How to make your Python Code fast?

High-Performance Computing

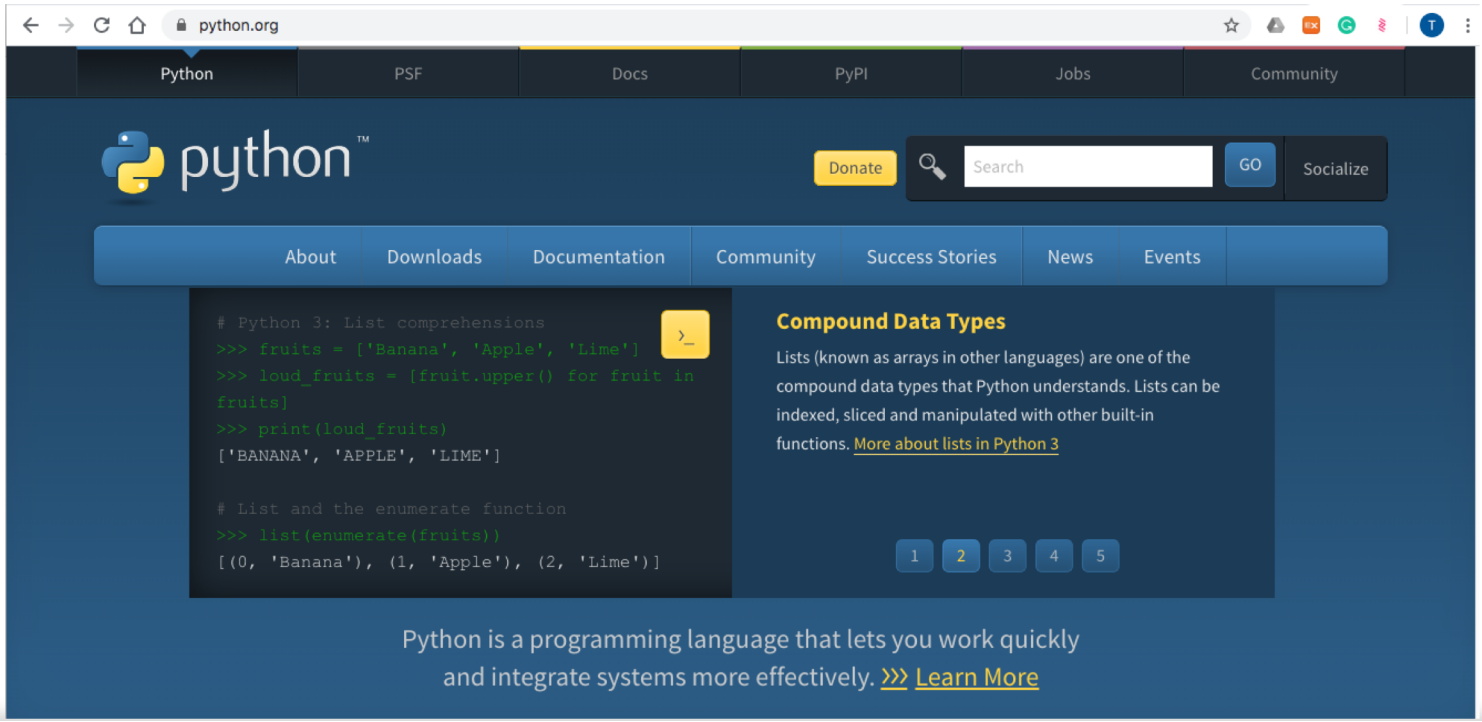
Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

What is Python?

- Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.
- Python is a programming language that appeared in 1991. Compare with Fortran (1957), C (1972), C++ (1983).
- While the older languages still dominate High Performance Computing (HPC), popularity of Python is growing.



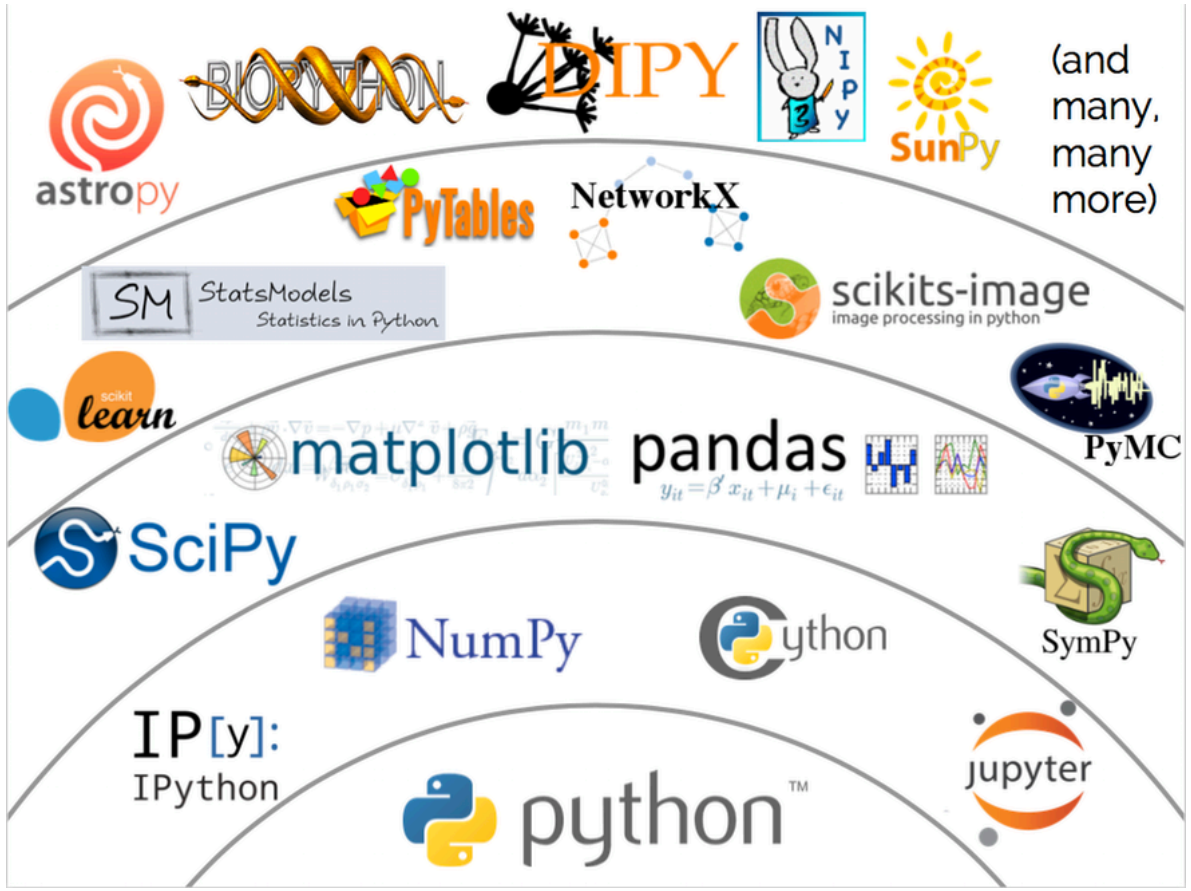
Python advantages

- Designed from the start for better code readability
- Allows expression of concepts in fewer lines of code
- Has dynamic type system, variables do not have to be declared
- Has automatic memory management
- Has large number of easily accessible, extensive libraries (eg. NumPy, SciPy)
- All this makes developing new codes easier

Python disadvantages

- Python is generally slower than compiled languages like C, C++ and Fortran
- Complex technical causes include dynamic typing and the fact that Python is interpreted, not compiled
- This does not matter much for a small desktop program that runs quickly.
- However, this will matter a lot in a High Performance Computing environment.
- Python use in HPC parallel environments is relatively recent, hence parallel techniques less well known.

Scientific Python



(Source: Jake VanderPlas, "The State of the Stack," SciPy Keynote (SciPy 2017).)

- Python is becoming a defacto standard for data science
- Python is not a scientific programming language, but a glue
- High-level syntax wraps high- performance low level libraries
- Interoperability allows legacy codes to be used
- Notebooks allow for interactive analysis on HPC and cloud infrastructures
- New projects are focused on performance and parallelization

Python2 vs Python3??

- Something to be aware of is that there are two major versions of Python currently in use (the so called Python 2.x branch and Python 3.x branch). **We will be using the 3.x line in this course** (and it is not backward compatible with the 2.x branch).
- If you are using your account on our department's hopper system, Python is already installed and can be started by typing the command **python3** in a console window (not to be confused with the command python which is the default 2.x branch). There is also a development environment for Python known as IDLE that is available by navigating the start menu to Applications -> Development -> IDLE3.
- We note that there is a bit of an inherent slowdown when running Python programs within IDLE rather than directly in the interpreter, so for large-scale computations it is often helpful to know how to run a script directly in the interpreter. This can be done through a typical command line interface in a terminal window (or command prompt on Windows), with a command such as
- python3 myscript.py

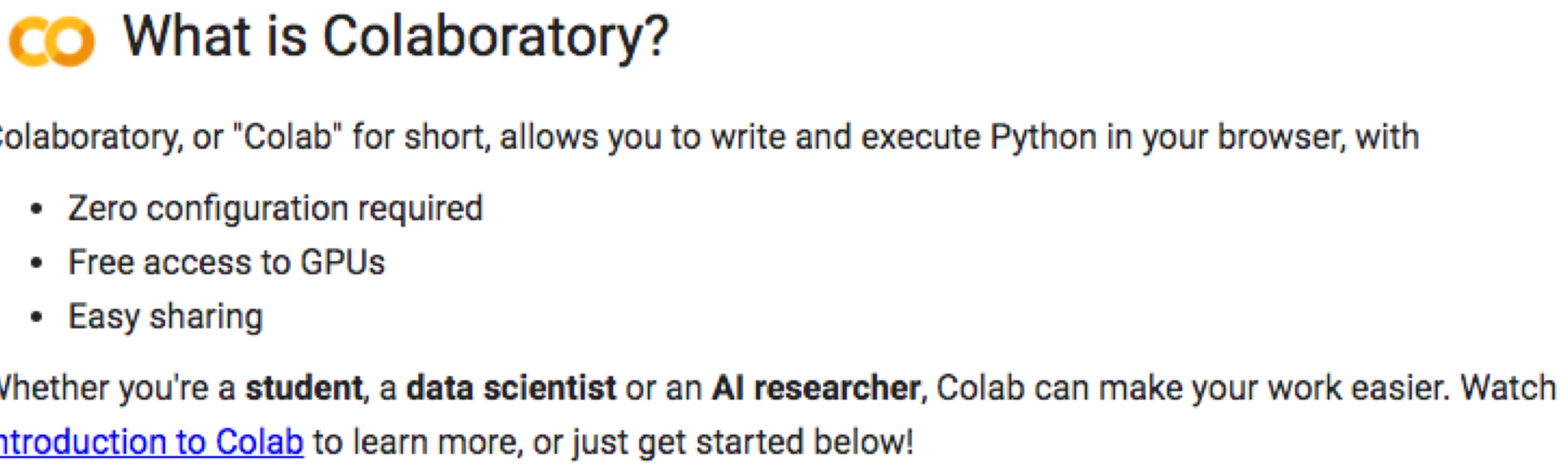
Local Installation

- If you wish to run locally on your own computer, Python can be easily installed.
- In fact, if you have an Apple computer, Python and IDLE are already pre-installed as part of the OSX distribution.
- You can open a terminal window and start Python by typing the command python, or IDLE by typing the command idle.
- For other platforms, you can download a Python installer from www.python.org/download; **make sure to download the Python 3.9 line (Python 3.9.6 is the most current release as of July 2021)**.

Web-based Python Tools

- here is an interactive interpreter online at https://repl.it/languages/python3.

- Jupyter Notebook: https://jupyter.org/
- Google colab: https://colab.research.google.com/



What is Colaboratory?

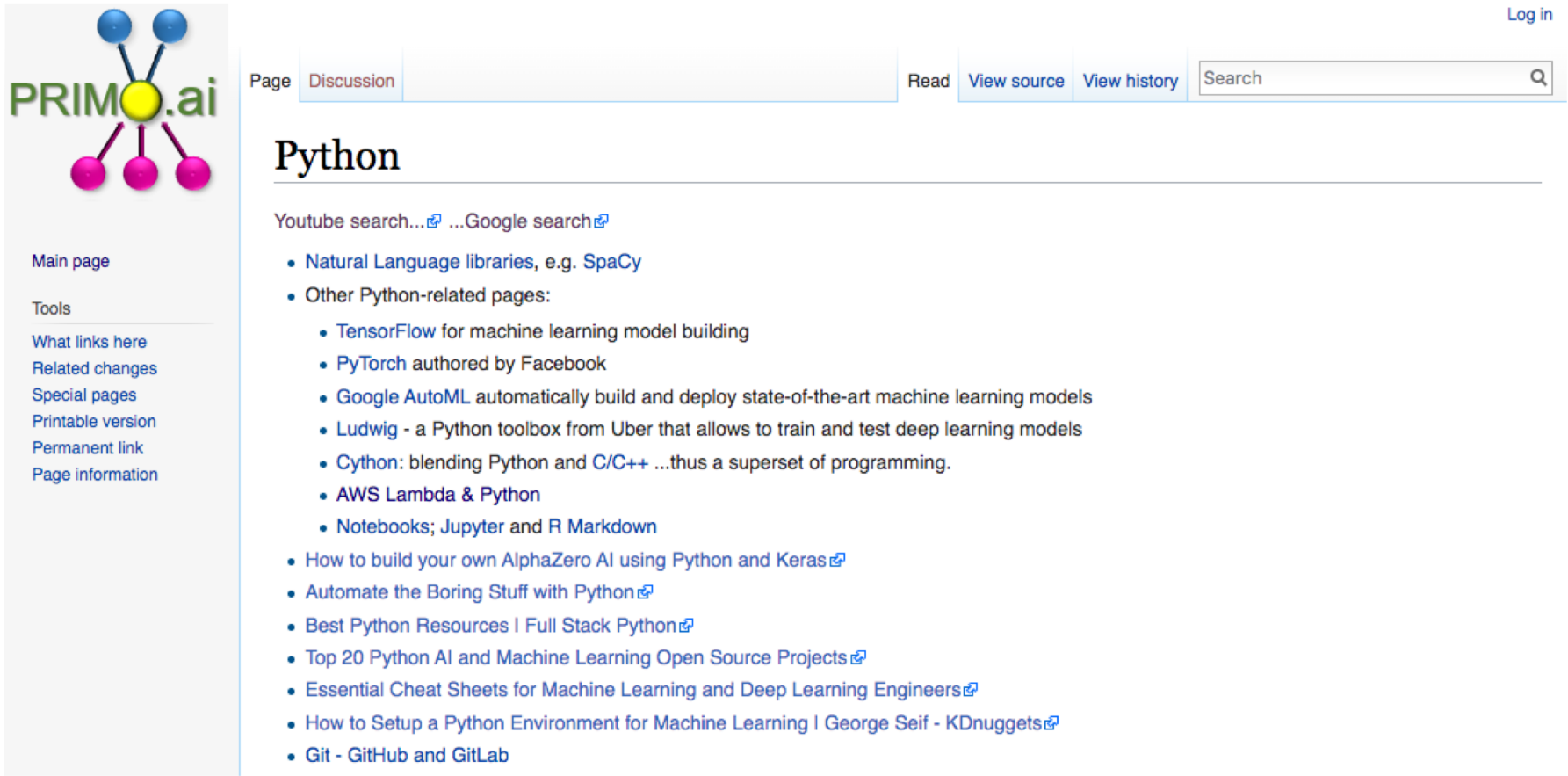
Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

- Zero configuration required
- Free access to GPUs
- Easy sharing

Whether you're a **student**, a **data scientist** or an **AI researcher**, Colab can make your work easier. Watch Introduction to Colab to learn more, or just get started below!

Scientific Python

- If you want to learn the Scientific Python, everything is here!
- http://primo.ai/index.php?title=Python



We are HPC class

- How to make if fast ?
- HPC + Python ?

How to make your Python Code fast?

You can consider many things, but below considerations will be good starting:

- Speeding up Python with Numpy
- Speeding up Python with PyPy
- Speeding up Python with Cython
- Speeding up Python with Numba
- Parallel Programming with
  - mpi4Py (distributed memory)
  - multiprocessing (shared memory)
  - PyCUDA, PyOPENCL, CuPy (GPU)
  - Dask (flexible library for parallel computing in Python)

Virtual Environment

- https://docs.python.org/3/tutorial/venv.html
- Create your own VE at hopper
  - $ cd Your HPC Course Working Directory
  - $ python3 -m venv python-hpc-env
  - $ source python-hpc-env/bin/activate

  - This will change your prompt as `(python-hpc-env) ai@ubuntu-20-04:~/Lab$`
  - $ which pip3
  - $ pip3 install --upgrade pip // This will install pip pip-21.1.3

Example: Computing the value of π=3.14159…

- For

$$F(x) = \frac{4.0}{(1 + x^2)}$$

it is known that the value of π can be computed by the numerical integration

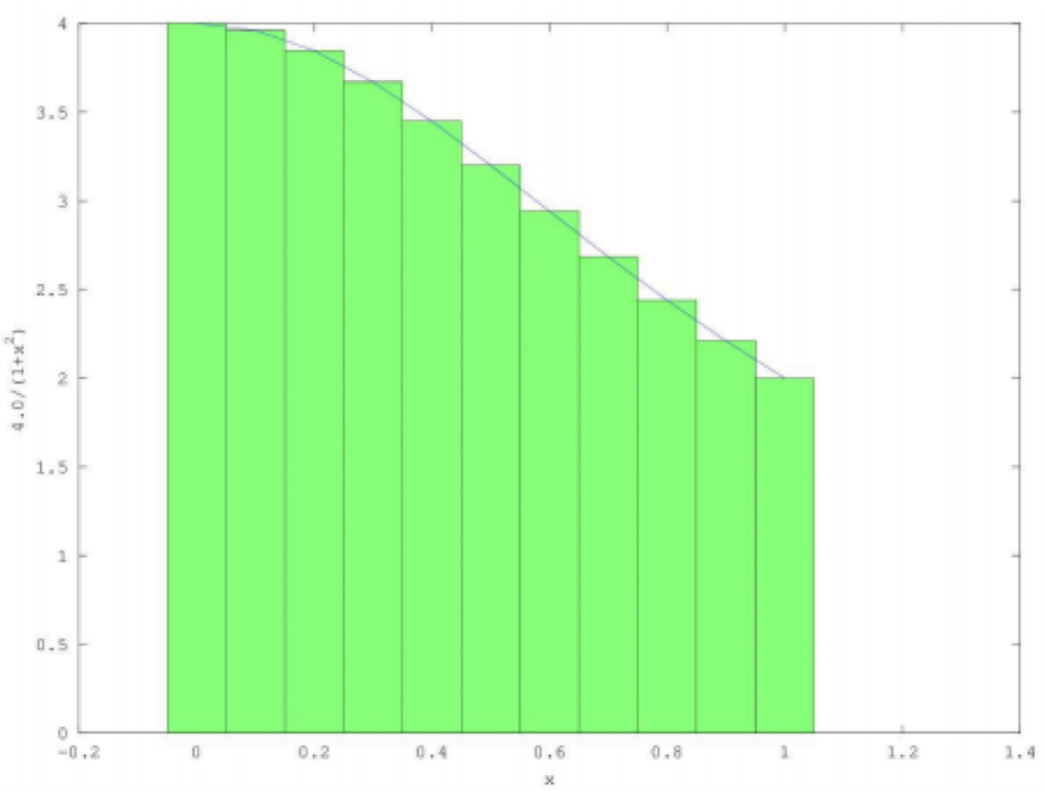
$$\int_0^1 F(x)dx = \pi$$

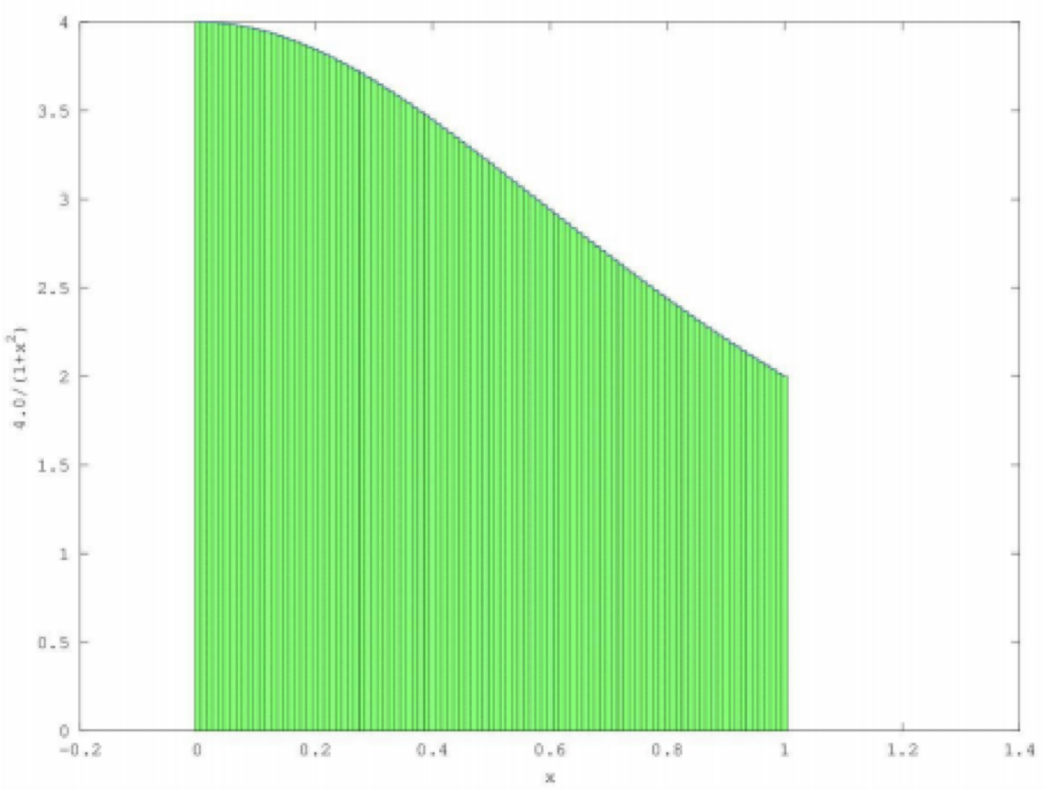
- This can be approximated by

$$\sum_{i=0}^{n} F(x_i)\Delta x \approx \pi$$

Example: Computing the value of π=3.14159…

- By increasing the number of steps (ie. smaller Δx), the approximation gets more precise.





https://www.nesi.org.nz/sites/default/files/mpi-in-python.pdf

Implementing C and Python (Hands On Assignment)

```c
#include <stdio.h>
#include <time.h>

void Pi(int num_steps)
{
  double begin, end, elapsed,  pi, step, x, sum;
  int i;
  begin = clock();
  step = 1.0/(double)num_steps;
  sum = 0;
  for (i=0;i<num_steps;i++) {
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
  end = clock();
  elapsed = (float)(end-begin)/CLOCKS_PER_SEC;
  printf("Pi with %d steps is %f in %f secs\n", num_steps, pi,elapsed);
}

int main() {
  Pi(100000000);
  return 0;
}
```

```
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ gcc pi.c -o pi
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ ./pi
Pi with 100000000 steps is 3.141593 in 0.786185 secs
```

```python
import time

def Pi(num_steps):
    begin = time.time()
    step = 1.0/num_steps
    sum = 0
    for i in range(num_steps):
        x= (i+0.5)*step
        sum = sum + 4.0/(1.0+x*x)
        pi = step * sum
    end = time.time()
    elapsed = end-begin
    print("Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))

if __name__ == '__main__':
    Pi(100000000)
```

```
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ python3 pi.py
Pi with 100000000 steps is 3.141593 in 13.721580 secs
```

Profiling

- The Python Profilers: https://docs.python.org/3/library/profile.html
- Line-by-line profiling is often useful: https://github.com/rkern/line_profiler
  - Put @profile above the function that you're interested in

- Hands On:
  - Install line_profiler with a user mode
    - $ pip3 install line_profiler
  - Copy pi.py to pi_profile.py as

```
$ cp pi.py pi_profile.py
```

  - Add @profile to the function pi
  - $ ../../hpc-env/bin/kernprof -l -v pi_profile.py

Profiling

- The Python Profilers: https://docs.python.org/3/library/profile.html
- Line-by-line profiling is often useful: https://github.com/rkern/line_profiler
  - Put @profile above the function that you're interested in

- Hands On:
  - Install line_profiler with a user mode
    - `$ pip3 install line_profiler`
  - Copy pi.py to pi_profile.py and reduce the number of iterations to 10000000
  - Add @profile to the function pi
  - Run profiling as below
    - `$ kernprof -l -v pi_profile.py`

```
1  import time
2
3  @profile
4  def Pi(num_steps):
5      begin = time.time()
6      step = 1.0/num_steps
7      sum = 0
8      for i in range(num_steps):
9          x= (i+0.5)*step
10         sum = sum + 4.0/(1.0+x*x)
11         pi = step * sum
12     end = time.time()
13     elapsed = end-begin
14     print("Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))
15
16 if __name__ == '__main__':
17     Pi(10000000)
```

Profiling

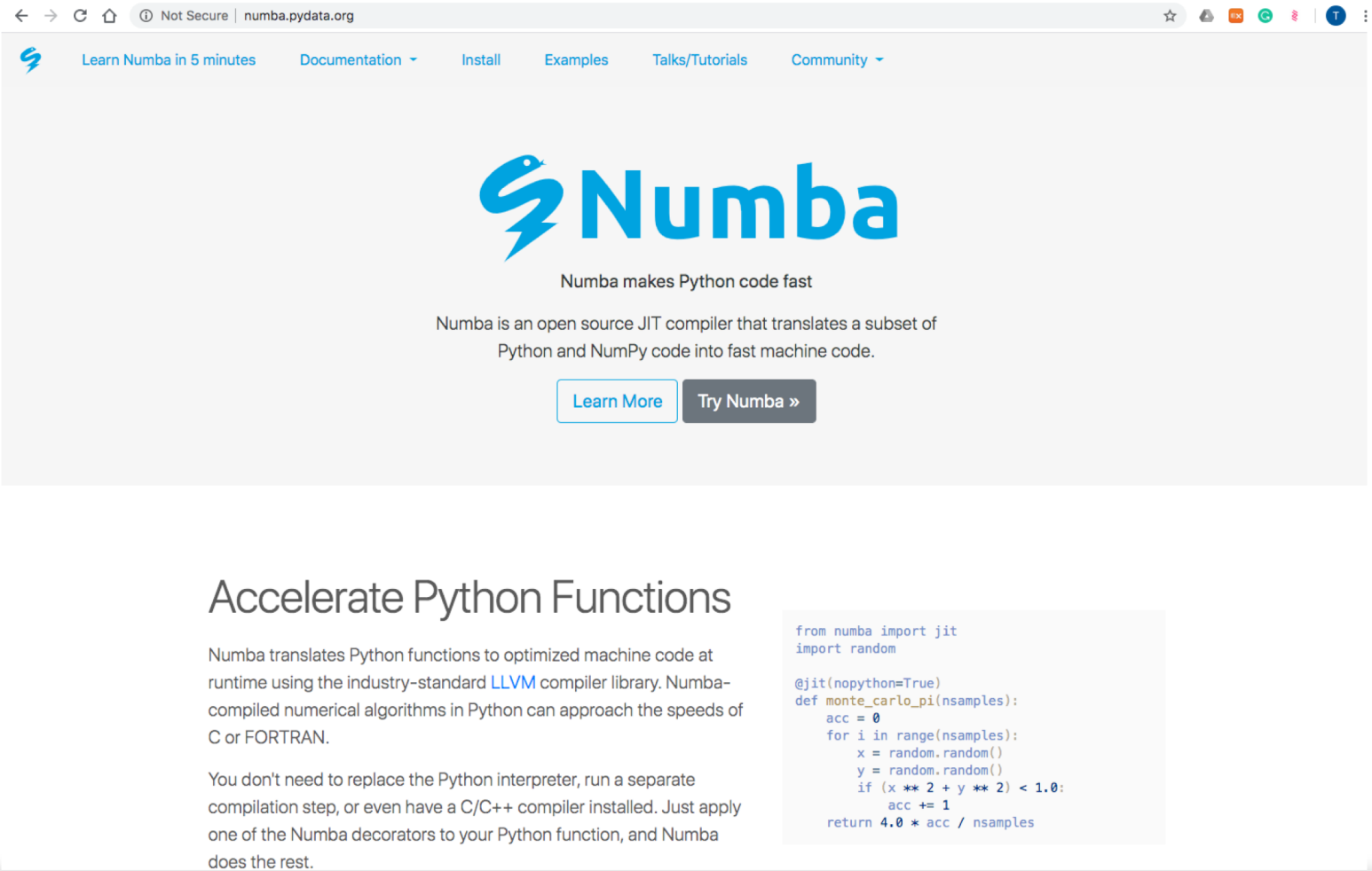
```
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ kernprof -l -v pi_profile.py
Pi with 10000000 steps is 3.141593 in 32.005736 secs
Wrote profile results to pi_profile.py.lprof
Timer unit: 1e-06 s

Total time: 15.8814 s
File: pi_profile.py
Function: Pi at line 3

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           @profile
     4                                           def Pi(num_steps):
     5         1          6.0      6.0      0.0      begin = time.time()
     6         1          2.0      2.0      0.0      step = 1.0/num_steps
     7         1          0.0      0.0      0.0      sum = 0
     8  10000001    3696589.0      0.4     23.3      for i in range(num_steps):
     9  10000000    4038106.0      0.4     25.4          x= (i+0.5)*step
    10  10000000    4357205.0      0.4     27.4          sum = sum + 4.0/(1.0+x*x)
    11  10000000    3789360.0      0.4     23.9          pi = step * sum
    12         1         23.0     23.0      0.0      end = time.time()
    13         1          1.0      1.0      0.0      elapsed = end-begin
    14         1        135.0    135.0      0.0      print("Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))
```

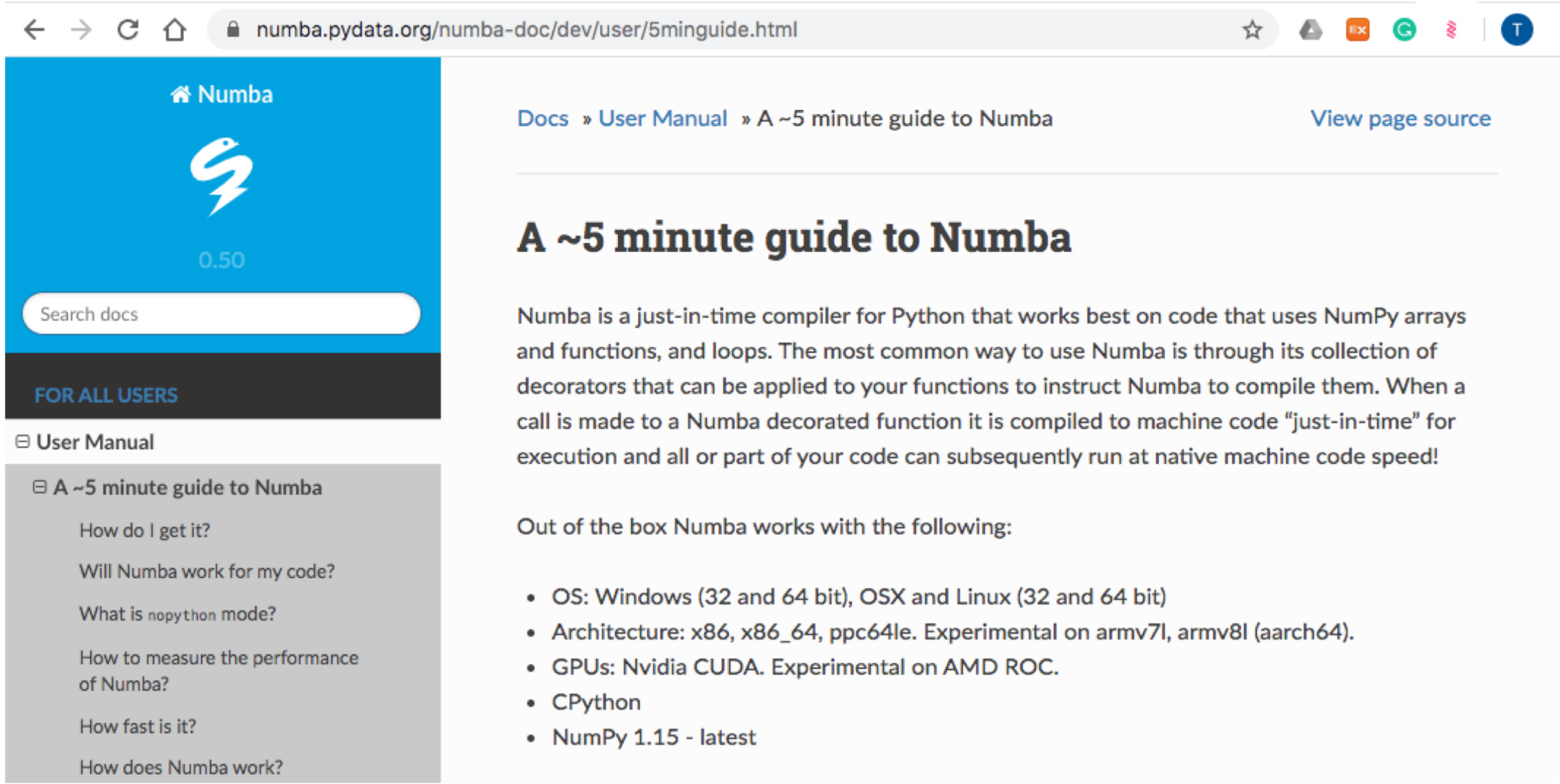
Numba

- Numba (http://numba.pydata.org/) is a just-in-time compiler and produces optimized native code from Python code.



Numba

- https://numba.pydata.org/numba-doc/dev/user/5minguide.html



Install Numba on your python-hpc-env

```
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ pip3 install numba
Collecting numba
  Downloading numba-0.53.1-cp38-cp38-manylinux2014_x86_64.whl (3.4 MB)
     |████████████████████████████████| 3.4 MB 2.3 MB/s
Collecting llvmlite<0.37,>=0.36.0rc1
  Downloading llvmlite-0.36.0-cp38-cp38-manylinux2010_x86_64.whl (25.3 MB)
     |████████████████████████████████| 25.3 MB 215 kB/s
Requirement already satisfied: setuptools in /home/ai/Lab/python-hpc-env/lib/python3.8/site-packages (from numba) (44.0.0)
Collecting numpy>=1.15
  Downloading numpy-1.21.1-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (15.8 MB)
     |████████████████████████████████| 15.8 MB 100.7 MB/s
Installing collected packages: numpy, llvmlite, numba
Successfully installed llvmlite-0.36.0 numba-0.53.1 numpy-1.21.1
```

Hands on Lab (Assignment)

- Separate the bottleneck into a function (make a new function)

```python
import time

def loop(num_steps):
    step = 1.0/num_steps
    sum = 0
    for i in range(num_steps):
        x= (i+0.5)*step
        sum = sum + 4.0/(1.0+x*x)
    return sum

def Pi(num_steps):
    begin = time.time()
    sum = loop(num_steps)
    pi = sum/num_steps
    end = time.time()
    elapsed = end-begin
    print("Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))

if __name__ == '__main__':
    Pi(100000000)
```

Compiling Python code with @jit

Compiling Python code with `@jit`

Numba provides several utilities for code generation, but its central feature is the `numba.jit()` decorator. Using this decorator, you can mark a function for optimization by Numba's JIT compiler. Various invocation modes trigger differing compilation options and behaviours.

Basic usage

Lazy compilation

The recommended way to use the `@jit` decorator is to let Numba decide when and how to optimize:

```
from numba import jit

@jit
def f(x, y):
    # A somewhat trivial example
    return x + y
```

In this mode, compilation will be deferred until the first function execution. Numba will infer the argument types at call time, and generate optimized code based on this information. Numba will also be able to compile separate specializations depending on the input types. For example, calling the `f()` function above with integer or complex numbers will generate different code paths:

```
>>> f(1, 2)
3
>>> f(1j, 2)
(2+1j)
```

pi_numba.py Hands on Lab (Assignment)

- Import Numba and add a @jit decorator

```
 1 #pi_numba.py
 2 import time
 3 from numba import jit
 4
 5 @jit
 6 def loop(num_steps):
 7     step = 1.0/num_steps
 8     sum = 0
 9     for i in range(num_steps):
10         x= (i+0.5)*step
11         sum = sum + 4.0/(1.0+x*x)
12     return sum
13
14 def Pi(num_steps):
15     begin = time.time()
16     sum = loop(num_steps)
17     pi = sum/num_steps
18     end = time.time()
19     elapsed = end-begin
20     print("Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))
21
22 if __name__ == '__main__':
23     Pi(100000000)
```

pi_numba.py Hands on Lab (Assignment)

- Run the python program and compare the elapsed time. Did you get a speed-up?

```
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ python3 pi_numba.py
Pi with 100000000 steps is 3.141593 in 0.272201 secs
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ python3 pi.py
Pi with 100000000 steps is 3.141593 in 13.412631 secs
```

- Compare the Python runtime with C code runtime.

```
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ python3 pi_numba.py
Pi with 100000000 steps is 3.141593 in 0.278413 secs
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ ./pi
Pi with 100000000 steps is 3.141593 in 0.837732 secs
```

What is nopython mode?

The Numba `@jit` decorator fundamentally operates in two compilation modes, `nopython` mode and `object` mode. In the `go_fast` example above, `nopython=True` is set in the `@jit` decorator, this is instructing Numba to operate in `nopython` mode. The behaviour of the `nopython` compilation mode is to essentially compile the decorated function so that it will run entirely without the involvement of the Python interpreter. This is the recommended and best-practice way to use the Numba `jit` decorator as it leads to the best performance.

Should the compilation in `nopython` mode fail, Numba can compile using `object mode`, this is a fall back mode for the `@jit` decorator if `nopython=True` is not set (as seen in the `use_pandas` example above). In this mode Numba will identify loops that it can compile and compile those into functions that run in machine code, and it will run the rest of the code in the interpreter. For best performance avoid using this mode!

pi_numba_nopython.py Hands on Lab (Assignment)

- Import Numba with a nopython compilation mode and test: Similar performance
- Using a nopython compilation mode and call the function twice!!

```python
#pi_numba.py
import time
from numba import jit

@jit(nopython=True)
def loop(num_steps):
    step = 1.0/num_steps
    sum = 0
    for i in range(num_steps):
        x= (i+0.5)*step
        sum = sum + 4.0/(1.0+x*x)
    return sum

def Pi(num_steps):
    begin = time.time()
    sum = loop(num_steps)
    pi = sum/num_steps
    end = time.time()
    elapsed = end-begin
    print("(Before Compilation) Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))

    begin = time.time()
    sum = loop(num_steps)
    pi = sum/num_steps
    end = time.time()
    elapsed = end-begin
    print("(After Compilation) Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))

if __name__ == '__main__':
    Pi(100000000)
```

pi_numba.py Hands on Lab (Assignment)

- Run the python program and compare the elapsed time. Did you get a speed-up?
- Compare the Python runtime with C code runtime.

```
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ python3 pi_numba_nopython.py
(Before compilation) Pi with 100000000 steps is 3.141593 in 0.268823 secs
(After compilation) Pi with 100000000 steps is 3.141593 in 0.165081 secs
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ ./pi
Pi with 100000000 steps is 3.141593 in 0.808183 secs
```

Explicit Parallel Loops

Another feature of the code transformation pass (when `parallel=True`) is support for explicit parallel loops. One can use Numba's `prange` instead of `range` to specify that a loop can be parallelized. The user is required to make sure that the loop does not have cross iteration dependencies except for supported reductions.

A reduction is inferred automatically if a variable is updated by a binary function/operator using its previous value in the loop body. The initial value of the reduction is inferred automatically for the `+=`, `-=`, `*=`, and `/=` operators. For other functions/operators, the reduction variable should hold the identity value right before entering the `prange` loop. Reductions in this manner are supported for scalars and for arrays of arbitrary dimensions.

pi_numba_nopython_parallel.py Hands on Lab (Assignment)

- Import Numba with a nopython compilation mode
- Import prange and change the range call to prange

```
1  #pi_numba.py
2  import time
3  from numba import jit, prange
4
5  @jit(nopython=True, parallel=True)
6  def loop(num_steps):
7      step = 1.0/num_steps
8      sum = 0
9      for i in prange(num_steps):
10         x= (i+0.5)*step
11         sum = sum + 4.0/(1.0+x*x)
12     return sum
```

Hands on Lab (Assignment)

- Run the python program and compare the elapsed time. Did you get a speed-up?
- Compare the Python runtime with C code runtime.

```
(python-hpc-env) ai@ubuntu-20-04:~/Lab/python$ python3 pi_numba_nopython_parallel.py
(Before compilation) Pi with 100000000 steps is 0.000000 in 0.386362 secs
(After compilation) Pi with 100000000 steps is 3.141593 in 0.049314 secs
```

Now, Python is 20 times faster than C code!! Yeah!!

Python on HPC Contd.

CSCI-4850/5850 High-Performance Computing

Spring 2021

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

How to make your Python Code fast?

You can consider many things, but below considerations will be good starting:

- Speeding up Python with Numpy
- Speeding up Python with PyPy
- Speeding up Python with Cython
- Speeding up Python with Numba
- Parallel Programming with
  - mpi4Py (distributed memory)
  - multiprocessing (shared memory)
  - PyCUDA, PyOPENCL, CuPy (GPU)
  - Dask (flexible library for parallel computing in Python)

mpi4py

- This package provides Python bindings for the **Message Passing Interface** (MPI) standard. It is implemented on top of the MPI-1/2/3 specification and exposes an API which grounds on the standard MPI-2 C++ bindings.
- **Features (https://mpi4py.readthedocs.io/en/stable/)**
  - Convenient communication of any *picklable* Python object
    - point-to-point (send & receive)
    - collective (broadcast, scatter & gather, reductions)
  - Fast communication of Python object exposing the *Python buffer interface* (NumPy arrays, builtin bytes/string/array objects)
  - Process groups and communication domains
  - Parallel input/output:
  - Dynamic process management
  - One-sided operations

Install mpi4py

https://mpi4py.readthedocs.io/en/stable/install.html#

Note

If the **mpicc** compiler wrapper is not on your search path (or if it has a different name) you can use **env** to pass the environment variable `MPICC` providing the full path to the MPI compiler wrapper executable:

```
$ [sudo] env MPICC=/path/to/mpicc pip install mpi4py

$ [sudo] env MPICC=/path/to/mpicc easy_install mpi4py
```

You don't need to install this package. The package was installed on system by a super-user. So, deactivate your environment and let us test. How to deactivate?

```
$ deactivate
```

Example: MPI HELLO WORLD

- Write hello_mpi.py as follows.

```
1 #hello_mpi.py
2 from mpi4py import MPI
3
4 comm = MPI.COMM_WORLD
5 rank = comm.Get_rank()
6 size = comm.Get_size()
7
8 print("hello world from process %d/%d" %(rank,size))
```

Example: MPI HELLO WORLD

- How to run?

```
ai@ubuntu-20-04:~/Lab/python$ python3 hello_mpi.py
hello world from process 0/1
ai@ubuntu-20-04:~/Lab/python$
```

- How to run in parallel? How do I set a fixed number of processes in mpi4py?

Running Python scripts with MPI

Most MPI programs can be run with the command **mpiexec**. In practice, running Python programs looks like:

```
$ mpiexec -n 4 python script.py
```

```
ai@ubuntu-20-04:~/Lab/python$ mpirun -np 4 python3 hello_mpi.py
hello world from process 0/4
hello world from process 3/4
hello world from process 2/4
hello world from process 1/4
ai@ubuntu-20-04:~/Lab/python$
```

Example: MPI HELLO WORLD

- How to run in parallel? Same as the MPI!

```
ai@ubuntu-20-04:~/Lab/python$ mpirun -np 16 python3 hello_mpi.py
hello world from process 15/16
hello world from process 2/16
hello world from process 8/16
hello world from process 11/16
hello world from process 0/16
hello world from process 3/16
hello world from process 1/16
hello world from process 12/16
hello world from process 13/16
hello world from process 6/16
hello world from process 7/16
hello world from process 9/16
hello world from process 4/16
hello world from process 5/16
hello world from process 14/16
hello world from process 10/16
```

Point-to-Point

- The following example “hello_p2p.py” shows the basic point-to-point communication, send and recv.

```python
#hello_p2p.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    for i in range(1, size):
        sendMsg = "Hello, Rank %d" %i
        comm.send(sendMsg, dest=i)
else:
    recvMsg = comm.recv(source=0)
    print(recvMsg)
```

```
ai@ubuntu-20-04:~/Lab/python$ mpirun -np 4 python3 hello_p2p.py
Hello, Rank 1
Hello, Rank 2
Hello, Rank 3
```

Collective Communications - Broadcast

- The following example “hello_bcast.py” shows the basic collective communication, bcast. Check other functions https://mpi4py.readthedocs.io/en/stable/tutorial.html

```
1 #hello_bcast.py
2 from mpi4py import MPI
3
4 comm = MPI.COMM_WORLD
5 rank = comm.Get_rank()
6 size = comm.Get_size()
7
8 if rank == 0:
9     data = {'key1' : [7, 2.72, 2+3j],
10             'key2' : ( 'abc', 'xyz')}
11 else:
12     data = None
13 data = comm.bcast(data, root=0)
14 print("Proc %d has %s" %(rank, data))
```

Collective Communications - Broadcast

- The following example “hello_bcast.py” shows the basic collective communication, bcast. Check other functions https://mpi4py.readthedocs.io/en/stable/tutorial.html

```
1 #hello_bcast.py
2 from mpi4py import MPI
3
4 comm = MPI
5 rank = com
6 size = com
7
8 if rank ==
9     data =
10
11 else:
12     data =
13 data = comm.bcast(data, root=0)
14 print("Proc %d has %s" %(rank, data))
```

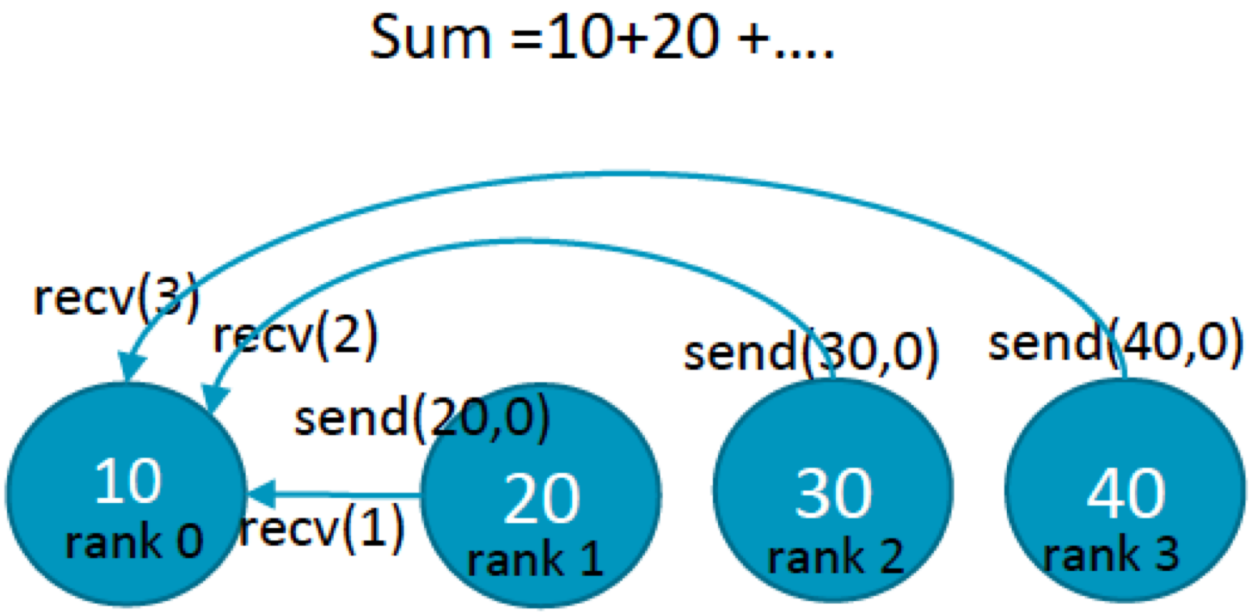
```
ai@ubuntu-20-04:~/Lab/python$ mpirun -np 4 python3 hello_bcast.py
Before Bcast: Proc 2 has None
Before Bcast: Proc 1 has None
Before Bcast: Proc 0 has {'key1': [7, 2.72, (2+3j)], 'key2': ('abc', 'xyz')}
After Bcast: Proc 0 has {'key1': [7, 2.72, (2+3j)], 'key2': ('abc', 'xyz')}
After Bcast: Proc 1 has {'key1': [7, 2.72, (2+3j)], 'key2': ('abc', 'xyz')}
After Bcast: Proc 2 has {'key1': [7, 2.72, (2+3j)], 'key2': ('abc', 'xyz')}
Before Bcast: Proc 3 has None
After Bcast: Proc 3 has {'key1': [7, 2.72, (2+3j)], 'key2': ('abc', 'xyz')}
```

Example: MPI Sum via P2P

```python
#sum_p2p.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank=comm.Get_rank()
size=comm.Get_size()

val = (rank+1)*10
print("Rank %d has value %d" %(rank, val))
if rank == 0:
    sum = val
    for i in range(1,size):
        sum += comm.recv(source=i)
    print("Rank 0 worked out the total %d" %sum)
else:
    comm.send(val, dest=0)
```



https://www.nesi.org.nz/sites/default/files/mpi-in-python.pdf

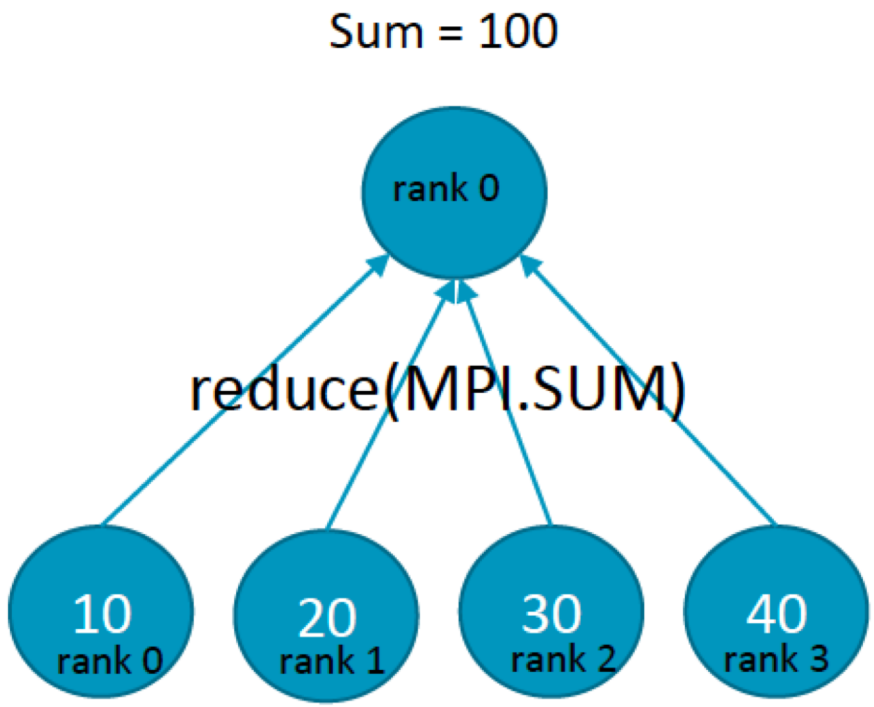
```
ai@ubuntu-20-04:~/Lab/python$ mpirun -np 4 python3 sum_p2p.py
Rank 1 has value 20
Rank 0 has value 10
Rank 3 has value 40
Rank 2 has value 30
Rank 0 worked out the total 100
ai@ubuntu-20-04:~/Lab/python$
```

Example: MPI Sum via Collective Reduce

```python
#sum_reduce.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

val = (rank+1)*10
print("Rank %d has value %d" %(rank, val))
sum = comm.reduce(val, op=MPI.SUM, root=0)
if rank == 0:
    print("Rank 0 worked out the total %d" %sum)
```



https://www.nesi.org.nz/sites/default/files/mpi-in-python.pdf

```
ai@ubuntu-20-04:~/Lab/python$ mpirun -np 4 python3 sum_reduce.py
Rank 1 has value 20
Rank 2 has value 30
Rank 0 has value 10
Rank 3 has value 40
Rank 0 worked out the total 100
```

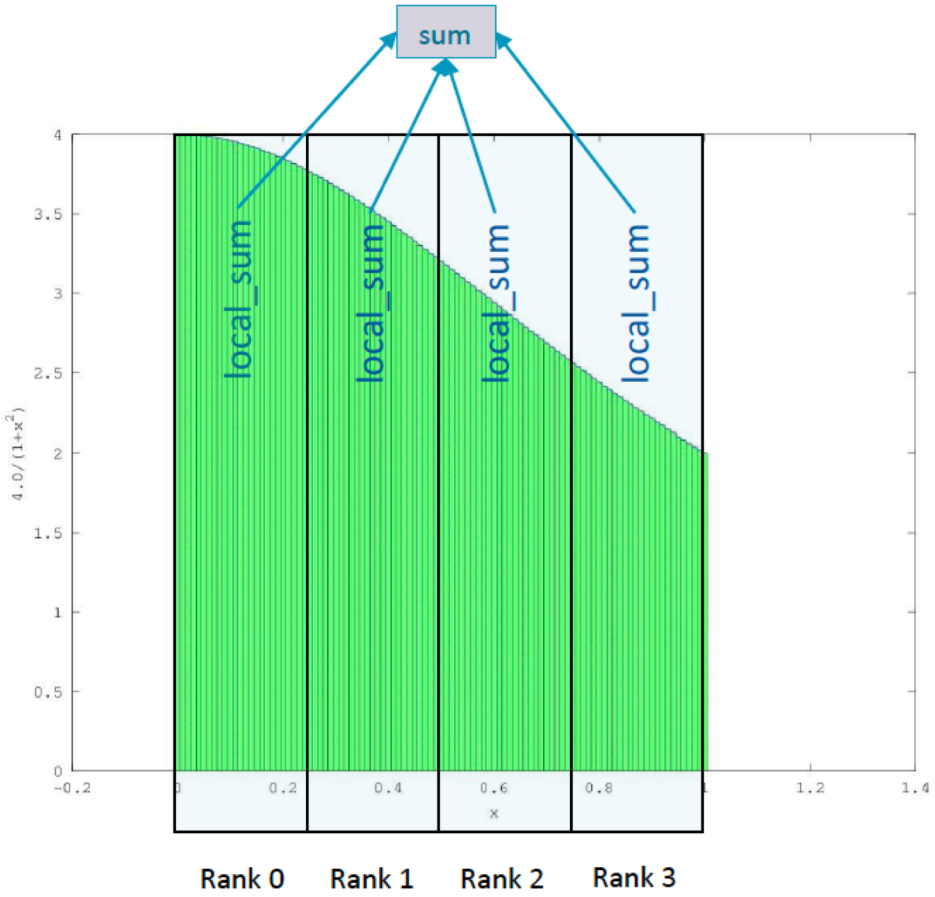
PI calculation

- Let's revisit pi_numba.py
- We have identified the "for" loop was the bottleneck and used NUMBA to make it fast

```
1 #pi_numba.py
2 import time
3 from numba import jit
4
5 @jit
6 def loop(num_steps):
7     step = 1.0/num_steps
8     sum = 0
9     for i in range(num_steps):
10        x= (i+0.5)*step
11        sum = sum + 4.0/(1.0+x*x)
12    return sum
13
14 def Pi(num_steps):
15    begin = time.time()
16    sum = loop(num_steps)
17    pi = sum/num_steps
18    end = time.time()
19    elapsed = end-begin
20    print("Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))
21
22 if __name__ == '__main__':
23    Pi(100000000)
```

PI calculation using mpi4py

- Suppose we wish to parallelize this with 4 processes. We will allocate "*num_steps*/4" steps to each process, such that
  - Steps [0..num_steps/4] allocated to Rank 0
  - Steps [num_steps/4..2*num_steps/4] allocated to Rank 1
  - Steps [2*num_steps/4..3*num_steps/4] allocated to Rank 2
  - Steps [3*num_steps/4..num_steps] allocated to Rank 3



https://www.nesi.org.nz/sites/default/files/mpi-in-python.pdf

PI calculation using mpi4py: pi_mpi4py.py

- Step 1: Delete Numba commands and modify function loop( ) to specify begin and end steps

```
1 #pi_mpi4py.py
2 import time
3 from mpi4py import MPI
4
5 def loop(num_steps, begin, end):
6     step = 1.0/num_steps
7     sum = 0
8     for i in range(begin, end):
9         x= (i+0.5)*step
10        sum = sum + 4.0/(1.0+x*x)
11    return sum
12
```

PI calculation using mpi4py

- Step 2: Add MPI

```
13 def Pi(num_steps):
14     comm = MPI.COMM_WORLD
15     rank = comm.Get_rank()
16     size = comm.Get_size()
17
```

PI calculation using mpi4py

- Step 3: Decompose the problem
  - The modified code makes each process compute “local_sum” from the allocated steps.
  - These “local_sum”s from processes will need to be collected and added up to get the total “sum”.

```
15 def Pi(num_steps):
16     comm = MPI.COMM_WORLD
17     rank = comm.Get_rank()
18     size = comm.Get_size()
19
20     begin = time.time()
21     num_steps_chunk = num_steps/size
22     local_sum = loop(num_steps,
```

Homework

PI calculation using mpi4py

- Step 4: Collect Results
  - You may choose either approach – “send/recv” or “reduce”, it is advisable to use “reduce”. It is simpler, more efficient and it scales better.

```
15 def Pi(num_steps):
16     comm = MPI.COMM_WORLD
17     rank = comm.Get_rank()
18     size = comm.Get_size()
19
20     begin = time.time()
21     num_steps_chunk = num_steps/size
22     local_sum =  Homework
23     #print("local sum of processor %d is %f" %(rank, local_sum))
24     sum =  Homework
25     end = time.time()
26
27     if rank == 0:
28         pi = sum/num_steps
29         elapsed = end-begin
30         print("Pi with %d steps is %f in %f secs" %(num_steps, pi, elapsed))
31
```

PI calculation using mpi4py

- Step 5: How does it scale?

```
ai@ubuntu-20-04:~/Assignments/week4$ mpirun -np 1 python3 pi_mpi4py_sol.py
Pi with 100000000 steps is 3.141593 in 11.019979 secs
ai@ubuntu-20-04:~/Assignments/week4$ mpirun -np 2 python3 pi_mpi4py_sol.py
Pi with 100000000 steps is 3.141593 in 5.622623 secs
ai@ubuntu-20-04:~/Assignments/week4$ mpirun -np 4 python3 pi_mpi4py_sol.py
Pi with 100000000 steps is 3.141593 in 2.994901 secs
ai@ubuntu-20-04:~/Assignments/week4$ mpirun -np 8 python3 pi_mpi4py_sol.py
Pi with 100000000 steps is 3.141592 in 2.113658 secs
ai@ubuntu-20-04:~/Assignments/week4$ mpirun -np 16 python3 pi_mpi4py_sol.py
Pi with 100000000 steps is 3.141592 in 1.510356 secs
```

- Try –n 1, 2, 4, 8, 16. How does it scale? → Homework

- Try –n 1, 2, 4, 8, 16 using increased the number of iterations 10 times bigger (1,000,000,000). How does it scale? → Homework

Jupyte Notebook

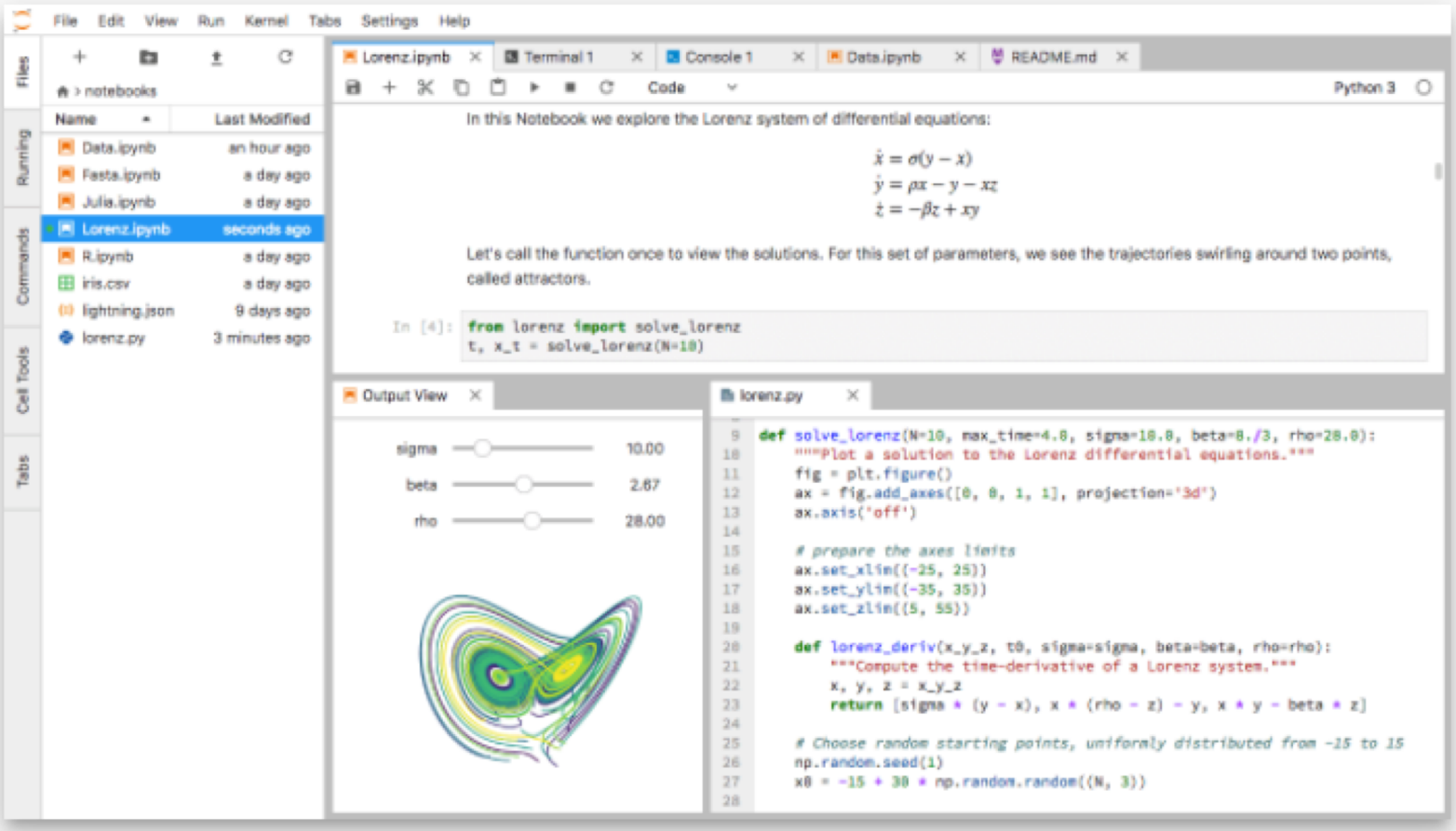
- https://jupyter.org/

Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.

Jupyter Lab

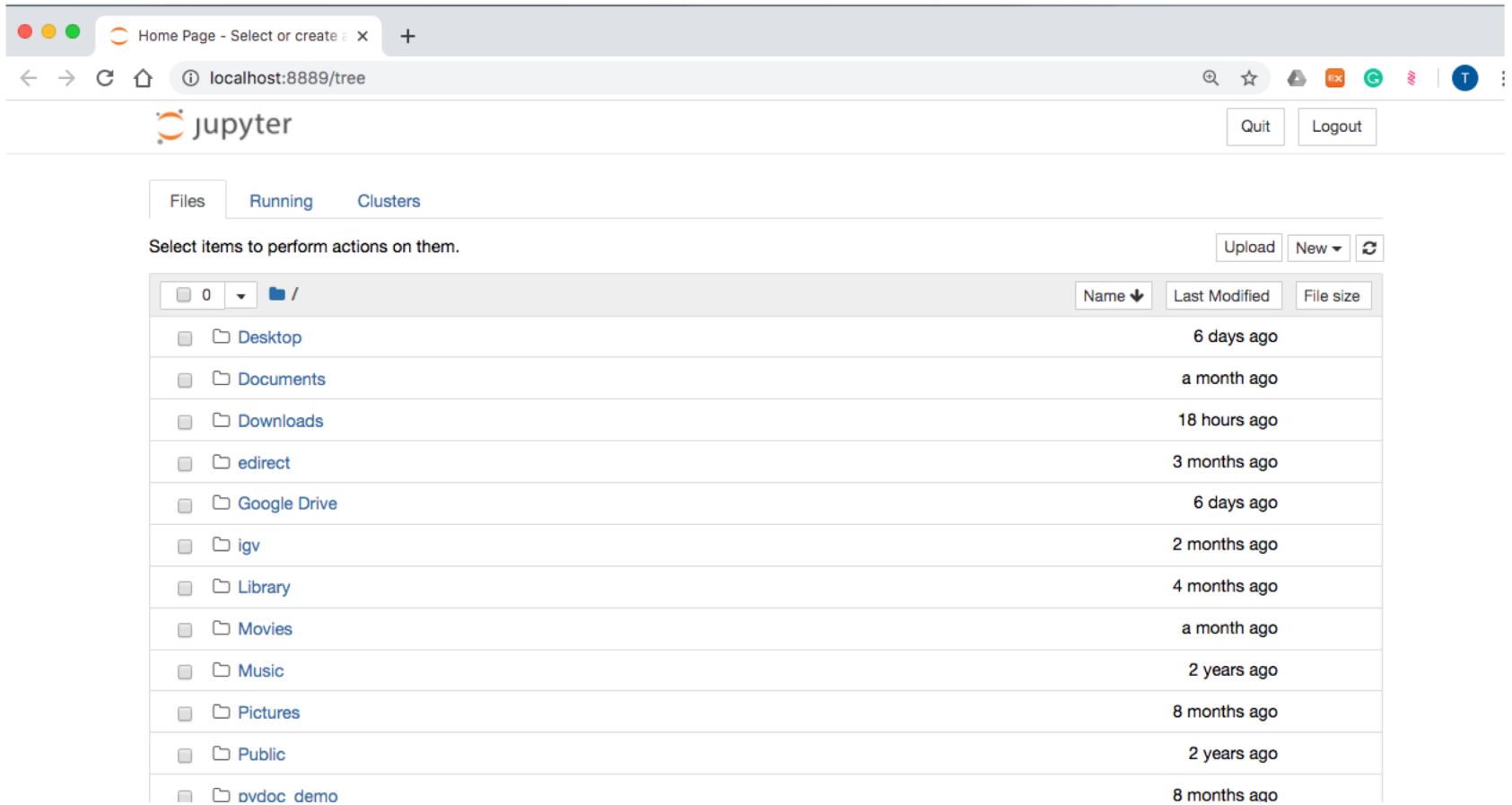
- https://jupyterlab.readthedocs.io/en/stable/

JupyterLab is the next-generation web-based user interface for Project Jupyter. Try it on Binder.
JupyterLab follows the Jupyter Community Guides.



Run Jupyter Notebook (Jupyter Lab)

- Install and run
  - Jupyter notebook: just type “jupyter notebook” and it will open a default web browser



How to run the program on GIST VM via Jupyter Notebook?

- https://techtalktone.wordpress.com/2017/03/28/running-jupyter-notebooks-on-a-remote-server-via-ssh/
- Remote (Server: our case is GIST VM):
  - $ jupyter notebook password
  - $ jupyter notebook --no-browser --port=8887
    - Be careful: Each user will use a unique port. If someone is using the port, change the 8887 to something else.
- Local (Mac or Windows):
  - $ ssh -N -L localhost:8888:localhost:8887 ai@210.125.85.148
  - Open a browser and type localhost:8888

How to run the program on Hopper via Jupyter Notebook?