HPC PYTHON – TESTING AND DEBUGGING

M. BELHORN (OLCF) , W. SCULLIN (ALCF), R. THOMAS (NERSC)

ECP Annual Meeting 2018
Knoxville, TN
2018-02-05

OUTLINE

- Basic Code Quality
  - Zen
  - Style conventions
  - Linting
- Unit Testing
  - Write meaningful tests
  - Mocking objects
- Online Debugging
  - Using `pbd`

To run all the demos and examples on `core.nersc.gov` it is necessary to setup your environment with

```
module load python/3.6-anaconda-4.4  # includes mpi4py, numpy, pylint...
pip install --user pytest-cov mock
```

THE BASICS

In [1]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

FULL AND ACCURATE TESTS ARE CRUCIAL

Python is **fantasticly simple and easy** thanks to features like

- dynamic typing
- support for mixed programming paradigms

...but these features also leave Python vulnerable to **catastrophic runtime failure**.

The best way to ensure your code runs successfully is to cover it with meaningful tests.

Before we can get to that, however, there are some basic code quality considerations that must be met.

YOU ALL ADHERE TO *PEP8* (HTTPS://WWW.PYTHON.ORG/DEV/PEPS/PEP-0008/), RIGHT?

Consistently following community conventions goes a long way to avoiding and solving bugs.

For items not covered by *PEP8*, I recommend following the *Google Python Style Guide* (https://google.github.io/styleguide/pyguide.html).

LINT YOUR CODE

Running your code through a linter like `pylint` prior to release

- ensures PEP8 compliance
- identifies logic errors
- identifies syntax errors
- may catch environment errors

before they become runtime failures.

PER-PROJECT CONFIGURATION

- package your source code with an empty/minimal linter configuration

```
$ pylint --generate-rcfile | less  # these are the defaults
$ touch .pylintrc
```

- enhance the defaults only when necessary.
- For instance, `pylint` may need permission to introspect some C extension modules.

```
[MASTER]
extension-pkg-whitelist=mpi4py,numpy

[TYPECHECK]
ignored-modules=numpy
```

Most exceptions to the linter configuration should be done through pragmas *within the source code* because exceptions

```
# This exception is unjustified - the linter is
# trying to tell us this can be improved.
HOSTFILE = 'login1 h41n10 h41n10 h41n10 h41n10 h41n10 h41n10 h41n10 h41n1
0 h41n10 h41n10 h41n10 h41n11 h41n11 h41n11 h41n11 h41n11 h41n11 h41n11 h
41n11'  #noqa pylint: disable=line-too-long
```

In [ ]:

```python
#!/usr/bin/env python3
'''
A contrived example of why linting is important.
'''

from mpi4py import MPI


def main():
    '''Print the MPI parameters for this rank.'''
    comm = MPI.COMM_WORLD
    size = comm.Get_size()
    rank = comm.Get_rank()
    name = MPI.Get_processor_name()

    padding = len(str(size))
    print("Greetings from rank '{1:0{0}d}' of '{2}' on '{3}'.".format(
        padding, rank, size, name))


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

Human eyes probably cannot see the error, but the linter does:

```
$ pylint mpi_hello_world.py
Using config file /home/matt/documents/jupyter/ecp/.pylintrc
************* Module mpi_hello_world
W: 17,10: Invalid format string (bad-format-string)

-------------------------------------------------------------------
Your code has been rated at 9.00/10 (previous run: 9.00/10, +0.00)
```

Here, the ： in the format string is actually a UTF8 look-alike glyph. While the error is still ambiguous, the exception that would have occured at runtime is more perplexing:

```
In [5]: print('rank_{1：0{0}d}'.format(0, 1))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-32f7d0225634> in <module>()
----> 1 print('rank_{1：0{0}d}'.format(0, 1))

ValueError: unexpected '{' in field name
```

UNIT TESTING

WHAT'S A TEST?

A set of functions that cover every possible execuction path of your code base and assert the behavior is always correct.

```
def rotate_operator(matrix):
    '''Prepends 90 degree rotation to linear transformation `matrix`.'''
    return matrix.dot([[0, -1], [1, 0]])


def test_rotate_operator():
    matrix = np.mat([[0, 1], [2, 3]])
    ouput = transform_matrix(matrix)
    assert isinstance(output, np.matrix)
    assert (output == np.mat([[1, 0], [3, -2]])).all()
```

Good practice to regularly test critical areas of your code as it is developed, e.g. continuous integration.

FRAMEWORKS: PYTEST VS UNITTTEST

The standard library contains `unittest`, but I recommend `pytest`. Requires installation:

```
pip install pytest
pytest tests
```

but

- less boilerplate in tests
- less learning curve, uses basic `python` syntax and elements
- shorter development time and rapid implementation
- extensible with many plugins, such as for *coverage*, *doctests*, etc.

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

WRITING TESTS

- Pytest searches ***modules*** in the target directory for module names
  - starting with `test_`
  - or ending with `_test`
- Within identified test modules, pytest treats *functions with the same prefixes as tests.*
- Test modules can be separate from source code or side-by-side

```
./examples/
|-- demo/
|   |-- __init__.py
|   |-- demo.py
|   |-- test_demo.py
|-- example/
|   |-- __init__.py
|   |-- example.py
|   |-- tests/
|   |   |-- test_example.py
```

TESTS SHOULD BE MEANINGFUL

As long as the test function exits without raising an exception, the test is a passing success.

```
def test_tait_bryan_rotation():
    vector, angles, expected_vector = ...  # test fixtures
    output = rotate_yxz_tait_bryan(vector, angles)
    assert output is not None  # alone, this is probably useless
    assert output == expected_vector  # better
```

Every runtime error or bug that occurs should have a test written or updated with the code base patch to prevent regression.

TOTAL FLEXIBILITY IN MAKING ASSERTIONS

unittest:

```
self.assertEqual(vector, expected_vector)
self.assertIn(element, expected_vector)
self.assertAlmostEqual(vector, expected_vector)
```

vs

pytest:

```
assert vector == expected_vector
assert element in expected_vector
assert almost_equal(vector, expected_vector)
```

```
def vectors_almost_equal(vector_a, vector_b, tolerance=0.01):
    '''Compare vector_b is nearly equal to vector_a.'''
    # Explicit understanding of "almost" is better than implicit.
    zipped = zip(vector_a, vector_b)
    approx = pytest.approx  # Ignore diffences near machine precision.
    return all((a == approx(b, tolerance) for a, b in zipped))
```

RUNNING TESTS

All `pytest` needs is the path to search for test modules (default `cwd`). If keeping tests separate from source, run `pytest` from top-level package directory to ensure source modules are in the `sys.path`:

```
$ pytest tests
========================== test session starts ==========================
platform linux -- Python 3.6.4, pytest-3.3.2, py-1.5.0, pluggy-0.6.0
rootdir: /home/matt/development/ecp_hpc_python/demos/testing, inifile:
plugins: mock-1.6.3, flake8-0.9.1, cov-2.5.1
collected 4 items

tests/test_ipc.py ..                                                [ 50%]
tests/test_linalg.py .x                                             [100%]

================== 3 passed, 1 xfailed in 0.25 seconds ==================
```

Any `pytest-<X>` plugin modules in the python path will be automatically enabled and can be controlled via arguments to `pytest`. For example, to include coverage reporting with `pytest-cov`:

```
$ pytest --cov
======================== test session starts ========================
platform linux -- Python 3.6.4, pytest-3.3.2, py-1.5.0, pluggy-0.6.0
rootdir: /home/matt/development/ecp_hpc_python/demos/testing, inifile:
plugins: mock-1.6.3, flake8-0.9.1, cov-2.5.1
collected 4 items

tests/test_ipc.py ..                                                [ 50%]
tests/test_linalg.py .s                                             [100%]

---------- coverage: platform linux, python 3.6.4-final-0 ----------
Name                    Stmts   Miss  Cover
-----------------------------------------
__init__.py                 0      0   100%
ipc.py                     37      8    78%
linalg.py                   8      3    62%
tests/__init__.py           0      0   100%
tests/test_ipc.py          21      0   100%
tests/test_linalg.py       12      2    83%
-----------------------------------------
TOTAL                      78     13    83%


================ 3 passed, 1 skipped in 0.27 seconds ================
```

Specific tests can be run by passing a specific test module path (including file suffix).

- All tests in the module will be run by default
- Specific test functions can be run by passing a full *node id*

```
$ pytest tests/test_linalg.py::test_fail_catastrophically
======================== test session starts ========================
platform linux -- Python 3.6.4, pytest-3.3.2, py-1.5.0, pluggy-0.6.0
rootdir: /home/matt/development/ecp_hpc_python/demo_testing, inifile:
plugins: mock-1.6.3, flake8-0.9.1, cov-2.5.1
collected 1 item

tests/test_linalg.py F                                          [100%]

============================== FAILURES ==============================
_____________________ test_fail_catastrophically _____________________

    def test_fail_catastrophically():
        '''Tests a failing function.'''
>       output = fail_catastrophically()

tests/test_linalg.py:27:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
linalg.py:21: in fail_catastrophically
    return rotate_operator(matrix)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

matrix = [[0, 1], [2, 3]]

    def rotate_operator(matrix):
        '''Prepends 90 degree rotation to linear transformation `matrix
`.'''
>       return matrix.dot([[0, -1], [1, 0]])
E       AttributeError: 'list' object has no attribute 'dot'

linalg.py:10: AttributeError
====================== 1 failed in 0.14 seconds ======================
```

FIXTURES AND MOCK OBJECTS

Static functional tests are fine, but what about everything else?

The hard part is dependency injection.
How do you deal with communication layers like MPI?

This isn't a problem for python web app developers testing code against

- DBs
- wsgi servers
- REST webhooks, etc.

HPC dependencies are not, in principle, more complex than these.
The solution is to provide your tests with fixtures and Mock dependency objects.

FIXTURES

Re-usable models, inputs, frameworks, and dependencies that your tests rely on.

Simply functions that are declared to `pytest` as fixtures with a decorator.

When tests that are written to take arguments, `pytest` searches for a fixture (function) of the same name and uses the return value as the argument value.

Some fixtures are built-in to pytest and do not need to be imported or present in your code at all.

Fixtures can accept other fixtures - allows for flexible re-use of common test dependencies.

It is even possible to test fixtures.

```
import pytest
import numpy as np

  @pytest.fixture
  def fixture_matrix():
      '''A fixture matrix that can be re-used in multiple tests.'''
      return np.mat([[0, 1], [2, 3]])


  def test_rotate_operator(fixture_matrix):
      '''Tests the `rotate_operator` function.'''
      output = rotate_operator(fixture_matrix)
      assert isinstance(output, np.matrix)
      assert (output == np.mat([[1, 0], [3, -2]])).all()
```

MOCK OBJECTS

The should be no expectation of manually replicating the API structure and return values for all of an external service's methods. We can do this with `mock` which can emulate the interface of an external package in a safe way.

Mock methods and attributes can be controlled to return exactly what is expected.

Mocked objects keep track of how and with what arguments they are called, often all that matters in tests invoking external packages.

```
Mock.assert_any_call()
Mock.assert_called()
Mock.assert_called_once()
Mock.assert_called_once_with()
Mock.assert_called_with()
Mock.assert_has_calls()
Mock.assert_not_called()
Mock.call_args  # Last call arguments
Mock.call_args_list  # List of call args over all calls
Mock.called  # Was it called?
Mock.call_count  # How many times?
Mock.method_calls  # What methods were called?
Mock.return_value  # Force the return value when called.
```

```
import pytest
from mpi4py import MPI
from mock import Mock
import ipc

# pylint: disable=redefined-outer-name

@pytest.fixture
def mock_mpi():
    '''A mocked MPI module.'''
    mpi = Mock(spec=MPI)
    mpi.COMM_WORLD = Mock(spec=MPI.COMM_WORLD)
    return mpi

def test_mock_mpi(mock_mpi):
    '''Tests the interface to mocked MPI.'''
    print(dir(mock_mpi.COMM_WORLD))
    assert hasattr(mock_mpi.COMM_WORLD, 'rank')
    assert hasattr(mock_mpi.COMM_WORLD, 'Allgather')
```

MONKEYPATCHING

When code under test depends on global scope objects and singletons, use the built-in pytest fixture `monkeypatch` to replace the dependency with a mocked object while the test is run. The changes exist only for the specific test function when it is called.

```
def test_demo_scatter_gather(monkeypatch, mock_mpi):
      '''Tests the ipc.demo_scatter_gather function.'''
      monkeypatch.setattr('ipc.MPI', mock_mpi)
      mock_mpi.COMM_WORLD.size = 2
      mock_mpi.COMM_WORLD.rank = 1
      ipc.demo_scatter_gather()
      mock_mpi.COMM_WORLD.Scatter.assert_called()
      mock_mpi.COMM_WORLD.Allgather.assert_called()
```

ONLINE DEBUGGING

Admit it, we've all done this:

```
def buggy_function(questionable_input, **kwargs):
    '''No, please, stop.'''
    # ...
    print(questionable_input)
    # ...
    return questionable_output
```

There's got to be a better way!

THE BUILT-IN DEBUGGER PBD

- Interactive shell to explore runtime state, branches
- Enter debugging shell at
  - end of execution
  - first unhandled exception
  - at hardcoded break points
- Lets you
  - introspect/alter object states
  - execute arbitrary python commands
  - Step through lines or functions individually
  - Continue running up to specific function calls

Some caveats when used in an HPC environment.

GETTING STARTED

The pdb module is part of the standard library.
There are several ways to get from runtime execution into a pdb shell:

RUN APPLICATION DIRECTLY UNDER THE DEBUGGER

```
$ python -m pdb [-c continue] mpi_hello_world.py
> /home/matt/documents/jupyter/ecp/mpi_hello_world.py(4)<module>()
-> '''
(Pdb)
```

VIA INTERACTIVE PYTHON SESSION

Start pdb and connect it to the `sys.last_traceback`.

```
$ mpiexec -n 2 python -i mpi_hello_world_pm.py
>>> Traceback (most recent call last):
Traceback (most recent call last):
  File "mpi_hello_world_pm.py", line 22, in <module>
    main()
  File "mpi_hello_world_pm.py", line 18, in main
    print(template.format(padding, rank, size, name))
ValueError: unexpected '{' in field name
>>> import pdb; pdb.pm()
> /home/matt/documents/ecp/hpc_python/mpi_hello_world_pm.py(18)main()
-> print(template.format(padding, rank, size, name))
(Pdb) print(rank, size)
0 2
(Pdb) print(template)
Greetings from rank '{1:0{0}d}' of '{2}' on '{3}'.
(Pdb) ':' in template
False
(Pdb)
```

Running `python -i` directly under MPI tends to break readline, breaking history and arrow key support in `pdb`.

The `stdin` is typically only directed to rank 0. If in the likely event the exception occurs on a different rank, this is not too helpful. In some cases, if the rank where failure occurs is known (see next slide), it **may be** possible to attach to `pdb` there with variations of

```
mpiexec -stdin 1 -n 2 python -i mpi_scatter_gather.py
srun -N 1 -n 2 -i 1 python -i mpi_scatter_gather.py
```

Although many bespoke HPC application launchers and configurations prevent reliable `stdin` redirection. Try to debug on smaller resources before scaling up. An alternative **HPC-unfriendly** workaround is to launch a terminal under MPI that invokes the python command:

```
mpiexec -n 2 xterm -e "python -i mpi_scatter_gather.py"
```

Again, not typically possible to connect to terminals launched this way on many HPC resources.

Discover what node raises an exception during a catastrophic crash:

```
from mpi4py import MPI

#...

if __name__ == '__main__':
    try:
        main()
    except Exception as err:
        print('Failure occured on rank '%s' % MPI.COMM_WORLD.rank')
        raise err
```

INVOKE SHELL FROM WITHIN THE APPLICATION

```
# Hard-code a breakpoint where you want to start debugging.
import pdb; pbd.set_trace()
```

- Usually only possible to interact with pdb on one rank (0) - entering pdb from all ranks will hang at first MPI blocking operation. If using MPI, only connect to debugger on one rank:

```
if comm.rank == 0:
    import pdb; pdb.set_trace()  # noqa pylint: disable=multiple-statemen
ts
```

or obtain a interactive terminal for each rank (again, not practical on most HPC resources).

OTHER METHODS

- Run specific code blocks with `pdb.run()`
- Enter `pdb` from failed unit tests `pytest -x --pdb ...`

GETTING OUR BEARINGS

- `(Pdb) help`
- `[l]ist`, `ll/longlist`: List code around current step frame
- `[w]here`: Show location of current frame in the stack
- `display <expression>`: show evaluation of expression at each step
- `source <object>`: Show (if possible) the source code of `<object>`

MOVING ABOUT THE CODE

- `[s]tep`: Execute into next function call or line
- `[r]eturn`: Execute to return of current function
- `[n]ext`: Execute to next line of instructions, stepping over function calls
- `[unt]il`: Execute to next greatest line number in source (step out of loops)
- `[cont]inue`: Execute to next breakpoint
- `run`, `restart`: Restart the debugged program, possibly with new `sys.argv`

MOVING ABOUT THE STACK FRAME (FUNCTION SCOPE)

- `[u]p`: Execute until location moves up the frame stack
- `[d]own`: Execute until location moves down the frame stack

INTROSPECTING RUNTIME STATE

- [a]rgs: Print argument names and values of current function call frame
- p, pp:Print/pretty-print objects in the stack

BREAKPOINTS

- `[b]reak`: List or set breakpoints at line, function, condition
- `condition`: Set/remove conditions that must be met to honor breakpoint
- `[cl]ear`: Remove breakpoints
- `commands`: Set list of commands to be run when encountering breakpoint
- `disable`/`enable`: Disable/enables set of breakpoints

```
(Pdb) break 23
Breakpoint 1 at /home/matt/documents/jupyter/ecp/mpi_scatter_gather.py:23
(Pdb) break 49
Breakpoint 2 at /home/matt/documents/jupyter/ecp/mpi_scatter_gather.py:49

(Pdb) break show_data_state
Breakpoint 3 at /home/matt/documents/jupyter/ecp/mpi_scatter_gather.py:17
(Pdb) break rank_print, local_data[-1] > 0 and rank == 2
Breakpoint 4 at /home/matt/documents/jupyter/ecp/mpi_scatter_gather.py:11
(Pdb) break
Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /home/matt/documents/jupyter/ecp/mpi_scatt
er_gather.py:23
2   breakpoint   keep yes   at /home/matt/documents/jupyter/ecp/mpi_scatt
er_gather.py:49
3   breakpoint   keep yes   at /home/matt/documents/jupyter/ecp/mpi_scatt
er_gather.py:17
4   breakpoint   keep yes   at /home/matt/documents/jupyter/ecp/mpi_scatt
er_gather.py:11
        stop only if local_data[-1] > 0 and rank == 2
```

FEEL FREE TO PLAY WITH THE DEMO CODE ON CORI

Basic login and setup:

```
ssh $TRAINACCT@cori.nersc.gov
salloc -N 1 -q regular -t 240 -C haswell -A "ntrain" --reservation="ecp_p
ython" -L SCRATCH
module load python/3.6-anaconda-4.4
pip install --user pytest-cov mock
git clone https://gitlab.com/matt.belhorn/ecp_hpc_python.git
```

Running MPI example entering pdb:

```
cd $HOME/ecp_hpc_python/demos
srun -n 2 -c 32 python launch_pdb_via_trace.py
```

Running unit tests:

```
cd $HOME/ecp_hpc_python/demos/testing
pytest --cov tests
```

QUESTIONS, COMMENTS?