Coding Bootcamp Code in Python

CONTEXT MANAGERS

Context manager in Python

```python
# before Python 2.5
f_obj = open(path, 'w')
f_obj.write(some_data)
f_obj.close()

# after Python 2.5  + with statement
with open(path, 'w') as f_obj:
    f_obj.write(some_data)
```

- with statement automatically creates a context manager
- The way this works under the covers is by using some of Python’s magic methods: **__enter__** and **__exit__**.

Creating a Context Manager class

```
import sqlite3


class DataConn:
    """"""

    def __init__(self, db_name):
        """Constructor"""
        self.db_name = db_name

    def __enter__(self):
        """
        Open the database connection
        """
        self.conn = sqlite3.connect(self.db_name)
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        """
        Close the connection
        """
        self.conn.close()
        if exc_val:
            raise

if __name__ == '__main__':
    db = 'test.db'
    with DataConn(db) as conn:
        cursor = conn.cursor()
```

Creating a Context Manager using contextlib

```
from contextlib import contextmanager

@contextmanager
def file_open(path):
    try:
        f_obj = open(path, 'w')
        yield f_obj
    except OSError:
        print("We had an error!")
    finally:
        print('Closing file')
        f_obj.close()

if __name__ == '__main__':
    with file_open('test.txt') as fobj:
        fobj.write('Testing context managers')
```

contextlib.closing(thing)

- The difference is that instead of a decorator, we can use the closing class itself in our with statement

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.google.com')) as webpage:
    for line in webpage:
        # process the line
        pass
```

contextlib.suppress(*exceptions)

- It can suppress any number of exceptions

```
from contextlib import suppress

with suppress(FileNotFoundError):
    with open('fauxfile.txt') as fobj:
        for line in fobj:
            print(line)
```

contextlib.redirect_stdout / redirect_stderr

- The contextlib library has a couple of tools for redirecting stdout and stderr

```
from contextlib import redirect_stdout

path = 'text.txt'
with open(path, 'w') as fobj:
    with redirect_stdout(fobj):
        help(redirect_stdout)
```

ExitStack

- Context manager that will allow to easily programmatically combine other context managers and cleanup functions

```
from contextlib import ExitStack

with ExitStack() as stack:
    file_objects = [stack.enter_context(open(filename))
        for filename in filenames]
```

Reentrant Context Managers

- If an instance of a context manager try running it twice with Python’s with statement.

- The second time it runs, it raises a RuntimeError.

- But what if we wanted to be able to run the context manager twice?
- We’d need to use one that is “**reentrant**”.

Code Pack 15

- See the files

1.Creating_a_Context_Manager_class

2.Creating_a_Context_Manager_using_contextlib

3.contextlib.closing(thing)

4.contextlib.suppress(exceptions)

5.contextlib.redirect_stdout_redirect_stderr

6.ExitStack

7.Reentrant_Context_Managers

Coding Bootcamp Code in Python

UNIT TESTING

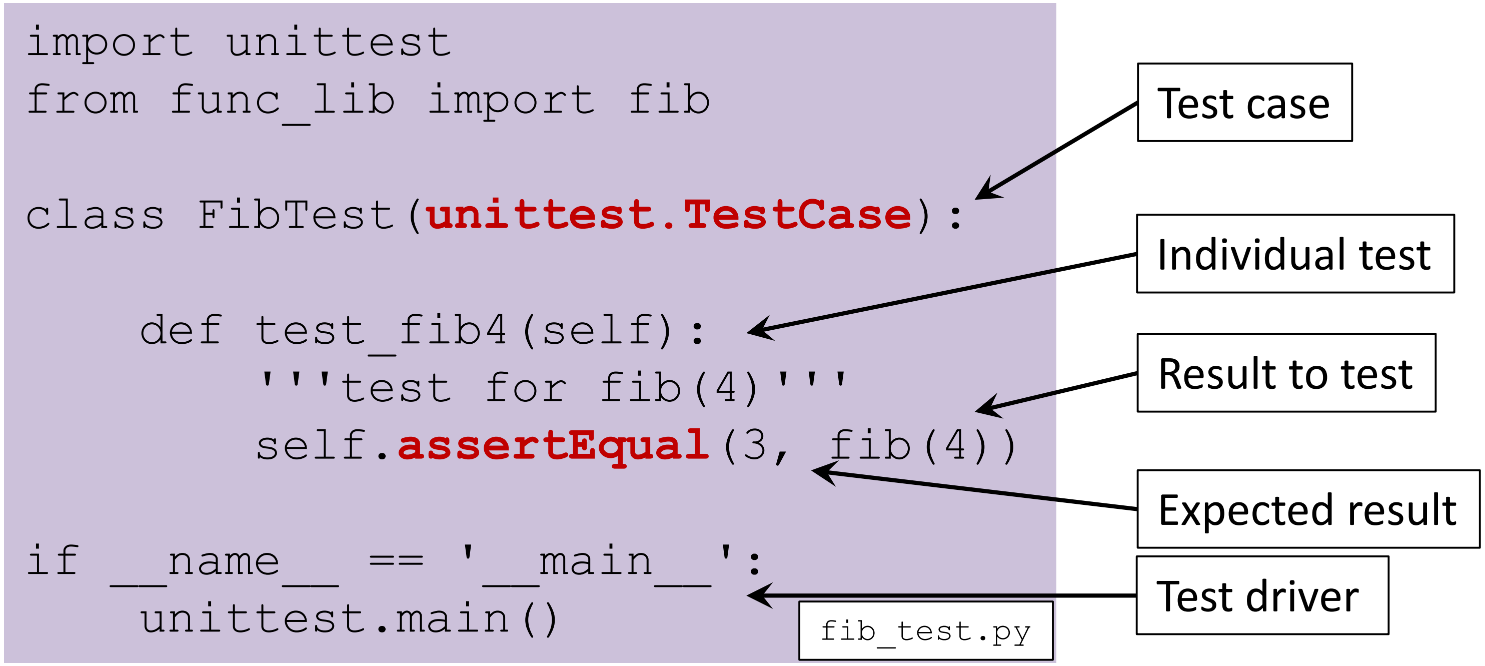
Unit testing

- 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

Test case

Individual test

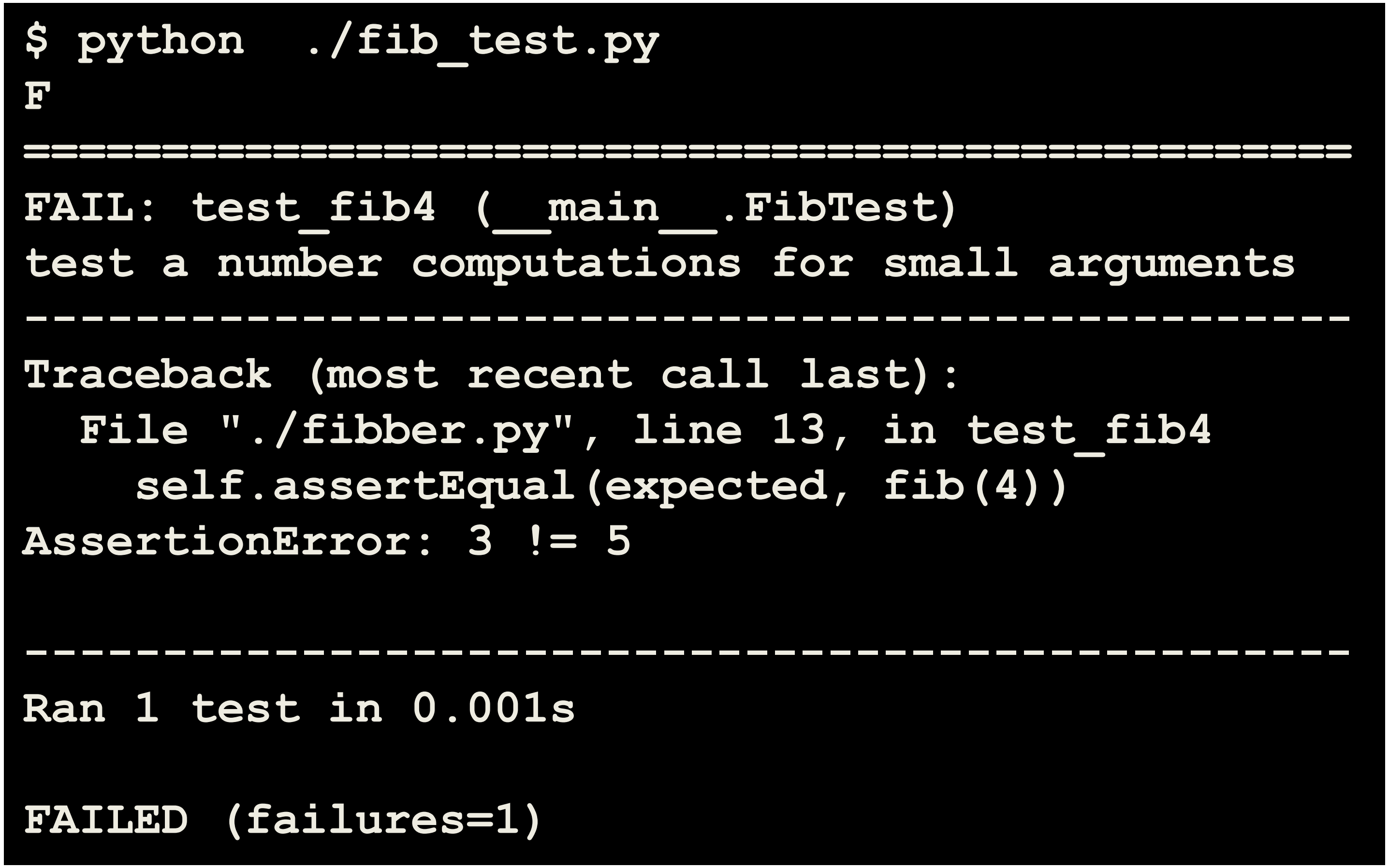
Result to test

Expected result

Test driver

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

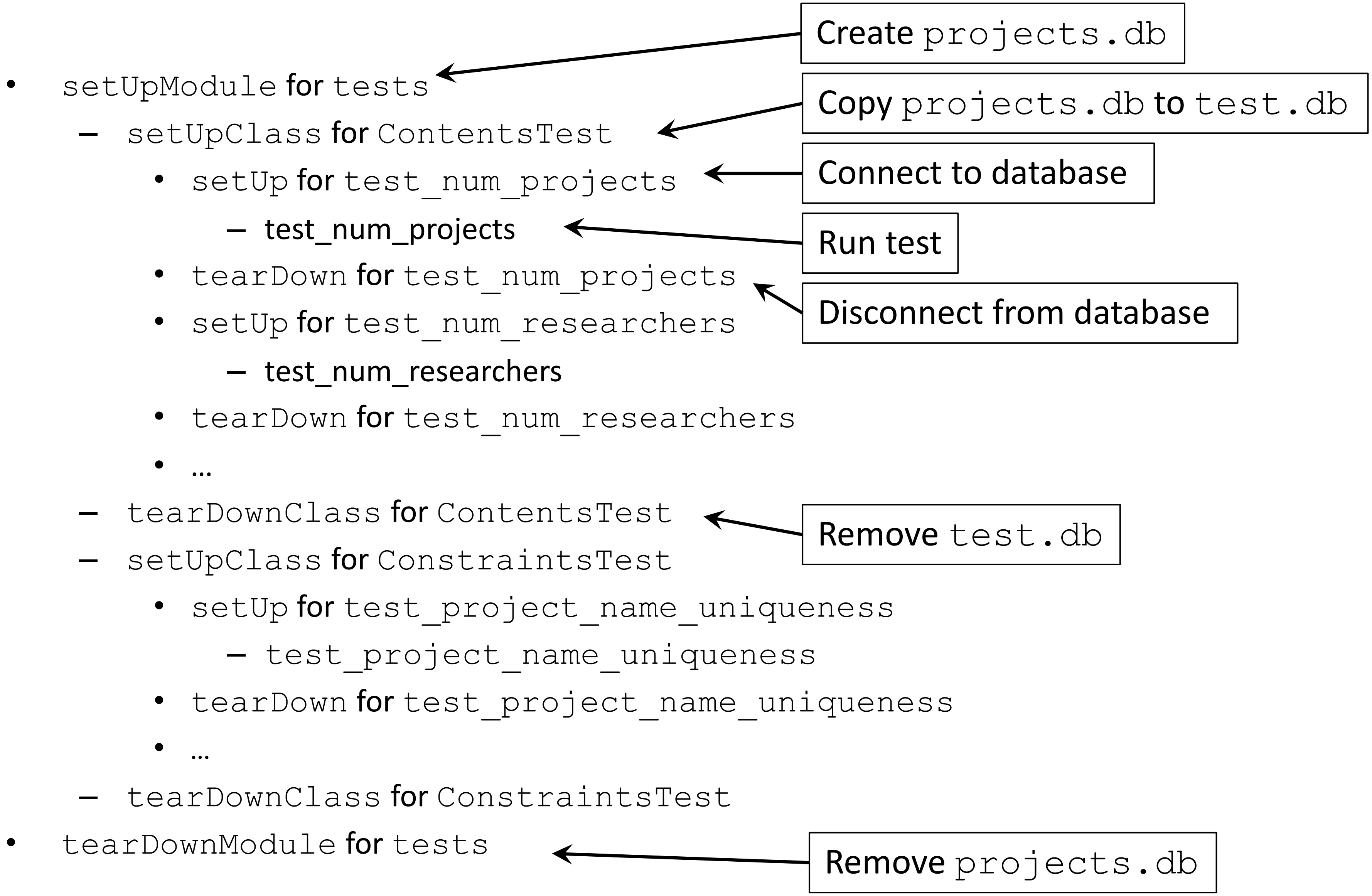
```
def setUp(self):
    '''open connection, create cursor'''
    self._conn = sqlite3.connect(self.__class__.test_name)
    self._conn.row_factory = sqlite3.Row
    self._cursor = self._conn.cursor()
```

- 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`

Create `projects.db`

Copy `projects.db` to `test.db`

Connect to database

Run test

Disconnect from database

Remove `test.db`

Remove `projects.db`

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 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

show line numbers missed

```
$ coverage  report  -m
coverage report -m
Name              Stmts   Miss  Cover   Missing
-----------------------------------------------
functions.py          9      3    67%   2-5
prog.py              14      2    86%   17-18
-----------------------------------------------
TOTAL                23      5    78%
```

line numbers missed

Coverage usage

- Create annotated source code

```
$ coverage  annotate  -d coverage_report
```

directory for reports

```
…
>     if options.no_iter:
>         n = options.max_n
>         print(f'fac({n}) = {func(n)}')
!     else:
!         for n in range(options.max_n + 1):
!             print(f'fac({n}) = {func(n)}')
…
```

run

not run

- 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

Code Pack 16

- See the files:

1.unittest

2.coverage.py

Coding Bootcamp Code in Python

ARGPARSE, CONFIGPARSER

Handling command line arguments

- Many tools start out as short script, evolve into applications used by many
- Model after Unix tools
  - Arguments
  - Flags
  - Options
- Python's `argparse` benefits
  - Easy to use
  - Self-documenting

Defining command line arguments

- Use `argparse` library module

```
from argparse import ArgumentParser
arg_parser = ArgumentParser(description='Gaussian random number generator')
```

- Add positional argument(s)

```
arg_parser.add_argument('nr', metavar='n', type=int, nargs='?', default=1,
                        help='number of random numbers to generate')
```

- Add flag(s)

```
arg_parser.add_argument('-idx', action='store_true', dest='index',
                        help='print index for random number')
```

- Add option(s)

```
arg_parser.add_argument('-mu', type=float, default=0.0,
                        help='mean of distribution')
```

`dest='mu'` is implicit

- Parse arguments

```
args = arg_parser.parse_args()
```

Using command line arguments

```
for i in range(args.nr):
    if args.index:
        prefix = '{0}\t'.format(i + 1)
    else:
        prefix = ''
    print('{0}{1}'.format(prefix, random.gauss(args.mu, args.sigma)))
```

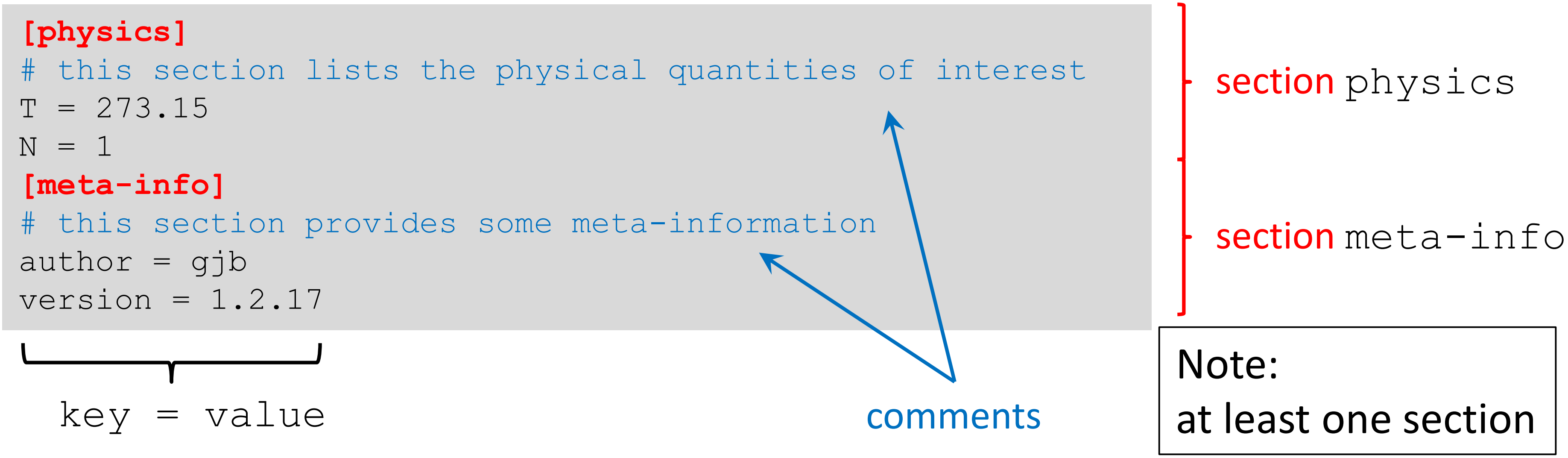
```
$ ./generate_gaussians -h
usage: generate_gaussians.py [-h] [-mu MU] [-sigma SIGMA] [-idx] [n]
Gaussian random number generator
positional arguments:
  n                number of random numbers to generate
optional arguments:
  -h, --help       show this help message and exit
  -mu MU           mean of distribution
  -sigma SIGMA     stddev of distribution
  -idx             print index for random number
```

Autogenerated help message

```
$ ./generate_gaussians -idx 3.0
usage: generate_gaussians.py [-h] [-mu MU] [-sigma SIGMA] [-idx] [n]
generate_gaussians.py: error: argument n: invalid int value: '3.0'
```

ConfigParser configuration files

- Configuration files
  - save typing of options
  - Document runs of applications
- Easy to use from Python: configparser module
- Configuration file (e.g., '`test.conf`')



```
[physics]
# this section lists the physical quantities of interest
T = 273.15
N = 1
[meta-info]
# this section provides some meta-information
author = gjb
version = 1.2.17
```

section `physics`

section `meta-info`

`key = value`

comments

Note:
at least one section

Reading & using configurations

- Reading configuration file

```
from configparser import ConfigParser
cfg = ConfigParser()
cfg.read('test.conf')
```

- Using configuration values

```
temperature = cfg.getfloat('physics', 'T')
number_of_runs = cfg.getint('physics', 'N')
version_str = cfg.get('meta-info', 'version')
if cfg.has_option('physics', 'g'):
    acceleration = cfg.getfloat('physics', 'g')
else:
    acceleration = 9.81
```

Further reading: argparse

- Argparse tutorial https://docs.python.org/3/howto/argparse.html

Code Pack 17

- See the files:

1.argparse

2.configparser

Coding Bootcamp Code in Python

PROFILING

If you don't profile…

Profiling approaches

- Microbenchmarking, i.e., timing functions
  - Easy
  - Can lead to premature optimization
    = waste of time
- Profiling with profiling tool
  - Slightly more complicated
  - Identifies true bottlenecks

Both are useful, when used appropriately

Timing functions

- ipython: use magic `%time` or `%timeit`

multiple runs

```
In [1]: from primes import primes
In [2]: %timeit result = primes(1000)
10 loops, best of 3: 172 ms per loop
```

timing result

- Command line: use `timeit` module

module to use

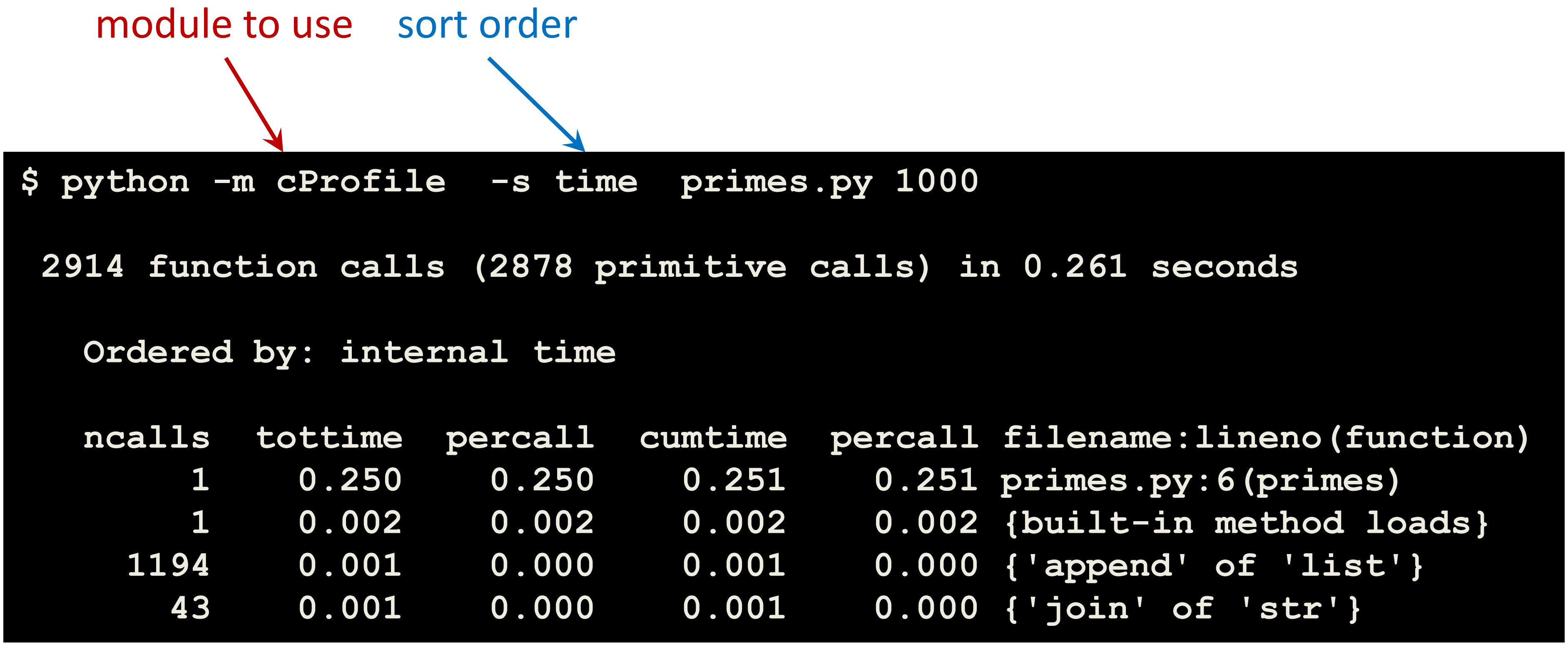
statements to execute, string per line

```
$ python -m timeit 'from primes import primes' 'primes(1000)'
10 loops, best of 3: 174 msec per loop
```

Don't forget indentation!

Profiler

- Use the `cProfile` module



module to use

sort order

```
$ python -m cProfile  -s time  primes.py 1000

 2914 function calls (2878 primitive calls) in 0.261 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.250    0.250    0.251    0.251 primes.py:6(primes)
        1    0.002    0.002    0.002    0.002 {built-in method loads}
     1194    0.001    0.000    0.001    0.000 {'append' of 'list'}
       43    0.001    0.000    0.001    0.000 {'join' of 'str'}
```

Visual profiles: snakeviz

- Use the `cProfile` module

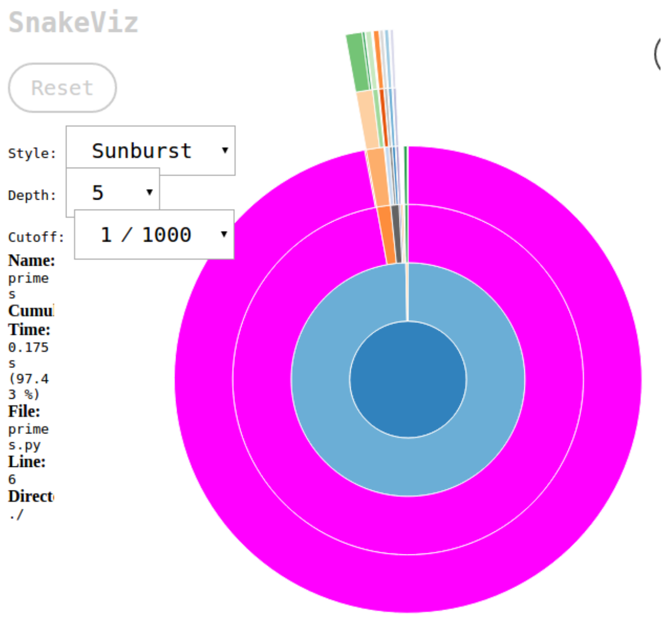
module to use

sort order

output file

```
$ python -m cProfile  -s time  -o primes.prof \
      primes.py 1000
```

```
$ snakeviz  primes.prof
```



Style: Sunburst

Depth: 5

Cutoff: 1 / 1000

Name: primes

Cumu Time: 0.175 s (97.43 %)

File: primes.py

Line: 6

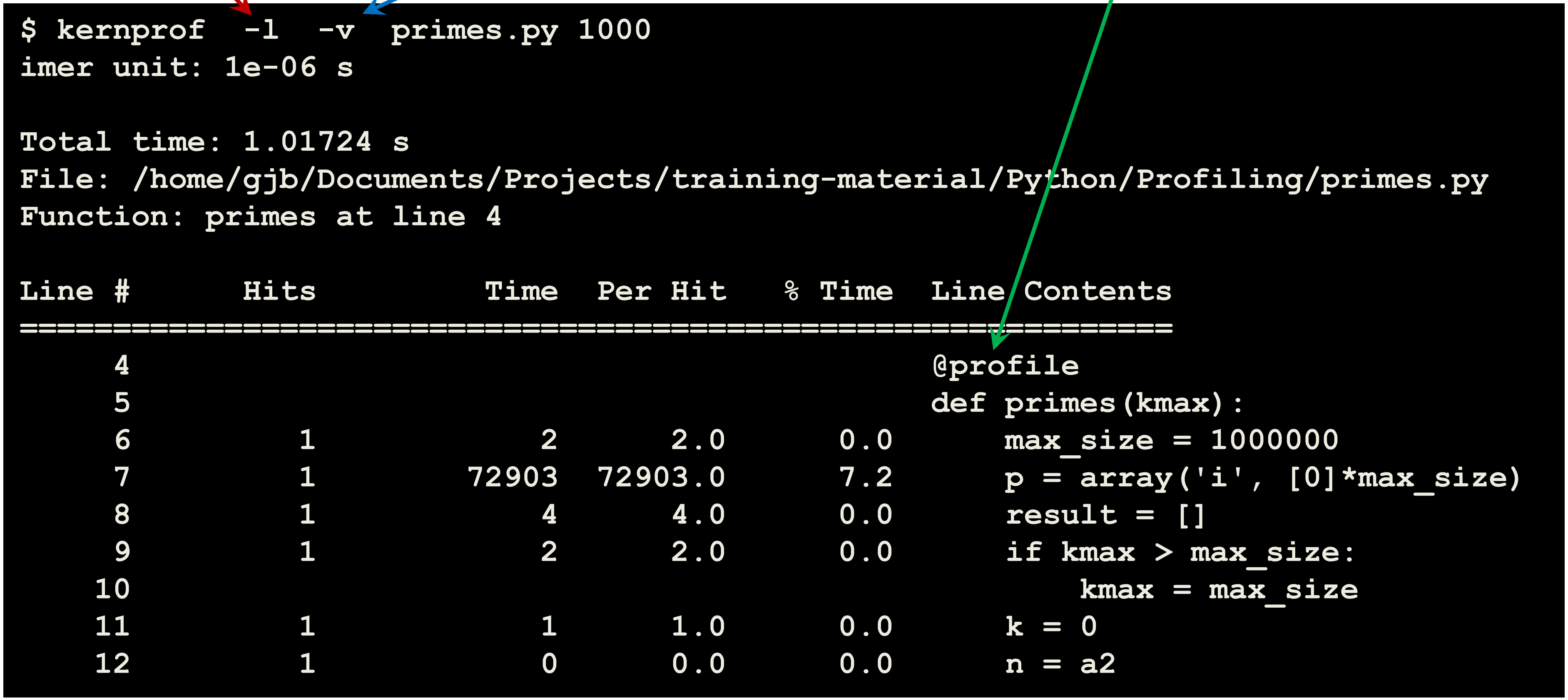
Direct ./

line_profiler

line by line

show profile on screen

decorate function to profile



```
$ kernprof  -l  -v  primes.py 1000
imer unit: 1e-06 s

Total time: 1.01724 s
File: /home/gjb/Documents/Projects/training-material/Python/Profiling/primes.py
Function: primes at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def primes(kmax):
     6         1            2      2.0      0.0      max_size = 1000000
     7         1        72903  72903.0      7.2      p = array('i', [0]*max_size)
     8         1            4      4.0      0.0      result = []
     9         1            2      2.0      0.0      if kmax > max_size:
    10                                                   kmax = max_size
    11         1            1      1.0      0.0      k = 0
    12         1            0      0.0      0.0      n = a2
```

Code Pack 18

- See the files:

1.Benchmarking

2.Profiling_Your_Code_with_cProfile

Coding Bootcamp Code in Python

LOGGING

Logging: motivation

- Useful to verify what an application does
  - in normal runs
  - in runs with problems
- Helps with debugging
  - alternative to print statements
- Various levels can be turned on or off
  - see only relevant output

Good practice

Initialize & configure logging

```
import logging
…
logging.basicConfig(level=level, filename=name, filemode=mode,
                    format=format_str)
…
```

- level: minimal level written to log
- filemode
  - 'w': overwrite if log exists
  - 'a': append if log exists
- format, e.g.,
  `'{asctime}:{levelname:{message}'`

Log levels

- `CRITICAL`: non-recoverable errors
- `ERROR`: error, but application can continue
- `WARNING`: potential problems
- `INFO`: feedback, verbose mode
- `DEBUG`: useful for developer

- User defined

Selecting log level

- CRITICAL
- ERROR
- WARNING
- INFO
- DEBUG

```
level = logging.ERROR
```

Log messages

- Log to `DEBUG` level

```
logging.debug('function xyz called with "{0}"'.format(x))
```

ignored at level `INFO` or above

- Log to `INFO` level

```
logging.info('application started')
```

ignored at level `WARNING` or above

- Log to `CRITICAL` level

```
logging.critical('input file not found')
```

Logging destinations

- File
- Rotating files
- syslog
- …

Further reading: logging

- Logging how-to
  https://docs.python.org/3/howto/logging.html
- Logging Cookbook
  https://docs.python.org/3/howto/logging-cookbook.html

Code Pack 19

- See the files:

Logging