PYTHON ON ARCHER2

Caoimhín Laoide-Kemp

epcc
THE UNIVERSITY OF EDINBURGH

Overview

- Python Background
- Useful modules
- Parallelism in Python
- Python Implementation on ARCHER2
- Running Python on Login Nodes
- Running Python on Compute Nodes
- Managing your Python Environment on ARCHER2

Why use Python?

- Easy to write simple/readable code
- Free and open-source language
- Supports many different programming styles
- Provides many different useful scientific libraries

Uses for Python

- Analysing Data
- Rapid Testing
- Gluing together C/Fortran programs
- Writing full programs

Python

- Interpreted language
- Can create a script file that is executed by the Python interpreter
- Or run the interpreter without a script file to start an interactive Python environment

Python

```
claoide@uan01:~> cat helloworld.py
print ("Hello world!")
```

Python

```
claoide@uan01:~> cat helloworld.py
print ("Hello world!")
claoide@uan01:~> python helloworld.py
Hello world!
```

Python

```
claoide@uan01:~> python
```

Python

```
claoide@uan01:~> python
Python 3.8.5 (default, Aug 24 2020,
19:11:09)
[GCC 9.3.0 20200312 (Cray Inc.)] on linux
Type "help", "copyright", "credits" or
"license" for more information.
>>>
```

Python

```
claoide@uan01:~> python
Python 3.8.5 (default, Aug 24 2020,
19:11:09)
[GCC 9.3.0 20200312 (Cray Inc.)] on linux
Type "help", "copyright", "credits" or
"license" for more information.
>>> print ("Hello world!")
```

Python

```
claoide@uan01:~> python
Python 3.8.5 (default, Aug 24 2020,
19:11:09)
[GCC 9.3.0 20200312 (Cray Inc.)] on linux
Type "help", "copyright", "credits" or
"license" for more information.
>>> print ("Hello world!")
Hello world!
```

Python on ARCHER2

- Recommend using the HPE Cray Python distribution
- Provides Python 3 but NOT Python 2
  - Python 3 is not backwards-compatible with Python 2
- Provides a number of useful packages for scientific computation and data analysis
  - NumPy
  - SciPy
  - mpi4py
  - Dask

NumPy

- Adds support for large multidimensional arrays
- Also provides a suite of functions to operate on those arrays

NumPy

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a.reshape(2,5)
>>> b
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

NumPy

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = a.reshape(2,5)
>>> b
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> c = np.linspace(1,10,4)
>>> c
array([ 1.,  4.,  7., 10.])
```

epcc

THE UNIVERSITY OF EDINBURGH

NumPy

```
>>> b
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

NumPy

```
>>> b
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> b.sum(axis=0)
array([ 5,  7,  9, 11, 13])
```

NumPy

```
>>> b
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> b.sum(axis=0)
array([ 5,  7,  9, 11, 13])
>>> b.cumsum(axis=1)
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35]])
```

SciPy

- Builds on top of NumPy
- Adds many useful algorithms and functions including:
  - FFTs
  - Linear Algebra
  - Signal Processing
  - Statistical
- Includes other useful packages like Pandas and Matplotlib

mpi4py

- Provides the Python bindings for the Message Passing Interface standard

- Supports:
  - Point-to-point communication
  - Custom communication group topologies
  - Parallel I/O
  - One-sided communication

mpi4py

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

mpi4py

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
      data = numpy.arange(100)
      comm.Send(data, dest=1)
```

mpi4py

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data = numpy.arange(100)
    comm.Send(data, dest=1)
elif rank == 1:
    data = numpy.empty(100)
    comm.Recv(data, source=0)
```

Dask

- Parallel computing library
- Provides data collections designed to be used in parallel
- Contains schedulers to allow building of task graphs, which can then be distributed across a cluster (or across threads on a single core)

Dask

```
def double(x):
      return 2*x

def triple(x):
      return 3*x

def add(x,y):
      return x+y
```

Dask

```
x = double(2) # 4
y = triple(2) # 6
z = add(x,y)  # 10
```

Dask

```
x = double(2) # 4
y = triple(2) # 6
z = add(x,y)  # 10
>>> z
10
```

Dask

```
import dask

double = dask.delayed(double)
triple = dask.delayed(triple)
add    = dask.delayed(add)
```

Dask

```
x = double(2) # 4
y = triple(2) # 6
z = add(x,y)  # 10
```

Dask

```
x = double(2) # 4
y = triple(2) # 6
z = add(x,y)  # 10
>>> z
Delayed('add-2dcfff32-f9aa-432d-84db-
a0c58c123ca8')
```

Dask

```
x = double(2) # 4
y = triple(2) # 6
z = add(x,y)  # 10
>>> z
Delayed('add-2dcfff32-f9aa-432d-84db-
a0c58c123ca8')
>>> z.compute()
10
```

Global Interpreter Lock

- Mutex lock preventing multiple threads from executing Python bytecodes simultaneously
- Purpose is to ensure thread safety of the Python interpreter
- Makes thread-based parallelism difficult
- Some operations happen outside the GIL (e.g I/O) and can be safely multithreaded

Running on Front-end

```
module load cray-python
```

- Can then run with either a Python script file or an interactive session
- Make sure to only run short jobs on the front-end!

Running on Front-end

Make sure to load the cray-python module!

```
claoide@uan01:~> which python
/usr/bin/python
```

Running on Front-end

Make sure to load the cray-python module!

```
claoide@uan01:~> which python
/usr/bin/python
claoide@uan01:~> module load cray-python
```

Running on Front-end

Make sure to load the cray-python module!

```
claoide@uan01:~> which python
/usr/bin/python
claoide@uan01:~> module load cray-python
claoide@uan01:~> which python
/opt/cray/pe/python/3.8.5.0/bin/python
```

Python in Submission Scripts

```
#!/bin/bash --login

#SBATCH --name=python_test
#SBATCH --nodes=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=00:10:00

#SBATCH --account=[budget code]
#SBATCH --partition=standard
#SBATCH --qos=standard

# Setup the batch environment
module load epcc-job-env

# Load the Python module
module load cray-python

# Run your Python progamme
python python_test.py
```

Python in Submission Scripts

```
#!/bin/bash --login

#SBATCH --job-name=mpi4py_test
#SBATCH --nodes=1
#SBATCH --tasks-per-node=2
#SBATCH --cpus-per-task=1
#SBATCH --time=0:10:0

#SBATCH --account=[budget code]
#SBATCH --partition=standard
#SBATCH --qos=standard

# Setup the batch environment
module load epcc-job-env

# Load the Python module
module load cray-python

# Run your Python programme
srun python mpi4py_test.py
```

Adding packages

- Any packages not provided by default can be added using `pip`
- By default new packages will be installed in `~/.local` - this is on the `/home` file system and won't be seen by the compute nodes!

Adding packages

- The following commands will allow the compute nodes to see your newly installed packages:

```
export PYTHONUSERBASE=/work/t01/t01/auser/.local
export PATH=$PYTHONUSERBASE/bin:$PATH
```

- Add these to the file `~/.bashrc` to ensure they are set by default on login

- Once this is done, install packages using:

```
pip install --user <package_name>
```

Conda

- conda vs pip
- Create your own Miniconda installation
- Can choose the version of Python you want

Useful Links

- ARCHER2 Service Desk: support@archer2.ac.uk
- Python Documentation: https://docs.archer2.ac.uk/user-guide/python/