

Strings

Chapter 6

This chapter covers

- Understanding strings as sequences of characters
- Using basic string operations
- Inserting special characters and escape sequences
- Converting from objects to strings
- Formatting strings
- Using the byte type

```
53 x = "Hello, World"
54
55 print(x[0])
56 print(x[-1])
57 print(x[1:])
58
59 x = "Goodbye\n"
60 x = x[:-1]
61 print(x)
62 print(len("Goodbye"))
```

Strings as sequences of characters

- For the purposes of extracting characters and substrings, strings can be considered to be sequences of characters, which means that you can use index or slice notation.
- Python strings can't be modified.

Basic string operations

- The simplest (and probably most common) way to combine Python strings is to use the string concatenation operator +

```
64 x = "Hello " + "World"
65 print(x)
66
67 print(8 * "x")
```

Special characters and escape sequences

Escape sequence	Character represented
\'	Single-quote character
\"	Double-quote character
\\	Backlash character
\a	Bell character
\b	Backspace character
\f	Formfeed character
\n	Newline character
\r	Carriage-return character
\t	Tab character
\v	Vertical tab character

String methods

- join()
- split()

```
1  # join
2  print(" ".join(["join", "puts", "spaces", "between", "elements"]))
3  print(":".join(["Separated", "with", "colons"]))
4  print("".join(["Separated", "by", "nothing"]))
5
6  # split
7  x = "You\t\t can have tabs\t\n \t and newlines \n\n mixed in"
8  print(x.split())
9
10 x = "Mississippi"
11 print(x.split("ss"))
```



QUICK CHECK

- How could you use split and join to change all the whitespace in string x to dashes, such as changing “this is a test” to “this-is-a-test”?

Converting strings to numbers

- Use the functions `int` and `float` to convert strings to integer or floating-point numbers, respectively. \
- If they're passed a string that can't be interpreted as a number of the given type, these functions raise a `ValueError` exception.

```
1 x = float('123.456')
2
3 y = float('xxyy')    # ValueError
4
5 # octal
6 print(int('10000', 8))
7
8 # binary
9 print(int('101', 2))
10
11 # hex
12 print(int('ff', 16))
```




QUICK CHECK

- Which of the following will not be converted to numbers, and why?

`int('a1')`

`int('12G', 16)`

`float("12345678901234567890")`

`int("12*2")`

Getting rid of extra whitespace

```
1 x = " Hello, World\t\t "  
2  
3 print(x.strip())  
4 print(x.lstrip())  
5 print(x.rstrip())  
6  
7 x = "www.python.org"  
8 print(x.strip("w"))
```

- strip()
- lstrip()
- rstrip()



QUICK CHECK

- If the string `x` equals `"(name, date),\n"`, which of the following would return a string containing `"name, date"`?

`x.rstrip(",")`

`x.strip("),\n")`

`x.strip("\n")(,")`

String searching

```
1  x = "Mississippi"
2
3  print(x.find("ss"))
4  print(x.find("zz"))
5  print(x.find("ss", 3))
6  print(x.find("ss", 0, 3))
7  print(x.rfind("ss"))
8  print(x.count("ss"))
9  print(x.startswith("Miss"))
10 print(x.endswith("pi"))
```

- find()
- rfind()
- index()
- rindex()
- count()
- startswith()
- endswith()

Modifying strings

- Strings are immutable, but string objects have several methods that can operate on that string and return a new string that's a modified version of the original string.
- This provides much the same effect as direct modification for most purposes.

```
1  x = "Mississippi"
2  print(x.replace("ss", "+++"))
```

Modifying strings with list manipulations

- Strings are immutable objects.
- Turn the string into a list of characters, do whatever you want, and then turn the resulting list back into a string.

```
1  x = "Mississippi"
2  print(x.replace("ss", "+++"))
3
4  text = "Hello, World"
5  wordList = list(text)
6  print(wordList)
7
8  wordList[6:] = []
9  print(wordList)
10
11 wordList.reverse()
12 print(wordList)
13
14 text = "".join(wordList)
15 print(text)
```



QUICK CHECK

- What would be a quick way to change all punctuation in a string to spaces?

Converting from objects to strings

- In Python, almost anything can be converted to some sort of a string representation by using the built-in repr function.

```
1  print(repr([1, 2, 3]))
2
3  x = [1]
4  x.append(2)
5
6  print(x)
7  x.append([3, 4])
8  print(x)
```


Using the format method

```
1  print("{0} is the {1} of {2}".format("Ambrosia", "food", "the gods"))
2
3  print("{0} is the {1} of {2}".format("Ambrosia", "food", "the gods"))
4
5  print("{food} is the food of {user}".format(food="Ambrosia", user="the gods"))
6
7  print("{0} is the food of {user[1]}".format("Ambrosia", user=["men", "the gods", "others"]))
8
9  print("{0:10} is the food of gods".format("Ambrosia"))
10 print("{0:{1}} is the food of gods".format("Ambrosia", 10))
11 print("{food:{width}} is the food of gods".format(food="Ambrosia", width=10))
12 print("{0:>10} is the food of gods".format("Ambrosia"))
13 print("{0:&>10} is the food of gods".format("Ambrosia"))
```

Formatting strings with %

```
15 print("%s is the %s of %s" % ("Ambrosia", "food", "the gods"))
16
17 print("%s is the %s of %s" % ("Nectar", "drink", "gods"))
18
19 print("%s is the %s of the %s" % ("Brussels Sprouts", "food", "foolish"))
20
21 x = [1, 2, "three"]
22 print("The %s contains: %s" % ("list", x))
```

Using formatting sequences

- All formatting sequences are substrings contained in the string on the left side of the central %.
- Each formatting sequence begins with a percent sign and is followed by one or more characters that specify what is to be substituted for the formatting sequence and how the substitution is to be accomplished.

```
25  # sequences
26  print("Pi is <%-6.2f>" % 3.14159) # use of the formatting sequence: %-6.2f
```

Named parameters and formatting sequences

```
28 # named parameters
29 num_dict = {'e': 2.718, 'pi': 3.14159}
30 print("%(pi).2f - %(pi).4f - %(e).2f" % num_dict)
```



QUICK CHECK

- What would be in the variable x after the following snippets of code have executed?

```
x = "%.2f" % 1.1111
```

```
x = "%(a).2f" % {'a':1.1111}
```

```
x = "%(a).08f" % {'a':1.1111}
```

String interpolation

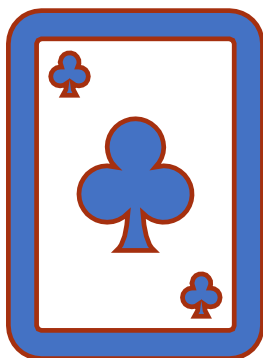
- Starting in Python 3.6, there's a way to create string constants containing arbitrary values, which is called string interpolation.
- String interpolation is a way to include the values of Python expressions inside literal strings.
- These f-strings, as they're commonly called because they are prefixed with f, use a syntax similar to that of the format method, but with a little less overhead.

```
32  # string interpolation
33  value = 42
34  message = f"The answer is {value}"
35  print(message)
```

Lab

Preprocessing Text

Summary



- Python strings have powerful text-processing features, including searching and replacing, trimming characters, and changing case.
- Strings are immutable; they can't be changed in place.
- Operations that appear to change strings actually return a copy with the changes.
- The re (regular expression) module has even more powerful string capabilities, which are discussed in chapter 16.