PRACE Training Centre @ SURFsara

Carlos Teijeiro Barjas

SURFsara

SURFsara

- Created in 1971 as a collaboration between CWI, UvA and VU
- Provides integrated ICT infrastructure for research (data storage, visualization, networking, cloud and supercomputing)
- Host of Dutch national supercomputers since 1984
- Partner of the PRACE project

Partnership for Advanced Computing in Europe (PRACE)

- Enable high impact scientific discovery
- Engineering research and development across all disciplines
- Enhance European competitiveness for the benefit of society
- Established as an international not-for-profit association with seat in Brussels
- Collaboration between 26 member countries whose representative organizations create a pan-European supercomputing infrastructure
- Extensive education and training effort: seasonal schools, workshops…
- Currently at the Fifth Implementation Phase (PRACE-5IP)

PRACE Training Centres (PTCs)

- BSC - Barcelona Supercomputing Center (Spain)
- CSC - IT Center for Science (Finland)
- CINECA - Consorzio Interuniversitario (Italy)
- EPCC at the University of Edinburgh (UK)
- GCS - Gauss Supercomputing Center (Germany)
- GRNET - Greek Research and Technology Network (Greece)
- ICHEC - Irish Centre for High-End Computing (Ireland)
- IT4I - IT4Innovations National Supercomputing Center (Czech Republic)
- MdlS - Maison de la Simulation (France)
- SURFsara (The Netherlands)

PRACE Training Centre at SURFsara

- Organization of training workshops from 1 to 3 days
- All events are organized in the Netherlands (Amsterdam/Utrecht)
- Support for research and development institutions in the Netherlands
- All trainings and materials are provided in English
- All information can be found in the in the PRACE Training Portal

http://www.training.prace-ri.eu/

Presentation of the course: Parallel and GPU Programming in Python

Carlos Teijeiro Barjas

SURFsara

Timetable for the 10th December

- 09:00 - 09:15: Welcome & Introduction
- 09:15 - 10:30: Best practices in Scientific Computing & Python
- 10:30 - 10:45: Coffee break
- 10:45 - 12:00: Introduction to efficient shared memory programming
- 12:00 - 13:00: Lunch
- 13:00 - 14:30: Hands-on: Introduction to efficient Python CPU programming
- 14:30 - 14:45: Coffee break
- 14:45 - 15:30: Shared Memory Programming in Python: Numba, Cython and OpenMP
- 15:30 - 15:45: Coffee break
- 15:45 - 17:15: Hands-on: Numba, Cython

Timetable for the 11th December

- 09:00 - 10:30: Introduction to the GPU ecosystem
- 10:30 - 10:45: Coffee break
- 10:45 - 12:00: Hands-on: Programming GPUs with Numba
- 12:00 - 13:00: Lunch
- 13:00 - 14:30: Hands-on: Programming GPUs with PyCUDA
- 14:30 - 14:45: Coffee break
- 14:45 - 15:30: Distributed Memory Architecture & MPI
- 15:30 - 15:45: Coffee break
- 15:45 - 16:55: Hands-on: Introduction to mpi4py
- 16:55 - 17:00: Closing session

Best practices in Scientific Computing & Python

Carlos Teijeiro Barjas

SURFsara

Outline

- General best practices in scientific programming
- Useful tools to start your project
- Python Enhancement Proposals (PEPs)
- Some language conventions

Outline

- General best practices in scientific programming
- Useful tools to start your project
- Python Enhancement Proposals (PEPs)
- Some language conventions

Four simple recommendations to encourage best practices in research software:

- https://f1000research.com/articles/6-876/v1

Write programs for people, not computers

- A program should not require its readers to hold more than a handful of facts in memory at once.
- Make names consistent, distinctive, and meaningful.
- Make code style and formatting consistent.

Let the computer do the work

- Make the computer repeat tasks.
- Save recent commands in a file for re-use.
- Use a build tool to automate workflows.

Make incremental changes

- Work in small steps with frequent feedback and course correction.
- Use a version control system.
- Put everything that has been created manually in version control.

Don’t repeat yourself (or others)

- Every piece of data must have a single authoritative representation in the system.
- Modularize code rather than copying and pasting.
- Re-use code instead of rewriting it.

Plan for mistakes

- Add assertions to programs to check their operation.
- Use an off-the-shelf unit testing library.
- Turn bugs into test cases.
- Use a symbolic debugger.

Optimize software only after it works correctly

- Use a profiler to identify bottlenecks.
- Write code in the highest-level language possible.

Document design and purpose, not mechanics

- Document interfaces and reasons, not implementations.
- Refactor code in preference to explaining how it works.
- Embed the documentation for a piece of software in that software.

Collaborate

- Use pre-merge code reviews.
- Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
- Use an issue tracking tool.

Outline

- General best practices in scientific programming
- Useful tools to start your project
- Python Enhancement Proposals (PEPs)
- Some language conventions

Some tools may help building a project from scratch…

- Cookiecutter
- Pytest
- Documentation: Doxygen and Sphinx

Outline

- General best practices in scientific programming
- Useful tools to start your project
- Python Enhancement Proposals (PEPs)
- Some language conventions

PEP 20: the Zen of Python…… import this!

PEP 8: Style Guide for Python Code

Main source of information: https://www.python.org/dev/peps/pep-0008/

Improving the readability of code: consistency!

- Consistency with the style guide is important.
- Consistency within a project is more important.
- Consistency within one module or function is the most important.

“A Foolish Consistency is the Hobgoblin of Little Minds”

Improving the readability of code: consistency!

- Consistency with the style guide is important.
- Consistency within a project is more important.
- Consistency within one module or function is the most important.

“A Foolish Consistency is the Hobgoblin of Little Minds”

Code Layout – Indentation & Line Breaks

- Spaces are the preferred indentation method (4 spaces per level)
- Limit all lines to a maximum of 79 characters

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# More indentation included to distinguish this from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

Code Layout – Indentation & Line Breaks

- Should a line break before or after a binary operator?
- → For old code, consistency is the key
- → For new code, it is recommended to break the line before the operator

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Code Layout – Blank Lines

- Surround top-level function and class definitions with two blank lines.
- Method definitions inside a class are surrounded by a single blank line.

```
from setuptools import setup
from setuptools.command.test import test as TestCommand


class PyTest(TestCommand):
    user_options = [('pytest-args=', 'a', "Args list")]

    def initialize_options(self):
        TestCommand.initialize_options(self)
        self.pytest_args = []
```

Code Layout – Imports

- Imports should be on separate lines.

Yes:

```
import os
import sys
```

No:

```
import sys, os
```

- It's okay to say this though:

```
from subprocess import Popen, PIPE
```

Code Layout – Imports

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.

- Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

- Wildcard imports (`from <module> import *`) should be avoided

Code Layout – Comments

- Comments that contradict the code are worse than no comments.
- Always make a priority of keeping the comments up-to-date when the code changes!
- Comments should be complete sentences.
- “Python coders from non-English speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.” (!!! ...)

Code Layout – Comments

- Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code.
- Each line of a block comment starts with a # and a single space.

```
# This is a typical comment for a Python code.
# It continues in the next line.
```

- Use inline comments sparingly: in fact they are distracting if they state the obvious.

Avoid this:

```
x = x + 1   # Increment x
```

But this can be useful:

```
x = x + 1   # Compensate for border
```

Code Layout – Comments

- A documentation string (docstring) is a string literal that can be included as the first statement of the definition of a class, function, method or module
- The conventions for docstrings are described in PEP 257
- The closing characters of a multiline docstring (""") should be on a line by itself, except for one liner docstrings

```
"""Return a foobang.

Optional plotz says to frobnicate the bizbaz first.
"""
```

Code Layout – Comments

- A combination of docstrings and block comments is useful in order to provide a description of the function and notes for programmers together

```
# The following function represents a performance bottleneck
def heavy_computation(x, y, z)
"""Perform some really heavy computation"""
```

- The leading comment block is a programmer's note, whereas the docstring describes the operation of the function or class and will be shown in an interactive Python session when the user types:

```
>>> help(heavy_computation)
```

Self-documenting Code – Naming

- Variable, class or function names should speak for themselves.

```
decay()
decay_constant()
get_decay_constant()

p = 100
pressure = 100
```

- Moreover, the naming convention should be consistent

```
var, VAR, _var, var_, MyVar, myVar, my_var, MY_VAR
```

Self-documenting Code – Simple Functions

- Functions must be small to be understandable: they should do only one clear thing.

```
import numpy as np


def initial_cond(N, Dim):
    """Generates initial conditions for N unity masses at rest
       starting at random positions in D-dimensional space.
    """
    position0 = np.random.rand(N, Dim)
    velocity0 = np.zeros((N, Dim), dtype=float)
    mass = np.ones(N, dtype=float)
    return position0, velocity0, mass
```

Outline

- General best practices in scientific programming
- Useful tools to start your project
- Python Enhancement Proposals (PEPs)
- Some language conventions

Pitfalls that should be avoided

- Multiple and messy circular dependencies
- Hidden coupling: modifying code in one class should never break tests in unrelated test cases
- Heavy use of global state or context
- Spaghetti code: multiple pages of nested if clauses and for loops with a lot of copy-pasted procedural code and no proper segmentation
- Ravioli code: hundreds of similar little pieces of logic without proper structure

Dynamic Typing

- Avoid using the same variable name for different things
- Good practice: assign a variable only once
- Check your code: Pylint, Pyflakes, Flakes8, Pychecker

Bad

```
a = 1
a = 'a string'
def a():
    pass # Do something

items = 'a b c d'
items = items.split(' ')
items = set(items)
```

Good

```
count = 1
msg = 'a string'
def func():
    pass # Do something

items_string = 'a b c d'
items_list = items_string.split(' ')
items = set(items_list)
```

Alternatives to checking for equality

Bad

```
# Checking for True
if attr == True:
    print('True!')

# Checking for None
if attr == None:
    print('attr is None!')
```

Good

```
# Just check the value
if attr:
    print('attr is truthy!')

# or check for the opposite
if not attr:
    print('attr is falsey!')

# or, since None is
# considered false,
# explicitly check for it
if attr is None:
    print('attr is None!')
```

Accessing dictionary elements

Bad (also removed in Python 3.x)

```
d = {'hello': 'world'}
if d.has_key('hello'):
    print(d['hello'])
else:
    print('default_value')
```

Good

```
d = {'hello': 'world'}
print(d.get('hello', 'default_value'))
print(d.get('thingy', 'default_value'))
# Or:
if 'hello' in d:
    print(d['hello'])
```

Looping over dictionaries

```
d = {'matthew': 'blue', 'rachel': 'green','raymond': 'red'}
```

Bad (not working in Python 3.x anymore)

```
for k in d:
    print(k)

for k in d.keys():
    if k.startswith('r'):
        del d[k]
```

Good

```
d = {k: d[k] for k in d if not
k.startswith('r')}
print(d)
```

Manipulating lists

Bad

```
# Filter elements greater than 4
a = [3, 4, 5]
b = []
for i in a:
    if i > 4:
        b.append(i)
```

Good

```
# List comprehension
a = [3, 4, 5]
b = [i for i in a if i > 4]

# Or:
b = filter(lambda x: x > 4, a)
```

Manipulating lists

Bad

```
# Add three to all list members.
a = [3, 4, 5]
for i in range(len(a)):
a[i] += 3
```

Good

```
# List comprehension
a = [3, 4, 5]
a = [i + 3 for i in a]

# Or:
a = map(lambda i: i + 3, a)
```

Looping over a collection and indices

```
colors = ['red', 'green', 'blue', 'yellow']
```

What people usually do

```
for i in range(len(colors)):
    print(i, '--->', colors[i])
```

Better

```
for i, color in enumerate(colors):
    print(i, '--->', color)
```

Distinguishing multiple exit points in loops

What people usually do

```
def find(seq, target):
    found = False
    for i, value in enumerate(seq):
        if value == target:
            found = True
            break
    if not found:
        return -1
    return i
```

Better

```
def find(seq, target):
    for i, value in enumerate(seq):
        if value == target:
            break
    else:
        return -1
    return i
```

Unpacking sequences

```
p = 'Raymond', 'Hettinger', 0x30, 'python@example.com'
```

What people usually do

```
fname = p[0]
lname = p[1]
age = p[2]
email = p[3]
```

Better

```
fname, lname, age, email = p
```

Updating multiple state variables

What people usually do

```
def fibonacci(n):
    x = 0
    y = 1
    for i in range(n):
        print(x)
        t = y
        y = x + y
        x = t
```

Better

```
def fibonacci(n):
    x, y = 0, 1
    for i in range(n):
        print(x)
        x, y = y, x+y
```

Concatenating strings

```
names = ['raymond', 'rachel', 'matthew', 'roger', 'betty',
         'melissa', 'judith', 'charlie']
```

What people usually do

```
s = names[0]
for name in names[1:]:
    s += ', ' + name

print(s)
```

Better

```
print(', '.join(names))
```

Reading the contents from a file

What people usually do

```
f = open('data.txt')
try:
    data = f.read()
finally:
    f.close()
```

Better

```
with open('data.txt') as f:
    data = f.read()
```

References

- Four simple recommendations to encourage best practices in research software. Jiménez RC et al. F1000Research. https://f1000research.com/articles/6-876/v1
- Best Practices for Scientific Computing. Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, et al. (2014). PLOS Biology 12(1): e1001745. https://doi.org/10.1371/journal.pbio.1001745
- Best Practices in Scientific Computing – Software Carpentry http://swcarpentry.github.io/slideshows/best-practices/#slide-0
- Cookiecutter at GitHub: https://github.com/audreyr/cookiecutter
- Python Template from the NLeSC: https://github.com/NLeSC/python-template
- Sphinx documentation: http://www.sphinx-doc.org
- Doxygen documentation: http://doxygen.nl/

References

- The Hitchhacker's guide to Python by Kenneth Reitz, Tanya Schlusser. Publisher: O'Reilly Media, Inc. http://python-guide-pt-br.readthedocs.io/en/latest/
- Transforming Code into Beautiful, Idiomatic Python by Raymond Hettinger - PyCon 2013. https://www.youtube.com/watch?v=OSGv2VnC0go
- Raymond Hettinger - Beyond PEP 8 -- Best practices for beautiful intelligible code – PyCon 2015 https://www.youtube.com/watch?v=wf-BqAjZb8M
- Effective Computation in Physics by Anthony Scopatz, Kathryn D. Huff. Publisher: O'Reilly Media, Inc. http://physics.codes/
- Ned Batchelder: Getting Started Testing - PyCon 2014 https://www.youtube.com/watch?v=FxSsnHeWQBY
- pytest web site (https://pytest.org/) and a nice tutorial: https://semaphoreci.com/community/tutorials/testing-python-applications-with-pytest

THANK YOU FOR YOUR ATTENTION

www.prace-ri.eu