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 (CFI, 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 Fortran 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 matplotlib.pyplot
from numpy import arange, pi
from constants import *
from pylab import *

import cffi

ffi = cffi.FFI()

ffi.cdef('double function(double a, double b);')

lib = ffi.dlopen('lib.so')

r = lib.function(1.0, 2.0)
```

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

generate_tests.py

# import 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(qv, rc, ET, qm):
    qv = np.array(qv[0:31], RR = np.array(rr[0:31]))
    # import library for testing, lib is specified during calling

import matplotlib.pyplot
lib = np.importlib.import_module('condensation_lib')

calling a function from python library
rv, rr = lib.adired(qv, rc, ET)
```

Content of conf/test.py file:
```

def test gev
    f = read_value()
    a = 1.0
    b = 2.0
    assert f == pytest.approx(a + b)
```

Acknowledgement

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