Testing

Table of contents

Outline . . . 1
Docstring testing . . . 2
Catching mistakes . . . 2
Writing good doctests . . . 3
Whitespace pitfalls . . . 3
String testing vs. equality testing . . . 4
Black-box testing . . . 4
Testing sets . . . 5
Testing unordered types . . . 5
Multi-line docstrings . . . 6
Testing outside the module: `doctest.testfile` . . . 7
Assertions . . . 7
Practice exercises . . . 8
Summary . . . 9

See csse1001[1] for course logistics. This is Lecture 6C, which concludes the module on imperative programming. It follows 2025-09-01-string-methods[2] (6A) and 2025-09-01-file-io[3] (6B). See python-testing[4] 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

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

- `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.

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[5] and 2025-09-09-object-oriented-programming[6] (Lecture 7).

[5] courses/csse1001/2025-09-09-exceptions.pdf
[6] courses/csse1001/2025-09-09-object-oriented-programming.pdf