Contributing to IC

testing

Why bother writing tests?

Scenarios:
1. I have a written code that I need to test (most of the time)
2. I know what my code should do, I can write tests to ensure the desired behavior **before** writing the code itself (test-driven development) (fixing bugs)

Why tests:
- Avoids code being broken by someone else
- Ensures that an optimized version of the code leads to the same results
- Makes sure your results are reproducible

Structuring a test

- Make sure the code runs
- If a code in some limit behaves as already build-in math/numpy… function test that the result is the same
- Check that conserved values are indeed conserved (energy, mass, momentum...)
- If you are not changing mutable inputs of your code, check that they stay unchanged
- If you are rewriting/optimizing working code, produce test (or test file) with the working version to ensure both behave the same
- If your code is non-deterministic (statistical distributions etc) expect test to fail reasonable amount of times (flaky test)
- Expose your test to all kind of weird or extreme inputs (hypothesis helps here)

pytest

- A framework for easy unit testing (allows you to write and execute tests easily)
- Install via pip
  $ pip install -U pytest # it is already installed in IC environment
- pytest command searches and runs the test as:
  - With no arguments then it starts from the current directory
  - Recurse into directories and search for test_*.py or *_test.py files
  - Collects “test”-prefixed functions or methods outside of class
- The layout type we use in IC is:

```
/my_project
        __init__.py
        setup.py
        sub_folder/
                __init__.py
                one_func.py
                other_func.py
                one_func_test.py
                other_func_test.py
```

*Note __init__.py files, it allows Python to understand that one_func is part of my_project module. You need them for relative imports.

pytest call

- To run pytest suite call:
  $ pytest or $ python -m pytest

- To execute tests within a specific directory:
  $ pytest testing/

- To execute the tests within a specific module:
  $ pytest test_module.py

- To execute a specific test:
  $ pytest test_mod.py::test_func

- To execute a collection of tests that share part of their name
  $ pytest -k test_myfunc
  # runs test_myfunc_0, test_myfunc_1, test_myfunc_whatever

  Exercise: Go to IC your local directory and run all the tests for esmeralda, and test_penthesilea_KrMC in penthesilea tests.

pytest basics

To write a test just name it “test_” in the name of the function.
If the function runs without interruptions that means the test has passed.
You should see a green light at the end.

Exercise: complete first exercise in pytest_tutorial.py

pytest fixtures

Fixtures are functions that produce an input for a test. It can be configured to hold the result for different scopes (avoids repeating the code).

```
@pytest.fixture
def input_value():
   input = 39
   return input

def test_divisible_by_3(input_value):
   assert input_value % 3 == 0

def test_divisible_by_6(input_value):
   assert input_value % 6 == 0 # this will fail
```

scope is a keyword option in pytest.fixture decorator. Mostly used ones are:
- fixture(scope='module') runs fixture once per module (file) and
- fixture(scope='session') runs a fixture once per session

pytest fixtures

- If you use the same fixture for multiple test files use conftest.py to store it, no need to import it, gets automatically detected by pytest

- tmpdir fixture provides a temporary directory unique to each test

- tmpdir_factory is a session-scoped fixture that allows you to set temporary directory for the whole session

Exercise: complete second exercise in pytest_tutorial.py

pytest mark

Marks can be added to your test as a decorator (ie @pytest.mark.xxx). Some are:

- **skip** - don't run this test
- **xfail** – allow test to fail.

Flaky is a module that allows you to mark tests that occasionally fail (ie statistical tests):

@flaky(max_runs=3, min_passes=2)

- **dependency** (depends=[tests_to run_before])
- **parametrize** repeats the test for a set of different inputs:

@mark.parametrize("threshold", (4, 10))

Exercise: complete third exercise in pytest_tutorial.py

hypothesis

Is a module that allows property-based testing

- User:
  - Describes valid inputs
  - Writes a test that passes for any valid input
- Engine:
  - Generates many test cases
  - Runs your test for each input
  - Reports minimal failing inputs (usually)

Can be installed via pip (already included in IC environment)

hypothesis

You will probably only use:

- strategies – an engine to draw parameters to feed in test (integers, floats, lists etc)
- given – a decorator for the test to be able to accept strategies output as arguments

```
from hypothesis            import given
from hypothesis.strategies import integers

@given(integers())
def test_even_integers(i):
    assert type(i) == int
```

hypothesis

Other modules:

- settings (you can check some of them in IC/conftest.py)
- note – to print results within the test function
- example – draw example from strategies (useful when making composite strategies)
- assume – if you don't want examples with certain condition to be used

```
from hypothesis            import given
from hypothesis            import assume
from hypothesis.strategies import floats

@given(floats())
def test_positive_floats(i):
    assume num > 0
    assert num > 0
```

For details about strategies look
https://hypothesis.readthedocs.io/en/latest/data.html

Exercise: complete Exercises 4-6

hypothesis

Sometimes you need to adapt the provided strategies. It can be done via :

- Mapping : applies given function to produced examples
- Filtering : reject examples that don’t satisfy the condition
- Composite : lets you combine strategies in arbitrary ways.

```
from hypothesis import composite
from hypothesis.strategies import lists

  @composite
  def list_and_index(draw, elements=integers()):
     xs = draw(lists(elements, min_size=1))
     i = draw(integers(min_value=0, max_value=len(xs) - 1))
     return xs, i
```

Exercise: complete Exercises 7-10

Some basics profiling

Profiling time can be done with **cProfile**.

The easiest way is to have a python script (.py) file and run:
$ python -m cProfile python_script.py args
This will print the output to the console. You can also use
$ python -m cProfile -o output.prof python_script.py args
that writes in pickled non-readable form.
To recover the text output use pstats:

```
import sys
import pstats
sys.stdout = open('readable.profile', 'w')
p = pstats.Stats('esmeralda.prof')
p.print_stats()
```

To be able to graphically see prof outputs use **SnakeViz** (https://jiffyclub.github.io/snakeviz/) (pip install snakeviz)
To run it:
$ snakeviz output.prof

Exercise: run esmeralda profiling using esmeralda.py script (also examine it to see what it does), saving it to some output, install snakeviz and run it. What are the most time consuming functions?

Some basics profiling

Memory profile can be done with **memprofile** (`pip install -U memory_profiler`)

```
mprof run <executable>
mprof plot
```

Exercise: examine memory consumption of esmeralda.py script

Some basics profiling

Another interesting tool is **coverage** (https://coverage.readthedocs.io/en/coverage-5.0.3/) : it gives report of all the function being called by a given script (or test module)

To install:
$ pip install coverage

And use:
$ coverage run script.py arg1 arg2…
or as a part of pytest module
$ coverage run -m pytest

To produce a report and html files:
$ coverage report -m
$ coverage html

Exercise: Install coverage and produce html files for our esmeralda.py script. Checkout the html output.