Scalable and Computationally Reproducible Approaches to Arctic Research

Matt Jones, Bryce Mecum, Jeanette Clark, Sam Csik

September 19, 2022

Table of contents

Preface

About

This 5-day in-person workshop will provide researchers with an introduction to advanced topics in computationally reproducible research in python and R, including software and techniques for working with very large datasets. This includes working in cloud computing environments, docker containers, and parallel processing using tools like parsl and dask. The workshop will also cover concrete methods for documenting and uploading data to the Arctic Data Center, advanced approaches to tracking data provenance, responsible research and data management practices including data sovereignty and the CARE principles, and ethical concerns with data-intensive modeling and analysis.

Schedule

Code of Conduct

Please note that by participating in this activity you agree to abide by the NCEAS Code of Conduct.

	Monday	Tuesday	Wednesday	Thursday	Friday
08:00-08:30	Coffee (optional)	Coffee (optional)	Coffee (optional)	Coffee (optional)	Coffee (optional)
08:30-09:00 09:00-09:30	**1. Welcome and Course Overview** (Matt)	**6. Group project I Data staging and pre-processing** (Jeanette)	**10. Spatial and Image Data using GeoPandas** (Jeanette)	**15. Google Earth Engine** (Ingmar, Sam)	**19. What is cloud computing anyways?** (Matt)
09:30-10:00 10:00-10:30	**2. Remote computing** (Sam)		**11. Data futures: Parquet and Arrow** (Jeanette)		
10:30-11:00	BREAK	BREAK	BREAK	BREAK	BREAK
11:00-11:30	**3. Python programming on clusters** (Jeanette)	**7. Software design I** (Matt)	**12. Software Design II** (Carmen)	**16. Billions of Ice Wedge Polygons** (Chandi)	**20. Reproducibility redux via containers** (Matt)
11:30-12:00					**Survey Feedback** **Q & A**
12:00-12:30 12:30-13:00	Lunch	Lunch	Lunch	Lunch	Adjourn
13:00-13:30 13:30-14:00 14:00-14:30 14:30-15:00	**4. Pleasingly Parallel Programming** (Matt)	**8. Data structures and formats for large data** (Carmen)	**13. Group project II Parallel data processing** (Jeanette)	**17. Group project III Visualizing big geospatial data** (Jeanette)	
15:00-15:30	Break	Break	Break	Break	
15:30-16:00	**5. Documenting and Publishing Data** (Daphne)	**9. Parallelization with Dask** (Carmen)	**14. Data Ethics** (Tash)	**18. Workflows for data staging and publishing** (Jeanette)	
16:00-16:30			**Breather** Catch-up		
16:30-17:00	Q&A	Q&A	Q&A	Q&A	

Setting Up

In this course, we will be using Python (3.9.13) as our primary language, and VS Code as our IDE. Below are instructions on how to get VS Code set up to work for the course. If you are already a regular Python user, you may already have another IDE set up. We strongly encourage you to set up VS Code with us, because we will use your local VS Code instance to write and execute code on one of the NCEAS servers.

Download VS Code and Remote - SSH Extension

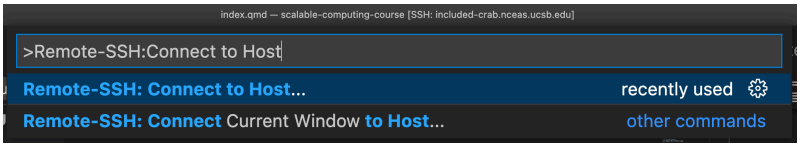
First, download VS Code if you do not already have it installed.

You'll also need to download the Remote - SSH extension.

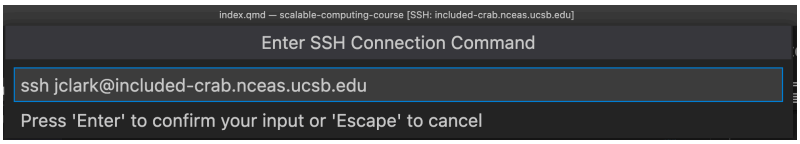
Log in to the server

To connect to the server using VS Code follow these steps, from the VS Code window:

- open the command pallette (Cmd + Shift + P)
- enter "Remote SSH: Connect to Host"

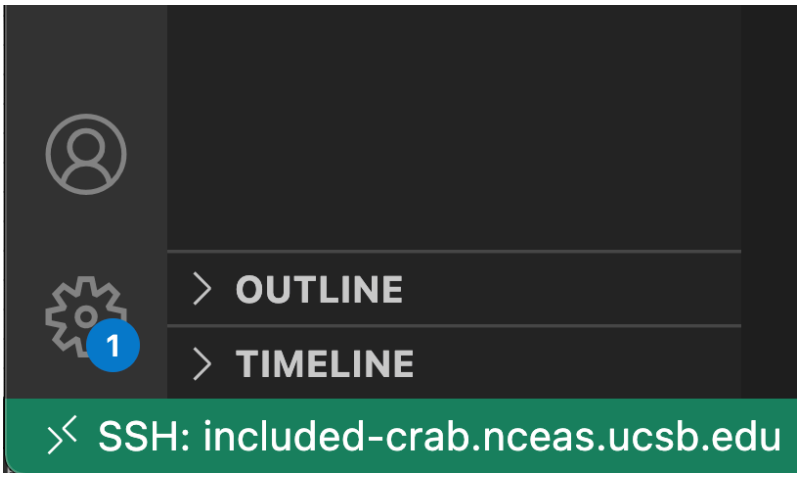


- select "Add New SSH Host"
- enter the ssh command to connect to the host as if in a terminal (`ssh username@included-crab.nceas.ucsb.edu`)
  - Note: you will only need to do this step once



- select the SSH config file to update with the name of the host. You should select the one in your user directory (eg: `/Users/jclark/.ssh/config`)
- click "Connect" in the popup in the lower right hand corner
    - Note: If the dialog box does not appear, reopen the command palette (Cmd + Shift + P), type in "Remote-SSH: Connect to Host…", choose included-crab.nceas.ucsb.edu from the options of configured SSH hosts, then enter your password into the dialog box that appears
- enter your password in the dialog box that pops up

When you are connected, you will see in the lower left hand corner of the window a green bar that says "SSH: included-crab.nceas.ucsb.edu."



Install extensions on the server

After connecting to the server, in the extensions pane (View > Extensions) search for, and install, the following extensions:

```
- Python
- Jupyter
- Jupyter Keymap
```

Note that these extensions will be installed on the server, and not locally.

Test your local setup (Optional)

We are going to be working on the server exclusively, but if you are interested in setting up VS Code to work for you locally with Python, you can follow these instructions. This local setup section summarizes the official VS Code tutorial. For more detailed instructions and screenshots, see the source material. This step is 100% optional, if you already have an IDE set up to work locally that you like, or already have VS code set up to work locally, you are welcome to skip this.

Locally (not connected to the server), check to make sure you have Python installed if you aren't sure you do. File > New Window will open up a new VS Code window locally.

To check your python, from the terminal run:

```
python3 --version
```

If you get an error, it means you need to install Python. Here are instructions for getting installed, depending on your operating system. Note: There are many ways to install and manage your Python installations, and advantages and drawbacks to each. If you are unsure about how to proceed, feel free to reach out to the instructor team for guidance.

- Windows: Download and run an installer from Python.org.
- Mac: Install using homebrew. If you don't have homebrew installed, follow the instructions from their webpage.
  - `brew install python3`

After you run your install, make sure you check that the install is on your system PATH by running `python3 --version` again.

Next, install the Python extension for VS Code.

Open a terminal window in VS Code from the Terminal drop down in the main window. Run the following commands to initialize a project workspace in a directory called `training`. This example will show you how to do this locally. Later, we will show you how to set it up on the remote server with only one additional step.

```
mkdir training
cd training
code .
```

Next, select the Python interpreter for the project. Open the **Command Palette** using Command + Shift + P (Control + Shift + P for windows). The Command Palette is a handy tool in VS Code that allows you to quickly find commands to VS Code, like editor commands, file edit and open commands, settings, etc. In the Command Palette, type "Python: Select Interpreter." Push return to select the command, and then select the interpreter you want to use (your Python 3.X installation).

To make sure you can write and execute code in your project, create a Hello World test file.

- From the File Explorer toolbar, or using the terminal, create a file called `hello.py`
- Add some test code to the file, and save

```
msg = "Hello World"
print(msg)
```

- Execute the script using either the Play button in the upper-right hand side of your window, or by running `python3 hello.py` in the terminal.
  - For more ways to run code in VS Code, see the tutorial

Finally, to test Jupyter, download the Jupyter extension. You'll also need to install `ipykernel`. From the terminal, run `pip install ipykernel`.

You can create a test Jupyter Notebook document from the command pallete by typing "Create: New Jupyter Notebook" and selecting the command. This will open up a code editor pane with a notebook that you can test.

Create a (free) Google Earth Engine (GEE) account

In order to code along during the Google Earth Engine lesson (Ch 15) on Thursday, you'll need to sign up for an account at https://signup.earthengine.google.com. Following the link above will take you to a form that looks like this:

Once submitted, you'll receive an email with some helpful links and a message that it may take a few days for your account to be up and running. **Please be sure to do this a few days ahead of needing to use GEE.**

About this book

These written materials reflect the continuous development of learning materials at the Arctic Data Center and NCEAS to support individuals to understand, adopt, and apply ethical open science practices. In bringing these materials together we recognize that many individuals have contributed to their development. The primary authors are listed alphabetically in the citation below, with additional contributors recognized for their role in developing previous iterations of these or similar materials.

Citation: Matthew B. Jones, Bryce Mecum, S. Jeanette Clark, Samantha Csik. 2022. Scalable and Computationally Reproducible Approaches to Arctic Research.

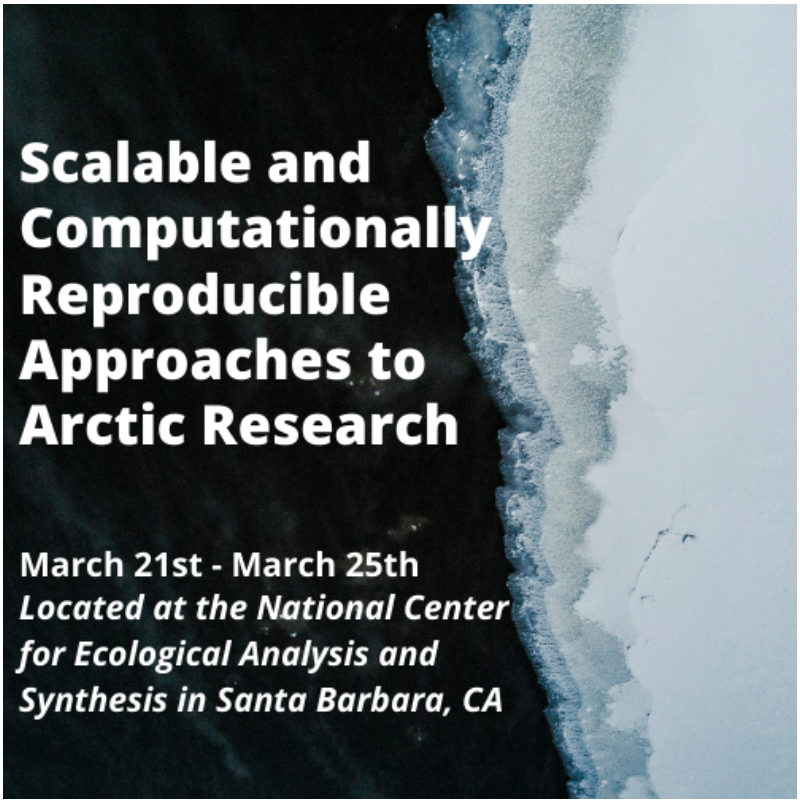
Additional contributors: Amber E. Budden, Natasha Haycock-Chavez, Noor Johnson, Stephanie Hampton, Jim Regetz, Bryce Mecum, Julien Brun, Julie Lowndes, Erin McLean, Andrew Barrett, David LeBauer, Jessica Guo.

This is a Quarto book. To learn more about Quarto books visit https://quarto.org/docs/books.

1 Welcome and Introductions

This course is one of three that we are currently offering, covering fundamentals of open data sharing, reproducible research, ethical data use and reuse, and scalable computing for reusing large data sets.


Reproducible Practices for Arctic Research Using R
February 14th - February 18th, 2022
This course will be taught virtually


Scalable and Computationally Reproducible Approaches to Arctic Research
March 21st - March 25th
Located at the National Center for Ecological Analysis and Synthesis in Santa Barbara, CA

2 Remote Computing

- Understand the basic architecture of computer networks
- Learn how to connect to a remote computer via a shell
- Become familiarized with Bash Shell programming to navigate your computer's file system, manipulate files and directories, and automate processes

2.1 Introduction

Scientific synthesis and our ability to effectively and efficiently work with big data depends on the use of computers and the internet. Working on a personal computer may be sufficient for many tasks, but as data get larger and analyses more computationally intensive, scientists often find themselves needing more computing resources than they have available locally. Remote computing, or the process of connecting to a computer(s) in another location via a network link is becoming more and more common in overcoming big data challenges.

In this lesson, we'll learn about the architecture of computer networks and explore some of the different remote computing configurations that you may encounter, we'll learn how to securely connect to a remote computer via a shell, and we'll become familiarized with using Bash Shell to efficiently manipulate files and directories. We will begin working in the VS Code IDE (integrated development environment), which is a versatile code editor that supports many different languages.

2.2 Servers & Networking

Remote computing typically involves communication between two or more "host" computers. Host computers connect via

networking equipment and can send messages to each other over communication protocols (aka an Internet Protocol, or IP). Host computers can take the role of **client** or **server**, where servers share their resources with the client. Importantly, these client and server roles are not inherent properties of a host (i.e. the same machine can play either role).

- **Client**: the host computer *intiating* a request
- **Server**: the host computer *responding* to a request

> **Note**
>
> Hosts typically have one network address but can have many different ones (for example, adding multiple network cards to a single server increases bandwith).

Fig 1. Examples of different remote computing configurations. (a) A client uses secure shell protocol (SSH) to login/connect to a server over the internet. The client and the server exist in the physical world, but in different locations. (b) A client uses SSH to login/connect to a computing cluster (i.e. a set of computers (nodes) that work together so that they can be viewed as a single system) over the internet. The connection is first made through a gateway node (i.e. a computer that routes traffic from one network to another). The client and the cluster (server) exist in the physical world, but in different locations. (c) A client uses SSH to login/connect to a computing cluser where each node is a virtual machine (VM) hosted by a cloud computing service (e.g. Amazon Web Services, Google Cloud, Microsoft Azure, etc.). The connection is first made through a gateway node. The client and the gateway are located in the physical world, while the VM nodes are hosted in the cloud.

2.3 IP addressing

Hosts are assigned a **unique numerical address** used for all communication and routing called an Internet Protocol Address (IP Address). They look something like this: **128.111.220.7**. Each IP Address can be used to communicate

over various "ports", which allows multiple applications to communicate with a host without mixing up traffic.

Because IP addresses can be difficult to remember, they are also assigned **hostnames**, which are handled through the global Domain Name System (DNS). Clients first look up a hostname in the DNS to find the IP address, then open a connection to the IP address.

> **Note**
>
> Throughout this course, we'll be working on a server with the hostname, **included-crab** and an IP address, 128.111.85.1.

2.4 Bash Shell Programming

What is a shell? From Wikipedia:

> "a computer program which exposes an operating system's services to a human user or other programs. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation."

What is Bash? Bash, or **B**ourne-**a**gain **Sh**ell, is a command line tool (language) commonly used to manipulate files and directories. Accessing and using bash is slightly different depending on what type of machine you work on:

- **Mac:** bash via the Terminal, which comes ready-to-use with all Macs
- **Windows:** bash via Git Bash, which needs to be installed

> **Note**
>
> Mac users may have to switch from Z Shell, or zsh, to bash. Use the command `exec bash` to switch your default shell

to bash (or `exec zsh` to switch back).

2.4.1 Some commonly used (and very helpful) bash commands:

Below are just a few bash commands that you're likely to use. Some may be extended with options (more on that in the next section) or even piped together (i.e. where the output of one command gets sent to the next command, using the `|` operator). You can also find some nice bash cheat sheets online, like this one. Alternatively, the Bash Reference Manual has *all* the content you need, albeit a bit dense.

bash command	what it does
`pwd`	print your current working directory
`cd`	change directory
`ls`	list contents of a directory
`tree`	display the contents of a directory in the form of a tree structure (not installed by default)
`echo`	print text that is passed in as an argument
`mv`	move or rename a file
`cp`	copy a file(s) or directory(ies)
`touch`	create a new empty file
`mkdir`	create a new directory
`rm`/`rmdir`	remove a file/ empty directory (be careful – there is not "trash" folder!)
`grep`	searches a given file(s) for lines containing a match to a given pattern list
`sed`	stands for **S**tream **Ed**itor; a versatile command for editing files
`cut`	extract a specific portion of text in a file
`join`	join two files based on a key field present in both
`top`, `htop`	view running processes in a Linux system (press `Q` to quit)

2.4.2 General command syntax

Bash commands are typically are written as: `command [options] [arguments]` where the command must be an executable on your PATH and where options (settings that change the shell and/or script behavior) take one of two forms: **short form** (e.g. `command -option-abbrev`) or **long form** (e.g. `command --option-name` or `command -o option-name`). An example:

```
# the `ls` command lists the files in a directory
ls file/path/to/directory

# adding on the `-a` or `--all` option lists all files (including hidden files) in a directo
ls -a file/path/to/directory # short form
ls --all file/path/to/directory # long form
ls -o all file/path/to/directory # long form
```

2.5 Connecting to a remote computer via a shell

In addition to navigating your computer/manipulating your files, you can also use a shell to gain accesss to and remotely control other computers. To do so, you'll need the following:

- a remote computer (e.g. server) which is turned on
- client and server ssh clients installed/enabled
- the IP address or name of the remote computer
- the necessary permissions to access the remote computer

Secure Shell, or SSH, is a network communication protocol that is often used for securely connecting to and running shell commands on a remote host, tremendously simplifying remote computing. It is supported out-of-the-box on Linux and Macs via Teminal and on Windows machines via Windows PowerShell.

2.6 Git via a shell

Git, a popular version control systemm and command line tool can be accessed via a shell. While there are lots of graphical user interfaces (GUIs) that faciliatate version control with Git, they often only implement a small subset of Git's most-used functionality. By interacting with Git via the command line, you have access to *all* Git commands. While all-things Git is outside the scope of this workshop, we will use some basic Git commands in the shell to clone GitHub (remote) repositories to the server and save/store our changes to files. A few important Git commands:

Git command	what it does
`git clone`	create a copy (clone) of repository in a new direcotry ad a different location
`git add`	adds a change in the working directory to the staging area
`git commit`	record a snapshot of a repository; the `-m` option adds a commit message
`git push`	send commits from a local repository to a remote repository

2.7 Let's practice!

We'll now use bash commands to do the following:

- connect to the server (**included-crab**) that we'll be working on for the remainder of this course
- navigate through directories on the server and add/change/manipulate files
- clone a GitHub repository to the server
- automate some of the above processes by writing a bash script

2.7.1 Exercise 1: Connect to a server using the `ssh` command (or using VS Code's command palette)

Let's connect to a remote computer (**included-crab**) and practice using some of above commands.

1. Launch a terminal in VS Code

- There are two options to open a terminal window, if a terminal isn't already an open pane at the bottom of VS Code

    a) Click on `Terminal > New Terminal` in top menu bar

    b) Click on the `+ (dropdown menu) > bash` in the bottom right corner

> **Note**
>
> You don't *need* to use the VS Code terminal to ssh into a remote computer, but it's conveniently located in the same window as your code when working in the VS Code IDE.

2. Connect to a remote server

- You can choose to SSH into the server (included-crab.nceas.ucsb.edu) through **(a)** the command line by using the `ssh` command, or **(b)** through VS Code's command palette. If you prefer the latter, please refer back to the **Log in to the server** section. To do so via the command line, use the `ssh` command followed by `yourusername@included-crab.nceas.ucsb.edu`. You'll be prompted to type/paste your password to complete the login. It should look something like this:

```
yourusername:~$ ssh yourusername@included-crab.nceas.ucsb.edu
yourusername@included-crab.nceas.ucsb.edu's password:
yourusername@included-crab:~$
```

> **Important**
>
> You won't see anything appear as you type or paste your password – this is a security feature! Type or paste your password and press enter/return when done to finish connecting to the server.

> **Warning**
>
> DO WE NEED THIS SECTION?
>
> 3. Change your password
>
> ```
> yourusername@included-crab:~$ passwd
> Changing password for yourusername.
> (current) UNIX password:
> Enter new UNIX password:
> Retype new UNIX password:
> ```

> **Note**
>
> To log out of the server, type `exit` – it should look something like this:
>
> ```
> yourusername@included-crab.nceas.ucsb.edu:$ exit
> logout
> Connection to included-crab.nceas.ucsb.edu closed.
> (base) .....
> ```

2.7.2 Exercise 2: Practice using some common bash commands

1. Use the `pwd` command to print your current location, or working directory. You should be in your home directory on the server (e.g. `/home/yourusername`).
2. Use the `ls` command to list the contents (any files or subdirectories) of your home directory

3. Use the `mkdir` command to create a new directory named `bash_practice`:

```
mkdir bash_practice
```

4. Use the `cd` command to move into your new `bash_practice` directory:

```
# move from /home/yourusername to home/yourusername/bash_practice
cd bash_practice
```

- To move *up* a directory level, use two dots, `..` :

```
# move from /home/yourusername/bash_practice back to /home/yourusername
$ cd ..
```

> **Note**
>
> To quickly navigate back to your home directory from wherever you may be on your computer, use a tilde, `~` :
>
> ```
> # e.g. to move from from some subdirectory, /home/yourusername/Projects/project1/data, bac
> $ cd ~
>
> # or use .. to back out three subdirectories
> $ cd ../../..
> ```

5. Add some `.txt` files (`file1.txt`, `file2.txt`, `file3.txt`) to your `bash_practice` subdirectory using the `touch` command (**Note:** be sure to `cd` into `bash_practice` if you're not already there):

```
# add one file at a time
touch file1.txt
touch file2.txt
touch file3.txt

# or add all files simultanously like this:
touch file{1..3}.txt
```

```
# or like this:
touch file1.txt file2.txt file3.txt
```

6. You can also add other file types (e.g. `.py`, `.csv`, etc.)

```
touch mypython.py mycsv.csv
```

7. Print out all the `.txt` files in `bash_practice` using a wildcard, `*`:

```
ls *.txt
```

8. Count the number of `.txt` files in `bash_practice` by combining the `ls` and `wc` (word count) funtions using the pipe, `|`, operator:

```
# `wc` returns a word count (lines, words, chrs)
# the `-l` option only returns the number of lines
# use a pipe, `|`, to send the output from `ls *.txt` to `wc -l`
ls *.txt | wc -l
```

9. Delete `mypython.py` using the `rm` command:

```
rm mypython.py
```

10. Create a new directory inside `bash_practice` called `data` and move `mycsv.csv` into it.

```
mkdir data
mv mycsv.csv ~/bash_practice/data

# add the --interactive option (-i for short) to prevent a file from being overwritten by ac
mv -i mycsv.csv ~/bash_practice/data
```

11. Use `mv` to rename `mycsv.csv` to `mydata.csv`

```
mv mycsv.csv mydata.csv
```

12. Add column headers `col1`, `col2`, `col3` to `mydata.csv` using `echo` + the `>` operator

```
echo "col1, col2, col3" > mydata.csv
```

> **Tip**
>
> You can check to see that `mydata.csv` was updated using GNU nano, a text editor for the command line that comes preinstalled on Linux machines (you can edit your file in nano as well). To do so, use the `nano` command followed by the file you want to open/edit:
>
> ```
> nano mydata.csv
> ```
>
> To save and quit out of nano, use the `control` + `X` keyboard shortcut.
> You can also create and open a file in nano in just one line of code. For example, running `nano hello_world.sh` is the same as creating the file first using `touch hello_world.sh`, then opening it with nano using `nano hello_world.sh`.

13. Append a row of data to `mydata.csv` using `echo` + the `>>` operator

```
# using `>` will overwrite the contents of an existing file; `>>` appends new information to
echo "1, 2, 3" >> mydata.csv
```

2.7.3 Exercise 3: Clone a GitHub repository to the server

IDEs commonly have helper buttons for cloning (i.e. creating a copy of) remote repositories to your local computer (or in this case, a server), but using git commands in a terminal can be just as easy. We can practice that now, following the steps below:

1. Go to the **bash-babynames** repository on GitHub at https://github.com/samanthacsik/bash-babynames

and Fork (make your own copy of the repository) it by clicking on the **Fork** button (top right corner of the repository's page).

2. Once forked, click on the green **Code** button and copy the URL to your clipboard.

3. In the VS Code terminal, use the `git clone` command to create a copy of the bash-babynames repository on the server.

```
git clone <url>
```

4. You should now have a copy of the **bash-babynames** repository to work on on the server. Use the `tree` command to see the structure of the repo (you need to be in the `bash-babynames` directory for this to work) – there should be a respository `README.MD` file, a `KEY.sh` file (this is a functioning bash script available for reference; we'll be recreating it together in the next exercise) and a `namesbystate` folder containing 51 `.TXT` files and a `StateReadMe.pdf` file with some metadata.

2.7.4 Bonus Exercise: Automate data processing with a Bash script

As we just demonstrated, we can use bash commands in the terminal to accomplish a variety of tasks like navigating our computer's directories, manipulating/creating/adding files, and much more. However, writing a bash *script* allows us to gather and save our code for automated execusion.

We just cloned the `bash-babynames` GitHub repository to the server in **Exercise 3** above. This contains 51 `.TXT` files (one for each of the 50 US states + The District of Columbia), each with the top 1000 most popular baby names in that state. We're going to use some of the bash commands we learned in **Exercise 2** to concatenate all rows of data from these 51 files into a single `babynames_allstates.csv` file.

Let's begin by creating a simple bash script that when executed, will print out the message, "Hello, World!" This simple script

will help us determine whether or not things are working as expected before writing some more complex (and interesting) code.

1. Open a terminal window and determine where you are by using the `pwd` command – we want to be in our `bash-babynames` repository. If necessary, navigate here using the `cd` command.
2. Next, we'll create a shell script called `mybash.sh` using the `touch` command:

```
$ touch hello_world.sh
```

3. There are a number of ways to edit a file or script – we'll use Nano, a terminal-based text editor, as we did earlier. Open your `mybash.sh` with nano by running the following in your terminal:

```
$ nano mybash.sh
```

4. We can now start to write our script. Some important considerations:

- Anything following a `#` will not be executed as code – these are useful for adding comments to your scripts
- The first line of a Bash script starts with a **shebang**, `#!`, followed by a path to the Bash interpreter – this is used to tell the operating system which interpreter to use to parse the rest of the file. There are two ways to use the shebang to set your interpreter (read up on the pros & cons of both methods on this Stack Overflow post):

```
# (option a): use the absolute path to the bash binary
#!/bin/bash

# (option b): use the env untility to search for the bash executable in the user's $PATH env
#!/usr/bin/env bash
```

5. We'll first specify our bash interpreter using the shebang, which indicates the start of our script. Then, we'll use the

`echo` command, which when executed, will print whatever text is passed as an argument. Type the following into your script (which should be opened with nano), then save (Use the keyboard shortcut `control` + `X` to exit, then type `Y` when it asks if you'd like to save your work. Press `enter/return` to exit nano).

```
# specify bash as the interpreter
#!/bin/bash

# print "Hello, World!"
echo "Hello, World!"
```

6. To execute your script, use the `bash` command followed by the name of your bash script (be sure that you're in the same working directory as your `mybash.sh` file or specify the file path to it). If successful, "Hello, World!" should be printed in your terminal window.

```
bash mybash.sh
```

7. Now let's write our script. Re-open your script in nano by running `nano mybash.sh`. Using what we practiced above and the hints below, write a bash script that does the following:

- prints the number of `.TXT` files in the `namesbystate` subdirectory
- prints the first 10 rows of data from the `CA.TXT` file (HINT: use the `head` command)
- prints the last 10 rows of data from the `CA.TXT` file (HINT: use the `tail` command)
- creates an empty `babynames_allstates.csv` file in the `namesbystate` subdirectory (this is where the concatenated data will be saved to)
- adds the column names, `state`, `gender`, `year`, `firstname`, `count`, in that order, to the `babynames_allstates.csv` file
- concatenates data from all `.TXT` files in the `namesbystate` subdirectory and appends those data to the `babynames_allstates.csv` file (HINT: use the `cat` function to concatenate files)

Here's a script outline to fill in (**Note:** The `echo` statements below are not necessary but can be included as progress indicators for when the bash script is executed – these also make it easier to diagnose where any errors occur during execution.):

```
#!bin/bash
echo "THIS IS THE START OF MY SCRIPT!"

echo "-----Verify that we have .TXT files for all 50 states + DC-----"
<add your code here>

echo "-----Printing head of CA.TXT-----"
<add your code here>

echo "-----Printing tail of CA.TXT-----"
<add your code here>

echo "-----Creating empty .csv file to concatenate all data-----"
<add your code here>

echo "-----Adding column headers to csv file-----"
<add your code here>

echo "-----Concatenating files-----"
<add your code here>

echo "DONE!"
```

Answer

```
#!bin/bash
echo "THIS IS THE START OF MY SCRIPT!"

echo "-----Verify that we have .TXT files for all 50 states + DC-----"
ls namesbystate/*.TXT | wc -l

echo "-----Printing head of CA.TXT-----"
head namesbystate/CA.TXT

echo "-----Printing tail of CA.TXT-----"
tail namesbystate/CA.TXT

echo "-----Creating empty .csv file to concatenate all data-----"
touch namesbystate/babynames_allstates.csv

echo "-----Adding column headers to csv file-----"
echo "state, gender, year, firstname, count" > namesbystate/babynames_allstates.csv

echo "-----Concatenating files-----"
cat namesbystate/*.TXT >> namesbystate/babynames_allstates.csv

echo "DONE!"
```

3 Python Programming on Clusters

- Basic Python review
- Using virtual environments
- Writing in Jupyter notebooks
- Writing functions in Python

3.1 Introduction

We've chosen to use VS Code in this training, in part, because it has great support for developing on remote machines. Hopefully, your VS Code setup went easily, and you were able to connect to our server `included-crab`. Once connected, the VS Code interface looks just like you were working locally, and connection to the server is seamless.

Other aspects of VS Code that we like: it supports all languages thanks to the extensive free extension library, it has built in version control integration, and it is highly flexible/configurable.

We will also be working quite a bit in Jupyter notebooks in this course. Notebooks are great ways to interleave rich text (markdown formatted text, equations, images, links) and code in a way that a 'literate analysis' is generated. Although Jupyter notebooks are not subsitutes for python scripts, they can be great communication tools, and can also be convenient for code development.

3.2 Starting a project

To get set up for the course, let's connect to the server again. If you were able to work through the setup for the lesson without

difficulty, follow these steps to connect:

- open the command pallette (Cmd + Shift + P)
- enter "Remote SSH: Connect to Host"
- select `included-crab`
- enter your password in the dialog box that pops up

Now we can get set up with a project to work in for the course. Head over to the `scalable-computing-examples` github repository and fork it to your account.

Back in VS Code, in the terminal clone your fork of the `scalable-computing-examples` repo (`git clone <url-to-forked-repo-here>`) into the top level of your user directory. Run `cd ~/` if you are in some other directory.

To open the project, open the folder into your workspace

- File > Open Folder
- Enter password again if prompted

3.3 Virtual Environments

When you install a python library, let's say `pandas`, via `pip`, unless you specify otherwise, `pip` will go out and grab the most recent version of the library, and install it somewhere on your system path (where, exactly, depends highly on how you install python originally, and other factors). This is all great, untl you realize that as part of a new project, a new library you are starting to work with requires a older version of `pandas`, what do you do? You need both `pandas` versions for each of your projects. Virtual environments help to solve this issue without making the all to common situation in the comic above even more complicated.

A virtual environment is a folder structure which creates a symlink (pointer) to all of the libraries that you need into the folder. The three main components will be: the python distribution itself, its configuration, and a site-packages directory (where your

libraries like `pandas` live). So the folder is a self contained directory of all the version-specific python software you need for your project.

Virtual environments are very helpful to create reproducible workflows, and we'll talk more about this concept of reproducible environements later in the course. Perhaps most importantly though, virtual environments also help you maintain your sanity when python programming. Because they are just folders, you can create and delete new ones at will, without worrying about bungling your underlying python setup.

In this course, we are going to use `virtualenv` as our tool to create and manage virtual environments. Other virtual environment tools used commonly are `conda` and `pipenv`. One reason we like using `virtualenv` is there is an extension to it called `virtualenvwrapper`, which provides easy to remember wrappers around common `virtualenv` operations that make creating, activating, and deactivating a virtual environment very easy.

First we will create a `.bash_profile` file to create variables that point to the install locations of python and `virtualenvwrapper`. `.bash_profile` is just a text file that contains bash commands that are run every time you start up a new terminal. Although setting up this file is not required to use `virtualenvwrapper`, it is convenient because it allows you to set up some reasonable defaults to the commands (meaning less typing, overall), and it makes sure that the package is available every time you start a new terminal.

3.3.0.1 Setup

- In VS Code, select 'File > New Text File'
- Paste this text into the file:

```
export VIRTUALENVWRAPPER_VIRTUAWORKON_HOME=$HOME/.virtualenvs
source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
```

The first line points `virtualenvwrapper` to the directory where your virtual environments will be stored. We point it to a hidden directory (`.virtualenvs`) in your home directory. The last

line sources a bash script that ships with `virtualenvwrapper`, which makes all of `virtualenvwrapper` commands available in your terminal session.

- Save the file in the top of your home directory as `.bash_profile`.
- Restart your terminal (Terminal > New Terminal)
- Check to make sure it was installed and configured correctly by running this in the terminal:

```
mkvirtualenv --version
```

It should return some content that looks like this (with more output, potentially).

```
virtualenv 20.13.0+ds from /usr/lib/python3/dist-packages/virtualenv/__init__.py
```

3.3.0.2 Course environment

Now we can create the virtual environment we will use for the course. In the terminal run:

```
mkvirtualenv -p python3.9 scomp
```

Here, we've specified explicitly which python version to use by using the `-p` flag, and the path to the python 3.9 installation on the server. After making a virtual environment, it will automatically be activated. You'll see the name of the env you are working in on the left side of your terminal prompt in parentheses. To deactivate your environment (like if you want to work on a different project), just run `deactivate`. To activate it again, run:

```
workon scomp
```

You can get a list of all available environments by just running:

```
workon
```

Now let's install the dependencies for this course into that environment. To install our libraries we'll use `pip`. As of Python 3.4, `pip` is automatically included with your python installation. `pip` is a package manager for python, and you might have used it already to install common python libraries like `pandas` or `numpy`. `pip` goes out to PyPI, the Python Package Index, to download the code and put it in your `site-packages` directory. Note that on this shared server, your user directory will ahve a `site-packages` directory, in addition to one that our systems administrator manages as the root of the system.

```
pip install -r requirements.txt
```

3.3.0.3 Installing locally (optional)

`virtualenvwrapper` was already installed on the server we are working on. To install on your local computer, run:

```
pip3 install virtualenvwrapper
```

And then follow the instructions as described above, making sure that you have the correct paths set when you edit your `.bash_profile`.

3.4 Brief overview of python syntax

We'll very briefly go over some basic python syntax and the base variable types. First, open a python script. From the File menu, select New File, type "python", then save it as 'python-intro.py' in the top level of your directory.

In your file, assign a value to a variable using `=` and print the result.

```
x = 4
print(x)
```

```
4
```

To run this code in python we can:

- execute `python python-intro.py` in the terminal
- click the Play button in the upper right hand corner of the file editor
- right click any line and select: "Run to line in interactive terminal"

In that interactive window you can then run python code interactively, which is what we'll use for the next bit of exploring data types.

There are 5 standard data types in python

- Number (int, long, float, complex)
- String
- List
- Tuple
- Dictionary

We already saw a number type, here is a string:

```
str = 'Hello World!'
print(str)
```

```
Hello World!
```

Lists in python are very versatile, and are created using square brackets `[]`. Items in a list can be of different data types.

```
list = [100, 50, -20, 'text']
print(list)
```

```
[100, 50, -20, 'text']
```

You can access items in a list by index using the square brackets. Note indexing starts with 0 in python. The slice operator enables you to easily access a portion of the list without needing to specify every index.

```
list[0] # print first element
list[1:3] # print 2nd until 4th elements
list[:2] # print first until the 3rd
list[2:] # print last elements from 3rd
```

```
100
```

```
[50, -20]
```

```
[100, 50]
```

```
[-20, 'text']
```

The + and * operators work on lists by creating a new list using either concatenation (+) or repetition (*).

```
list2 = ['more', 'things']

list + list2
list * 3
```

```
[100, 50, -20, 'text', 'more', 'things']
```

```
[100, 50, -20, 'text', 100, 50, -20, 'text', 100, 50, -20, 'text']
```

Tuples are similar to lists, except the values cannot be changed in place. They are constructed with parentheses.

```
tuple = ('a', 'b', 'c', 'd')
tuple[0]
tuple * 3
tuple + tuple
```

```
'a'
```

```
('a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd')
```

```
('a', 'b', 'c', 'd', 'a', 'b', 'c', 'd')
```

Observe the difference when we try to change the first value. It works for a list:

```
list[0] = 'new value'
list
```

```
['new value', 50, -20, 'text']
```

...and errors for a tuple.

```
tuple[0] = 'new value'
```

```
TypeError: 'tuple' object does not support item assignment
```

Dictionaries consist of key-value pairs, and are created using the syntax `{key: value}`. Keys are usually numbers or strings, and values can be any data type.

```
dict = {'name': ['Jeanette', 'Matt'],
        'location': ['Tucson', 'Juneau']}

dict['name']
dict.keys()
```

```
['Jeanette', 'Matt']
```

```
dict_keys(['name', 'location'])
```

To determine the type of an object, you can use the `type()` method.

```
type(list)
type(tuple)
type(dict)
```

`list`

`tuple`

`dict`

3.5 Jupyter notebooks

To create a new notebook, from the file menu select File > New File > Jupyter Notebook

At the top of your notebook, add a first level header using a single hash. Practice some markdown text by creating:

- a list
- **bold** text
- a link

Use the Markdown cheat sheet if needed.

You can click the plus button below any chunk to add a chunk of either markdown or python.

3.5.1 Load libraries

In your first code chunk, lets load in some modules. We'll use `pandas`, `numpy`, `matplotlib.pyplot`, `requests`, `skimpy`, and `exists` from `os.path`.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import requests
import skimpy
import os
```

A note on style: There are a few ways to construct import statements. The above code uses three of the most common:

```
import module
import module as m
from module import function
```

The first way of importing will make the module a function comes from more explicitly clear, and is the simplest. However for very long module names, or ones that are used very frequently (like `pandas`, `numpy`, and `matplotlib.plot`), the code in the notebook will be more cluttered with constant calls to longer module names. So `module.function()` instead is written as `m.function()`

The second way of importing a module is a good style to use in cases where modules are used frequently, or have extremely long names. If you import every single module with a short name, however, you might have a hard time remembering which modules are named what, and it might be more confusing for others trying to read your code. Many of the most commonly used libraries for python data science have community-driven styling for how they are abbreviated in import statements, and these community norms are generally best followed.

Finally, the last way to import a single object from a module can be helpful if you only need that one piece from a larger module, but again, like the first case, results in less explicit code and therefore runs the risk of your or someone else misremembering the usage and source.

3.5.2 Read in a csv

Create a new code chunk that will download the csv that we are going to use for this tutorial.

- Navigate to Rohi Muthyala, Åsa Rennermalm, Sasha Leidman, Matthew Cooper, Sarah Cooley, et al. 2022. 62 days of Supraglacial streamflow from June-August, 2016 over southwest Greenland. Arctic Data Center. doi:10.18739/A2XW47X5F.
- Right click the download button for 'Discharge_timeseries.csv'
- Click 'copy link address'

Create a variable called URL and assign it the link copied to your clipboard. Then use `requests.get` to download the file, and `open` to write it to disk, to a directory called `data/`. We'll write this bundled in an `if` statement so that we only download the file if it doesn't yet exist. First, we create the directory if it doesn't exist:

```
if not os.path.exists ('data/'):
        os.mkdir('data/')
```

```
if not os.path.exists('data/discharge_timeseries.csv'):

        url = 'https://arcticdata.io/metacat/d1/mn/v2/object/urn%3Auuid%3Ae248467d-e1f9-4a32

        data = requests.get(url)
        a = open('data/discharge_timeseries.csv', 'wb').write(data.content)
```

Now we can read in the data from the file.

```
df = pd.read_csv('data/discharge_timeseries.csv')
df.head()
```

```
/opt/hostedtoolcache/Python/3.9.13/x64/lib/python3.9/site-packages/IPython/core/formatters.py:3

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `Styl
```

	Date	Total Pressure [m]	Air Pressure [m]	Stage [m]	Discharge [m3/s]	temperature [degree
0	6/13/2016 0:00	9.816	9.609775	0.206225	0.083531	
1	6/13/2016 0:05	9.810	9.609715	0.200285	0.077785	
2	6/13/2016 0:10	9.804	9.609656	0.194344	0.072278	
3	6/13/2016 0:15	9.800	9.609596	0.190404	0.068756	
4	6/13/2016 0:20	9.793	9.609537	0.183463	0.062804	

The column names are a bit messy so we can use `clean_columns` from `skimpy` to make them cleaner for programming very quickly. We can also use the `skim` function to get a quick summary of the data.

We can see that the `date` column is classed as a string, and not a date, so let's fix that.

```
clean_df = skimpy.clean_columns(df)
skimpy.skim(clean_df)
```

```
6 column names have been cleaned
                           skimpy summary
         Data Summary                Data Types

   dataframe           Values    Column Type   Count

   Number of rows      17856     float64       5
   Number of columns   6         string        1

                                                 number

   column_name               NA    NA %      mean       sd       p0          p25      p75     p10

   total_pressure_m           0        0       9.9     0.12        9.6      9.8       10
   air_pressure_m             0        0       9.6     0.06        9.5      9.6      9.7     9
   stage_m                    0        0      0.28     0.12    0.00056     0.17     0.37    0.
   discharge_m_3_s            0        0      0.22     0.19    4.7e-08    0.055     0.35    0.
   temperature_degrees_       8    0.045    -0.034    0.053       -0.1     -0.1        0     0

                                                 string

   column_name                NA        NA %        words per row                 total words

   date                            0           0                           2

                              End
```

```
clean_df['date'] = pd.to_datetime(clean_df['date'])
```

```
skimpy.skim(clean_df)
```

```
skimpy summary
        Data Summary                Data Types

 dataframe           Values     Column Type   Count

 Number of rows      17856      float64       5
 Number of columns   6          datetime64    1

                                                number

 column_name                NA    NA %      mean       sd       p0          p25      p75     p10

 total_pressure_m            0         0       9.9     0.12        9.6      9.8       10
 air_pressure_m              0         0       9.6     0.06        9.5      9.6      9.7     9
 stage_m                     0         0      0.28     0.12    0.00056     0.17     0.37    0.
 discharge_m_3_s             0         0      0.22     0.19    4.7e-08    0.055     0.35    0.
 temperature_degrees_        8     0.045    -0.034    0.053       -0.1     -0.1        0     0

                                               datetime

 column_name          NA      NA %       first              last                            fr

 date                      0          0      2016-06-13           2016-08-13 23:55:00         5T

                            End
```

If we wanted to calculate the daily mean flow (as opposed to the flow every 5 minutes), we need to:

- create a new column with only the date
- group by that variable
- summarize over it by taking the mean

First we should probably rename our existing date/time column to prevent from getting confused.

```
clean_df = clean_df.rename(columns = {'date': 'datetime'})
```

Now create the new date column

```
clean_df['date'] = clean_df['datetime'].dt.date
```

Finally, we use group by to split the data into groups according to the date. We can then apply the `mean` method to calculate the mean value across all of the columns. Note that there are other methods you can use to calculate different statistics across different columns (eg: `clean_df.groupby('date').agg({'discharge_m_3_s': 'max'})`).

```
daily_flow = clean_df.groupby('date', as_index = False).mean()
```

- create a simple plot

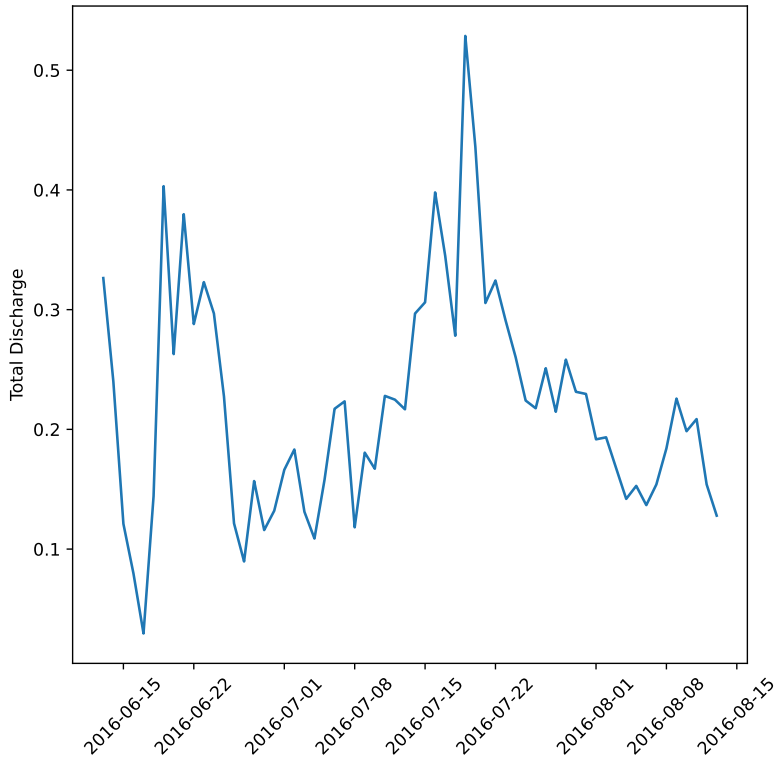
```
var = 'discharge_m_3_s'
var_labs = {'discharge_m_3_s': 'Total Discharge'}

fig, ax = plt.subplots(figsize=(7, 7))
plt.plot(daily_flow['date'], daily_flow[var])
plt.xticks(rotation = 45)
ax.set_ylabel(var_labs.get('discharge_m_3_s'))
```

```
(array([16967., 16974., 16983., 16990., 16997., 17004., 17014., 17021.,
        17028.]),
 [Text(0, 0, ''),
  Text(0, 0, ''),
```

```
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```

```
Text(0, 0.5, 'Total Discharge')
```



3.6 Functions

The plot we made above is great, but what if we wanted to make it for each variable? We could copy paste it and replace some things, but this violates a core tenet of programming: Don't

Repeat Yourself! Instead, we'll create a function called `myplot` that accepts the data frame and variable as arguments.

- create `myplot.py`

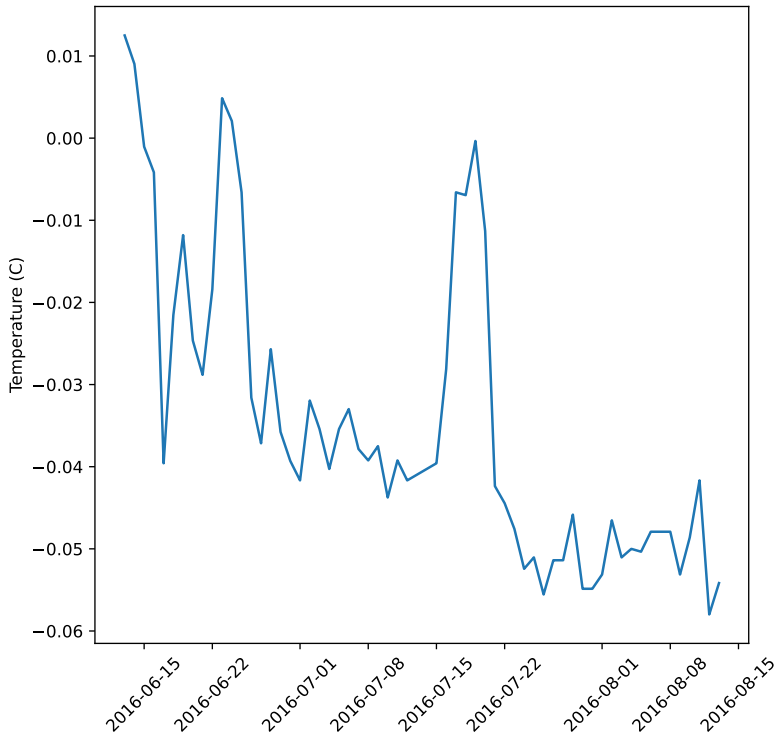
```
def myplot(df, var):

        var_labs = {'discharge_m_3_s': 'Total Discharge (m^3/s)',
                    'total_pressure_m': 'Total Pressure (m)',
                    'air_pressure_m': 'Air Pressure (m)',
                    'stage_m': 'Stage (m)',
                    'temperature_degrees_c': 'Temperature (C)'}

        fig, ax = plt.subplots(figsize=(7, 7))
        plt.plot(df['date'], df[var])
        plt.xticks(rotation = 45)
        ax.set_ylabel(var_labs.get(var))
```

- load myplot into jupyter notebook (`from myplot.py import myplot`)
- replace old plot method with new function

```
myplot(daily_flow, 'temperature_degrees_c')
```



We'll have more on functions in the software design sections.

4 Pleasingly Parallel Programming

- Understand what parallel computing is and when it may be useful
- Understand how parallelism can work
- Review sequential loops and map functions
- Build a parallel program using `concurrent.futures`
- Build a parallel program using `parsl`
- Understand Thread Pools and Process pools

4.1 Introduction

Processing large amounts of data with complex models can be time consuming. New types of sensing means the scale of data collection today is massive. And modeled outputs can be large as well. For example, here's a 2 TB (that's Terabyte) set of modeled output data from Ofir Levy et al. 2016 that models 15 environmental variables at hourly time scales for hundreds of years across a regular grid spanning a good chunk of North America:

There are over 400,000 individual netCDF files in the Levy et al. microclimate data set. Processing them would benefit massively from parallelization.

Alternatively, think of remote sensing data. Processing airborne hyperspectral data can involve processing each of hundreds of bands of data for each image in a flight path that is repeated many times over months and years.

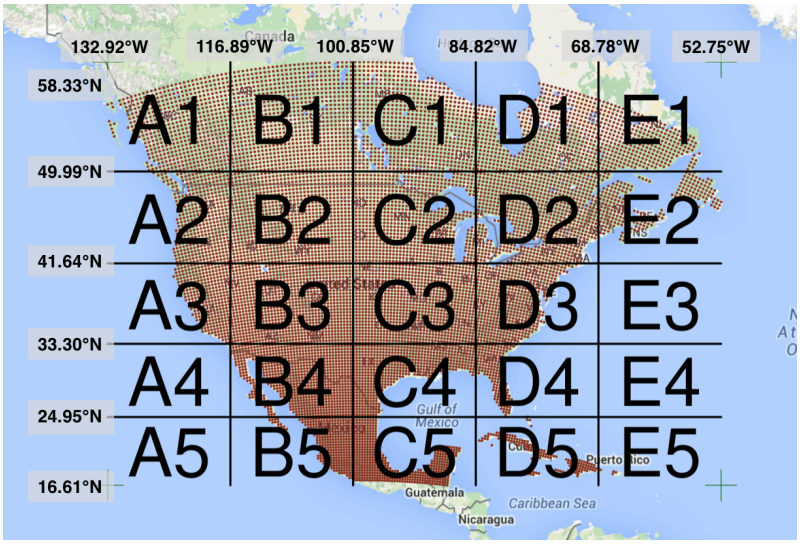


Figure 4.1: Levy et al. 2016. doi:10.5063/F1Z899CZ

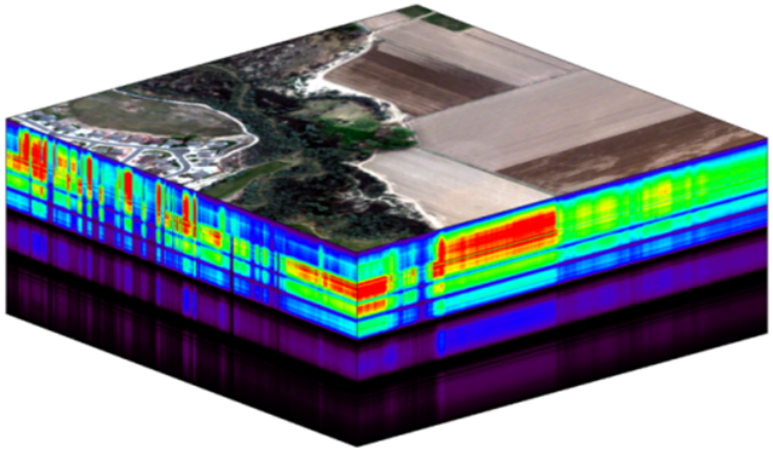


Figure 4.2: NEON Data Cube

4.2 Why parallelism?

Much R code runs fast and fine on a single processor. But at times, computations can be:

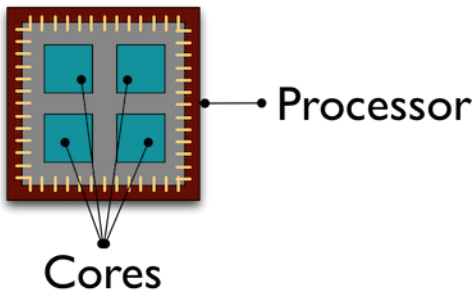
- **cpu-bound**: Take too much cpu time
- **memory-bound**: Take too much memory
- **I/O-bound**: Take too much time to read/write from disk
- **network-bound**: Take too much time to transfer

To help with **cpu-bound** computations, one can take advantage of modern processor architectures that provide multiple cores on a single processor, and thereby enable multiple computations to take place at the same time. In addition, some machines ship with multiple processors, allowing large computations to occur across the entire cluster of those computers. Plus, these machines also have large amounts of memory to avoid **memory-bound** computing jobs.

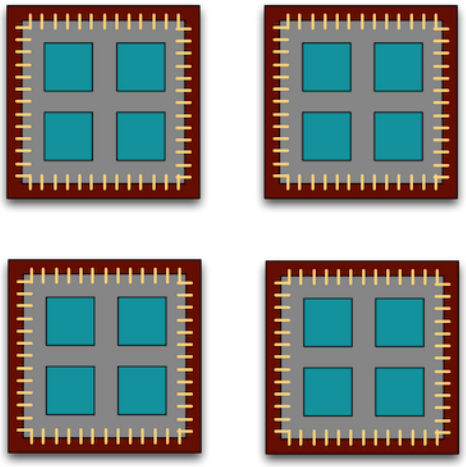
4.3 Processors (CPUs) and Cores

A modern CPU (Central Processing Unit) is at the heart of every computer. While traditional computers had a single CPU, modern computers can ship with mutliple processors, which in turn can each contain multiple cores. These processors and cores are available to perform computations.

A computer with one processor may still have 4 cores (quad-core), allowing 4 computations to be executed at the same time.



A typical modern computer has multiple cores, ranging from one or two in laptops to thousands in high performance compute clusters. Here we show four quad-core processors for a total of 16 cores in this machine.



You can think of this as allowing 16 computations to happen at the same time. Theroetically, your computation would take 1/16 of the time (but only theoretically, more on that later).

Historically, R has only utilized one processor, which makes it single-threaded. Which is a shame, because the 2017 MacBook Pro that I am writing this on is much more powerful than that:

```
{bash eval=FALSE} jones@powder:~$ sysctl hw.ncpu
hw.physicalcpu hw.ncpu: 8 hw.physicalcpu: 4
```

To interpret that output, this machine `powder` has 4 physical CPUs, each of which has two processing cores, for a total of 8 cores for computation. I'd sure like my R computations to use all of that processing power. Because its all on one machine, we can easily use *multicore* processing tools to make use of those cores. Now let's look at the computational server `aurora` at NCEAS:

```
{bash eval=FALSE} jones@included-crab:~$ lscpu
| egrep 'CPU\(s\)|per core|per socket' CPU(s):
88 On-line CPU(s) list:   0-87 Thread(s) per core:
2 Core(s) per socket:    22 NUMA node0 CPU(s):
```

```
0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,62,64,6
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,5
```

Now that's more compute power! `included-crab` has 384 GB of RAM, and ample storage. All still under the control of a single operating system.

However, maybe one of these NSF-sponsored high performance computing clusters (HPC) is looking attractive about now:

- JetStream
  - 640 nodes, 15,360 cores, 80TB RAM
- Stampede2 at TACC is coming online in 2017
  - 4200 nodes, 285,600 cores
- TODO: update with modern cluster sizes

Note that these clusters have multiple nodes (hosts), and each host has multiple cores. So this is really multiple computers clustered together to act in a coordinated fashion, but each node runs its own copy of the operating system, and is in many ways independent of the other nodes in the cluster. One way to use such a cluster would be to use just one of the nodes, and use a multi-core approach to parallelization to use all of the cores on that single machine. But to truly make use of the whole cluster, one must use parallelization tools that let us spread out our computations across multiple host nodes in the cluster.

4.4 Modes of parallelization

- TODO: develop diagram(s) showing
  - Single memory image task parallelization

```
Serial  Launch tasks --> Task 1 --> Task 2 --> Task 3
--> Task 4 --> Task 5 --> Finish
```

```
Parallel  Launch tasks -->                    Task 1
--\                   Task 2 ---\                    Task
3 -----> Finish                    Task 4 ---/
Task 5 --/
```

- Cluster task parallelization

```
Cluster    parallel    Show dispatch to cluster nodes and
reassembly of data    Launch tasks -->                      Marshal
--> Task 1 --> Unmarshal --\                     Marshal
--> Task 2 --> Unmarshal ---\                     Marshal
--> Task 3 --> Unmarshal -----> Finish                        Marshal
--> Task 4 --> Unmarshal ---/                     Marshal
--> Task 5 --> Unmarshal --/
```

- TODO: Should we also include figure with data or functional dependencies?

4.5 Task parallelism with `concurrent.futures`

When you have a list of repetitive tasks, you may be able to speed it up by adding more computing power. If each task is completely independent of the others, then it is a prime candidate for executing those tasks in parallel, each on its own core. For example, let's build a simple loop that downloads the data files that we need for an analysis. First, we start with the serial implementation.

```
# Use loop for serial execution of tasks

# Tasks are to download data from a dataset
```

The issue with this loop is that we execute each trial sequentially, which means that only one of our 8 processors on this machine are in use. In order to exploit parallelism, we need to be able to dispatch our tasks as functions, with one task going to each processor. To do that, we need to convert our task to a function, and then use the `map()` function to apply that function to all of the members of a set. Here's the same code rewritten to use `map()`, which applies a function to each of the members of a list (in this case the files we want to download):

```
# Use `map` for serial execution of tasks

# Tasks are to download data from a dataset
```

4.6 Approaches to parallelization

When parallelizing jobs, one can:

- Use the multiple cores on a local computer through `mclapply`
- Use multiple processors on local (and remote) machines using `makeCluster` and `clusterApply`
  - In this approach, one has to manually copy data and code to each cluster member using `clusterExport`
  - This is extra work, but sometimes gaining access to a large cluster is worth it

4.7 concurrent.futures

```
# Loop versus map for parallel execution of tasks

# Using concurrent.futures and ThreadPool

# Tasks are to download data from a dataset
```

4.8 parsl

- Overview of parsl and it's use of python decorators.

```
# Loop versus map for parallel execution of tasks

# Using parsl decorators and ThreadPool

# Tasks are to download data from a dataset
```

- Configurable Executors in parsl
  - HightThroughputExecutor for cluster jobs

```
# Loop versus map for parallel execution of tasks

# Using parsl decorators and ThreadPool

# Tasks are to download data from a dataset
```

4.9 When to parallelize

It's not as simple as it may seem. While in theory each added processor would linearly increase the throughput of a computation, there is overhead that reduces that efficiency. For example, the code and, importantly, the data need to be copied to each additional CPU, and this takes time and bandwidth. Plus, new processes and/or threads need to be created by the operating system, which also takes time. This overhead reduces the efficiency enough that realistic performance gains are much less than theoretical, and usually do not scale linearly as a function of processing power. For example, if the time that a computation takes is short, then the overhead of setting up these additional resources may actually overwhelm any advantages of the additional processing power, and the computation could potentially take longer!

In addition, not all of a task can be parallelized. Depending on the proportion, the expected speedup can be significantly reduced. Some propose that this may follow Amdahl's Law, where the speedup of the computation (y-axis) is a function of both the number of cores (x-axis) and the proportion of the computation that can be parallelized (see colored lines):

```
#| eval: false
library(ggplot2)
library(tidyr)
amdahl <- function(p, s) {
  return(1 / ( (1-p) + p/s  ))
}
```

```
doubles <- 2^(seq(0,16))
cpu_perf <- cbind(cpus = doubles, p50 = amdahl(.5, doubles))
cpu_perf <- cbind(cpu_perf, p75 = amdahl(.75, doubles))
cpu_perf <- cbind(cpu_perf, p85 = amdahl(.85, doubles))
cpu_perf <- cbind(cpu_perf, p90 = amdahl(.90, doubles))
cpu_perf <- cbind(cpu_perf, p95 = amdahl(.95, doubles))
#cpu_perf <- cbind(cpu_perf, p99 = amdahl(.99, doubles))
cpu_perf <- as.data.frame(cpu_perf)
cpu_perf <- cpu_perf %>% gather(prop, speedup, -cpus)
ggplot(cpu_perf, aes(cpus, speedup, color=prop)) +
  geom_line() +
  scale_x_continuous(trans='log2') +
  theme_bw() +
  labs(title = "Amdahl's Law")
```

So, its important to evaluate the computational efficiency of requests, and work to ensure that additional compute resources brought to bear will pay off in terms of increased work being done. With that, let's do some parallel computing…

5 Parallel Pitfalls and their solutions

- Race conditions
- Deadlocks

5.1 Summary

In this lesson, we showed examples of computing tasks that are likely limited by the number of CPU cores that can be applied, and we reviewed the architecture of computers to understand the relationship between CPU processors and cores. Next, we reviewed the way in which traditional `for` loops in R can be rewritten as functions that are applied to a list serially using `lapply`, and then how the `parallel` package `mclapply` function can be substituted in order to utilize multiple cores on the local computer to speed up computations. Finally, we installed and reviewed the use of the `foreach` package with the `%dopar` operator to accomplish a similar parallelization using multiple cores.

5.2 Further Reading

Ryan Abernathey & Joe Hamman. 2020. Closed Platforms vs. Open Architectures for Cloud-Native Earth System Analytics. Medium.

6 Documenting and Publishing Data

6.1 Introduction

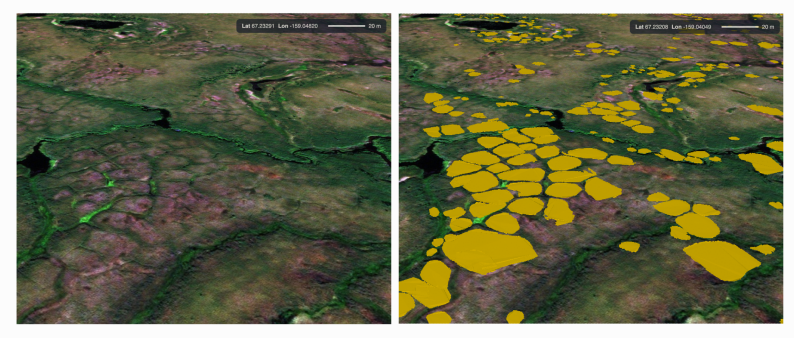
7 Group Project: Staging and Preprocessing

- Get familiarized with the overall group project workflow
- Write a parsl app that will stage and tile the IWP example data in parallel

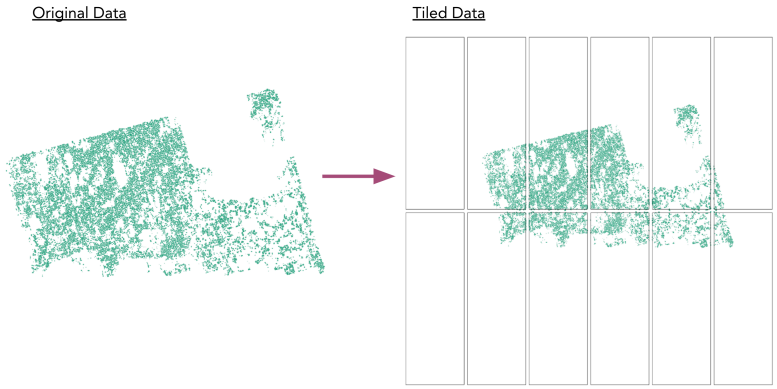
7.1 Introduction

The Permafrost Discovery Gateway is an online platform for archiving, processing, analysis, and visualization of permafrost big imagery products to enable discovery and knowledge-generation. The PDG utilizes and makes available products derived from high resolution satellite imagrey from the Polar Geospatial Center, Planet (3m), Sentinel (10 m), Landsat (30 m), and MODIS (250 m). One of these products is a dataset showing Ice Wedge Polygons (IWP) that form in melting permafrost.

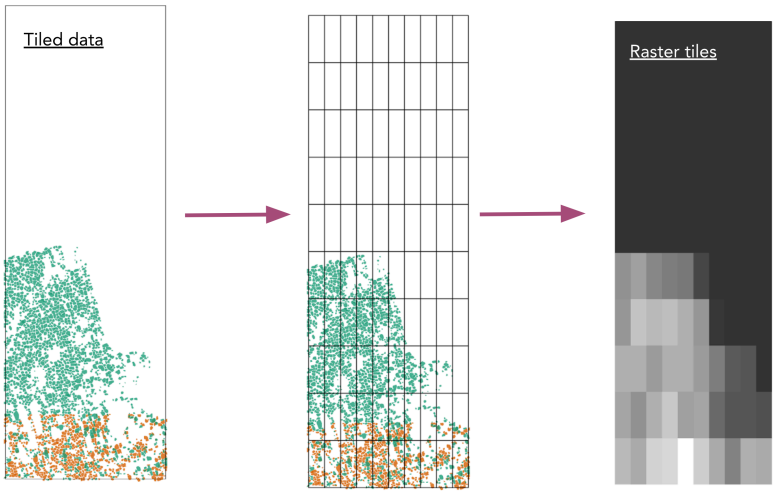
Ice wedges form as a result of thermal contraction during melt/freeze cycles of permafrost. They can form very distinctive geometries clearly visible in satellite images. The PDG is using advanced analysis and computational tools to take high resolution satellite imagery and automatically dectect where ice wedge polygons form. Below is an example of a satellite image (left) and the detected ice wedge polygons in geospatial vector format (right) of that same image.



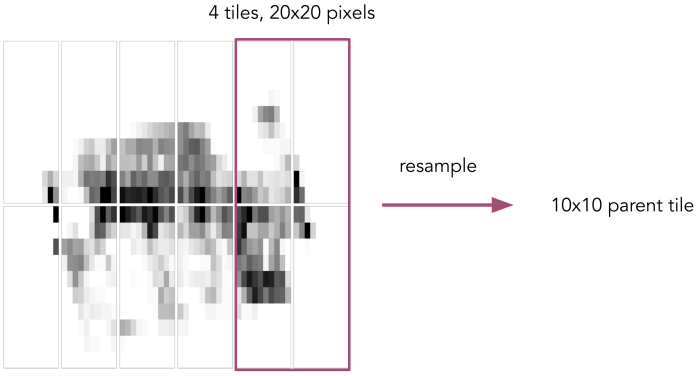
In the group project, we are going to use a subset of the high resolution dataset of these detected ice wedge polygons in order to learn some of the reproducible, scalable techniques that will allow us to process it. Our workflow will start with a set of large geopackage files that contain the detected ice wedge polygons. These files all have irregular extents due to the variation in satellite coverage, clouds, etc. Our first processing step will take these files and "tile" them into smaller files which have regular extents.



In step two of the workflow, we will take those regularly tiled geopackage files and rasterize them. The files will be regularly gridded, and a summary statistic will be calculated for each grid cell (such as the proportion of pixel area covered by polygons).



In the final step of the workflow, we will take the raster files and resample them to create a set of raster tiles at different resolutions. This last step is what will enable us to visualize our raster data dynamically, such that we look at lower resolutions when very zoomed out (and high resolution data would take too long to load), and higher resolution data when zoomed in and the extent is smaller.



7.2 Staging and Tiling

Today we will undertake the first step of the workflow, staging and tiling the data.

In your fork of the `scalable-computing-examples` repository, open the Jupyter notebook in the `group-project` directory called `session-06.ipynb`. This workbook will serve as a skeleton for you to work in. It will load in all the libraries you need, including a few helper functions we wrote for the course, show an example for how to use the method on one file, and then lays out blocks for you and your group to fill in with code that will run that method in parallel.

In your small groups, work together to write the solution, but everyone should aim to have a working solution on their own fork of the repository. In other words, everyone should type out the solution themselves as part of the group effort. Writing the code out yourself (even if others are contributing to the content) is a great way to get "mileage" as you develop these skills.

8 Software Design I

9 Data Structures and Formats for Large Data

Efficient and reproducible data analysis begins with choosing a proper format to store our data, particularly when working with large, complex, multi-dimensional datasets. Consider, for example, the following Earth System Data Cube from Mahecha et al. 2020, which measures nine environmental variables at high resolution across space and time. We can consider this dataset large (high-resolution means we have a big file), complex (multiple variables), and multi-dimensional (each variable is measured along three dimensions: latitude, longitude, and time). Additionally, necessary metadata must accompany the dataset to make it functional, such as units of measurement for variables, information about the authors, and processing software used.

Keeping complex datasets in a format that facilitates access, processing, sharing, and archiving can be at least as important as how we parallelize the code we use to analyze them. In practice, it is common to convert our data from less efficient formats into more efficient ones before we parallelize any processing. In this lesson, we will

1. familiarize ourselves with the NetCDF data format, which enables us to store large, complex, multi-dimensional data efficiently, and
2. learn to use the `xarray` Python package to read, process, and create NetCDF files.

9.1 Objectives

- Learn about the NetCDF data format:

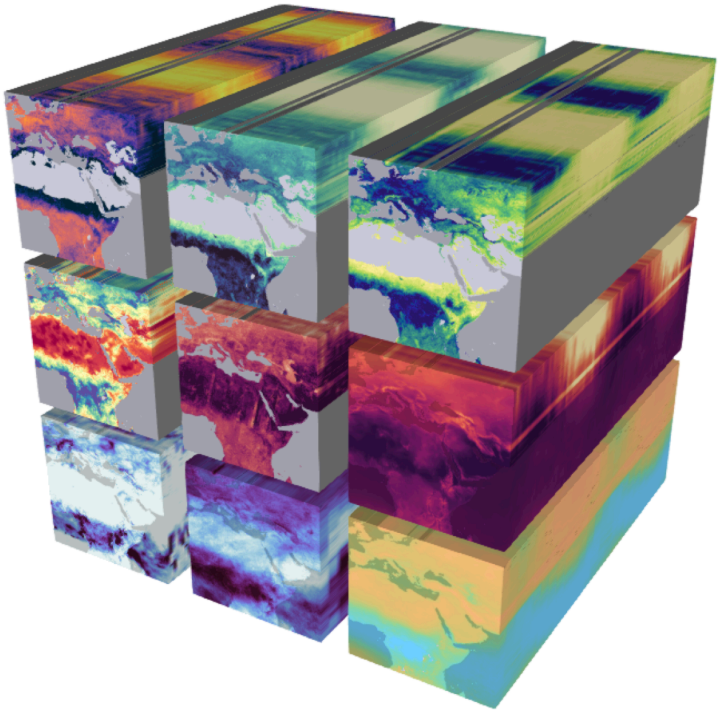


Figure 9.1: Mahecha et al. 2020 . *Visualization of the implemented Earth system data cube. The figure shows from the top left to bottom right the variables sensible heat (H), latent heat (LE), gross primary production (GPP), surface moisture (SM), land surface temperature (LST), air temperature (Tair), cloudiness (C), precipitation (P), and water vapour (V). The resolution in space is 0.25° and 8 d in time, and we are inspecting the time from May 2008 to May 2010; the spatial range is from 15° S to 60° N, and 10° E to 65° W.*

  - Characterstics: self-describing, scalable, portable, appendable, shareble, and archivable
  - Understand the NetCDF data model: what are dimensions, variables and attributes
  - Advantages of random/arbitrary access and how that applies to parallel computing

- Learn how to use the `xarray` Python package to work with NetCDF files:

  - Describe the core `xarray` data structures, the `xarray.DataArray` and the `xarray.Dataset`, and their components, including: data variables, dimensions, coordinates, and attributes
  - Create `xarray.DataArrays` and `xarra.DataSets` out of raw `numpy` arrays and save them as netCDF files
  - Load `xarray` datasets from netCDF and understand the attributes view
  - Perform basic indexing, processing and reduction of `xarray.DataArrays`

9.2 NetCDF data format

NetCDF (network Common Data Form) is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data. NetCDF was originaly developed at the Unidata Program Center and is supported on almost all platforms, and parsers exist for the vast majority of scientific programming languages.

9.2.1 Characteristics

[REF: https://docs.unidata.ucar.edu/netcdf-c/current/faq.html#ncFAQGeneral]
TO DO: maybe add a 5 word discussion of each point

NetCDF files are designed to be:

1. **Self-describing:** Information describing the data contents of the file are embedded within the data file itself. This means that there is a header which describes the layout of the rest of the file as well as arbitrary file metadata.
2. **Scalable:** Small subsets of large datasets may be accessed efficiently through netCDF interfaces, even from remote servers.
3. **Portable:** A NetCDF file is machine-independent i.e. it can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
4. **Appendable:** Data may be appended to a properly structured NetCDF file without copying the dataset or redefining its structure.
5. **Sharable:** One writer and multiple readers may simultaneously access the same NetCDF file.
6. **Archivable:** Access to all earlier forms of NetCDF data will be supported by current and future versions of the software.

9.2.2 Data Model

The NetCDF data model is the way that NetCDF organizes data. The *Classic NetCDF Data Model* consists of variables, dimensions, and attributes. This way of thinking about data was introduced with the very first NetCDF release, and is still the core of all netCDF files. There exists a new Enhanced Data Model, but for maximum interoperability with existing code, new data should be created with the Classic Model.

[REF https://docs.unidata.ucar.edu/netcdf-c/current/netcdf_data_model.html#classic_model]

The model consists of three key components:

Variables are N-dimensional arrays of data. Variables in netCDF files can be one of six types (char, byte, short, int, float, double). We can think of these as the varying/measured/dependent quantities.

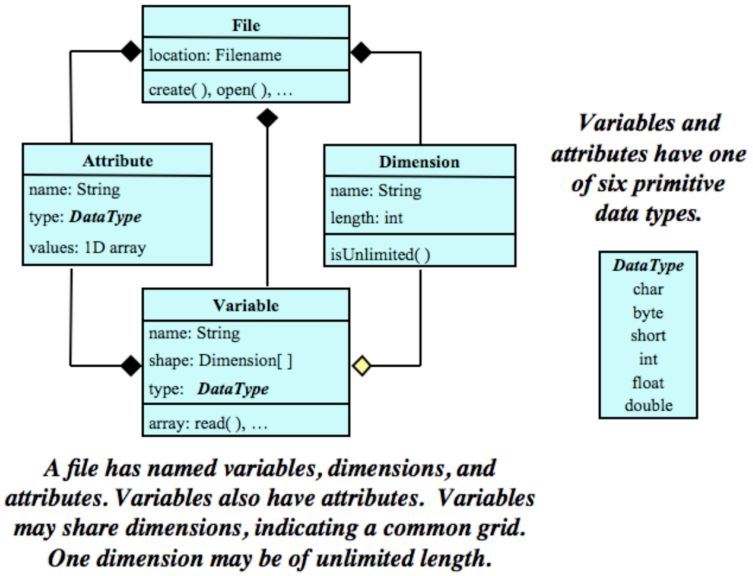


Figure 9.2: Classic NetCDF Data Model (NetCDF documentation)

Dimensions describe the axes of the data arrays. A dimension has a name and a length. An unlimited dimension has a length that can be expanded at any time, as more data are written to it. NetCDF files can contain at most one unlimited dimension. We can think of these as the constant/fixed/independent quantities at which we measure the variables.

Attributes are small notes or supplementary metadata to annotate either a variable or the file as a whole. Although there is no enforced limit, the user is expected to keep attributes small.

> **Note**
>
> *The most commonly used metadata standard for geospatial data is the Climate and Forecast metadata standard, also called the CF conventions. These standards are specifically designed to promote the processing and sharing of files created with the NetCDF API. Principles of CF include self-describing data (no external tables needed for*

> *understanding); metadata equally readable by humans and software; minimum redundancy and maximum simplicity.*

9.2.3 Exercise

[TO DO: check exercise setup]

Imagine the following scenario: we have a network of 25 weather stations. They are located in a square grid: starting at 30°0 N 60°0 E, there is a station every 10° North and every 10° East. Each station measures the air temperature at a set time for three days, starting on September 1st, 2022. The first day all stations record a temperature of 0°C, the second day all temperatures are 1°C, and the third day all temperatures are 3°C. What are the *variables*, *dimensions* and *attributes* for this data?

Variables: There is a single variable being measured: temperature. The variable values can be represented as a 3x5x5 array, with constant values for each day as seen in the diagram.

Dimensions: There are three dimensions in this dataset: date, latitude, and longitude. Time indicates when the measurement happened, we can encode it as the dates 2022-09-1, 2022-09-02, and 2022-09-03. The pairs of latitude and longitude values indicate the positions of the weather stations. Latitude has values 30, 40, 50, 60, and 70, measured in degrees North, while longitude has values 60, 70, 80, 90, and 100, measured in degrees East.

Attributes: Let's divide these in attributes for the variable, the dimensions, and the whole dataset:

- Temperature attributes:
  - units: degrees Celsius
- Time attributes:
  - description: date of measurement
- Latitude attributes:
  - units: degrees North

- Longitude attributes:
  - units: degrees East
- Dataset attributes:
  - title: temperature at weather stations
  - summary: an example of NetCDF data format

Our next step is to see how we can translate all this information into something we can store and handle in our computer.

9.3 `xarray`

https://docs.xarray.dev/en/stable/getting-started-guide/why-xarray.html https://docs.xarray.dev/en/stable/getting-started-guide/faq.html

Multi-dimensional arrays or ND arrays are frequently encountered in geosciences. Consider, for example, how many variables can be measured with respect to space-time dimensions, making those datasets three or even four-dimensional (for instance, if we use latitude, longitude, height/depth, and time). In Python, the `NumPy` package provides the fundamental data structure and API for working with raw ND arrays. However, real-world datasets are usually more than just raw numbers; they have labels that encode information about how the array values correspond to locations in space, time, etc. `xarray` is an answer to this necessity: an `xarray.DataArray` has labeled dimensions (e.g. "time", "latitude") that can be directly referenced for processing. It is easier to keep track of a dimension labeled "time" than to remember that time is the n-th dimension of the array. Moreover, `xarray` is based on the netCDF data model, making it the appropriate tool to open, process and create datasets in netCDF format.

9.3.1 Creating an `xarray.DataArray`

An `xarray.DataArray` is an N-dimensional array with labeled coordinates and dimensions. It is the primary data structure of the `xarray` package. We can think of it as

representing a single variable in the NetCDF data format: it holds the variable's values, dimensions, and attributes. Additionally, each dimension has at least one set of *coordinates*, which indicate the values the dimension takes. We can thin of the coordinate's values as the tick labels along a dimension. [REF https://docs.xarray.dev/en/stable/user-guide/terminology.html]

As our first example, let's suppose we want to make an `xarray.DataArray` that includes the information from our previous exercise about measuring temperature across three days. First, we import all our necessary libraries.

```
import os
import requests
import pandas as pd
import numpy as np

import xarray as xr   # This is the package we'll explore
```

The underlying data in the `xarray.DataArray` is a `numpy.ndarray` that holds the variable values. So we can start by making a `numpy.ndarray` with our mock temperature data:

```
# values of a single variable at each point of the coords
temp_data = np.array([np.zeros((5,5)),
                      np.ones((5,5)),
                      np.ones((5,5))*2]).astype(int)
temp_data
```

```
array([[[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]],

       [[1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
```

```
       [1, 1, 1, 1, 1]],

      [[2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2],
       [2, 2, 2, 2, 2]]])
```

We could think that this is "all" we need to represent our data. But if we stopped at this point, we would need to

1. remember that the numbers in this array represent temperature in degrees Celsius (doesn't seem too bad),
2. remember that the first dimension of the array represents time, the second latitude and the third longitude (maybe ok), and
3. keep track of the range of values that time, latitude and longitude take (not so good).

Keeping track of all this information separately could quickly get messy and could make it challenging to share our data and analyses with others . This is what the netCDF data model and `xarray` aim to simplify. We can get data and its descriptors together in an `xarray.DataArray` by adding the dimensions over which the variable is being measured and including attributes that appropriately describe dimensions and variables.

```
# names of the dimensions
dims = ('time', 'lat', 'lon')

# coordinates (tick labels) to use for indexing along each dimension
coords = {'time' : pd.date_range("2022-09-01", "2022-09-03"),
          'lat' : np.arange(30,80,10),
          'lon' : np.arange(60,110,10)}

# attributes (metadata) of the data array
attrs = { 'title' : 'temperature across weather stations',
          'units' : 'degrees_celsius'}

# initialize xarray.DataArray
```

```
temp = xr.DataArray(data = temp_data,
                    dims = dims,
                    coords = coords,
                    attrs = attrs)
temp
```

```
<xarray.DataArray (time: 3, lat: 5, lon: 5)>
array([[[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]],

       [[1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1]],

       [[2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2]]])
Coordinates:
  * time     (time) datetime64[ns] 2022-09-01 2022-09-02 2022-09-03
  * lat      (lat) int64 30 40 50 60 70
  * lon      (lon) int64 60 70 80 90 100
Attributes:
    title:    temperature across weather stations
    units:    degrees_celsius
```

We can also update the variable's attributes after creating the object. Notice that each of the coordinates is also an `xarray.DataArray`, so we can add attributes to them.

```
# update attributes
temp.attrs['description'] = 'simple example of an xarray.DataArray'
```

```
# add attributes to coordinates
temp.time.attrs = {'standard_name':'date of collection'}

temp.lat.attrs['standard_name']= 'latitude'
temp.lat.attrs['units'] = 'degrees_north'

temp.lon.attrs['standard_name']= 'longitude'
temp.lon.attrs['units'] = 'degrees_east'
temp
```

```
<xarray.DataArray (time: 3, lat: 5, lon: 5)>
array([[[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]],

       [[1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1]],

       [[2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2],
        [2, 2, 2, 2, 2]]])
Coordinates:
  * time     (time) datetime64[ns] 2022-09-01 2022-09-02 2022-09-03
  * lat      (lat) int64 30 40 50 60 70
  * lon      (lon) int64 60 70 80 90 100
Attributes:
    title:        temperature across weather stations
    units:        degrees_celsius
    description:  simple example of an xarray.DataArray
```

At this point, since we have a single variable, the dataset attributes and the variable attributes are the same.

9.3.2 Indexing

An `xarray.DataArray` allows both positional indexing (like `numpy`) and label-based indexing (like `pandas`). Positional indexing is the most basic and it's done using Python's `[]` syntax, as in `array[i,j]` with i and j both integers. Label-based indexing takes advantage of dimensions in the array having names and coordinate values that we can use to access data, instead of remembering the positional order of each dimension.

As an example, suppose we want to know what was the temperature recorded by the weather station located at 40°0 N 80°0 E on September 1st, 2022. By recalling all the information about how the array is setup with respect to the dimensions and coordiantes, we can access this data positionally:

```
temp[0,1,2]
```

```
<xarray.DataArray ()>
array(0)
Coordinates:
    time     datetime64[ns] 2022-09-01
    lat      int64 40
    lon      int64 80
Attributes:
    title:        temperature across weather stations
    units:        degrees_celsius
    description:  simple example of an xarray.DataArray
```

Or, we can use the dimensions names and their coordinates to access the same value:

```
temp.sel(time='2022-09-01', lat=40, lon=80)
```

```
<xarray.DataArray ()>
array(0)
Coordinates:
    time     datetime64[ns] 2022-09-01
    lat      int64 40
```

```
    lon      int64 80
Attributes:
    title:        temperature across weather stations
    units:        degrees_celsius
    description:  simple example of an xarray.DataArray
```

Notice that the result of this indeixing is a 1x1 `xarray.DataArray`. This is because operations on a `xarray.DataArray`n(resp. `xarray.DataSet`) always return another`xarray.DataArray` (resp. `xarray.DataSet`) object. In particular, operations returning scalar values will also return an `xarray` objects, so we need to cast as numbers them manualy. See xarray.DataArray.item.

More about `xarray` indexing.

9.3.3 Reduction

`xarray` has implemented a number of methods to reduce an `xarray.DataArray` along any number of dimensions. One of the advantages of `xarray.DataArray` is that, if we chose to, it can carry over attributes when doing calculations. For example, we can calculate the average temperature at each weather station over time and obtain a new `xarray.DataArray`.

```
avg_temp = temp.mean(dim = 'time')
# to keep attributes add keep_attrs = True

avg_temp.attrs = {'title':'average temperature over three days'}
avg_temp
```

```
<xarray.DataArray (lat: 5, lon: 5)>
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
Coordinates:
  * lat      (lat) int64 30 40 50 60 70
```

```
  * lon      (lon) int64 60 70 80 90 100
Attributes:
    title:    average temperature over three days
```

More about xarray computations.

9.3.4 Creating an `xarray.DataSet`

An `xarray.DataSet` resembles an in-memory representation of a NetCDF file, and consists of *multiple* variables, with coordinates and attributes, which together form a self describing dataset. We can make create a `xarray.DataSet` by putting together the temperature data with the average temperature data. We also add some attributes that now describe the whole dataset, not only each variable. Take some time to click through the data viewer and notice how all the data and metadata is ordered within the dataset.

```
# make dictionaries with variables and attributes
data_vars = {'avg_temp': avg_temp,
             'temp': temp}

attrs = {'creator_name':'CGG',
        'title':'temperature data at weather stations: daily and and average',
        'description':'simple example of an xarray.Dataset'}

# create xarray.Dataset
temp_dataset = xr.Dataset( data_vars = data_vars,
                          attrs = attrs)
temp_dataset
```

```
<xarray.Dataset>
Dimensions:   (lat: 5, lon: 5, time: 3)
Coordinates:
  * lat       (lat) int64 30 40 50 60 70
  * lon       (lon) int64 60 70 80 90 100
  * time      (time) datetime64[ns] 2022-09-01 2022-09-02 2022-09-03
Data variables:
    avg_temp  (lat, lon) float64 1.0 1.0 1.0 1.0 1.0 1.0 ... 1.0 1.0 1.0 1.0 1.0
```

```
    temp      (time, lat, lon) int64 0 0 0 0 0 0 0 0 0 0 ... 2 2 2 2 2 2 2 2 2 2
Attributes:
    creator_name:  CGG
    title:         temperature data at weather stations: daily and and average
    description:   simple example of an xarray.Dataset
```

9.3.5 Save and reopen

Finally, we want to save our dataset as a NetCDF file. To do this, first specify the file path and use the *.nc* extension for the file name. Then save the datset by using the `to_netcdf` method with your file path. Opening NetCDF is similarly straightforward using `xarray.open_dataset()`.

```
# specify file path: don't forget the .nc extension!
fp = os.path.join(os.getcwd(),'temp_dataset.nc')
# save file
temp_dataset.to_netcdf(fp)

# open to check:
check = xr.open_dataset(fp)
check
```

```
<xarray.Dataset>
Dimensions:   (lat: 5, lon: 5, time: 3)
Coordinates:
  * lat       (lat) int64 30 40 50 60 70
  * lon       (lon) int64 60 70 80 90 100
  * time      (time) datetime64[ns] 2022-09-01 2022-09-02 2022-09-03
Data variables:
    avg_temp  (lat, lon) float64 ...
    temp      (time, lat, lon) int64 ...
Attributes:
    creator_name:  CGG
    title:         temperature data at weather stations: daily and and average
    description:   simple example of an xarray.Dataset
```

9.3.6 Exercise

For this exercise we will use a dataset including timeseries of annual Arctic freshwater fluxes and storage terms. The data was produced for the publication Jahn and Laiho, 2020 about changes in the Arctic freshwater budget and is archived at the Arctic Data Center doi:10.18739/A2280504J

```
url = 'https://arcticdata.io/metacat/d1/mn/v2/object/urn%3Auuid%3A792bfc37-416e-409e-80b1-fd

response = requests.get(url)
open("FW_data_CESM_LW_2006_2100.nc", "wb").write(response.content)
```

208086

```
fp = os.path.join(os.getcwd(),'FW_data_CESM_LW_2006_2100.nc')
fw_data = xr.open_dataset(fp)
fw_data
#netPrec_annual
```

```
<xarray.Dataset>
Dimensions:                          (time: 95, member: 11)
Coordinates:
  * time                             (time) float64 2.006e+03 ... 2.1e+03
  * member                           (member) float64 1.0 2.0 3.0 ... 10.0 11.0
Data variables: (12/16)
    FW_flux_Fram_annual_net          (time, member) float64 ...
    FW_flux_Barrow_annual_net        (time, member) float64 ...
    FW_flux_Nares_annual_net         (time, member) float64 ...
    FW_flux_Davis_annual_net         (time, member) float64 ...
    FW_flux_BSO_annual_net           (time, member) float64 ...
    FW_flux_Bering_annual_net        (time, member) float64 ...
    ...                               ...
    Solid_FW_flux_BSO_annual_net     (time, member) float64 ...
    Solid_FW_flux_Bering_annual_net  (time, member) float64 ...
    runoff_annual                    (time, member) float64 ...
    netPrec_annual                   (time, member) float64 ...
    Liquid_FW_storage_Arctic_annual  (time, member) float64 ...
    Solid_FW_storage_Arctic_annual   (time, member) float64 ...
```

```
Attributes:
    creation_date:   02-Jun-2020 15:38:31
    author:          Alexandra Jahn, CU Boulder, alexandra.jahn@colorado.edu
    title:           Annual timeseries of freshwater data from the CESM Low W...
    description:     Annual mean Freshwater (FW) fluxes and storage relative ...
    data_structure:  The data structure is |Ensemble member | Time (in years)...
```

9.3.7 Exercise 3

https://arcticdata.io/catalog/view/doi:10.18739/A26T0GX63

9.4 INCLUDE THIS? - Bryce's notes

- Look into parallel access and NetCDF and whether all wheels can take advantage of the built-in parallel access
- Data Organization / File Naming?
- Using hierarchical folder structures
  - Encoding hierarchy into filenames
  - File naming for natural ordering (principle of most sig. fist)
  - Setting things up for Dask multi-file support
- Understand the advantages of random and parallel formats such as NetCDF
- Talk about NetCDF and its role in data archival
- Is it a good archival format: Yes! Better than CSV in many (not all) ways. The format is open and well-documented, support for the reading the format is ubiquitous, it's efficient w/ disk space (compared to CSV), it supports remote querying (unlike CSV).

10 Parallelization with Dask

Dask is a library for parallel computing in Python. It can scale up code to use the full capacity of your personal computer or to distribute work in a cloud cluster. By mirroring APIs of other commonly used Python libraries such as Pandas, NumPy, and Scikit-learn, Dask provides a familiar interface that makes it easier to parallelize your code. In this lesson, we will get acquainted with some of Dask's most commonly used objects and Dask's way of distributing and evaluating computations.

10.1 Objectives

- Become familiar with Dask processing workflow:
  - What are the client, scheduler, workers, and cluster
  - Understand delayed computations and "lazy" evaluation
  - Obtain information about computations via the Dask dashboard
- Learn basics of `dask.arrays` and `dask.dataframes`:
  - Load data and specifying partition/chunk sizes
  - Interpret a task graph
- Integrate `xarray` with Dask for geospatial computations

- Share best practices and resources for a deeper dive

10.2 Dask Cluster

We can deploy a Dask cluster on a single machine or an actual cluster with multiple machines. Here we have chosen to use the Dask cluster instead of the default Dask scheduler to take advantage of the *cluster dashboard*, which keeps track of the performance and progress of our computations.

Dask clusters have three main components for processing computations in parallel. These are the *client*, the *scheduler* and the *workers*.

- When we code, we communicate directly with the **client**, which is responsible for submitting tasks to be executed to the scheduler.
- After receiving the tasks from the client, the **scheduler** determines how tasks will be distributed among the workers and coordinates them to process tasks in parallel.
- Finally, the **workers** are threads, processes, or separate machines in a cluster. They compute tasks and store and return computations results.

In order to interact with the client and generate tasks that can be processed in parallel we need to use Dasks' objects to read our data. Here we will see examples of how to use `dask.dataframes` and `dask.arrays`.

10.2.1 Setting up a Local Cluster

We can create a local cluster as follows:

```
from dask.distributed import LocalCluster, Client
# https://distributed.dask.org/en/stable/install.html
```

```
# cluster = LocalCluster()
# cluster
```

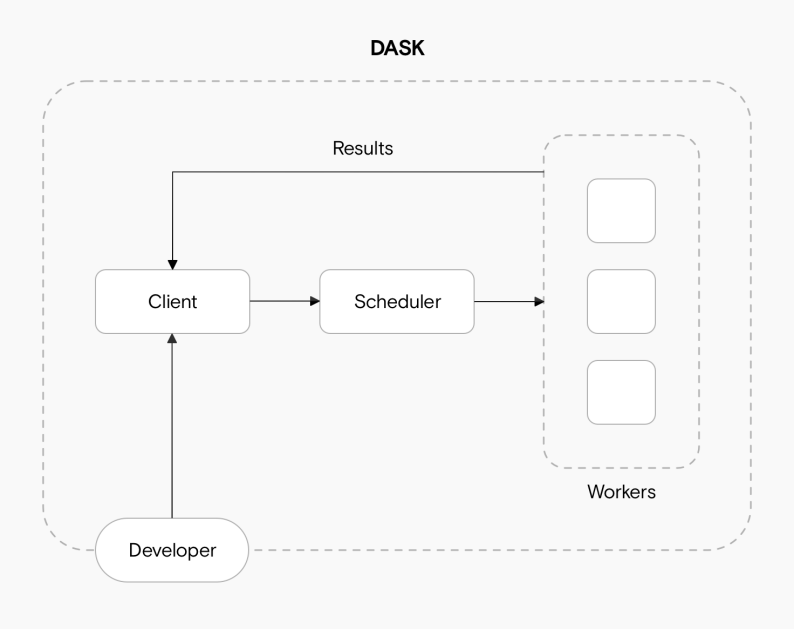


Figure 10.1: From: https://www.datarevenue.com/en-blog/understanding-dask-architecture-client-scheduler-workers Make this into a collored graph, add cluster envelope

And then we create a client to connect to our cluster:

```
# client = Client(cluster)
# client
```

Read more about Dask clusters

10.2.2 Dask Dashboard

https://www.youtube.com/watch?v=N_GqzcuGLCY

10.3 `dask.dataframes`

When we analyze tabular data, we usually start our analysis by loading it into memory as a Pandas DataFrame. But, what if this data does not fit in memory? Or maybe our analyzes crash because we run out of memory. These scenarios are typical entry points into parallel computing. In such cases, Dask's scalable alternative to a Pandas DataFrame is the `dask.dataframe`. A `dask.dataframe` is made up of many `pd.DataFrames`, each containing a subset of rows of the original dataset. We call each of these pandas pieces a **partition** of the `dask.dataframe`.

10.3.1 Reading a csv

To get familiar with `dask.dataframes` we will use tabular data of soil mositure measurements at six forest stands in northeastern Siberia. The data has been collected since 2014 and is archived at the Arctic Data Center (Loranty & Alexander, doi:10.18739/A24B2X59C). Just as we did on the previous lesson, we download the data using the `requests` package and the data's url obtained from the Arctic Data Center.

```
import os
import requests
```

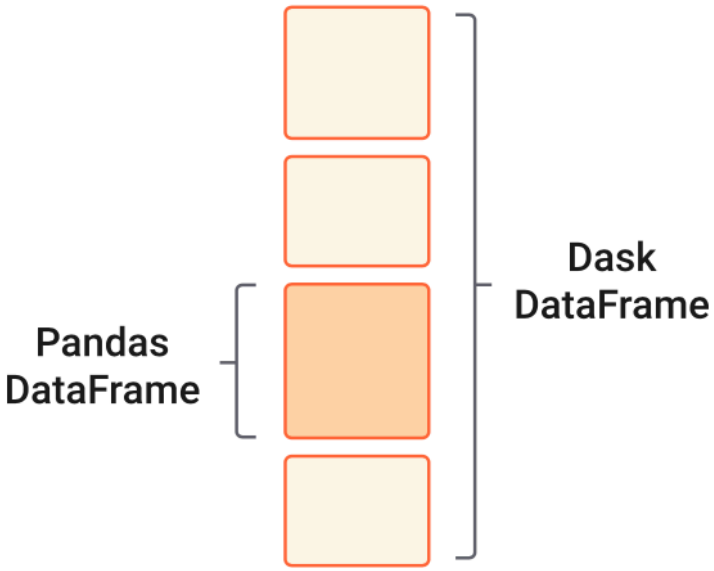


Figure 10.2: Dask Array design (dask documentation)

```
url = 'https://arcticdata.io/metacat/d1/mn/v2/object/urn%3Auuid%3A27e4043d-75eb-4c4f-9427-0d

response = requests.get(url)
open("dg_soil_moisture.csv", "wb").write(response.content)
```

121002029

In the Arctic Data Center metadata we can see this file is 115MB. To import this file as `a dask.dataframe` with more than one partition we need to specify the size of each partition with the `blocksize` parameter. In this example we will split the data frame into six partitions, which would mean a blocksize of approximately 20 MB.

> **Note**
>
> *About the encoding parameter:*
> If we try to import the file directly we will obtain the error `UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb3 in position 192: invalid start byte`. This means XXXX. To find the encoding of the file and add the appropiate encodign to `dask.dataframe.read_csv` we can run the following code.
>
> ```
> # import chardet
> # fp = os.path.join(os.getcwd(),'dg_soil_moisture.csv')
> # with open(fp, 'rb') as rawdata:
> #     result = chardet.detect(rawdata.read(100000))
> # result
> ```

```
import dask.dataframe as dd

fp = os.path.join(os.getcwd(),'dg_soil_moisture.csv')
df = dd.read_csv(fp, blocksize = '20MB' , encoding='ISO-8859-1')
```

Notice that we cannot see any values in the data frame. This is because Dask has not really loaded the data, it will wait until we explicitely ask it to print or compute something to do so.

However, we can still do `df.head()`, it's not costly for memory to just access a few rows of the data frame.

```
df.head()
```

```
/opt/hostedtoolcache/Python/3.9.13/x64/lib/python3.9/site-packages/IPython/core/formatters.py:3

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `Styl
```

	timestamp	year	doy	hour	minute	site	logger	port	sensor	se
0	2014-07-07 16:30:00	2014	188	16	30	MDF1	MDF1met	Port 3	5TM Moisture/Temp	
1	2014-07-07 16:30:00	2014	188	16	30	MDF1	MDF1met	Port 4	5TM Moisture/Temp	
2	2014-07-07 17:00:00	2014	188	17	0	LDF2	LDF2met	Port 3	5TM Moisture/Temp	
3	2014-07-07 17:00:00	2014	188	17	0	LDF2	LDF2met	Port 4	5TM Moisture/Temp	
4	2014-07-07 17:00:00	2014	188	17	0	MDF1	MDF1met	Port 3	5TM Moisture/Temp	

10.3.2 Lazy Computations

The application programming interface (API) of a `dask.dataframe` is a subset of the `pd.DataFrame` API. So if you are familiar with pandas, many of the core `pandas.DataFrame` methods directly translate to `dask.dataframes`.

```
# an example of a simple df transformation
```

Notice that XXXX. A major difference between `pandas.DataFrames` and `dask.dataframes`and is that `dask.dataframes` are *lazy*. This means that an object will queue transformations and calculations without executing them, up until we explicitly ask for the result of that chain of computations using the `compute` method. Once we run `compute`, the scheduler can allocate memory and workers to execute the computations in parallel. This kind of **lazy computations** is how most Dask workloads work. This varies from **eager evaluation** methods and functions, which start computing results right when they are executed.

10.3.3 Task Graph

Lead into task graphs: XXX

```
### THIS IS AN EXAMPLE OF TASK GRAPH

##!! cannot run graph after installing graphviz with
#pip install graphviz

# Error:
# ExecutableNotFound: failed to execute PosixPath('dot'), make sure the Graphviz executables

# See:
#https://stackoverflow.com/questions/42014458/dask-not-installing-graphviz-dependency
```

10.4 `dask.arrays`

Another common object we might want to parallelize is a NumPy array. The equivalent Dask object is the `dask.array`, which coordinates many NumPy arrays that may live on disk or other machines. Each of these NumPy arrays within the `dask.array` is called a **chunk**. Choosing how these chunks are arranged within the `dask.array` and their size can significantly affect the performance of our code. Here you can find more information about chunks and best practices.

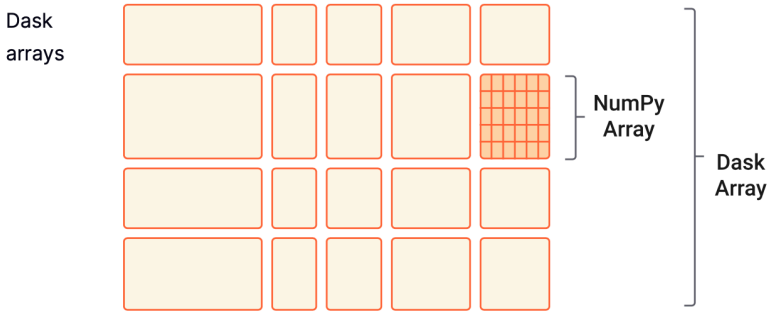


Figure 10.3: Dask Array design (dask documentation

```
import dask.array as da
import numpy as np
```

```
data = np.arange(100_000).reshape(200, 500)
a = da.from_array(data, chunks=(100, 100))
```

Indexing Dask collections feels just like slicing NumPy arrays:

```
a[:2, 2]
```

And computations also work lazily:

```
a.mean()
```

```
a.mean().compute()
```

10.5 Dask and xarray

Going forward, it might be more common having to read some big array-like dataset (like a high-resolution multiband raster) than creating one from scratch from a NumPy array. In this case it can be useful to use the `xarray` module together with Dask. It is simple to wrap Dask around `xarray` objetcs, we only need to specify the number of chunks argument as an argument when we are reading in a dataset (see also [1]).

10.5.1 Open .tif file

As an example, let's do an NDVI calculation using remote sensing imagery collected by aerial vehicles over northeastern Siberia (Loranty, Forbath, Talucci, Alexander, DeMarco, et al. 2020. Uncrewed aerial vehicle remote sensing imagery of postfire vegetation in Siberian larch forests 2018-2019. Arctic Data Center. doi:10.18739/A2ZC7RV6H.).

First we download the data for the near-infrared (NIR) and red bands:

```
# download red band
url = 'https://arcticdata.io/metacat/d1/mn/v2/object/urn%3Auuid%3Aac25a399-b174-41c1-b6d3-09
response = requests.get(url)
open("RU_ANS_TR2_FL005M_red.tif", "wb").write(response.content)

# download nir band
url = 'https://arcticdata.io/metacat/d1/mn/v2/object/urn%3Auuid%3A1762205e-c505-450d-90ed-d4
response = requests.get(url)
open("RU_ANS_TR2_FL005M_nir.tif", "wb").write(response.content)
```

```
79182870
```

Because these are .tif files and have geospatial metadata, we will use `xarray.open_rasterio` to read them. We need to specify either a shape or the size in bytes for each chunk to indicate we will open these files with `dask.arrays` as the underlying object to the `xarray.DataArray` (instead of a `numpy.array`). Both files are 76 MB, so let's have chunks of 15 MB to have roughly six chunks.

```
import xarray as xr
```

```
# read in the file
fp_red = os.path.join(os.getcwd(),"RU_ANS_TR2_FL005M_red.tif")
red = xr.open_rasterio(fp_red, chunks = '15MB')
```

```
/tmp/ipykernel_53126/1226644855.py:3: DeprecationWarning:

open_rasterio is Deprecated in favor of rioxarray. For information about transitioning, see: ht
```

We can see a lot of useful information here:

- There are eight chunks in the array. We were aiming for six, but this often happens with the way Dask distributes the memory (76MB is not divisible by 6).
- There is geospatial information (transformation, CRS, resolution) and no-data values.

- There is an unnecessary dimension: a constant value for the band. So our next step is to squeeze the array to flatten it.

```
# getting rid of unnecessary dimension
red = red.squeeze()
```

Next we read in the NIR band and do the same pre-processing.

```
fp_nir = os.path.join(os.getcwd(),"RU_ANS_TR2_FL005M_nir.tif")
nir = xr.open_rasterio(fp_nir, chunks = '15MB')
print('shape before squeeze:', nir.shape)
print('chunks before squeeze:', nir.chunks )
nir = nir.squeeze()
print('shape after squeeze:', nir.shape)
print('chunk after squeeze:', nir.chunks )
```

```
shape before squeeze: (1, 3499, 7443)
chunks before squeeze: ((1,), (1936, 1563), (1936, 1936, 1936, 1635))
shape after squeeze: (3499, 7443)
chunk after squeeze: ((1936, 1563), (1936, 1936, 1936, 1635))
```

```
/tmp/ipykernel_53126/1858536642.py:2: DeprecationWarning:

open_rasterio is Deprecated in favor of rioxarray. For information about transitioning, see: ht
```

10.5.2 Calculating NDVI

Next we set up the NDVI calculation. This step is easy because we can handle xarrays and Dask arrays as numpy arrays for arithmetic operations. Also, both bands have values of type float32, so we won't have trouble with the division.

```
ndvi = (nir - red) / (nir + red)
```

When we look at the ndvi we can see the result is another `dask.array`, nothing has been computed yet. Remember Dask

computations are lazy, so we need to call `compute()` to actually bring the results to memory.

```
ndvi_values = ndvi.compute()
```

And finally we can see how these look like. Notice that `xarray` uses the value of the dimensions as labels along the x and y axes. We use `robust=True` to ignore the no-data values when plotting.

```
ndvi_values.plot(robust=True)
```

10.6 Best Practices

It is important to remember that, while APIs may be similar, some differences do exist. Additionally, the performance of some algorithms may differ from their in-memory counterparts due to the advantages and disadvantages of parallel programming. Some thought and attention is still required when using Dask. https://docs.dask.org/en/latest/user-interfaces.html

> **Warning**
>
> For data that fits into RAM, Pandas can often be faster and easier to use than Dask DataFrame. While "Big Data" tools can be exciting, they are almost always worse than normal data tools while those remain appropriate. https://docs.dask.org/en/stable/dataframe-best-practices.html

```
- https://docs.dask.org/en/stable/best-practices.html#load-data-with-dask
- https://coiled.io/blog/common-dask-mistakes/
```

Overhead

Choosing chunking

11 Spatial and Image Data Using GeoPandas

- Manipulating raster data with `rasterio`
- Manipulating vector data with `geopandas`
- Working with raster and vector data together

11.1 Introduction

In this lesson, we'll be working with geospatial raster and vector data to do an analysis on vessel traffic in south central Alaska. If you aren't already familiar, geospatial vector data consists of points, lines, and/or polygons, which represent locations on the Earth. Geospatial vector data can have differing geometries, depending on what it is representing (eg: points for cities, lines for rivers, polygons for states.) Raster data uses a set of regularly gridded cells (or pixels) to represent geographic features.

Both geospatial vector and raster data have a coordinate reference system, which describes how the points in the dataset relate to the 3-dimensional sphereoid of Earth. A coordinate reference system contains both a datum and a projection. The datum is how you georeference your points (in 3 dimensions!) onto a spheroid. The projection is how these points are mathematically transformed to represent the georeferenced point on a flat piece of paper. All coordinate reference systems require a datum. However, some coordinate reference systems are "unprojected" (also called geographic coordinate systems). Coordinates in latitude/longitude use a geographic (unprojected) coordinate system. One of the most commonly used geographic coordinate systems is WGS 1984.

Coordinate reference systems are often referenced using a shorthand 4 digit code called an EPSG code. We'll be working with two coordinate reference systems in this lesson with the following codes:

- 3338: Alaska Albers
- 4326: WGS84 (World Geodetic System 1984), used in GPS

In this lesson, we are going to take two datasets:

- Alaskan commercial salmon fishing statisical areas
- North Pacific and Arctic Marine Vessel Traffic Dataset

and use them to calculate the total distance travelled by ships within each fishing area.

The high level steps will be

- read in the datasets
- reproject them so they are in the same projection
- extract a subset of the raster and vector data using a bounding box
- turn each polygon in the vector data into a raster mask
- use the masks to calculate the total distance travelled (sum of pixels) for each fishing area

11.2 Pre-processing raster data

First we need to load in our libraries. We'll use `geopandas` for vector manipulation, `rasterio` for raster maniupulation, and `shapely` for manipulating geospatial data generally.

```
import geopandas as gpd

import rasterio
import rasterio.mask
import rasterio.warp
import rasterio.plot
from rasterio import features
```

```
from shapely.geometry import box

import requests

import matplotlib.pyplot as plt
import matplotlib.ticker

import pandas as pd
import numpy as np
```

First, we'll use `requests` to download the ship traffic raster from Kapsar et al.. We grab a one month slice from August, 2020 of a coastal subset of data with 1km resolution. To get the URL in the code chunk below, you can right click the download button for the file of interest and select "copy link address."

```
url = 'https://arcticdata.io/metacat/d1/mn/v2/object/urn%3Auuid%3A6b847ab0-9a3d-4534-bf28-3a

response = requests.get(url)
open("Coastal_2020_08.tif", "wb").write(response.content)
```

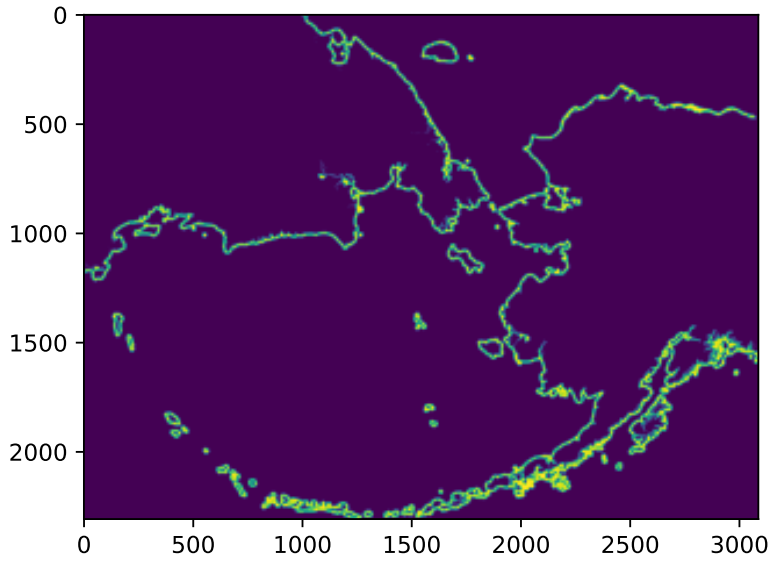
```
1473505
```

Open the raster file, plot it, and look at the metadata. We use the `with` here as a context manager. This ensures that the connection to the raster file is closed and cleaned up when we are done with it.

```
with rasterio.open("Coastal_2020_08.tif") as ship_con:
    # read in raster (1st band)
    ships = ship_con.read(1)
    ships_meta = ship_con.profile

plt.imshow(ships)
print(ships_meta)
```

```
{'driver': 'GTiff', 'dtype': 'float32', 'nodata': -3.3999999521443642e+38, 'width': 3087, 'heig
       0.0, -999.9687691991521, 2711703.104608573), 'tiled': False, 'compress': 'lzw', 'interle
```



Now download a vector shapefile of commercial fishing districts in Alaska.

```
url = 'https://knb.ecoinformatics.org/knb/d1/mn/v2/object/urn%3Auuid%3A7c942c45-1539-4d47-b4

response = requests.get(url)
open("Alaska_Commercial_Salmon_Boundaries.gpkg", "wb").write(response.content)
```

```
36544512
```

Read in the data

```
comm = gpd.read_file("Alaska_Commercial_Salmon_Boundaries.gpkg")

comm.crs
```

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
```

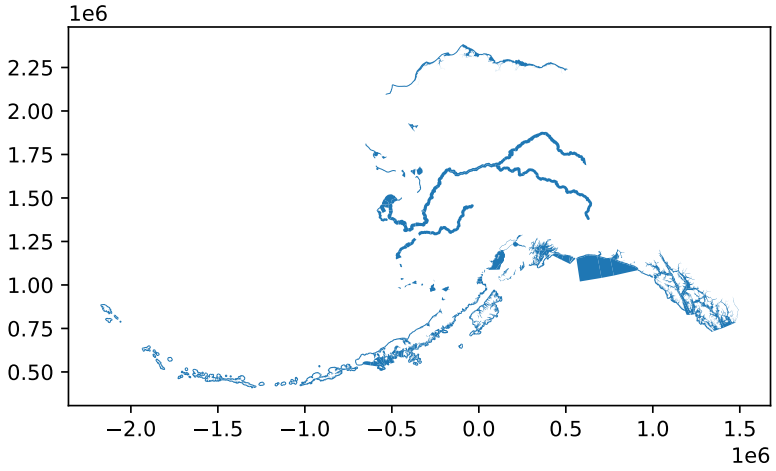
```
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

The raster data is in 3338, so we need to reproject this. We use the `to_crs` method on the `comm` object to transform it from 4326 (WGS 84) to 3338 (Alaska Albers).

```
comm.crs
comm_3338 = comm.to_crs("EPSG:3338")

comm_3338.plot()
```

```
<AxesSubplot:>
```



We can create a bounding box for the area of interest, and use that to clip the original raster data to just the extent we need. We use the `box` function from `shapely` to create the bounding box, then create a `geoDataFrame` from the points, and convert the WGS 84 coordinates to the Alaska Albers projection.

```
coord_box = box(-159.5, 55, -144.5, 62)

coord_box_df = gpd.GeoDataFrame(
    crs = 'EPSG:4326',
    geometry = [coord_box]).to_crs("EPSG:3338")
```

Now, we can read in raster again cropped to bounding box. We use the `mask` function from `rasterio.mask`. Note that we apply this to the connection to the raster file (`with rasterio.open(...)`), update the metadata associated with the raster, and then write it back out again.

```
with rasterio.open("Coastal_2020_08.tif") as ship_con:
    out_image, out_transform = rasterio.mask.mask(ship_con, coord_box_df["geometry"], crop=T
    out_meta = ship_con.meta
```

```
out_meta.update({"driver": "GTiff",
                 "height": out_image.shape[1],
                 "width": out_image.shape[2],
                 "transform": out_transform,
                 "compress": "lzw"})

with rasterio.open("Coastal_2020_08_masked.tif", "w", **out_meta) as dest:
    dest.write(out_image)
```

11.2.1 Check extents

Let's read in the clipped raster data, and make a quick plot to ensure they line up the way they should.

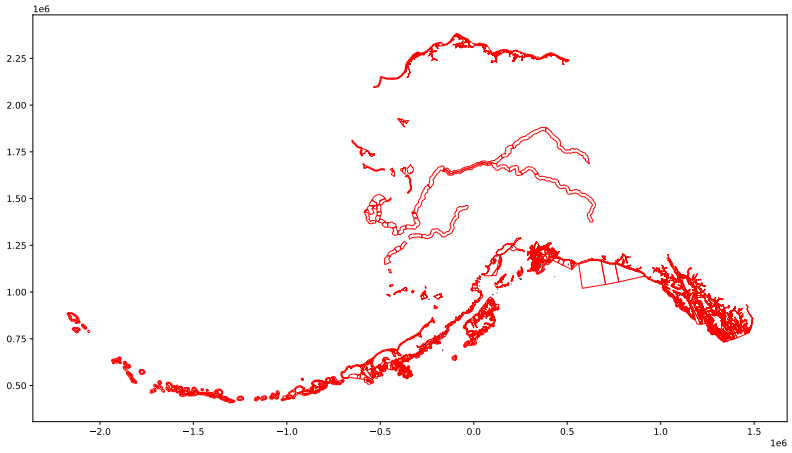
First we read in the cropped data again, since we'll need it later. We also save the `shape` and `transform` attributes of the raster into their own attributes.

```
with rasterio.open('Coastal_2020_08_masked.tif') as ship_con:
    shape = ship_con.shape
    transform = ship_con.transform
    # read in the cropped raster (1st band only)
```

```
    ship_arr = ship_con.read(1)
    # turn no data values into actual NaNs
    ship_arr[ship_arr == ship_con.nodata] = np.nan
```

```
# set up plot
fig, ax = plt.subplots(figsize=(15, 15))
# plot the raster
rasterio.plot.show(ship_arr,
                   ax=ax,
                   vmin = 0,
                   vmax = 50000,
                   transform = transform)
# plot the vector
comm_3338.plot(ax=ax, facecolor='none', edgecolor='red')
```

```
<AxesSubplot:>
```

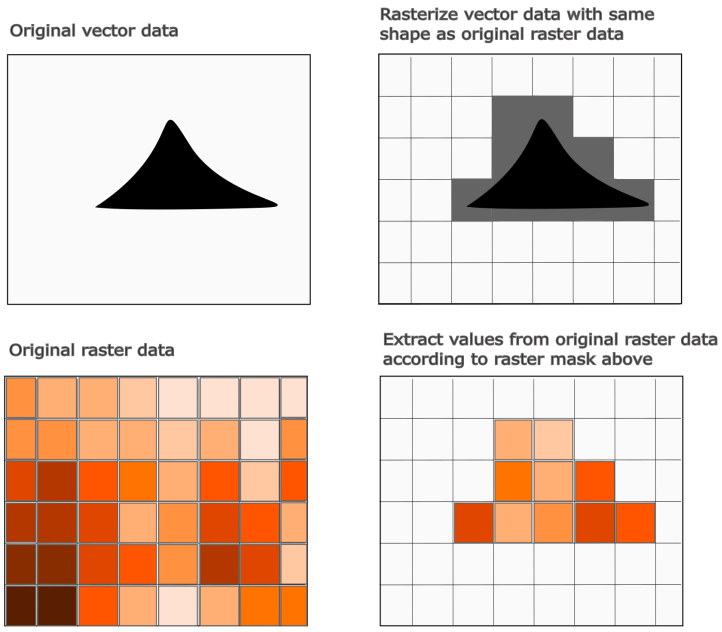


11.3 Calculate total distance per fishing area

In this step, we rasterize each polygon in the shapefile, such that pixels in or touching the polygon get a value of 1, and pixels not touching it get a value of 0. Then, for each polygon, we extract the indices of the raster array that are equal to 1.

We then extract the values of these indicies from the original ship traffic raster data, and calculate the sum of the values over all of those pixels.

Here is a simplified diagram of the process:



11.3.0.1 Zonal statistics over one polygon

Let's look at how this works over just one fishing area first. We use the `rasterize` method from the `features` module in `rasterio`. This takes as arguments the data to rasterize (in this case the 40th row of our dataset), the shape and transform the output raster will take (these were extracted from our raster data when we read it in). We alo set the `all_touched` argument to true, which means any pixel that touches a boundary of our vector will be burned into the mask.

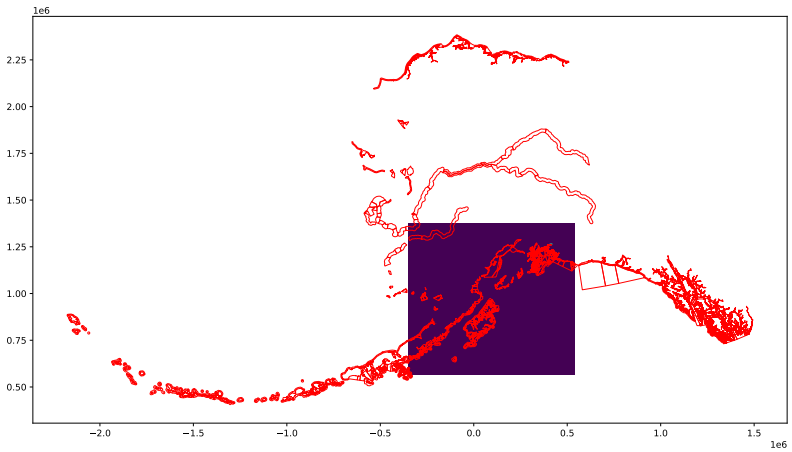
```
r40 = features.rasterize(comm_3338['geometry'][40].geoms,
                                        out_shape=shape,
```

```
                                    transform=transform,
                                    all_touched=True)
```

If we have a look at a plot of our rasterized version of the single fishing district, we can see that instead of a vector, we now have a raster with the shape of the district.

```
# set up plot
fig, ax = plt.subplots(figsize=(15, 15))
# plot the raster
rasterio.plot.show(r40,
                   ax=ax,
                   vmin = 0,
                   vmax = 1,
                   transform = transform)
# plot the vector
comm_3338.plot(ax=ax, facecolor='none', edgecolor='red')
```

```
<AxesSubplot:>
```



A quick call to `np.unique` shows our unique values are 0 or 1, which is what we expect.

```
np.unique(r40)
```

```
array([0, 1], dtype=uint8)
```

Finally, we need to know is the indices of the original raster where the fishing district is. We can use `np.where` to extract this information

```
r40_index = np.where(r40 == 1)
print(r40_index)
```

```
(array([108, 108, 108, 108, 108, 109, 109, 109, 109, 109, 109, 109, 109,
       109, 109, 110, 110, 110, 110, 110, 110, 110, 110, 110, 110, 110,
       110, 110, 110, 111, 111, 111, 111, 111, 111, 111, 111, 111, 111,
       111, 111, 111, 111, 112, 112, 112, 112, 112, 112, 112, 112, 112,
       112, 112, 112, 112, 112, 112, 113, 113, 113, 113, 113, 113, 113,
       113, 113, 113, 113, 113, 113, 113, 113, 114, 114, 114, 114, 114,
       114, 114, 114, 114, 114, 114, 115, 115, 115, 115, 115, 115, 116]), array([759, 760, 762
       764, 765, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763,
       764, 765, 766, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762,
       763, 764, 765, 766, 753, 754, 755, 756, 757, 758, 759, 760, 761,
       762, 763, 764, 765, 766, 767, 753, 754, 755, 756, 757, 758, 759,
       760, 761, 762, 763, 764, 765, 766, 767, 753, 754, 755, 756, 757,
       758, 759, 760, 761, 762, 763, 753, 754, 755, 756, 757, 758, 754]))
```

In the last step, we'll using these indices to extract the values of the data from the fishing raster, and sum them to get a total distance travelled.

```
np.nansum(ship_arr[r40_index])
```

```
14369028.0
```

Now that we know the individual steps, let's run this over all of the districts. First we'll create an `id` column in the vector data frame. This will help us track unique fishing districts later.

```
comm_3338['id'] = range(0,len(comm_3338))
```

For each district (with `geometry` and `id`), we run the `features.rasterize` function. If any values equal 1 (some of

the districts are outside the bounds of the raster), we calculate the sum of the values of the shipping raster `r_array` based on the indicies in the raster where the district is located.

```
distance_dict = {}
for geom, idx in zip(comm_3338['geometry'], comm_3338['id']):
    rasterized = features.rasterize(geom.geoms,
                                    out_shape=shape,
                                    transform=transform,
                                    all_touched=True)
    # only save polygons that have a non-zero value
    if any(np.unique(rasterized)) == 1:
        r_index = np.where(rasterized == 1)
        distance_dict[idx] = np.nansum(ship_arr[r_index])
```

Now we just create a data frame from that dictionary, and join it to the vector data using `pandas` operations.

```
# create a data frame from the result
distance_df = pd.DataFrame.from_dict(distance_dict,
    orient='index',
    columns=['distance'])

# extract the index of the data frame as a column to use in a join
distance_df['id'] = distance_df.index
distance_df['distance'] = distance_df['distance']/1000
```

Now we join the result to the original geodataframe.

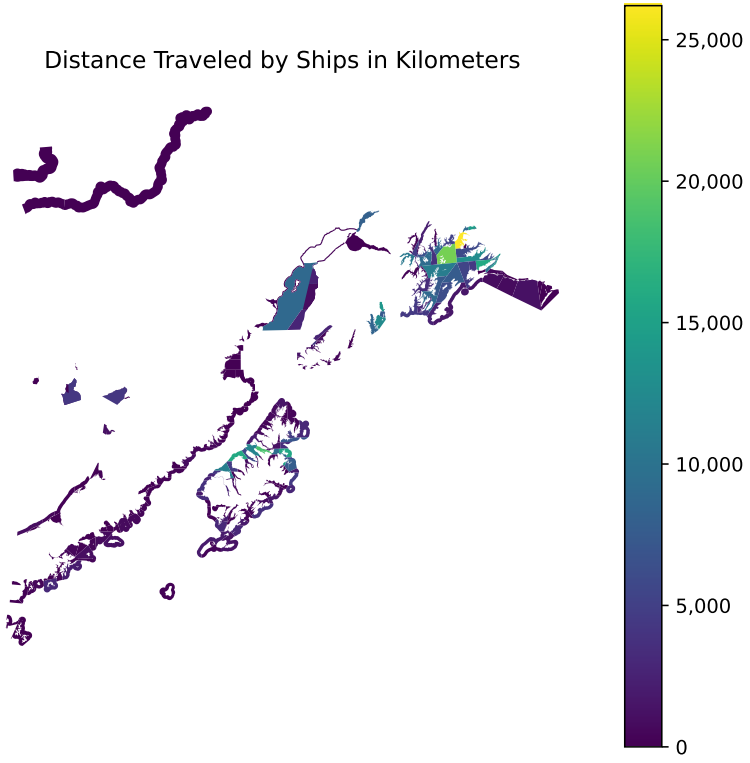
```
# join the sums to the original data frame
res_full = comm_3338.merge(distance_df, on = "id", how = 'inner')
```

Finally, we can plot our result!

```
fig, ax = plt.subplots(figsize=(7, 7))

ax = res_full.plot(column = "distance", legend = True, ax = ax)
fig = ax.figure
label_format = '{:,.0f}'
cb_ax = fig.axes[1]
```

```
ticks_loc = cb_ax.get_yticks().tolist()
cb_ax.yaxis.set_major_locator(matplotlib.ticker.FixedLocator(ticks_loc))
cb_ax.set_yticklabels([label_format.format(x) for x in ticks_loc])
ax.set_axis_off()
ax.set_title("Distance Traveled by Ships in Kilometers")
plt.show()
```



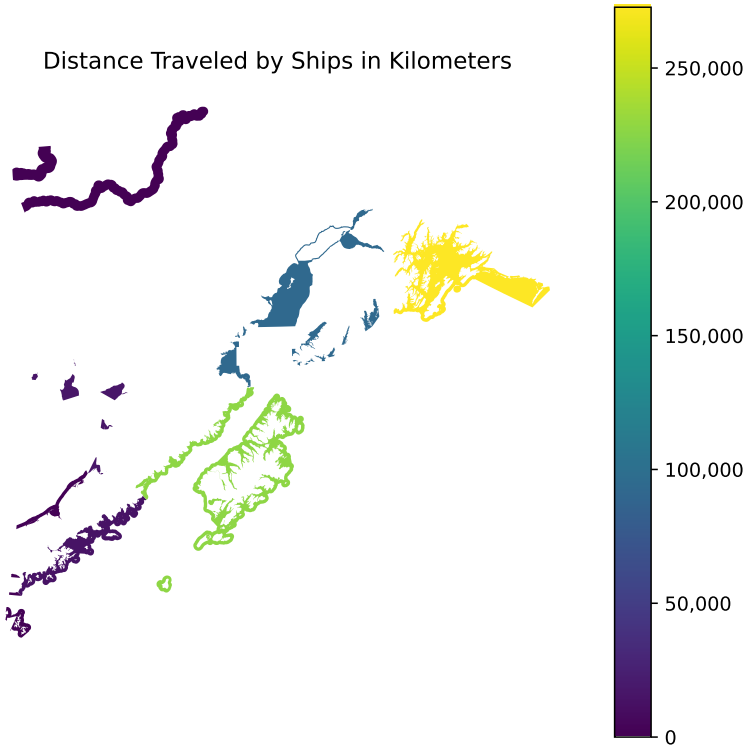
From here we can do any additional `geopandas` operations we might be interested in. For example, what if we want to calculate the total distance by registration area (a superset of fishing district). We can do that using `dissolve` from geopandas.

```
reg_area = res_full.dissolve(by = "REGISTRATION_AREA_NAME", aggfunc = 'sum')
```

Let's have a look at the same plot as before, but this time over our aggregated data.

```
fig, ax = plt.subplots(figsize=(7, 7))

ax = reg_area.plot(column = "distance", legend = True, ax = ax)
fig = ax.figure
label_format = '{:,.0f}'
cb_ax = fig.axes[1]
ticks_loc = cb_ax.get_yticks().tolist()
cb_ax.yaxis.set_major_locator(matplotlib.ticker.FixedLocator(ticks_loc))
cb_ax.set_yticklabels([label_format.format(x) for x in ticks_loc])
ax.set_axis_off()
ax.set_title("Distance Traveled by Ships in Kilometers")
plt.show()
```



12 Data Futures: Parquet and Arrow

- The difference between column major and row major data
- Speed advantages to columnnar data storage
- How `arrow` enables faster processing

12.1 Introduction

System calls are calls that are run by the operating system within their own process. There are several that are relevant to reading and writing data: open, read, write, seek, and close. Open establishes a connection with a file for reading, writing, or both. On open, a file offset points to the beginning of the file. After reading or writing `n` bytes, the offset will move `n` bytes forward to prepare for the next opration. Close closes the connection to the file. Read will read data from the file into a memory buffer, and write will write data from a memory buffer to a file. Seek is used to change the location of the offset pointer, for either reading or writing purposes.

If you've worked with even moderately sized datasets, you may have encounted an "out of memory" error. Memory is where a computer stores the information needed immediately for processes. This is in contrast to storage, which is typically slower to access than memory, but has a much larger capacity. When you `open` a file, you are establishing a connection between your processor and the information in storage. On `read`, the data is read into memory that is then available to your python process, for example.

So what happens if the data you need to read in are larger than your memory? 32GB is a common memory size, but this would be considered a modestly sized dataset by this courses's

standards. There are a number of solutions to this problem, which don't involve just buying a computer with more memory. In this lesson we'll discuss the difference between row major and column major file formats, and how leveraging column major formats can increase memory efficiency. We'll also learn about another python library called `pyarrow`, which has a memory format that allows for "zero copy" read.

12.2 Row major vs column major

The difference between row major and column major is in the ordering of items in the array.

Take the array:

```
a11 a12 a13

a21 a22 a23
```

This array in a row-major order would be read in as:

`a11, a12, a13, a21, a22, a23`

You could also read it in column-major order as:

`a11, a21, a12, a22, a13, a33`

By default, C and SAS use row major order for arrays, and column major is used by Fortran, MATLAB, R, and Julia.

Python uses neither, instead representing arrays as lists of lists, though `numpy` uses row-major order.

12.2.1 Row major versus column major files

The same concept can be applied to file formats as the example with arrays above. In row-major file formats, the values (bytes) of each record are read sequentially.

Name	Location	Age
John	Washington	40
Mariah	Texas	21
Allison	Oregon	57

In the above row major example, data are read in the order: `John, Washingon, 40, [new line], Mariah, Texas, 21`.

This means that getting a subset of all columns would be easy; you can specify to read in only the first X rows. However, if we are only interested in `Name` and `Location`, we would still have to read in all of the rows before discarding the Age column.

If these data were organized in a column major format, they might look like this:

```
Name: John, Mariah, Allison
Location: Washington, Texas, Oregon
Age: 40, 21, 57
```

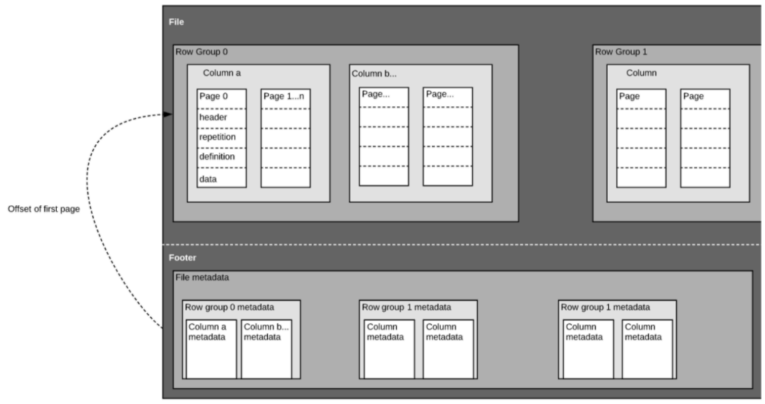
And the read order would first be the names, then the locations, then the age. This means that selecting all values from a set of columns is quite easy (all of the Names and Ages, or all Names and Locations), but reading in only the first few records from each column would require reading in the entire dataset. Another advantage to column major formats is that compression is more efficient since compression can be done across each column, where the data type is uniform, as opposed to across rows with many data types.

12.3 Parquet

Parquet is an open-source file format that stores data in a column-major format. The format contains several key components:

- row group
- column

- page
- footer



Row groups are blocks of data over a set number of rows that contain data from the same columns. Within each row group, data are organized in column-major format, and within each column are pages that are typically 1MB. The footer of the file contains metaata like the schema, encodings, unique values in each column, etc.

The parquet format has many tricks to to increase storage efficiency, and is increasingly being used to handle large datasets.

12.4 Arrow

So far, we have discussed the difference between organizing information in row-major and column-major format, how that applies to arrays, and how it applies to data storage on disk using Parquet.

Arrow is a language-agnostic specification that enables representation of column-major information in memory without having to serialize data from disk. The Arrow project provides implementation of this specification in a number of languages, including Python.

Let's say that you have utilized the Parquet data format for more efficient storage of your data on disk. At some point, you'll need to read that data into memory in order to do analysis on it. Arrow enables data transfer between the on disk Parquet files and in-memory Python computations, via the `pyarrow` library.

`pyarrow` is great, but relatively low level. It supports basic group by and aggregate functions, as well as table and dataset joins, but it does not support the full operations that `pandas` does.

12.5 Example

In this example, we'll read in a dataset of fish abundance in the San Francisco Estuary, which is published in csv format on the Environmental Data Initiative. This dataset isn't huge, but it is big enough (3 GB) that working with it locally can be fairly taxing on memory. Motivated by user difficulties in actually working with the data, the `deltafish` R package was written using the R implementation of `arrow`. It works by downloading the EDI repository data, writing it to a local cache in parquet format, and using `arrow` to query it. In this example, I've put the Parquet files in a sharable location so we can explore it using `pyarrow`.

First, we'll load the modules we need.

```
import pyarrow.dataset as ds
import numpy as np
import pandas as pd
```

Next we can read in the data using `ds.dataset()`, passing it the path to the parquet directory and how the data are partitioned.

```
deltafish = ds.dataset("/home/shares/deltafish/fish", format="parquet", partitioning=["Speci
```

You can check out a file listing using the `files` method. Another great feature of parquet files is that they allow you to

partition the data accross variables of the dataset. These partitions mean that, in this case, data from each species of fish is written to it's own file. This allows for even faster operations down the road, since we know that users will commonly need to filter on the species variable. Even though the data are partitioned into different files, `pyarrow` knows that this is a single dataset, and you still work with it by referencing just the directory in which all of the partitioned files live.

```
deltafish.files
```

```
['/home/shares/deltafish/fish/Taxa=Acanthogobius flavimanus/part-0.parquet',
 '/home/shares/deltafish/fish/Taxa=Acipenser medirostris/part-0.parquet',
 '/home/shares/deltafish/fish/Taxa=Acipenser transmontanus/part-0.parquet',
 '/home/shares/deltafish/fish/Taxa=Acipenser/part-0.parquet'...
```

You can view the columns of a dataset using `schema.to_string()`

```
deltafish.schema.to_string()
```

```
SampleID: string
Length: double
Count: double
Notes_catch: string
Species: string
```

If we are only interested in a few species, we can do a filter:

```
expr = ((ds.field("Species")=="Dorosoma petenense")|
        (ds.field("Species")=="Morone saxatilis") |
        (ds.field("Species")== "Spirinchus thaleichthys"))

fishf = deltafish.to_table(filter = expr)
```

There is another dataset included, the survey information. To do a join, we can just use the `join` method on the `arrow` dataset.

First read in the survey dataset.

```
survey = ds.dataset("/home/jclark/deltafish/survey", format="parquet", partitioning=["Source
```

```
survey.schema.to_string()
```

Then do the join, and convert to a pandas `data.frame`.

```
fish_j = fishf.join(survey, "SampleID").to_pandas()
```

Note that when we did our first manipulation of this dataset, we went from working with a `FileSystemDataset`, which is a representation of a dataset on disk without reading it into memory, to a `Table`, which is read into memory. `pyarrow` has a number of functions that do computations on datasets without reading them into memory. However these are evaluated "eagerly," as opposed to "lazily." These are useful in some cases, like above, where we want to take a larger than memory dataset and generate a smaller dataset (via filter, or group by/summarize), but are not as useful if we need to do a join before our summarization/filter.

More functionality for lazy evaluation is on the horizon for `pyarrow` though, by leveraging Ibis.

13 Software Design II

14 Group Project: Data Processing

In your fork of the `scalable-computing-examples` repository, open the Jupyter notebook in the `group-project` directory called `session-13.ipynb`. This workbook will serve as a skeleton for you to work in. It will load in all the libraries you need, including a few helper functions we wrote for the course, show an example for how to use the method on one file, and then lays out blocks for you and your group to fill in with code that will run that method in parallel.

In your small groups, work together to write the solution, but everyone should aim to have a working solution on their own fork of the repository. In other words, everyone should type out the solution themselves as part of the group effort. Writing the code out yourself (even if others are contributing to the content) is a great way to get "mileage" as you develop these skills.

15 Data Ethics

16 Google Earth Engine

SAM NOTES, DELETE LATER - need to make sure that .ipynb that student's will work out of are running in same virutal enviroment as everything else - embed .ipynb into quarto notebook (get book to build from those examples) - use Ryan Abernathey's post to help frame introduction

- Understand what Google Earth Engine provides and its applications
- Learn about some real-world applications of Google Earth Engine
- Learn how to get started using Google Earth Engine on your own computer
- Learn how to find and access Google Earth Engine Data

16.1 Introduction (15-20min)

Google Earth Engine (GEE) is a geospatial processing platform powered by Google Cloud Platform. It contains over 30 years (and petabytes) of satellite imagery and geospatial datasets that are continually updated and available instantly. Users can process data using Google Cloud Platform and built-in algorithms or by using the Earth Engine API, which is available in Python (and JavaScript) for anyone with an account (a free tier service is available).

So why should we be excited about GEE? As data have gotten larger, the typical download-data-work-locally workflow is no longer always feasible. GEE offers web access to an extensive catalog of analysis-ready geospatial data and scalable computing power via their cloud service, making global-scale analyses and visualizations possible for anyone with an account (sign up here!)

Explore the public Earth Engine Data Catalog which includes a variety of standard Earth science raster datasets. Browse by dataset tags or by satellite (Landsat, MODIS, Sentinel).

16.2 Exercise 1: Getting started with Google Earth Engine (GEE) – mapping

1. Create a Google Earth Engine account (if you haven't already done so)

- Find instructions on how to do so here

> ❗ Move this to course setup
>
> 1. Sign up for a GEE Account
>
> - GEE is currently free for educational use. Sign up for an account at https://signup.earthengine.google.com, which will take you to a form that looks like this:
>
> - You'll receive an email with some helpful links and a message that it may take a few days for your account to be up and running. **Be sure to do this a few days ahead of needing to use GEE.**

- Install the Google Earth Engine API in your `gee_env`

```
#| eval: false
conda install -c conda-forge earthengine-api
```

2. Sign up for a GEE Account

GEE is currently free for educational use. Sign up for an account at https://signup.earthengine.google.com (you'll need this to authenticate in the next step).

3. Set up GEE Authentication

In order to begin using GEE, you'll need to connect your GEE envionment (`gee_env`) to the authentication credentials associated with your Google account. This will need to be done each time you connect to GEE, but should only be done once per session.

- On the command line, type:

```
#| eval: false
earthengine authenticate
```

This should launch a browser window where you can login with your Google account to the Google Earth Engine Authenticator. Following the prompts will generate a code, which you'll then need to copy and paste back onto the command line. This will be saved as an authentication token so you won't need to go through this process again until the next time you start a new session.

4. Install necessary packages (if you don't already have them)

```
#| eval: false
pip install ee # Earth Engine API package
pip install geemap # package for interactive maping with GEE
pip install pandas # contains useful tools for data manipulation (may not need this)
```

16.3 Visualize global precipitation data using Google Earth Engine

Content for this section was adapted from Dr. Sam Stevenson's Visualizing global precipitation using Google Earth Engine lesson, given in her EDS 220 course in Fall 2021.

1. Import necessary packages

```
import ee # MODULENOTFOUNDERROR
import geemap
import pandas as pd
```

2. Create an interactive basemap

The default basemap is (you guessed it) Google Maps. The following code displays an empty Google Map that you can manipulate just like you would in the typical Google Maps interface. Do this using the `Map` method from the `geemap` library. We'll also center the map at a specified latitude and longitude (here, 40N, 100E), set a zoom level, and save our map as an object called `myMap`.

```
myMap = geemap.Map(center = [40, -100], zoom = 2)
myMap
```

3. Load ERA5 Image Collections from GEE

- NOTE: ADC has worked with these data – took 3 weeks to download
- EE colleciton is all you need to load and analyze imgage collection
- precursor to Ingmar's stuff

We'll be using the ERA5 daily aggregates reanalysis dataset, produced by the European Centre for Medium-Range Weather Forecasts (ECMWF), found here, which models atmospheric weather observations. We'll load the `total_precipitation` field (check out the dataset metadata on here).

The `ImageCollection` method extracts a set of individual images that satisfies some criterion that you pass to GEE through the `ee` package. This is stored as an **ImageCollection** object which can be filtered and processed in various ways. We can pass the `ImageCollction` method agruments to tell GEE which data we want to retrieve. Below, we retrieve all daily ERA5 data (so we can see individual rain events).

```
weatherData = ee.ImageCollection('ECMWF/ERA5/DAILY')
```

4. Select an image to plot

To plot a map over our Google Maps basemap, we need an "Image" rather than an "ImageCollection." ERA5 contains many different climate variables, so we need to pick what we'd like to

plot. We'll use the `.select` method to choose the parameter(s) we're interested in from our `weatherData` object.

```
precip = weatherData.select("total_precipitation")
```

We can look at our `precip` object using the `print` method to see that it's still an "ImageCollection" which contains daily infomration from 1979 to 2020.

```
print(precip)
```

We want to filter it down to a single field for a time of interest – let's say December 1-2, 2019. We apply the `.filter` method to our `precip` object and apply the `ee.Filter.date` method (from the `ee` package) to filter for data from our chosen date range. We also apply the `.mean` method, which takes whatever precedes it and calculates the average.

```
precip_filtered = precip.filter(ee.Filter.date('2019-12-01', '2019-12-02')).mean()
```

5. Add data to map

We can fist use the `setCenter` method to tell the map where to center itself. It takes the longitude and latitude as the first two coordinates, followed by the zoom level.

```
Map.setCenter(-152.505706, 59.432367, 2) # Cook Inlet, Alaska (WE CAN CHANGE THIS LOCATION)
```

Next, set a color palette to use when plotting the data layer. The following is a palette specified for precipitation in the GEE description page for ERA5. GEE has lots of color tables like this that you can look up.

```
precip_palette = {
    'min':0,
    'max':0.1,
    'palette': ['#FFFFFF', '#00FFFF', '#0080FF', '#DA00FF', '#FFA400', '#FF0000']
}
```

Finally, plot our filtered data, `precip_filtered` on top of our basemap using the `.addLayer` method. We'll aslo

pass it our visualization parameters (colors and ranges stored in `precip_palette`, the name of the data field `total precipitation`, and opacity so that we can see the basemap underneath)

```
Map.addLayer(precip_filtered, precip_palette, 'total precipitation', opacity = 0.3)
```

16.4 INGMAR'S DEMONSTRATION HERE?(30-40 min)

16.5 Conclusion/Summary

- lessons learned
- utilities
- etc.

16.6 Other Resources

- GEE Code Editor is a web-based IDE for using GEE (JavaScript)

17 Group Project: Visualization

In your fork of the `scalable-computing-examples` repository, open the Jupyter notebook in the `group-project` directory called `session-17.ipynb`. This workbook will serve as a skeleton for you to work in. It will load in all the libraries you need, including a few helper functions we wrote for the course, show an example for how to use the method on one file, and then lays out blocks for you and your group to fill in with code that will run that method in parallel.

In your small groups, work together to write the solution, but everyone should aim to have a working solution on their own fork of the repository. In other words, everyone should type out the solution themselves as part of the group effort. Writing the code out yourself (even if others are contributing to the content) is a great way to get "mileage" as you develop these skills.

18 Workflows for data staging and publishing

- NSF archival policies for large datasets
- Data transfer tools
- How to manage co-locating data and code
  - eg: where model runs only has 1 TB of storage but model outputs 10 TB of data
  - workflow tools (pegasus, condor, slurm, snakemake)
- Uploading large datasets to the Arctic Data Center

18.1 NSF policy for large datasets

- there are many different research methods that can generate large volumes of data. Numerical modeling (such as climate or ocean models) and anything generating high resolution imagery are two examples we see very commonly.

  The Office of Polar Programs policy requires that metadata files, full data sets, and derived data products, must be deposited in a long-lived and publicly accessible archive.

  Metadata for all Arctic supported data sets must be submitted to the NSF Arctic Data Center (https://arcticdata.io).

  Exceptions to the above data reporting requirements may be granted for social science and indigenous knowledge data, where privacy or intellectual property rights might take precedence.

Such requested exceptions must be documented in the Data Management Plan.

- datasets that are already published on a long lived archive do not need to be replicated to the Arctic Data Center
  - example: a research project accesses many terabytes of VIIRS satellite data. The original satellite data does not need to be published on the Arctic Data Center, but the code that accessed it, and derived products, can be published
- for some numerical models, if the model results can be faithfully reproduced from code, the code that generates the models can be a sufficient archival product, as opposed to the code and the model output
  - if the model is difficult to set up, or takes a very long time to run, we would probably reccommend publishing the output as well as code
- the Arctic Data Center is committed to archiving data of any volume

18.2 Data transfer tools

- scenario: you need to send a bunch of data to the Arctic Data Center. after getting the credentials, you use `scp` to start the transfer. You know this typically takes around 12 hours so you start it at 5pm right when you leave the office expecting it to be done when you get back. When you arrive, you see there was a short network outage in the middle of the night. The whole job failed so you have to start it again…

There is a better way!

Three key elements to data transfer

```
- endpoints
- network
- transfer tool
```

18.2.0.1 Endpoints

The from and to locations of the transfer, an endpoint is a remote computing device that can communicate back and forth with the network to which it is connected. The speed with which an endpoint can communicate with the network varies depending on how it is configured. Performance depends on the CPU, RAM, OS, and disk configuration. Examples:

- NCEAS `datateam` server:
- Standard laptop

18.2.0.2 Network speed

Determines how quickly information can be sent between endpoints, largely dependent on what you pay for. Wired networks get significantly more speed than wireless.

- not all networks are created equal
- server to server (north hall to san diego) versus server to your house

18.2.0.3 Transfer tools

- scp
  - uses `ssh` for authentication and transfer
  - if you can `ssh` to a server, you can probably use `scp` to move files without any other setup
  - copies all files linearly and simply. if a transfer fails in the middle, difficult to know exactly what files didn't make it, so you have to start the whole thing over and re-transfer all the files
- rsync
  - similar to `scp` but syncs files/directories as opposed to copying
  - if the file already exists on the other side, it is skipped
- globus

  - parellelizes transfers by utilizing multiple network sockets simultaneously
  - is able to fail and restart itself efficiently
  - requires more setup, endpoints need to be configured as globus nodes

18.2.0.4 Globus

- easy to use, as long as your data are accessible via an endpoint configured as a Globus node
- leverage your institutions computing resources! they may be able to help get you access to a data transfer node already configured correctly
- there are paid options to set up a node from your own workstation (Globus Connect Personal - check the naming here, and feature list)
  - remember the other factors though! Globus won't help you overcome a 1 Gb/s laptop connection speed, or a 50 Mb/s network speed

18.3 Documenting large datasets

- the Arctic Data Center is working to support large datasets, but we have performance considerations as well
- self documenting file formats are preferred, to prevent us from needing to document thousands-millions of files in a single metadata document
  - netcdf
  - geotiff, geopackage
- regular, parseable filenames and consistent file formatting is key
- communicate early and often with the Arctic Data Center staff

19 What is Cloud Computing Anyways?

20 Reproducibility and Containers

- TODO: Decide about if/how to talk about WholeTale
- TODO: This lesson should be have a wow-factor and emphasize why we're focusing all of this
- TODO: This lesson should be more about wrapping up and tying everything together than showing off new tech
- ~~Learn about software versioning~~
- Become familiar with Docker as a tool to improve computational reproducibility

20.1 Outline

- Introduce software reproducibility
  - Motivate the idea with examples and data
  - Talk about software collapse
    - http://blog.khinsen.net/posts/2017/01/13/sustainable-software-and-reproducible-research-dealing-with-software-collapse/
    - https://xkcd.com/2347/
- Semantic versioning and the reality of it e.g., https://pandas.pydata.org/docs/development/policies.html#version-policy
- MyBinder
- WholeTale?

Examples to look at including:

- https://numpy.org/neps/nep-0023-backwards-compatibility.html#example-cases
- https://github.com/scipy/scipy/issues/16418 > https://pandas.pydata.org/docs/whatsnew/v1.4.0.html#de
  DataFrame.append() and Series.append() have been deprecated and will be removed in a future version. Use pandas.concat() instead (GH35407).

Principles to get across:

1. You probably should be thinking about software versioning
    - Know which version of versions of Python your code was written/tested under and keep track of that in a machine-readable way
    - Know the specific versions, of at least the specific MAJOR.MINOR of the packages your code was written+tested under and keep track of them in a machine-readable way (ie requirements.txt)

20.2 Hands-off Demo

Show students an example of containerizing a workflow so it runs using a past version of Python and pinned versions of packages. Ideally find an example where behavior changes based on the Python or one or more package versions.