String Methods

Table of contents

Outline 1
Why learn string methods? 2
Built-in string methods 2
Methods 2
Interpreting `help` 3
Exercise: title case 4
Exercise: centering text 5
Splitting strings 6
Joining strings 7
Sanitizing data 7
Summary 7

Lecture 6A. Course logistics are in csse1001[1]. The full reference on these string methods is python-string-methods[2].

Outline

- Built-in string methods and `help()`
- Method invocation syntax
- Reading a `help` signature: optional parameters in square brackets
- Exercises: `title`, `center`
- Splitting and joining strings
- Sanitizing data with `strip`

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-string-methods.pdf

Why learn string methods?

Three points frame the topic:

1. During **file processing** (Lecture 6B, 2025-09-01-file-io[3]) we will be working extensively with strings.
2. We **are** able to replicate the functionality of *any* string method with our current tools.
3. So: strive to implement any string method you use **at least once**.

The point is not that these methods do something you couldn't write yourself; it's that you should know what's in the box before you rewrite it.

Built-in string methods

Python's `str` type has a myriad of built-in methods. Review them via `help(str)`:

```
>>> help(str)
...
| capitalize(self, /)
|     Return a capitalized version of the string.
|
| casefold(self, /)
|     Return a version of the string suitable for
|     caseless comparisons.
```

Type Enter to scroll and `q` to escape (quit) help.

Methods

Strings are **objects** — we'll eventually learn what that means fully (2025-09-09-object-oriented-programming[4]). Objects have **methods**, which are like functions but have a different invocation syntax. We don't say:

```
>>> capitalize("hello")
NameError: name 'capitalize' is not defined
```

but rather:

[3] courses/csse1001/2025-09-01-file-io.pdf
[4] courses/csse1001/2025-09-09-object-oriented-programming.pdf

```
>>> "hello".capitalize()   # Note the ()
'Hello'
```

The receiver goes *before* the dot, and the empty parentheses are still required even when there are no arguments.

For information on a particular method:

```
>>> help(str.find)
find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end]. Optional
    arguments start and end are interpreted as with slice notation.

    Return -1 on failure.
(END)
```

Interpreting `help`

The square brackets in the parameters for `find`

```
find(sub[, start[, end]])
```

indicate that the parameter is **optional**. This rule **recurses** — an optional parameter can itself have optional parameters — so the nesting says that `end` is only available once `start` has been given.

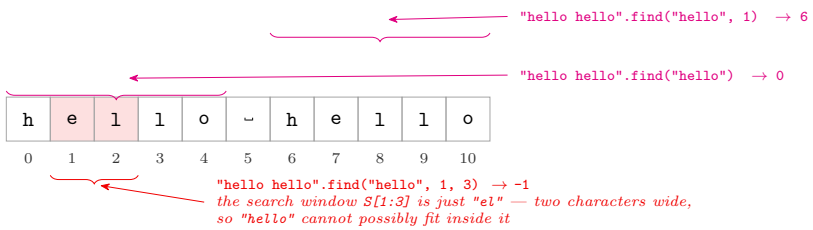
The comprehensive ways of calling `find` are therefore:

```
"team".find("I")
"team".find("I", 1)
"team".find("I", 1, -1)
```

whereas `find(1, -1)` is not allowed, since `sub` is required.

```
>>> "hello world".find("world", 6)
6
>>> "hello hello".find("hello")
0
>>> "hello hello".find("hello", 1)
6
>>> "hello hello".find("hello", 1, 3)
-1
```

The last two calls show what `start` and `end` do: they do not shift the answer, they shrink the haystack.



`start` and `end` are read exactly as slice notation: the search happens inside `S[start:end]`, but the index returned is an index into the whole *string.*

Figure 1: The three `find` calls on the string `hello hello`, drawn against the indexed string. Searching for `hello` with no window matches at 0; passing `start=1` skips past that match and finds the second copy at 6; passing `start=1, end=3` restricts the search to `S[1:3]`, which is the two characters `el` and so cannot contain a five-character needle — hence `-1`. The returned index is still an index into the whole string, not into the window.

Exercise: title case

> Convert a string of text into *title format*. For example, `'a tale of two cities'` should be converted to `'A Tale Of Two Cities'`.

This can be accomplished by invoking the appropriate string method correctly:

```
>>> cs = "a tale of two cities"
>>> cs.title()
'A Tale Of Two Cities'

>>> "a tale of two cities".title()
'A Tale Of Two Cities'
```

Both forms work: the receiver can be a variable or a literal, since the literal *is* the object.

Exercise: centering text

> Write a function that, given a word and an integer width, **centers** the word in a sequence of `x`s.

The expected behaviour, for a function `foo`:

```
>>> foo('spam', 10)
'xxxspamxxx'

>>> foo('101', 20)
'xxxxxxxx101xxxxxxxxx'

>>> foo('UQQU', 30)
'xxxxxxxxxxxxxUQQUxxxxxxxxxxxxx'
```

A string method already does this. Its help:

```
>>> help(str.center)
Help on method_descriptor:

center(self, width, fillchar=' ', /)
    Return a centered string of length width.

    Padding is done using the specified fill character
    (default is a space).
(END)

>>> 'spam'.center(10, 'x')
'xxxspamxxx'
>>> '101'.center(20, 'x')
'xxxxxxxx101xxxxxxxxx'
```

Compare `fillchar=' '` here with the bracket notation on the `find` help: **both** mean "optional", but this signature also tells you the default value. The `/` marks the end of the positional-only parameters.

`foo('UQQU', 30)` has no such problem: $30 - 4 = 26$ splits evenly into 13 and 13.

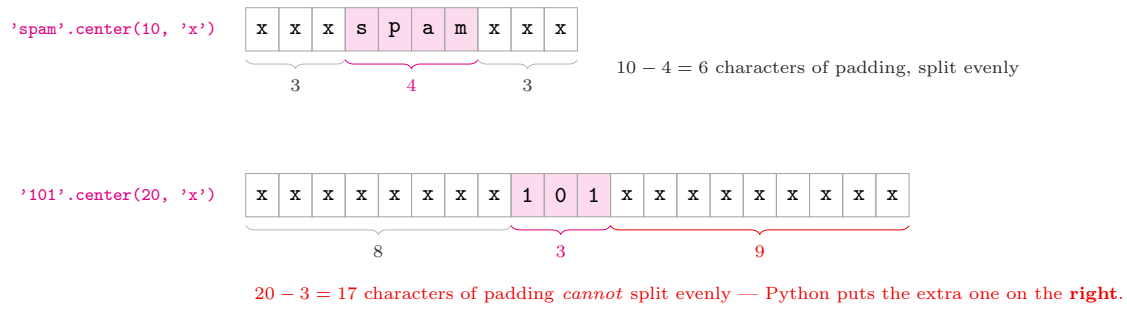


Figure 2: Why `'101'.center(20, 'x')` is not symmetric. `center` fills the leftover width with `fillchar`; when that leftover is odd it cannot split evenly, and the output here has 8 padding characters on the left and 9 on the right, so in this call the extra one goes on the right.

Splitting strings

```
>>> "a b c".split()          # Default split at space.
['a', 'b', 'c']
>>> "axbxc".split()
['axbxc']
>>> "axbxc".split('x')
['a', 'b', 'c']
>>> "axbxc".split('bx')
['ax', 'c']
>>> "a, b, c".split(',')     # Useful for CSV processing
['a', ' b', ' c']
>>> "a, b, c".split(', ')    # Removing tailing/leading spaces
['a', 'b', 'c']
```

What the transcript shows:

- With **no argument**, `split` splits on whitespace: any run of whitespace serves, not only a single space. That is why `"axbxc".split()` finds nothing to split on and returns a one-element list.
- The separator is an arbitrary **string**, not a single character (`'bx'`).
- The separator is **consumed**. Splitting `"a, b, c"` on `','` leaves the spaces attached to the following fields; splitting on `', '` eats them too. See python-string-methods[5] for a drawing of exactly which characters each call removes.

That last point is the one that matters for 2025-09-01-file-io[6] — CSV rows read from a file

[5] courses/csse1001/python-string-methods.pdf
[6] courses/csse1001/2025-09-01-file-io.pdf

arrive with whatever spacing the file happens to use.

Joining strings

```
>>> ",".join(["A", "B", "C"])
'A,B,C'
>>> "".join(["A", "B", "C"])
'ABC'
>>> ", ".join(["A", "B", "C"])
'A, B, C'
>>> "xxx".join(["A", "B", "C"])
'AxxxBxxxC'
```

The separator is the string the method is called *on*, and the list is the argument — the opposite way round from how it reads in English. The separator goes **between** elements only, never at the ends.

Sanitizing data

```
>>> help(str.strip)
strip(self, chars=None, ) unbound builtins.str method
    Return a copy of the string with leading and trailing
    whitespace removed.

    If chars is given and not None, remove characters in
    chars instead.

>>> "   123   \n".strip()   # A newline is considered whitespace
'123'
```

`strip` removing `\n` is exactly what makes it useful in the next lecture: every line a file hands you (except possibly the last) ends in a newline.

Summary

Sometimes datatypes come with extra functionality by way of **methods**. In particular, there are many **string methods** that will aid with text processing.

Next: 2025-09-01-file-io[7] (Lecture 6B).

[7] courses/csse1001/2025-09-01-file-io.pdf