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

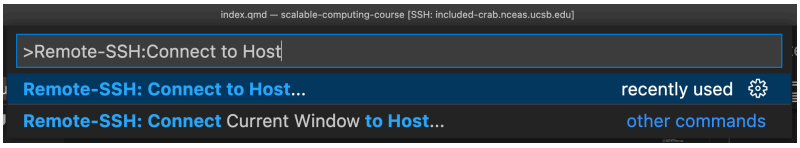
First, download VS Code if you do not already have it installed.

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

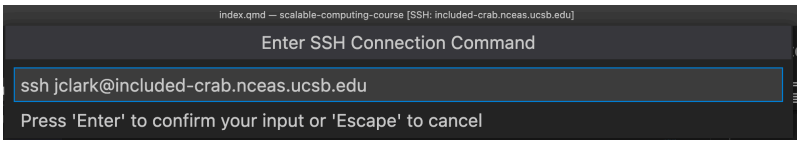
Log in to the server

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"

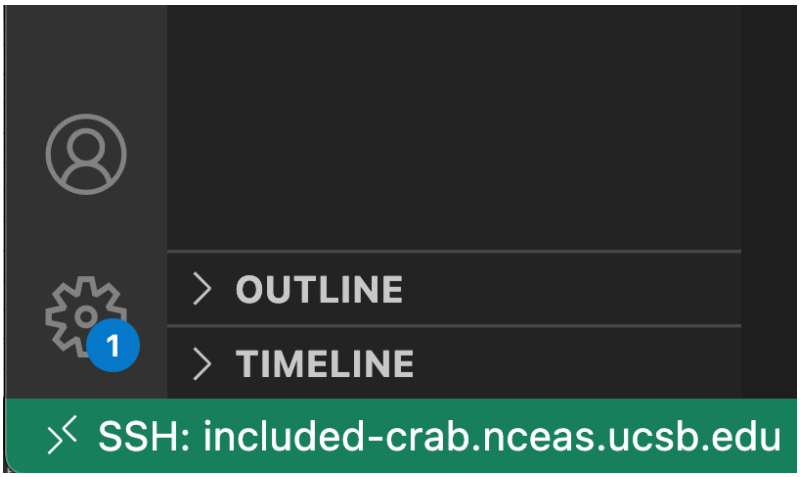


- select "Add New SSH Host"
- enter the ssh command to connect to the host as if in a terminal (`ssh username@included-crab.nceas.ucsb.edu`)
  - Note: you will only need to do this step once



- 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.

This work is licensed under a Creative Commons Attribution 4.0 International License.

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

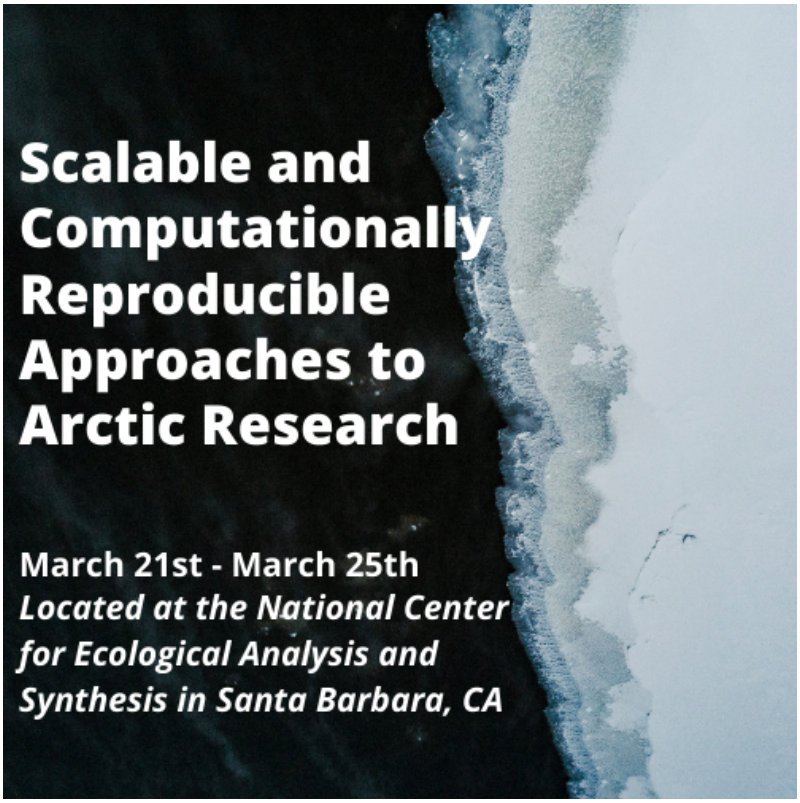
Additional contributors: Amber E. Budden, Natasha Haycock-Chavez, Noor Johnson, Stephanie Hampton, Jim Regetz, Bryce Mecum, Julien Brun, Julie Lowndes, Erin McLean, Andrew Barrett, David LeBauer, Jessica Guo.

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

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. (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. In this example, servers A - I are each nodes on this single cluster. The connection is first made through a gateway node (i.e. a computer that routes traffic from one network to another). (c) A client uses SSH to login/connect to a computing cluser where each node is a virtual machine (VM). In this example, the cluster comprises three servers (A, B, and C). VM1 (i.e. node 1) runs on server A while VM4 runs on server B, etc. The connection is first made through a gateway node.

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 allow multiple applications to communicate with a host without mixing up traffic.

> Port numbers are divided into three ranges:
>
> 1. *well-known ports*, range from 0 through 1023 and are reserved for the most commonly used services (see table below for examples of some well-known

port numbers)
2. *registered ports*, range from 1024 through 49151 and are not assigned or controlled, but can be registered (e.g. by a vendor for use with thier own server application) to prevent duplication
3. *dynamic ports*, range from 49152 through 65535 and are not assigned, controlled, or registered but may instead be used as temporary or private ports

well-known port	assignment
20, 21	File Transfer Protocol (FTP), for transfering files between a client & server
22	secure shell (SSH), to create secure network conections
53	Domain Name System (DNS) service, to match domain names to IP addresses
80	Hypertext Transfer Protocol (HTTP), used in the World Wide Web
443	HTTP Secure (HTTPS), an encrypted version of HTTP

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.

ℹ In order to connect to remote servers, computing clusters, virtual machines, etc., you need to know their IP address (or hostname)

A couple important ones: 1. Throughout this course, we'll be working on a server with the hostname, **included-crab** and IP address, 128.111.85.1 (in just a little bit, we'll learn how to connect to **included-crab** using SSH) 2. **localhost** is a hostname that refers to your local computer

and is assigned the IP address 127.0.0.1 – the concept of localhost is important for tasks such as website testing and also is important to understand when provisioning local execution resources (e.g. we'll practice this in during the section 6 exercise when working with `Parsl`.)

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 and Linux machines
- **Windows:** running bash depends on which version of Windows you have – newer versions may ship with bash or may require a separate install (e.g. Windows Subsystem for Linux (WSL) or Git Bash), however there are a number of different (non-bash) shell options as well (they all vary slightly; e.g. PowerShell, Command Prompt).

> **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 no "trash" folder!)
`grep`	searches a given file(s) for lines containing a match to a given pattern list
`awk`	a text processing language that can be used in shell scripts or at a shell prompt for actions like pattern matching, printing specified fields, etc.
`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
```

2.4.3 Some useful keyboard shortcuts

It can sometimes feel messy working on the command line. These keyboard shortcuts can make it a little easier:

- `Ctrl` + `L`: clear your terminal window
- `Ctrl` + `U`: delete the current line
- `Ctrl` + `C`: abort a command
- up & down arrow keys: recall previously executed commands in chronological order
- TAB key: autocompletion

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.

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 directory in a different location
`git add`	add 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
`git fetch`	downloads contents (e.g. files, commits, refs) from a remote repo to a local repo
`git pull`	fetches contents of a remote repo and merges chnanges into the local repo

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.

> **i** 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 `scalable-computing-exercises` repository on GitHub at https://github.com/NCEAS/scalable-computing-examples – this repo contains example files for you to edit and practice in throughout this course. Fork (make your own copy of the repository) this repo by clicking on the **Fork** button (top right corner of the repository's page).

2. Once forked, click on the green **Code** button (from the *forked* version of the GitHub repo) and copy the URL to your clipboard.

3. In the VS Code terminal, use the `git clone` command to create a copy of the `scalable-computing-examples` repository in the top level of your user directory (i.e. your home directory) on the server (**Note:** use `pwd` to check where you are; use `cd ~/` to navigate back to your home directory if you find that you're somewhere else).

```
git clone <url-of-forked-repo>
```

4. To open the project, open the folder into your workpace using `File` > `Open Folder`. Enter your password, if prompted.

5. You should now have a copy of the `scalable-computing-examples` repository to work on on the server. Use the `tree` command to see the structure of the repo (you need to be in the `scalable-computing-examples` directory for this to work) – there should be a subdirectory called `bash-babynames` that contains (i) a `README.MD` file, (ii) a `KEY.sh` file (this is a functioning bash script available for reference; we'll be recreating it together in the next exercise) and (iii) 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 `scalable-computing-examples` GitHub repository to the server in **Exercise 3** above. This contains a `bash-babynames` folder with 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 `scalable-computing-examples/bash-babynames`. If necessary, navigate here using the `cd` command.
2. Next, we'll create a shell script called `mybash.sh` using the `touch` command:

```
$ touch bybash.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` command 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 Connect to the server (if you aren't already connected)

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

Alternatively, you may see a popup window that says **"Cannot reconnect. Please reload the window"**. Choose the blue **Reload Window** button and enter your password, if prompted.

Once connected, use the `pwd` command to make sure you're in your home directory (`/home/yourusername`). Use the `ls` command to list out the contents of your home directory and ensure that you have a clone of the `scalable-computing-examples` respository (this should have been completed during Exercise 3 in the Remote Computing session).

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         5

                            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()
```

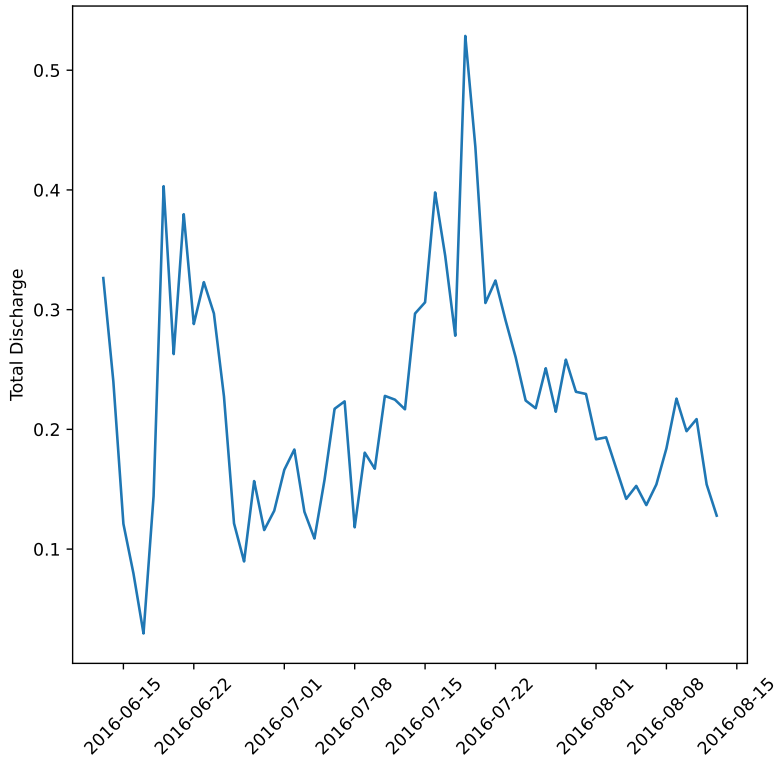
- create a simple plot

```
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

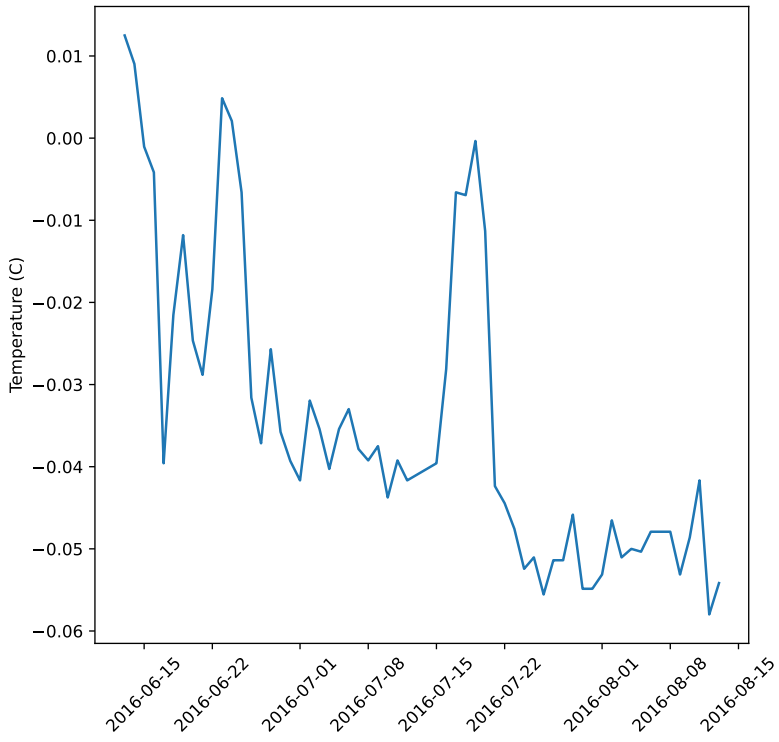
Repeat Yourself! Instead, we'll create a function called `myplot` that accepts the data frame and variable as arguments.

- 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 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.

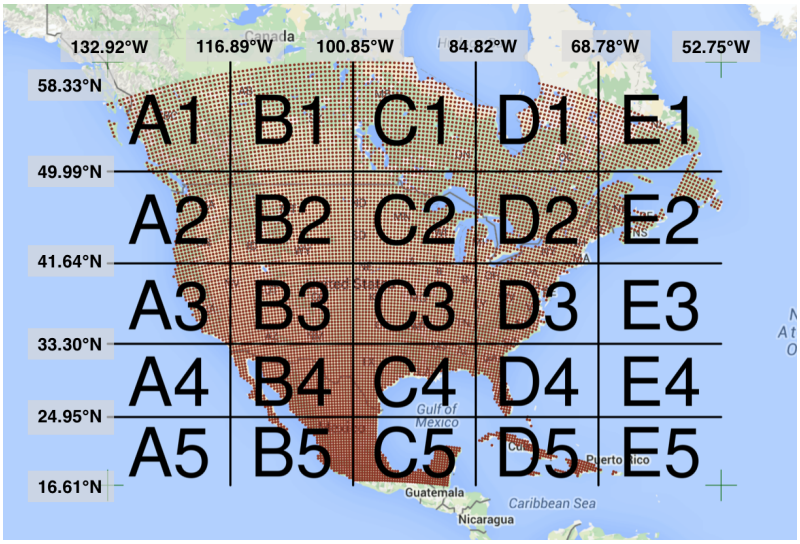


Figure 4.1: Levy et al. 2016. doi:10.5063/F1Z899CZ

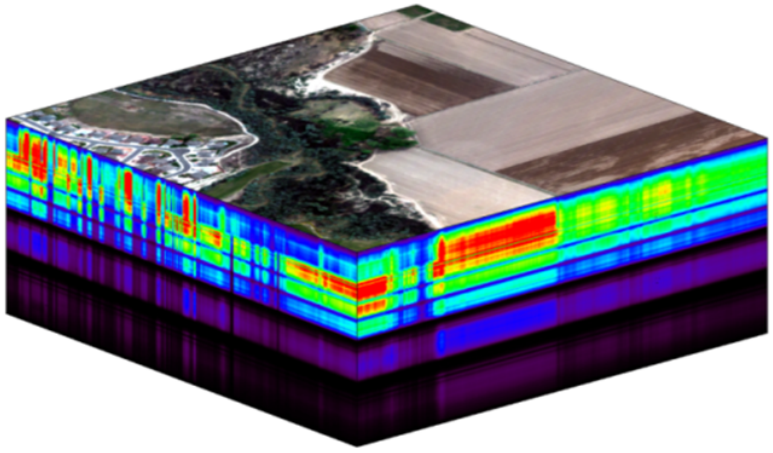


Figure 4.2: NEON Data Cube

4.2 Why parallelism?

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

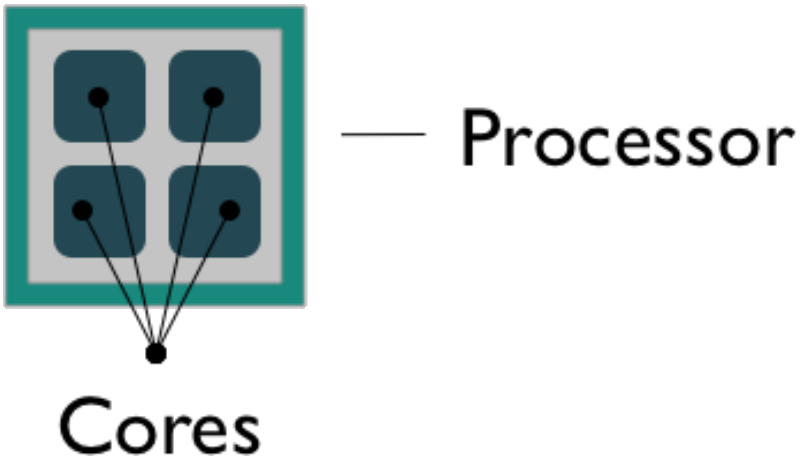
- **cpu-bound**: Take too much cpu time
- **memory-bound**: Take too much memory
- **I/O-bound**: Take too much time to read/write from disk
- **network-bound**: Take too much time to transfer

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.

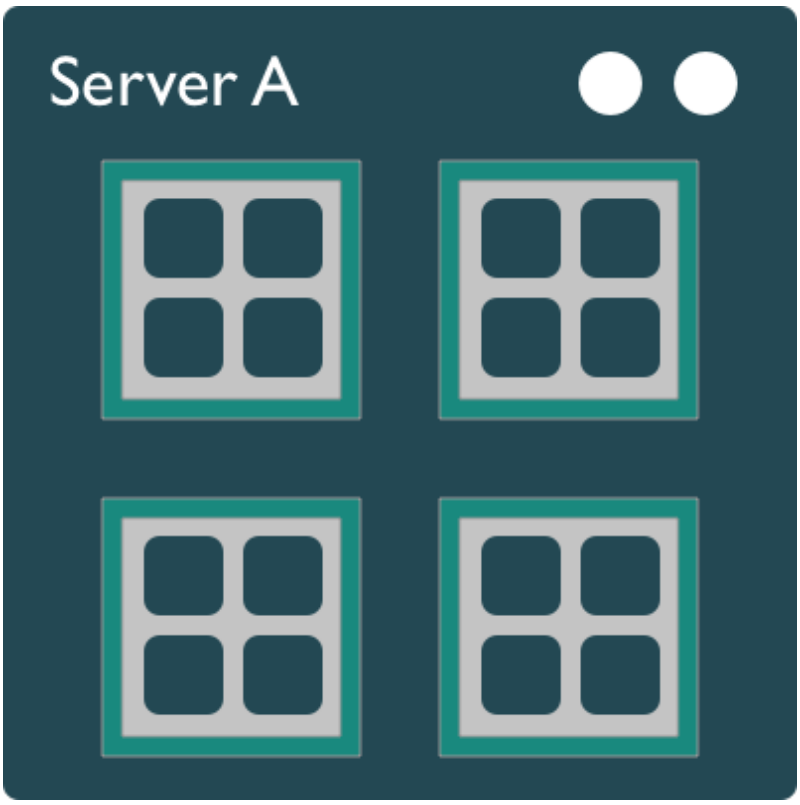
4.3 Processors (CPUs) and Cores

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:

```
jones@powder:~$ sysctl hw.ncpu hw.physicalcpu
hw.ncpu: 12
hw.physicalcpu: 6
```

To interpret that output, this machine `powder` has 6 physical CPUs, each of which has two processing cores, for a total of 12 cores for computation. I'd sure like my 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 `included-crab` at NCEAS:

```
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:              1
Core(s) per socket:              1
NUMA node0 CPU(s):               0-87
```

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
  - 4200 KNL nodes: 285,600 cores
  - 1736 SKX nodes: 83,328 cores
  - 224 ICX nodes: 17,920 cores

  - TOTAL: 386,848 cores

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 Simple task parallelization

Start with a task that is a little expensive:

```
def task(x):
    import numpy as np
    result = np.arange(x*10**8).sum()
    return result
```

```
import numpy as np

@timethis
def run_tasks_s(task_list):
    return [task(x) for x in task_list]

run_tasks_s(np.arange(10))
```

```
run_tasks_s: 126481.0152053833 ms
```

```
[0,
 4999999950000000,
 19999999900000000,
 44999999850000000,
 79999999800000000,
 124999999750000000,
 179999999700000000,
 244999999650000000,
 319999999600000000,
 404999999550000000]
```

```
from concurrent.futures import ThreadPoolExecutor

@timethis
def run_tasks_p(task_list):
    with ThreadPoolExecutor(max_workers=20) as cf_executor:
        return cf_executor.map(task, task_list)

results = run_tasks_p(np.arange(10))
[x for x in results]
```

4.6 Setup – downloading data for staging

In this exercise, we're going to parallelize a simple task that is often very time consuming – downloading data. And we'll compare performance of simple downloads using first a serial loop, and then using two parallel execution libraries: `concurrent.futures` and `parsl`. More on those later. But note that parallel execution won't always speed up this task, as this is likely an I/O bound task if you're downloading a lot of data. But we still should be able to speed things up a lot until we hit the limits of our disk arrays.

The data we are downloading is a pan-Arctic time series of TIF images containing rasterized Arctic surface water indices from:

> Elizabeth Webb. 2022. Pan-Arctic surface water (yearly and trend over time) 2000-2022. Arctic Data Center doi:10.18739/A2NK3665N.

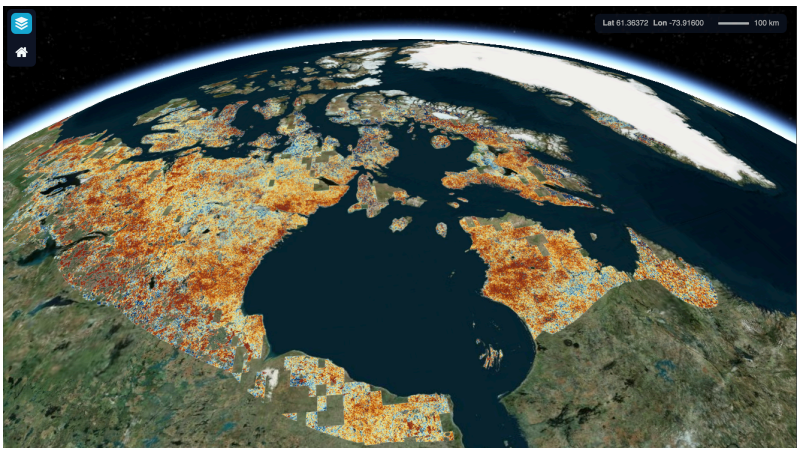


Figure 4.3: Webb water data

First, let's download the data serially to set a benchmark. The data files are listed in a table with their filename and identifier, and can be downloaded directly from the Arctic Data Center using the identifier:

```
/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
```

	filename	identifier
0	SWI_2007.tif	urn:uuid:5ee72c9c-789d-4a1c-95d8-cb2b24a20662
1	SWI_2019.tif	urn:uuid:9cd1cdc3-0792-4e61-afff-c11f86d3a9be
2	SWI_2021.tif	urn:uuid:14e1e509-77c0-4646-9cc3-d05f8d84977c
3	SWI_2020.tif	urn:uuid:1ba473ff-8f03-470b-90d1-7be667995ea1
4	SWI_2001.tif	urn:uuid:85150557-05fd-4f52-8bbd-ec5a2c27e23d

4.7 Task parallelism: serial downloads

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.

```
import urllib

def download_file(row):
    service = "https://arcticdata.io/metacat/d1/mn/v2/object/"
    pid = row[1]['identifier']
    filename = row[1]['filename']
    url = service + pid
    print("Downloading: " + filename)
    msg = urllib.request.urlretrieve(url, filename)
    return filename

@timethis
def download_list(dl_list):
    return [download_file(row) for row in dl_list.iterrows()]

results = download_list(id_list[0:5])
print(results)
```

```
Downloading: SWI_2007.tif

Downloading: SWI_2019.tif

Downloading: SWI_2021.tif

Downloading: SWI_2020.tif

Downloading: SWI_2001.tif

download_list: 58174.9153137207 ms
['SWI_2007.tif', 'SWI_2019.tif', 'SWI_2021.tif', 'SWI_2020.tif', 'SWI_2001.tif']
```

In this code, I have one function (`download_file`) that downloads a single data file and saves it to disk. It is called iteratively from the function `download_list`. The serial execution takes about 15.2 seconds, but can vary by a few seconds.

The issue with this loop is that we execute each trial sequentially, which means that only one of our processors on this machine is in use. In order to exploit parallelism, we need to be able to dispatch our tasks and allow each to run at the same time, with one task going to each core. To do that, we can use one of the many parallelization libraries in python to help us out.

4.8 Task parallelism with `concurrent.futures`

In this case, we'll use the same `download_file` function from previously, but will switch up the `download_list` to use `concurrent.futures`.

```
from concurrent.futures import ThreadPoolExecutor

@timethis
def download_list(dl_list):
```

```
    with ThreadPoolExecutor(max_workers=15) as cf_executor:
        return cf_executor.map(download_file, dl_list.iterrows(), timeout=60)

results = download_list(id_list[0:5])
for result in results:
    print(result)
```

```
Downloading: SWI_2007.tif
Downloading: SWI_2019.tif
Downloading: SWI_2021.tif
Downloading: SWI_2020.tif
Downloading: SWI_2001.tif

download_list: 37146.09479904175 ms
SWI_2007.tif
SWI_2019.tif
SWI_2021.tif
SWI_2020.tif
SWI_2001.tif
```

```
from concurrent.futures import ProcessPoolExecutor

@timethis
def download_list(dl_list):
    with ProcessPoolExecutor(max_workers=15) as cf_executor:
        return cf_executor.map(download_file, dl_list.iterrows(), timeout=60)

results = download_list(id_list[0:5])
for result in results:
    print(result)
```

```
Downloading: SWI_2019.tif

Downloading: SWI_2020.tif

Downloading: SWI_2021.tif
```

```
Downloading: SWI_2001.tif

Downloading: SWI_2007.tif
```

```
download_list: 14630.003452301025 ms
SWI_2007.tif
SWI_2019.tif
SWI_2021.tif
SWI_2020.tif
SWI_2001.tif
```

4.9 parsl

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

```
# Required packages
import parsl
from parsl import python_app
from parsl.config import Config
from parsl.executors import HighThroughputExecutor
from parsl.providers import LocalProvider

# Configure the parsl executor
activate_env = 'workon scomp'
htex_local = Config(
    executors=[
        HighThroughputExecutor(
```

```
            max_workers=15,
            provider=LocalProvider(
                worker_init=activate_env
            )
        )
    ],
)
parsl.clear()
parsl.load(htex_local)

# Define a parsl_app for our download function using a decorator
@python_app
def download_parsl(row):
    import urllib
    service = "https://arcticdata.io/metacat/d1/mn/v2/object/"
    pid = row[1]['identifier']
    filename = row[1]['filename']
    url = service + pid
    print("Downloading: " + filename)
    msg = urllib.request.urlretrieve(url, filename)
    return filename

@timethis
def download_list(dl_list):
    results = []
    for row in dl_list.iterrows():
        result = download_parsl(row)
        results.append(result)
    return(results)

@timethis
def wait_for_futures():
    results = download_list(id_list[0:5])
    done = [app_future.result() for app_future in results]
    print(done)

wait_for_futures()
htex_local.executors[0].shutdown()
parsl.clear()
```

```
download_list: 182.96051025390625 ms

['SWI_2007.tif', 'SWI_2019.tif', 'SWI_2021.tif', 'SWI_2020.tif', 'SWI_2001.tif']
wait_for_futures: 43672.90258407593 ms
```

4.10 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

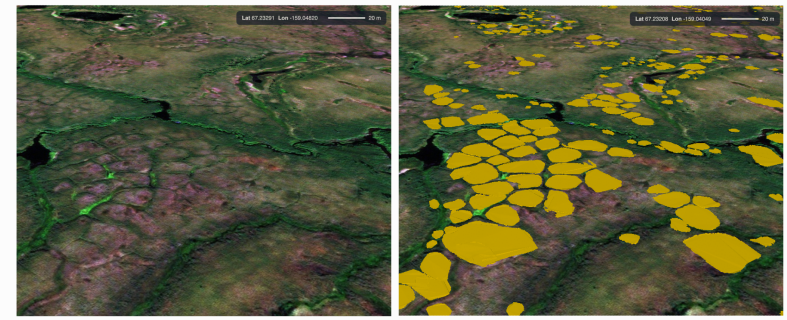
7 Group Project: Staging and Preprocessing

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

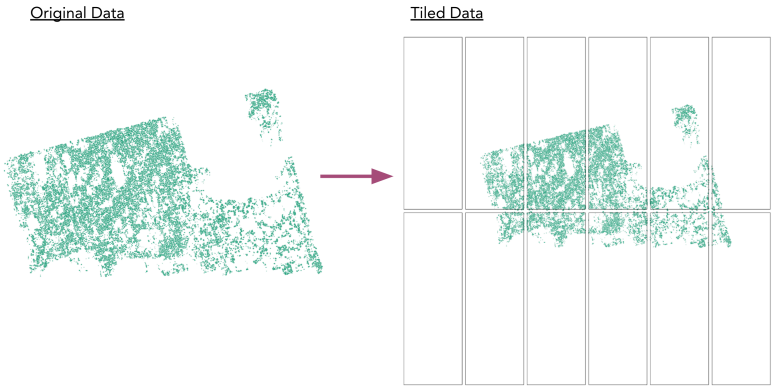
7.1 Introduction

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.

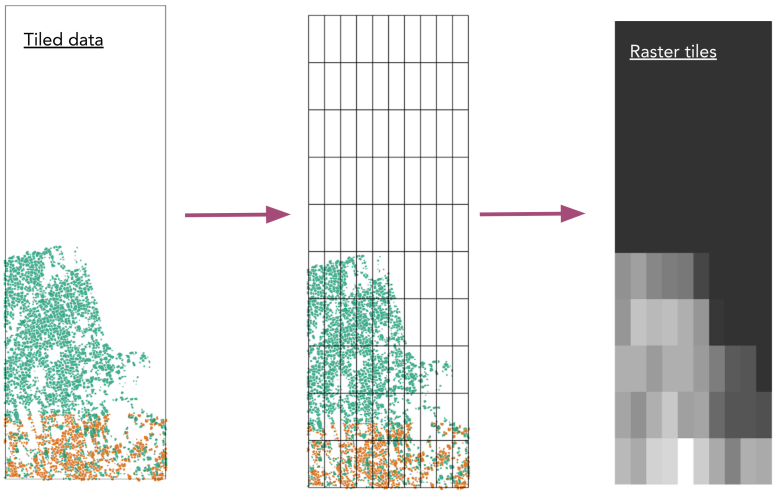
Ice wedges form as a result of thermal contraction during melt/freeze cycles of permafrost. They can form very distinctive geometries clearly visible in satellite images. The PDG is using advanced analysis and computational tools to take high resolution satellite imagery and automatically dectect where 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.



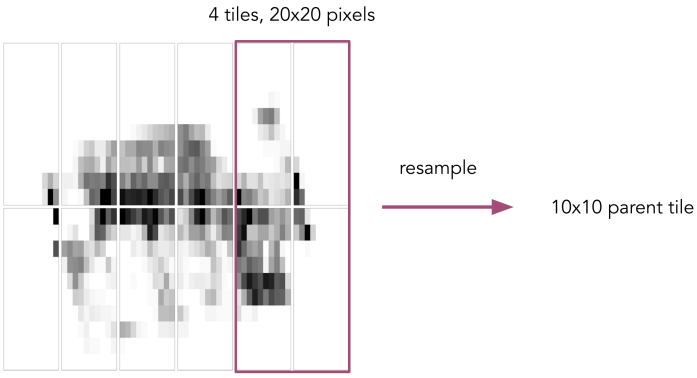
In the group project, we are going to use a subset of the high resolution dataset of these detected ice wedge polygons in order to learn some of the reproducible, scalable techniques that will allow us to process it. Our workflow will start with a set of large geopackage files that contain the detected 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

- Learn about the NetCDF data format:
  - Characteristics: self-describing, scalable, portable, appendable, shareable, and archivable
  - Understand the NetCDF data model: what are dimensions, variables, and attributes
  - Advantages and differences between NetCDF and tabular data formats
- 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 files and understand the attributes view
  - Perform basic indexing, processing, and reduction of `xarray.DataArrays`
  - Convert `pandas.DataFrames` into `xarray.DataSets`

9.1 Introduction

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.

9.2 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 initially developed at the Unidata Program Center and is supported on almost all platforms, and parsers exist for most scientific programming languages.

The NetCDF documentation outlines that this data format is desgined to be:

1. **Self-describing:** Information describing the data contents of the file is embedded within the data file itself. This means that there is a header describing the layout of the rest of the file and arbitrary file metadata.

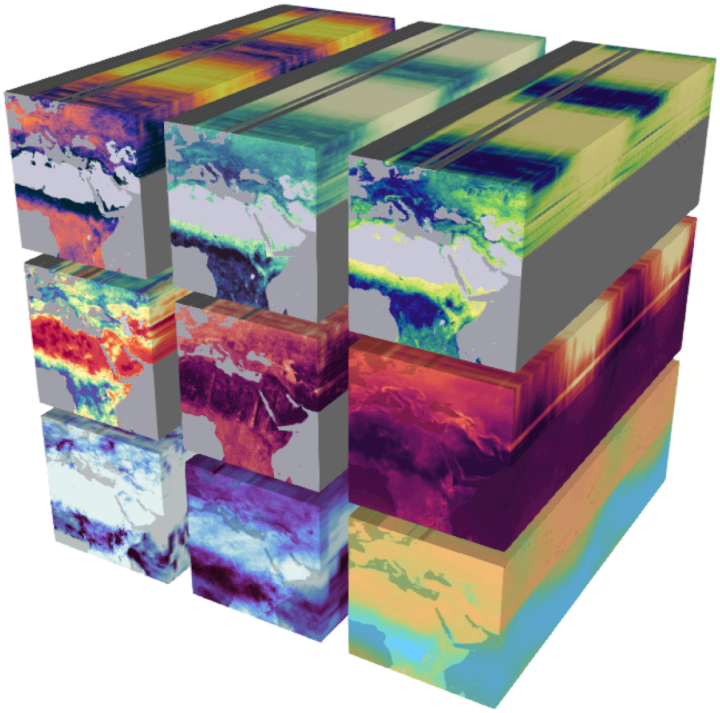


Figure 9.1: Mahecha et al. 2020 . *Visualization of the implemented Earth system data cube. The figure shows from the top left to bottom right the variables sensible heat (H), latent heat (LE), gross primary production (GPP), surface moisture (SM), land surface temperature (LST), air temperature (Tair), 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.*

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.

9.2.1 Data Model

The NetCDF data model is the way that NetCDF organizes data. This lesson will follow the Classic NetCDF Data Model, which is at the core of all netCDF files.

The model consists of three key components: **variables**, **dimensions**, and **attributes**.

- **Variables** are N-dimensional arrays of data. We can think of these as varying/measured/dependent quantities.
- **Dimensions** describe the axes of the data arrays. A dimension has a name and a length. 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 a variable or the file as a whole.

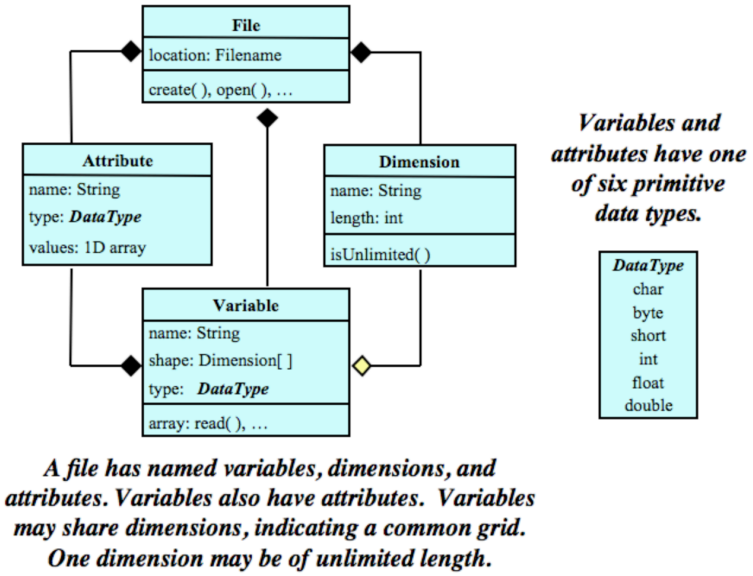


Figure 9.2: Classic NetCDF Data Model (NetCDF documentation)

9.2.2 Metadata Standards

The most commonly used metadata standard for geospatial data is the **Climate and Forecast metadata standard**, also called the **CF conventions**.

> The CF conventions 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. (CF conventions FAQ)

The CF conventions provide a unique standardized name and precise description of over 1,000 physical variables. To maximize the reusability of our data, it is best to include a variable's standardized name as an attribute called `standard_name`. Variables should also include a `units` attribute. This attribute should be a string that can be recognized by UNIDATA's UDUNITS package. In these links you can find:

- a table with all of the CF convention's standard names, and
- a list of the units found in the UDUNITS database maintained by the North Carolina Institute for Climate Studies.

9.2.3 Exercise

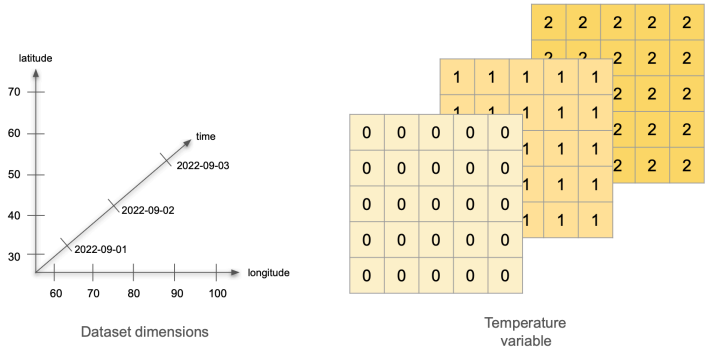
Let's do a short practice now that we have reviewed the classic NetCDF model and know a bit about metadata best practices.

Part 1

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. On the first day, all stations record a temperature of 0°C. On the second day, all temperatures are 1°C, and on the third day, all temperatures are 2°C. What are the *variables*, *dimensions* and *attributes* for this data?

Answer

Variables: There is a single variable being measured: temperature. The variable values can be represented as a 5x5x3 array, with constant values for each day.
Dimensions: This dataset has three dimensions: time, latitude, and longitude. Time indicates when the measurement happened, we can encode it as the dates 2022-09-01, 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. Longitude has values 60, 70, 80, 90, and 100, measured in degrees East.



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

- Variable attributes:
  - Temperature attributes:
    - standard_name: air_temperature
    - units: degree_C
- Dimension attributes:
  - Time attributes:
    - description: date of measurement
  - Latitude attributes:
    - standard_name: grid_latitude
    - units: degrees_N
  - Longitude attributes:
    - satandard_name: grid_longitude
    - units: degree_E
- Dataset attributes:
  - title: Temperature Measurements at Weather Stations
  - summary: an example of NetCDF data format

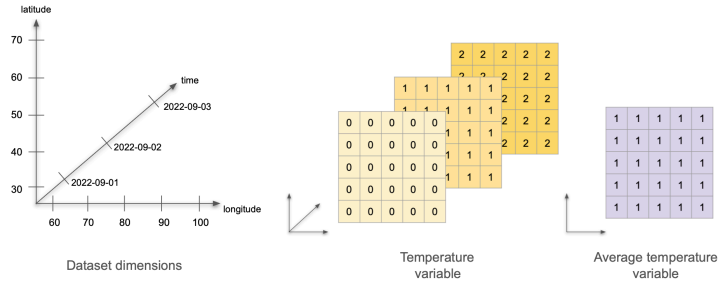
Part 2

Now imagine we calculate the average temperature over time at each weather station, and we wish to incorporate this data into the same dataset. How will adding the average tempera-

ture data change the dataset's variables, attributes, and dimensions?

Answer

Variables: Now we are measuring two variables: temperature and average temperature. The temperature data stays the same. We can represent the average temperature as a single 5x5 array with value 1 at each cell.
Dimensions: This dataset still has three dimensions: time, latitude, and longitude. The temperature variable uses all three dimensions, and the average temperature variable only uses two (latitude and longitude). This is ok! The dataset's dimensions are the union of the dimensions of all the variables in the dataset. Variables in the same dataset may have all, some, or no dimensions in common.



Attributes: To begin with, we need to keep all the previous attributes. Notice that the dataset's title is general enough that we don't need to update it. The only update we need to do is add the attributes for our new average temperature variable:

- Average temperature attributes:
  - standard_name: average_air_temperature
  - description: average temperature over three days

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

9.3 `xarray`

`xarray` is an open source project and Python package that augments NumPy arrays by adding labeled dimensions, coordinates and attributes. `xarray` is based on the netCDF data model, making it the appropriate tool to open, process, and create datasets in netCDF format.

Figure 9.3: `xarray`'s development portal

9.3.1 `xarray.DataArray`

The `xarray.DataArray` is the primary data structure of the `xarray` package. It is an n-dimensional array with **labeled dimensions**. We can think of it as representing a single variable in the NetCDF data format: it holds the variable's values, dimensions, and attributes.

Apart from variables, dimensions, and attributes, `xarray` introduces one more piece of information to keep track of a dataset's content: in `xarray` each dimension has at least one set of **coordinates**. A dimension's coordinates indicate the dimension's values. We can think of the coordinate's values as the tick labels along a dimension. For example, in our previous exercise about temperature measured in weather stations, latitude is a dimension, and the latitude's coordinates are 30, 40, 50, 60, and 70 because those are the latitude values at which we are collecting temperature data. In that same exercise, time is a dimension, and its coordinates are 2022-09-1, 2022-09-02, and 2022-09-03.

Here you can read more about the `xarray` terminology.

9.3.1.1 Create an `xarray.DataArray`

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 the 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 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 the 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',
          'standard_name' : 'air_temperature',
          'units' : 'degree_c'}

# 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
    standard_name:  air_temperature
    units:          degree_c
```

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 = {'description':'date of measurement'}
```

```
temp.lat.attrs['standard_name']= 'grid_latitude'
temp.lat.attrs['units'] = 'degree_N'

temp.lon.attrs['standard_name']= 'grid_longitude'
temp.lon.attrs['units'] = 'degree_E'
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
    standard_name:  air_temperature
    units:          degree_c
    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.

9.3.1.2 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 coordinates, 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
    standard_name:  air_temperature
    units:          degree_c
    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
    standard_name:  air_temperature
    units:          degree_c
    description:    simple example of an xarray.DataArray
```

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

More about `xarray` indexing.

9.3.1.3 Reduction

`xarray` has implemented several methods to reduce an `xarray.DataArray` along any number of dimensions. One of the advantages of `xarray.DataArray` is that, if we choose 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.

9.3.2 xarray.DataSet

An xarray.DataSet resembles an in-memory representation of a NetCDF file and consists of *multiple* variables (each being an xarray.DataArray), with dimensions, coordinates, and attributes, forming a self-describing dataset. Attributes can be specific to each variable, each dimension, or they can describe the whole dataset. The variables in an xarray.DataSet can have the same dimensions, share some dimensions, or have no dimensions in common. Let's see an example of this.

9.3.2.1 Create an xarray.DataSet

Following our previous example, we can create an xarray.DataSet by combining the temperature data with the average temperature data. We also add some attributes that now describe the whole dataset, not only each variable.

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

attrs = {'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)
```

Take some time to click through the data viewer and read through the variables and metadata in the dataset. Notice the following:

- `temp_dataset` is a dataset with three dimensions (time, latitude, and longitude),
- `temp` is a variable that uses all three dimensions in the dataset, and
- `aveg_temp` is a variable that only uses two dimensions (latitude and longitude).

```
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:
    title:        temperature data at weather stations: daily and and average
    description:  simple example of an xarray.Dataset
```

9.3.2.2 Save and Reopen

Finally, we want to save our dataset as a NetCDF file. To do this, specify the file path and use the *.nc* extension for the file name. Then save the dataset 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:
    title:        temperature data at weather stations: daily and and average
    description:  simple example of an xarray.Dataset
```

9.3.3 Exercise

For this exercise, we will use a dataset including time series 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
```

```
<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)...
```

1. How many dimensions does the `runoff_annual` variable have? What are the coordinates for the second dimension of this variable?

Answer

We can see in the object viewer that the `runoff_annual` variable has two dimensions: `time` and `member`, in that order. We can also access the dimensions by calling:

```
fw_data.runoff_annual.dims
```

The second dimensions is `member`. Near the top of the object viewer, under coordinates, we can see that that member's coordinates is an array from 1 to 11. We can directly see this array by calling:

```
fw_data.member
```

2. Select the values for the second member of the `netPrec_annual` variable.

Answer

```
member2 = fw_data.netPrec_annual.sel(member=2)
```

3. What is the maximum value of the second member of the `netPrec_annual` variable in the time period 2022 to 2100? Hint.

Answer

Based on our previous answer, this maximum is:

```
x_max = member2.loc[2022:2100].max()
xmax.item(0)
```

Notice we had to use `item` to transform the array into a number.

9.4 NetCDF and Tabular Data

Undoubtedly, tabular data is one of the most popular data formats. In this last section, we will discuss the relation between tabular data and the NetCDF data format and how to transform a `pandas.DataFrame` into an `xarray.DataSet`.

9.4.1 Tabular to NetCDF

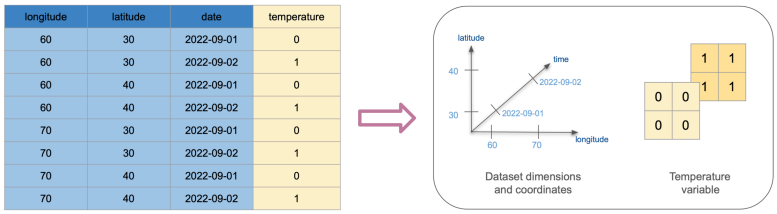
We assume our starting point is tabular data that meets the criteria for **tidy data**, which means:

- Each column holds a different variable.
- Each row holds a different observation.

Take, for example, this tidy data subset from our exercise about weather stations measuring temperature:

longitude	latitude	date	temperature
60	30	2022-09-01	0
60	30	2022-09-02	1
60	40	2022-09-01	0
60	40	2022-09-02	1
70	30	2022-09-01	0
70	30	2022-09-02	1
70	40	2022-09-01	0
70	40	2022-09-02	1

To understand how this will transform into NetCDF format, we first need to identify which columns will act as dimensions and which as variables. We can also think of the values of the dimension columns as the coordinates in the `xarray.DataSet`. The diagram below shows how these columns transform into variables, dimensions, and coordinates.



Tabular data format

NetCDF data format

Tabular formats like csv do not offer a standard way to encode attributes for the dimensions or variables, this is why we don't see any attributes in the resulting NetCDF data. One of the most significant advantages of NetCDF is its self-describing properties.

9.4.2 `pandas` to `xarray`

What does the previous example look like when working with `pandas` and `xarray`?

Let's work with a csv file containing the previous temperature measurements. Essentially, we need to read this file as a `pandas.DataFrame` and then use the `pandas.DataFrame.to_xarray()` method, taking into account that the dimensions of the resulting `xarray.DataSet` will be formed using the index column(s) of the `pandas.DataFrame`. In this case, we know the first three columns will be our dimension columns, so we need to group them as a multindex for the `pandas.DataFrame`. We can do this by using the `index_col` argument directly when we read in the csv file.

```
fp = os.path.join(os.getcwd(),'netcdf_temp_data.csv')
```

```
# specify columns representing dimensions
dimension_columns = [0,1,2]

# read file
temp = pd.read_csv(fp, index_col=dimension_columns)
temp
```

```
/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
```

			temperature
longitude	latitude	date	
60	30	2022-09-01	0
		2022-09-02	1
	40	2022-09-01	0
		2022-09-02	1
70	30	2022-09-01	0
		2022-09-02	1
	40	2022-09-01	0
		2022-09-02	1

And this is our resulting `xarray.DataSet`:

```
temp.to_xarray()
```

```
<xarray.Dataset>
Dimensions:      (longitude: 2, latitude: 2, date: 2)
Coordinates:
  * longitude    (longitude) int64 60 70
  * latitude     (latitude) int64 30 40
  * date         (date) object '2022-09-01' '2022-09-02'
Data variables:
    temperature  (longitude, latitude, date) int64 0 1 0 1 0 1 0 1
```

For further reading and examples about switching between `pandas` and `xarray` you can visit the following:

- `xarray`'s Frequently Asked Questions
- `xarray`'s documentation about working with pandas
- `pandas.DataFrame.to_xarray` documentation

10 Parallelization with Dask

- Become familiar with the 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 to load data and specify partition/chunk sizes of `dask.arrays`/`dask.dataframes`.
- Integrate `xarray` and `rioxarray` with Dask for geospatial computations
- Share best practices and resources for further reading

10.1 Introduction

Dask is a library for parallel computing in Python. It can scale up code to use your personal computer's full capacity or distribute work in a cloud cluster. By mirroring APIs of other commonly used Python libraries, such as Pandas and NumPy, 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.

10.2 Dask Cluster

We can deploy a **Dask cluster** on a single machine or an actual cluster with multiple machines. The cluster has 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. You can read more about threads and processes here.

To interact with the client and generate tasks that can be processed in parallel we need to use Dasks' objects to read our data. In this lesson, we will see examples of how to use `dask.dataframes` and `dask.arrays`. Apart from data frames and arrays, other Dask objects can be used to parallelize your workflow.

10.2.1 Setting up a Local Cluster

We can create a **local cluster** as follows:

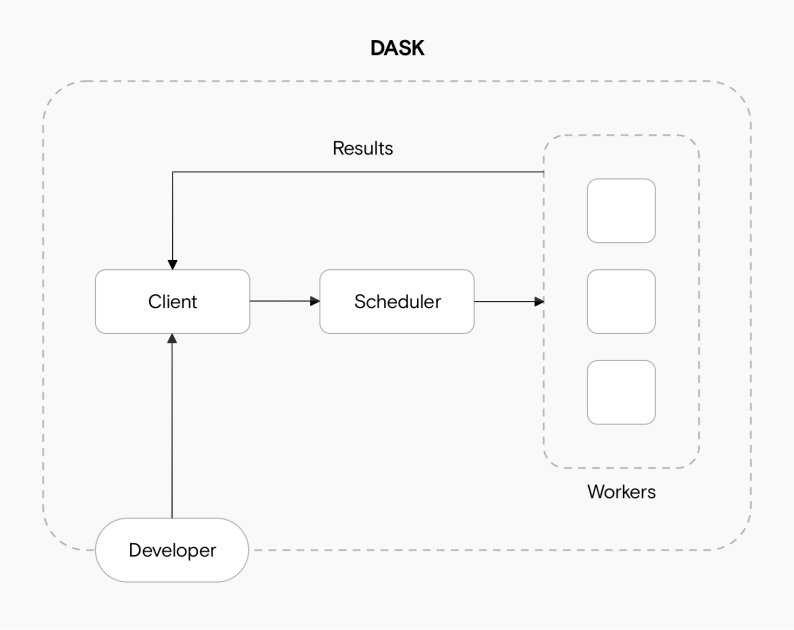
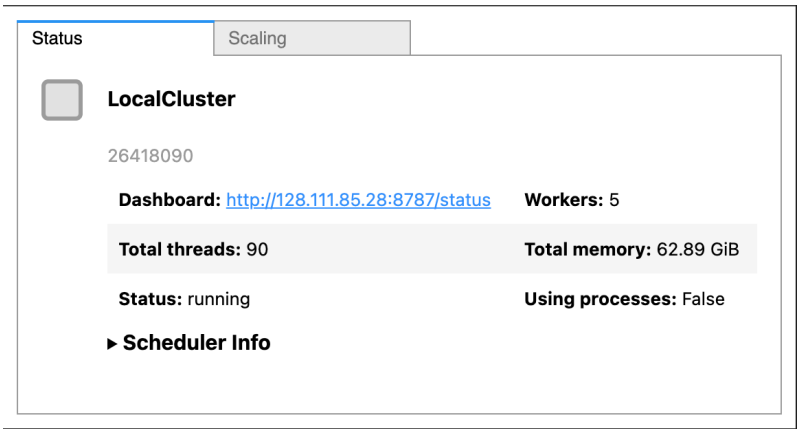


Figure 10.1: From: https://www.datarevenue.com/en-blog/understanding-dask-architecture-client-scheduler-workers Make this into a collored graph, add cluster envelope

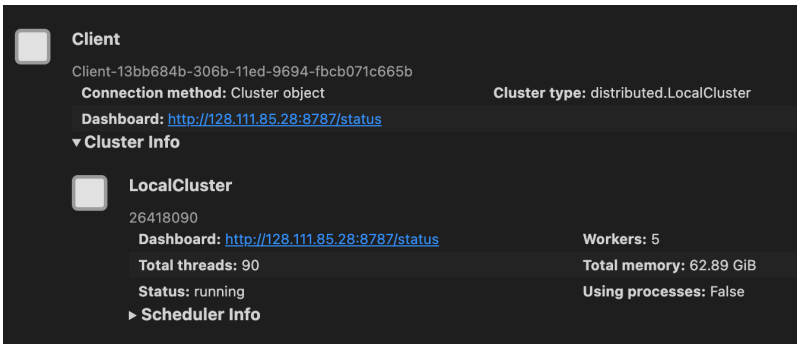
```
from dask.distributed import LocalCluster, Client
```

```
cluster = LocalCluster(n_workers=5, memory_limit=0.1, processes=False)
cluster
```



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

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



This is a good place to learn more about different Dask clusters.

10.2.2 Dask Dashboard

We chose to use the Dask cluster in this lesson instead of the default Dask scheduler to take advantage of the **cluster dash-**

board, which offers live monitoring of the performance and progress of our computations. When we set up a cluster, we can see the dashboard's address by looking at either the client or the cluster. The dashboard's main page shows diagnostics about:

- the cluster's and individual worker's memory usage,
- number of tasks being processed by each worker,
- individual tasks being processed across workers, and
- progress towards completion of individual tasks.

There's much to talk about interpreting the Dask dashboard's diagnostics. We recommend this documentation to understand the basics of the dashboard diagnostics and this video as a deeper dive into the dashboard's functions.

Figure 10.2: A Dask dashboard.

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` comprises 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`.

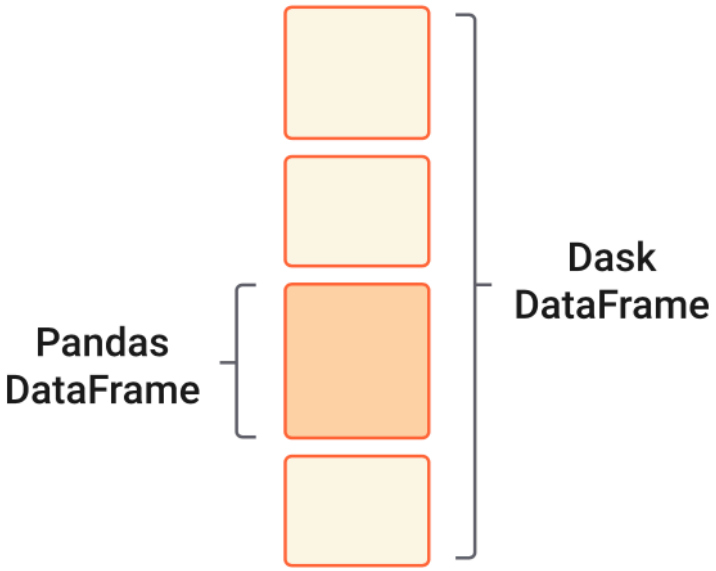


Figure 10.3: Dask Array design (dask documentation)

10.3.1 Reading a csv

To get familiar with `dask.dataframes`, we will use tabular data of soil moisture 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 in the previous lesson, we will download the data using the `requests` package and the data's URL obtained from the Arctic Data Center.

```
import os
import requests
```

```
import dask.dataframe as dd
```

```
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)
```

In the Arctic Data Center metadata we can see this file is 115 MB. 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, meaning a block size of approximately 20 MB.

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

> **Note**
>
> *About the encoding parameter:* If we try to import the file directly, we will receive an `UnicodeDecodeError`. We can run the following code to find the file's encoding and add the appropriate encoding to `dask.dataframe.read_csv`.
>
> ```
> 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
> ```

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 explicitly ask it to print or compute something to do so. However, we can still do `df.head()`. It's not costly for memory to access a few data frame rows.

```
df.head(2)
```

10.3.2 Lazy Computations

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

```
averages = df.groupby('year').mean()
averages
```

Notice that we cannot see any values in the resulting data frame. A major difference between `pandas.DataFrames` and `dask.dataframes` is that `dask.dataframes` are "lazy". This means an object will queue transformations and calculations without executing them 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 evaluation** (or **delayed computation**) is how most Dask workloads work. This varies from **eager evaluation** methods and functions, which start computing results right when they are executed.

Before calling `compute` on an object, open the Dask dashboard to see how the parallel computation is happening.

```
averages.compute()
```

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.

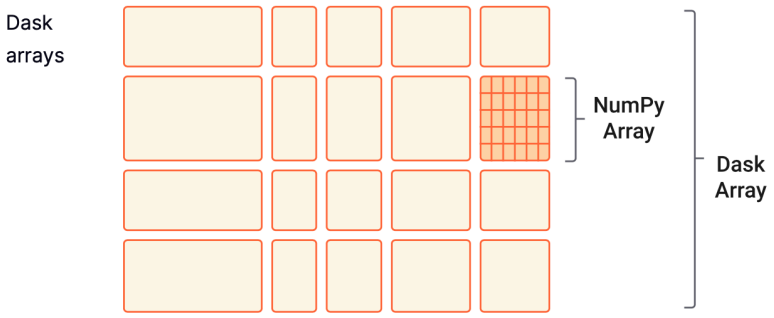


Figure 10.4: Dask Array design (dask documentation)

In this short example we will create a 200x500 `dask.array` by specifying chunk sizes of 100x100.

```
import numpy as np

import dask.array as da
```

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

Computations for `dask.arrays` also work lazily. We need to call `compute` to trigger computations and bring the result to memory.

```
a.mean()
```

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

10.5 Dask and `xarray`

In the future, it might be more common having to read some big array-like dataset (like a high-resolution multiband raster) than creating one from scratch using NumPy. In this case, it can be useful to use the `xarray` module together with Dask. It is simple to wrap Dask around `xarray` objects. We only need

to specify the number of chunks 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. doi:10.18739/A2ZC7RV6H.).

First, we download the data for the near-infrared (NIR) and red bands from the Arctic Data Center:

```
# 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 `rioxarray` to read them. `rioxarray` extends `xarray` with the *rio* accessor, which stands for "raster input and output". You can find more information about `rioxarray` here.

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

```
import rioxarray as rioxr
```

```
#raster = rioxr.open_rasterio(href, chunks={"x": int(x_n/workers_sqrt), "y":int(y_n/workers_

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

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 how 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:

```
# open data
fp_nir = os.path.join(os.getcwd(),"RU_ANS_TR2_FL005M_nir.tif")
nir = rioxr.open_rasterio(fp_nir, chunks = '15MB')

# print current shape
print('shape before squeeze:', nir.shape)
print('chunks before squeeze:', nir.chunks )

#squeeze
nir = nir.squeeze()

# print squeezed shape
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))
```

10.5.2 Calculating NDVI

Now 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 bring the results to memory.

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

And finally, we can see what 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

Dask is an exciting tool for parallel computing, but it may take a while to understand its nuances to make the most of it. There are many best practices and recommendations. These are some of the basic ones to take into consideration:

- For data that fits into RAM, pandas, and NumPy can often be faster and easier to use than Dask workflows. The simplest solution can often be the best.

- While Dask may have similar APIs to pandas and NumPy, there are differences, and not all the methods for the `pandas.DataFrames` and `numpy.arrays` translate in the same way (or with the same efficiency) to Dask objects. When in doubt, always read the documentation.
- Choose appropriate chunk and partition sizes and layouts. This is crucial to best use how the scheduler distributes work. You can read here about best practices for chunking.
- Avoid calling compute repeatedly. It is best to group similar computations together and then compute once.

Further reading:

- A friendly article about common dask mistakes
- General Dask best practices
- `dask.dataframe` best practices
- `dask.array` best practices

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.

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."

```
import requests

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

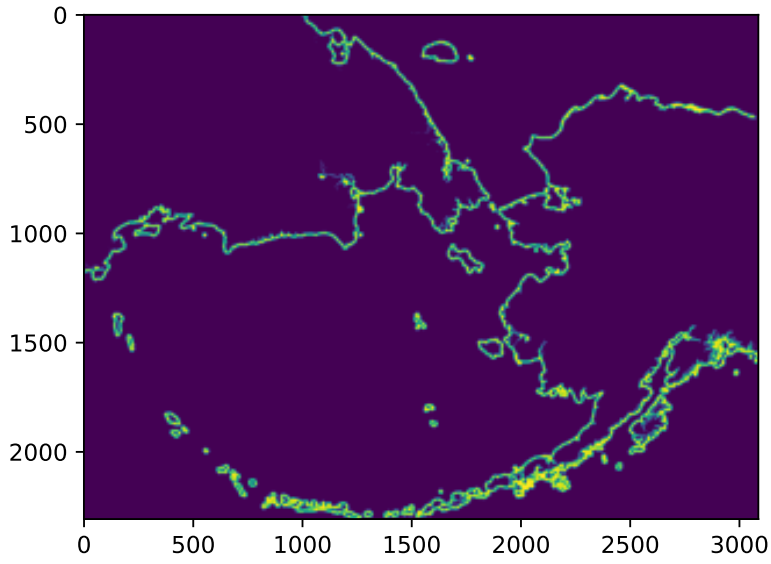
```
response = requests.get(url)
open("Coastal_2020_08.tif", "wb").write(response.content)
```

```
1473505
```

Using `rasterio`, 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.

```
import rasterio
import matplotlib.pyplot as plt

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
```

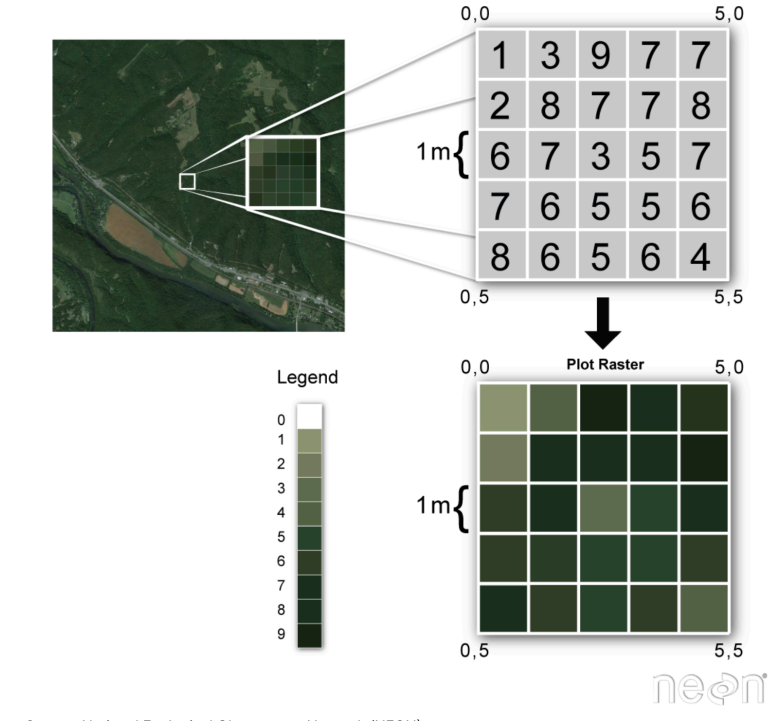


You'll notice that we are saving two objects here, `ships` and `ships_meta`. Looking at the types of these two objects is useful to understand what `rasterio` is doing.

```
type(ships)
type(ships_meta)
```

```
rasterio.profiles.Profile
```

The `ships` object is a `numpy` array, while the `ships_meta` is a special `rasterio` class called `Profile`. To understand why the raster data is represented as an array, and what that profile object is, let's look into what raster data are, exactly.



Source: Leah A. Wasser, Megan A. Jones, Zack Brym, Kristina Riemer, Jason Williams, Jeff Hollister, Mike Smorul. Raster 00: Intro to Raster Data in R. National Evologial Observatory Network (NEON).

The upper left panel of the figure above shows some satellite imagery data. These data are in raster format, which when you zoom in, really consist of regularly gridded pixels, each of which contails a value. When we plot these data, we can assign a color map to the pixel values, which generates the image we see. The data themselves, though, are just a two dimensional grid of numbers. Another way we might describe this is…an array! So, this is why raster data is represented in python using a `numpy` array.

This is all great, and the array of values is a lot of information, but there are some key items that are missing. This array isn't imaginary, it represents a physical space on this earth, so where is all of that contextual information? The answer is in the `rasterio` profile object. This object contains all of the

metadata needed to interpret the raster array. Here is what our `ships_meta` contains:

```
'driver': 'GTiff',
'dtype': 'float32',
'nodata': -3.3999999521443642e+38,
'width': 3087,
'height': 2308,
'count': 1,
'crs': CRS.from_epsg(3338),
'transform': Affine(999.7994153462766, 0.0, -2550153.29233849, 0.0, -999.9687691991521, 271170
'tiled': False,
'compress': 'lzw',
'interleave': 'band'}
```

This object gives us critical information, like the CRS of the data, the no data value, and the transform. The transform is what allows us to move from image pixel (row, col) coordinates to and from geographic/projected (x, y) coordinates. The transform and the CRS are crititally important, and related. If the CRS are instructions for how the coordinates can be represented in space and on a flat surface (in the case of projected coordinate systems), then the transform describes how to locate the raster array positions in the correct coordinates given by the CRS.

Note that since the array and the profile are in separate objects it is easy to lose track of one of them, accidentally overwrite it, etc. Try to adopt a naming convention that works for you because they usually need to work together in geospatial operations.

11.3 Pre-processing vector data

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 using `geopandas`.

```
import geopandas as gpd

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

Note the "pandas" in the library name "geopandas." Our `comm` object is really just a special type of pandas data frame called a geodataframe. This means that in addition to any geospatial stuff we need to do, we can also just do regular `pandas` things on this data frame.

For example, we can get a list of column names (there are a lot!)

```
comm.columns.values
```

```
array(['OBJECTID', 'GEOMETRY_START_DATE', 'GEOMETRY_END_DATE',
       'STAT_AREA', 'STAT_AREA_NAME', 'FISHERY_GROUP_CODE',
       'GIS_SERIES_NAME', 'GIS_SERIES_CODE', 'REGION_CODE',
       'REGISTRATION_AREA_NAME', 'REGISTRATION_AREA_CODE',
       'REGISTRATION_AREA_ID', 'REGISTRATION_LOCATION_ABBR',
       'MANAGEMENT_AREA_NAME', 'MANAGEMENT_AREA_CODE', 'DISTRICT_NAME',
       'DISTRICT_CODE', 'DISTRICT_ID', 'SUBDISTRICT_NAME',
       'SUBDISTRICT_CODE', 'SUBDISTRICT_ID', 'SECTION_NAME',
       'SECTION_CODE', 'SECTION_ID', 'SUBSECTION_NAME', 'SUBSECTION_CODE',
       'SUBSECTION_ID', 'COAR_AREA_CODE', 'CREATOR', 'CREATE_DATE',
       'EDITOR', 'EDIT_DATE', 'COMMENTS', 'STAT_AREA_VERSION_ID',
       'Shape_Length', 'Shape_Area', 'geometry'], dtype=object)
```

We can also look at the head of the data frame:

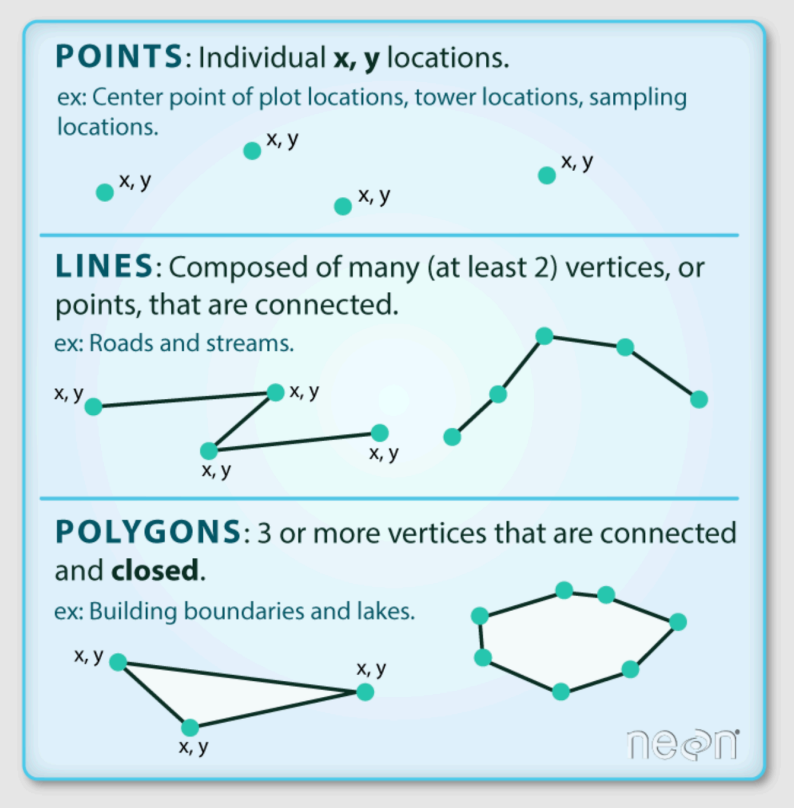
```
comm.head
```

```
<bound method NDFrame.head of      OBJECTID       GEOMETRY_START_DATE GEOMETRY_END_DATE  STAT_A
0          12 1975-01-01 00:00:00+00:00              NaT      33461
1          13 1975-01-01 00:00:00+00:00              NaT      33462
2          18 1978-01-01 00:00:00+00:00              NaT      33431
3          19 1980-01-01 00:00:00+00:00              NaT      33442
4          20 1980-01-01 00:00:00+00:00              NaT      33443
..        ...                       ...              ...        ...
860       959                       NaT              NaT      19241
861       960                       NaT              NaT      19242
862       961 1994-01-01 00:00:00+00:00              NaT      19245
863       962                       NaT              NaT      19250
864       963 1994-01-01 00:00:00+00:00              NaT      18252

                                STAT_AREA_NAME FISHERY_GROUP_CODE  \
0        Tanana River mouth to Kantishna River                   B
1                  Kantishna River to Wood River                 B
2                     Toklik to Cottonwood Point                 B
3    Right Bank, Bishop Rock to Illinois Creek                   B
4      Left Bank, Cone Point to Illinois Creek                   B
..                                         ...                ...
860                          Kaliakh River                       B
861                             Tsiu River                       B
862                         Midtimber River                      B
863                             Seal River                       B
864                          Middle Italio                       B

    GIS_SERIES_NAME GIS_SERIES_CODE  REGION_CODE    REGISTRATION_AREA_NAME  \
0            Salmon               B            3                Yukon Area
1            Salmon               B            3                Yukon Area
2            Salmon               B            3                Yukon Area
3            Salmon               B            3                Yukon Area
4            Salmon               B            3                Yukon Area
..              ...             ...          ...                       ...
860          Salmon               B            1  Southeastern Alaska Area
861          Salmon               B            1  Southeastern Alaska Area
862          Salmon               B            1  Southeastern Alaska Area
863          Salmon               B            1  Southeastern Alaska Area
864          Salmon               B            1  Southeastern Alaska Area

     ... COAR_AREA_CODE         CREATOR               CREATE_DATE  \
0    ...             YU   Evelyn Russel 2006-03-26 00:00:00+00:00
```

```
1    ...             YU   Evelyn Russel 2006-03-26 00:00:00+00:00
2    ...             YL   Evelyn Russel 2006-03-26 00:00:00+00:00
3    ...             YU   Evelyn Russel 2006-03-26 00:00:00+00:00
4    ...             YU   Evelyn Russel 2006-03-26 00:00:00+00:00
..   ...            ...            ...                       ...
860  ...             A2   Evelyn Russel 2006-03-26 00:00:00+00:00
861  ...             A2   Evelyn Russel 2006-03-26 00:00:00+00:00
862  ...             A2   Evelyn Russel 2006-03-26 00:00:00+00:00
863  ...             A2   Evelyn Russel 2006-03-26 00:00:00+00:00
864  ...             A2  Sabrina Larsen 2017-05-05 00:00:00+00:00

              EDITOR                 EDIT_DATE  \
0    Sabrina Larsen 2017-02-02 00:00:00+00:00
1    Sabrina Larsen 2017-02-02 00:00:00+00:00
2    Sabrina Larsen 2017-02-02 00:00:00+00:00
3    Sabrina Larsen 2017-02-02 00:00:00+00:00
4    Sabrina Larsen 2017-02-02 00:00:00+00:00
..              ...                       ...
860  Sabrina Larsen                       NaT
861  Sabrina Larsen                       NaT
862  Sabrina Larsen                       NaT
863  Sabrina Larsen                       NaT
864            None                       NaT

                                                COMMENTS STAT_AREA_VERSION_ID  \
0    Yukon District, 6 Subdistrict and 6-A Section ...                  None
1    Yukon District, 6 Subdistrict and 6-B Section ...                  None
2      Yukon District and 3 Subdistrict until 1/1/1980                  None
3                                                 None                  None
4                                                 None                  None
..                                                 ...                   ...
860                                               None                  None
861                                               None                  None
862                                               None                  None
863                                               None                  None
864                                               None                  None

    Shape_Length Shape_Area                                          geometry
0       4.610183   0.381977  MULTIPOLYGON (((-151.32805 64.96913, -151.3150...
1       3.682421   0.321943  MULTIPOLYGON (((-149.96255 64.70518, -149.9666...
2       2.215641   0.198740  MULTIPOLYGON (((-161.39853 61.55463, -161.4171...
```

```
3       9.179852   0.382788  MULTIPOLYGON (((-153.15234 65.24944, -153.0761...
4       9.500826   0.378262  MULTIPOLYGON (((-152.99905 65.17027, -152.9897...
..           ...        ...                                                ...
860     0.223565   0.000408  MULTIPOLYGON (((-142.90787 60.09177, -142.9051...
861     0.030506   0.000006  MULTIPOLYGON (((-143.00416 60.07711, -143.0046...
862     0.019805   0.000012  MULTIPOLYGON (((-143.28504 60.05800, -143.2861...
863     0.096016   0.000238  MULTIPOLYGON (((-143.49701 60.04832, -143.5083...
864     0.016237   0.000006  MULTIPOLYGON (((-139.15063 59.30748, -139.1489...

[865 rows x 37 columns]>
```

Note the existence of the `geometry` column. This is where the actual geospatial points that comprise the vector data are stored, and this brings up the important difference between raster and vector data - while raster data is regularly gridded at a specific resolution, vector data are just points in space.



Source: Leah A. Wasser, Megan A. Jones, Zack Brym, Kristina

Riemer, Jason Williams, Jeff Hollister, Mike Smorul. Raster 00: Intro to Raster Data in R. National Evologial Observatory Network (NEON).

The diagram above shows the three different types of geometries that geospatial vector data can take, points, lines or polygons. Whatever the geometry type, the geometry information (the x,y points) is stored in the column named `geometry` in the geopandas data frame. In this example, we have a dataset containing polygons of fishing districts. Each row in the dataset corresonds to a district, with unique attributes (the other columns in the dataset), and it's own set of points defining the boundaries of the district, contained in the `geometry` column.

```
comm['geometry'][:5]
```

```
/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
```

	geometry
0	MULTIPOLYGON (((-151.32805 64.96913, -151.3150...
1	MULTIPOLYGON (((-149.96255 64.70518, -149.9666...
2	MULTIPOLYGON (((-161.39853 61.55463, -161.4171...
3	MULTIPOLYGON (((-153.15234 65.24944, -153.0761...
4	MULTIPOLYGON (((-152.99905 65.17027, -152.9897...

So, now we know where our x,y points are, where is all of the other information like the `crs`? With vector data, all of this information is contained within the geodataframe. We can access the `crs` attribute on the data frame and print it like so:

```
comm.crs
```

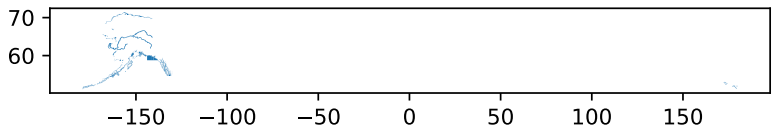
```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
```

```
- 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
```

Now that we know a little about what we are working with, let's get this data ready to work with. First, we can make a plot of it just to see what we have.

```
comm.plot()
```

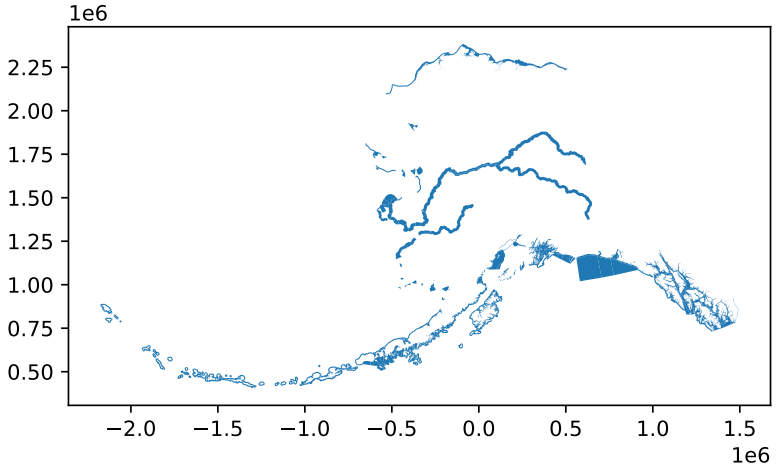
```
<AxesSubplot:>
```



This plot doesn't look so good. Turns out, these data are in WGS 84 (EPSG 4326), as opposed to Alaska Albers (EPSG 3338), which is what our raster data is in. To make pretty plots, and allow our raster data and vector data to be analyzed together, we'll need to reproject the vector data into 3338. To to this, we'll use the `to_crs` method on our `comm` object, and specify as an argument the projection we want to transform to.

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

comm_3338.plot()
```

```
<AxesSubplot:>
```



Much better!

11.4 Crop data to area of interest

For this example, we are only interested in south central Alaska, encompassing Prince William Sound, Cook Inlet, and Kodiak. Our raster data is significantly larger than that, and the vector data is statewide. So, as a first step we might want to crop our data to the area of interest.

First, we'll need to create a bounding box. 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.

```
from shapely.geometry import box

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(...)`), then update the metadata associated with the raster.

```
import rasterio.mask
import numpy as np

with rasterio.open("Coastal_2020_08.tif") as ship_con:
    shipc_arr, shipc_transform = rasterio.mask.mask(ship_con, coord_box_df["geometry"], crop
    shipc_meta = ship_con.meta
    # select just the 2-D array (by default a 3-D array with 1 band is returned)
    shipc_arr = shipc_arr[0,:,:]
    # turn the no-data values into NaNs.
    shipc_arr[shipc_arr == ship_con.nodata] = np.nan
```

```
shipc_meta.update({"driver": "GTiff",
                   "height": shipc_arr.shape[0],
                   "width": shipc_arr.shape[1],
                   "transform": shipc_transform,
                   "compress": "lzw"})
```

Next, we'll use a spatial inner join for the vector data to select polygons that are within the bounding box.

```
comm_clip = gpd.sjoin(comm_3338, coord_box_df, how='inner', predicate='within')
```

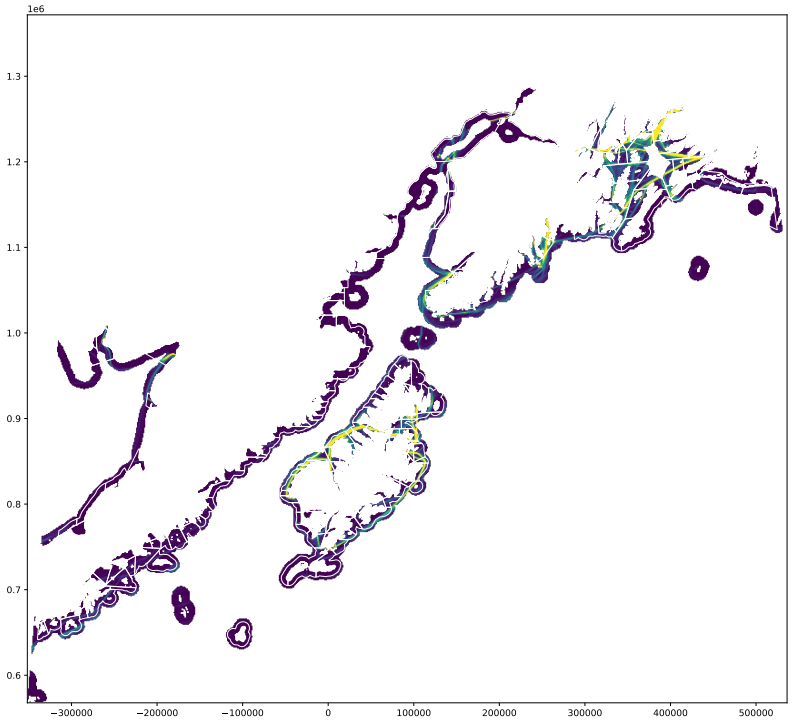
11.4.1 Check extents

Now let's look at the two cropped datasets overlayed on each other to ensure that the extents look right.

```
import rasterio.plot

# set up plot
fig, ax = plt.subplots(figsize=(15, 15))
# plot the raster
rasterio.plot.show(shipc_arr,
                   ax=ax,
```

```
                    vmin = 0,
                    vmax = 50000,
                    transform = shipc_transform)
# plot the vector
comm_clip.plot(ax=ax, facecolor='none', edgecolor='white')
```

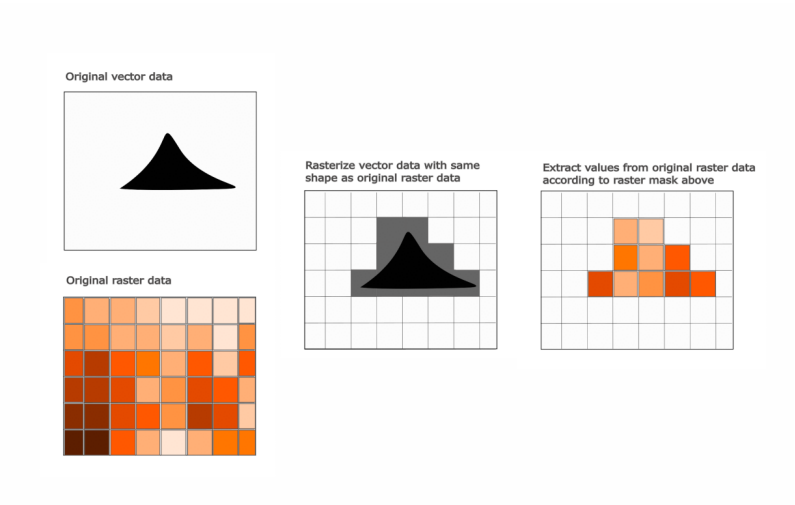
```
<AxesSubplot:>
```



11.5 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:



11.5.0.1 Zonal statistics over one polygon

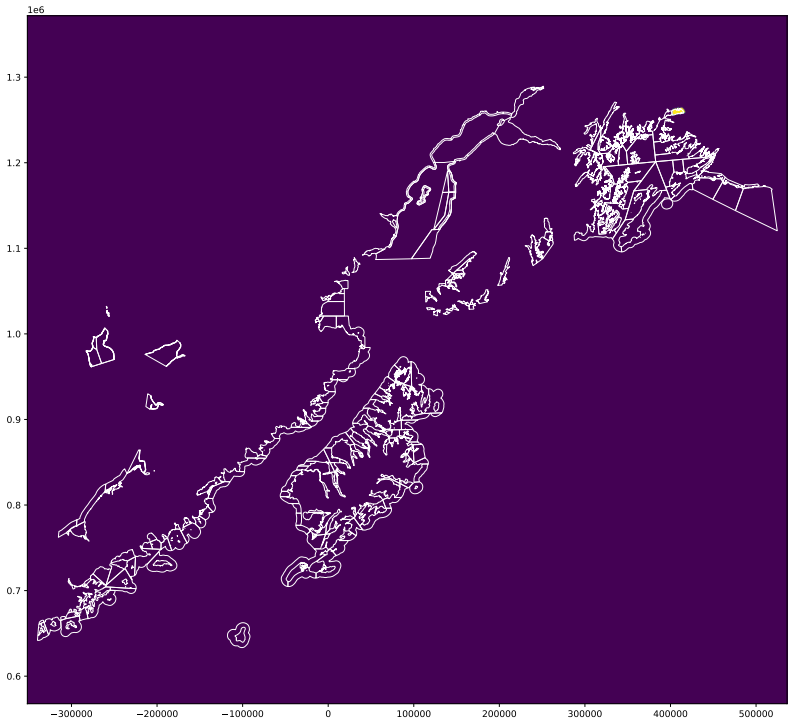
Let's look at how this works over just one fishing area first. We use the `rasterize` method from the `features` module in `rasterio`. This takes as arguments the data to rasterize (in this case the 40th row of our dataset), the shape and transform the output raster will take (these were extracted from 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.

```
from rasterio import features

r40 = features.rasterize(comm_clip['geometry'][40].geoms,
                                    out_shape=shipc_arr.shape,
                                    transform=shipc_meta['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 = shipc_meta['transform'])
# plot the vector
comm_clip.plot(ax=ax, facecolor='none', edgecolor='white')
```

```
<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(shipc_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_clip['id'] = range(0,len(comm_clip))
```

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_clip['geometry'], comm_clip['id']):
    rasterized = features.rasterize(geom.geoms,
                                    out_shape=shipc_arr.shape,
                                    transform=shipc_meta['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(shipc_arr[r_index])
```

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

```
import pandas as pd

# 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
```

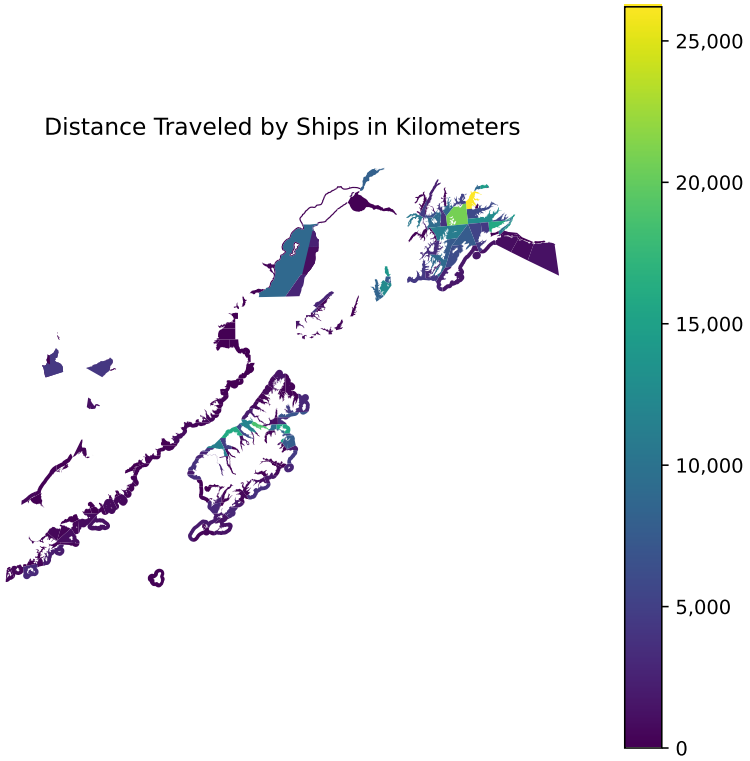
Now we join the result to the original geodataframe.

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

Finally, we can plot our result!

```
import matplotlib.ticker
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()
```

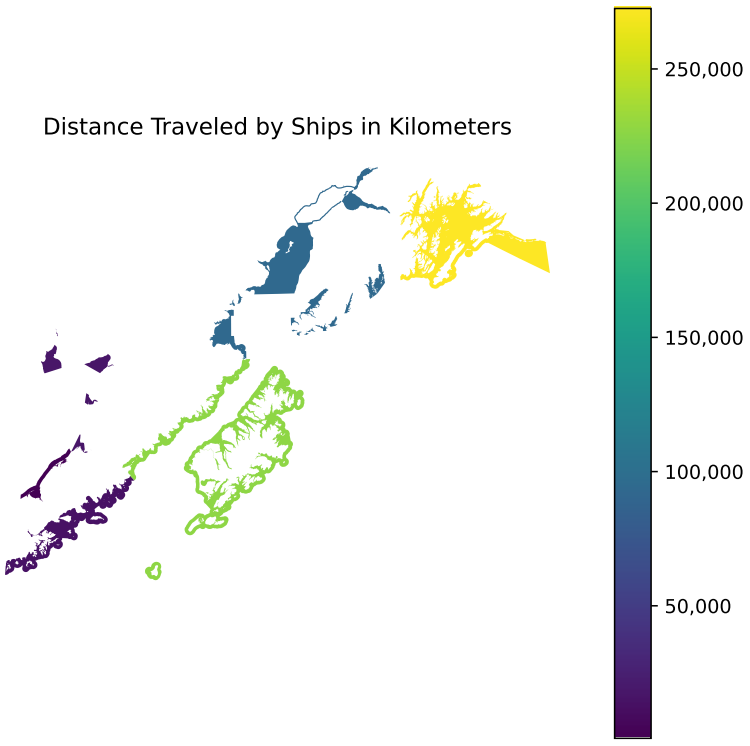


From here we can do any additional `geopandas` operations we might be interested in. For example, what if we want to calculate the total distance by registration area (a superset of fishing district). We can do that using `dissolve` from geopandas.

```
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()
```



11.6 Summary

We covered a lot of ground here, so let's recap some of the high level points:

- Raster data consists of regularly gridded values, and can be represented in python as an array
- Vector data consists of any number of points, that might be connected, and is represented in python as a geodataframe
- We can do geospatial operations like changing the projection or cropping the data to a particular extent on both raster and vector data
- You can use vector data to help analyze raster data (and vice versa!) by rasterizing the vector data and using numpy operations on the resulting array.

12 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 when they are read into memory.

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 rows with all the 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
```

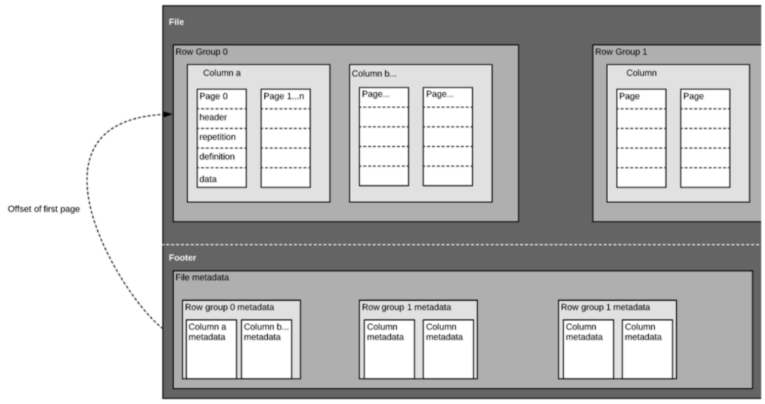
And the read order would first be the names, then the locations, then the age. This means that selecting all values from a set of columns is quite easy (all of the Names and Ages, or all Names and Locations), but reading in only the first few records from each column would require reading in the entire 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 metadata 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='hive')
```

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("Taxa")=="Dorosoma petenense")|
        (ds.field("Taxa")=="Morone saxatilis") |
        (ds.field("Taxa")== "Spirinchus thaleichthys"))

fishf = deltafish.to_table(filter = expr, columns =['SampleID', 'Length', 'Count', 'Taxa'])
```

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='hive')
```

Take a look at the columns again:

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

Let's pick out only the ones we are interested in.

```
survey_s = survey.to_table(columns=['SampleID','Datetime', 'Station', 'Longitude', 'Latitude
```

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

```
fish_j = fishf.join(survey_s, "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 for Scalable Computing

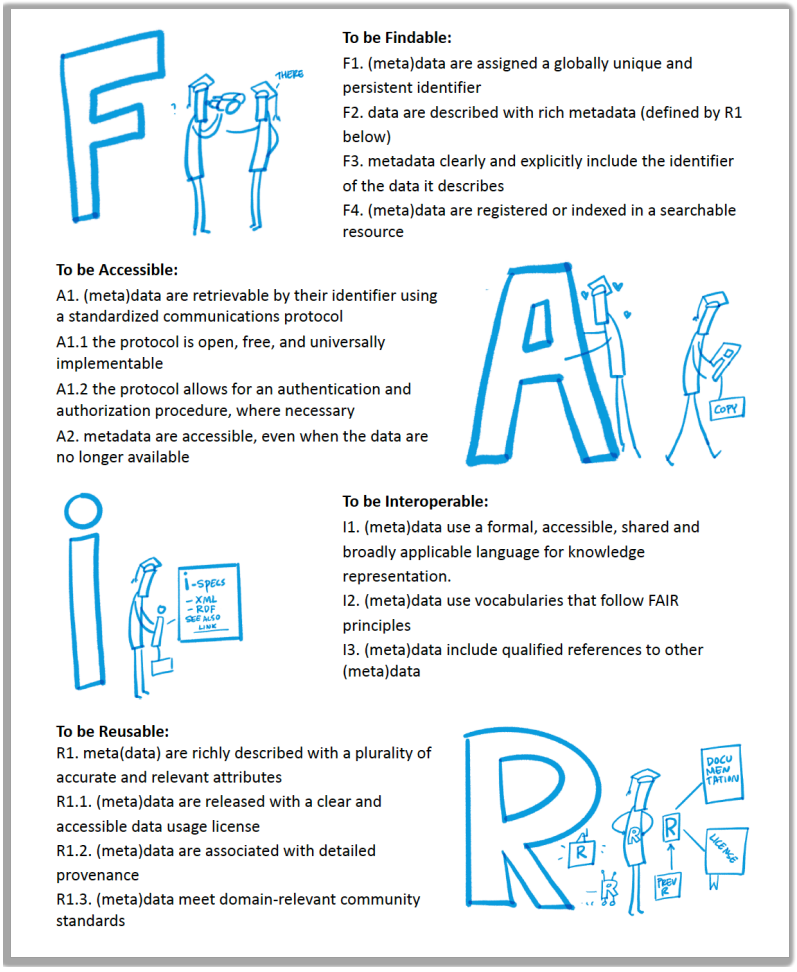
To recap, the Arctic Data Center is an openly-accessible data repository and the data published through the repository is open for anyone to reuse, subject to conditions of the license (at the Arctic Data Center, data is released under one of two licenses: CC-0 Public Domain and CC-By Attribution 4.0). In facilitating use of data resources, the data stewardship community has converged on principles surrounding best practices for open data management.

Two principles that the Arctic Data Center explicitly adopts are FAIR Principles (Findable, Accessible, Interoperable, and Reproducible) and CARE Principles for Indigenous Governance (Collective Benefit, Authority to Control, Responsibility, Ethics).

Menti question:

1. What is your familiarity with FAIR principles?
2. What is your familiarity with CARE principles?

FAIR and CARE principles are relevant in the context of data ethics for multiple reasons. FAIR speaks to how metadata is managed, stored, and shared.



FAIR principles and open science are overlapping concepts, but are distinctive from one another. Open science supports a culture of sharing research outputs and data, and FAIR focuses on how to prepare the data. **The FAIR principles place emphasis on machine readability, "distinct from peer initiatives that focus on the human scholar" (Wilkinson et al 2016) and as such, do not fully engage with sensitive data considerations and with Indigenous rights and interests (Research Data Alliance International Indigenous Data Sovereignty Interest Group, 2019).** Metadata can be FAIR but not open. For example, sensitive data (data that contains personal information) may not be appropriate to share, however sharing the anonymized metadata that is easily

understandable will reduce research redundancy.

Research has historically perpetuated colonialism and represented extractive practices, meaning that the research results were not mutually beneficial. These issues also related to how data was owned, shared, and used. **To address issues like these, the Global Indigenous Data Alliance (GIDA) introduced CARE Principles for Indigenous Data Governance to support Indigenous data sovereignty.** CARE Principles speak directly to how the data is stored and shared in the context of Indigenous data sovereignty. CARE Principles stand for:

- *Collective Benefit* - Data ecosystems shall be designed and function in ways that enable Indigenous Peoples to derive benefit from the data
- *Authority to Control* - Indigenous Peoples' rights and interests in Indigenous data must be recognized and their authority to control such data be empowered. Indigenous data governance enables Indigenous Peoples and governing bodies to determine how Indigenous Peoples, as well as Indigenous lands, territories, resources, knowledges and geographical indicators, are represented and identified within data.
- *Responsibility* - Those working with Indigenous data have a responsibility to share how those data are used to support Indigenous Peoples' self-determination and collective

benefit. Accountability requires meaningful and openly available evidence of these efforts and the benefits accruing to Indigenous Peoples.

- *Ethics* - Indigenous Peoples' rights and wellbeing should be the primary concern at all stages of the data life cycle and across the data ecosystem. To many, the FAIR and CARE principles are viewed by many as complementary: CARE aligns with FAIR by outlining guidelines for publishing data that contributes to open-science and at the same time, accounts for Indigenous' Peoples rights and interests.

15.1 Ethics at the Arctic Data Center

Transparency in data ethics is a vital part of open science. Regardless of discipline, various ethical concerns are always present, including professional ethics such as plagiarism, false authorship, or falsification of data, to ethics regarding the handling of animals, to concerns relevant to human subjects research. As the primary repository for the Arctic program of the National Science Foundation, the Arctic Data Center accepts Arctic data from all disciplines. Recently, a new submission feature was released which asks researchers to describe the ethical considerations that are apparent in their research. This question is asked to all researchers, regardless of disciplines.

Sharing ethical practices openly, similar in the way that data is shared, enables deeper discussion about data management practices, data reuse, sensitivity, sovereignty and other considerations. Further, such transparency promotes awareness and adoption of ethical practices.

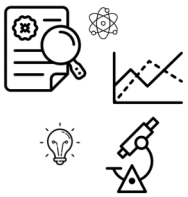
Inspired by CARE Principles for Indigenous Data Governance (Collective Benefit, Authority to Control, Responsibility, Ethics) and FAIR Principles (Findable, Accessible, Interoperable, Reproducible), we include a space in the data submission process for researchers to describe their ethical research practices. These statements are published with each dataset, and the purpose of these statements is to promote greater transparency in data collection and to guide other researchers.

For more information about the ethical research practices statement, check out this blog.

To help guide researchers as they write their ethical research statements, we have listed the following ethical considerations that are available on our website. The concerns are organized first by concerns that should be addressed by all researchers, and then by discipline.

Consider the following ethical considerations that are relevant for your field of research.

15.1.0.0.1 Ethical Considerations for all Arctic Researchers



Research Planning

1. Were any permits required for your research?
2. Was there a code of conduct for the research team decided upon prior to beginning data collection?
3. Was institutional or local permission required for sampling?
4. What impact will your research have on local communities or nearby communities (meaning the nearest community within a 100 mile radius)?

Data Collection

5. Were any local community members involved at any point of the research process, including study site identification, sampling, camp setup, consultation or synthesis?
6. Were the sample sites near or on Indigenous land or communities?

Data Sharing and Publication

7. How were the following concerns accounted for: misrepresentation of results, misrepresentation of experience, plagiarism, improper authorship, or the falsification or data?
8. If this data is intended for publication, are authorship expectations clear for everyone involved? Other professional ethics can be found here

15.1.0.0.2 Archaeological and Paleontological Research

Research Planning

1. Were there any cultural practices relevant to the study site? If yes, how were these practices accounted for by the research methodologies.

Data Collection

2. Did your research include the removal or artifacts?
3. Were there any contingencies made for the excavation and return of samples after cleaning, processing, and analysis?

Data Sharing and Publication 4. Were the samples deposited to a physical repository? 5. Were there any steps taken to account for looting threats? Please explain why or why not?

15.1.0.0.3 Human Participation and Sensitive Data

Research Planning

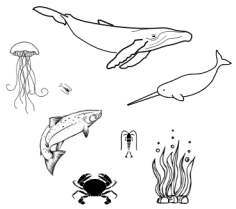
1. Please describe the institutional IRB approval that was required for this research.
2. Was any knowledge provided by community members?

Data Collection 3. Did participants receive compensation for their participation? 4. Were decolonization methods used?

Data Sharing and Publication

5. Have you shared this data with the community or participants involved?

15.1.0.0.4 Marines Sciences (e.g. Marine Biology Research)



Research Planning

1. Were any of the study sites or species under federal or local protection?

Data Collection

2. Were endangered, threatened, or otherwise special-status species collected?
3. How were samples collected? Please describe any handling practices used to collect data.
4. What safety measures were in place to keep researchers, research assistants, technicians, etc., out of harms way during research?
5. How were animal care procedures evaluated, and do they follow community norms for organismal care?

Data Sharing and Publication 6. Did the species or study area represent any cultural importance to local communities, or include culturally sensitive information? Please explain how you came to this conclusion and how any cultural sensitivity was accounted for.

15.1.0.0.5 Physical Sciences (e.g. Geology, Glaciology, and Ice Research)

Research Planning 1. Was any knowledge provided by community members, including information regarding the study site?

Data Collection

2. What safety measures were in place to keep researchers, research assistants, technicians, etc., out of harms way during research?
3. Were there any impacts to the environment/habitat before, during or after data collection?

Data Sharing and Publication

4. Is there any sensitive information including information on sensitive sites, valuable samples, or culturally sensitive information?

15.1.0.0.6 Plant and Soil Research

Research Planning

1. Were any of the study sites protected under local or federal regulation?
2. Was any knowledge provided by nearby community members, including information regarding the study site?

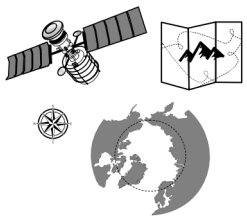
Data Collection

3. Did sample collection result in erosion of soil or other physical damage? If so, how was this addressed?
4. What safety measures were in place to keep researchers, research assistants, technicians, etc., out of harms way during research?

Data Sharing and Publication

5. Do any of the study sites or specimens represent culturally sensitive areas or species? Please explain how you came to this conclusion, and if yes, how was this accounted for?

15.1.0.0.7 Spatial Data



Research Planning

1. Were any land permits required for this research?

Data Collection

2. Were any data collected using citizen science or community participation?
3. If yes, were community members compensated for their time and made aware of their data being used and for what purpose?

Data Sharing and Publication

4. If data were ground-truthed, was institutional or local permissions required and/or obtained for land/property access?
5. Have you shared this data with the community or participants involved?
6. If location sensitive data was obtained (endangered/threatened flora & fauna location, archaeological and historical sites, identifiable ships, sensitive spatial information), how were the data desensitized?

15.1.0.0.8 Wildlife Sciences (e.g. Ecology and Biology Research)

Research Planning

1. Were any permits required for data sampling?

Data Collection

2. Were endangered, threatened, or otherwise special-status plants or animal species collected?
3. How were samples collected? Please describe any handling practices used to collect data.
4. What safety measures were in place to keep researchers, research assistants, technicians, etc., out of harms way during research?
5. How were animal care procedures evaluated, and do they follow community norms for organism care?

Data Sharing and Publication

6. Do any of the study sites or specimens represent culturally sensitive areas or species? Please explain how you came to this conclusion, and if yes, how was this accounted for?

Menti question:

1. Have you thought about any of the ethical considerations listed above before?
2. Were any of the considerations new or surprising?
3. Are there any for your relevant discipline that are missing?

15.2 Ethics in Machine Learning

Menti poll 1. What is your level of familiarity with machine learning 2. Have you thought about ethics in machine learning prior to this lesson? 3. Can anyone list potential ethical considerations in machine learning?

What comes to mind when considering ethics in machine learning?

The stories that hit the news are often of privacy breaches or biases seeping into the training data. Bias can enter at any point of the research project, from preparing the training data, designing the algorithms, to collecting and interpreting the data. When working with sensitive data, a question to also consider is how to deanonymize, anonymized data. A unique aspect to machine learning is how personal bias can influence the analysis and outcomes. A great example of this is the case of ImageNet.

15.2.1 ImageNet: A case study of ethics and bias in machine learning

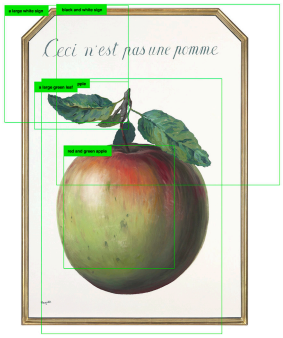


Image source: Kate Crawford and Trevor Paglen, "Excavating AI: The Politics of Training Sets for Machine Learning" (September 19, 2019).

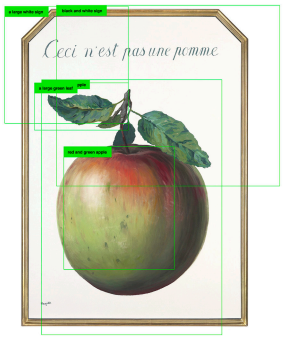
ImageNet is a great example of how personal bias can enter machine learning through the training data. ImageNet was a training data set of photos that was used to train image classifiers. The data set was initially created as a large collection of pictures, which were mainly used to identify objects, but

some included images of people. The creators of the data set created labels to categorize the images, and through crowdsourcing, people from the internet labeled these images. (This example is from Kate Crawford and Trevor Paglen, "Excavating AI: The Politics of Training Sets for Machine Learning", September 19, 2019).

15.2.1.1 Discussion:

1. Where are the two areas bias could enter this scenario?
2. Are there any ways that this bias could be avoided?
3. While this example is specific to images, can you think of any room for bias in your research?

15.3 References and Further Reading



Carroll, S.R., Herczog, E., Hudson, M. et al. (2021) Operationalizing the CARE and FAIR Principles for Indigenous data futures. Sci Data 8, 108 https://doi.org/10.1038/s41597-021-00892-0

Chen, W., & Quan-Haase, A. (2020) Big Data Ethics and Politics: Towards New Understandings. Social Science Computer Review. https://journals.sagepub.com/doi/10.1177/0894439318810734

Crawford, K., & Paglen, T. (2019) Excavating AI: The Politics of Training Sets for Machine Learning. https://excavating.ai/

Gray, J., & Witt, A. (2021) A feminist data ethics of care framework for machine learning: The what, why, who and how. First Monday, 26(12), Article number: 11833

Puebla, I., & Lowenberg, D. (2021) Recommendations for the Handling for Ethical Concerns Relating to the Publication of Research Data. FORCE 11. https://force11.org/post/recommendations-for-the-handling-of-ethical-concerns-relating-to-the-publication-of-research-data/

Research Data Alliance International Indigenous Data Sovereignty Interest Group. (2019). "CARE Principles for Indigenous Data Governance." The Global Indigenous Data Alliance. GIDA-global.org

Wilkinson, M., Dumontier, M., Aalbersberg, I. et al. (2016) The FAIR Guiding Principles for scientific data management and stewardship. Sci Data 3, 160018. https://doi.org/10.1038/sdata.2016.18

Zwitter, A., Big Data ethics. (2014) Big Data and Society. DOI: 10.1177/2053951714559253

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 1.1: Getting started with Google Earth Engine (GEE)

> **i** GEE libraries were installed when you set up your `scomp` virtual environment
>
> We'll be using a few different libraries in the next exercises/demos that should already be installed if you successfully set up your `scomp` virtual environment. If you eventually find yourself working outside our virtual environment, you'll want need to install the following libraries:
>
> ```
> #| eval: false
> pip install earthengine-api
> pip install ee # Google Earth Engine interface
> pip install geemap # package for interactive maping with GEE
> ```

1. Create a Google Earth Engine account (if you haven't already done so) – it's best to use a personal gmail address.

- Find instructions on how to do so here

> **i** Combine this step with importing libraries below??

2. Set up GEE Authentication

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

16.3 Exercise 1.2: 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
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

SAM NOTE DELETE LATER * 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 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).

Reanalysis combines observations with model data to provide the most complete picture of past weather and climate. To read more about reanalyses, check out the European Centre for Medium-Range Weather Forecasts' webites

```
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 Demo

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
- Uploading large datasets to the Arctic Data Center
- Workflow tools

18.1 NSF policy for large datasets

Many different research methods 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 NSF requirements state:

> 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.

This means that datasets that are already published on a long lived archive do not need to be replicated to the Arctic Data Center, only a metadata record needs to be included. Often, the

curation staff at the Arctic Data Center can replicate metadata programmatically such that the researcher in this case doesn't have to publish their data twice. As an example of the myriad of scenarios that can arise in this realm, say a research project accesses many terabytes of VIIRS satellite data. In this case, the original satellite data does not need to be republished on the Arctic Data Center, since it is already available publicly, but the code that accessed it, and derived products, can be published, along with a citation to the satellite data indicating provenance.

Similarly, 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. However, 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, and our curation team is there to help researchers make decisions alongside NSF program officers, if necessary, to decide which portions of a large-data collection effort should be published in the archive.

18.2 Data transfer tools

Now that we've talked about what types of large datasets you might have that need to get published on the Arctic Data Center, let's discuss how to actually get the data there. If you have even on the order of only 50GB, or more than 500 files, it will likely be more expedient for you to transfer your files via a command line tool than uploading them via our webform. So you know that you need to move a lot of data, how are you going to do it? More importantly, how can you do it in an efficient way?

There are three key elements to data transfer efficiency:

- endpoints
- network
- transfer tool

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. One key factor that affects data transfer speed is how quickly that machine can write data to disk. Slow write speeds will throttle a data transfer on even the fastest internet connection with the most streamlined transfer tool. Examples of endpoints could be:

- NCEAS `included-crab` server
- Your standard laptop
- A cloud service like AWS

Network speed

Network speed determines how quickly information can be sent between endpoints. It is largely, but not enitrely, dependent on what you pay for. Importantly, not all networks are created equal, even if they nominally have the same speed capability. Wired networks get significantly more speed than wireless. Networks with lots of "stuff" along the pipe (like switches or firewalls) can perform worse than those that don't. Even the length and type of network cabling used can matter.

Transfer tools

Poll: what data transfer tools do you use regularly?

Finally, the tool or software that you use to transfer data can also significantly affect your transfer speed. There are a lot of tools out there that can move data around, both GUI driven and command line. We'll discuss a few here, and their pros and cons.

18.2.0.0.1 * `scp`

`scp` or secure copy uses `ssh` for authentication and transfer, and it is included with both unix and linux. It requires no setup (unless you are on a Windows machine and need to install), and if you can `ssh` to a server, you can probably use `scp` to move files without any other setup. `scp` copies all files linearly and simply. If a transfer fails in the middle, it is difficult to know exactly what files didn't make it, so you might have to start the whole thing over and re-transfer all the files. This, obviously, would not be ideal for large data transfers. For a file or two, `scp` is a fine tool to use.

18.2.0.0.2 * `rsync`

`rsync` is similar to `scp`, but syncs files/directories as opposed to copying. This means that `rsync` checks the destination to see if that file (with the same size and modified date) already exists. If it does, `rsync` will skip the file. This means that if an `rsync` transfer fails, it can be restarted again and will pick up where it left off, essentially. Neat!

18.2.0.0.3 * Globus

Globus is a software that uses multiple network sockets simultaneously on endpoints, such that data transfers can run in parallel. As you can imagine, that parallelization can dramatically speed up data transfers. Globus, like `rsync` can also fail gracefully, and even restart itself. Globus does require that each endpoint be configured as a Globus node, which is more setup than is required of either `scp` or `rsync`. Many instituions computing resources may have endpoints already configured as Globus endpoints, so it is always worth checking in with any existing resources that might already be set up before setting up your own. Although Globus is a free software, there are paid options which provide support for configuring your local workstation as a Globus node. Globus is a fantastic tool, but remember the other two factors controlling data transfer, it can only help so much in overcoming slow network or write speeds.

18.2.0.1 AWS `sync`

Amazon Web Services (AWS) has a Command Line Interface (CLI) that includes a `sync` utility. This works much like `rsync` does in that it only copies new or updated files to the destination. The difference, of course, is that AWS `sync` is specifically built to work with interacting with the AWS cloud, and is compatible with S3 buckets.

18.2.0.2 `nc`

`nc` (or netcat) is a low level file transfer utility that is extremely efficient when moving files around on nodes in a cluster. It is not the easiest of these tools to use, however, in certain situations it might be the best option because it has the least overhead, and therefore can run extremely efficiently.

18.3 Documenting large datasets

The Arctic Data Center works hard to support datasets regardless of size, but we have performance considerations as well, and large datasets sometimes need special handling and require more processing time from our curation team. To help streamline a large dataset submission we have the following reccommendations:

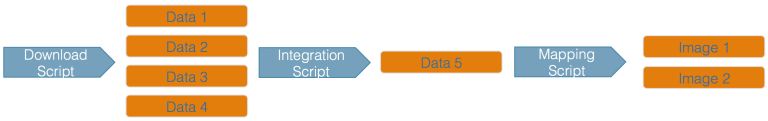
- use self documenting file formats, for metadata efficiency
  - netcdf
  - geotiff, geopackage
- regular, parseable filenames and consistent file formatting
- communicate early and often with the Arctic Data Center staff, preferably before you start a submission and well before your final report is due

18.4 Workflow tools

Preparing data for running analysis, models, and visualization processes can be complex, with many dependencies among datasets, as well as complex needs for data cleaning, munging, and integration that need to occur before "analysis" can begin.

Many research projects would benefit from a structured approach to organizing these processes into workflows. A research workflow is an ordered sequence of steps in which the outputs of one process are connected to the inputs of the next in a formal way. Steps are then chained together to typically create a directed, acyclic graph that represents the entire data processing pipeline.

This hypothetical workflow shows three processing stages for downloading, integrating, and mapping the data, along with the outputs of each step. This is a simplified rendition of what is normally a much more complex process.



Whether simple or complex, it is helpful to conceptualize your entire workflow as a directed graph, which helps to identify the explicit and implicit dependencies, and to plan work collaboratively.

18.4.1 Workflow dependencies and encapsulation

While there are many thousands of details in any given analysis, the reason to create a workflow is to structure all of those details so that they are understandable and traceable. Being explicit about dependencies and building a hierarchical workflow that encapsulates the steps of the work as independent modules. So the idea is to focus the workflow on the major steps in the pipeline, and to articulate each of their dependencies.

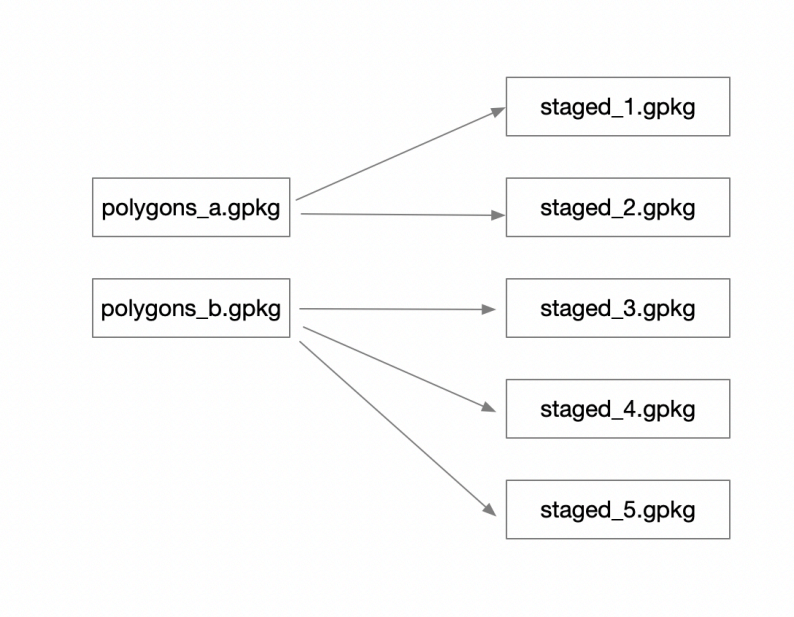
Workflows can be implemented in many ways, with various benefits:

- as a conceptual diagram
- as a series of functions that perform each step through a controlling script
- as a series functions managed by a workflow tool like `parsl`, `snakemake`, or `ray`
- many others…

18.4.2 DAGs

While managing workflows solely as linked functions works, the presence of side-effects in a workflow can make it more difficult to efficiently run only the parts of the workflow where items have changed. Many workflow systems have been created to provide a structured way to specify, analyze, and track dependencies, and to execute only the parts of the workflow that are needed.

A Directed Acyclic Graph (DAG) is a diagram that shows the dependencies of a workflow, whether they are data dependencies or process dependencies. Below is an example of a simplified DAG of the first step of the group project:



This graph only shows data dependencies, but process dependencies can also exist.

In your groups, draw out a simplified version of the rest of the workflow. What dependencies, both data and process based, exist?

A more realistic workflow: https://github.com/NCEAS/scalable-computing-examples/tree/main/workflows

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.