Python Testing (doctest & Assertions)

Table of contents

`doctest` 1
A doctest is a REPL transcript 2
What a doctest actually compares 2
Writing a comprehensive doctest 4
Worked example: centring a word 4
Optional parameters multiply the calling forms 6
Testing code that reads a file 6
Type hints document; only tests and assertions check 8
Assertions 8

This is the reference note for testing in CSSE1001. 2025-09-01-testing[1] (Lecture 6C) develops it end to end. Two things it leans on come from 2025-09-01-string-methods[2] (Lecture 6A) and 2025-09-01-file-io[3] (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[4].

`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)
```

[1] courses/csse1001/2025-09-01-testing.pdf
[2] courses/csse1001/2025-09-01-string-methods.pdf
[3] courses/csse1001/2025-09-01-file-io.pdf
[4] courses/csse1001/csse1001-diagram-notation.pdf

`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[5].

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.

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

```
>>> line = file.readline()
>>> line                                   an expression: the REPL echoes repr(line)
'What a\n'                                 the quotes and the \n are part of the text
```

```
>>> print(line)                            a statement: print writes str(line), then adds a newline
What a
¶                                          the ¶ is the blank line print leaves behind
```

expression → value → `repr(value)` → text doctest compares

`print(x)` → `str(x)` + newline → text doctest compares

Boxed: the exact text a doctest must reproduce. One value, two texts — so two different doctests.

Figure 1: 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.

More pitfalls, all instances of the same rule:

- 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 `x`s. 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.

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.

> **i** 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`.

```
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 2: 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.

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.

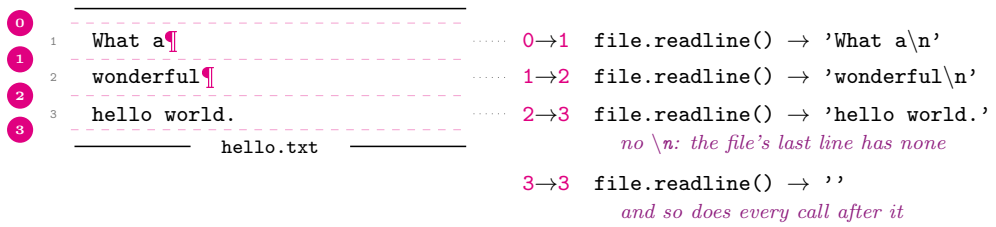

Figure 3: The file pointer walking hello.txt. A magenta pilcrow marks each newline in the file. There is no pilcrow after the last line, so that line comes back without a newline. The numbered circles are the pointer positions and the dashed lines are the line boundaries. Each `readline()` returns one 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[6] for `AssertionError` among Python's other exceptions, and python-file-io[7] and python-string-methods[8] for the machinery the file and string examples above use.

[6] courses/csse1001/python-exceptions.pdf
[7] courses/csse1001/python-file-io.pdf
[8] courses/csse1001/python-string-methods.pdf