class.

Here's how this class works:

Python

```
>>> from person import Person

>>> john = Person("John Doe", 35)
>>> str(john)
"I'm John Doe, and I'm 35 years old."
```

In this code snippet, you create an instance of `Person`. Then, you call `str()` using the object as an argument. As a result, you get a descriptive message back, which is the user-friendly string representation of your class.

> **Note:** To learn more about objects' string representations in Python, check out the When Should You Use `.__repr__()` vs `.__str__()` in Python? tutorial.

Similarly, when you pass an object to the built-in `repr()` function, you get a developer-friendly string representation of the object itself:

Python

```
>>> repr(42)
'42'

>>> repr(3.14)
'3.14'

>>> repr([1, 2, 3])
'[1, 2, 3]'

>>> repr({"one": 1, "two": 2, "three": 3})
"{'one': 1, 'two': 2, 'three': 3}"

>>> repr({"A", "B", "C"})
"{'B', 'C', 'A'}"
```

In the case of built-in types, the string representation you get with `repr()` is the same as the one you get with the `str()` function. This is because the representations are the literals of each object, and you can directly use them to re-create the object at hand.

Ideally, you should be able to re-create the current object using this representation. To illustrate, go ahead and update the `Person` class:

Python person.py

```python
class Person:
    # ...

    def __repr__(self):
        return f"{type(self).__name__}(name='{self.name}', age={self.age})"
```

The `.__repr__()` special method allows you to provide a developer-friendly string representation for your class:

Python

```python
>>> from person import Person

>>> jane = Person("Jane Doe", 28)
>>> repr(jane)
"Person(name='Jane Doe', age=28)"
```

You should be able to copy and paste the resulting representation to re-create the object. That's why this string representation is said to be developer-friendly.

Bytes and Byte Arrays

Bytes are immutable sequences of single bytes. In Python, the `bytes` class allows you to build sequences of bytes. This data type is commonly used for manipulating binary data, encoding and decoding text, processing file input and output, and communicating through networks.

Python also has a `bytearray` class as a mutable counterpart to `bytes` objects:

Python

```python
>>> type(b"This is a bytes literal")
<class 'bytes'>

>>> type(bytearray(b"Form bytes"))
<class 'bytearray'>
```

In the following sections, you'll learn the basics of how to create and work with `bytes` and `bytearray` objects in Python.

Bytes Literals

To create a `bytes` literal, you'll use a syntax that's largely the same as that for string literals. The difference is that you need to prepend a `b` to the string literal. As with string literals, you can use different types of quotes to define bytes literals:

Python

```python
>>> b'This is a bytes literal in single quotes'
b'This is a bytes literal in single quotes'

>>> b"This is a bytes literal in double quotes"
b'This is a bytes literal in double quotes'
```

There is yet another difference between string literals and `bytes` literals. To define bytes literals, you can only use ASCII characters. If you need to insert binary values over the 127 characters, then you have to use the appropriate escape sequence:

Python

```
>>> b"Espa\xc3\xb1a"
b'Espa\xc3\xb1a'

>>> b"Espa\xc3\xb1a".decode("utf-8")
'España'

>>> b"España"
  File "<input>", line 1
    b"España"
            ^
SyntaxError: incomplete input
```

In this example, `\xc3\xb1` is the escape sequence for the letter `ñ` in the Spanish word `"España"`. Note that if you try to use the `ñ` directly, you get a `SyntaxError`.

The Built-in `bytes()` Function

The built-in `bytes()` function provides another way to create `bytes` objects. With no arguments, the function returns an empty `bytes` object:

Python

```
>>> bytes()
b''
```

You can use the `bytes()` function to convert string literals to `bytes` objects:

Python

```
>>> bytes("Hello, World!", encoding='utf-8')
b'Hello, World!'

>>> bytes("Hello, World!")
Traceback (most recent call last):
    ...
TypeError: string argument without an encoding
```

In these examples, you first use `bytes()` to convert a string into a `bytes` object. Note that for this to work, you need to provide the appropriate character encoding. In this example, you use the UTF-8 encoding. If you try to convert a string literal without providing the encoding, then you get a `TypeError` exception.

You can also use `bytes()` with an iterable of integers where each number is the Unicode code point of the individual characters:

Python

```
>>> bytes([65, 66, 67, 97, 98, 99])
b'ABCabc'
```

In this example, each number in the list you use as an argument to `bytes()` is the code point for a specific letter. For example, `65` is the code point for `A`, `66` for `B`, and so on. You can get the Unicode code point of any character using the built-in `ord()` function.

Note: To learn more about working with bytes objects, check out the Bytes Objects: Handling Binary Data in Python tutorial.

The Built-in `bytearray()` Function

Python doesn't have dedicated literal syntax for `bytearray` objects. To create them, you'll always use the class constructor `bytearray()`, which is also known as a built-in function in Python. Here are a few examples of how to create `bytearray` objects using this function:

Python

```
>>> bytearray()
bytearray(b'')

>>> bytearray(5)
bytearray(b'\x00\x00\x00\x00\x00')

>>> bytearray([65, 66, 67, 97, 98, 99])
bytearray(b'ABCabc')

>>> bytearray(b"Using a bytes literal")
bytearray(b'Using a bytes literal')
```

In the first example, you call `bytearray()` without an argument to create an empty `bytearray` object. In the second example, you call the function with an integer as an argument. In this case, you create a `bytearray` with five zero-filled items.

Next, you use a list of code points to create a `bytearray`. This call works the same as with `bytes` objects. Finally, you use a `bytes` literal to build up the `bytearray` object.

Bytes and Bytearray Methods

In Python, `bytes` and `bytearray` objects are quite similar to strings. Instead of being sequences of characters, `bytes` and `bytearray` objects are sequences of integer numbers, with values from `0` to `255`.

Because of their similarities with strings, the `bytes` and `bytearray` types support mostly the same methods as strings, so you won't repeat them in this section. If you need detailed explanations of specific methods, then check out the Bytes and Bytearray Operations section in Python's documentation.

Finally, both `bytes` and `bytearray` objects support the common sequence operations that you learned in the Common Sequence Operations on Strings section.

Booleans

Boolean logic relies on the **truth value** of expressions and objects. The truth value of an expression or object can take one of two possible values: **true** or **false**. In Python, these two values are represented by `True` and `False`, respectively:

Python

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

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

Both `True` and `False` are instances of the `bool` data type, which is built into Python. In the following sections, you'll learn the basics about Python's `bool` data type.

Boolean Literals

Python provides a built-in Boolean data type. Objects of this type may have one of two possible values: `True` or `False`. These values are defined as built-in constants with values of `1` and `0`, respectively. In practice, the `bool` type is a subclass of `int`. Therefore, `True` and `False` are also instances of `int`:

Python

```
>>> issubclass(bool, int)
True

>>> isinstance(True, int)
True
>>> isinstance(False, int)
True

>>> True + True
2
```

In Python, the `bool` type is a subclass of the `int` type. It has only two possible values, `0` and `1`, which map to the constants `False` and `True`.

These constant values are also the literals of the `bool` type:

Python

```
>>> True
True

>>> False
False
```

Boolean objects that are equal to `True` are truthy, and those equal to `False` are falsy. In Python, non-Boolean objects also have a truth value. In other words, Python objects are either truthy or falsy.

The Built-in `bool()` Function

You can use the built-in `bool()` function to convert any Python object to a Boolean value. Internally, Python uses the following rules to identify falsy objects:

- Constants that are defined to be false: `None` and `False`
- The zero of any numeric type: `0`, `0.0`, `0j`, `Decimal(0)`, `Fraction(0, 1)`
- Empty sequences and collections: `''`, `()`, `[]`, `{}`, `set()`, `range(0)`

The rest of the objects are considered truthy in Python. You can use the built-in `bool()` function to explicitly learn the truth value of any Python object:

Python

```
>>> bool(0)
False
>>> bool(42)
True

>>> bool(0.0)
False
>>> bool(3.14)
True

>>> bool("")
False
>>> bool("Hello")
True

>>> bool([])
False
>>> bool([1, 2, 3])
True
```

In these examples, you use `bool()` with arguments of different types. In each case, the function returns a Boolean value corresponding to the object's truth value.

Note: You rarely need to call `bool()` yourself. Instead, you can rely on Python calling `bool()` under the hood when necessary. For example, you can say `if numbers:` instead of `if bool(numbers):` to check whether `numbers` is truthy.

You can also use the `bool()` function with custom classes:

Python

```python
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...

>>> point = Point(2, 4)
>>> bool(point)
True
```

By default, all instances of custom classes are true. If you want to modify this behavior, you can use the `.__bool__()` special method. Consider the following update of your `Point` class:

Python `point.py`

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __bool__(self):
        if self.x == self.y == 0:
            return False
        return True
```

The `.__bool__()` method returns `False` when both coordinates are equal to `0` and `True` otherwise. Here's how your class works now:

Python

```python
>>> from point import Point

>>> origin = Point(0, 0)
>>> bool(origin)
False

>>> point = Point(2, 4)
>>> bool(point)
True
```

Now, when both coordinates are `0`, you get `False` from calling `bool()`. For the rest of the points, you get `True`.

Conclusion

You've learned about the basic built-in **data types** that Python provides. These types are the building blocks of most Python programs. With them, you can represent numeric, textual, byte, and Boolean data.

In this tutorial, you've learned about:

- Python's **numeric** types, such as **`int`**, **`float`**, and **`complex`**
- The **`str`** data type, which represents **textual** data in Python
- The **`bytes`** and **`bytearray`** data types for storing **bytes**
- **Boolean** values with Python's **`bool`** data type

With this knowledge, you're ready to start using all of the basic data types that are built into Python.

Get Your Code: **Click here to download the free sample code** that you'll use to learn about basic data types in Python.

Frequently Asked Questions

Now that you have some experience with Python's basic data types, you can use the questions and answers below to check your understanding and recap what you've learned.

These FAQs are related to the most important concepts you've covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer.

What are the basic data types in Python? Show/Hide

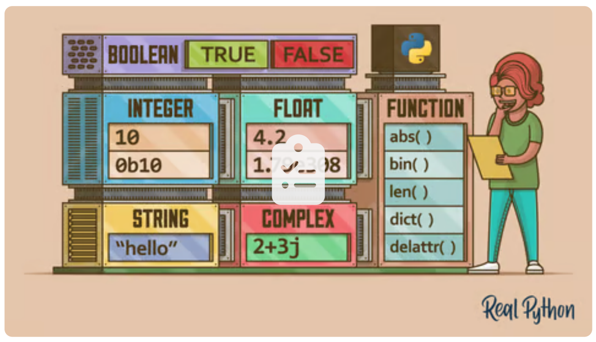
How can I check the type of a variable in Python? Show/Hide

How do I convert one data type into another in Python? Show/Hide

How do I perform type checking or validation in Python? Show/Hide

What's the taxonomy of data types in Python? Show/Hide

Take the Quiz: Test your knowledge with our interactive "Basic Data Types in Python: A Quick Exploration" quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Basic Data Types in Python: A Quick Exploration

Take this quiz to test your understanding of the basic data types that are built into Python, like numbers, strings, bytes, and Booleans.

Mark as Completed Share

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: **Exploring Basic Data Types in Python**

About **Leodanis Pozo Ramos**

Leodanis is a self-taught Python developer, educator, and technical writer with over 10 years of experience.

» More about Leodanis

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:

Aldren

Brenda

Dan

Geir Arne

Jon

Joanna

John

Martin

What Do You Think?

Rate this article:

LinkedIn Twitter Bluesky Facebook Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Topics: basics python

Recommended Video Course: Exploring Basic Data Types in Python

Related Tutorials:

- Variables in Python: Usage and Best Practices
- Conditional Statements in Python
- Operators and Expressions in Python
- Defining Your Own Python Function
- Strings and Character Data in Python