Python and Multiprocessing on the FASRC clusters

Learning objectives

- Python using CLI (Command Line Interface)
  - Interactive
  - Sbatch
- Python Package installation
- Python using OOD (Open On Demand)
- Jupyter Notebook
  - Create conda environment (i.e., jupyter kernel)
- Accelerate Python
  - Multiprocessing
  - Other Tools

Python Programming Language

- High-level, general-purpose, and object-oriented programming language with emphasis on code readability and use of significant indentation.
- Ideal for scripting and rapid application development given its dynamic typing, elegant syntax, and automatic memory management (garbage collection).
- Has a comprehensive standard library. Also known as “batteries-included” language.
- Python’s implementation is mostly in C.
  - Python’s core interpreter, CPython, written in C.
- Interpreted language, hence slower than compiled languages, like C and Fortran.
  - Compiled generates executable
  - Interpreted executes instructions directly on the fly without compiling a program into machine language

Python using CLI - Interactive

- Login to Cannon: `ssh <username>@login.rc.fas.harvard.edu`
- Change to a desired location if don't want *$HOME*: `pwd`
- Go to a compute node on the test partition:

```
salloc -p test --nodes=1 --cpus-per-task=2 --mem=12GB --time=00:30:00
```

- Check Python modules available on Cannon:

```
module spider python
```

- Get detailed information on specific module, e.g.:

```
module spider python/3.10.13-fasrc01
```

- Load the latest (usually also the default) Python module:

```
module load python
```

Python using CLI - Interactive

```
# Login to Cannon
ssh <username>@login.rc.fas.harvard.edu

# Change to a desired location if don't want $HOME:
cd <location>

# Go to a compute node on the test partition:
salloc -p test --nodes=1 --cpus-per-task=2 --mem=12GB --time=00:30:00

# Check Python modules available on Cannon:
module spider python

# Get detailed information on specific module, e.g.:
module spider python/3.10.13-fasrc01

# Load the latest (usually also the default) Python module:
module load python
```

Python using CLI - Interactive

- Check Python version: `python --version`
- Invoke Python interpreter: `python`
- Execute Python programming interactively:

```
def square(x):
    """square a number"""
    return x ** 2
for N in range(1, 4):
    print(N, "squared is", square(N))
```

- Exit Python: `exit()`
- Or run a python script interactively: `python myscript.py`

myscript.py available in User_Codes

Python using CLI - sbatch

- FASRC Examples repository - [GitHub - fasrc/User_Codes]

```
ssh <username>@login.rc.fas.harvard.edu
```

```
pwd (cd to a desired location if not $HOME)
git clone git@github.com:fasrc/User_Codes.git
```

```
mkdir python-training
cd python-training
cp -r ../User_Codes/Languages/Python .
cp -r ../User_Codes/Parallel_Computing/Python/Python-Multiprocessing-Tutorial .
```

CLI Training:
https://docs.rc.fas.harvard.edu/wp-content/uploads/2013/10/Getting-started-on-FASRC-clusters-with-CLI.pptx.pdf

Example 1 - Pi using Monte-Carlo

https://github.com/fasrc/User_Codes/tree/master/Languages/Python/Example1

```
cd Python/Example1
sbatch run.sbatch
```

- run.sbatch: Batch-job submission script for queuing the job
- mc_pi.py: Source code for calculating Pi using Monte-Carlo method

```
#!/bin/bash
#SBATCH -J mc_pi               # job name
#SBATCH -o mc_pi.out           # standard output file
#SBATCH -e mc_pi.err           # standard error file
#SBATCH -N 1                   # number of nodes
#SBATCH -c 1                   # number of cores
#SBATCH -p serial_requeue      # partition
#SBATCH -t 0-00:30             # time in D-HH:MM
#SBATCH --mem=4000             # memory in MB

# Load required modules
module load python

# Run program
srun -c 1  python mc_pi.py
```

Example 2 - Figures with LaTex Font

https://github.com/fasrc/User_Codes/tree/master/Languages/Python/Example2

```
cd Python/Example2
sbatch run.sbatch
```

- tex_demo.py: source code for generating figures using LaTex fonts

```
#!/bin/bash
#SBATCH -J tex_demo
#SBATCH -o tex_demo.out
#SBATCH -e tex_demo.err
#SBATCH -c 1
#SBATCH -p serial_requeue
#SBATCH -t 0-00:30
#SBATCH --mem=4000

# Load required modules
module load python/3.10.12-fasrc01
module load texlive/2018.06.15-fasrc01

# Run program
srun -c 1python tex_demo.py
```

Python Package Installation

- Go to a compute node on the test partition:

```
salloc -p test --nodes=1 --cpus-per-task=2 --mem=12GB --time=01:00:00
```

- Create a vanilla mamba/conda environment (for multiprocessing exercise):

```
module load python
conda create --prefix=/n/holylabs/LABS/<desired-folder>/multiproc_env
python=3.11 -y
```

- Alternatively, if default *$HOME* is desired, then do following instead:

```
module load python
conda create --name multiproc_env python=3.11 -y
```

- See Python – FASRC DOCS

Python Package Installation

- Activate conda/mamba environment:

```
source activate /n/holylabs/LABS/<desired-folder>/multiproc_env
```

- Or if $HOME used, then: `source activate multiproc_env`
- Install relevant python packages (Mamba recommended):

```
conda install numpy pandas matplotlib -y
pip install jupyterlab swifter
```

- Always pip install inside a conda environment to avoid package conflicts
- Deactivate the environment: `conda deactivate`
- See https://docs.rc.fas.harvard.edu/kb/python/#Mamba

Python Using Open OnDemand (OOD)

- Open-source web portal to access clusters
- Web-based, no software needs be installed on your local laptop/desktop (except for a modern browser like Google Chrome, Mozilla Firefox)
- Easy to learn and simple to use
- Very similar to desktop applications
- The easiest way to run GUI applications remotely on a cluster
- Safari is not recommended for OOD
- OOD Training: https://docs.rc.fas.harvard.edu/wp-content/uploads/2013/10/Getting-started-on-FASRC-clusters-with-OOD-May2024.pdf

How to access OOD on FASRC Clusters

- Accessing OOD from Cannon
  - Connect to FASRC VPN - Virtual Desktop (VDI) through Open OnDemand – FASRC DOCS
  - Then go to https://rcood.rc.fas.harvard.edu
- Accessing OOD from FASSE
  - Connect to FASSE VPN - FASSE VDI Apps – FASRC DOCS
  - Then go to https://fasseood.rc.fas.harvard.edu

Filling a form to launch an app

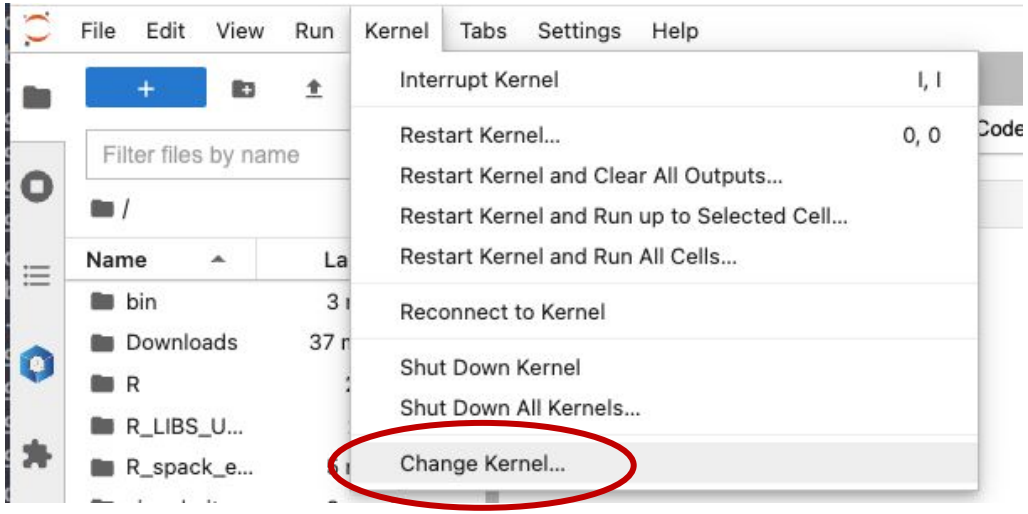
- Request the resources that you need
  (If you don't know for a first trial run, use similar resources as your laptop/desktop)
  - Partition (Name): depends on Cannon (URL) vs FASSE (URL)
  - Memory (RAM): amount of memory in GB
  - Number of cores: recommended at least 2
  - Number of GPUs: if >= 1, make sure you **select** a gpu partition
  - Allocated time: time you would like your session to run

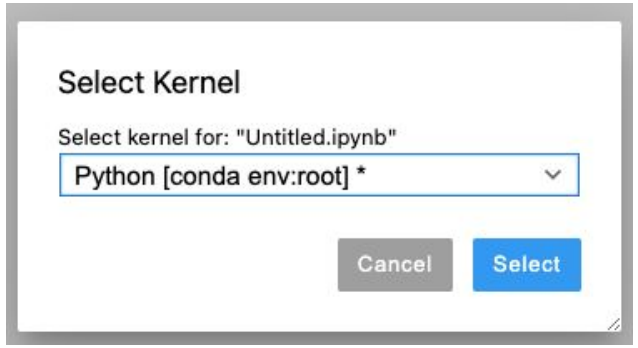
  } the minimum and/or maximum values of each field depends on the selected partition

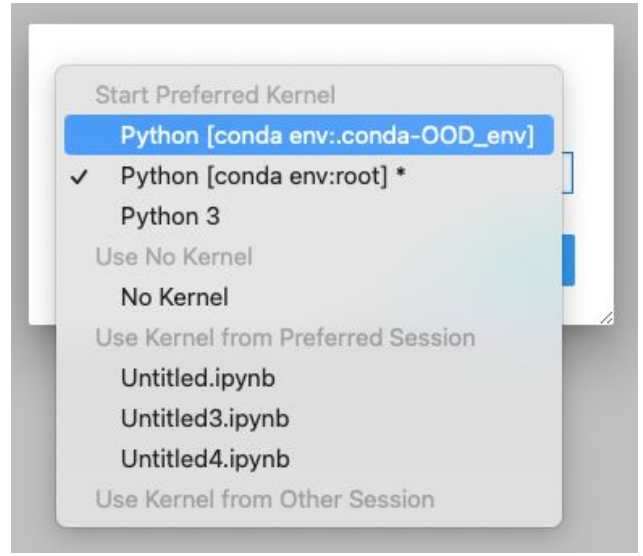
  - Email for status notification: to know when job starts, ends
  - Reservation: if you have a special reservation (this requires approval from FASRC)
  - Account: use this if you have more than one PI_lab affiliation

Jupyter Notebook

- Launch **new** Jupyter Notebook session (existing session will not work!)
- Select newly created conda environment as the kernel
  a. Open a notebook
  b. On the top menu, click Kernel -> Select Kernel -> Click on OOD_env
  c. Note: kernels is the same as conda, python, mamba environment







Closing running OOD windows/tabs

- In most OOD apps, you can close the browser tab while the code is running, and the code will continue to run on the background
- Jupyter Notebook will not! The cell that is running will lose the data and output files will not be written
  - Solution: run Remote Desktop app and launch Jupyter Notebook from within Remote Desktop
  - Documentation: Open OnDemand (OOD/VDI) Remote Desktop: How to open software – FASRC DOCS

FASSE proxy

Documentation: FASSE Proxy Settings – FASRC DOCS

- You may need to set FASSE proxy on
  - Firefox (web browsing)
  - Jupyter Notebook
  - Access Github
  - (Basically, anything outside of FASSE)

Multiprocessing in Python

- Ability of a system to run multiple processors at one time
- Allows several processes to run simultaneously
- Multiprocessing module allocates tasks to different processors and makes better use of a multi-core machine
- Different from threading, which is subject to Global Interpreter Local (GIL) and one thread can execute only one Python code
- Multiprocessing vs Threading Python - Stack Overflow &
- multiprocessing — Process-based parallelism — Python 3.12.4 documentation

Multiprocessing in Python

- On the cluster, difference between number of CPUs allocated to the job vs total number of CPUs available on the node
- Go to a compute node on the test partition requesting 10 cores:

```
salloc -p test --nodes=1 --cpus-per-task=10 --mem=12GB --time=01:00:00
```

- See total number of cores available on the node:

```
scontrol show node <nodename>
```

- Execute cpu-count.py to see which command gives you the number of cores allocated to your job:

```
cd Python-Multiprocessing-Tutorial
python cpu-count.py
```

- See How to find out the number of CPUs using python - Stack Overflow

Multiprocessing - Process-based Parallelism - Basic

- Multiprocessing in Python - MachineLearningMastery.com
- Two functions declared to execute print statements after sleeping for 2 & 3 seconds, resp.
- 3 processes created using *multiprocessing.Process* inside *main()*
- The *Process()* utilizes *target* argument to run target process
- Processes are run using *start()*
- Use *join()* to run & exit a processes before the main program process

```
import multiprocessing
import time
def worker():
    name = multiprocessing.current_process().name
    print(name, 'Starting')
    time.sleep(2)
    print(name, 'Exiting')
def my_service():
    name = multiprocessing.current_process().name
    print(name, 'Starting')
    time.sleep(3)
    print(name, 'Exiting')
if __name__ == '__main__':
    service = multiprocessing.Process(name='my_service', target=my_service)
    worker_1 = multiprocessing.Process(name='worker 1', target=worker)
    worker_2 = multiprocessing.Process(target=worker)
    worker_1.start()
    worker_2.start()
    service.start()
```

Multiprocessing - Pooling

- Run 1000 processes together - may not be possible
- Create a process pool to limit number of processes that can be run at a time
- Function declared to return the cube
- The *multiprocessing.Process* doesn't work with p.start() & p.join(), would need an output queue as well. But faster than *Pool()*
- The *multiprocessing.Pool* module easier to use, returns ordered result using *pool.map()*, & causes less overhead
- See Python multiprocessing: How to know to use Pool or Process? - Stack Overflow

```
import multiprocessing
import time
import os

def cube(x):
    return x**3

if __name__ == '__main__':

  # The Process class
  processes = [multiprocessing.Process(target=cube, args=(x,)) for x in
range(1,len(os.sched_getaffinity(0)))]
  [p.start() for p in processes]
  result_process = [p.join() for p in processes]

  # The Pool class
  pool = multiprocessing.Pool(processes=len(os.sched_getaffinity(0)))
  result_pool = pool.map(cube, range(1,len(os.sched_getaffinity(0))))]
```

Multiprocessing + Numpy

- Using Multiprocessing along with Numpy to accelerate python program
- Go to OOD (Cannon or FASSE) & launch JupyterLab notebook on *test* with
  - *52 CPUs*
  - *gcc/12.2.0-fasrc01* loaded as a module
  - *multiproc_env* loaded as a kernel
  - In *python-training/Python-Multiprocessing-Tutorial*
- Problem Statement:
  - A sample data file has 4 columns and 1000 entries. Columns correspond to the time a job was submitted, when it started, when it ended, and number of CPUs allocated.
  - Calculate the total number of CPUs in use by currently running jobs for every submitted job

Multiprocessing + Numpy

- Convert numerical columns to Numpy arrays.
- Declare a function to calculate CPUs utilized: *calculate_cpus_utilized()*
- Multiple methods utilized for the calculation:
  - Use the function over each submitted-job entry
    - Pandas *apply()*
    - swifter.apply()
  - Using Numpy arrays & for-loop
  - Using Multiprocessing with a pool of processes = #CPUs requested for OOD job
- Run the notebook to see which method gives the fastest result
- Fastest: Combination of Numpy and Multiprocessing

Accelerate Python - Other Tools

- Numba
  - https://numba.pydata.org/
- Swifter
  - Speed up your Pandas Processing with Swifter | by Cornellius Yudha Wijaya | Towards Data Science
  - GitHub - jmcarpenter2/swifter: A package which efficiently applies any function to a pandas dataframe or series in the fastest available manner
- Dask
  - https://www.dask.org/

Other Training(s) on Python

Training portal: https://trainingportal.harvard.edu/Saba/Web_spf/NA1PRD0068/app/dashboard

Data Handling in Python Workshop:
https://github.com/HarvardRC/Python_Data_Handling

- Requirement: working FASRC account with cluster access
- Audience
  - Users familiar with command-line interface
  - New to Cannon and FASSE, but familiar with Python
- Content
  - Python Data Types: Focus on Collections
  - File I/O Operations: Emphasis on profiling & performance
  - Parallel Processing

FASRC Upcoming Trainings

Training calendar: https://www.rc.fas.harvard.edu/upcoming-training/

Parallel Job Workflows

This training would focus on best practices for running parallel workflows on FASRC clusters. The module would provide the basic knowledge to execute OpenMP and MPI applications efficiently on the cluster.

Objectives:

- Best Practices
- Brief Introduction to Parallel Computing
- Embarrassingly Parallel Jobs/Workflows
- OpenMP Jobs/Workflows
- MPI Jobs/Workflows
- Hybrid (MPI+OpenMP) Jobs/Workflows

FASRC documentation

- FASRC docs: https://docs.rc.fas.harvard.edu/
- GitHub User_codes: https://github.com/fasrc/User_Codes/
- Getting help
  - Office hours: https://www.rc.fas.harvard.edu/training/office-hours/
  - Ticket
    - Portal: http://portal.rc.fas.harvard.edu/rcrt/submit_ticket (requires login)
    - Email: rchelp@rc.fas.harvard.edu

Survey

Please, fill out our course survey. Your feedback is essential for us to improve our trainings!!

http://tinyurl.com/FASRCsurvey

Thank you :)

FAS Research Computing