Early experience on cloud microphysics modeling with Python

Davide Del Vento[1], Dorota Jarecka[2,1], Sylwester Arabas[2]

1. National Center for Atmospheric Research, Boulder, Colorado; 2. Faculty of physics, University of Warsaw, Warsaw, Poland

Synopsis

Python was developed with emphasis on code readability and as an easy-to-learn language. It is a widely used, general-purpose language, with many scientific-oriented libraries and offers great tools for unit, functional and regression tests. There are libraries that allow for relatively easy bindings between Python and other languages such as C/Fortran (CFFI, F2Py) and C++ (Boost.Python).
This poster presents experiences of working on cloud microphysical schemes in Python. A WRF microphysical scheme is used as an example of Fortran code that can be called from Python, tested and compared with other schemes. All presented codes are available at https://bitbucket.org/djarecka/ams2015_poster.

Using CFFI to call Fortran code from Python

Calling Fortran code from Python:

- enables to reuse existing Fortan code
- enables to use Python environment for:
  - * developing new code with testing driven techniques (TDD)
  - * trying quickly new ideas in Python (no compilation, ipython notebook, etc.)
  - * writing regression and performance tests with little effort

CFFI - C Foreign Function Interface for Python

- enables to call compiled C and Fortran code from Python
- enables to call Python code from C and Fortran using callback mechanism
- attempts to support both PyPy and CPython
- has different trade-offs compared to alternatives such as f2py

Example of calling a WRF microphysical scheme from python

```
import numpy as np
from constants_kessler import xlv, cp, EP2, SVP1, SVP2, SVP3, SVPT0, rhowater
# use cffi library - a C Foreign Function Interface for Python
from cffi import FFI
ffi = FFI()

# define a python function - binding a C function
def kessler(nx, ny, nz, dt_in, t_np, qv_np, qc_np, qr_np,
            rho_np, pii_np, dz8w_np, z_np, rainnc_np, rainncv_np):
    # provide a signature for the C function
    ffi.cdef("void c_kessler(double t[], double qv[], double qc[], double qr[], double rho[], \
double pii[], double dt_in, double z[], double xlv, double cp, double EP2, double SVP1, double\
 SVP2, double SVP3, double SVPT0, double rhowater, double dz8w[], double RAINNC[], double RAIN\
NCV[], int ids, int ide, int jds, int jde, int kds, int kde, int ims, int ime, int jms, int jm\
e, int kms, int kme, int its, int ite, int jts, int jte, int kts, int kt);", override=True)

    # load a library with the C function
    lib = ffi.dlopen('libkessler.so')

    # create cdata variables of a type "double *" for each numpy array
    # the cdata variables will be passed to the C function and can be changed
    t       = ffi.cast("double*", t_np.__array_interface__['data'][0])
    qv      = ffi.cast("double*", qv_np.__array_interface__['data'][0])
    qc      = ffi.cast("double*", qc_np.__array_interface__['data'][0])
    qr      = ffi.cast("double*", qr_np.__array_interface__['data'][0])
    rho     = ffi.cast("double*", rho_np.__array_interface__['data'][0])
    pii     = ffi.cast("double*", pii_np.__array_interface__['data'][0])
    dz8w    = ffi.cast("double*", dz8w_np.__array_interface__['data'][0])
    z       = ffi.cast("double*", z_np.__array_interface__['data'][0])
    RAINNC  = ffi.cast("double*", rainnc_np.__array_interface__['data'][0])
    RAINNCV = ffi.cast("double*", rainncv_np.__array_interface__['data'][0])
    # create additional variables that will be passed to the C function as values
    [ims, ime, ids, ide, its, ite] = [1, nx] * 3
    [jms, jme, jds, jde, jts, jte] = [1, ny] * 3
    [kms, kme, kds, kde, kts, kte] = [1, nz] * 3

    # call the C function
    lib.c_kessler(t, qv, qc, qr, rho, pii, dt_in, z, xlv, cp, EP2, SVP1, SVP2,
                  SVP3, SVPT0, rhowater, dz8w, RAINNC, RAINNCV, ids, ide, jds, jde,
                  kds, kde, ims, ime, jms, jme, kms, kme, its, ite, jts, jte, kts, kte)
```

Passing to CFFI the C signature of the Fortran function

Creating CFFI objects storing pointers to first elements of NumPy arrays

Calling the Fortran function with pointers to NumPy array data as arguments

Using py.test for testing microphysical schemes

Testing and Python

- tests rise confidence that your code works properly
- tests allow you to make changes faster and reliable
- TDD and/or unit tests - checking if basic blocks work properly
- integration tests - checking if various blocks fit together properly
- functional and/or sanity tests - checking if the basic output makes sense
- regression tests - increasing confidence that the results are as expected and changes to the code do not produce unexpected output
- Python offers variety of testing frameworks: unittest, nose, pytest

Pytest - useful features:

- simple structure of test code - easy to learn
- compatible with nose and unittest
- same framework for unit, integration, functional and regression tests

Example of testing microphysical scheme using pytest

```
import numpy as np
# use pytest library for testing
import pytest

# function takes the initial values of pressure, temperature and mixing ratios
# returns water vapour and cloud water mixing ratios after condensation/evaporation
def condensation(lib, rv, rc, dt = 1, press = np.array([900.e2]),
                 T = np.array([283.15]), rr = np.array([0.])):
    # import library for testing, lib is specified during calling
    import importlib
    lib_adj = importlib.import_module(lib)
    # calling a function from chosen library
    rv, rc, rr = lib_adj.adj_cellwise(press, T, rv, rc, rr, dt)
    return rv, rc

# check if the function returns values close to expected values
# various sets of arguments are tested
@pytest.mark.parametrize("arg, expected", [
                  # no cloud water and supersaturation
                  ({"rv" : np.array([10.e-3]),  "rc" : np.array([0.])},
                   {"rv" : np.array([9.44e-3]), "rc" : np.array([.56e-3])}),
                  # subsaturation leads to some evaporation
                  ({"rv" : np.array([8.e-3]),   "rc" : np.array([1.e-3])},
                   {"rv" : np.array([8.26e-3]), "rc" : np.array([0.74e-3])}),
                  # supersaturation leads to condensation
pytest.mark.xfail(({"rv" : np.array([9.e-3]),   "rc" : np.array([1.e-3])},{
                    "rv" : np.array([8.85e-3]), "rc" : np.array([1.15e-3])})),
                  ])
def test_expected_output_evapcond(libname, arg, expected, epsilon = 0.1):
    rv, rc = condensation(lib=libname, **arg)
    for key, value in expected.items():
        assert abs(eval(key) - value) <= epsilon * abs(value)
```

Tested function can be from various libraries

Parametrization of test function - wide range of parameters can be tested at the same time

skip and xfail - easy dealing with tests that are expected to fail

```
# enable to choose tested library during calling
def pytest_addoption(parser):
    parser.addoption("--libname", action="append", default=[],
        help="name of the tested library")

def pytest_generate_tests(metafunc):
    if 'libname' in metafunc.fixturenames:
        metafunc.parametrize("libname", metafunc.config.option.libname)
```

Content of conftest.py file:
pytest_generate_tests hook - enables to generate a set of tests depending on command lines (e.q., allows for testing various libraries)

References

Arabas, Jarecka, Jaruga & Fijalkowski, 2014, Sci. Prog. (available at http://arxiv.org/abs/1301.1334)
http://www.wrf-model.org/index.php
https://cffi.readthedocs.org/en/release-0.8/#
https://pytest.org/latest/
http://pythontesting.net/start-here/

Acknowledgement

DJ acknowledges support from the Polish Ministry of Science and Higher Education (project no. 1119/MOB/13/2014/0).

MNiSW