Python software engineering

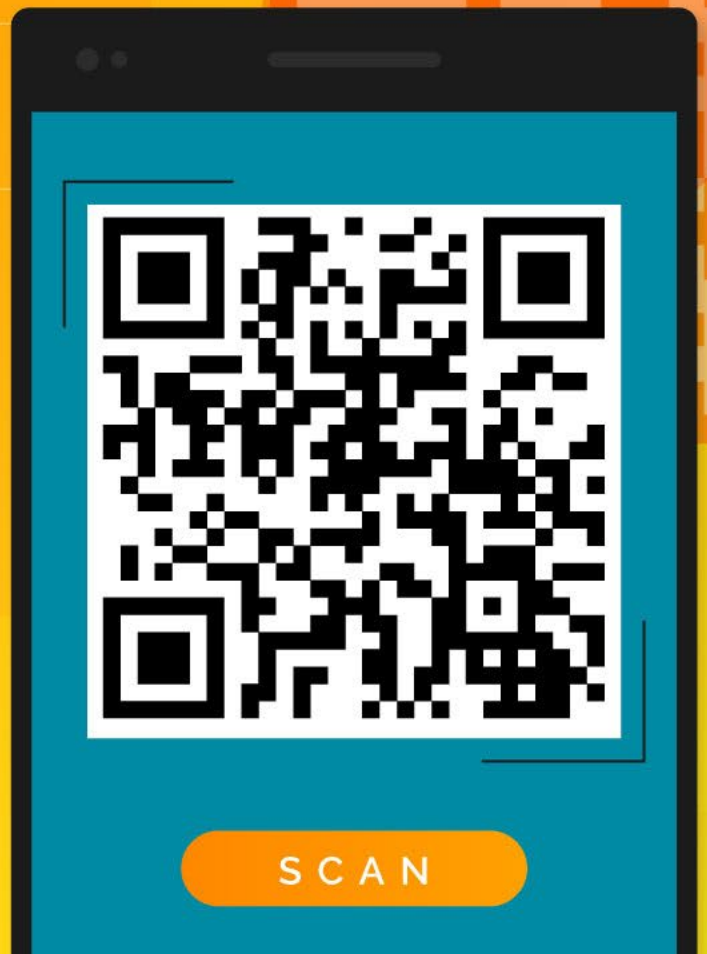
Geert Jan Bex

(geertjan.bex@uhasselt.be)

http://bit.ly/33Vx1T9

Stay Connected to VSC

LinkedIn®



Typographical conventions

- Inline code fragments and file names are rendered as, e.g., `hello_world.py`
- Longer code fragments are rendered as

```
#!/usr/bin/env python
…

if __name__ == '__main__':
    print('hello world!')
```

- Data files are rendered as

```
case dim temp
1 1 -0.5
2 1 0.0
3 1 0.5
4 2 -0.5
…
```

fragment not shown

BEST PRACTICES

Code style matters

- Readability
- Coding conventions
- Exception handling
- Code organization
- Documentation
- Testing

Code must be maintained
- Over long periods of time
- By multiple people

Coding == story telling

- Names should be descriptive
  - Variables are nouns
  - Functions are verbs
  - Boolean functions are questions
- Avoid long functions
  - Should fit on screen
- Don’t try to be too clever
  - Others should understand it (including your future self)
  - Add comments and documentation

Python is easy to read,
Take advantage of that!

Coding conventions

- Be consistent!
  - Many conventions, choose one, stick to it
- PEP 8
  - Use tools to check: flake8, pylint
  - Consistent formatting: black
  - IDE hooks

Use language idioms

- Don’t

```
for i in range(len(my_list)):
    print(my_list[i])
```

```
for i in range(len(my_list1)):
    print(my_list1[i] + my_list2[i])
```

- Do

```
for item in my_list:
    print(item)
```

```
for item1, item2 in zip(my_list1, my_list2):
    print(item1 + item2)
```

Further reading

- PEP 8
  https://www.python.org/dev/peps/pep-0008/
- Jef Knupp (2013) *Writing idiomatic Python 3*
- Mariano Anaya (2016) *Clean code in Python*, Packt>

https://github.com/gjbex/Python-software-engineering/tree/master/source-code/testing

ERRORS:
DEALING WITH EXCEPTIONS

Errors

```
…
def main():
    file_name = sys.argv[1]
    with open(file_name) as in_file:
        for line in in_file:
            print(f"|{line.rstrip('\r\n')}|")
    return 0
…
```

```
$ python quote.py
Traceback (most recent call last):
  File "./quote.0.py", line 13, in <module>
    status = main()
  File "./quote.py", line 6, in main
    file_name = sys.argv[1]
IndexError: list index out of range
```

exception thrown

Either check length of sys.argv, or deal with error!

Playing catch

```
…
def main():
    try:
        file_name = sys.argv[1]
    except IndexError as e:
        print('### error: no input file',
              file= sys.stderr)
        return 1
    with open(file_name) as in_file:
        for line in in_file:
            print(f"|{line.rstrip('\r\n')}|")
    return 0
…
```

```
$ python quote.py
### error: no input file
```

More trouble

```
$ python quote.py  bla
Traceback (most recent call last):
  File "./quote.py", line 17, in <module>
    status = main()
  File "./quote.py", line 11, in main
    with open(file_name) as in_file:
IOError: [Errno 2] No such file or directory: 'bla'
```

exception thrown

Catching more

```
…
def main():
    try:
        file_name = sys.argv[1]
        in_file = open(file_name)
        with in_file:
            for line in in_file:
                print('|{0}|'.format(line.rstrip('\r\n')))
    except IndexError as e:
        sys.stderr.write('### error: no input file\n')
        return 1
    except IOError as e:
        print(f"### I/O error on '{e.filename}': {e.strerror}“,
              file=sys.stderr)
        return 2
    return 0
…
```

All handled!

- Now all exceptions are handled

```
$ python quote.py  bla
### I/O error on 'bla': No such file or directory
```

- Note that code size increased from 5 to 16 lines
  - Handling errors takes effort
  - Worthwhile if others are using your software!
- One can create own exceptions, derive class from `Exception`

CODE ORGANIZATION

Python modules & packages

- Code organization
  - Functions common to multiple scripts can be put in separate file = module
  - Modules can be organized hierarchically in directory structure = packages

Don't forget `__init__.py` in package directories!

- Python standard library is organized in packages

Example module & use

- Module file:

data_parsing.py

```
from collections import namedtuple
Line_Data = namedtuple('Line_Data', ['case_nr', 'dim_nr', 'temp'])

def parse_line(line, sep=None):
    data = line.rstrip('\r\n').split(sep)
    return Line_Data(case_nr=int(data[0]),dim_nr=int(data[1]),
                     temp=float(data[2]))
```

- Using the module in script:

counting.py

```
import data_parsing
def main():
    …
    for line in sys.stdin:
        line_data = data_parsing.parse_line(line)
        …
```

Importing functions directly

- Importing function `parse_line` from module `data_parsing` in script `counting.py`:

counting.py

```
…
from data_parsing import parse_line

def main():
    …
    for line in sys.stdin:
        data = parse_line(line)
        …
```

```
from math import sqrt
from cmath import sqrt as csqrt
```

Never, ever

```
from my_module import *
```

More concise, but name clashes can occur!
E.g., `math.sqrt` versus `cmath.sqrt`

Double duty

data_parsing.py

```
#!/usr/bin/env python

from collections import namedtuple
Line_Data = namedtuple('Line_Data',
                       ['case_nr', 'dim_nr', 'temp'])

def parse_line(line, sep=None):
    data = line.rstrip('\r\n').split(sep)
    return Line_Data(case_nr=int(data[0]),
                     dim_nr=int(data[1]),
                     temp=float(data[2]))

if __name__ == '__main__':
    …
    for line in sys.stdin:
        line_data = parse_line(line)
        …
```

Only executed when run as script

Package layout & use example

- weave.py
- vsc
  - __init__.py
  - util.py
  - parameter_weaver
    - __init__.py
    - artifact.py
    - base_formatter.py
    - c
      - __init__.py
      - formatter.py
      - …
    - fortran
      - __init__.py
      - formatter.py
      - …
    - …

```
…
from vsc.parameter_weaver.c.formatter import Formatter
…
```

```
…
from vsc.parameter_weaver.base_formatter import BaseFormatter
…
```

https://github.com/gjbex/Python-software-engineering/tree/master/source-code/testing

WRITING DOCUMENTATION & SIMPLE TESTING

Writing documentation

- Documentation is very important! ⇒ use DocString

```
def parse_line(line, sep=None):
    '''Split a line into its fields, convert to the
    appropriate types, and return as a tuple.
    '''
# using \r, \n should work for Windows & *nix
    data = line.rstrip('\r\n').split(sep)
    return (int(data[0]), int(data[1]), float(data[2]))
```

```
>>> import data_parsing
>>> help(data_parsing.parse_line)
Help on function parse_line in module validator:

parse_line(line)
    Split a line into its fields, convert to the
    appropriate types, and return as a tuple
```

Formatting docstrings

```
def parse_line(line, sep=None):
    '''Split line into fields,
    converted to appropriate types

    Parameters
    ----------
    line: str
        line of input to parse
    sep: str
        field separator, default
        whitespace

    Returns
    -------
    tuple[int, int, float]
        data fields: case number,
        dimension number, temperature
    '''
    …
```

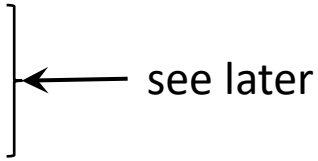
numpy/scipy

Many options

- Google
- reStructured Text
- numpy/scipy

What to document and how?

- DocString for
  - functions
  - classes
  - methods
  - modules
  - packages
- Comments
  - particular code fragments you had to think about



Assertions

- Testing pre and post conditions
  - Programming by contract

```
def fac(n):
    assert type(n) == int, 'argument must be integer'
    assert n >= 0, 'argument must be positive'
    if n < 2:
        return 1
    else:
        return n*fac(n - 1)
```

Optional

```
$ python  -c 'from fac import fac; print(fac(-1))'
…
assert n >= 0, 'argument must be positive'
AssertionError: argument must be positive
```

Assert use cases

- For development only, *not* production!
- *Not* a substitute for error handling, i.e., exception handling
- Run without assertions, run optimized: `-O`

```
$ python  -O  -c 'from fac import fac; print(fac(-1))'
1
```

Useful feature, but don't abuse!

Testing: meeting expectations

- Tests are important!
  - unittest: more features, but harder
  - doctest: simple

> A program that has not been tested does not work.
>
> — Bjarne Stroustrup

Statement to execute

```
def parse_line(line):
    '''Split a line into its fields, convert to the
    appropriate types, and return as a tuple.
    >>> parse_line('5  3  3.7')
    (5, 3, 3.7)
    '''
    data = line.rstrip('\r\n').split()
    return (int(data[0]), int(data[1]), float(data[2]))
```

Expected result

- Run tests

No output: hooray, all tests passed!

```
$ python  -m doctest data_parsing.py
$
```

Failing tests

```
def parse_line(line):
    '''Split a line into its fields, convert to the
    appropriate types, and return as a tuple.
    >>> parse_line('5  3  3.7')
    (5, 3, 3.7)
    >>> parse_line('5 3 3')
    (5, 3, 3)
    ''
    data = line.rstrip('\r\n').split()
    return (int(data[0]),
            int(data[1]),
            float(data[2]))
```

```
$ python  -m doctest  data_parsing.py
**********************************************
File "./data_parsing.py", line 9, in __main__.parse_line
Failed example:
    parse_line('5 3 3')
Expected:
    (5, 3, 3)
Got:
    (5, 3, 3.0)
**********************************************
1 items had failures:
   1 of   2 in __main__.parse_line
***Test Failed*** 1 failures.$
```

Further reading: documentation

- Documenting Python: a complete guide
  https://realpython.com/documenting-python-code/#docstring-formats

https://github.com/gjbex/Python-software-engineering/tree/master/source-code/testing

UNIT TESTING

Unit testing

- Key concepts
  - Implementation tested through API
  - Testing should be easy
  - Tests are independent of one another
- Find problems early/fast
- Facilitates change
  - Make small change, run tests
- TDD: Test Driven Development
  - Write tests first, then implement
- Programming framework, e.g., Python's unittest

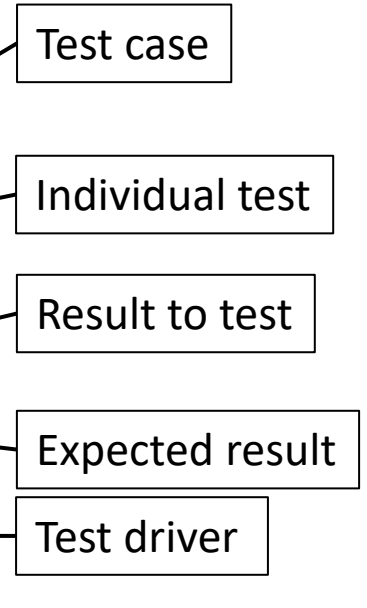
> "How to test?" is a question that cannot be answered in general. "When to test?" however, does have a general answer: as early and as often as possible.
>
> — Bjarne Stroustrup

Test case

- Subclass of `unittest.TestCase`
- Methods `test_<name>` are tests
- `unittest` provides driver for running tests

```
import unittest
from func_lib import fib

class FibTest(unittest.TestCase):

    def test_fib4(self):
        '''test for fib(4)'''
        self.assertEqual(3, fib(4))

if __name__ == '__main__':
    unittest.main()
```

fib_test.py



Running tests

- Run Python script

```
$ python  ./fib_test.py
F
======================================================================
FAIL: test_fib4 (__main__.FibTest)
test a number computations for small arguments
----------------------------------------------------------------------
Traceback (most recent call last):
  File "./fibber.py", line 13, in test_fib4
    self.assertEqual(expected, fib(4))
AssertionError: 3 != 5

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

Assert methods

- Many methods: provide accurate feedback
  - `assertEqual` for `int`, `str`
  - `assertAlmostEqual` for `float`, `complex`
  - `assertTrue`, `assertFalse` for `bool`
  - `assertListEqual`, `assertSetEqual`, `assertDictEqual`, `assertTupleEqual`
  - `assertIn`
  - `assertIsNone`
  - `assertIsInstance`
  - `assertRegex`

+ negations, e.g., `assertNotEqual`, …

Checking for expected failure

- Exceptions

```
from func_lib import fib, InvalidArgumentException
…
def test_negative_values(self):
    '''test for call with negative argument'''
    with self.assertRaises(InvalidArgumentException):
        fib(-1)
…
```

- Also useful: `assertRaisesRegex`
- Warnings: `assertWarns`

Subtests

- To check for a series of values

```
…
def test_low_values(self):
    '''test a number computations for small arguments'''
    expected = [0, 1, 1, 2, 3, 5, 8, 13]
    for n in range(len(expected)):
        with self.subTest(i=n):
            self.assertEqual(expected[n], fib(n))
…
```

Fixtures

- Prepare for test(s), clean up after test(s), e.g.,
  - Open/close a file
  - Open/close a database connection, initialize a cursor
  - Initialize data structures/objects
- Three levels
  - Before/after any test in module is run
    - `setUpModule()`/`tearDownModule()`
  - Before/after any test in test case class is run
    - `setUpClass(cls)`/`tearDownClass(cls)` (mark as `@classmethod`)
  - Before/after each individual test
    - `setUp(self)`/`tearDown(self)`

Module-level

- setUpModule: create and fill database

```
import init_db
…
def setUpModule():
    '''create and fill the database'''
    conn = sqlite3.connect(master_name)
    init_db.execute_file(conn, 'create_db.sql')
    init_db.execute_file(conn, 'fill_db.sql')
```

- tearDownModule: remove database

```
def tearDownModule():
    '''remove database file once testing is done'''
    os.remove(master_name)
```

Test case-level

- setUpClass: create copy of database

```
test_name = 'test.db'

@classmethod
def setUpClass(cls):
    '''copy original database'''
    shutil.copyfile(master_name, cls.test_name)
```

Test cases must be independent!

- tearDownClass: remove copy of database

```
@classmethod
def tearDownClass(cls):
    '''remove test database'''
    os.remove(cls.test_name)
```

Test-level

- setUp: create connection & cursor

```
def setUp(self):
    '''open connection, create cursor'''
    self._conn = sqlite3.connect(self.__class__.test_name)
    self._conn.row_factory = sqlite3.Row
    self._cursor = self._conn.cursor()
```

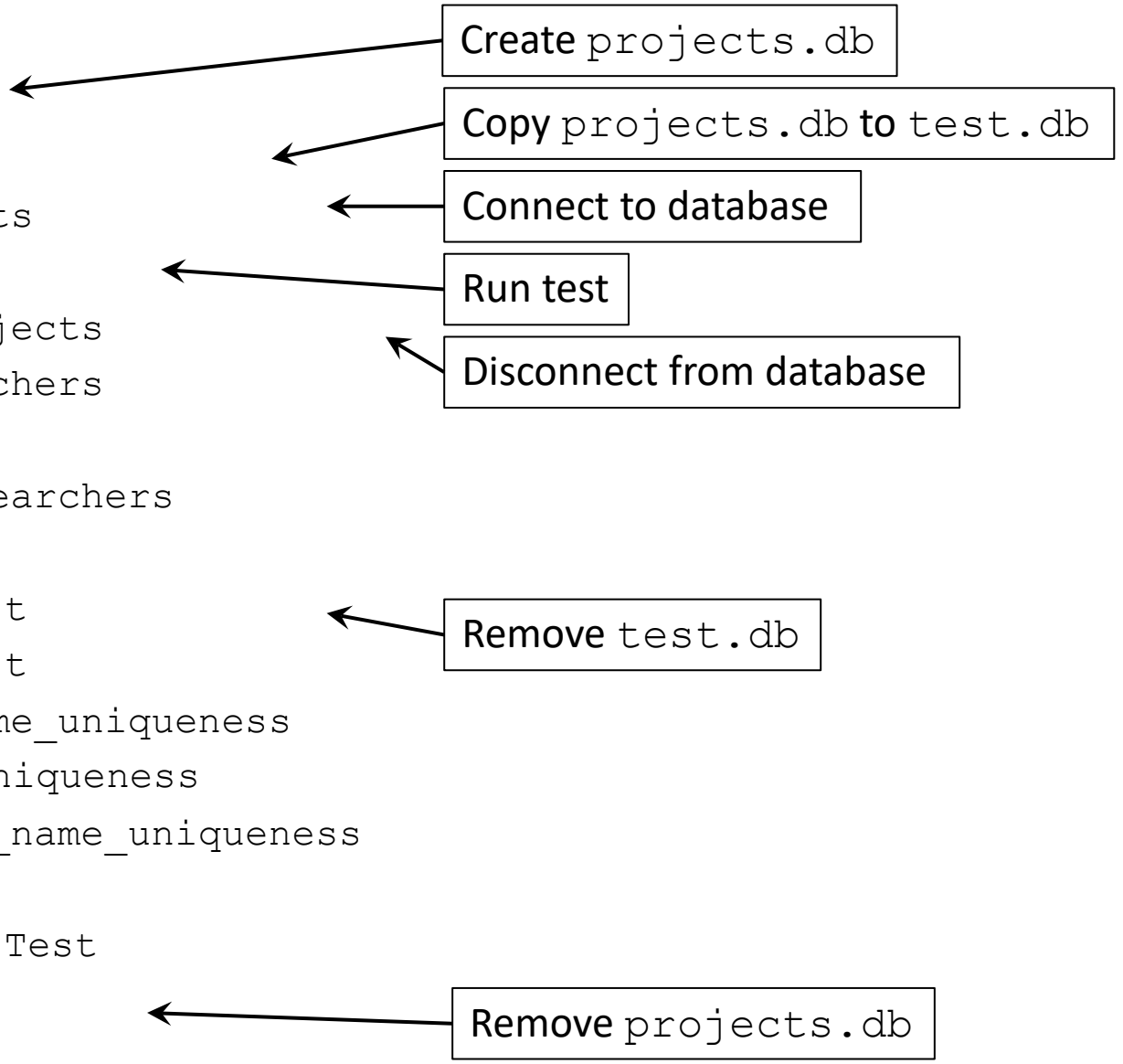
- tearDown: close connection

```
def tearDown(self):
    '''close database connection'''
    self._conn.close()
```

Tests must be independent!

Flow for fixtures

- `setUpModule` for `tests`
  - `setUpClass` for `ContentsTest`
    - `setUp` for `test_num_projects`
      - test_num_projects
    - `tearDown` for `test_num_projects`
    - `setUp` for `test_num_researchers`
      - test_num_researchers
    - `tearDown` for `test_num_researchers`
    - …
  - `tearDownClass` for `ContentsTest`
  - `setUpClass` for `ConstraintsTest`
    - `setUp` for `test_project_name_uniqueness`
      - `test_project_name_uniqueness`
    - `tearDown` for `test_project_name_uniqueness`
    - …
  - `tearDownClass` for `ConstraintsTest`
- `tearDownModule` for `tests`



Running all tests

- In module

```
…
if __name__ == '__main__':
    unittest.main()
```

fib_test.py

```
$ python  ./fib_test.py
```

- In all modules

```
$ python  -m unittest  discover  -p '*_test.py'
```

Test coverage

- Easy to overlook
  - functions/methods
  - code paths
- Use code coverage tool
  https://coverage.readthedocs.io/
- Steps
  - run code using `coverage run`
  - create detailed report using `coverage annotate`
  - add tests until covered

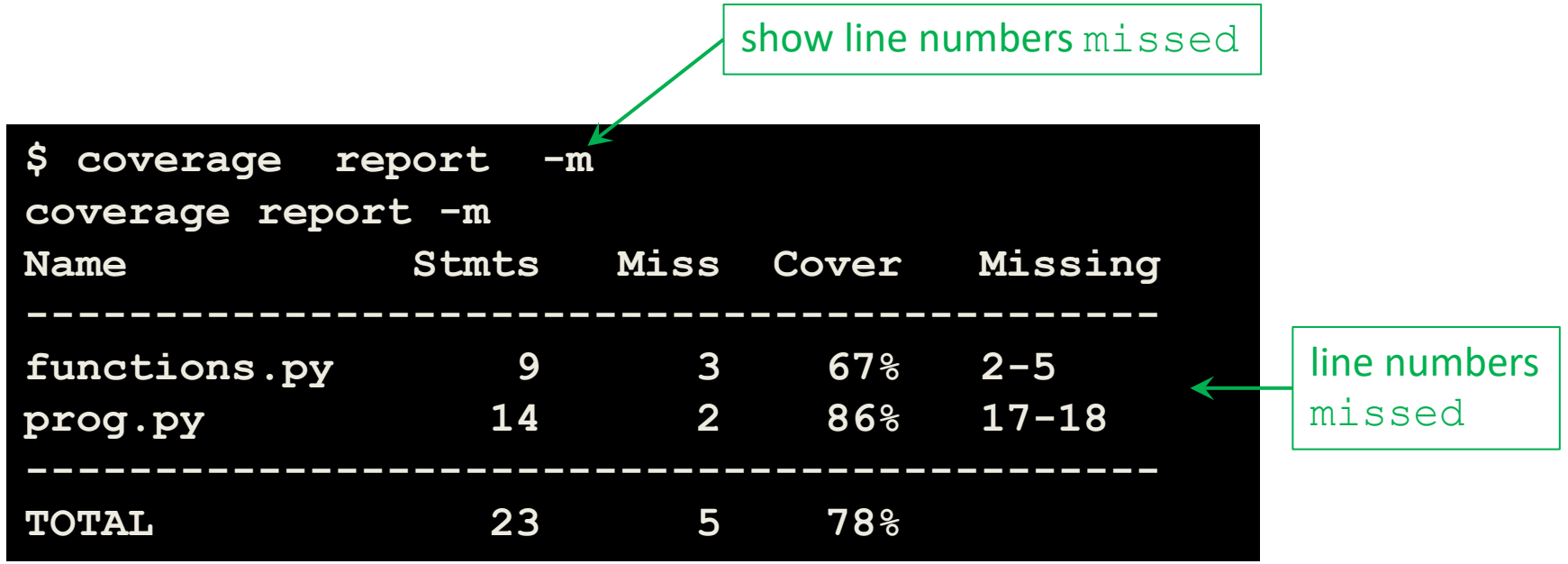
> *A program that has not been tested does not work.*
> — Bjarne Stroustrup

Coverage usage

- Run code

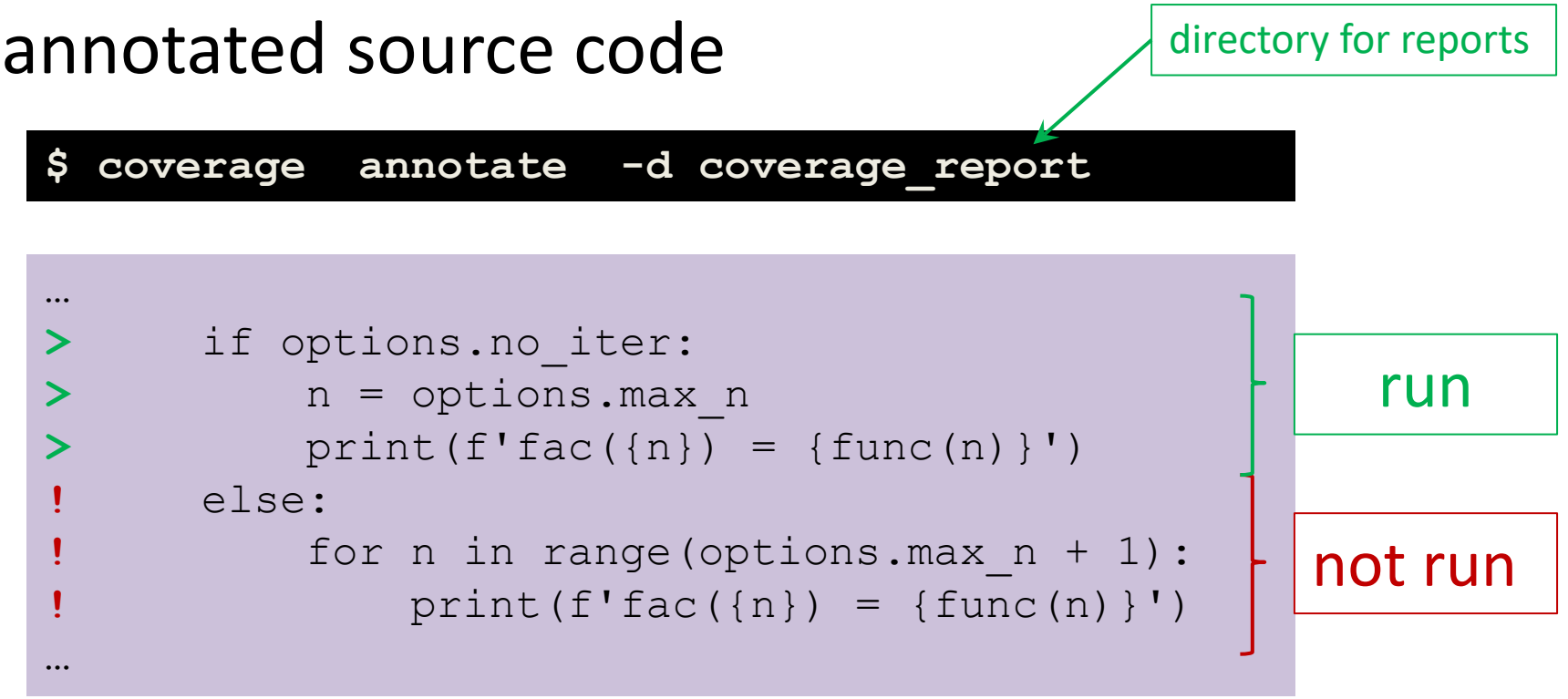
```
$ coverage  run  ./prog.py
…
```

- Report



Coverage usage

- Create annotated source code



- Remove coverage data

```
$ coverage erase
```

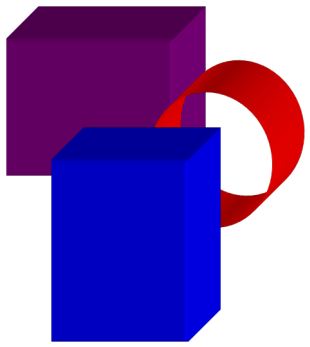
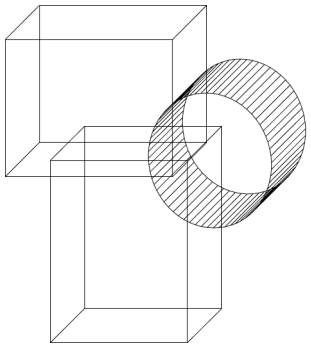
Further reading

- B. Kernighan & R. Pike (1999) *The practice of programming*, Addison-Wesley
- M. Fowler (1999) *Refactoring: improving the design of existing code*, Addison-Wesley

https://github.com/gjbex/Python-software-engineering/tree/master/source-code/object-orientation

OBJECT-ORIENTED PYTHON

Motivation: what is programming?

Real world	Model	Implementation
	 $\begin{cases} x_1 + 3y_1 + 2z_1 = c_1 \\ 3x_2 + 2y_2 + 2z_2 = c_2 \\ x_3 + 4y_3 - z_3 = c_3 \end{cases}$	def volume(object): …

Minimize discrepancies: real world ↔ model ↔ implementation

Object-orientation

- Python types are classes
  - e.g., `(14).bit_length() == 4`
    - `14` is an object of class `int`
    - `bit_length` is object method defined in class `int`

You are using objects all the time!

- Objects of simple Python types are immutable
  - Operations/methods instantiate new objects

Value versus object identity

- Simple Python types
  - Value identity: `(14 == 14) == True`
  - Object identity: `(14 is 14) == True`
    - However, Python version dependent!
- Other Python types, general classes
  - e.g., two `set` objects:
    `a = {'alpha'},b = {'alpha'}`
    - Value identity: `(a == b) == True`
    - Object identity: `(a is b) == False`

Defining your own classes

- Class definition:

```
class Point:
    …
```

- Objects are instances of classes
  - instantiated by calling constructor
  - have
    - attributes
    - methods
- Classes have
  - attributes
  - methods

A simple point…

```
from math import sqrt

class Point:

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def distance(self, other):
        return sqrt((self.x - other.x)**2 +
                    (self.y - other.y)**2)

    def __repr__(self):
        return f'({self.x}, {self.y})'
```

constructor for `Point` objects

method to compute distance

creates string representation for `Point` object

Making a point… or two

```
…
def main():
    p = Point(3, 4)
    q = Point(-2, 5)
    print(p.x, p.y)
    print(p, q)
    print(p.distance(q))
    p.x = 12.3
    print(p)…
```

create `Point p` at 3, 4

create `Point q` at -2, 5

access `p`'s x- and y-coordinates

calls `__str__` method indirectly on `p` and `q`

compute distance from `p` to `q`

modifying `p`

```
$ python point_driver.py
3.0 4.0
(3.0, 4.0) (-2.0, 5.0)
5.0990195136
(12.3, 4.0)
```

`distance` method invoked on `Point p`, with `Point q` as argument

More to the point…

- What if points should not be moved?

```
class Point:

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y
    …
```

constructor for `Point` objects

getter for object's `__x` attribute

getter for object's `__y` attribute

Making a definite point

```
…
def main():
    p = Point(3, 4)
    print(p.x, p.y)
    p.x = 12.3
…
```

create `Point p` at 3, 4

try to access `p`'s x-coordinate

```
$ python point_driver.py
3.0 4.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Object attributes

- Make object attributes "private" by hiding them, by convention, use `___` prefix

```
self.__x = x
```

- Create getter/setter method to control access to object attributes

```
@property
def x(self):
    return self.__x
```

Determine object's state

Object attribute can not accidently be modified, i.e., read-only

Object attributes: control

- Getter, but no setter

```
…
def main():
    p = Point(3, 4)
    print(p.x)
    p.x = 4.4
    print(p.x)
…
```

Protects against modification of read-only attributes

```
$ python point_driver.py
3.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Object attribute: setter

- Implementing setters improves control, assignment to attribute is "intercepted" by setter method

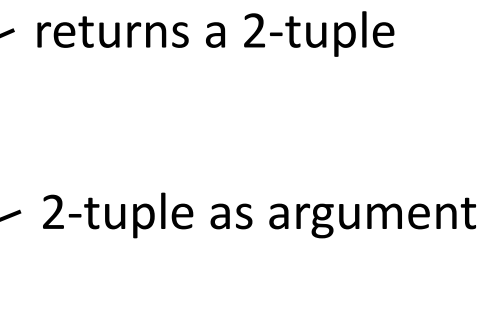
```
class Point:
…
    @x.setter
    def x(self, value):
        self.__x = float(value)
…
```

E.g., ensures proper type conversion:
`p.x = 3` results in `float`, not `int` for `__x` attribute

Non-trivial getter/setter

- Derived attribute: coordinates as 2-tuple

```
class Point:
…
    @property
    def coords(self):
        return (self.x, self.y)

    @coords.setter
    def coords(self, value):
        self.x = value[0]
        self.y = value[1]
…
# Use coords getter/setter
print(p.coords)
p.coords = (3.5, 7.1)
```



More object methods

```
from math import sqrt, isclose

class Point:
    …
    def on_line(self, p, q, tol=1.0e-6):
        if not isclose(p.x, q.x, abs_tol=tol):
            a = (q.y - p.y)/(q.x - p.x)
            b = p.y - a*p.x
            return isclose(self.y, a*self.x + b, abs_tol=tol)
        else:
            return isclose(self.x, p.x, abs_tol=tol)
…
# check whether r is on line defined by p and q
if r.on_line(p, q):
    …
```

Python 3.5+

`on_line` method invoked on `Point r`, with `Point p` and `q` as argument

Object methods

- Used to
  - retrieve information on object
  - modify or manipulate object
  - derive information from object with respect to other objects
  - …

Determine what objects can do, or can be done with

Static methods

```
…
class Point:
    …
    @staticmethod
    def all_on_line(p, q, *points):
        for r in points:
            if not r.on_line(p, q):
                return False
        return True
…
# check whether p, q, r, v and w are on a line
if Point.all_on_line(p, q, r, v, w):
    …
```

`all_on_line` method invoked on `Point` class with `Point p, q, r, v, w` as arguments, class ignored

Variable length argument lists

- Arbitrary positional arguments: `*argv`

```
@staticmethod
def all_on_line(p, q, *points):
    for r in points:
        if not r.on_line(p, q):
            return False
    return True
```

arguments

available as tuple

- Arbitrary keyword arguments: `**argv`
  - Available as dictionary

Note: not specific to object oriented programming

More elegant solution

- Semantics: `True` if `True` for all elements in points

```
@staticmethod
def all_on_line(p, q, *points):
    for r in points:
        if not r.on_line(p, q):
            return False
    return True
```

- More elegant: `all(…)`

```
@staticmethod
def all_on_line(p, q, *points):
    return all(r.on_line(p, q) for r in points)
```

- Similar: `any(…)`

Quick interlude

- What attributes/methods does a class have?

```
>>> from point import Point
>>> p = Point(3.7, 5.1)
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__doc__',
 '__format__', '__getattribute__', '__hash__',
 '__init__', '__module__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_Point__x', '_Point__y',
 'all_on_line', 'coords',
 'distance', 'on_line', 'x', 'y']
```

Inheritance

- Class can extend other class
- For Python 2: make classes inherit from `object`, ensure they can be extended later:
  ```
  class Point(object):
  ```
- New class inherits attributes & methods from parent class
- New class can implement new methods, define new attributes
- New method can override methods of parent class
- New class can inherit from multiple parent classes

Points with mass

```
class PointMass(Point):

  def __init__(self, x, y, mass):
    super().__init__(x, y)
    self.__mass = float(mass)

  @property
  def mass(self):
    return self.__mass

  def __repr__(self):
    return '{0}: {1}'.format(
      super().__repr__(),
      self.mass)
```

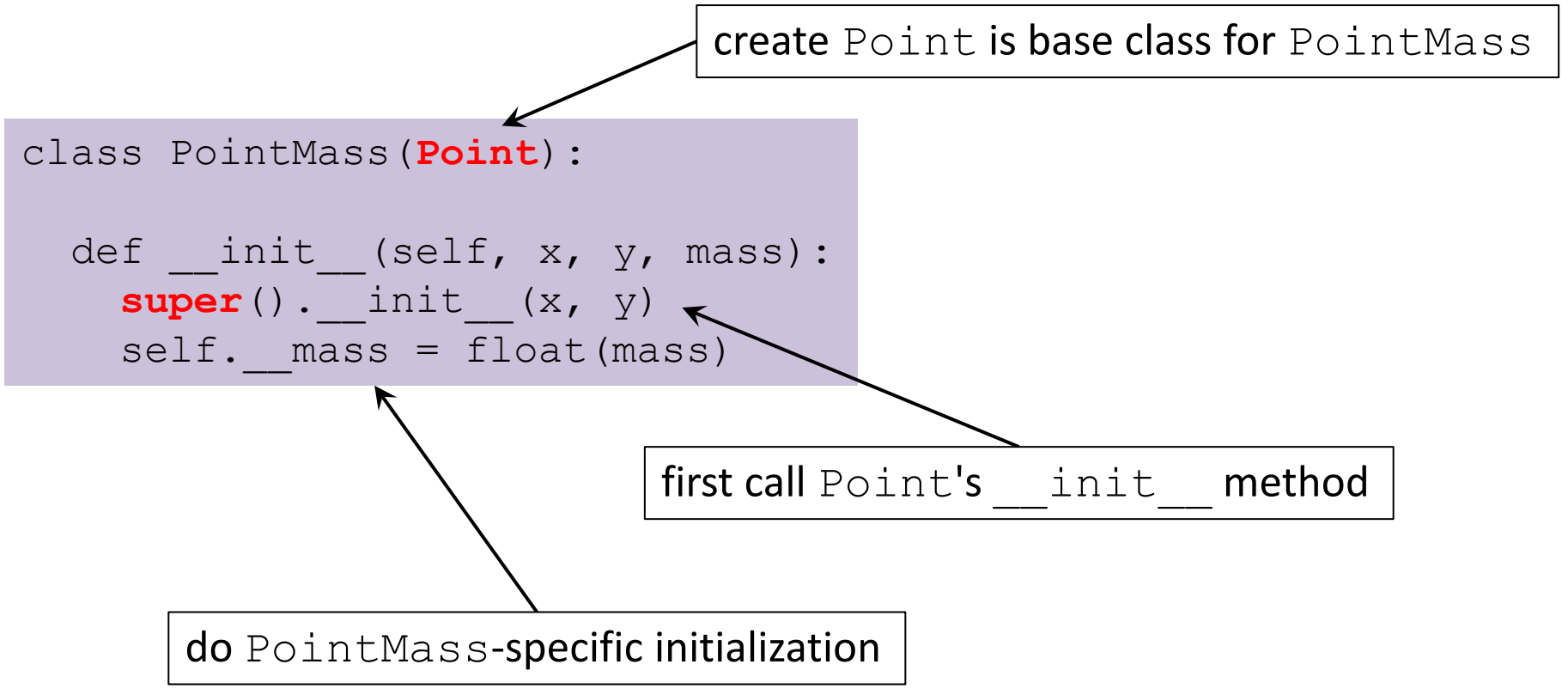
constructor of `Point` overridden

new object method

`__repr__` method of `Point` overridden

`PointMass` objects have `x`, `y`, `distance`, `on_line` methods as well
`PointMass` class has `all_on_line` methods

Base classes & derivation



Point with mass is still Point

```
def main():
    p = PointMass(3, 4, 1)
    q = Point(-2, 5)
    print(p.x, p.y, p.mass)
    print(p.distance(q))
```

create `PointMass p` at 3, 4 and mass 1

create `Point q` at -2, 5

`p` is a Point, so has `distance` method

```
$ python point_driver.py
3.0 4.0 1.0
5.09902
```

Class attributes

```
class PointMass(Point):

    __default_mass = 1.0

    def __init__(self, x, y, mass=None):
        super().__init__(x, y)
        if mass is not None:
            self.__mass = float(mass)
        else:
            self.__mass = PointMass.__default_mass
    …
    @classmethod
    def set_default_mass(cls, mass):
        cls.__default_mass = float(mass)
```

class variable `__default_mass`

setter for class' `__default_mass` attribute

Determine state of class

All those methods

- Object methods
  - work on individual objects
  - take object as first argument (`self`)
- Class methods
  - `@classmethod`
    - work at class level
    - take class as first argument (`cls`)
  - `@staticmethod`
    - work at class level
    - ignores object or class it is called on

One more point

- String representation
  - `__repr__()`: for development, debugging, unambiguous
    - called by `repr()`, `!r` conversion in f-strings
    - called on iPython/Jupyter command line
    - called by `str()` if no `__str__ ()` defined
    - called by debug f-string (Python 3.8+)
  - `__str__()`: for human consumption
    - called by `str()`, `!s` conversion in f-strings

Alternatives

- (named) tuples
  - Lightweight √
  - No methods ✗
- `@dataclass` (Python 3.7+)
  - Methods √
  - No private attributes, no validation, no factories ✗
- attrs
  - Many features out of the box √
  - Third party ✗

Further reading

- dataclasses:
  https://realpython.com/python-data-classes/#more-flexible-data-classes

https://github.com/gjbex/Python-software-engineering/tree/master/source-code/design-patterns/finite-state-parser

DATA REPRESENTATION: PYTHON CLASSES CASE STUDY

Going OO: data abstraction

```
begin block_1
        0.322617473156
        0.369558068115
        0.732467264353
        0.584389170786
end block_1
begin block_2
        0.0705959106085
        0.65856743041
        0.526762713499
        0.823193820644
end block_2
begin block_3
        0.476180897605
        0.783168943997
        0.0172156882251
        0.755582680088
end block_3
```

block

- Data consists of multiple blocks
- Blocks have
  - Name
  - One or more data values
- How to represent?
  - Python class

Class Block: attributes

Class is abstract definition: attributes represent data

```
 1 class Block:
 2   def __init__(self, name):
 3     self.__name = name
 4     self.__data = []
```

```
>>> block1 = Block('my block')
```

object of type Block

```
>>> block1.__data.append(3.14)
>>> block1.__data.append(-7.18)
```

```
>>> block2 = Block('another')
```

second object of type Block

- "abstract" Block has attributes
  - `__name`: string
  - `__data`: list
- Block instance `block1` has
  - `'my block'` as `_name`
  - No values (yet) as `_data`
- Store some data, `block1` now has two data values: 3.14 and -7.18
- Create new block `block2`

Object is container for specific data: attribute values hold data

Class Block: methods

- What do we want to do with a block?
  - Convert it to a string
  - Retrieve its name
  - Add data to it
  - Sort its data
  - Retrieve its data
- How?
  - Define methods for the class Block

Class is abstract definition: methods represent actions on objects

Class Block: method implementations

```
 1 class Block:
 2   def __init__(self, name):
 3     self.__name = name
 4     self.__data = []
 5
 6   def __str__(self):
 7     header = f'begin {self.name()}'
 8     data = '\n\t'.join([str(item) for item in self.data()])
 9     footer = f'end {self.name()}'
10     return f'{header}\n\t{data}\n{footer}'
11
12   def name(self):
13     return self.__name
14
15   def add_data(self, data):
16     self._data.append(data)
17
18   def sort_data(self):
19     self._data.sort()
20
21   def data(self):
22     return self.__data
```

```
>>> block1.add_data(12.5)
>>> print(block1.name())
my_block
>>> print(block1)
begin my_block
    3.14
    -7.18
    12.5
end my_block
```

https://github.com/gjbex/Python-software-engineering/tree/master/source-code/operators-functools
https://github.com/gjbex/Python-software-engineering/tree/master/source-code/iterators

PYTHON FUNCTIONAL PROGRAMMING

Motivation

- Side effect free
  - Less bugs
  - Easier to reason on
- Natural
  - Resembles mathematics
- Promotes concurrency

Sorting a simple list

- Two ways to sort
  - Create new list: `sorted(…)`

```
>>> data = [3.1745, 18.14, -6.49043]
>>> sorted(data)
[-6.49043, 3.1745, 18.14]
```

  - In-place sort `list.sort()` method

```
>>> data = [3.1745, 18.14, -6.49043]
>>> data.sort()
>>> print(data)
[-6.49043, 3.1745, 18.14]
```

- Sorts in ascending order, add `reverse=True` for descending

Sorting a complex list: key function

- List of tuples, e.g., word counts

```
>>> data = [('table', 15), ('chair', 5), ('bed', 19)]
```

- Sort by
  - Word:

```
>>> sorted(data, key=lambda x: x[0])
[('bed', 19), ('chair', 5), ('table', 15)]
```

  - Count:

```
>>> sorted(data, key=lambda x: x[1])
[('chair', 5), ('table', 15) , ('bed', 19)]
```

- Simpler using `operator`

```
>>> from operator import itemgetter
>>> sorted(data, key=itemgetter(0))
[('bed', 19), ('chair', 5), ('table', 15)]
```

Going functional: mapping

- list comprehensions: construct list from elements of other list by applying function
  - E.g., `[str(x) for x in [0.15, 3.145]] == ['0.15', '3.145']`

```
>>> data = [3.1745, 18.14, -6.49043]
>>> print(','.join([str(number) for number in data]))
3.1745,18.14,-6.49043
```

```
>>> data = [3.1745, 18.14, -6.49043]
>>> print(','.join(map(str, data)))
3.1745,18.14,-6.49043
```

```
>>> data = [3.1745, 18.14, -6.49043]
>>> print(','.join([f'{number:.2f}' for number in data]))
3.17,18.14,-6.49
```

Going functional: filtering

- list comprehensions: construct list from elements of other list for elements that pass test
  - E.g., `[x for x in [0.15, -3.45, 1.3]`
    `if x >= 0.0]`
    `≡ [0.15, 1.3]`
  - E.g., `[sqrt(x) for x in [4.0, -4.0, 9.0]`
    `if x >= 0.0]`
    `≡ [2.0, 3.0]`
  - Alternative:

```
list(map(sqrt,
         filter(lambda x: x >= 0.0,
                [4, -4, 9])))
```

iterators

Don't forget `from math import sqrt`

Going functional: aggregating

- Data in a list should be aggregated, e.g.,
  - Summation: use `sum(…)`
  - Minimum: use `min(…)`
  - Maximum: use `max(…)`

work on `str`
have optional `key` argument

- More sophisticated, use `reduce(…)` and lambda functions

```
>>> from functools import reduce
>>> data = [3, 12, 7]
>>> reduce(lambda x, y: x*y, data, 1)
252
```

lambda function

list

initializer

Lambda functions: very small functions (expression) used once, not worth giving a name

Going functional: zip it

- Two (or more) lists should be processed element wise
  - E.g., [x*y for x, y in zip(a, b)]

```
>>> a = [3.5, 7.3, 5.7]
>>> b = [2.0, 1.5]
>>> [x*y for x, y in zip(a, b)]
[7.0, 10.95]
```

iterator produces tuples

  - As many tuples as length of shortest list

Primes version 1.0: iterator

- Function to check whether n is prime

```
def is_prime(n):
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return n > 1
```

- Work with all primes up to 10^6? Simple…

```
for n in (i for i in range(1000000) if is_prime(i)):
    …
```

List comprehensions vs. generators

- List comprehension

```
[i for i in range(1000000) if is_prime(i)]
```

  – all prime numbers up to 1000000 (≈ 78,500)

- Generator

```
(i for i in range(1000000) if is_prime(i))
```

  – get next prime number when needed, much more memory efficient

Primes version 2.0: `yield`

- What if we want the first 10^6 prime numbers?
  - Guess range?
- Function that returns next prime at each call?
  - use yield

```
def all_primes():
    i = 2
    while True:
        if is_prime(i):
            yield i
        i += 1
```

- Iterator: first call yields `2`, second `3`, third `5`, …

```
for n in all_primes():
    …
```

`yield` statement

- Somewhat like `return`
  - returns control to the calling function
  - returns a value
- However, callee function state is retained
  - on next call, continues at the point it was when it yielded

Allows to build your own iterators

Primes version 3.0: `filter`

- Standard library package `itertools` provides a lot of useful iterators, check it out!
  - `itertools.count()`: iterator over integers

```
import itertools
…
nr_primes = 0
for n in filter(is_prime, itertools.count()):
    if nr_primes > 1000000:
        break
    nr_primes += 1
…
```

Other useful functions in `itertools`

- Permutations of an iterable:
  `itertools.permutations(…)`
- Combinations `r` out of an iterable:
  - Without replacement:
    `itertools.combinations(…, r)`
  - With replacement:
    `itertools.combinations_with_replacement(…, r)`
- Carthesian product of two (or more) iterables:
  `itertools.product(…, …)`
- Take while boolean predicate `pred` is true:
  `itertools.takewhile(pred, …)`
- Cycle through values of iterable:
  `itertools.cycle(…)`

Generating data (again)

```
from itertools import product as iproduct
…
print('case', 'condition', 'temperature')
for i, data in enumerate(iproduct(range(1, 4),
                                  map(lambda x: 0.5*x,
                                      range(-1, 2)))):
    print(i + 1, *data)
```

```
case dim temp
1 1 -0.5
2 1 0.0
3 1 0.5
4 2 -0.5
5 2 0.0
…
9 3 0.5
```

Further reading: functional style

- Sorting how-to
  http://docs.python.org/3.7/howto/sorting.html
- Functional programming how-to
  http://docs.python.org/3.7/howto/functional.html
- Luciano Ramalho (2015) *Fluent Python,* O'Reilly

https://github.com/gjbex/Python-software-engineering/blob/master/source-code/design-patters/factory_design_pattern.ipynb
https://github.com/gjbex/Python-software-engineering/blob/master/source-code/design-patters/decorator_design_pattern.ipynb
https://github.com/gjbex/Python-software-engineering/tree/master/source-code/design-patterns/finite-state-parser

DESIGN PATTERNS

Motivation

- Observation: patterns in problems to solve
- Design pattern = recipe for software design
  - Don’t reinvent the wheel

Patterns

- Quite a lot (23 originally)
  - Creational patterns
    - E.g., builder, factory
  - Structural patterns
    - E.g., decorator
  - Behavioral patterns
    - E.g., state
  - Parallel patterns

Builder

- Configure, i.e., “build” an object step by step
  - E.g., mplemented for scipy’s ODE solver algorithm

```
from scipy.integrate import ode
…
ode_sys = ode(func, jac).set_integrator('dopri5’)                    \
                        .set_initial_value([theta0, omega0], t0)  \
                        .set_f_params(g, l, q, F_D, Omega_D)      \
                        .set_jac_params(g, l, q, F_D, Omega_D)

while ode_sys.successful() and ode_sys.t < t_max:
    ode_sys.integrate(ode_sys.t + delta_t)
    print(ode_sys.t, ode_sys.y[0], ode_sys.y[1])
```

Solver is built step by step

=

Reduces risk of mistakes

Factory

- Create factory objects
  - Encapsulate configuration in factory objects
  - Create new objects using factory object

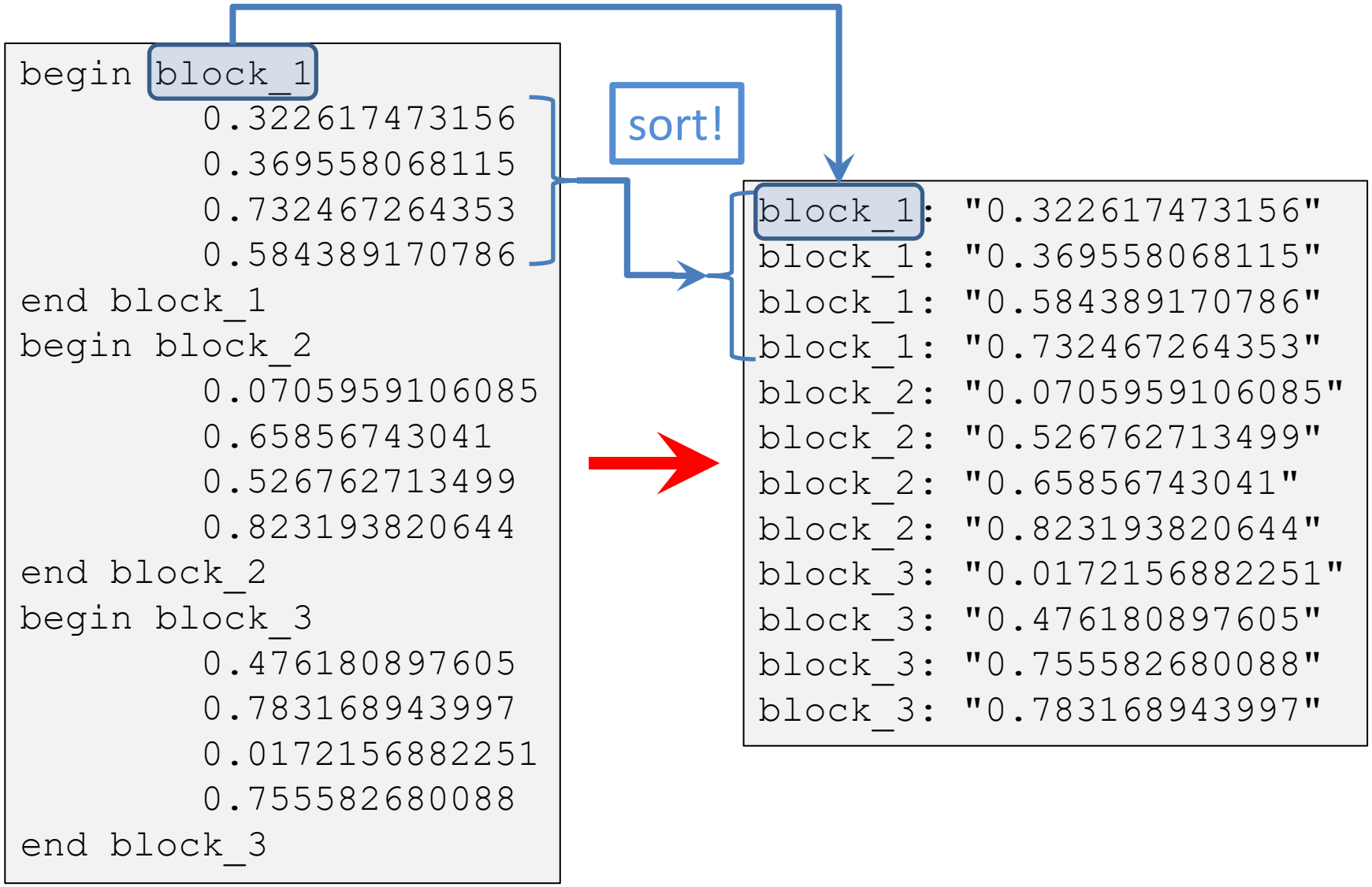
https://github.com/gjbex/Python-software-engineering/blob/master/source-code/design-patters/factory_design_pattern.ipynb

Decorator

- Wrapper objects that modify behavior of encapsulated objects
  - Intercept method calls to modify
  - Pass through all else

https://github.com/gjbex/Python-software-engineering/blob/master/source-code/design-patters/decorator_design_pattern.ipynb

State pattern: convert data



State pattern: model the data

```
begin block_1
        0.322617473156
        0.369558068115
        0.732467264353
        0.584389170786
end block_1
begin block_2
        0.0705959106085
        0.65856743041
        0.526762713499
        0.823193820644
end block_2
begin block_3
        0.476180897605
        0.783168943997
        0.0172156882251
        0.755582680088
end block_3
```

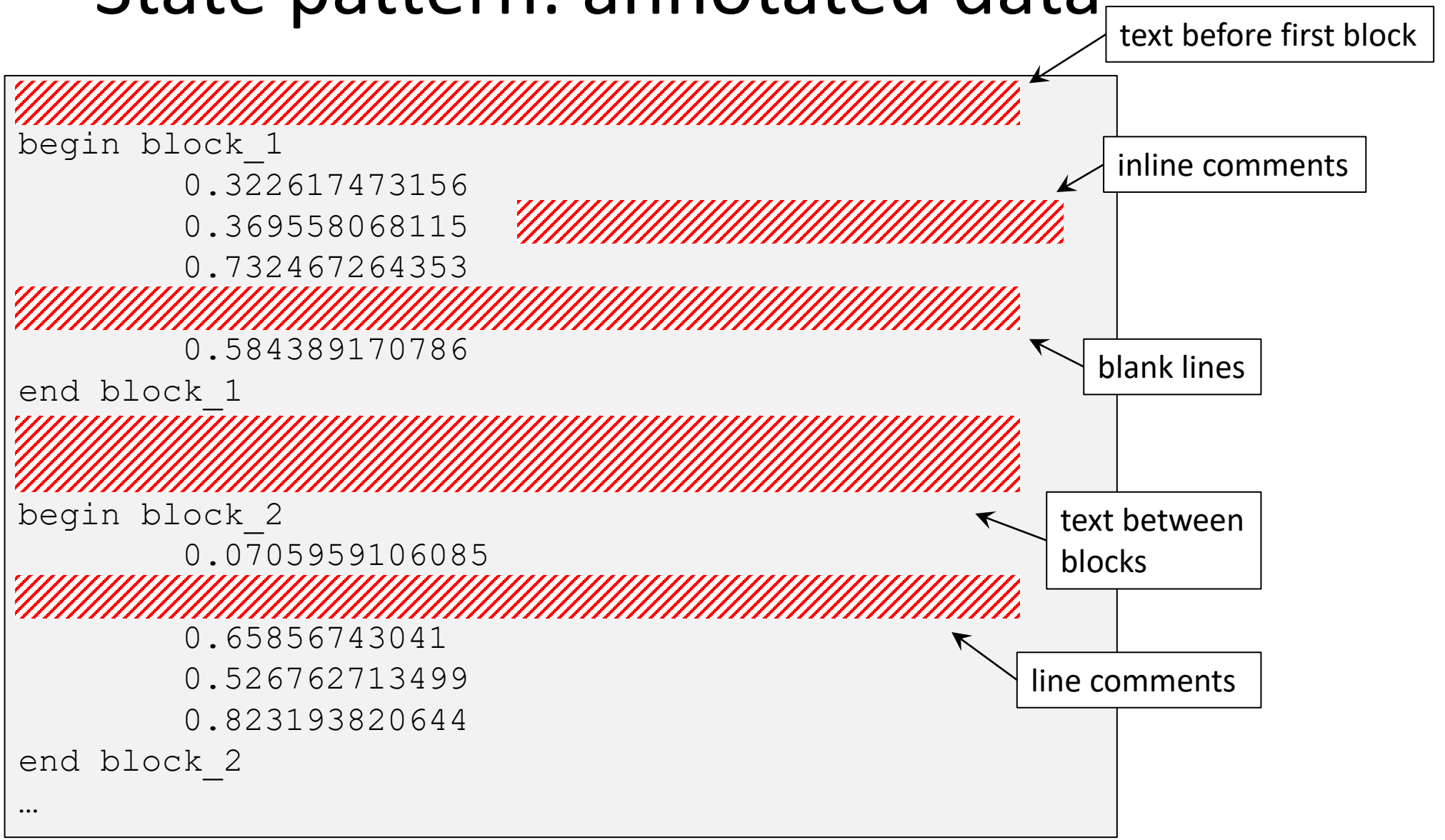
file := *block*…

block := begin *name*
  data
  …
  end *name*

name := string

data := real

State pattern: annotated data



State pattern: improved model

```
file    := junk* (block junk*)+
block   := begin name comment?
             (comment | empty line)*
             data comment?
             (comment | empty line)*
             …
           end name comment?
name    := string
data    := real
comment := # string
junk    := string |
           empty line
```

Notation:

?: zero or one

*: zero or more

+: one or more

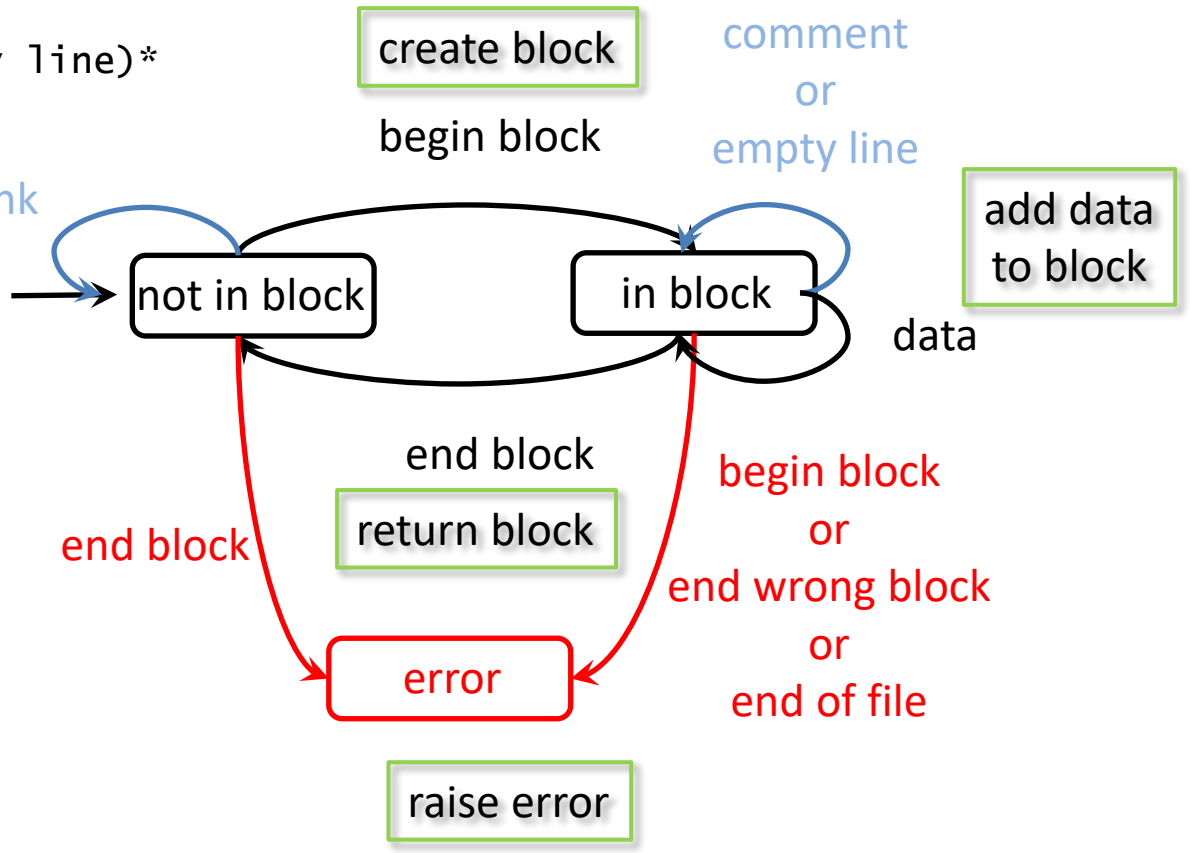
|: either, or (choice)

```
This data was produced using SomeSoftware™
begin block_1
        0.322617473156
        0.369558068115   # this value is suspicious
        0.732467264353

        0.584389170786
end block_1
There can be anything between blocks,
all kind of useful information or inane chatter.
begin block_2
        0.0705959106085
# this is a comment about all values below
        0.65856743041
        0.526762713499
        0.823193820644
end block_2
…
```

State pattern: computable model

```
file    := junk* (block junk*)+

block   := begin name comment?
             (comment | empty line)*
             data comment?
             (comment | empty line)*
             …
           end name comment?

name    := string

data    := real

comment := # string

junk    := string |
           empty line
```

Finite state automaton



State pattern: class BlockParser

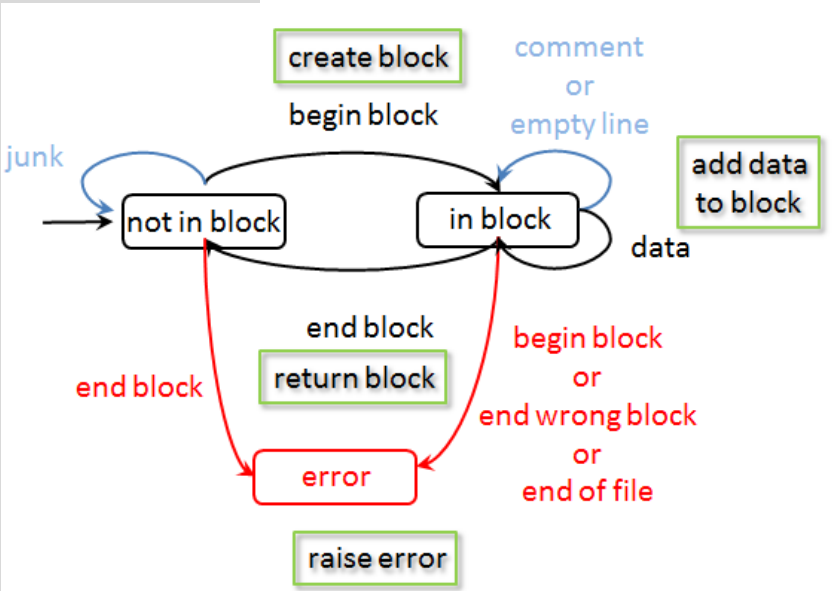
```
class BlockParser:

 1 def __init__(self):
 2   self.__states = ['in_block', 'not_in_block', 'error']
 3   self.__comment_pattern = re.compile(r'#.*')
 4   self.__block_begin_pattern = re.compile(r'begin\s+(\w+)')
 5   self.__block_end_pattern = re.compile(r'end\s+(\w+)')
 6   self.__state = None
 7   self.__current_block = None
 8   self.__blocks = None
 9   self.__match = None
10   self.__reset()
```

```
 1 def __reset(self):
 2   self.__set_state('not_in_block')
 3   self.__current_block = None
 4   self.__blocks = []
 5   self.__match = None
```

Must be called before parsing a new file

State pattern: from model to code

```
 1 for line in block_file:
 2   line = self.__preprocess(line)
 3   if self.__is_in_state('not_in_block'):
 4     if self.__is_begin_block(line):
 5       self.__init_block()
 5       self.__set_state('in_block')
 6     elif self.__is_end_block(line):
 7       raise DanglingEndBlockError(self)
 8   elif self.__is_in_state('in_block'):
 9     if self.__is_begin_block(line):
10       raise NestedBlocksError(self)
11     elif self.__is_end_block(line):
12       if self.__end_matches_begin():
13         self.__finish_block()
14         self.__set_state('not_in_block')
15       else:
16         raise NonMatchingBlockDelimitersError(self)
17     elif self.__is_data(line):
18       self.__add_data(line)
19   else:
20     raise UnknownStateError(self.get_state())
21 if self.__s_in_state('in_block'):
22   raise NonClosedBlockError(self)
```



Further reading: design patterns

- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (1994) *Design patterns: elements of reusable object-oriented software*, Addison-Wesley
- Kamon Ayeva and Sakis Kesampalis (2018) *Mastering Python Design patters,* Packt>