

---

# MINI-COURSE ON CODE DEVELOPMENT AND PACKAGING

---





---

# TESTING YOUR CODE

---



---

# BASICS OF GOOD TESTING

---

- Presumably you test whether each piece of code that you write works in some way, but you probably
    - only run it when developing the code and then assume all is fine
    - don't write the tests as a set of functions, but rather run them in a Python terminal / `jupyter` notebook
    - don't exhaustively test how new parts of your code work together with older parts of your code
  - This means that your code is very vulnerable to big and small issues that come up, making it hard to use and maintain
-



---

# BASICS OF GOOD TESTING

---

- Better to use a *test suite*:
    - A set of Python functions (or classes) with checks on your code's functionality
    - Becomes part of your code repository, so it can be saved and changes tracked, and applied to future changes to the code
    - Consists of *unit tests* and *integration tests*
  - If designed well, can be run with standard commands, keep track of *test coverage*, and be run automatically by online services every time you update your code
-



---

# EXAMPLE

---

```
def test_square_direct():  
    # Direct test that the square works based on known solutions  
    import math  
    import exampy  
    tol = 1e-10  
    assert math.fabs(exampy.square(1.)-1.) < tol, \  
        "exampy.square does not agree with known solution"  
    assert math.fabs(exampy.square(2.)-4.) < tol, \  
        "exampy.square does not agree with known solution"  
    assert math.fabs(exampy.square(3.)-10.) < tol, \  
        "exampy.square does not agree with known solution"  
    return None
```



---

# UNIT AND INTEGRATION TESTS

---

- *Unit tests:*

- Test whether a small unit of your code works as expected:
- Break your code into smallest unit (e.g., function) that makes sense and build more complex functionality from these smallest units
- Unit tests check that each of the smallest units does what it is supposed to do

- *Integration tests:*

- Even if each unit works as expected, they may not work together as they should
  - Integration tests check that different units of your code work together well and give correct results
-



---

# GOOD QUALITIES OF A TEST SUITE

---

- Should be as minimal, short, and atomic as possible: Keep tests as simple as you can and still achieve a useful test
  - Should run in as little time as possible: One you add many tests, the time the test suite takes to run will get long...
  - Should test expected outputs, but also errors and warnings raised: If you are raising exceptions or warnings upon certain behavior, test that that is correctly done as well
  - Should test setting non-default keywords for functions and methods to make sure that works as expected
  - Should test different invocations of functions: don't just test one, perhaps special case, but make sure the function works for different cases of the inputs
  - Should be well-commented!
-



---

# WHAT ARE GOOD THINGS TO TEST?

---

- Context specific!
  - Can check known answers: often we write code to solve problems that can only be solved analytically in certain special cases. Test that the special cases work.
  - Can check known properties of the answer: even if we don't know any solution, we may require that the solution has certain properties (e.g., follows a conservation law or symmetry). Test that known properties work.
  - Can check against alternative ways of getting the answer:
    - Alternative method that only applies in certain cases (but not analytic)
    - Alternative code implementation, e.g., in an external package
-



---

# WRITING A TEST SUITE

---



---

# WHERE DOES MY TEST SUITE GO?

---

- Inside your package:
    - Include the tests in the package itself and distribute them with the code
    - Advantage: users can easily run the test suite on their own machine and convince themselves that the code works for them
    - Disadvantages: Adds a lot of code to your package that is not necessary for it to work, typically not well-documented, and not tested itself (who tests the tests?)
    - Not typical to get a problem when code works as expected on one machine and it and its dependencies install fine on the user's machine
  - Not recommended
-



---

# WHERE DOES MY TEST SUITE GO?

---

- Outside your package:
  - Include the tests in a sub-directory of your top-level directory, outside of the package itself

```
TOP-LEVEL_DIRECTORY/  
  docs/  
  exampy/  
  tests/  
  README.md  
  setup.py
```

- This way tests are part of your package's `git` repository without being part of your package's distribution
    - Advantage: Your tests can depend on hard-to-install dependencies, as long as you can get them to install, without having to worry about user complaints
  - My recommendation
-



---

# WHAT DOES MY TEST SUITE LOOK LIKE?

---

- We will be using `pytest` to run the tests
- `pytest` automatically detects tests, provided that
  - Files start with `test_` and end in `.py`
  - Functions that are tests start with `test`, classes start with `Test`

```
TOP-LEVEL_DIRECTORY/  
docs/  
  build/  
  source/  
  Makefile  
  make.bat  
exampy/  
  integrate/  
  __init__.py  
  _math.py  
tests/  
  test_basic_math.py  
README.md  
setup.py
```



---

# WHAT DO MY TESTS LOOK LIKE?

---

- A test statement is a simple `assert` statement
- These are the *only* statements that make up formal tests, don't do something like `if res < tol: print("Didn't work")`
- Add a message to display when the `assert` fails

```
def test_square_direct():  
    # Direct test that the square works based on known solutions  
    import math  
    import exampy  
    tol = 1e-10  
    assert math.fabs(exampy.square(1.)-1.) < tol, \  
        "exampy.square does not agree with known solution"  
    assert math.fabs(exampy.square(2.)-4.) < tol, \  
        "exampy.square does not agree with known solution"  
    assert math.fabs(exampy.square(3.)-10.) < tol, \  
        "exampy.square does not agree with known solution"  
    return None
```



# EXAMPLE: KNOWN VALUE

```
>>> import test_basic_math
>>> test_basic_math.test_square_direct()
```

```
AssertionError: exampy.square does not agree with known solution
```

```
def test_square_direct():
    # Direct test that the square works based on known solutions
    import math
    import exampy
    tol = 1e-10
    assert math.fabs(exampy.square(1.)-1.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(2.)-4.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(3.)-10.) < tol, \
        "exampy.square does not agree with known solution"
    return None
```



---

# EXAMPLE: SYMMETRY PROPERTY

---

```
def test_cube_oddfunction():  
    # Test of the cube function by checking that it is an odd function  
    tol= 1e-10  
    assert math.fabs(exampy.cube(1.)+exampy.cube(-1.)) < tol, \  
        "exampy.cube is not an odd function"  
    assert math.fabs(exampy.cube(2.)+exampy.cube(-2.)) < tol, \  
        "exampy.cube is not an odd function"  
    assert math.fabs(exampy.cube(3.)+exampy.cube(-3.)) < tol, \  
        "exampy.cube is not an odd function"  
    return None
```

```
>>> import test_basic_math  
>>> test_basic_math.test_cube_oddfunction()
```



---

# OR

---

```
def test_cube_oddfunction():  
    # Test of the cube function by checking that it is an odd function  
    tol= 1e-10  
    for nn in range(1,10):  
        assert math.fabs(exampy.cube(nn)+exampy.cube(-nn)) < tol, \  
            "exampy.cube is not an odd function"  
    return None
```



---

# EXAMPLE: TEST AGAINST ALTERNATIVE

---

```
def test_simps_against_scipy():  
    # Test that exampy.integrate.simps integration agrees with  
    # scipy.integrate.quad  
    from scipy import integrate as sc_integrate  
    complicated_func= lambda x: x*np.cos(x**2)/(1+np.exp(-x))  
    tol= 1e-14  
    n_int= 1000  
    assert np.fabs(exampy.integrate.simps(complicated_func,0,1,n=n_int)  
                  -sc_integrate.quad(complicated_func,0,1)[0])\  
    < tol, \  
    """exampy.integrate.simps gives a different result from ""\  
    ""scipy.integrate.quad for a complicated function""  
  
    return None
```



---

# RUNNING A TEST SUITE WITH `pytest`

---



---

# RUNNING A TEST SUITE WITH `pytest`

---

- `pytest` is the preferred test runner for Python code
  - Automatically detects your tests (see before), prints overview of what happens
  - Many options for running, skipping, verbosity of output, etc.
  - + additional functionality for testing errors and warnings, labeling known failures, etc.
-



# pytest EXAMPLE

```
pytest -v tests/test_basic_math.py
```

in a regular terminal in the top-level of the package. This produces output that looks like

```
===== test session starts =====
platform darwin -- Python 3.7.3, pytest-5.1.0, py-1.8.0, pluggy-0.12.0 -- /PATH/
TO/PYTHON/BINARY
cachedir: .pytest_cache
rootdir: /PATH/TO/exampy
plugins: arraydiff-0.3, doctestplus-0.3.0, openfiles-0.4.0, remotedata-0.3.1
collected 2 items

tests/test_basic_math.py::test_square_direct PASSED [ 50%]
tests/test_basic_math.py::test_cube_oddfunction PASSED [100%]

===== 2 passed in 0.07s =====
```



```

===== test session starts =====
platform darwin -- Python 3.7.3, pytest-5.1.0, py-1.8.0, pluggy-0.12.0 -- /PATH/
TO/PYTHON/BINARY
cachedir: .pytest_cache
rootdir: /PATH/TO/exampy
plugins: arraydiff-0.3, doctestplus-0.3.0, openfiles-0.4.0, remotedata-0.3.1
collected 2 items

tests/test_basic_math.py::test_square_direct FAILED [ 50%]
tests/test_basic_math.py::test_cube_oddfunction PASSED [100%]

===== FAILURES =====
_____ test_square_direct _____

def test_square_direct():
    # Direct test that the square works based on known solutions
    tol = 1e-10
    assert math.fabs(exampy.square(1.)-1.) < tol, \
        "exampy.square does not agree with known solution"
    assert math.fabs(exampy.square(2.)-4.) < tol, \
        "exampy.square does not agree with known solution"
> assert math.fabs(exampy.square(3.)-10.) < tol, \
    "exampy.square does not agree with known solution"
E AssertionError: exampy.square does not agree with known solution
E assert 1.0 < 1e-10
E + where 1.0 = <built-in function fabs>((9.0 - 10.0))
E + where <built-in function fabs> = math.fabs
E + and 9.0 = <function square at 0x1080fad90>(3.0)
E + where <function square at 0x1080fad90> = exampy.square

tests/test_basic_math.py:12: AssertionError
===== 1 failed, 1 passed in 0.10s =====

```



---

# USEFUL `pytest` OPTIONS

---

- `-x`: Exit upon the first failure (default is to run all tests)
  - `-s`: Print `stdout` and `stderr` outputs (default is to not print these)
  - `-k EXPRESSION`: Only run tests with `EXPRESSION` in the their name
  - `--lf`: only run the last-failed test
  - `--disable-pytest-warnings`: don't print all warnings (as a summary at the end)
-



---

# TESTING ERRORS

---

- You can test whether your code correctly raises an exception using `pytest.raises`

```
def test_simps_typerror():  
    # Test that exampy.integrate.simps properly raises a TypeError  
    # when called with a non-array function  
    import math  
    import pytest  
    with pytest.raises(TypeError):  
        out= exampy.integrate.simps(lambda x: math.exp(x),0,1)  
    return None
```

---



---

# TESTING ERRORS

---

- You can test whether your code correctly raises an exception using `pytest.raises`
- You can test the entire error string as well:

```
def test_simps_typerror():  
    # Test that exampy.integrate.simps properly raises a TypeError  
    # when called with a non-array function  
    import math  
    import pytest  
    with pytest.raises(TypeError) as excinfo:  
        out= exampy.integrate.simps(lambda x: math.exp(x),0,1)  
    assert str(excinfo.value) == "Provided func needs to be callable on arrays of inputs"  
    return None
```

---



---

# TEST COVERAGE

---



---

# WHAT IS TEST COVERAGE?

---

- Once you have a test suite, you will wonder “how much of my code is actually used when running the test suite”
  - This question has different answers depending on what you mean:
    - *Function coverage*: what fraction of functions is used by the test suite —> should aim for 100%
    - *Statement coverage*: what fraction of statements is used by the test suite —> aim for 100%, can be difficult to get because of edge cases
    - *Branch coverage*: when my code branches, does the test suite cover all possibilities (if ... then... else...)?
    - *Condition coverage*: for complex conditional statements, does the test suite produce True/False for each boolean sub-expression? ( if  $x > 0$  and  $y < 0$  )
-



---

# coverage.py: MEASURING TEST COVERAGE IN YOUR CODE

---

- coverage.py is a Python package that will report the test coverage of your test suite, most easily *statement coverage*

- Simply run your test suite as

```
coverage run -m pytest ...
```

- instead of

```
pytest ...
```

- This collects the coverage info, but does not yet display it
-



---

# `coverage.py`: MEASURING TEST COVERAGE IN YOUR CODE

---

- Once collected, you can display the results in different ways
    - `coverage report: text report`
    - `coverage html: HTML output`
-



# EXAMPLE REPORT

```
coverage run -m pytest -v tests/  
coverage report
```

we get

Name	Stmts	Miss	Cover	Missing
exampy/___init__.py	1	0	100%	
exampy/_math.py	9	2	78%	73, 88
exampy/integrate/___init__.py	2	0	100%	
exampy/integrate/_integrate.py	8	0	100%	
tests/test_basic_math.py	13	0	100%	
tests/test_integrate.py	27	1	96%	52
TOTAL	60	3	95%	



# USE `--SOURCE` TO SPECIFY THE PACKAGE

```
coverage run --source=exampy/ -m pytest -v tests/  
coverage report
```

we now get

Name	Stmts	Miss	Cover	Missing
exampy/___init__.py	1	0	100%	
exampy/_math.py	9	2	78%	73, 88
exampy/integrate/___init__.py	2	0	100%	
exampy/integrate/_integrate.py	8	0	100%	
TOTAL	20	2	90%	



# EXAMPLE HTML REPORT

Coverage report: 90%



<i>Module</i> ↓	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
exampy/___init__.py	1	0	0	100%
exampy/_math.py	9	2	0	78%
exampy/integrate/___init__.py	2	0	0	100%
exampy/integrate/_integrate.py	8	0	0	100%
<b>Total</b>	<b>20</b>	<b>2</b>	<b>0</b>	<b>90%</b>

coverage.py v5.0.3, created at 2020-03-06 15:40



---

# EXCLUDING CODE FROM TEST COVERAGE STATISTICS

---

- Sometimes you want to exclude some parts of your code from the test-coverage statistics
    - Lines that you don't think have to be tested (use sparingly!)
    - Lines that cannot be executed by the test suite
      - `if False:`
      - `if __name__ == '__main__':`
  - For a single line, use `# pragma: no cover`
-



---

# EXCLUDING CODE FROM TEST COVERAGE STATISTICS

---

- More complex patterns, use `.coveragerc` in the directory where you run the tests

```
[run]
source= exampy/

[report]
# Regexes for lines to exclude from consideration
exclude_lines =
    # Have to re-enable the standard pragma
    pragma: no cover

    # Don't complain if tests don't hit defensive assertion code:
    raise AssertionError
    raise NotImplementedError

    # Don't complain if non-runnable code isn't run:
    if 0:
    if __name__ == __main__.:

omit =
    exampy/__init__.py
    exampy/integrate/*

ignore_errors = True

[html]
directory = coverage_html_report
```



---

# CONTINUOUS INTEGRATION

---



---

# WHAT IS CONTINUOUS INTEGRATION

---

- Refers to running ‘integration tests’ on a regular basis, at high cadence
  - ‘Integration tests’ in this context is the combination of building your package and running the test suite, making sure that all parts of the code package work as expected (incl. installation)
  - Nowadays largely done by online services whenever:
    - You push a commit or set of commits to GitHub for any branch
    - Somebody opens or updates a pull request
  - Try to catch changes to the code (easy) and to dependencies (harder) that may cause your code’s installation or tests to fail
  - Many services available, focus on Travis CI here, see notes for AppVeyor for Windows integration (and we will talk about GitHub actions later)
-



---

# CONTINUOUS INTEGRATION WITH `Travis CI`

---

- Runs on <https://travis-ci.com/>
  - Sign up with your GitHub account, select repository you want to add
  - Configure using a `.travis.yml` file in your code repository's top-level directory
  - Then any push to GitHub triggers a `Travis CI` build
-



---

# A SIMPLE `.travis.yml` FILE

---

```
language: python

python:
  - "3.7"

install:
  - python setup.py develop

script:
  - echo 0
```





Search all repositories

My Repositories **Running (0/0)** +

There are no jobs running

There are no jobs queued

# jobovy / exampy build unknown

**Current** Branches Build History Pull Requests

More options 

✓ **master** Update travis to add a trivial script

↔ #2 passed

Restart build

- ↔ Commit 2c41a55 
- ↔ Compare acfb6be...2c41a55 
- ↔ Branch master 


⌚ Ran for 17 sec  
 📅 3 minutes ago


 Jo Bovy

 Python: 3.7  
 AMD64

[Job log](#)

[View config](#)

 Remove log

 Raw log

```

1
117 Build system information
118 Build language: python
119 Build group: stable
120 Build dist: xenial
121 Build id: 152239139
122 Job id: 295565151

```

0.06s  
0.05s



---

# A `.travis.yml` FILE THAT RUNS THE TESTS

---

```
language: python
```

```
python:
```

```
- "3.7"
```

```
install:
```

- pip install pytest
- pip install pytest-cov
- pip install scipy
- python setup.py develop

```
script:
```

```
- pytest -v tests/ --cov=exampy/
```

---



```

▶ 172 $ pip install pytest
▶ 181 $ pip install pytest-cov
▶ 197 $ pip install scipy
▶ 204 $ python setup.py develop
227 $ pytest -v tests/ --cov=exampy/
228 ===== test session starts =====
229 platform linux -- Python 3.7.1, pytest-4.3.1, py-1.7.0, pluggy-0.8.0 -- /home/travis/virtualenv/python3.7.1/bin/python
230 cachedir: .pytest_cache
231 rootdir: /home/travis/build/jobovy/exampy, inifile:
232 plugins: cov-2.8.1
233 collected 6 items
234
235 tests/test_basic_math.py::test_square_direct PASSED [ 16%]
236 tests/test_basic_math.py::test_cube_oddfunction PASSED [ 33%]
237 tests/test_integrate.py::test_simps_against_riemann PASSED [ 50%]
238 tests/test_integrate.py::test_simps_against_scipy PASSED [ 66%]
239 tests/test_integrate.py::test_simps_typererror PASSED [ 83%]
240 tests/test_integrate.py::test_simps_scalarfunc XFAIL [100%]
241
242 ===== warnings summary =====
243 exampy/integrate/_integrate.py:56
244 /home/travis/build/jobovy/exampy/exampy/integrate/_integrate.py:56: DeprecationWarning: invalid escape sequence \l
245     """
246
247 -- Docs: https://docs.pytest.org/en/latest/warnings.html
248
249 ----- coverage: platform linux, python 3.7.1-final-0 -----
250 Name                               Stmts  Miss  Cover
251 -----
252 exampy/__init__.py                   1      0  100%
253 exampy/_math.py                      9      2   78%
254 exampy/integrate/__init__.py         2      0  100%
255 exampy/integrate/_integrate.py       8      0  100%
256 -----
257 TOTAL                               20      2   90%
258
259 ===== 5 passed, 1 xfailed, 1 warnings in 0.52 seconds =====
260 The command "pytest -v tests/ --cov=exampy/" exited with 0.
261
262
263 Done. Your build exited with 0.

```



---

# THE `.travis.yml` FILE

---

- Sections that set up the environment and what Travis CI runs: `language:`, `python:`, `env:` for defining environment variables, `matrix:` for build matrices (see later), `addons:`, `services:`
  - Things to run before the main installation: `before_install:`
  - Commands to perform the installation: `install:`
  - Commands to run before you run the tests: `before_script:`
  - The tests: `script:`
  - What to do if the tests were successful: `after_success`
  - `notifications:`
-



```
language: python
```

```
python:
```

```
- "3.7"
```

```
env:
```

```
- NUMPY_VERSION=1.18
```

```
  SCIPY_VERSION=1.4
```

```
before_install:
```

```
- pip install numpy==$NUMPY_VERSION
```

```
install:
```

```
- python setup.py develop
```

```
before_script:
```

```
- pip install pytest
```

```
- pip install pytest-cov
```

```
- pip install scipy==$SCIPY_VERSION
```

```
script:
```

```
- pytest -v tests/ --cov=exampy/
```



---

# BUILD MATRICES

---

- One of the great advantages of online continuous integration services is that it is easy to test your code for different versions of your dependencies
  - Travis CI has lots of support for creating build matrices: combinations of different version that all get built and tested
  - Created automatically by combining different options in sections such as `python:` and `env:`
  - Can also manually adjust the matrix in the `matrix:` section, e.g, `include:` additional jobs or `exclude:` some jobs from the matrix
-



```
language: python
```

```
python:
```

- "3.8"
- "3.7"

```
env:
```

- NUMPY\_VERSION=1.18
- NUMPY\_VERSION=1.17
- NUMPY\_VERSION=1.16

```
before_install:
```

- pip install numpy==\$NUMPY\_VERSION

```
install:
```

- python setup.py develop

```
before_script:
```

- pip install pytest
- pip install pytest-cov
- pip install scipy

```
script:
```

- pytest -v tests/ --cov=exampy/





Search all repositories

My Repositories **Running (6/6)** +

- ✓ jobovy/exampy # 8.1  
⌚ Duration: 37 sec
- 🔴 jobovy/exampy # 8.2  
⌚ Duration: 1 min 11 sec
- 🔴 jobovy/exampy # 8.3  
⌚ Duration: 1 min 9 sec
- ✓ jobovy/exampy # 8.4  
⌚ Duration: 27 sec
- ✓ jobovy/exampy # 8.5  
⌚ Duration: 25 sec
- ✓ jobovy/exampy # 8.6  
⌚ Duration: 27 sec

There are no jobs queued

# jobovy / exampy build passing

**Current** [Branches](#) [Build History](#) [Pull Requests](#)

More options

**master** Make build matrix with two Python versions and three numpy v... #8 started Cancel build

Commit 810868b [↗](#) Running for 1 min 11 sec

Compare 7979d6a...810868b [↗](#)

Branch master [↗](#)

Jo Bovy

Build jobs [View config](#)

✓ # 8.1	AMD64	Python: 3.8	NUMPY_VERSION=1.18	⌚ 37 sec	⊙
🔴 # 8.2	AMD64	Python: 3.8	NUMPY_VERSION=1.17	⌚ 1 min 11 sec	⊗
🔴 # 8.3	AMD64	Python: 3.8	NUMPY_VERSION=1.16	⌚ 1 min 9 sec	⊗
✓ # 8.4	AMD64	Python: 3.7	NUMPY_VERSION=1.18	⌚ 27 sec	⊙
✓ # 8.5	AMD64	Python: 3.7	NUMPY_VERSION=1.17	⌚ 25 sec	⊙
✓ # 8.6	AMD64	Python: 3.7	NUMPY_VERSION=1.16	⌚ 27 sec	⊙



---

# USING Miniconda

---

- Installing many dependencies can quickly lead to long build times
  - You can install Miniconda and use that to install everything in your Python installation that you need
  - See the notes for info on how to do this
-