CSSE1001 — Week 6 Notes

Table of contents

File IO **1**

Outline . . . 1
Memory hierarchy . . . 1
Why files? . . . 2
Opening a file . . . 2
Reading a file . . . 3
The file-pointer is exhausted after one pass . . . 4
Closing files . . . 5
Exercise: counting empty lines . . . 5
Reading numbers from a file . . . 7
Exercise: the highest-rated band . . . 8
Exercise: reading a Sudoku board . . . 9
Writing to files . . . 11
Appending to files . . . 11
Summary . . . 12

String Methods **12**

Outline . . . 12
Why learn string methods? . . . 12
Built-in string methods . . . 13
Methods . . . 13
Interpreting `help` . . . 14
Exercise: title case . . . 15
Exercise: centering text . . . 15
Splitting strings . . . 16
Joining strings . . . 17
Sanitizing data . . . 18
Summary . . . 18

Testing **19**
Outline . . . 19
Docstring testing . . . 19
Catching mistakes . . . 20
Writing good doctests . . . 21
Whitespace pitfalls . . . 21
String testing vs. equality testing . . . 22
Black-box testing . . . 22
Testing sets . . . 23
Testing unordered types . . . 23
Multi-line docstrings . . . 23
Testing outside the module: `doctest.testfile` . . . 24
Assertions . . . 25
Practice exercises . . . 26
Summary . . . 27

Reference material **27**
Python Comprehensions . . . 27
Python File IO . . . 30
Python String Methods . . . 33
Python Testing (doctest & Assertions) . . . 35

File IO

Lecture 6B. See csse1001[1] for course logistics and python-file-io[2] for the full reference on file modes and reading/writing patterns.

Outline

- Memory hierarchy and why files
- Opening, reading, and closing files
- The file-pointer, and why it can only be spent once
- Reading numbers out of a file, and casting
- Exercises: counting empty lines, finding the highest-rated band, reading a Sudoku board
- Writing and appending to files

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-file-io.pdf

Memory hierarchy

Five kinds of memory, with the capacity and the persistence of each:

Type	Order	2016 MacBook Pro	Persistence
CPU Cache L2	KB	256KB	Requires power
CPU Cache L3	MB	8MB	Requires power
Random Access Memory	GB	16GB	Requires power
Disk	GB/TB	256GB	Persistent
Cloud	PB	Functionally infinite	Persistent

As you go down the list you get **more space** and, past Random Access Memory, you also get **permanence**.

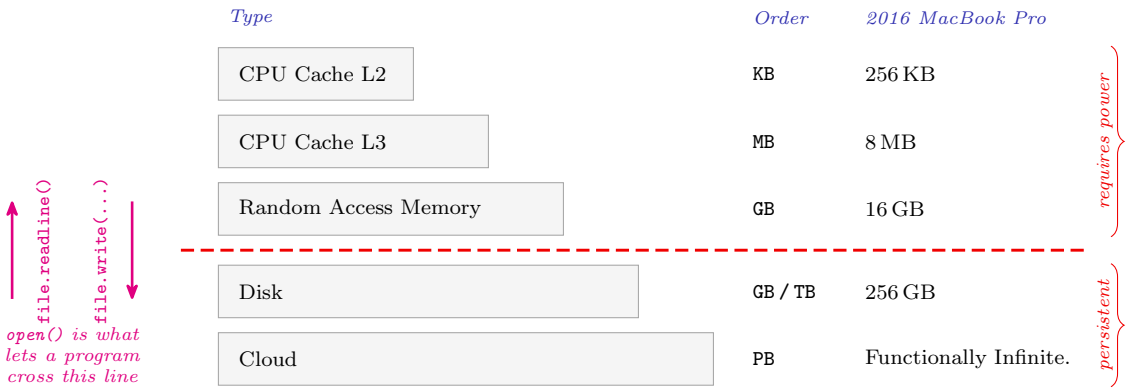


Figure 1: The memory hierarchy and its one boundary: everything above the dashed line is gone when the power goes, everything below it survives. That boundary is why files are needed. A Python program lives in RAM, and `open` is the one thing that lets it reach across. The stepped bar widths stand for the growing Order column, and the two arrows are a write and a read crossing the line.

Why files?

When Python (i.e. IDLE) launches it gets **allocated space in RAM**, which is possible to fill up. This is a problem if we want to solve memory-intensive problems — analyzing tweets, payroll, scientific computing. We may also want to *save* our data with **persistence**.

In either case, it is necessary to instruct Python to use the disk memory one level up.

Opening a file

To open a file assumed to be in the **same directory** as your running Python:

```
file = open("file.dat", "<mode>")
```

Mode	Description
`r`	read
`w`	write
`a`	append

Append means write at the end of the file.

`os.getcwd()` and `os.chdir()` are what you reach for when the file is **not** in the same directory as the running program.

Reading a file

The example file, `hello.txt`, is three lines:

```
What a¶
wonderful¶
hello world.
```

where ¶ is the **new-line character**, which normally is not displayed. The third line carries none.

```
>>> file = open("hello.txt", "r")     # open for read.
>>> file.readline()
'What a\n'
>>> file.readline()
'wonderful\n'
>>> file.readline()
'hello world.'      # note lack of newline character
>>> file.readline()
''
>>> file.readline()
''
```

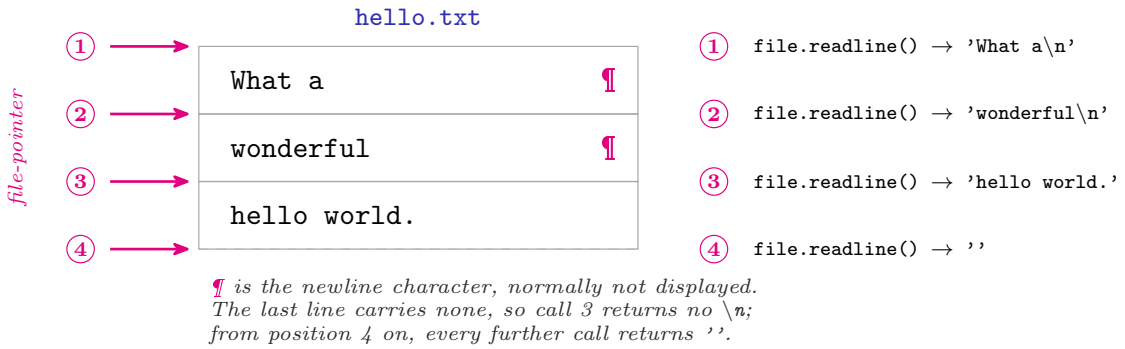


Figure 2: How readline moves the file-pointer. Each call to readline returns everything from the file-pointer up to and including the next newline, and then leaves the pointer past it. So the five calls in the transcript are not five readings of the same file, but reads taken from four successive pointer positions, the fifth call finding the pointer exactly where the fourth left it. Once the pointer reaches position 4 there is nothing left and every further call returns the empty string, which is why only four positions are drawn. This is why, to read a line twice, the file must be opened twice. The pilcrow stands for the newline character.

Past the end of the file `readline` returns `''` every time. That is the empty string, not a newline, and it is what makes `readline` a usable loop-termination test.

A for-loop iterates line by line:

```
>>> file = open("hello.txt", "r")     # open for read.
>>> for line in file:
...     print(line)
What a
¶
wonderful
¶
hello world.
¶
>>>
```

The three ¶ in that output are inserted by Python's `print`: every `print` call ends with a newline of its own. For the first two lines that added newline lands on top of the `\n` the line already carries, so the terminal shows a blank line between outputs. The third line carries no `\n`, so its ¶ is the only newline printed and nothing blank follows it.

The file-pointer is exhausted after one pass

Run the same loop again on the same `file`:

```
>>> for line in file:
...     print(line)
>>>
```

Nothing prints. The **file-pointer** `file` has already reached the end of the file during the last loop. To reset the file pointer to the beginning, open the file again:

```
>>> file = open("hello.txt", "r")
>>> for line in file:
...     line
'What a\n'
'wonderful\n'
'hello world.'
```

This loop uses a bare `line` rather than `print(line)`, which is why the newline characters are visible here and were invisible in the previous transcript.

Closing files

If you do not close your files they remain open and vulnerable to side-effects. That is, you may find some of your file is missing, or extra bits in your file, when you neglect to close after use.

```
>>> file = open("hello.txt", "r")
>>> for line in file:
...     line
'What a\n'
'wonderful\n'
'hello world.'
>>> file.close()
```

Alternative to close

Using the `with` construct has the advantage of **closing your file for you** — even if the program crashes while it is executing its code block.

```
with open("hello.txt", "r") as file:
    for line in file:
        print(line)
```

Exercise: counting empty lines

Write a function `def num_empty_lines(file: str) -> int:` that counts the number of **empty lines** (those that *only* contain `\n`) in a file.

```
>>> def num_empty_lines(path: str) -> int:
...     """
...     Count 'empty' lines.
...     """
...     ans = 0
...     with open(path, 'r') as file:
...         for line in file:
...             if line == '\n':
...                 ans = ans + 1
...     return ans
>>> num_empty_lines("filename.txt")
```

The answer calls the parameter `path` rather than `file`. `path` is the better name: the argument is a *file name*, not an open file, and the next exercise turns on exactly that distinction.

The equivalent written with the compound assignment operator, and with the file variable named so it cannot be confused with the parameter:

```
def num_empty_lines(path: str) -> int:
    ans = 0
    with open(path, "r") as the_file:
        for line in the_file:
            if line == "\n":
                ans += 1
    return ans
```

The same function over a file-pointer

Write a function that counts empty lines, but takes an already-open file:

```
from typing import TextIO

def num_empty_lines(file_pointer: TextIO) -> int:
```

This is different from the last one, which took a **name** and not a **pointer** (to a file).

`TextIO` is a new doc-hint type, imported from `typing`.

```
from typing import TextIO

def num_empty_lines(file_pointer: TextIO) -> int:
    """
    Count 'empty' lines.
    """
    ans = 0
    for line in file_pointer:
        if line == '\n':
            ans = ans + 1
    return ans

>>> fp = open("filename.txt")
>>> num_empty_lines(fp)
```

Two consequences of the change, both visible in the code:

- There is no `open` and no `with` inside the function any more — **the caller** now owns opening (and closing) the file.
- The function is handed a pointer that may already have been partly consumed. It counts from wherever the pointer happens to be, not from the start of the file.

Reading numbers from a file

Given `numbers.dat`, one number per line (the file ends without a newline after the `6`):

```
1¶
2¶
3¶
4¶
5¶
6
```

When reading from a file we are **always reading strings**. Do not forget to **cast** your strings to more appropriate types when necessary:

```
>>> with open("numbers.dat", "r") as file:
...     ans = []
...     for line in file:
...         ans.append(int(line))
>>> ans
[1, 2, 3, 4, 5, 6]
```

`int("1\n")` works because `int` tolerates surrounding whitespace. That is the reason no `strip()` is needed here (see python-string-methods[3]).

Exercise: the highest-rated band

Write a function that finds the highest-rated band in `bands.txt`:

```
Band,Rating,Plays          <- this is called a header
Metallica,7,512
The Black Keys,8,193
Amy MacDonald,5,290
The Killers,9,402
U2,9,789
Portishead,6,116
Wiggles,2,1849
Coldplay,7,0
Taylor Swift,11,1234567
Drake,0,1
```

Each line of the body goes through the same short pipeline:

`line` → `.split(',')` → `band, rating, _` → `int(rating)` → compare against the running best

```
def highest_rated(path: str) -> str:
    """ Return the highest-rated band in the file at <path>.
    Precondition: the file has a header row.
    """
    current_most_popular_band = ""
    current_highest_rating = -float('inf')   # guarantees the first row passes the > test

    with open(path, "r") as file:
        file.readline()   # skip the header
```

[3] courses/csse1001/python-string-methods.pdf

```
        for line in file:
            band, rating, _ = line.split(',')   # don't give names to values you don't intend
            rating = int(rating)
            if rating > current_highest_rating:
                current_highest_rating = rating
                current_most_popular_band = band

    return current_most_popular_band
```

The three comments in that block mark the three ideas in it:

- `-float('inf')` as the starting rating **guarantees** the first row passes the `if rating > current_highest_rating:` test, so no special case is needed for "the first line".
- The bare `file.readline()` before the loop **skips the header** — it is called for its side effect on the file-pointer, and its return value is thrown away. This only works because the pointer moves (see Figure 10).
- `_` is the conventional name for a value you unpack but do not want: *don't give names to values you don't intend on using.*

The parameter `path` is a `str`, not a `TextIO`: the body passes it straight to `open(path, "r")`, exactly as in the earlier exercise.

The third column, `Plays`, is unpacked into `_` and thrown away. On this particular data the highest rating (11) and the highest play count (1234567) both belong to Taylor Swift, so a version that ranked by the wrong column would still return the right answer here.

Exercise: an arbitrary attribute

Extend the previous answer to take an arbitrary file **with a header of attributes** like

```
name,grade,age
```

and write a function

```
def most(path: str, attribute: str) -> str:
```

that finds the maximum of that attribute.

The header is what makes it possible: `attribute` has to be looked up in the header row to find *which* column to compare.

Exercise: reading a Sudoku board

> Write a function `def read_board(fname: str) -> list[list[int]]:` that, given a file containing a partially filled Sudoku board, returns a list of lists of integers with `None` in unfilled spots.

In `board.txt`, `|` separates the 3x3 blocks column-wise, a row of `-` separates them row-wise, and a space marks an empty cell:

```
685|13 | 47
7  |   | 1
 1 |764| 5
-----------
9  | 7 |5 4
8 1|  9| 72
4 3|  6|
-----------
   |427|39
 4 |9  | 68
1 7|   |4
```

A return type of `list[list[int]]` has no room for the `None`s the statement asks for, so the stub is annotated `list[list[int | None]]`:

```
def read_board(path: str) -> list[list[int | None]]:
    """ Return a board representation for the Sudoku board in the
    file at <path>. None denotes an empty position.
    """
```

Character-by-character:

```
def read_board_v1(path: str) -> list[list[int | None]]:
    with open(path, "r") as the_file:
        board = []
        for line in the_file:
            row = []
            if "-" in line:
                continue   # skip horizontal dividers
            for x in line:
                if x.isdigit():
                    row.append(int(x))
```

```
                if x == " ":
                    row.append(None)
            board.append(row)
    return board
```

The same thing with a list comprehension. Where `v1` simply declines to append on `|` and on the trailing newline, a comprehension has to *filter* both out explicitly, and each omission costs a different amount: drop the `if x != '|'` and every row comes back with eleven entries instead of nine, drop the `rstrip("\n")` and every row but the file's last comes back with ten.

```
def read_board_v2(path: str) -> list[list[int | None]]:
    with open(path, "r") as the_file:
        board = []
        for line in the_file:
            if "-" in line:
                continue   # skip horizontal divider
            row = [int(x) if x.isdigit() else None
                   for x in line.rstrip("\n") if x != '|']
            board.append(row)
    return board
```

See python-comprehensions[4] for the comprehension form.

Exercise: has the Sudoku been won?

```
def has_won(board: list[list[int]]) -> bool:
    """ Return True when the Sudoku board is solved (contains the
    digits 1 through 9 in each row, column, and 3x3 grid).
    """
```

Writing to files

```
>>> with open("numbers.dat", "w") as file:
...     file.write("Hello World.\n")             # Write single string.
...     file.writelines(["Hello\n", "World\n"])  # Write a list of strings.
```

Both calls write exactly the characters given. The newlines in the example are part of the strings themselves, not something the call adds.

Careful! Opening a file for write will create the `numbers.dat` file, or **overwrite the old one** if it exists.

[4] courses/csse1001/python-comprehensions.pdf

Appending to files

Appending will open a file **without over-writing**, instead **appending** to the end of the file. A file is created if one does not exist.

```
>>> with open("numbers.dat", "a") as file:
...     file.write("Hello World.\n")
```

See python-file-io[5] for a side-by-side picture of what `r`, `w` and `a` each do to an existing file.

Summary

We can **read** strings **from** files and **write** strings **to** files. A **file-pointer** is moved every time a line from the file is read. Therefore, if we want to read a line twice the file must be opened twice.

Next: 2025-09-01-testing[6] (Lecture 6C).

String Methods

Lecture 6A. Course logistics are in csse1001[7]. The full reference on these string methods is python-string-methods[8].

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`

[5] courses/csse1001/python-file-io.pdf
[6] courses/csse1001/2025-09-01-testing.pdf
[7] courses/csse1001/csse1001.pdf
[8] 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[9]) 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[10]). 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:

[9] courses/csse1001/2025-09-01-file-io.pdf
[10] 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.

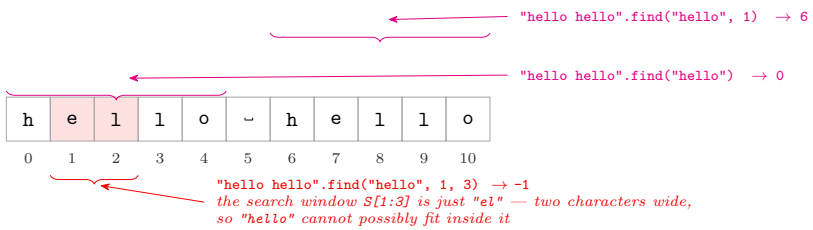
The comprehensive ways of calling `find` are therefore:

```
"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 3: 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.

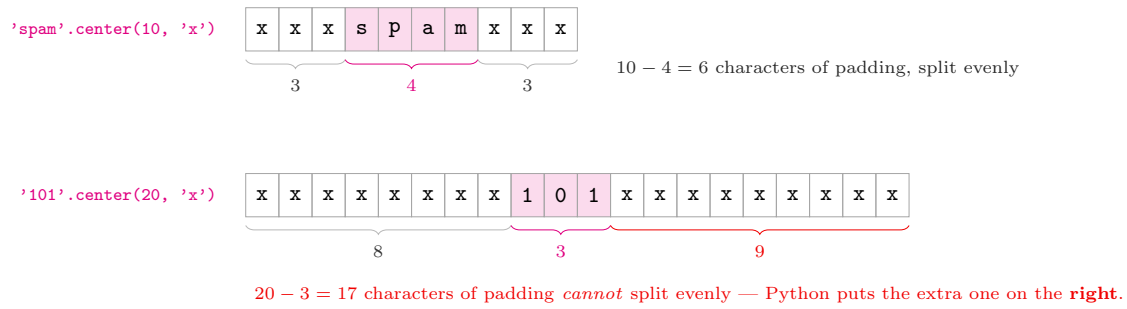


Figure 4: Why `'101'.center(20, 'x')` is not symmetric. `center` fills the leftover width with `fillchar`; when that leftover is odd it cannot split evenly, and the output here has 8 padding characters on the left and 9 on the right, so in this call the extra one goes on the right.

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[11] for a drawing of exactly which characters each call removes.

That last point is the one that matters for 2025-09-01-file-io[12] — CSV rows read from a file

[11] courses/csse1001/python-string-methods.pdf
[12] 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[13] (Lecture 6B).

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

Testing

See csse1001[14] for course logistics. This is Lecture 6C, which concludes the module on imperative programming. It follows 2025-09-01-string-methods[15] (6A) and 2025-09-01-file-io[16] (6B). See python-testing[17] for the full reference on `doctest` and assertions.

Outline

- Docstring (value) testing with `doctest.testmod()`
- Writing good doctests, and common whitespace/equality pitfalls
- Testing unordered types, and multi-line docstrings
- Black-box testing
- `doctest.testfile()`
- Assertions
- Practice exercises

Docstring testing

The docstrings so far have included doctests, e.g.:

```
def factorial(k: int) -> int:
    """Returns k! where k! = k*(k-1)! and 0! = 1.
    Assumes k > 0
    >>> factorial(3)
    6
    >>> factorial(0)
    1
    """
```

`doctest.testmod()` actually *runs* these tests, rather than just documenting intended usage.

[14] courses/csse1001/csse1001.pdf
[15] courses/csse1001/2025-09-01-string-methods.pdf
[16] courses/csse1001/2025-09-01-file-io.pdf
[17] courses/csse1001/python-testing.pdf

Catching mistakes

```
def factorial(k: int) -> int:
    """
    >>> factorial(3)
    6
    >>> factorial(0)
    1
    """
    ans = 1
    for ell in range(k):
        ans *= ell     # bug: multiplies by ell, not k - ell
    return ans


>>> import doctest
>>> doctest.testmod(verbose=True)
**********************************************************************
File "__main__", line 5, in __main__.factorial
Failed example:
    factorial(3)
Expected:
    6
Got:
    0
**********************************************************************
1 items had failures:
   1 of 2 in __main__.factorial
***Test Failed*** 1 failures.
TestResults(failed=1, attempted=2)
```

The corrected version:

```
def factorial(k: int) -> int:
    ans = 1
    for ell in range(k):
        ans *= k - ell
    return ans

>>> doctest.testmod()
TestResults(failed=0, attempted=2)
```

Writing good doctests

A comprehensive doctest suite should:

1. Test typical cases *and* edge cases.
2. Test the **zero** of the data type — e.g. `0`, `[]`, `""`.
3. Test the **singleton** of the data type — e.g. `1`, `[1]`, `"a"`.
4. Test for correctness, not violations of the function's contract (precondition).
5. Avoid redundant tests.

Whitespace pitfalls

Doctest compares **printed output** exactly, character for character — not equality of values. Both of the following would fail:

```
def identity(x):
    """
    >>> identity([])
    [ ]
    >>> identity([1, 2, 3])
    [1, 2, 3]
    """
```

(an extra space inside `[ ]` doesn't match Python's actual `[]` output)

```
def identity(x):
    """
    >>> identity([])
    []
    >>> identity([1, 2, 3])
    [1,2,3]
    """
```

(the empty-list line matches this time, but `[1,2,3]` doesn't match Python's own printed form, `[1, 2, 3]` — Python always prints a space after each comma in a collection literal)

String testing vs. equality testing

```
>>> identity(1.0)
1
```

fails because doctest compares the printed *string* `1.0` against the expected string `1` — they don't match, even though `1.0 == 1` is `True` as *values*. Expected output must match exactly what Python would print.

Black-box testing

Suppose we're given a function whose code is hidden. Confidence in its correctness then has to come through testing alone.

```
def pow(x: int, y: int) -> float:
    """ Returns x**y. Precondition: y >= 0. """
    return x*pow(x, y-1) if y else 1
```

```
def pow(x: int, y: int) -> int:
    """
    >>> pow(0, 0)      # zero
    1
    >>> pow(1, 0)      # unit and zero
    1
    >>> pow(0, 1)      # zero and unit
    0
    >>> pow(3, 1)      # typical and unit
    3
    >>> pow(1, 3)      # unit and typical
    1
    >>> pow(6, 10)     # typical
    60466176
    """
```

Exercise: `ourmax`

```
def ourmax(x: int, y: int) -> int:
    """ Return the larger of x and y. """
```

This is left as an open exercise: write the doctests, then implement.

Testing sets

A set does not print in the order it was written. Below, the small-integer sets happen to come out sorted and the string sets do not — neither is a language guarantee, since the printed order simply follows the hash table's layout. That is what makes a set's printed form a poor thing to write a doctest against:

```
>>> {3, 2, 1}
{1, 2, 3}
>>> {2, 1, 3}
{1, 2, 3}
>>> {"a", "b", "c"}
{'c', 'b', 'a'}
>>> {"b", "c", "a"}
{'c', 'b', 'a'}
```

Testing unordered types

Since string-testing an unordered type's printed representation is unreliable, compare against a literal value with `==` instead:

```
def identity(x):
    """
    >>> {3, 1, 2} == identity({1, 2, 3})
    True
    >>> {1: "A", 2: "B"} == identity({1: "A", 2: "B"})
    True
    """
```

Multi-line docstrings

Setting up intermediate values across multiple `>>>` lines within one doctest is allowed:

```
def identity(x: int) -> int:
    """
    >>> a = 2
    >>> b = 1
    >>> identity(a + b)
    3
    """
```

Exercise: `poly_min`

```
def poly_min(a: int, b: int, c: int) -> float:
    """ Return the (approximate) minimum value of
    f(x) = a*x**2 + b*x + c
    for x any float.
    """
```

Float testing is complicated by the fact that float arithmetic is inexact — we usually only insist on answers being *close enough*, rather than equal, using a tolerance:

```
def poly_min(a: int, b: int, c: int) -> float:
    """
    >>> tolerance = 10**-3
    >>> abs(poly_min(1, 0, 0) - 0) < tolerance
    True
    >>> abs(poly_min(3, -5, 10) - 7.916666666666666) < tolerance
    True
    """
```

The implementation is left as an open exercise; the doctests show the tolerance-based comparison technique.

Testing outside the module: `doctest.testfile`

Docstring examples inside a function aren't meant to fully test a module — they explain usage to users. A full test suite belongs *outside* the functions, in its own file:

```
# testing.txt
sandbox.py should be in the same directory as this file
and contain fact. This entire file will be treated as
a docstring. For instance, this paragraph is considered
a comment despite not having quotes around it.
>>> from sandbox import fact
>>> fact(3)
6
>>> fact(0)
1
```

```
>>> doctest.testfile("testing.txt", verbose=True)
...
1 items passed all tests:
   3 tests in testing.txt
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
TestResults(failed=0, attempted=3)
```

Assertions

An **assertion** is a truth claim that Python enforces at runtime. Programming with assertions helps catch problems early, by preventing (what are supposed to be) impossible situations from silently propagating — a failed assertion raises an `AssertionError` and stops the program.

```
def fact(x: int) -> int:
    ans = 1
    for k in range(x):
        ans *= k
    assert ans > 0   # all factorials are positive/non-zero
    return ans
```

```
>>> fact(3)
AssertionError
```

(This deliberately reuses the earlier buggy pattern — multiplying by the loop variable itself, which starts at 0 — to demonstrate the assertion catching the bug.)

```
>>> fact(3)
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    fact(3)
  File "sandbox.py", line 7, in fact
    assert ans > 0
AssertionError
```

`assert False` can also mark a line that's assumed to be unreachable — e.g. after an exhaustive `if`/`else` that's supposed to cover every case:

```
def maximum(x: int, y: int) -> int:
    if x > y:
        return x
    else:
        return y

    assert False   # (supposed to be) unreachable
```

Practice exercises

The following all ask: write doctests for the given signature, then implement it.

```
def indices(cs: str, subcs: str) -> list[int]:
    """ Return the indices in cs at which non-overlapping copies of
    subcs start. subcs is non-empty.
    >>> indices("A Coool pool look", "oo")
    [3, 9, 14]
    """


def insert_after(xs: list[int], a: int, b: int) -> list[int]:
    """ Insert <a> after each occurrence of <b> in list <xs>. """


def increment_count(hash: dict[str, int], key: str) -> None:
    """ Increment the value associated with key in hash in-place.
    If key is not a key in hash, add key with value 1.
    """
    if key in hash:
        hash[key] += 1
    else:
        hash[key] = 1
    return None


def average_grade(grades: list[list[object]]) -> float:
    """ Return the average grade for all the students in grades,
    where the inner lists contain a student ID and a grade.
    >>> grades = [['998765', 70], ['111234', 90], ['444567', 83]]
    >>> average_grade(grades)
    81.0
    """
```

```
def choose_chars(xs: str, ys: str, mask: str) -> str:
    """ Return a string where index i is xs[i] if mask[i] is '0'
    and ys[i] if mask[i] is '1'.
    Precondition:
        1. xs, ys, and mask are all of the same length.
        2. mask consists only of characters '0' and '1'.
    """
```

`indices`, `insert_after`, `average_grade` and `choose_chars` are left as open exercises. `increment_count` is already implemented; only its doctests are left to write.

Summary

We can verify our docstring examples using `doctest`. Tests should have sufficient coverage and not be redundant. Testing cannot guarantee a function works in general — it gives *confidence* that it's working, and helps prevent coding mistakes.

This concludes the module on imperative programming.

Next: 2025-09-09-exceptions[18] and 2025-09-09-object-oriented-programming[19] (Lecture 7).

Reference material

Python Comprehensions

A **comprehension** is a concise way to build a list or dictionary directly from an iterable, without writing an explicit accumulator for-loop.

List comprehension

```
>>> [k**2 for k in range(4)]
[0, 1, 4, 9]
```

Equivalent to:

```
>>> acc = []
>>> for k in range(4):
...     acc.append(k**2)
```

[18] courses/csse1001/2025-09-09-exceptions.pdf
[19] courses/csse1001/2025-09-09-object-oriented-programming.pdf

`sum(... for ...)`, without the surrounding `[]`, computes a running total directly. The bracket-less form is a *generator expression*, and so totals without building the intermediate list.

```
>>> sum(k**2 for k in range(4))
14
```

Dictionary comprehension

Same shape, but the output expression is a `key: value` pair and the braces are `{}`:

```
>>> {x: f"value is {x}" for x in range(3)}
{0: 'value is 0', 1: 'value is 1', 2: 'value is 2'}
```

Useful for assigning default values across a set of keys:

```
>>> {x: 0 for x in "ABC"}
{'A': 0, 'B': 0, 'C': 0}
```

Duplicate keys collapse — iterating a string with repeated characters still yields one entry per distinct character:

```
>>> cs = "hello world"
>>> {x: cs.count(x) for x in cs}
{'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

Nested comprehensions

Multiple `for` clauses can appear in one comprehension. The order of the `for` clauses matters — it matches the order they'd be nested as for-loops:

```
>>> [(a, x) for a in "AB" for x in range(2)]
[('A', 0), ('A', 1), ('B', 0), ('B', 1)]

>>> [(a, x) for x in range(2) for a in "AB"]
[('A', 0), ('B', 0), ('A', 1), ('B', 1)]
```

Filtering

A trailing `if` filters which elements are included:

```
>>> [x for x in range(101) if not (x % 3) and not (x % 7)]
[0, 21, 42, 63, 84]
```

The general pattern

```
[x for x in xs if P(x)]
[x for x in xs for y in ys if P(x, y)]
[x for x in xs for y in ys for z in zs if P(x, y, z)]
...
```

where `P` is some predicate.

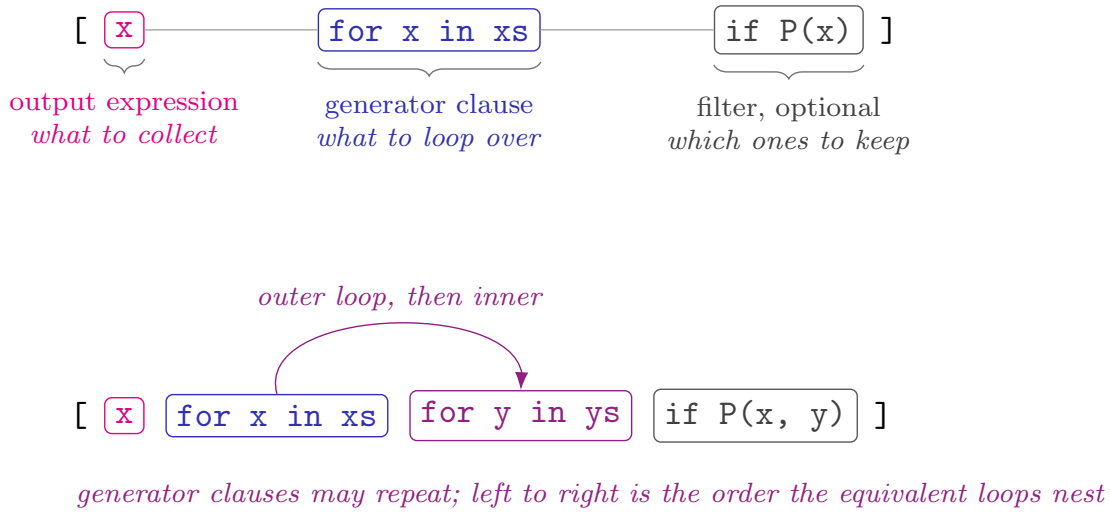


Figure 5: The three parts of a comprehension, named on the first two forms of the general pattern: output expression, generator clause and filter. Magenta is the output expression, blue the first generator clause, purple the second. That key covers the boxed code only: the purple arrow labelled *outer loop, then inner* and the purple line beneath the second form are annotations, not a fourth part.

The same three parts appear in a dictionary comprehension, except the output expression is a `key: value` pair.

`True/False` as `1/0`

`True/False` behave as `1/0` in arithmetic — a common idiom for counting how many elements of an iterable satisfy some condition. It works because Python's `bool` is a subtype of `int`.

```
>>> True + True + False
2
>>> sum(x > 0 for x in [-1, 2, 3, -4])
2
```

`all` and `any`

Python's built-in `all(xs)`/`any(xs)` check whether every/some element of `xs` is truthy:

```
>>> all(['A' <= x <= 'Z' for x in "HELLO WORLD"])
False
>>> any(['A' <= x <= 'Z' for x in "HELLO WORLD"])
True
```

> Careful: redefining a function named `all` or `any` in your own code shadows these built-ins for the rest of that scope. 2025-08-25-comprehensions[20]'s exercises do exactly that, deliberately, to reimplement them.

See python-for-loops[21] for the equivalent for-loop/accumulator forms these shortcut.

Python File IO

Opening a file

```
file = open("path", "<mode>")
```

Mode	Description
`r`	read
`w`	write (creates or **overwrites**)
`a`	append (creates or adds to the end)

A bare file name is resolved relative to the directory the running Python is in; `os.getcwd()` reports that directory and `os.chdir()` changes it.

[20] courses/csse1001/2025-08-25-comprehensions.pdf
[21] courses/csse1001/python-for-loops.pdf

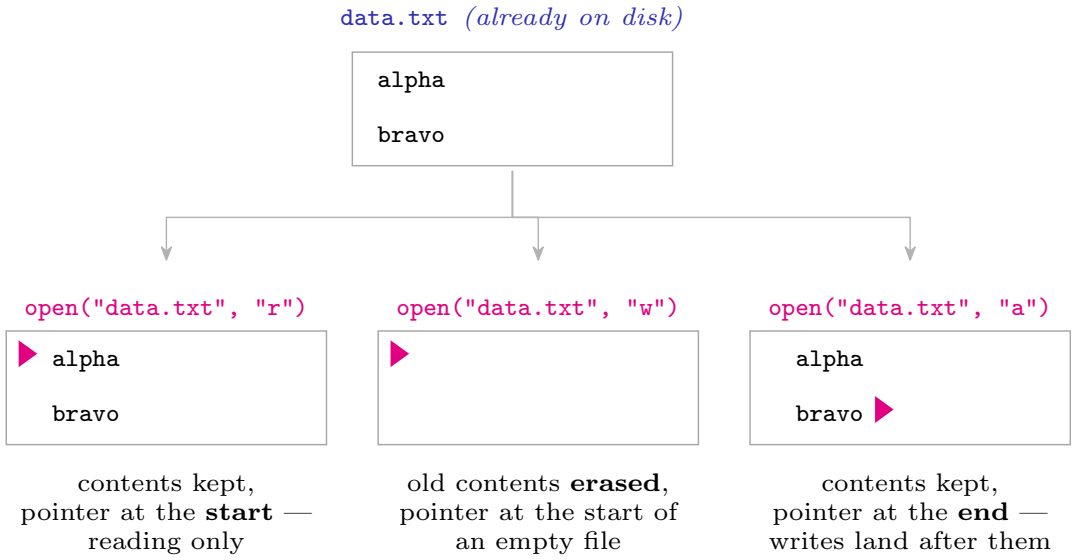


Figure 6: What each mode does to a file that already exists, and where it leaves the file-pointer. This is the difference that matters in practice: `w` destroys what was there before anything is written, while `a` keeps it and starts writing past the end.

Reading

`file.readline()` returns one line (including its trailing `\n`, except possibly the last line of the file), then `''` forever once exhausted. A for-loop iterates line by line and is the idiomatic way to consume a whole file:

```
with open("path", "r") as file:
    for line in file:
        ...
```

Reading always gives back **strings** — cast (`int(...)`, `float(...)`, …) as needed.

A file-pointer only moves forward: once a `for` loop (or repeated `readline()` calls) has consumed a file, iterating again yields nothing further unless the file is reopened or the pointer is reset with `file.seek(0)`. One consequence is useful: a lone `file.readline()` before the loop is how you skip a CSV header, because the call's only lasting effect is on the pointer.

Closing

`file.close()` releases the file; forgetting to close leaves it vulnerable to side effects (missing or extra data). `with open(...) as file:` closes it automatically — even if the block raises an exception — so it's preferred over manual `open`/`close`.

Passing files around: `str` vs `TextIO`

A function that does something to a file can take either the file's **name** or an already-open **file-pointer**, and the two are genuinely different interfaces:

```
from typing import TextIO

def f(path: str) -> ...:          # takes a name; opens (and closes) the file itself
    with open(path, "r") as file:
        ...

def g(file_pointer: TextIO) -> ...:   # takes an open file; the caller owns it
    for line in file_pointer:
        ...
```

`TextIO` is the type hint (imported from `typing`) for an open text file. A function of the second shape reads from wherever the pointer currently is, and leaves it wherever it stops — so its behaviour depends on what the caller has already read.

Writing and appending

```
with open("path", "w") as file:
    file.write("a single string\n")
    file.writelines(["one string\n", "per element\n"])
```

`"w"` overwrites any existing file; `"a"` instead appends to the end without touching existing content. Both write exactly the characters given — any newlines have to be part of the strings.

Python String Methods

Strings are objects with built-in **methods** — callable via `obj.method()` syntax, unlike free functions. The receiver goes before the dot, and the parentheses are required even with no arguments: `capitalize("hello")` is a `NameError`, `"hello".capitalize()` is `'Hello'`.

Review all of them with `help(str)`, or a specific one with `help(str.<name>)`. In the pager, Enter scrolls and `q` quits.

Reading a method's `help`

Square brackets in a signature indicate optional parameters, and the rule recurses:

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

means `sub` is required, `start` is optional, and `end` is optional *but only meaningful once* ***`start`*** *is given.* So `find("I")`, `find("I", 1)` and `find("I", 1, -1)` are the three legal shapes, and `find(1, -1)` is not one of them.

A signature written the other way, with a default value shown, means the same thing:

```
center(self, width, fillchar=' ', /)
```

`self` is the receiver (the string before the dot), `fillchar` is optional with default `' '`, and the trailing `/` marks the end of the positional-only parameters.

Common string methods

Method	Description
`s.find(sub[, start[, end]])`	Lowest index where `sub` occurs in `s[start:end]`, or `-1`
`s.title()`	Title-cased version of `s`
`s.center(width, fillchar=' ')`	Centre `s` in a string of length `width`
`s.split(sep=None)`	Split `s` on `sep` (default: any whitespace) into a list
`sep.join(xs)`	Join a list of strings `xs`, placing `sep` between each
`s.strip(chars=None)`	Remove leading/trailing whitespace (or `chars`) from `s`

```
>>> "team".find("I", 1, -1)
-1
>>> "a tale of two cities".title()
'A Tale Of Two Cities'
>>> "spam".center(10, "x")
'xxxspamxxx'
>>> "a, b, c".split(", ")
['a', 'b', 'c']
>>> "xxx".join(["A", "B", "C"])
'AxxxBxxxC'
>>> "  123  \n".strip()
'123'
```

Three behaviours worth naming, because they are the ones that bite when processing files:

- `find` returns `-1` on failure rather than raising, so its result must be tested before it is used as an index.
- `split` with **no** argument splits on whitespace; with an argument, the argument is an arbitrary **string**, not a set of characters (`"axbxc".split("bx")` is `['ax', 'c']`).
- `strip` counts `\n` as whitespace, which is what makes it the standard way to clean a line just read from a file.

`split` and `join` are inverses — via the separator

`center` does not always split the padding evenly

`center(width, fillchar)` pads `s` out to `width` characters. When `width - len(s)` is odd the padding cannot be halved, and the extra fill character goes on the **right**:

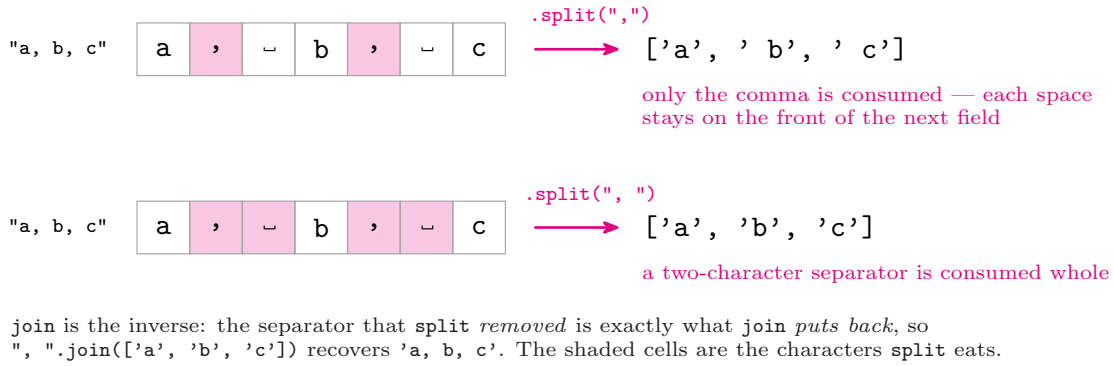


Figure 7: Which characters `split` actually removes, on a CSV-style example (splitting on a comma is useful for CSV processing). The separator is consumed, so splitting on a comma alone leaves each following space attached to the next field, and splitting on comma-space eats both, which removes the leading spaces. `join` is the exact inverse: it puts back between elements whatever `split` took out. The shaded cells are the characters `split` consumes.

```
>>> 'spam'.center(10, 'x')      # 10 - 4 = 6, split 3 / 3
'xxxspamxxx'
>>> '101'.center(20, 'x')       # 20 - 3 = 17, split 8 / 9
'xxxxxxxx101xxxxxxxxx'
```

See 2025-09-01-string-methods[22] for a drawing of both cases.

Python Testing (doctest & Assertions)

This is the reference note for testing in CSSE1001. 2025-09-01-testing[23] (Lecture 6C) develops it end to end. Two things it leans on come from 2025-09-01-string-methods[24] (Lecture 6A) and 2025-09-01-file-io[25] (Lecture 6B): the **REPL transcript**, which is the form a doctest is written in, and the **file pointer**, which is what makes file-reading functions awkward to test. Both are drawn below. Drawing conventions follow csse1001-diagram-notation[26].

[22] courses/csse1001/2025-09-01-string-methods.pdf
[23] courses/csse1001/2025-09-01-testing.pdf
[24] courses/csse1001/2025-09-01-string-methods.pdf
[25] courses/csse1001/2025-09-01-file-io.pdf
[26] courses/csse1001/csse1001-diagram-notation.pdf

`doctest`

A **docstring test** is an `>>>` example embedded in a function's docstring. `doctest.testmod()` actually *runs* every such example in the current module and reports failures:

```
>>> import doctest
>>> doctest.testmod()
TestResults(failed=0, attempted=2)
```

`doctest.testfile(path)` instead runs every `>>>` example found in an arbitrary text file (useful for a test suite kept *outside* the functions being tested).

Nothing runs a doctest by itself — a docstring full of examples is inert documentation until `testmod` or `testfile` is called on it.

A doctest is a REPL transcript

A REPL transcript is exactly the shape a doctest takes:

- `>>>` marks a line of **input**;
- `...` marks a **continuation** line, so an entire `def` can be typed this way;
- the unprompted line(s) that follow are the **expected output**;
- a blank line, or the end of the docstring, ends the example.

So a literal transcript can be pasted into a docstring as it stands and becomes a test. The exceptions are transcripts that are abbreviated rather than literal (the `strip` example below) or that mark newlines with a pilcrow (¶). This `title` example is already a complete doctest:

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

as are the `find` transcript below and the `split/join` transcripts in 2025-09-01-string-methods[27].

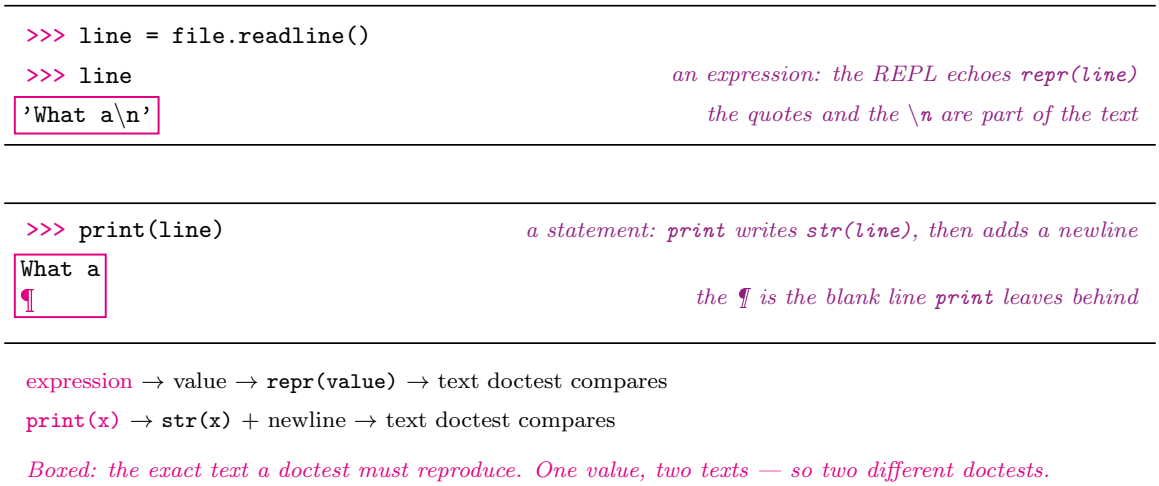


Figure 8: What text a doctest has to reproduce. Evaluating the expression `line` echoes its repr, `'What a\n'`, quote marks and newline escape included. `print(line)` writes the string itself and adds a newline, which leaves a blank line; the magenta pilcrow on a line of its own marks it. Each magenta box holds the exact text doctest matches, and the two arrow chains give the rule for each case.

What a doctest actually compares

`doctest` compares Python's *exact printed output* against the expected text — not whether two values are `==`. The rule underneath that is a rule about the REPL, and it has two halves.

The blank line `print` leaves behind cannot be written literally as expected output, because a blank line is how a doctest example *ends* (see the anatomy above). `doctest` spells it `<BLANKLINE>` instead. It is one more reason to prefer the echoing form.

Two practical consequences of the echo rule:

- **The quote marks are part of the expected text.** `"hello".capitalize()` is expected to give `'Hello'`, and the `.title()` example above `'A Tale Of Two Cities'` — with the quotes. A doctest expecting bare `Hello` fails.
- **A call that returns `None` echoes nothing at all.** `print` is such a call: no `None` appears after the printed lines. So an example that only calls `print` has no expected-output line for the return value.

More pitfalls, all instances of the same rule:

[27] courses/csse1001/2025-09-01-string-methods.pdf

- Whitespace matters — `[ ] != []`, and Python always prints `, ` (comma-space) inside collection literals.
- Unordered types (`set`, `dict`) don't have a guaranteed print order for non-numeric elements — compare with `==` inside the doctest instead of relying on printed order:

  ```
  >>> {3, 1, 2} == some_function({1, 2, 3})
  True
  ```

- Floats are inexact — compare with a tolerance instead of exact equality:

  ```
  >>> abs(f(x) - expected) < 10**-3
  True
  ```

Why `repr` is the friendlier half for testing

Because the echo goes through `repr`, whitespace that is invisible when printed is visible in a doctest. That is what makes `str.strip()` testable at all: `strip` returns a copy with leading and trailing whitespace removed, and a newline counts as whitespace. Abbreviated, the example is the literal `"  123  \n"` on the input line and `123` on the output line, which drops both the `.strip()` call and the quotes. Written out as a doctest it is:

```
>>> "   123   \n".strip()
'123'
```

and there the difference between `'  123  \n'` and `'123'` is plainly on the page. Printed, the two are nearly indistinguishable.

Writing a comprehensive doctest

1. Typical cases *and* edge cases.
2. The zero of the data type (`0`, `[]`, `""`).
3. The singleton of the data type (`1`, `[1]`, `"a"`).
4. Correctness, not contract violations (don't test precondition failures).
5. No redundant tests.

Worked example: centring a word

The exercise: given a word and an integer width, centre the word in a sequence of xs. The answer is `str.center`, whose help text is

```
center(self, width, fillchar=' ', /)
    Return a centered string of length width.
    Padding is done using the specified fill character
    (default is a space).
```

The calls are `'spam'.center(10, 'x')` and `'101'.center(20, 'x')`. Three transcripts of typical calls *are* three doctest examples already; the checklist above turns them into a suite.

```
def center_word(word:  str, width:  int) -> str:
    """ Centre word in a field of x's of the given width.
    >>> center_word('spam', 10)                                   typical
    'xxxspamxxx'
    >>> center_word('101', 20)          typical, odd padding: 8 on the left, 9 on the right
    'xxxxxxxx101xxxxxxxxx'
    >>> center_word('UQQU', 30)                  typical, even padding: 13 each side
    'xxxxxxxxxxxxxUQQUxxxxxxxxxxxxx'
    >>> center_word('', 4)                                        the zero of str
    'xxxx'
    >>> center_word('a', 1)                                     the singleton of str
    'a'
    >>> center_word('spam', 2)                  edge: no room to pad, so nothing is padded
    'spam'
    """
    return word.center(width, 'x')
```

The three examples beside the magenta bar come from the coverage checklist.
Their outputs were checked against CPython.

Figure 9: The centring exercise written as a doctest, with each example labelled with the job it does. The function is `center_word`, and its body is `str.center` with `'x'` as the fill character. The first three examples are typical cases. The three beside the magenta bar come from the coverage checklist above: the zero, the singleton and an edge case. The whole docstring passes testmod with six attempted, zero failed.

The three typical examples might look redundant against checklist rule 5. They are not: `'101'` in width 20 leaves an *odd* amount of padding to share out, while `'spam'` in 10 and `'UQQU'` in 30 leave *even* amounts, and `str.center` gives the leftover character to the right-hand side. That is two different paths through the method, not one tested three times.

> **An oracle for re-implementations**
>
> Implement any string method you use at least once. When you do, the built-in method is a ready-made oracle: a doctest can assert your version agrees with it, e.g. `>>> my_center('spam', 10) == 'spam'.center(10, 'x')` expecting `True`.

Optional parameters multiply the calling forms

The help text for `find` is

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

The square brackets mean *optional*, and the rule **recurses** — an optional parameter can itself carry optional parameters. So the legal calling forms are exactly

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

and `find(1, -1)` is illegal because `sub` is required. Each form is a different path into the function, so a comprehensive suite needs one example per form. This transcript covers three of them plus the failure return:

```
>>> "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 line is worth its place on its own: the help text says *return -1 on failure*, so `-1` is part of the contract, not an error condition, and a suite that never provokes it has not tested the documented behaviour.

Testing code that reads a file

The same function can be written with two different signatures, and the difference matters for testing.

```
def num_empty_lines(path: str) -> int:      # takes a file name
    ans = 0
    with open(path, 'r') as file:
        for line in file:
            if line == '\n':
                ans = ans + 1
    return ans

from typing import TextIO                   # takes a file pointer

def num_empty_lines(file_pointer: TextIO) -> int:
    ans = 0
    for line in file_pointer:
        if line == '\n':
            ans = ans + 1
    return ans
```

These are different functions: the first takes a file **name**, the second a **pointer** to a file. The testing consequence follows from how the pointer behaves.

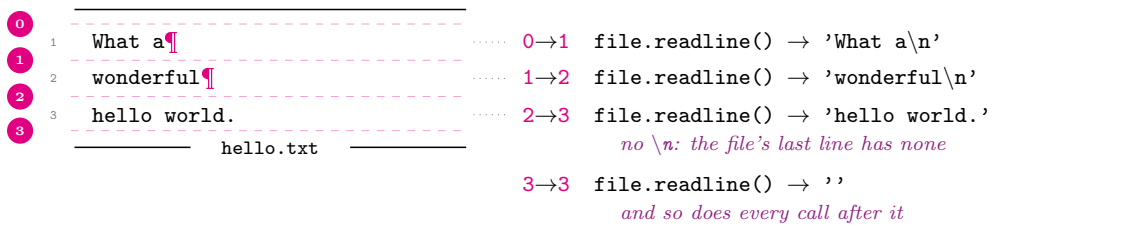


Figure 10: The file pointer walking hello.txt. A magenta pilcrow marks each newline in the file. There is no pilcrow after the last line, so that line comes back without a newline. The numbered circles are the pointer positions and the dashed lines are the line boundaries. Each `readline()` returns one line and moves the pointer past it. Once the pointer is at the end of the file, every further call returns the empty string.

The file-pointer version is not re-runnable. Doctest executes the examples in one docstring in order, in a single namespace, so a second call on the same handle reads an already-exhausted file:

```
>>> fp = open("notes.txt")     # a file with two empty lines
>>> num_empty_lines(fp)
2
>>> num_empty_lines(fp)
0
```

Both lines pass — and the second is a *lie* about the function. Either re-open the file for each example, or test the name-taking version, which opens the file itself and so starts from position **0** every time.

Either way, a file-reading doctest depends on the file system. A bare name is resolved relative to the directory Python is running in (see `os.getcwd()` and `os.chdir()`). So the test passes or fails depending on the working directory, and it needs a fixture file to exist. `doctest.testfile` is one way out: keep the suite in a text file that sits beside the fixture it reads, so the two travel together.

> ⚠ Never open a fixture in `"w"` mode from a test
>
> A `"w"` open on `numbers.dat` creates the `numbers.dat` file, or **overwrites the old one** if it exists. A doctest that opens its own fixture with `"w"` destroys the fixture on the first run and then passes forever on an empty file. Use `"r"` to read it and `"a"` (which appends, and creates the file only if it is absent) if a test genuinely has to add to one.

Also prefer `with` in anything a test calls. An unclosed file stays open and vulnerable to side effects: some of the file may go missing, or extra bits may appear in it. `with` closes the file even if the program crashes while it is executing its code block, which is exactly what a failing test does.

Type hints document; only tests and assertions check

The two file-reading signatures above are a good place to see this. `TextIO` is the type hint for a file pointer, as in `num_empty_lines(file_pointer: TextIO)`. Write that parameter as `file_pointer: str` and the function runs just the same. The highest-rated-band function in the next section can annotate its parameter as `path: TextIO` and then call `open(path, 'r')` on it, and that runs too, although the parameter is really a `str` path. Both annotations disagree with their own code and Python runs them anyway. Nothing in the language checks an annotation. In this material the only things that check anything at run time are the doctests you write and the assertions you place.

Assertions

`assert <condition>` raises `AssertionError` (halting the program) if `<condition>` is false — useful for catching impossible situations as soon as they occur, rather than letting them silently propagate:

```
assert ans > 0   # all factorials are positive/non-zero
```

`assert False` documents a line that should be unreachable (e.g. after an exhaustive `if`/`else`).

An assertion is the executable form of a claim you would otherwise only argue in a comment. A function that finds the highest-rated band in a file is built on exactly such a claim: it initialises

```
current_highest_rating = -float('inf')      # Guarantees line 11 satisfied
```

where line 11 is the comparison `if rating > current_highest_rating:`. The comment is a piece of reasoning about program state — *this starting value is below every rating the file can hold, so the first row always wins* — and nothing checks it.

The same function shows what an assertion buys you. `current_most_popular_band` starts as `""`, so on a `bands.txt` that has only its header row the loop never runs and the function silently returns the empty string. Adding

```
assert current_most_popular_band != ""      # the file held at least one band
```

before the `return` turns that silent wrong answer into an `AssertionError` at the point it occurs.

See python-exceptions[28] for `AssertionError` among Python's other exceptions, and python-file-io[29] and python-string-methods[30] for the machinery the file and string examples above use.

[28] courses/csse1001/python-exceptions.pdf
[29] courses/csse1001/python-file-io.pdf
[30] courses/csse1001/python-string-methods.pdf