Q1. Does assigning a value to a string's indexed character violate Python's string immutability?

One final thing that makes strings different from some other Python collection types is that you are not allowed to modify the individual characters in the collection. It is tempting to use the [ ] operator on the left side of an assignment, with the intention of changing a character in a string. For example, in the following code, we would like to change the first letter of greetings.

greeting = "Hello, world!"

greeting[0] = 'J' # ERROR!

print(greeting)

Instead of producing the output jello, world!, this code produces the runtime error type: ‘ str ‘ object does not support item assignment .

Strings are **immutable**, which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original.

greeting = "Hello, world!"

newGreeting = 'J' + greeting[1:]

print(newGreeting)

print(greeting) # same as it was

Q2. Does using the += operator to concatenate strings violate Python's string immutability? Why or why not?

I'm using Python 3.8.3 & I got some unexpected output like below when checking id of strings.

>>> a="d"

>>> id(a)

1984988052656

>>> a+="e"

>>> id(a)

1985027888368

>>> a+="h"

>>> id(a)

1985027888368

>>> a+="i"

>>> id(a)

1985027888368

>>>

After the line which adding "h" to a, id(a) didn't change. How is that possible when strings are immutable ? I got this same output when I use a=a+"h" instead of a+="h" and run this code in a .py file also(I mentioned that because there is some situations we can see different output when running in the shell and running same code after save to a file)

There's a weird string concatenation optimization that does this. It violates the rules of how ID values and += are supposed to work - the ID values produced with the optimization in place would be not only impossible, but prohibited, with the unoptimized semantics - but the developers care more about people who would see bad concatenation performance and assume Python sucks.

Q3. In Python, how many different ways are there to index a character?

Like the Data type that has items that correspond to an index number, each of a string’s characters also correspond to an index number, starting with the index number 0.

For the string Sammy shark! the index breakdown is like this:

| **S** | **a** | **m** | **m** | **y** |  | **S** | **h** | **a** | **r** | **k** | **!** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

As you can see, the first S starts at index 0, and the string ends at index 11 with the ! symbol.

We also notice that the whitespace character between SAMMY and SHARK also corresponds with its own index number. In this case, the index number associated with the whitespace is 5.

The exclamation point (!) also has an index number associated with it. Any other symbol or punctuation mark, such as \*#&@, is also a character and would be associated with its own index number.

The fact that each character in a Python string has a corresponding index number allows us to access and manipulate strings in the same ways we can with other sequential data types.

**Accessing Characters by Positive Index Number**

By referencing index numbers, we can isolate one of the characters in a string. We do this by putting the index numbers in square brackets. Let’s declare a string, print it, and call the index number in square brackets:

When we refer to a particular index number of a string, Python returns the character that is in that position. Since the letter y is at index number 4 of the string ss = “Sammy shark “ when we print ss [4] we receive y as the output.

Index numbers allow us to access specific characters within a string.

**Accessing Characters by Negative Index Number**

If we have a long string and we want to pinpoint an item towards the end, we can also count backwards from the end of the string, starting at the index number -1.

For the same string Sammy shark! the negative index breakdown is like this:

| **S** | **a** | **m** | **m** | **y** |  | **S** | **h** | **a** | **r** | **k** | **!** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

By using negative index numbers, we can print out the character r, by referring to its position at the -3 index, like so:

Q4. What is the relationship between indexing and slicing?

I was preparing to demonstrate indexing and slicing lists to a group of fellow Python beginners a few days ago, and I got stuck on what seemed like a couple of pretty basic use cases. So after poking around a bit to get un-stuck, I figured it was worth sharing what I learned.

Accessing the items in a list (and in other [iterables](https://stackoverflow.com/questions/9884132/what-exactly-are-iterator-iterable-and-iteration" \t "_blank) like tuples and strings) is a fundamental skill for Python coders, and many Python tools follow similar conventions for indexing and slicing (e.g. numpy Arraysand pandas DataFrames). So it’s worth being familiar with the ins and outs.

Definitions and Stage-Setting

“Indexing” means referring to an element of an iterable by its position within the iterable. “Slicing” means getting a subset of elements from an iterable based on their indices.

By way of analogy, I was recently summoned to jury duty, and they assigned each potential juror a number. You might say my juror number was my index. When they said, “Juror number 42; please stand,” I knew they were talking to me. When they said, “Jurors 37 through 48, please report back at 3:30 pm,” that slice of the larger group collectively sighed and went to lun

Indexing

To retrieve an element of the list, we use the index operator ([])

My list [0]

‘a’

Lists are “zero indexed”, so [0]r eturns the zero-th (i.e. the left-most) item in the list, and [1] returns the one-th item (i.e. one item to the right of the zero-th item). Since there are 9 elements in our list ([0] through [8]), attempting to, since it is actually trying to get the tenth element, and there isn’t one.

Python also allows you to index from the end of the list using a negative number, where [-1] returns the last element. This is super-useful since it means you don’t have to programmatically find out the length of the iterable in order to work with elements at the end of it. The indices and reverse indices of my list are as follows:

0 1 2 3 4 5 6 7 8  
 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓  
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']  
 ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑  
 -9 -8 -7 -6 -5 -4 -3 -2 -1

Slicing

A slice is a subset of list elements. In the case of lists, a single slice will always be of contiguous elements. Slice notation takes the form

my\_list[start:stop]

where start is the index of the first element to include, and stop is the index of the item to stop at without including it in the slice. So my\_list[1:5] returns ['b', 'c', 'd', 'e']:

0 1 2 3 4 5 6 7 8  
 × ↓ ↓ ↓ ↓ × × × ×  
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

Leaving either slice boundary blank means start from (or go to) the end of the list. For example:

my\_list[5:]['f', 'g', 'h', 'i']  
my\_list[:4]['a', 'b', 'c', 'd']

Using a negative indexer will set the start/stop bounds relative to their position from the end of the list, so my\_list[-5:-2] returns ['e', 'f', 'g']:

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']  
 × × × × ↑ ↑ ↑ × ×  
 -9 -8 -7 -6 -5 -4 -3 -2 -1

Note that if you try my\_list[-2:-5], you’ll get an empty list. This was something I got tripped up on, but here’s what’s happening: in order to be included in the slice, an element must be at or to the right of the start boundary AND to the left of the stop boundary. Because the -2 is already to the right of -5, the slicer stops before populating any value into the slice.

A for loop works exactly the same way; the first loop below has no output, but the second does:

for i in range(-2,-5):  
 print(i)for i in range(-5,-2):  
 print(i)-5  
-4  
-3

Stepping

The slicer can take an optional third argument, which sets the interval at which elements are included in the slice. So my\_list[::2] returns ['a', 'c', 'e', 'g', 'i']:

0 1 2 3 4 5 6 7 8  
 ↓ ×¹ ↓² ×¹ ↓² ×¹ ↓² ×¹ ↓²  
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

And my\_list[1::2] returns ['b', 'd', 'f', 'h']:

0 1 2 3 4 5 6 7 8  
 × ↓ ×¹ ↓² ×¹ ↓² ×¹ ↓² ×¹  
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

Negative step values reverse the direction in which the slicer iterates through the original list:

my\_list[::-1]['i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']

The indexed positions of list elements don’t change, but the order in which the elements are returned does. The sense of the start and stop boundaries is also reversed, so the start value should be the right-most position in the slice, and the stop value should be to the left of that. So my\_list[5:3:-1] gives us [‘f’, ‘e’]:

<----<----<----<----<----<----<----<----<--  
 0 1 2 3 4 5 6 7 8  
 × × × × ↓ ↓ × × ×  
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

Likewise, my\_list[-2:-5:-1] gives us ['h', 'g', 'f']:

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']  
 × × × × × ↑ ↑ ↑ ↑  
 -9 -8 -7 -6 -5 -4 -3 -2 -1  
<----<----<----<----<----<----<----<----<--

And my\_list[-1:-8:-3] gives us ['i', 'f', 'c']:

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']  
 × ×¹ ↑³ ×² ×¹ ↑³ ×² ×¹ ↑  
 -9 -8 -7 -6 -5 -4 -3 -2 -1  
<----<----<----<----<----<----<----<----<--

I hope this saves you some of the confusion I encountered while working through indexing and slicing, especially with negative steps.

Q5. What is an indexed character's exact data type? What is the data form of a slicing-generated substring?

You can get the character at a particular index within a string, string buffer, or string builder by invoking the charAt accessor method. The index of the first character is 0; the index of the last is length()-1. For example, the following code gets the character at index 9 in a string:

String anotherPalindrome = "Niagara. O roar again!";

char aChar = anotherPalindrome.charAt(9);

Indices begin at 0, so the character at index 9 is 'O',

Use the charAt method to get a character at a particular index.

The figure also shows that to compute the index of the last character of a string, you have to subtract 1 from the value returned by the length method.

If you want to get more than one character from a string, string buffer, or string builder, you can use the substring method. The substring method has two versions, as shown in the following table:

|  |  |
| --- | --- |
| \*The substring Methods in the String, StringBuffer and StringBuilder Classes | |
| Method | Description |
| String substring(int) String substring(int, int) | Returns a new string that is a substring of this string, string buffer, or string builder.The first integer argument specifies the index of the first character. The second integer argument is the index of the last character -1. The length of the substring is therefore the second int minus the first int. If the second integer is not present, the substring extends to the end of the original string. |

\* The substring methods were added to the StringBuffer class for Java 2 SDK 1.2.

The following code gets from the Niagara palindrome the substring that extends from index 11 to index 15, which is the word "roar":

String anotherPalindrome = "Niagara. O roar again!";

String roar = anotherPalindrome.substring(11, 15);

Use the substring method to get part of a string, string buffer, or string builder. Remember that indices begin at 0.

the data form of a slicing-generated substring?\

**What Is a Substring in Python?**

Part of a string is known as a substring. There are multiple methods for creating a substring in python and checking if a substring is present in a string, index of a substring, and more. Let us look at various operations related to substrings.

How to Create a Substring

A substring can be created using many ways. One of the popular ways is string slicing.

To create a list of substrings based on a specific delimiter present in a string, we use the split() function.

A substring in python can be generated by slicing as follows:

string[begin: end: step]

where,

begin: is the starting index of the substring. The substring includes this element. If begin is not mentioned while slicing, it is assumed as 0.

Step:  is the character that has to be included after the current character. The default value of the step is one and is assumed as one if not mentioned.

End: is the ending index of the substring. This element is excluded in the substring. It is assumed to be the same as the string's length if the value is not specified or the given value exceeds the string length.

The syntax for slicing a string:

1. string[begin: end]: substring includes characters from start to end-1. Example:

s= 'substring in python.'

s[2:10]

Output: ‘bstring’

2. string[: end]: substring includes characters from start to end-1. Example for slicing without begin index:

s= ' substring in python.'

s[:7]

Output: 'substri

3. string[begin:]: substring includes characters from the begin index to the end of the string. Example for slicing without end index:

s= ‘substring in python.'

s[1:]

Output: ‘ubstring in python'

4. string[begin:end: step]: substring includes characters from start to end-1 excusing every step character. Example for slicing with step-index:

s= ‘substring in python'

s[2:10:2]

Output: ‘btig’

5. s[:]- substring includes the entire string. Example for slicing without begin or end index :

s= ‘substring in python'

s[:]

Output: 'usbstring in python'

6. s[index]: substring includes a single character. Example for getting character at a given index:

s= ‘substring in python'

s[1]

Output : ‘u’

7. Negative slicing- Use a negative index for getting a substring. Example:

s= ' substring in python '

s[0:-3]

Output: ‘substring in pyt’

8. Reversing a string- Slicing can be used to return the reverse of string using a negative step. Example:

s= 'substring in python'

s[::-1]

Output: ‘nohtyp ni gnirtsbus’

Code to demonstrate the creation of substring in python:

s= ' substring in python '

# create a substring using slicing

a = s[11:]

print(a)

# Create list of substrings using split function

b = s.split()

print(b)

Q6. What is the relationship between string and character "types" in Python?

objects that contain sequences of character data. Processing character data is integral to programming. It is a rare application that doesn’t need to manipulate strings at least to some extent.

String Operators

You have already seen the operators + and \* applied to numeric operands in the tutorial operators and expressions in python . These two operators can be applied to strings as well.

The + Operator

The + operator concatenates strings. It returns a string consisting of the operands joined together, as shown here:

>>>

>>> s = 'foo'

>>> t = 'bar'

>>> u = 'baz'

>>> s + t

'foobar'

>>> s + t + u

'foobarbaz'

>>> print('Go team' + '!!!')

Go team!!!

The \* Operator

The \* operator creates multiple copies of a string. If s is a string and n is an integer, either of the following expressions returns a string consisting of n concatenated copies of s:

s \* n  
n \* s

Here are examples of both forms:

>>>

>>> s = 'foo.'

>>> s \* 4

'foo.foo.foo.foo.'

>>> 4 \* s

'foo.foo.foo.foo.'

The multiplier operand n must be an integer. You’d think it would be required to be a positive integer, but amusingly, it can be zero or negative, in which case the result is an empty string:

>>>

>>> 'foo' \* -8

''

If you were to create a string variable and initialize it to the empty string by assigning it the value 'foo' \* -8, anyone would rightly think you were a bit daft. But it would work.

The in Operator

Python also provides a membership operator that can be used with strings. The in operator returns True if the first operand is contained within the second, and False otherwise:

>>>

>>> s = 'foo'

>>> s in 'That\'s food for thought.'

True

>>> s in 'That\'s good for now.'

False

There is also a not in operator, which does the opposite:

>>>

>>> 'z' not in 'abc'

True

>>> 'z' not in 'xyz'

False

Built-in String Functions

As you saw in the tutorial on [Basic Data Types in Python](https://realpython.com/python-data-types/), Python provides many functions that are built-in to the interpreter and always available. Here are a few that work with strings:

| Function | Description |
| --- | --- |
| chr() | Converts an integer to a character |
| ord() | Converts a character to an integer |
| len() | Returns the length of a string |
| str() | Returns a string representation of an object |

These are explored more fully below.

ord(c)

Returns an integer value for the given character.

At the most basic level, computers store all information as [numbers](https://realpython.com/python-numbers/). To represent character data, a translation scheme is used which maps each character to its representative number.

The simplest scheme in common use is called [ASCII](https://en.wikipedia.org/wiki/ASCII). It covers the common Latin characters you are probably most accustomed to working with. For these characters, ord(c) returns the ASCII value for character c:

>>>

>>> ord('a')

97

>>> ord('#')

35

ASCII is fine as far as it goes. But there are many different languages in use in the world and countless symbols and glyphs that appear in digital media. The full set of characters that potentially may need to be represented in computer code far surpasses the ordinary Latin letters, numbers, and symbols you usually see.

[Unicode](http://www.unicode.org/standard/WhatIsUnicode.html) is an ambitious standard that attempts to provide a numeric code for every possible character, in every possible language, on every possible platform. Python 3 supports Unicode extensively, including allowing Unicode characters within strings.

For More Information: See [Unicode & Character Encodings in Python: A Painless Guide](https://realpython.com/python-encodings-guide/) and [Python’s Unicode Support](https://docs.python.org/3/howto/unicode.html#python-s-unicode-support) in the Python documentation.

As long as you stay in the domain of the common characters, there is little practical difference between ASCII and Unicode. But the ord() function will return numeric values for [Unicode characters](https://realpython.com/courses/python-unicode/) as well:

>>>

>>> ord('€')

8364

>>> ord('∑')

8721

chr(n)

Returns a character value for the given integer.

chr() does the reverse of ord(). Given a numeric value n, chr(n) returns a string representing the character that corresponds to n:

>>>

>>> chr(97)

'a'

>>> chr(35)

'#'

chr() handles Unicode characters as well:

>>>

>>> chr(8364)

'€'

>>> chr(8721)

'∑'

len(s)

Returns the length of a string.

With len(), you can check Python string length. len(s) returns the number of characters in s:

>>>

>>> s = 'I am a string.'

>>> len(s)

14

str(obj)

Returns a string representation of an object.

Virtually any object in Python can be rendered as a string. str(obj) returns the string representation of object obj:

>>>

>>> str(49.2)

'49.2'

>>> str(3+4j)

'(3+4j)'

>>> str(3 + 29)

'32'

>>> str('foo')

'foo'

Q7. Identify at least two operators and one method that allow you to combine one or more smaller strings to create a larger string.

Concatenation operators join multiple strings into a single string. There are two concatenation operators, + and & . Both carry out the basic concatenation operation

You have several small strings that you need to combine into one larger string.

Solution

The + operator concatenates strings and therefore offers seemingly obvious solutions for putting small strings together into a larger one. For example, when you have all the pieces at once, in a few variables:

largeString = small1 + small2 + ' something ' + small3 + ' yet more'

Or when you have a sequence of small string pieces:

largeString = ''

for piece in pieces:

largeString += piece

Or, equivalently, but a bit more compactly:

import operator

largeString = reduce(operator.add, pieces, '')

However, none of these solutions is generally optimal. To put together pieces stored in a few variables, the string-formatting operator % is often best:

largeString = '%s%s something %s yet more' % (small1, small2, small3)

To join a sequence of small strings into one large string, the string operator join is invariably best:

largeString = ''.join(pieces)

Discussion

In Python, string objects are immutable. Therefore, any operation on a string, including string concatenation, produces a new string object, rather than modifying an existing one. Concatenating N strings thus involves building and then immediately throwing away each of N-1 intermediate results. Performance is therefore quite a bit better for operations that build no intermediate results, but rather produce the desired end result at once. The string-formatting operator % is one such operation, particularly suitable when you have a few pieces (for example, each bound to a different variable) that you want to put together, perhaps with some constant text in addition. In addition to performance, which is never a major issue for this kind of task, the % operator has several potential advantages when compared to an expression that uses multiple + operations on strings, including readability, once you get used to it. Also, you don’t have to call str on pieces that aren’t already strings (e.g., numbers) because the format specifier %s does so implicitly. Another advantage is that you can use format specifiers other than %s, so that, for example, you can control how many significant digits the string form of a floating-point number should display.

When you have many small string pieces in a sequence, performance can become a truly important issue. The time needed for a loop using + or += (or a fancier but equivalent approach using the built-in function reduce) tends to grow with the square of the number of characters you are accumulating, since the time to allocate and fill a large string is roughly proportional to the length of that string. Fortunately, Python offers an excellent alternative. The join method of a string object s takes as its only argument a sequence of strings and produces a string result obtained by joining all items in the sequence, with a copy of s separating each item from its neighbors. For example, ''.join(pieces) concatenates all the items of pieces in a single gulp, without interposing anything between them. It’s the fastest, neatest, and most elegant and readable way to put a large string together.

Even when your pieces come in sequentially from input or computation, and are not already available as a sequence, you should use a list to hold the pieces. You can prepare that list with a list comprehension or by calling the append or extend methods. At the end, when the list of pieces is complete, you can build the string you want, typically with ''.join(pieces). Of all the handy tips and tricks I could give you about Python strings, I would call this one the most significant

Q8. What is the benefit of first checking the target string with in or not in before using the index method to find a substring?

If you’re new to programming or come from a programming language other than Python, you may be looking for the best way to check whether a string contains another string in Python.

Identifying such [substrings](https://en.wikipedia.org/wiki/Substring) comes in handy when you’re working with [text content from a file](https://realpython.com/read-write-files-python/) or after you’ve [received user input](https://realpython.com/python-input-output/). You may want to perform different actions in your program depending on whether a substring is present or not.

In this tutorial, you’ll focus on the most Pythonic way to tackle this task, using the membership operator in. Additionally, you’ll learn how to identify the right string methods for related, but different, use cases.

Finally, you’ll also learn how to find substrings in pandas columns. This is helpful if you need to search through data from a CSV file. You could use the approach that you’ll learn in the next section, but if you’re working with tabular data, it’s best to load the data into a pandas DataFrame and [search for substrings in pandas](https://realpython.com/python-string-contains-substring/#find-a-substring-in-a-pandas-dataframe-column).

The membership operator in is a great way to descriptively check whether there’s a substring in a string, but it doesn’t give you any more information than that. It’s perfect for conditional checks—but what if you need to know more about the substrings?

Python provides many additonal string methods that allow you to check how many target substrings the string contains, to search for substrings according to elaborate conditions, or to locate the index of the substring in your text.

In this section, you’ll cover some additional string methods that can help you learn more about the substring.

Note: You may have seen the following methods used to check whether a string contains a substring. This is possible—but they aren’t meant to be used for that!

Programming is a creative activity, and you can always find different ways to accomplish the same task. However, for your code’s readability, it’s best to use methods as they were intended in the language that you’re working with.

By using in, you confirmed that the string contains the substring. But you didn’t get any information on where the substring is located.

If you need to know where in your string the substring occurs, then you can use .index() on the string object:

>>>

>>> file\_content = """hi there and welcome.

... this is a special hidden file with a secret secret.

... i don't want to tell you the secret,

... but i do want to secretly tell you that i have one."""

>>> file\_content.index("secret")

59

When you call .index() on the string and pass it the substring as an argument, you get the index position of the first character of the first occurrence of the substring.

Note: If Python can’t find the substring, then .index() raises a ValueError [exception](https://realpython.com/python-exceptions/).

But what if you want to find other occurrences of the substring? The .index() method also takes a second argument that can define at which index position to start looking. By passing specific index positions, you can therefore skip over occurrences of the substring that you’ve already identified:

>>>

>>> file\_content.index("secret", 60)

66

When you pass a starting index that’s past the first occurrence of the substring, then Python searches starting from there. In this case, you get another match and not a ValueError.

That means that the text contains the substring more than once. But how often is it in there?

You can use .count() to get your answer quickly using descriptive and idiomatic Python code:

>>>

>>> file\_content.count("secret")

4

You used .count() on the lowercase string and passed the substring "secret" as an argument. Python counted how often the substring appears in the string and returned the answer. The text contains the substring four times. But what do these substrings look like?

You can inspect all the substrings by splitting your text at default word borders and printing the words to your terminal using a [for loop](https://realpython.com/python-for-loop/):

>>>

>>> for word in file\_content.split():

... if "secret" in word:

... print(word)

...

secret

secret.

secret,

secretly

In this example, you use [.split()](https://realpython.com/python-string-split-concatenate-join/) to separate the text at whitespaces into strings, which Python packs into a list. Then you iterate over this list and use in on each of these strings to see whether it contains the substring "secret".

Note: Instead of printing the substrings, you could also save them in a new list, for example by using a list comprehension with a conditional expression:

>>>

>>> [word for word in file\_content.split() if "secret" in word]

['secret', 'secret.', 'secret,', 'secretly']

In this case, you build a list from only the words that contain the substring, which essentially filters the text.

Now that you can inspect all the substrings that Python identifies, you may notice that Python doesn’t care whether there are any characters after the substring "secret" or not. It finds the word whether it’s followed by whitespace or punctuation. It even finds words such as "secretly".

Q9. Which operators and built-in string methods produce simple Boolean (true/false) results?

The Python Boolean type is one of Python’s [built-in data types](https://realpython.com/python-data-types/). It’s used to represent the truth value of an expression. For example, the expression 1 <= 2 is True, while the expression 0 == 1 is False. Understanding how Python Boolean values behave is important to programming well in Python.

The Python Boolean type has only two possible values:

True

False

No other value will have bool as its type. You can check the type of True and False with the built-in type():

>>>

>>> type(False)

<class 'bool'>

>>> type(True)

<class 'bool'>

The type() of both False and True is bool.

The type bool is built in, meaning it’s always available in Python and doesn’t need to be imported. However, the name itself isn’t a keyword in the language. While the following is considered bad style, it’s possible to assign to the name bool:

>>>

>>> bool

<class 'bool'>

>>> bool = "this is not a type"

>>> bool

'this is not a type'

Although technically possible, to avoid confusion it’s highly recommended that you don’t assign a different value to bool.

Python Booleans as Keywords

Built-in names aren’t keywords. As far as the Python language is concerned, they’re regular [variables](https://realpython.com/python-variables/). If you assign to them, then you’ll override the built-in value.

In contrast, the names True and False are not built-ins. They’re keywords. Unlike many other [Python keywords](https://realpython.com/python-keywords/), True and False are Python expressions. Since they’re expressions, they can be used wherever other expressions, like 1 + 1, can be used.

It’s possible to assign a Boolean value to variables, but it’s not possible to assign a value to True:

>>>

>>> a\_true\_alias = True

>>> a\_true\_alias

True

>>> True = 5

File "<stdin>", line 1

SyntaxError: cannot assign to True

Because True is a keyword, you can’t assign a value to it. The same rule applies to False:

>>>

>>> False = 5

File "<stdin>", line 1

SyntaxError: cannot assign to False

You can’t assign to False because it’s a keyword in Python. In this way, True and False behave like other numeric constants. For example, you can pass 1.5 to functions or assign it to variables. However, it’s impossible to assign a value to 1.5. The statement 1.5 = 5 is not valid Python. Both 1.5 = 5 and False = 5 are invalid Python code and will raise a [SyntaxError](https://realpython.com/invalid-syntax-python/) when parsed.