

Strings

Strings

In this section we shall take a closer look at the string type and some of the operations associated with them. The following section makes heavy reference to online notes by Dr. Andrew N. Harrington, [Hands-on Python 3 Tutorial](#) released under the [CC BY-NC-SA 4.0](#) license.

Concatenation +

For strings the + symbol is used to concatenate two strings together. For example:

```
[3]: print('One string' + ' and another')
```

One string and another

Duplication *

The duplication * operator takes a string and an integer and repeats the string as many times as the integer value:

```
[6]: print('hello '*4)
      print(2*'bye ')
```

hello hello hello hello
bye bye

Indexing []

Strings can be seen as a collection of characters. Each of these character has an integer index associated with it, based on it's position in the string. For example, take the string 'computer':

character

c
o
m
p
u
t

e
r
index

0
1
2
3
4
5
6
7

You can access individual characters in the string by index using:

`string[index]`

for example:

```
[1]: computer_string = 'computer'

print('Index 3:', computer_string[3])

print('Index 7:', computer_string[7])
```

Index 3: p

Index 7: r

If you use an index that is too large for the given string, Python will return an error:

```
[8]: print('Index 11', computer_string[11])
```

```

      □
↳ -----

IndexError                                Traceback (most recent call↳
↳ last)

    <ipython-input-8-abeba3add71f> in <module>()
----> 1 print('Index 11', computer_string[11])

IndexError: string index out of range
```

You can find the number of characters in a string using the `len()` function:

```
[9]: print('There are', len(computer_string), 'characters in the string')
```

There are 8 characters in the string

Notice how the length of `computer_string` is one greater than its largest index. This is because Python indexes from 0.

Thus, if we don't know how long a string is before hand (if a variable holding a string is subject to change for instance) and we want to index the last value of the string, we could use `len() - 1` as the index:

```
[11]: print('The last character:', computer_string[len(computer_string) - 1])
```

The last character: r

This method works, but Python gives us a far cleaner way of doing this: using an index of `-1`. This won't work for most other programming languages.

```
[12]: print('The last character:', computer_string[-1])
```

The last character: r

In general, negative indices in Python index the strings (and other objects) backwards:

```
[13]: print('Second last character', computer_string[-2])  
  
print('Third last character', computer_string[-3])
```

Second last character e

Third last character t

Note that the index `-8` corresponds to the 0 index (`len(computer_string) - 8` is 0) so anything less than this would be out of bounds.

Slicing

Slicing allows us to extract segments of the string, as apposed to individual characters. The syntax for string slicing is:

```
string[start_index:stop_index]
```

where the `stop_index` is not included in the slice, rather the slice stops before this index. For example, consider the slice:

```
[14]: print(computer_string[2:5])
```

mpu

where the last character is 'u', but the character with index 5 is 't'.

If we want to take a slice from the beginning of a string we could use 0 as the `start_index`:

```
[21]: print(computer_string[0:3])
```

com

Alternatively if we left the `start_index` blank Python will interpret this as starting from the beginning of the string:

```
[22]: print(computer_string[:3])
```

com

Similarly if we wanted to take a slice up to and including the last character in the string, we can use:

```
[25]: print(computer_string[3:len(computer_string)])
```

puter

or simply leave the `stop_index` blank:

```
[24]: print(computer_string[3:])
```

puter

Notice the slice above is not the same as if we used `-1` as the `stop_index`:

```
[27]: print(computer_string[3:-1])
```

pute

even though the same rules apply as with indexing, the slice always stops **before** the `stop_index`.

We can use a third index when slicing as a step size:

```
string[start_index: stop_index: step_size]
```

For example, we can get every second character from a string using a step size of 2:

```
[20]: print('Starting from 0:', computer_string[0:8:2])
      print('Starting from 1:', computer_string[1:8:2])
```

Starting from 0: cmue

Starting from 1: optr

The step size can be any integer. Note that by default it is set to 1. As another example lets print out every second character from `computer_string` starting from the first:

```
[4]: print(computer_string[::3])
```

cpe

The step size need not be positive. If a negative step size is used the string will be sliced backwards. For example if we want to print out the whole of `computer_string` backwards:

```
[6]: print(computer_string[::-1])
```

retupmoc

Note, when slicing with a negative step size you must ensure that **start_index** is greater than **stop_index**, otherwise your slice will be empty.

```
[9]: print('Empty slice:', computer_string[0:6:-1])  
     print('Not empty slice:', computer_string[6:0:-1])
```

Empty slice:

Not empty slice: etupmo

Also notice how, in the second slice above, the 0 index character is not present. Even when slicing with a negative step size the **stop_index** is **not** included in the slice.