1 Introduction

- Python is interpreted: Python source code is executed by a program known as an interpreter
- In compiled languages (e.g., C/C++, Fortran), source code is compiled to an executable program
- Compiled programs generally run faster than interpreted programs
- Interpreted languages are often better for rapid prototyping and high-level program control
- Optimizing “time to science” might suggest prioritizing program development time over computational run time (or maybe vice versa)
- Python has become a popular language for scientific computing for many reasons
- How should we best use Python for applications that require numerically intensive computations?
- Can we have the best of both worlds (expressiveness and performance)?
- If there are tradeoffs, where are they and how can they be mitigated?
- As with most things in Python, there is no one answer...

2 CPython and the Python/C API

- CPython: the reference implementation of the language, and the most widely used interpreter (generally installed as “python”)
- Alternative implementations/interpreters exist (e.g., IronPython, Jython, PyPy) — we will not consider these here
- IPython & Jupyter kernel are thin Python layers on top of CPython to provide additional functionality
  - but typically use python myprogram.py for long-running programs in batch
- CPython compiles Python source code to bytecodes, and then operates on those
- CPython, written in C, is accompanied by an Application Programming Interface (API) that enables communication between Python and C (and thus to basically any other language)
- Python/C API allows for compiled chunks of code to be called from Python or executed within the CPython interpreter → extension modules
- Much core functionality of the Python language and standard library are written in C
3 Extension Modules

- a compiled shared object library (.so, .dll, etc.) making use of Python/C API
  - compiled code executing operations of interest
  - wrapper/interface code consisting of calls to Python/C API and underlying compiled code
- can be imported into python interpreter just as pure Python source code can

3.1 Extension modules

[1]:
```python
import math
print(math.cos(math.pi))
print(math.__file__)
```

-1.0

[2]:
```
!nm $math.__file__
```

```
U  _PyArg_Parse
U  _PyArg_ParseTuple
U  _PyArg_UnpackTuple
U  _PyBool_FromLong
U  _PyErr_Clear
U  _PyErr_ExceptionMatches
U  _PyErr_Format
U  _PyErr_Occurred
U  _PyErr_SetFromErrno
U  _PyErr_SetString
U  _PyExc_MemoryError
U  _PyExc_OverflowError
U  _PyExc_TypeError
U  _PyExc_ValueError
U  _PyFloat_AsDouble
U  _PyFloat_FromDouble
U  _PyFloat_Type
00000000000012f0  T  _PyInit_math
U  _PyIter_Next
U  _PyLong_AsDouble
U  _PyLong_AsLongAndOverflow
U  _PyLong_FromDouble
U  _PyLong_FromLong
U  _PyLong_FromUnsignedLong
U  _PyMem_Free
U  _PyMem_Malloc
U  _PyMem_Realloc
```
U _PyModule_AddObject
U _PyModule_Create2
U _PyNumber_Index
U _PyNumber_Lshift
U _PyNumber_Multiply
U _PyNumber_TrueDivide
U _PyObject_GetIter
U _PyType_IsSubtype
U _PyType_Ready
U _Py_BuildValue

0000000000006b30 s _SmallFactorials
U __PyArg_ParseStack
U __PyArg_ParseStackAndKeywords
U __PyArg_UnpackStack
U __PyLong_Frexp
U __PyLong_GCD
U __PyObject_FastCallDict
U __PyObject_LookupSpecial
U __Py_dg_infinity
U __Py_dg_stdnan

00000000000062d0 T __Py_log1p
U ___error
U ___memcpy_chk
U ___stack_chk_fail
U ___stack_chk_guard
U _abort
U _acos
U _acosh
U _asin
U _asinh
U _atan
U _atan2
U _atanh
U _ceil
U _copysign
U _cos
U _cosh
U _erf
U _erfc
U _exp
U _expm1
U _fabs

0000000000006100 t _factorial_partial_product
U _floor
U _fmod
U _frexp

0000000000006a70 s _gamma_integral
U _hypot
4 Hybrid Codes

It is often advantageous to blend high-level languages for control with low-level languages for performance. Overall performance depends on the granularity of computations in compiled code and the overhead required to communicate between languages.

There are many different tools the support the interleaving of Python and compiled extension modules.
5 Comments and caveats about performance optimization

- Make sure you have the right algorithm for the task at hand
- Remember that “premature optimization is the root of all evil” (D. Knuth)
- Focus on performance optimization only for those pieces of code that need it
- Algorithms and computational architectures are complex: be empirical about performance

6 Outline

- Introduction
- Leveraging Compiled Code
  - Compiled Third-Party Libraries
  - Compiling Custom Code
- Leveraging Additional Computational Resources
  - Parallel Processing
- Writing Faster Python
- Performance Assessment

Material derived in part from Cornell Virtual Workshop (CVW) tutorial on “Python for High Performance”, at https://cvw.cac.cornell.edu/python.

These slides will be available as a Jupyter notebook linked from the CVW site. Note: some code presented here is in the form of incomplete snippets to illustrate a point. This notebook will therefore not run from start to finish without errors.

7 Third-Party Libraries for Numerical & Scientific Computing

a.k.a. The Python Scientific Computing Ecosystem

- Most specific functionality for scientific computing is provided by third-party libraries, which are typically a mix of Python code and compiled extension modules
  - NumPy: multi-dimensional arrays and array operations, linear algebra, random numbers
  - SciPy: routines for integration, optimization, root-finding, interpolation, fitting, etc.
  - Pandas: Series and Dataframes to handle tabular data (e.g., from spreadsheets)
  - Scikit-learn, TensorFlow, Caffe, PyTorch, Keras: machine learning
  - NetworkX: networks
  - Matplotlib, Seaborn: plotting
  - etc.
- Bundled distributions (e.g, Anaconda) contain many of these, with tools for installing additional packages

8 NumPy

- NumPy = “Numerical Python”, the cornerstone of the Python Scientific Computing Ecosystem
- largely written in C, with links to BLAS and LAPACK for linear algebra
- provides multidimensional arrays and array-level operations (a form of vectorization)
• conventional wisdom: avoid loops in Python and use array syntax
• a challenge: figuring out how to express complex operations solely using array syntax (including indexing, slicing, and broadcasting)
• a caveat: convenient syntax can disguise performance inefficiencies (e.g., temporary arrays)

9 Elementwise array operations

```
import numpy as np
a = np.random.random((1000,1000))
b = np.random.random((1000,1000))

log_a = np.log(a)  # an example of a ufunc

c = a + b  # throws ValueError if a and b not the same shape

assert(a.shape == b.shape)  # throws AssertionError if a and b not the same shape

c = np.zeros_like(a)  # prefills a zero array of the correct shape
for i in range(a.shape[0]):
    for j in range(a.shape[1]):
        c[i,j] = a[i,j] + b[i,j]
```

10 More complicated array operations

• a challenge: figuring out how to express complex operations solely using array syntax (including indexing, slicing, and broadcasting)

```
# numpy outer product

def outer(a,b):
    multiply(a.ravel()[:, newaxis], b.ravel()[newaxis, :], out)

# e.g., generalized outer product to compute pairwise Hamming distances between all subsequences in a 2D array
```
def Hamming_outer(a0, a1):
    return np.sum(np.bitwise_xor(a0[:, np.newaxis], a1[np.newaxis, :]), axis=2)

11 More complicated array operations

# e.g., approximate Laplacian on 2D array by summing up shifted copies of array

def Del2(a, dx):
    nx, ny = a.shape
    del2a = scipy.zeros((nx, ny), float)
    del2a[1:-1, 1:-1] = (a[1:-1, 2:] + a[1:-1, :-2] + 
                        a[2:, 1:-1] + a[:-2, 1:-1] - 4.*a[1:-1, 1:-1])/(dx*dx)
    return del2a

    # roughly equivalent to:
    for i in range(1, nx-1):
        for j in range(1, ny-1):
            del2a[i, j] = (a[i+1, j] + a[i-1, j] + a[i, j+1] + a[i, j-1] - 4*a[i, j])/
                          (dx*dx)

12 NumPy and optimized libraries

- NumPy performance can be enhanced if linked to optimized libraries
  - numpy.__config__.show() to list what blas, lapack libraries numpy is linked to
- Intel MKL (Math Kernel Library) is highly optimized for Intel processors, and bundled with Anaconda Python distribution
- On multi-core architectures, optimized libraries can provide NumPy-based parallelism for free by setting environment variables appropriate for processor
  - MKL_NUM_THREADS = N # if using MKL
  - OMP_NUM_THREADS = N # if libraries support OpenMP

13 NumPy and temporary array creation

\[
\frac{\partial u}{\partial t} = D \nabla^2 u + u(1-u)
\]

u += dt * ( D * Del2(u) + u * (1 - u) )

Temporary arrays created: * Del2(u) * D * Del2(u) * 1 - u * u * (1 - u) * D * Del2(u) + u * (1 - u) * dt * ( D * Del2(u) + u * (1 - u) )
14 SciPy

- SciPy = “Scientific Python”
- sits on top of NumPy, wrapping many C and Fortran numerical routines, with convenient Python interface
- routines for integration, optimization, root-finding, interpolation, fitting, etc.
- Python callbacks enable ease-of-use, but with some performance penalty

```python
from scipy.integrate import solve_ivp

def func(t, y):
    dydt = ...
    return dydt

# integrate dy/dt = func(t, y)
# with initial condition y=y0
sol = solve_ivp(func, tspan, y0)
```

- can hack things with the help of wrapper generation tools to get compiled routines to accept a pointer to a compiled callback function
- scipy.LowLevelCallable recently introduced to work with some functions (not solve_ivp)

15 Compiling Custom Code

- Compilation frameworks
  - Numba and Cython
  - internal code generation for deep learning neural networks: Tensorflow, PyTorch, etc.
- Wrapper and interface generators

16 Numba

- a just-in-time (JIT) compiler that compiles a subset of Python and NumPy source to machine code for execution in CPython
• uses LLVM to convert Python bytecodes to intermediate representation (IR), and generates optimizes C code from that
• can be configured in more detail, but some performance improvements simply via addition of @jit decorator

17  Upending the conventional wisdom via compilation

[3]:

```python
import numpy as np
from numba import jit

X = np.random.random((1000,1000))
Y = np.random.random((1000,1000))
Z = np.random.random((1000,1000))

def f1(x,y,z):
    return x + 2*y + 3*z

def f2(x,y,z):
    result = np.empty_like(x)
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            result[i,j] = x[i,j] + 2*y[i,j] + 3*z[i,j]
    return result

@jit
def f3(x,y,z):
    result = np.empty_like(x)
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            result[i,j] = x[i,j] + 2*y[i,j] + 3*z[i,j]
    return result
```

[4]:

```bash
%timeit f1(X,Y,Z)
%timeit f2(X,Y,Z)
%timeit f3(X,Y,Z)
```

2.96 ms ± 50.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
1.15 s ± 16.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.73 ms ± 133 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

18  Cython

• a C-like superset of the Python language that adds capabilities for type declarations
• a system for converting chunks of Python/Cython code into C for compilation and execution within CPython
• can be used to compile extension modules ahead of time, or for on-the-fly compilation using IPython magic functions (%%cython)
• mature technology used by many packages in Python ecosystem use Cython to optimize particular functionality (Pandas, Scikit-learn, etc.)
• provides support for OpenMP-based parallelism through the cython.parallel module

```
# Example: adapted from Cython tutorial at https://cython.readthedocs.io/en/latest/src/tutorial/cython_tutorial.html#primes

def primes(nb_primes):
    if nb_primes > 1000:
        nb_primes = 1000

    p = [0]*nb_primes
    len_p = 0  # The current number of elements in p.
    n = 2
    while len_p < nb_primes:
        # Is n prime?
        for i in p[:len_p]:
            if n % i == 0:
                break

        # If no break occurred in the loop, we have a prime.
        else:
            p[len_p] = n
            len_p += 1
            n += 1

    # Let's return the result in a python list:
    result_as_list = [prime for prime in p[:len_p]]
    return result_as_list

primes100 = primes(100)
print(primes100)
```

```
```

```
[6]: %load_ext Cython

[7]: %%cython -a
```
cdef int n, i, len_p
cdef int p[1000]
if nb_primes > 1000:
    nb_primes = 1000

len_p = 0  # The current number of elements in p.
n = 2
while len_p < nb_primes:
    # Is n prime?
    for i in p[:len_p]:
        if n % i == 0:
            break

    # If no break occurred in the loop, we have a prime.
    else:
        p[len_p] = n
        len_p += 1
        n += 1

    # Let's return the result in a python list:
result_as_list = [prime for prime in p[:len_p]]
return result_as_list
cprimes100 = primes_cython(100)
print(cprimes100)


[7]: <IPython.core.display.HTML object>
Cython output in Jupyter shows annotated version of code, which can be clicked on to reveal injected C code

[8]: %timeit primes(100)
%timeit primes_cython(100)

367 µs ± 1.15 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
18.8 µs ± 21.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
19 **Wrapper and Interface Generators**

Useful if you have an existing code base (C/C++, Fortran, etc.) that you would like to be able to call from Python

Typically a multi-step process:

- Python/C API code generation from function and class declarations
- wrapper code compiled and linked with underlying library → .so file
- possibly some additional Python code generated

Some of the many packages available:

- CFFI and ctypes provide "foreign function interfaces", or lightweight APIs, for calling C libraries from within Python
- Boost.python helps write C++ libraries that Python can import
- SWIG reads C and C++ header files and generates a library than Python (or many other scripting languages) can import
- F2PY reads Fortran code and generates a library that Python can import
- PyCUDA and PyOpenCL provide access within Python to GPUs

20 **Parallel Processing**

- Parallelization: across threads, across cores, across processors
  - …to reduce wallclock time to solution
  - …to solve bigger problems that don’t fit in a single processor
- CPython uses Global Interpreter Lock (GIL): only one thread can run at a time
- GIL does not apply to multithreaded code in compiled library (e.g., MKL)
- Multiple Python processes can run concurrently (each with their own GIL)
- (and new development is apparently underway to enable multiple interpreters within a single process)
21 Multiprocessing and concurrent.futures modules

- control of multiple processes and communication between them
- useful for multiprocessing within a multi-core, shared memory processor

```python
[4]: def f(x):
    return x*x

import multiprocessing

p = multiprocessing.Pool(processes=4)
print(p.map(f, range(10)))

# or, equivalently

import concurrent.futures

executor = concurrent.futures.ProcessPoolExecutor(max_workers=4)
print(list(executor.map(f, range(10))))
```

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
22  More multiprocessing (from networkx docs)

Link to networkx page

```python
def _betmap(G_normalized_weight_sources_tuple):
    return nx.betweenness_centrality_source(*G_normalized_weight_sources_tuple)

def betweenness_centrality_parallel(G, processes=None):
    """Parallel betweenness centrality function""
    p = Pool(processes=processes)
    node_divisor = len(p._pool) * 4
    node_chunks = list(chunks(G.nodes(), int(G.order() / node_divisor)))
    num_chunks = len(node_chunks)
    bt_sc = p.map(_betmap,
                  zip([G] * num_chunks,
                       [True] * num_chunks,
                       [None] * num_chunks,
                       node_chunks))

    # Reduce the partial solutions
    bt_c = bt_sc[0]
    for bt in bt_sc[1:]:
        for n in bt:
            bt_c[n] += bt[n]
    return bt_c
```

23  Message passing with mpi4py

- MPI = “Message Passing Interface” — a widely-used standard for parallel processing
- Many Python wrappers to MPI developed over the years; mpi4py is probably most widely used at this time
- use mpiexec or ibrun in shell to run a python program over multiple processors
  - e.g., ibrun python myprogram.py on Stampede2 at TACC

23.1  mpi4py

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
print(comm.Get_rank())
print(comm.Get_size())

import numpy as np
data = np.arange(1000, dtype=np.float64)
data2 = np.zeros(1000, dtype=np.float64)
print(data2[:5])
```
24 Exercise: Estimating π via Monte Carlo using mpi4py

Exercise in CVW on “Python for High Performance”

25 Other Parallel Tools

- Dask
  - distributed NumPy arrays and Pandas DataFrames
  - task scheduling interface for custom workflows
- Joblib
  - lightweight pipelining with support for parallel processing
  - specific optimizations for NumPy arrays
- ipyparallel
  - parallel and distributed computing within IPython

26 Writing Faster Python

- Collections and containers
- Lazy evaluation
- Memory management

26.1 Collections and containers

Use the right data structure for the job

```python
set1 = set(range(0,1000))  # create a set with a bunch of numbers in it
set2 = set(range(500,2000))  # create another set with a bunch of numbers in it
iset = set1 & set2  # same as iset = set1.intersection(set2)

list1 = list(range(0,1000))  # create a list with a bunch of numbers in it
list2 = list(range(500,2000))  # create another list with a bunch of numbers in it
iset = [e1 for e1 in list1 for e2 in list2 if e1==e2]  # uses list comprehensions
```
26.2 Lazy evaluation

- a strategy implemented in some programming languages whereby certain objects are not produced until they are needed
- often used in conjunction with functions that produce collections of objects
- if you only need to iterate over the items in a collection, you don’t need to produce that entire collection

Python provides a variety of mechanisms to support lazy evaluation:

- generators: like functions, but maintain internal state and yield next value when called
- dictionary views (keys and values)
- range: for i in range(N)
- zip: pairs = zip(seq1, seq2)
- enumerate: for i, val in enumerate(seq)
- open: with open(filename, 'r') as f

27 Performance Assessment

- Timing: getting timing information for particular operations
- Profiling: figuring out where time is being spent

IPython magic functions

- %timeit: uses timeit module from Python Standard Library
- %prun: uses profile module from Python Standard Library

```
[7]: from scipy.integrate import solve_ivp

sigma = 10
rho = 28
beta = 8.0/3

def lorenz(t, xyz):
    x,y,z = xyz
    xdot = sigma*(y-x)
    ydot = x*(rho-z) - y
    zdot = x*y - beta*z
    return [xdot, ydot, zdot]

def integrate_lorenz(ic=[1.,1.,1.]):
    sol = solve_ivp(lorenz, (0., 100.), ic, method='LSODA')
    return sol
```
Integrate the Lorenz system and plot the results.

```python
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(10,8))
ax = fig.gca(projection='3d')
ax.plot(xs, ys, zs, lw=0.5)
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_zlabel("Z Axis")
ax.set_title("I'll fly away");
```
Summary

- Python and Performance together in one big tent
- There is no one answer: think about how you can best optimize “time to science”
- Make sure you have the right algorithms, do not prematurely optimize, and be empirical about performance