CSSE1001 — Week 6 Notes

Table of contents

File IO

See csse1001[1] for course logistics — this note covers Lecture 6B's technical content. See python-file-io[2] for the full reference on file modes and reading/writing patterns.

Today's outline

- Memory hierarchy and why files
- Opening, reading, and closing files
- Exercises: counting empty lines, reading numbers, finding the highest-rated band, reading a Sudoku board
- Writing and appending to files

Memory hierarchy

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

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

Why files?

When Python launches, it's allocated space in RAM, which can fill up — a problem for memory-intensive problems (analyzing tweets, payroll, scientific computing). We may also want to *save* our data with persistence. Either way, we need to instruct Python to use the disk, one level up the hierarchy.

Opening a file

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

Mode	Description
`r`	read
`w`	write
`a`	append (write at end of file)

See also `os.getcwd()` and `os.chdir()` (a file assumed to be in the same directory as the running script).

Reading a file

Given `hello.txt` containing:

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

```
>>> file = open("hello.txt", "r")
>>> file.readline()
'What a\n'
>>> file.readline()
'wonderful\n'
>>> file.readline()
'hello world.'      # note: no trailing newline (last line of the file)
>>> file.readline()
''                  # empty string once exhausted, returned indefinitely
```

A for-loop iterates line by line:

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

wonderful

hello world.

>>>
```

(The extra blank lines above come from `print`'s own newline stacking on top of each line's own trailing `\n`.)

The file-pointer is exhausted after one pass

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

Nothing prints — the file-pointer already reached the end of the file during the loop above. We need to reopen the file (or seek back to the start) to read it again:

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

Closing files

Files left open are vulnerable to side-effects — you may find data missing, or extra bytes, if you neglect to close after use:

```
>>> file = open("hello.txt", "r")
>>> for line in file:
...     line
>>> file.close()
```

The `with` construct closes the file for you automatically, even if the code block crashes:

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

Exercise: counting empty lines

```
def num_empty_lines(path: str) -> int:
    """ Count the number of empty lines (those that only contain \n)
    in the file at <path>.
    """
```

```
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
```

A second version takes a file *pointer* directly, rather than a path — note the different parameter type (the caller is now responsible for opening the file):

```
from typing import TextIO

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

```
            ans += 1
    return ans

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

Reading numbers from a file

Given `numbers.dat` containing one number per line (`1` through `6`), reading always gives back **strings** — cast to the appropriate type when needed:

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

Exercise: the highest-rated band

Given a CSV file `bands.txt` with a header row (`Band,Rating,Plays`), find the highest-rated band:

```
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 updates it

    with open(path, "r") as file:
        file.readline()   # skip the header
        for line in file:
            band, rating, _ = line.split(',')   # don't name values you won't use
            rating = int(rating)
            if rating > current_highest_rating:
                current_highest_rating = rating
                current_most_popular_band = band

    return current_most_popular_band
```

The lecture slide names this function `higest_rated` (missing a `t`) — corrected to `highest_rated` above. The companion source file has a differently-named `most_played` function with the *same shape* of code, but it unpacks the *third* CSV column (`Plays`) instead of the second (`Rating`) — so despite its docstring claiming to return "the highest rated band", it actually returns the band with the most plays. Reproduced here as `highest_rated` (ranking by rating, matching the slide and this section's title) rather than perpetuating that docstring/behaviour mismatch.

Exercise: an arbitrary attribute

Extend the previous answer to take an arbitrary file with a header of attributes (e.g. `name,grade,age`) and a function `def most(path: str, attribute: str) -> str` that finds the maximum of that attribute.

No worked solution is given in the source for this exercise — left as an open exercise rather than invented here. (The companion source file also includes a `least_rated_band` function that is an intentional joke stub — `return "Drake"` with the comment `# This is a joke` — rather than a real implementation.)

Exercise: reading a Sudoku board

Given a partially-filled Sudoku board as a text file (`|` separates 3x3 blocks column-wise, a row of `-` separates them row-wise, and spaces mark empty cells):

```
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
```

```
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 version:

```
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
```

Equivalent using a list comprehension (filtering out `|` characters, rather than simply not appending on them):

```
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 if x != '|']
            board.append(row)
    return board
```

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).
    """
```

No worked solution is given in the source for this exercise — left as an open exercise rather than invented here.

Writing to files

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

Careful! Opening a file for write (`"w"`) creates the file if it doesn't exist, or **overwrites** it if it does.

Appending to files

Appending opens a file without overwriting, instead adding to the end (creating the file if it doesn't exist):

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

Summary

We can read strings from files and write strings to files. A file-pointer moves forward every time a line is read — to read a line (or the whole file) twice, the file must be reopened.

Next: 2025-09-01-testing[3] (Lecture 6C).

String Methods

See csse1001[4] for course logistics — this note covers Lecture 6A's technical content. See python-string-methods[5] for the full reference on the string methods covered here.

Today's outline

- Built-in string methods and `help()`
- Method invocation syntax
- Exercises: `title`, `center`
- Splitting and joining strings
- Sanitizing data with `strip`

[3] courses/csse1001/2025-09-01-testing.pdf
[4] courses/csse1001/csse1001.pdf
[5] courses/csse1001/python-string-methods.pdf

Why learn string methods?

During file processing (next lecture) we'll be working extensively with strings. We're already able to replicate the functionality of any string method with our current tools — but it's worth striving to implement any string method you use at least once.

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, `q` to quit help.)

Methods

Strings are **objects** — we'll eventually learn what that means fully. Objects have **methods**, which are like functions but invoked differently. We don't say:

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

but rather:

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

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.
```

Interpreting `help`

Square brackets in a signature like `find(sub[, start[, end]])` indicate an *optional* parameter — and this rule recurses (an optional parameter can itself have optional parameters). The valid ways to call `find` are:

```
"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
```

Exercise: title case

Convert a string of text into title format:

```
def title_case(cs: str) -> str:
    """ Convert cs to title case.
    >>> title_case("a tale of two cities")
    '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'
```

Exercise: centering text

Write a function that, given a word and an integer width, centers the word in a sequence of xs:

```
def foo(word: str, width: int) -> str:
    """
    >>> foo('spam', 10)
    'xxxspamxxx'
    >>> foo('101', 20)
    'xxxxxxxx101xxxxxxxxx'
    >>> foo('UQQU', 30)
    'xxxxxxxxxxxxxUQQUxxxxxxxxxxxxx'
    """


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

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

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

Splitting strings

```
>>> "a b c".split()          # default: split at whitespace
['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(', ')    # removes leading/trailing spaces
['a', 'b', 'c']
```

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'
```

Sanitizing data

```
>>> help(str.strip)
strip(self, chars=None, /)
    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'
```

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[6] (Lecture 6B).

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

Testing

See csse1001[7] for course logistics — this note covers Lecture 6C's technical content, which concludes the module on imperative programming. See python-testing[8] for the full reference on `doctest` and assertions.

Today's 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

We've been diligently including doctests in our docstrings, 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
```

[7] courses/csse1001/csse1001.pdf
[8] courses/csse1001/python-testing.pdf

```
    >>> 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]
    """
```

(a trailing space is fine here, 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 — how do we gain confidence in its correctness 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. """
```

> No worked solution is given in the source for this exercise — left as an open exercise (write the doctests, then implement) rather than invented here.

Testing sets

Only sets containing numbers print in sorted order — string-keyed sets print in an implementation-defined order:

```
>>> {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
    """
```

> No worked implementation is given in the source for this exercise — only the doctests demonstrating 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 "/Users/pvrbik/Desktop/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'.
    """
```

No worked solutions are given in the source for `indices`, `insert_after`, `average_grade`, or `choose_chars` — left as open exercises rather than invented here. `increment_count`'s implementation *is* given in the source; only its doctests are left as the open exercise.

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: exceptions and an introduction to object-oriented programming.

Reference material

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)

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 seeked back to the start).

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`.

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.

Python String Methods

Strings are objects with built-in **methods** — callable via `obj.method()` syntax, unlike free functions. Review all of them with `help(str)`, or a specific one with `help(str.<name>)`.

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.*

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'
```

Python Testing (doctest & Assertions)

`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).

Doctest compares printed strings, not values

`doctest` compares Python's *exact printed output* against the expected text — not whether two values are `==`. `identity(1.0)` printing `1.0` will not match an expected `1`, even though `1.0 == 1`.

Consequences:

- 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
  ```

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.

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`).