Scalable and Computationally Reproducible Approaches to Arctic Research

S. Jeanette Clark, Matthew B. Jones, Samantha Csik, Carmen Galaz García, Bryce Mc

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, 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.

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. Parquet and Arrow** (Jeanette)		
10:30-11:00	BREAK	BREAK	BREAK	BREAK	BREAK
11:00-11:30 11:30-12:00	**3. Python programming on clusters** (Jeanette)	**7. Software design I** (Matt)	**12. Software Design II** (Matt)	**16. Billions of Ice Wedge Polygons** (Chandi)	**20. Reproducibility redux via containers** (Matt) **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 16:00-16:30	**5. Documenting and Publishing Data** (Daphne)	**9. Parallelization with Dask** (Carmen)	**14. Data Ethics** (Tash)	**18. Workflows for data staging and publishing** (Jeanette)	
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

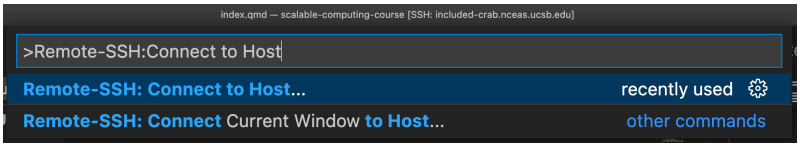
First, download VS Code if you do not already have it installed.

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

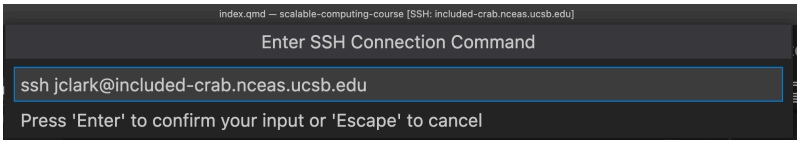
Log in to the server

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"

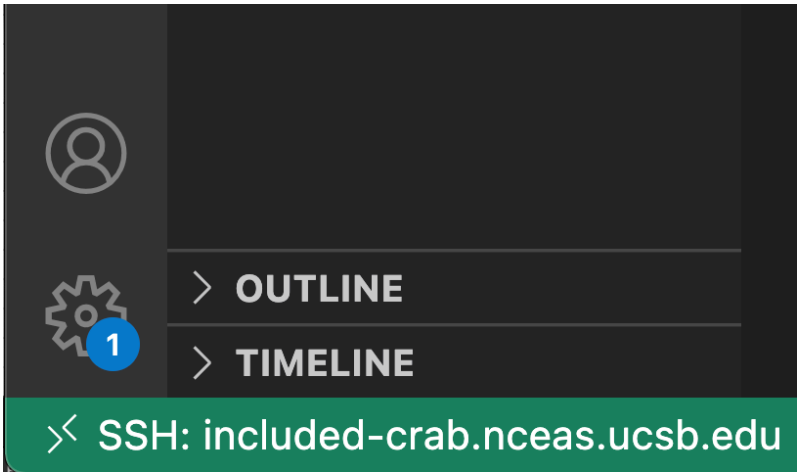


- select "Add New SSH Host"
- enter the ssh command to connect to the host as if in a terminal (`ssh username@included-crab.nceas.ucsb.edu`)
  - Note: you will only need to do this step once



- 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. 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.**

> **!** Important
>
> GEE authentication (more on that in Lesson 15) uses Cloud Projects. Some organizations control who can create Cloud Projects, which may prevent you from completing the authentication process. To circumvent authentication issues, we recommend creating your GEE account using a non-organizational account (e.g. a personal email account). Check out GEE's authentication troubleshooting recommendations if you continue to run into issues.

About this book

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

Citation: S. Jeanette Clark, Matthew B. Jones, Samantha Csik, Carmen Galaz García, Bryce Mecum, Natasha Haycock-ChavezDaphne Virlar-Knight. 2022. Scalable and Computationally Reproducible Approaches to Arctic Research.

Additional contributors: Amber E. Budden, Noor Johnson

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

1 Welcome and Overview

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.

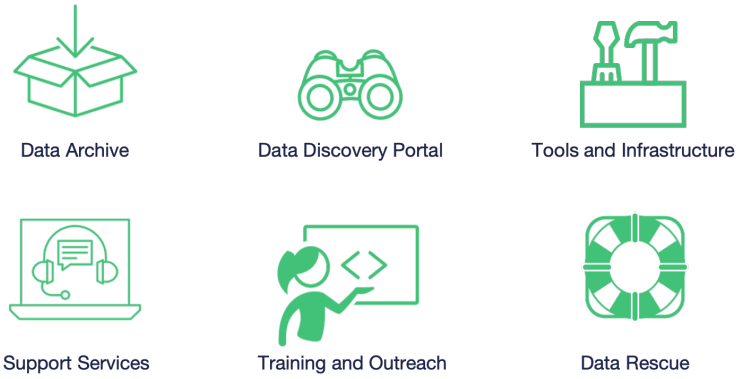
- Welcome and Introductions
- Mission and structure of the Arctic Data Center
- Plan for the Scalable Computing Course

1.1 Arctic Data Center Overview

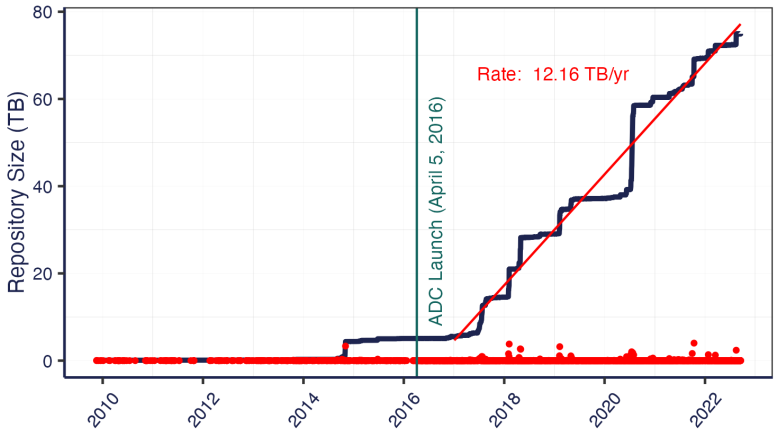
The Arctic Data Center is the primary data and software repository for the Arctic section of National Science Foundation's Office of Polar Programs (NSF OPP).

We're best known in the research community as a data archive – researchers upload their data to preserve it for the future and make it available for re-use. This isn't the end of that data's life, though. These data can then be downloaded for different

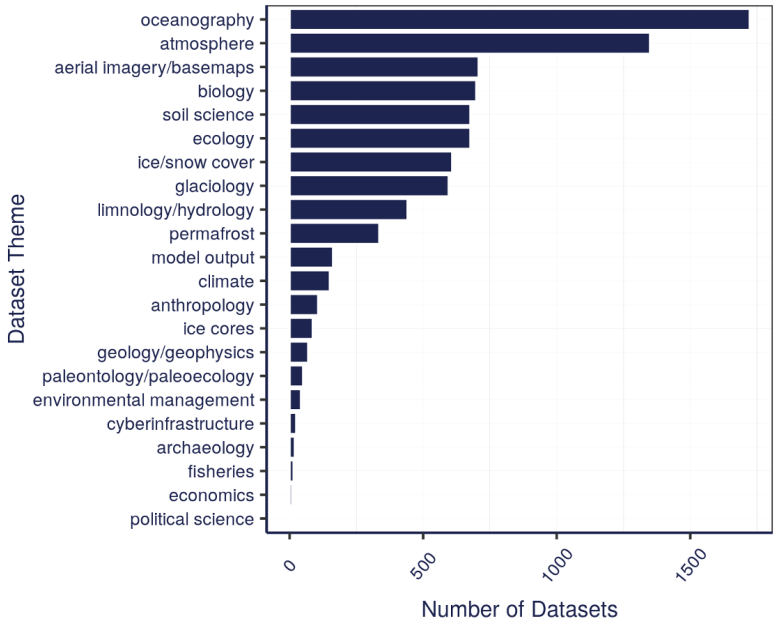
analyses or synthesis projects. In addition to being a data discovery portal, we also offer top-notch tools, support services, and training opportunities. We also provide data rescue services.



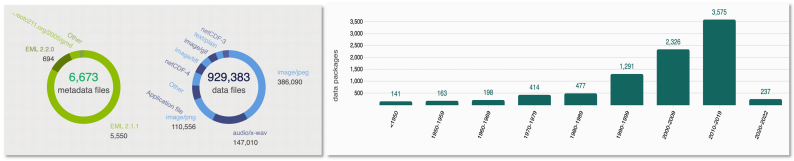
NSF has long had a commitment to data reuse and sharing. Since our start in 2016, we've grown substantially – from that original 4 TB of data from ACADIS to now over 71 TB. In 2021 alone, we saw 16% growth in dataset count, and about 30% growth in data volume. This increase has come from advances in tools – both ours and of the scientific community, plus active community outreach and a strong culture of data preservation from NSF and from researchers. We plan to add more storage capacity in the coming months, as researchers are coming to us with datasets in the terabytes, and we're excited to preserve these research products in our archive. We're projecting our growth to be around several hundred TB this year, which has a big impact on processing time. Give us a heads up if you're planning on having larger submissions so that we can work with you and be prepared for a large influx of data.



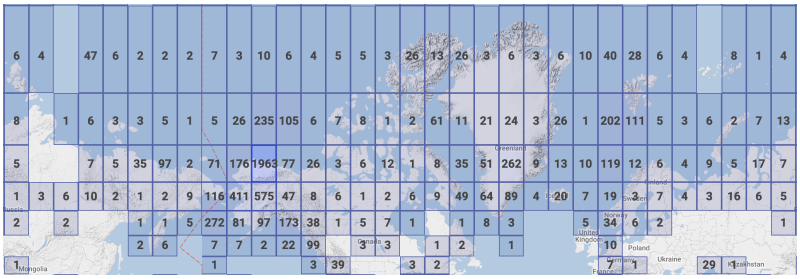
The data that we have in the Arctic Data Center comes from a wide variety of disciplines. These different programs within NSF all have different focuses – the Arctic Observing Network supports scientific and community-based observations of biodiversity, ecosystems, human societies, land, ice, marine and freshwater systems, and the atmosphere as well as their social, natural, and/or physical environments, so that encompasses a lot right there in just that one program. We're also working on a way right now to classify the datasets by discipline, so keep an eye out for that coming soon.



Along with that diversity of disciplines comes a diversity of file types. The most common file type we have are image files in four different file types. Probably less than 200-300 of the datasets have the majority of those images – we have some large datasets that have image and/or audio files from drones. Most of those 6600+ datasets are tabular datasets. There's a large diversity of data files, though, whether you want to look at remote sensing images, listen to passive acoustic audio files, or run applications – or something else entirely. We also cover a long period of time, at least by human standards. The data represented in our repository spans across centuries.

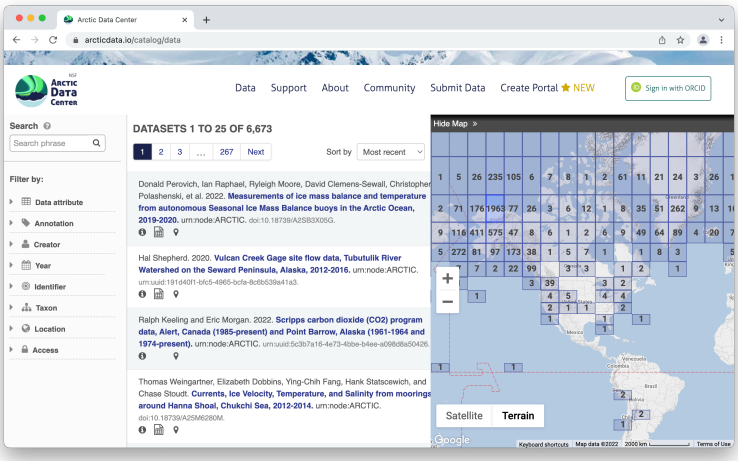


We also have data that spans the entire Arctic, as well as the sub-Arctic, regions.

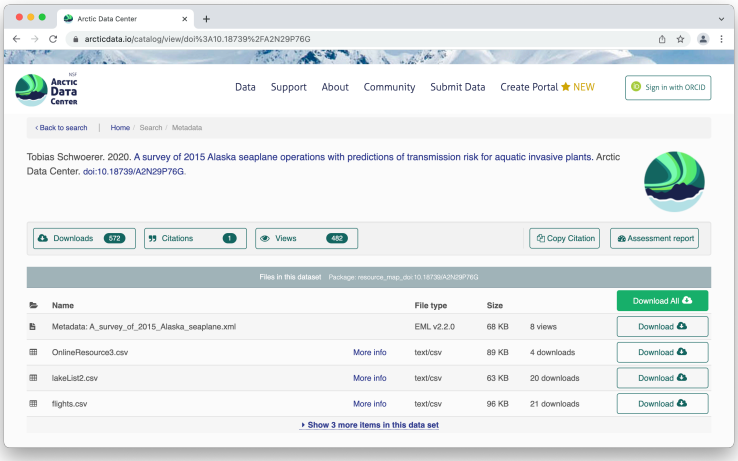


1.1.1 Data Discovery Portal

To browse the data catalog, navigate to arcticdata.io. Go to the top of the page and under data, go to search. Right now, you're looking at the whole catalog. You can narrow your search down by the map area, a general search, or searching by an attribute.

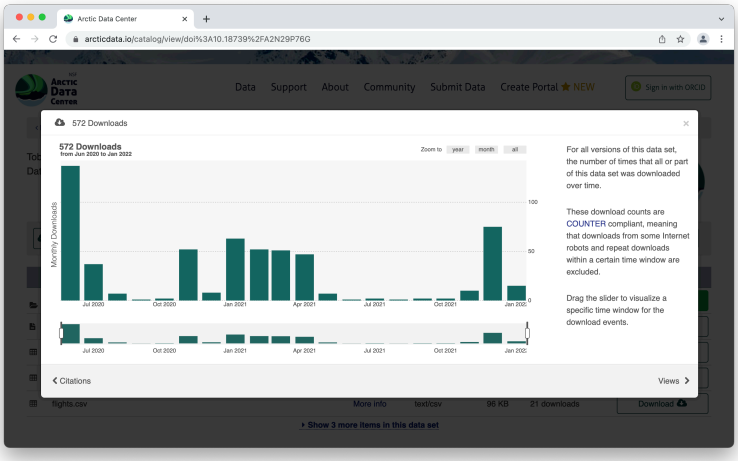


Clicking on a dataset brings you to this page. You have the option to download all the files by clicking the green "Download All" button, which will zip together all the files in the dataset to your Downloads folder. You can also pick and choose to download just specific files.



All the raw data is in open formats to make it easily accessible and compliant with FAIR principles – for example, tabular documents are in .csv (comma separated values) rather than Excel documents.

The metrics at the top give info about the number of citations with this data, the number of downloads, and the number of views. This is what it looks like when you click on the Downloads tab for more information.

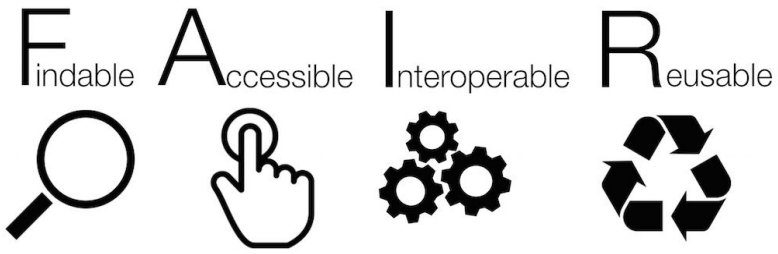


Scroll down for more info about the dataset – abstract, keywords. Then you'll see more info about the data itself. This

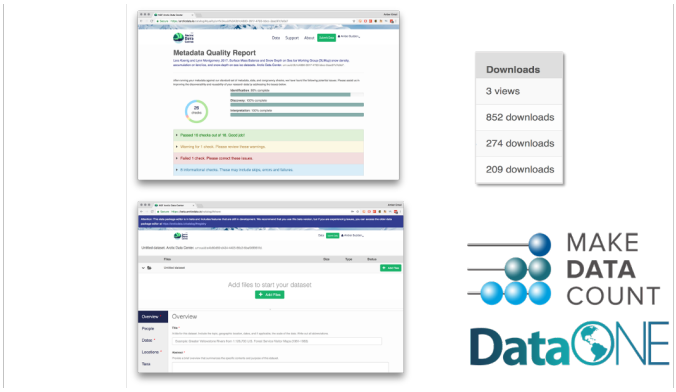
shows the data with a description, as well as info about the attributes (or variables or parameters) that were measured. The green check mark indicates that those attributes have been annotated, which means the measurements have a precise definition to them. Scrolling further, we also see who collected the data, where they collected it, and when they collected it, as well as any funding information like a grant number. For biological data, there is the option to add taxa.

1.1.2 Tools and Infrastructure

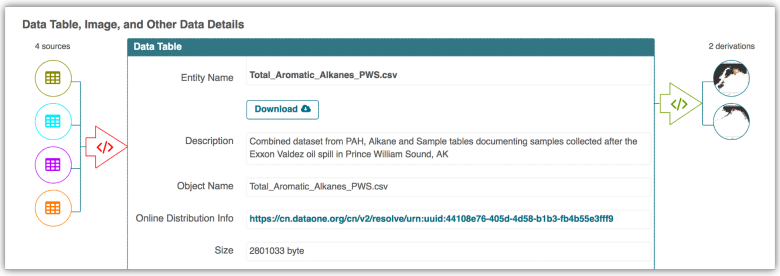
Across all our services and partnership, we are strongly aligned with the community principles of making data FAIR (Findable, Accesible, Interoperable and Reusable).



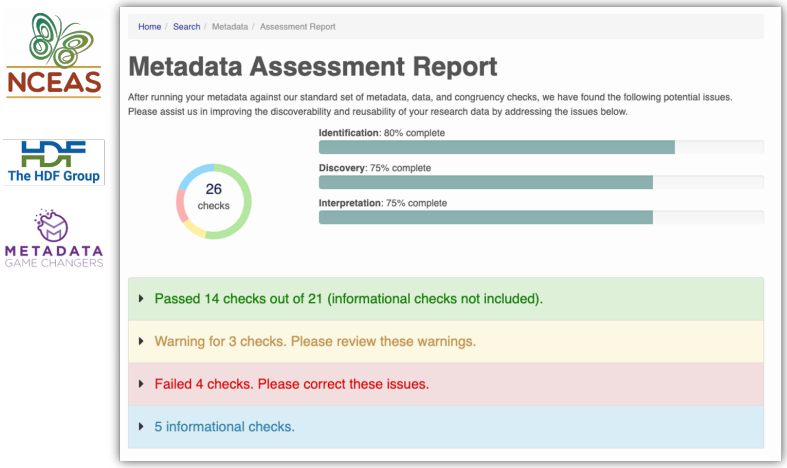
We have a number of tools available to submitters and researchers who are there to download data. We also partner with other organizations, like Make Data Count and DataONE, and leverage those partnerships to create a better data experience.



One of those tools is provenance tracking. With provenance tracking, users of the Arctic Data Center can see exactly what datasets led to what product, using the particular script that the researcher ran.

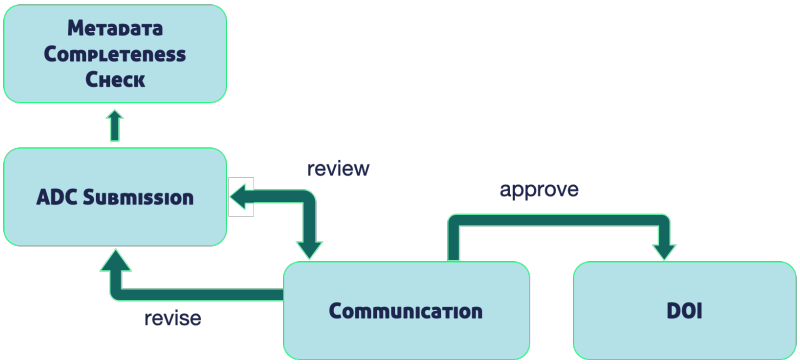


Another tool are our Metadata Quality Checks. We know that data quality is important for researchers to find datasets and to have trust in them to use them for another analysis. For every submitted dataset, the metadata is run through a quality check to increase the completeness of submitted metadata records. These checks are seen by the submitter as well as are available to those that view the data, which helps to increase knowledge of how complete their metadata is before submission. That way, the metadata that is uploaded to the Arctic Data Center is as complete as possible, and close to following the guideline of being understandable to any reasonable scientist.



1.1.3 Support Services

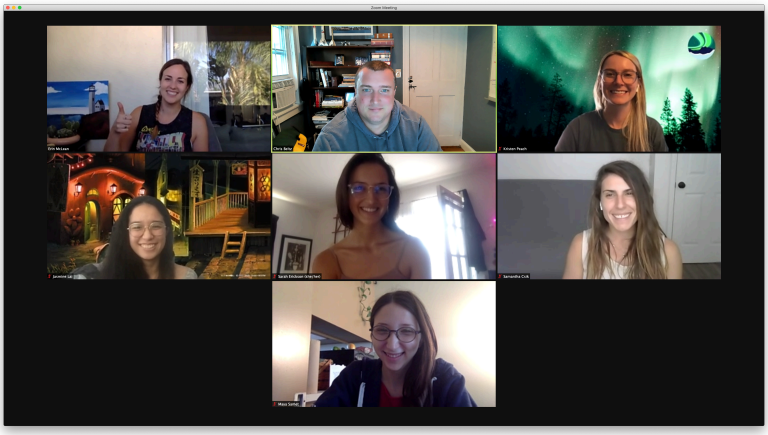
Metadata quality checks are the automatic way that we ensure quality of data in the repository, but the real quality and curation support is done by our curation team. The process by which data gets into the Arctic Data Center is iterative, meaning that our team works with the submitter to ensure good quality and completeness of data. When a submitter submits data, our team gets a notification and beings to evaluate the data for upload. They then go in and format it for input into the catalog, communicating back and forth with the researcher if anything is incomplete or not communicated well. This process can take anywhere from a few days or a few weeks, depending on the size of the dataset and how quickly the researcher gets back to us. Once that process has been completed, the dataset is published with a DOI (digital object identifier).



1.1.4 Training and Outreach

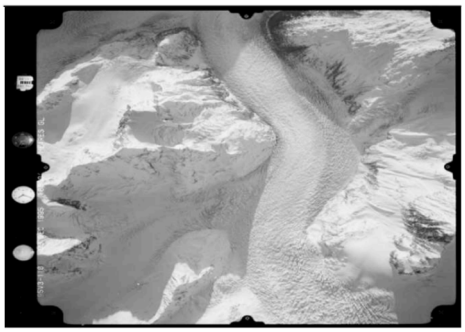
In addition to the tools and support services, we also interact with the community via trainings like this one and outreach events. We run workshops at conferences like the American Geophysical Union, Arctic Science Summit Week and others. We also run an intern and fellows program, and webinars with different organizations. We're invested in helping the Arctic science community learn reproducible techniques, since it facilitates a more open culture of data sharing and reuse.

We strive to keep our fingers on the pulse of what researchers like yourselves are looking for in terms of support. We're active on Twitter to share Arctic updates, data science updates, and specifically Arctic Data Center updates, but we're also happy to feature new papers or successes that you all have had with working with the data. We can also take data science questions if you're running into those in the course of your research, or how to make a quality data management plan. Follow us on Twitter and interact with us – we love to be involved in your research as it's happening as well as after it's completed.



1.1.5 Data Rescue

We also run data rescue operations. We digitiazed Autin Post's collection of glacier photos that were taken from 1964 to 1997. There were 100,000+ files and almost 5 TB of data to ingest, and we reconstructed flight paths, digitized the images of his notes, and documented image metadata, including the camera specifications.



Meares Glacier, Prince William Sound, AK
61.187448, -147.457573, taken from 18,000'
December 3, 1995, Roll 3, Frame 110
doi:10.18739/A2FF6Z (NAGAP_95V3_110.jpg)

1.1.6 Who Must Submit

Projects that have to submit their data include all Arctic Research Opportunities through the NSF Office of Polar Programs. That data has to be uploaded within two years of collection. The Arctic Observing Network has a shorter timeline – their data products must be uploaded within 6 months of collection. Additionally, we have social science data, though that data often has special exceptions due to sensitive human subjects data. At the very least, the metadata has to be deposited with us.

Arctic Research Opportunities (ARC)

- Complete metadata and all appropriate data and derived products
- Within 2 years of collection or before the end of the award, whichever comes first

ARC Arctic Observation Network (AON)

- Complete metadata and all data
- Real-time data made public immediately
- Within 6 months of collection

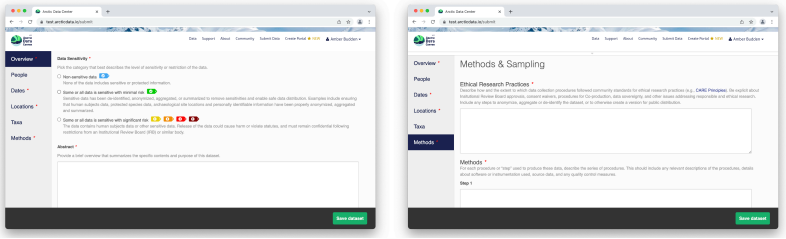
Arctic Social Sciences Program (ASSP)

- NSF policies include special exceptions for ASSP and other awards that contain sensitive data

- Human subjects, governed by an Institutional Review Board, ethically or legally sensitive, at risk of decontextualization
- Metadata record that documents non-sensitive aspects of the project and data
  - Title, Contact information, Abstract, Methods

For more complete information see our "Who Must Submit" webpage

Recognizing the importance of sensitive data handling and of ethical treatment of all data, the Arctic Data Center submission system provides the opportunity for researchers to document the ethical treatment of data and how collection is aligned with community principles (such as the CARE principles). Submitters may also tag the metadata according to community develop data sensitivity tags. We will go over these features in more detail shortly.

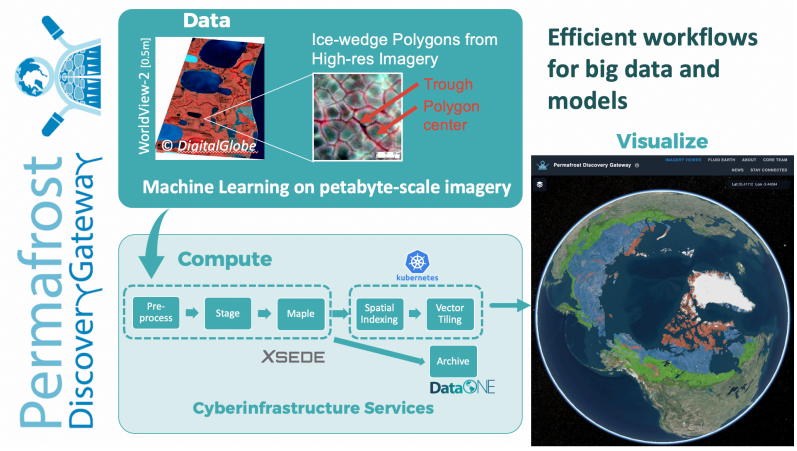


1.1.7 Supporting data reuse

As the Arctic Data Center has grown in size, we have envisioned new capabilities to support efficient reuse of these valuable data. We are developing new tools both within the Arctic Data Center, and through collaborative projects with other researchers in the community. In house, some of the new features we plan to support include:

- Efficient submission and access to multi-Terabyte datasets
- Data Quality assessment services
- Automated workflows for building derived data products

As a concrete example, we are collaborating with the Permafrost Discovery Gateway to build new capacity for computing and visualization at sub-meter spatial resolution across the global Arctic. The services we are building support workflows for pre-processing massive image data collections to prepare them for modeling, as well as automating machine learning and other computations across large data on distributed compute clusters, and finally post-processing the results to create viable pathways for results mapping and visualization at global scales.



1.1.8 Summary

All the above informtion can be found on our website or if you need help, ask our support team at support@arcticdata.io or tweet us @arcticdatactr!

6,958 Datasets
71.4 TB
2,584 Creators
16,162 Users

1.2 Scalable Computing Topics

The overall objective of this course is to facilitate effective analysis and modeling of the massive data resources that we curate for the Arctic research community. We assume a baseline proficiency, and plan to build on that to better support time-consuming computations with multi-terabyte datasets.

Activities will include:

- *Technical tutorials* with a lot of hands-on time in Python
- *Lectures and discussions* on core concepts for scalable computing
- *Semi-structured group projects* to practice and cement key skills

Key topics will include:

- Arctic Data Center services and tools
- Overview and review of core computing concepts
- Fundamentals of concurrent programming
  - concurrent.futures
  - Parsl
  - Dask
- Fundamentals of working with big data and imagery
- Good practices in research software design for efficiency, reproducibility, and reuse
- Fundamentals of cloud computing
  - Docker, containers, reproducibility, and more

As this is a lot to cover in a short survey course, we will break it down to fundamentals daily, and provide ample time for hands-on practice.

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 connections
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, 128.111.85.28 (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 is also important to understand when provisioning local execution resources (e.g. we'll practice this 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 system 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 changes 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.

> **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-examples`** 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. 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 `02-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 `02-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/02-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 mybash.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 VS Code
- 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.

To open the `scalable-computing-examples` workspace, select File > Open Workspace from File. Then navigate to and select `~/scalable-computing-examples/scalable-computing-examples.code-workspace`. If you cannot find the `scalable-computing-examples` directory, in a terminal you can use the `ls` command to list out the contents of your home directory and ensure that you have a clone of the respository. If not, follow the instructions in Exercise 3 in the Remote Computing session).

Once connected and in the workslace, use the `pwd` command in the terminal to make sure you're in your project directory (`/home/yourusername/scalable-computing-examples`).

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). That library is going to sit alongside every other python library you have ever installed (probably a lot of them) in your `site-packages` directory. This

might work okay, until you start a new project, have no idea what version of `pandas` you had installed, realize that version conflicts with a new library that you need, and then instead of writing code you are spending days trying to untangle the sphaghetti mess of your python install, libraries and their versions. Sound familiar? 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 specify 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. Go ahead and save this notebook at the top level of your `scalable-computing-examples` directory.

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 urllib
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 `urllib.request.urlretrieve` 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

        msg = urllib.request.urlretrieve(url, 'data/discharge_timeseries.csv')
```

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.

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

```
clean_df = skimpy.clean_columns(df)
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         object        1

                                        number

          missing   complet   mean     sd      p0       p25     p75    p100   hist
                    e rate

total_p         0         1      9.9    0.12      9.6     9.8     10     10
ressure
_m
air_pre         0         1      9.6    0.06      9.5     9.6    9.7    9.7
ssure_m
stage_m         0         1     0.28    0.12   0.0005    0.17   0.37   0.56
                                                    6
dischar         0         1     0.22    0.19   4.7e-0   0.055   0.35   0.96
ge_m_3_                                             8
s
tempera         8         1   -0.034   0.053     -0.1    -0.1      0    0.2
ture_de
grees_

                     End
```

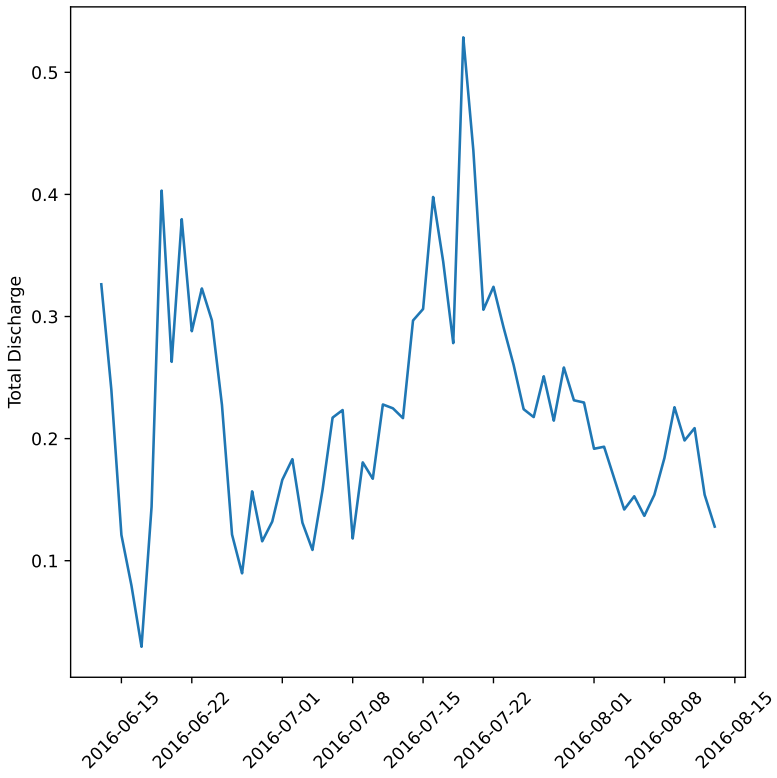
```
clean_df['date'] = pd.to_datetime(clean_df['date'])
skimpy.skim(clean_df)
```

```
                  skimpy summary
       Data Summary                Data Types

dataframe           Values    Column Type   Count

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

                                      number

          missing   complet   mean     sd      p0       p25     p75    p100   hist
                    e rate

total_p         0         1      9.9    0.12      9.6     9.8     10     10
ressure
_m
air_pre         0         1      9.6    0.06      9.5     9.6    9.7    9.7
ssure_m
stage_m         0         1     0.28    0.12   0.0005    0.17   0.37   0.56
                                                    6
dischar         0         1     0.22    0.19   4.7e-0   0.055   0.35   0.96
ge_m_3_                                             8
s
tempera         8         1   -0.034   0.053     -0.1    -0.1      0    0.2
ture_de
grees_

                                    datetime

        missing    complete rate     first         last                    frequency

date          0                1   2016-06-13     2016-08-13 23:55:00    5T

                    End
```

```
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.

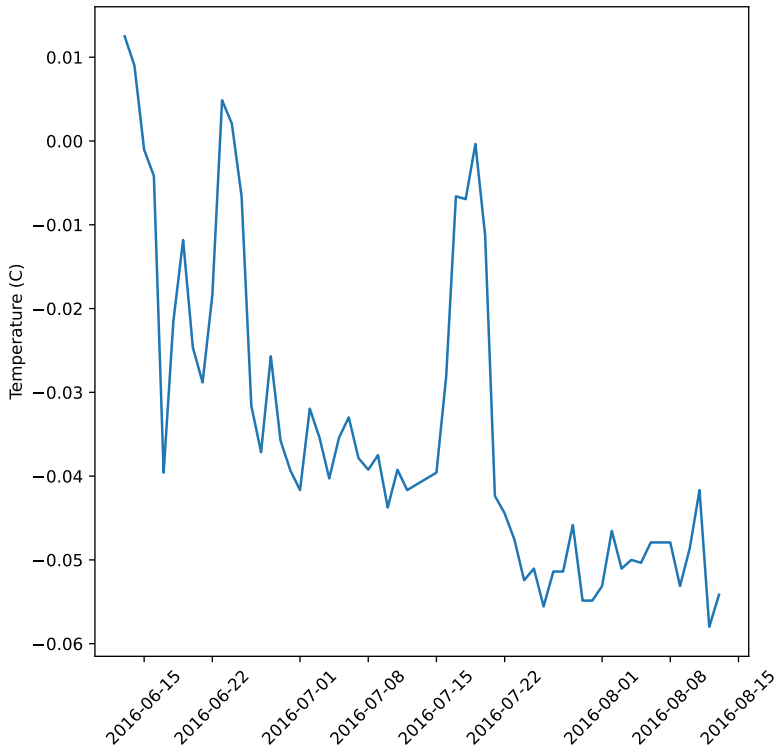
- create `myplot.py`

```
import matplotlib.pyplot as plt

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`)
- add libraries
- replace old plot method with new function

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



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

3.7 Summary

In this lesson we learned all about virtual environments, how to use them, and why. We got our environments set up for the course, did a brief python syntax review, and then jumped into Jupyter notebooks, pandas, and writing functions.

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.

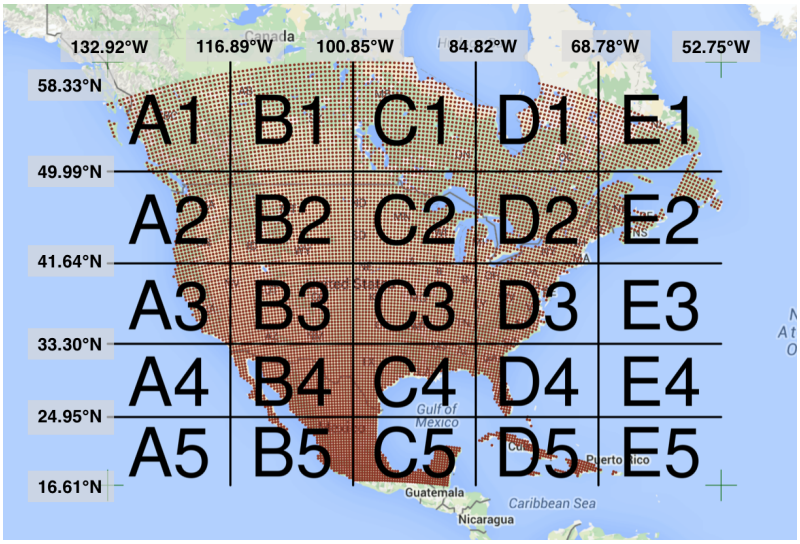


Figure 4.1: Levy et al. 2016. doi:10.5063/F1Z899CZ

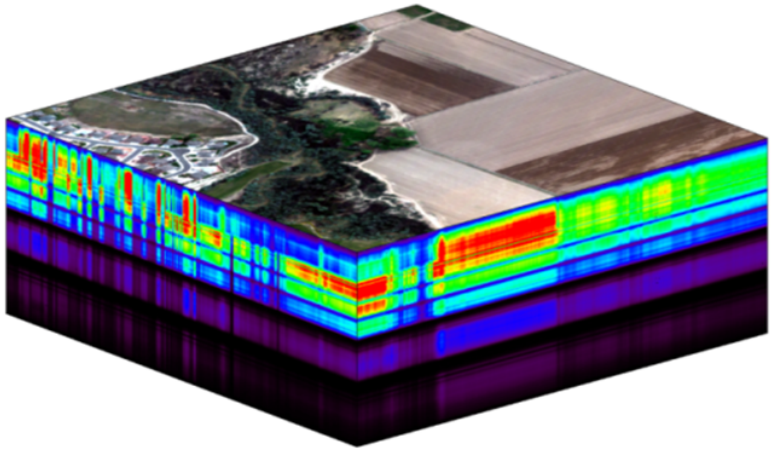


Figure 4.2: NEON Data Cube

4.2 Why parallelism?

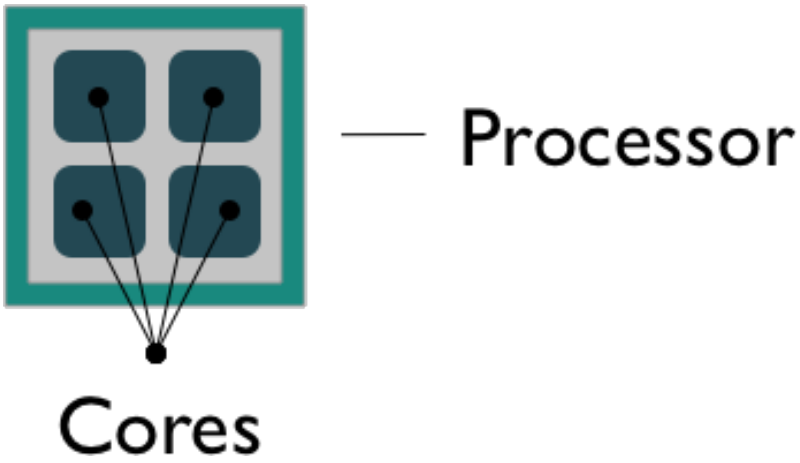
Much R code runs fast and fine on a single processor. But at times, computations can be:

- **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 set of those processors. Plus, these machines also have large amounts of memory to avoid **memory-bound** computing jobs.

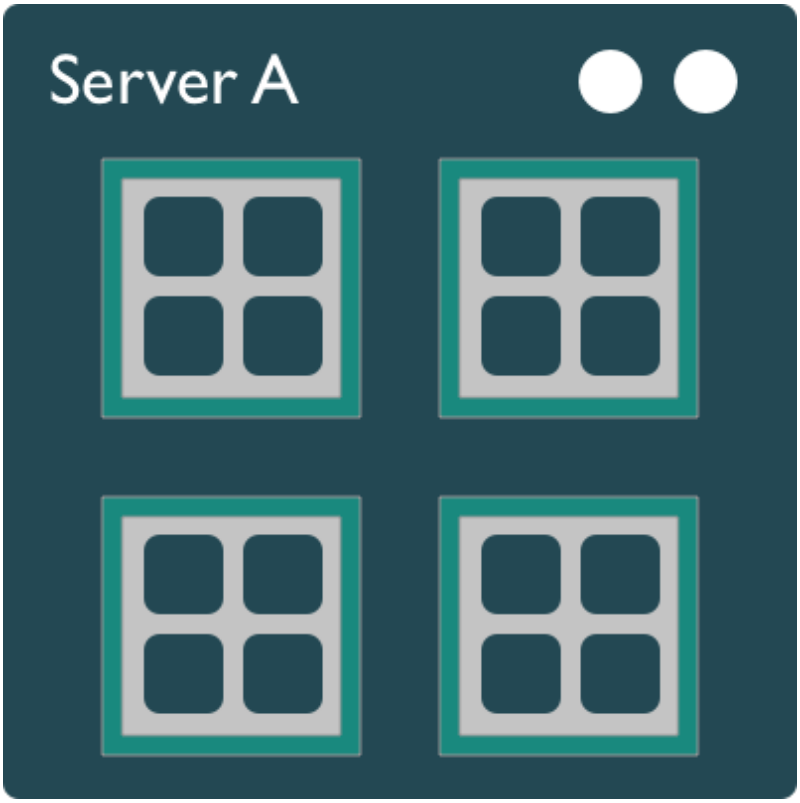
4.3 Processors (CPUs), Cores, and Threads

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, each of which in turn can contain multiple cores. These processors and cores are available to perform computations. But, just what's the difference between processors and cores? A computer with one processor may still have 4 cores (quad-core), allowing 4 (or possibly more) computations to be executed at the same time.



- **Microprocessor**: an integrated circuit that contains the data processing logic and control for a computer.
- **Multi-core processor**: a microprocessor containing multiple processing units (cores) on a single integrated circuit. Each core in a multi-core processor can execute program instructions at the same time.
- **Process**: an instance of a computer program (including instructions, memory, and other resources) that is executed on a microprocessor.
- **Thread**: a thread of execution is the smallest sequence of program instructions that can be executed independently, and is typically a component of a process. The threads in a process can be executed concurrently and typically share the same memory space. They are faster to create than a process.
- **Cluster**: a set of multiple, physically distinct computing systems, each with its own microprocessors, memory, and storage resources, connected together by a (fast) network that allows the nodes to be viewed as a single system.

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, many languages only utilized one processor, which makes them single-threaded. Which is a shame, because the 2019 MacBook Pro that I am writing this on is much more powerful than that, and has mutliple cores that would support concurrent execution of multiple threads:

```
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.

Finally, 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 Parallel processing in the shell

Shell programming helps massively speed up data management tasks, and even more so with simple use of the GNU `parallel` utility to execute bash commands in parallel. In its simplest form, this can be used to speed up common file operations, such as renaming, compression, decompression, and file transfer. Let's look at a common example – calculating checksums to verify file integrity for data files. Calculating a hash checksum using the `shasum` command can be time consuming, especially when you have a lot of large files to work on. But it is a classic processor-limited task, and one that can be massively faster using `parallel`.

```
$ for fn in `ls *.gpkg`; do shasum -a 256 ${fn}; done

real    35.081s
user    32.745s
system  2.336s

$ ls *.gpkg | parallel "shasum -a 256 {}"

real    2.97s
user    37.16s
system  2.70s
```

The first invocation takes *35 seconds* to execute the tasks one at a time serially, while the second version only takes :tada: *3 seconds* :tada: to do the same tasks. Note that the computational time spent in `user` and `system` processing is about the same, with the major difference being that the user-space tasks were conducted on multiple cores in parallel, resulting in more than 10x faster performance, Using `htop`, you can see processor cores spiking in usage when the command is run:

4.5 Modes of parallelization

Several different approaches can be taken to structuring a computer program to take advantage of the hardware capabilities of

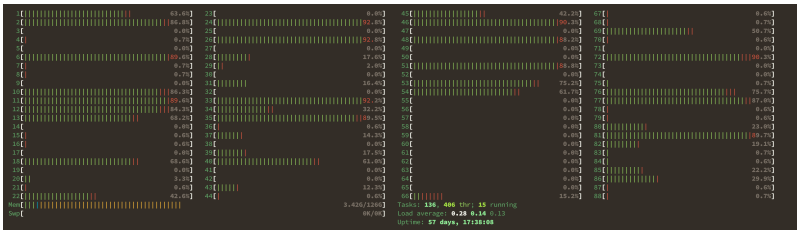


Figure 4.3: Processor usage for parallel tasks.

multi-core processors. In the typical, and simplest, case, each task in a computation is executed serially in order of first to last. The total computation time is the sum of the time of all of the subtasks that are executed. In the next figure, a single core of the processor is used to sequentially execute each of the five tasks, with time flowing from left to right.

In comparison, the middle panel shows two approaches to parallelization on a single computer. With **multi-threaded** execution, a separate thread of execution is created for each of the 5 tasks, and these are executed concurrently on 5 of the cores of the processor. All of the threads are in the same process and share the same memory and resources, so one must take care that they do not interfere with each other.

With **multi-process** execution, a separate process is created for each of the 5 tasks, and these are executed concurrently on the cores of the processor. The difference is that each process has it's own copy of the program memory, and changes are merged when each child process completes. Because each child process must be created and resources for that process must be marshaled and unmarshaled, there is more overhead in creating a process than a thread.

Finally, **cluster parallel** execution is shown in the last panel, in which a cluster with multiple computers is used to execute multiple processes for each task. Again, there is a setup task associated with creating and mashaling resources for the task, which now includes the overhead of moving data from one machine to the others in the cluster over the network. This further increases the cost of creating and executing multiple processes, but can be highly advantageous when accessing exceedingly large numbers of processing cores on clusters.

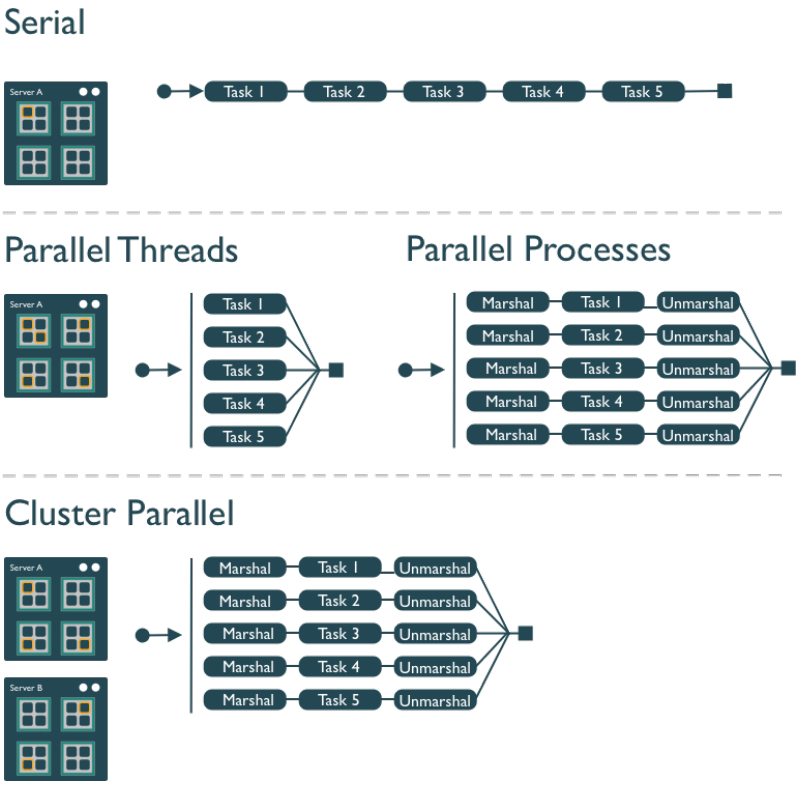


Figure 4.4: Serial and parallel execution of tasks using threads and processes.

The key to performance gains is to ensure that the overhead associated with creating new threads or processes is small relative to the time it takes to perform a task. Somewhat unintuitively, when the setup overhead time exceeds the task time, parallel execution will likely be slower than serial.

4.6 Task parallelization in Python

Python also provides a number of easy to use packages for concurrent processing. We will review two of these, `concurrent.futures` and `parsl`, to show just how easy it can be to parallelize your programs. `concurrent.futures` is built right into the python3 release, and is a great starting point for learning concurrency.

We're going to start with a task that is a little expensive to compute, and define it in a function. All this `task(x)` function does is to use numpy to create a fairly large range of numbers, and then sum them.

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

We can start by executing this task function serially ten times with varying inputs. In this case, we create a function `run_serial` that takes a list of inputs to be run, and it calls the `task` function for each of those inputs. The `@timethis` decorator is a simple way to wrap the function with timing code so that we can see how long it takes to execute.

```
import numpy as np

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

run_serial(np.arange(10))
```

```
run_serial: 83291.87178611755 ms


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

In this case, it takes around *25 seconds* to execute 10 tasks, depending on what else is happening on the machine and network.

So, can we make this faster using a multi-threaded parallel process? Let's try with `concurrent.futures`. The main concept in this package is one of a `future`, which is a structure which represents the value that will be created in a computation in the future when the function completes execution. With `concurrent.futures`, tasks are scheduled and do not block while they await their turn to be executed. Instead, threads are created and executed *asynchronously*, meaning that the function returns it's `future` potentially before the thread has actually been executed. Using this approach, the user schedules a series of tasks to be executed asynchronously, and keeps track of the futures for each task. When the future indicates that the execution has been completed, we can then retrieve the result of the computation.

In practice this is a simple change from our serial implementation. We will use the `ThreadPoolExecutor` to create a pool of workers that are available to process tasks. Each worker is set up in its own thread, so it can execute in parallel with other workers. After setting up the pool of workers, we use concurrent.futures `map()` to schedule each task from our `task_list` (in this case, an input value from 1 to 10) to run on one of the workers. As for all map() implementations, we are asking for each value in `task_list` to be executed in the `task` function

we defined above, but in this case it will be executed using one of the workers from the `executor` that we created.

```
from concurrent.futures import ThreadPoolExecutor

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

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

```
run_threaded: 4440.109491348267 ms
[0,
 4999999950000000,
 19999999900000000,
 44999999850000000,
 79999999800000000,
 124999999750000000,
 179999999700000000,
 244999999650000000,
 319999999600000000,
 404999999550000000]
```

This execution took about :tada: *4 seconds* :tada:, which is about 6.25x faster than serial. Congratulations, you wrote your a multi-threaded python program!

4.7 Exercise: Parallel downloads

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`. We're going to see an example here where 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.

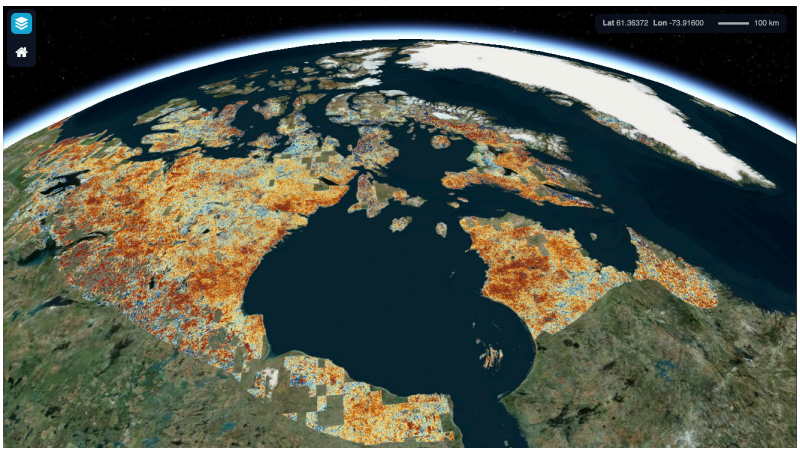


Figure 4.5: Webb surface water index data

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 their identifier. To make things easier, we've already provided a data frame with the names and identifiers of each file that could be downloaded.

```
/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

4.7.1 Serial

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 pleasingly parallel and 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_serial(table):
    results = [download_file(row) for row in table.iterrows()]
    return results

results = download_serial(file_table[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_serial: 101647.98069000244 ms
['SWI_2007.tif', 'SWI_2019.tif', 'SWI_2021.tif', 'SWI_2020.tif', 'SWI_2001.tif']
```

In this code, we have one function (`download_file`) that downloads a single data file and saves it to disk. It is called iteratively from the function `download_serial`. The serial execution takes about *20-25 seconds*, but can vary considerably based on network traffic and other factors.

The issue with this loop is that we execute each download task 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.

4.7.2 Multi-threaded with `concurrent.futures`

In this case, we'll use the same `download_file` function from before, but let's switch up and create a `download_threaded()` function to use `concurrent.futures` with a `ThreadPoolExecutor` just as we did earlier.

```
from concurrent.futures import ThreadPoolExecutor

@timethis
def download_threaded(table):
    with ThreadPoolExecutor(max_workers=15) as executor:
        results = executor.map(download_file, table.iterrows(), timeout=60)
        return results

results = download_threaded(file_table[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_threaded: 22690.69242477417 ms
SWI_2007.tif
SWI_2019.tif
SWI_2021.tif
SWI_2020.tif
SWI_2001.tif
```

Note how the "Downloading..." messages were printed immediately, but then it still took over 20 seconds to download the 5 files. This could be for several reasons, including that one of the files alone took that long (e.g., due to network congestion), or that there was a bottleneck in writing the files to disk (i.e., we could have been disk I/O limited). Or maybe the multithreaded executor pool didn't do a good job parallelizing the tasks. The trick is figuring out why you did or didn't get a speedup when parallelizing. So, let's try this another way, using a **multi-processing** approach, rather than multi-threading.

4.7.3 Multi-process with `concurrent.futures`

You'll remember from earlier that you can execute tasks concurrently by creating multiple threads within one process (multi-threaded), or by creating and executing muliple processes. The latter creates more independence, as each of the executing tasks has their own memory and process space, but it also takes longer to set up. With `concurrent.futures`, we can switch to a multi-process pool by using a `ProcessPoolExecutor`, analogously to how we used `ThreadPoolExecutor` previously. So, some simple changes, and we're running multiple processes.

```
from concurrent.futures import ProcessPoolExecutor

@timethis
def download_process(table):
    with ProcessPoolExecutor(max_workers=15) as executor:
        results = executor.map(download_file, table.iterrows(), timeout=60)
        return results

results = download_process(file_table[0:5])
for result in results:
    print(result)
```

```
Downloading: SWI_2021.tif

Downloading: SWI_2019.tif

Downloading: SWI_2020.tif

Downloading: SWI_2001.tif

Downloading: SWI_2007.tif
```

```
download_process: 16608.938455581665 ms
SWI_2007.tif
SWI_2019.tif
SWI_2021.tif
SWI_2020.tif
SWI_2001.tif
```

Again, the output messages print almost immediately, but then later the processes finish and report that it took between *10 to 15 seconds* to run. Your mileage may vary. When I increase the number of files being downloaded to 10 or even to 20, I notice it is actually about the same, around *10-15 seconds*. So, part of our time now is the overhead of setting up multiple processes. But once we have that infrastructure in place, we can make effective euse of the pool of processes to handle many downloads.

4.8 Parallel processing with `parsl`

`concurrent.futures` is great and powerful, but it has its limits. Particularly as you try to scale up into the thousands of concurrent tasks, other libraries like Parsl (docs), Dask, Ray, and others come into play. They all have their strengths, but Parsl makes it particularly easy to build parallel workflows out of existing python code through it's use of decorators on existing python functions.

In addition, Parsl supports a lot of different kinds of providers, allowing the same python code to be easily run multi-threaded using a `ThreadPoolExecutor` and via multi-processing on many different cluster computing platforms using the `HighThroughputExecutor`. For example, Parsl includes providers supporting local execution, and on Slurm, Condor, Kubernetes, AWS, and other platforms. And Parsl handles data staging as well across these varied environments, making sure the data is in the right place when it's needed for computations.

Similarly to before, we start by configuring an executor in parsl, and loading it. We'll use multiprocessing by configuring the `HighThroughputExecutor` to use our local resources as a cluster, and we'll activate our virtual environment to be sure we're executing in a consistent environment.

```
# 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)
```

```
<parsl.dataflow.dflow.DataFlowKernel at 0x7f3029171d30>
```

We now have a live parsl executor (`htex_local`) that is waiting to execute processes. We tell it to execute processes by annotating functions with decorators that indicate which tasks should be parallelized. Parsl then handles the scheduling and execution of those tasks based on the dependencies between them. In the simplest case, we'll decorate our previous function for downloading a file with the `@python_app` decorator, which tells parsl that any function calls with this function should be run on the default executor (in this case, `htex_local`).

```
# Decorators seem to be ignored as the first line of a cell, so print something first
print("Create decorated function")

@python_app
def download_file_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
```

```
Create decorated function
```

Now we just write regular python code that calls that function, and parsl handles the scheduling. Parsl app executors return an AppFuture, and we can call the `AppFuture.done()` function to determine when the future result is ready without blocking. Or, we can just block on `AppFuture.result()` which waits for each of the executions to complete and then returns the result.

```
#! eval: true
print("Define our download_futures function")

@timethis
def download_futures(table):
    results = []
    for row in table.iterrows():
        result = download_file_parsl(row)
        print(result)
        results.append(result)
    return(results)

@timethis
def wait_for_futures(table):
    results = download_futures(table)
    done = [app_future.result() for app_future in results]
    print(done)

wait_for_futures(file_table[0:5])
```

```
Define our download_futures function
<AppFuture at 0x7f30290f60a0 state=pending>
<AppFuture at 0x7f3028088700 state=pending>
<AppFuture at 0x7f3028088c40 state=pending>
```

```
<AppFuture at 0x7f302808f1c0 state=pending>
<AppFuture at 0x7f302808f700 state=pending>
download_futures: 89.98394012451172 ms

['SWI_2007.tif', 'SWI_2019.tif', 'SWI_2021.tif', 'SWI_2020.tif', 'SWI_2001.tif']
wait_for_futures: 64893.31364631653 ms
```

When we're done, be sure to clean up and shutdown the `htex_local` executor, or it will continue to persist in your environment and utilize resources. Generally, an executor should be created when setting up your environment, and then it can be used repeatedly for many different tasks.

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

4.9 When to parallelize

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

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

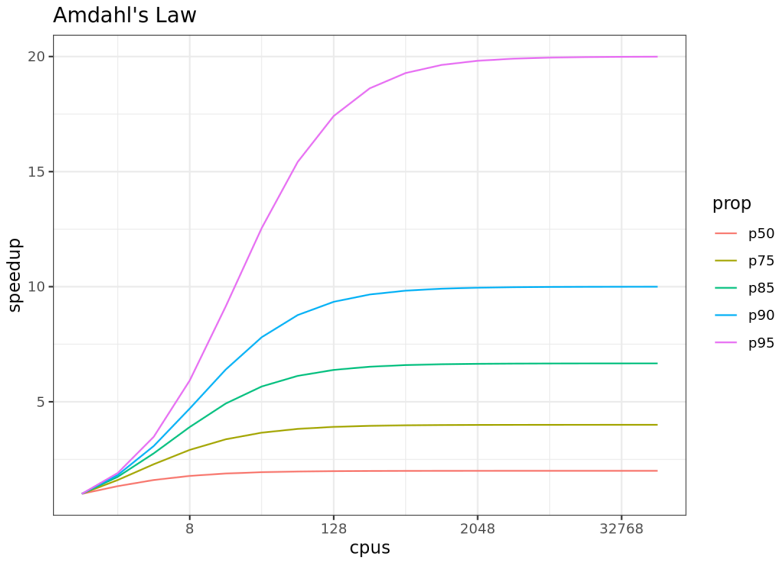


Figure 4.6: Amdahl's Law

So, it is 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.

4.10 Parallel Pitfalls

A set of tasks is considered 'pleasingly parallel' when large portions of the code can be executed indpendently of the other portions and have few or no dependencies on other parts of the execution. This situation is common, and we can frequently execute parallel tasks on independent subsets of our data. Nevertheless, dependencies among different parts of your computation can definitely create bottlenecks and slow down computations. Some of the challenges you may need to work around include:

- **Task dependencies**: occur when one task in the code depends on the results of another task or computation in the code.

- **Race conditions**: ooccur when two tasks execute in parallel, but produce different results based on which task finishes first. Ensuring that results are correct under different timing situations requires careful testing.
- **Deadlocks**: occur when two concurrent tasks block on the output of the other. Deadlocks cause parallel programs to lock up indefinitely, and can be difficult to track down.

Even when tasks exhibit strong dependencies, it is frequently possible to still optimize that code by parallelizing explicit code sections, sometimes bringing other concurrency tools into the mix, such as the Message Passing Interface (MPI). But simply improving the efficiency of pleasingly parallel tasks can be liberating.

4.11 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 sequential loops can be rewritten as functions that are applied to a list of inputs both serially and in parallel to utilize multiple cores to speed up computations. We reviewed the challenges of optimizing code, where one must constantly examine the bottlenecks that arise as we improve cpu-bound, I/O bound,and memory bound computations.

4.12 Further Reading

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

5 Documenting and Publishing Data

- Become familiar with the submission process
- Understand what constitutes as "large data"
  - Know when to reach out to support team for help
- Lean how data & code can be documented and published in open data archives

5.1 Introduction

A data repository is a database infrastructure that collects, manages, and stores data. In addition to the Arctic Data Center, there are many other repositories dedicated to archiving data, code, and creating rich metadata. The Knowledge Network for Biocomplexity (KNB), the Digital Archaeological Record (tDAR), Environmental Data Initiative (EDI), and Zenodo are all examples of dedicated data repositories.

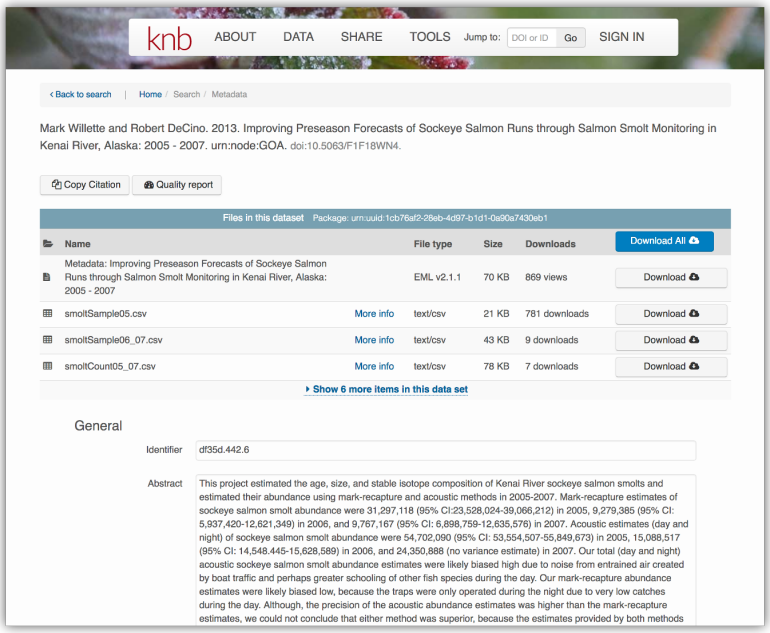
5.2 Metadata

Metadata are documentation describing the content, context, and structure of data to enable future interpretation and reuse of the data. Generally, metadata describe ***who*** collected the data, ***what*** data were collected, ***when*** and ***where*** they were collected, and ***why*** they were collected.

For consistency, metadata are typically structured following metadata content standards such as the Ecological Metadata Language (EML). For example, here's an excerpt of the machine-readable version of the metadata for a sockeye salmon dataset:

```
<?xml version="1.0" encoding="UTF-8"?>
<eml:eml packageId="df35d.442.6" system="knb"
    xmlns:eml="eml://ecoinformatics.org/eml-2.1.1">
    <dataset>
        <title>Improving Preseason Forecasts of Sockeye Salmon Runs through
            Salmon Smolt Monitoring in Kenai River, Alaska: 2005 - 2007</title>
        <creator id="1385594069457">
            <individualName>
                <givenName>Mark</givenName>
                <surName>Willette</surName>
            </individualName>
            <organizationName>Alaska Department of Fish and Game</organizationName>
            <positionName>Fishery Biologist</positionName>
            <address>
                <city>Soldotna</city>
                <administrativeArea>Alaska</administrativeArea>
                <country>USA</country>
            </address>
            <phone phonetype="voice">(907)260-2911</phone>
            <electronicMailAddress>mark.willette@alaska.gov</electronicMailAddress>
        </creator>
        ...
    </dataset>
</eml:eml>
```

Alternatively, the same metadata document can be converted to HTML format and displayed in a more readable form on the web:

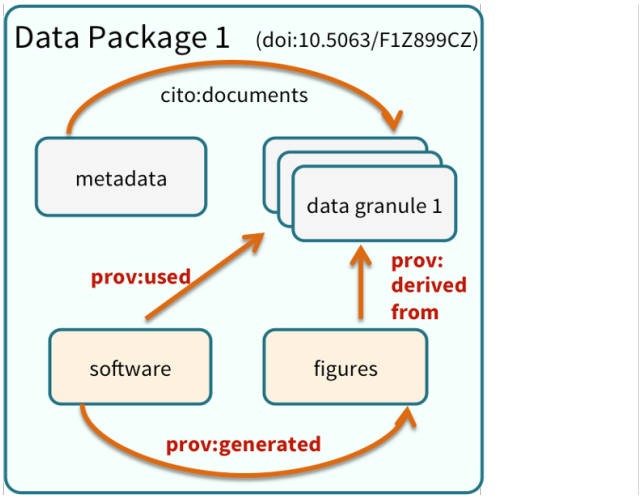


As you can see from the picture above, users can download either the whole dataset or its individual components. This makes the dataset and its associated data resuable.

Additionally, the repository tracks how many times each file has been downloaded, which gives great feedback to researchers on the activity for their published data.

5.3 Data Package Structure

Note that the dataset above lists a collection of files that are contained within the dataset. We define a data package as a scientifically useful collection of data and metadata that a researcher wants to preserve. Sometimes a data package represents all of the data from a particular experiment, while at other times it might be all of the data from a grant, or on a topic, or associated with a paper. Whatever the extent, we define a data package as having one or more data files, software files, and other scientific products such as graphs and images, all tied together with a descriptive metadata document.



These data repositories all assign a unique identifier to ***every version*** of every data file, similarly to how it works with source code commits in GitHub. Those identifiers usually take one of two forms. A DOI (digital object identifier) identifier is often assigned to the metadata and becomes the publicly citable identifier for the package. Each of the other files gets a global identifier, often a UUID (universally unique identifier) that is globally unique. In the example above, the package can be cited with the DOI `doi:10.5063/F1F18WN4`, and each of the individual files have their own identifiers as well.

5.4 Archiving Data: The Large Data Perspective

There are two components to any data package archived with the Arctic Data Center: the metadata & the data themselves. Data can be images, plain text documents, tabular data, spatial data, scripts used to analyze the data, a readme file, and more. To the best of your ability, please make sure that the data uploaded are in an open format, rather than proprietary format. We strongly recommend using open, self-documenting binary formats for large data archival. NetCDF, HDF, .mat (v7.3) and Parquet files are all examples of "self-documenting" files. In the case of a NetCDF file, users can input the attribute name, attribute description, missing value codes, units, and more into

the file itself. When these data are well-documented within themselves, it can save the time when users submit their data to us, since the documentation for variable level information is already mostly complete. We'll discuss NetCDF and metadata more in Session 8. For geospatial data, we recommend using geotiff for raster files, and geopackage files for vector files.

This section provides an overview of some highlights within the data submission process, and will specifically address issues related to datasets with large amounts of data, whether that be in number of files or cumulative file size.

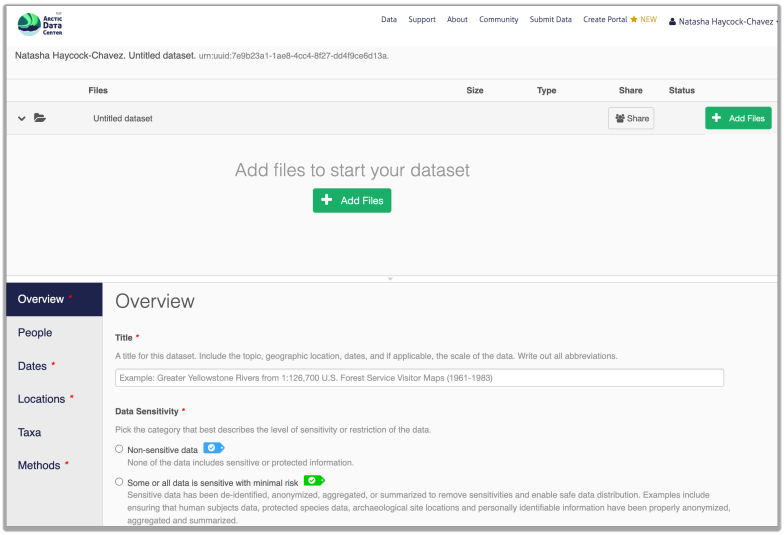
First we'll go over the metadata submission; then learn how to upload the data using a secure File Transfer Protocol; and finally how to add attribute information to the data.

5.4.1 Step 1: The Narrative Metadata Submission

5.4.1.1 ORCiDs

In order to archive data with the Arctic Data Center, you must log in with your ORCID account. If you do not have one, you can create at https://orcid.org/. ORCID is a non-profit organization made up of research institutions, funders, publishers and other stakeholders in the research space. ORCID stands for Open Researcher and Contributor ID. The purpose of ORCID is to give researchers a unique identifier which then helps highlight and give credit to researchers for their work. If you click on someone's ORCID, their work and research contributions will show up (as long as the researcher used ORCID to publish or post their work).

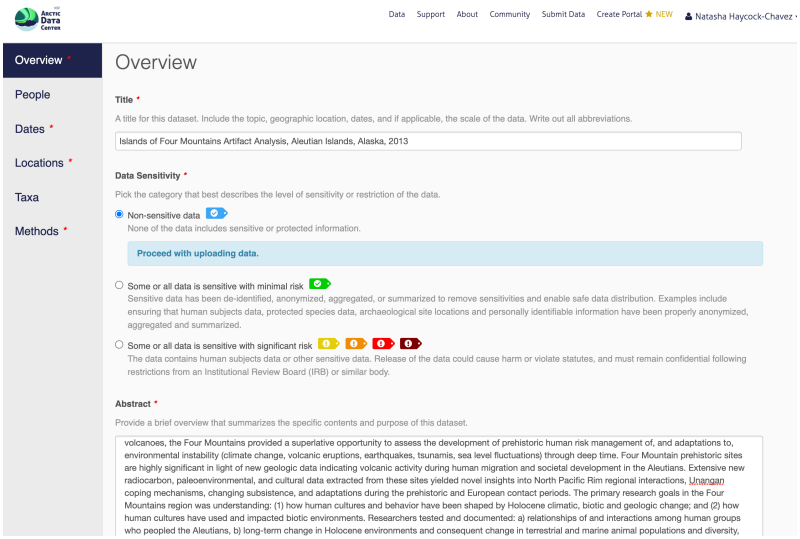
Once you're logged into the Arctic Data Center with your ORCID, you can access the data submission form by clicking "Submit Data" in the navigation bar. For most dataset submissions, you would submit your data and metadata at the same using the "Add Files" buttons seen in the image below. However, when you know you have a large quantity of files or large cumulative file size, you should focus only on submitting metadata through the web form. We'll discuss how to submit large quantities of data in the next section.



5.4.1.2 Overview Section

In the overview section, you will include a descriptive title of your data set, select the appropriate data sensitivity tag, an abstract of the data set, keywords, funding information, and a license.

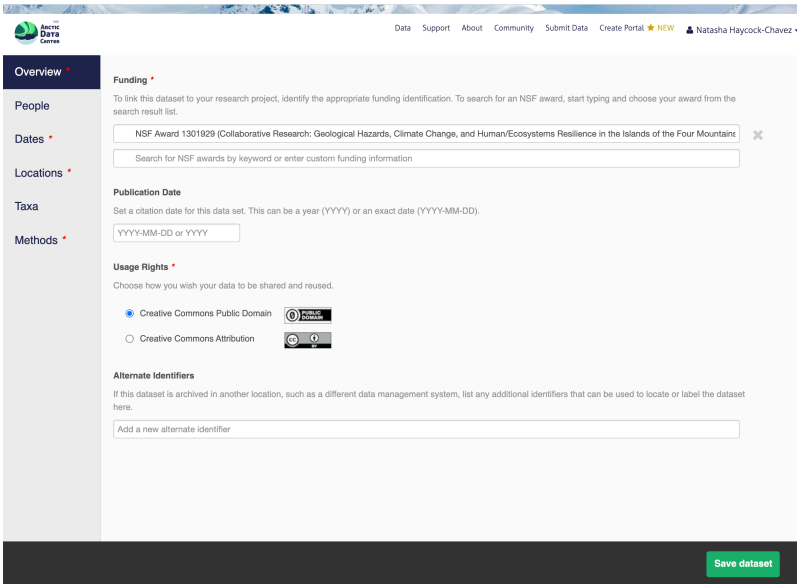
In general, if your data has been anonymized or de-identified in any way, your submission is no longer considered to have "Non-sensitive data". If you have not had to de-identify your data or through an Instituional Review Board process, you should select the "Non-sensitive data" tag. You can find a more in-depth review of the data sensitivity tag in Chapter 12 of our Fundamentals in Data Management coursebook.



You also must enter a funding award number and choose a license. The funding field will search for an NSF award identifier based on words in its title or the number itself. When including funding information not from an NSF award, please make sure to add an award number, title, and organization if possible.

The licensing options are CC-0 and CC-BY, both of which allow your data to be downloaded and re-used by other researchers.

- CC-0 Public Domain Dedication: "…can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission."
- CC-BY: Attribution 4.0 International License: "…free to…copy,…redistribute,…remix, transform, and build upon the material for any purpose, even commercially,…[but] must give appropriate credit, provide a link to the license, and indicate if changes were made."

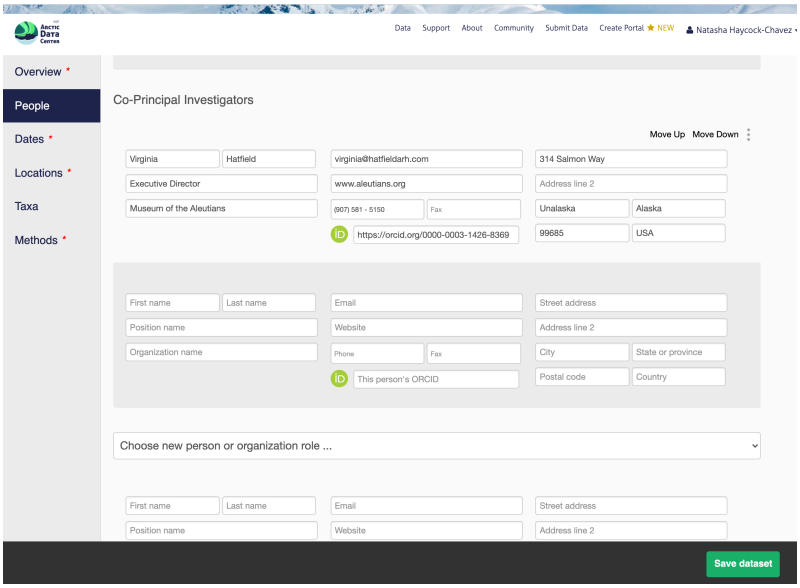


5.4.1.3 People Information

Information about the people associated with the dataset is essential to provide credit through citation and to help people understand who made contributions to the product. Enter information for the following people:

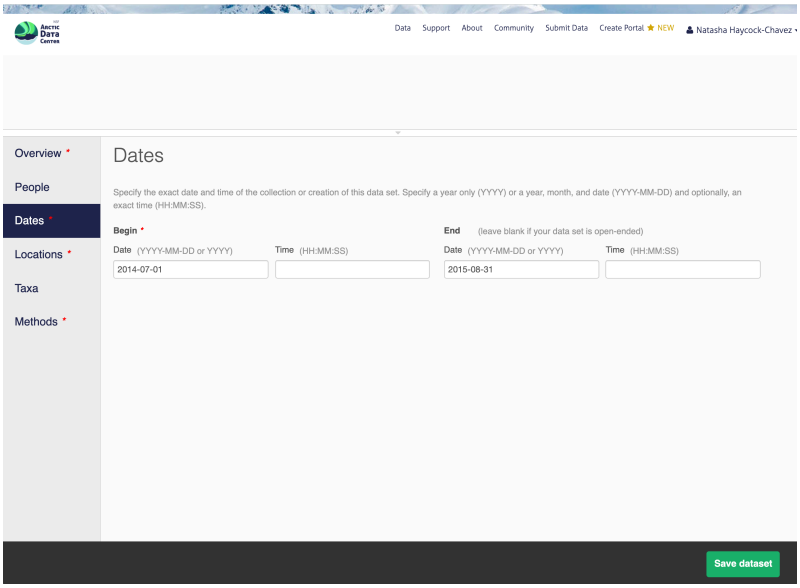
- Creators - **all the people who should be in the citation for the dataset**
- Contacts - one is required, but defaults to the dataset submitter if omitted
- Principal Investigators
- Any others that are relevant

For each, please provide their ORCID identifier, which helps link this dataset to their other scholarly works.



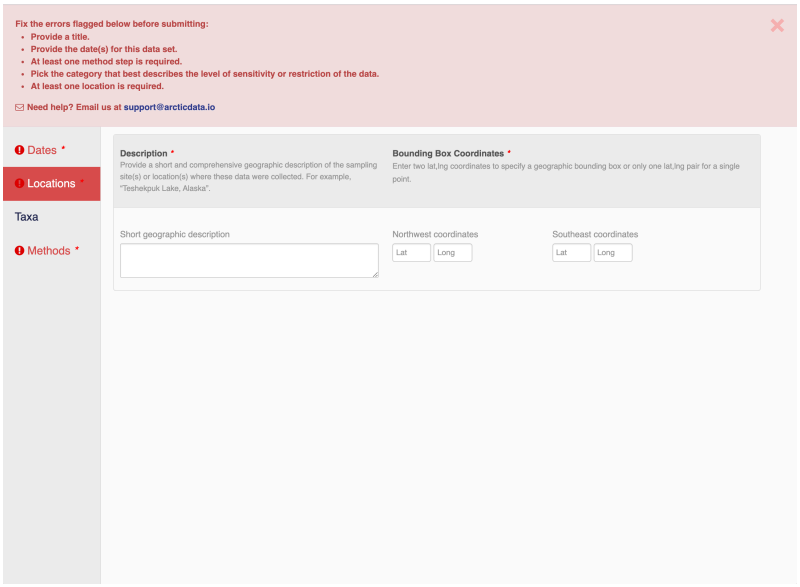
5.4.1.4 Temporal Information

Add the temporal coverage of the data, which represents the time period when data was collected. If your sampling over time was discontinuous and you require multiple date ranges to represent your data, please email us the date ranges and we will add that to the submission on the back-end.



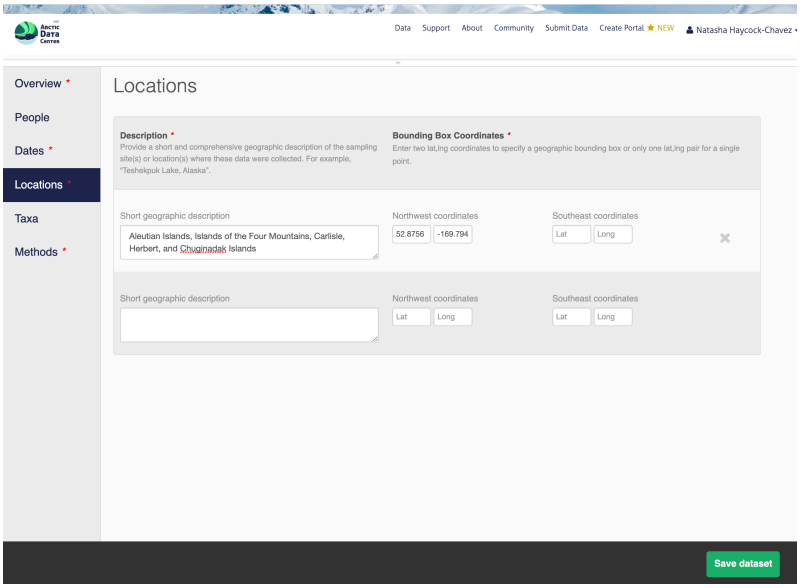
5.4.1.5 Location Information

The geospatial location that the data were collected is critical for discovery and interpretation of the data. Coordinates are entered in decimal degrees, and be sure to use **negative values** for West longitudes. The editor allows you to enter multiple locations, which you should do if you had noncontiguous sampling locations. This is particularly important if your sites are separated by large distances, so that spatial search will be more precise.



Note that, if you miss fields that are required, they will be highlighted in red to draw your attention. In this case, for the description, provide a comma-separated place name, ordered from the local to global:

- Mission Canyon, Santa Barbara, California, USA

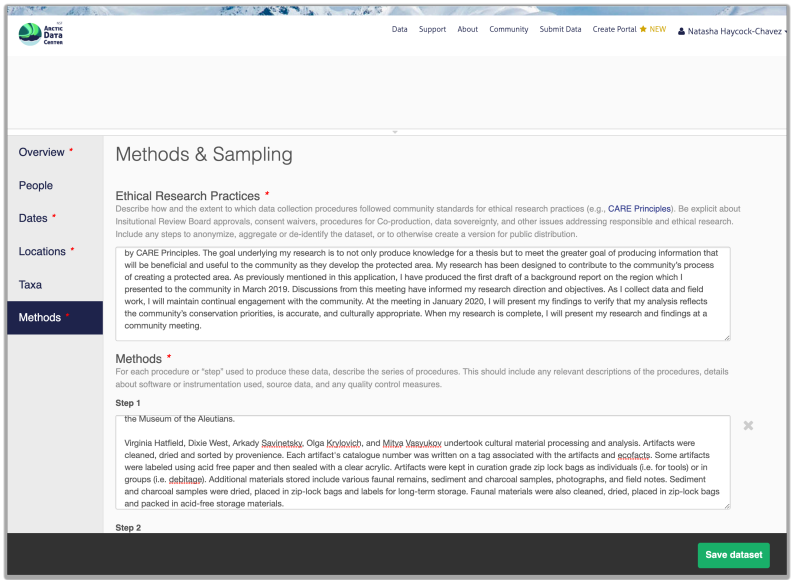


5.4.1.6 Methods

Methods are critical to the accurate interpretation and reuse of your data. The editor allows you to add multiple different methods sections, so that you can include details of sampling methods, experimental design, quality assurance procedures, and/or computational techniques and software. Please be complete with your methods sections, as they are fundamentally important to reuse of the data. Ideally, enough detail should be provided such that a reasonable scientist could interpret the study and data for reuse without needing to consult the researchers or any other resources.

Included in the methods section is a question that asks users about the ethical research practices that may or may not have been considered throughout the research process. You will learn more about this in Chapter 14, and can find more in-depth information on our website's data ethics page.

The ethical research practices response box must be filled out in order to save your dataset. If users feel as though this question is not applicable to their research, we encourage them to discuss why that is rather than simply stating "Not Applicable," or some variation thereof. For example, say a researcher has compiled satellite imagery of weather patterns over the open ocean. Rather than respond with "N/A", a user should instead include something to the effect of "Dataset contains satellite imagery of weather patterns over the Atlantic Ocean. Imagery was downloaded from NASA and NOAA, and does not contain any individual's identifiable information." When in doubt, you can always email the support team at `support@arcticdata.io`.

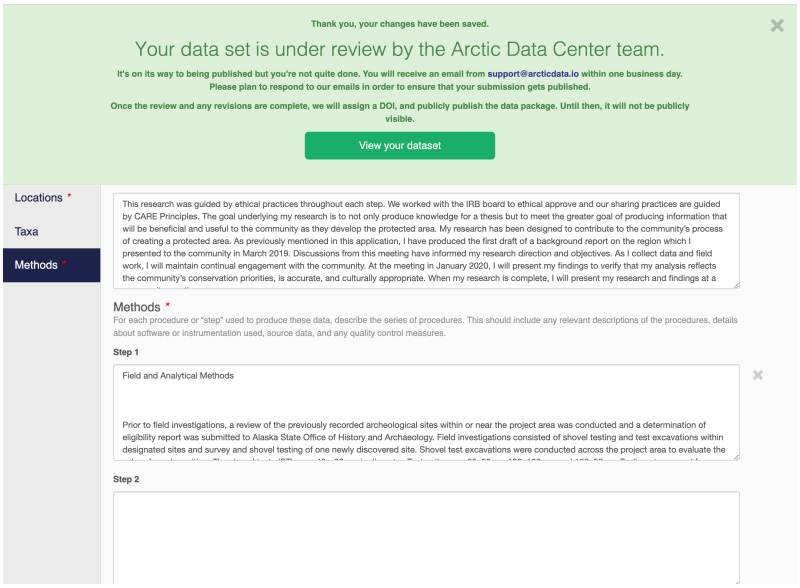


5.4.1.7 Save Metadata Submission

When you're finished editing the narrative metadata, click the *Save Dataset* button at the bottom right of your screen.

If there are errors or missing fields, they will be highlighted with a red banner as seen earlier. Correct those, and then try submitting again. If the save button disappears after making corrections, add a space in the abstract and the save button should reappear. If not, please reach out to the support team for assistance.

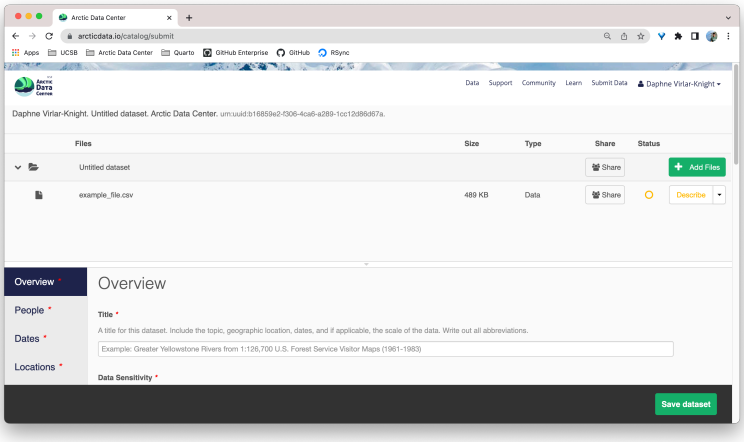
When you are successful, you should see a large green banner with a link to the current dataset view. Click the `X` to close that banner if you want to continue editing metadata.



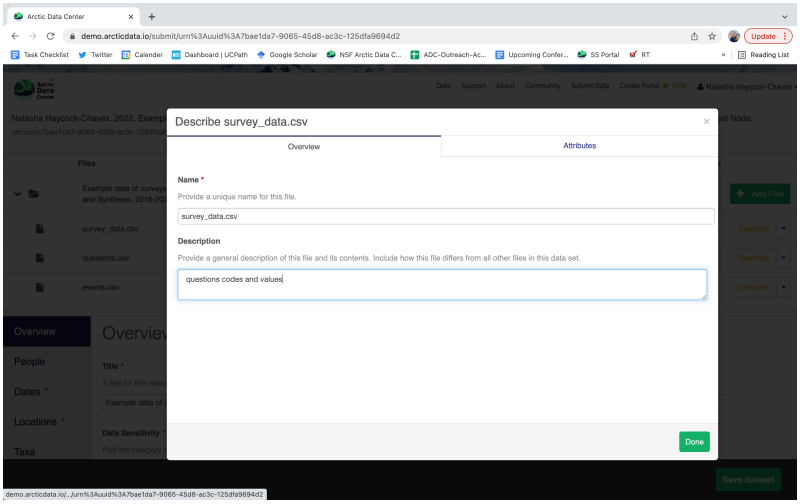
5.4.2 Step 2: Adding File & Variable Level Metadata

The final major section of metadata concerns the structure and content of your data files. Assuming there are many files (and not a few very large ones), it would be unreasonable for users to input file and variable level metadata for each file. When this situation occurs, we encourage users to fill out as much information as possible for each ***unique*** type of file. Once that is completed, usually with some assistance from the Data Team, we will then programmatically carry over the information to other relevant files.

When you're data are associated with your metadata submission, they will appear in the data section at the top of the page when you go to edit your dataset. Choose which file you would like to begin editing by selecting the "Describe" button to the right of the file name.

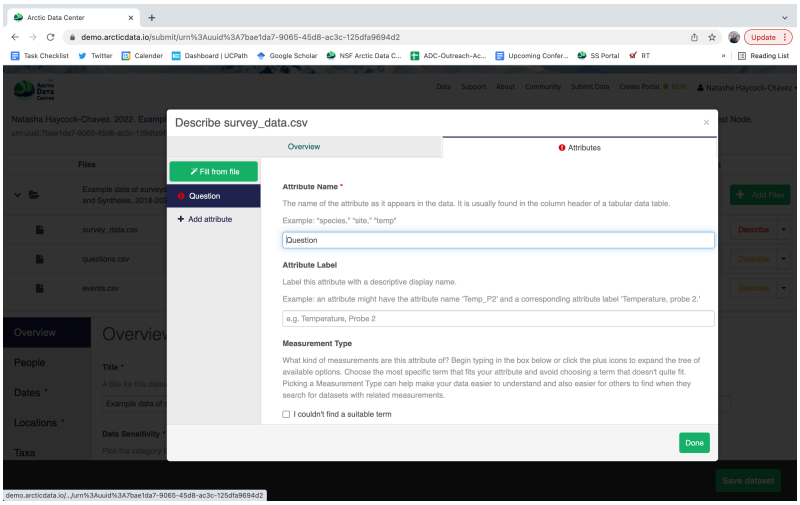


Once there, you will see the following screen. In the Overview section, we recommend ***not*** editing the file name, and instead add a descriptive overview of the file. Once done, click the Attributes tab.

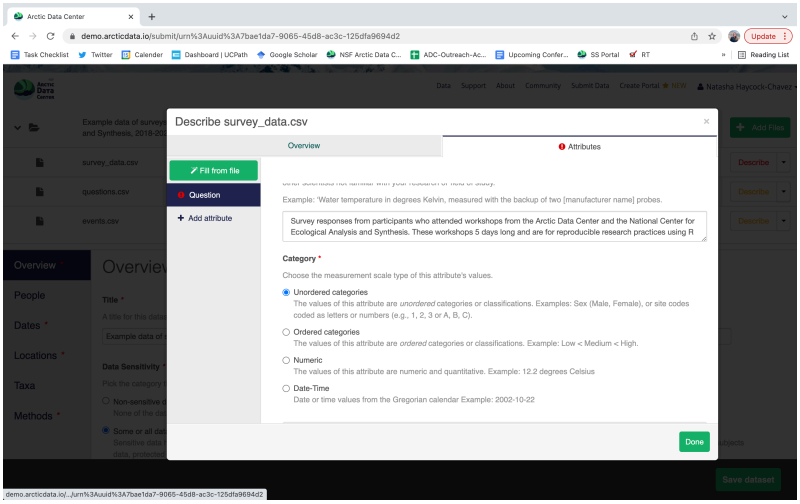


The **Attributes** tab is where you enter variable (aka attribute) information, including:

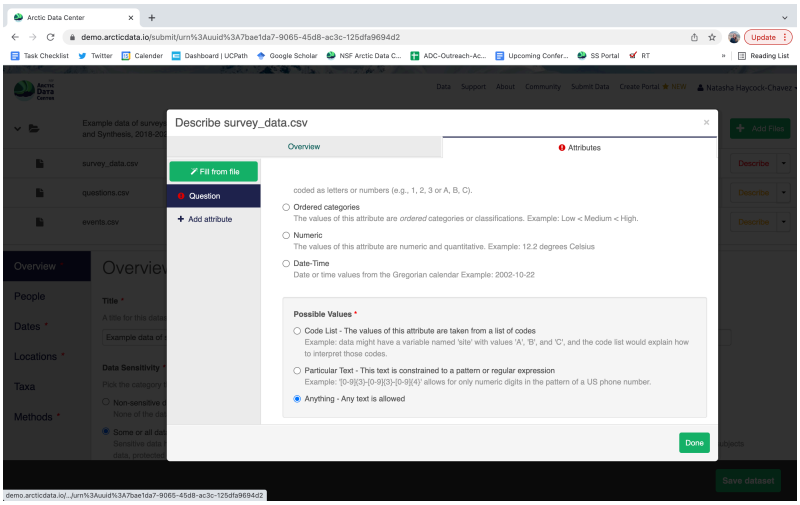
- attribute name (for programs)
- attribute label (for display)



- variable definition (be specific)
- type of measurement

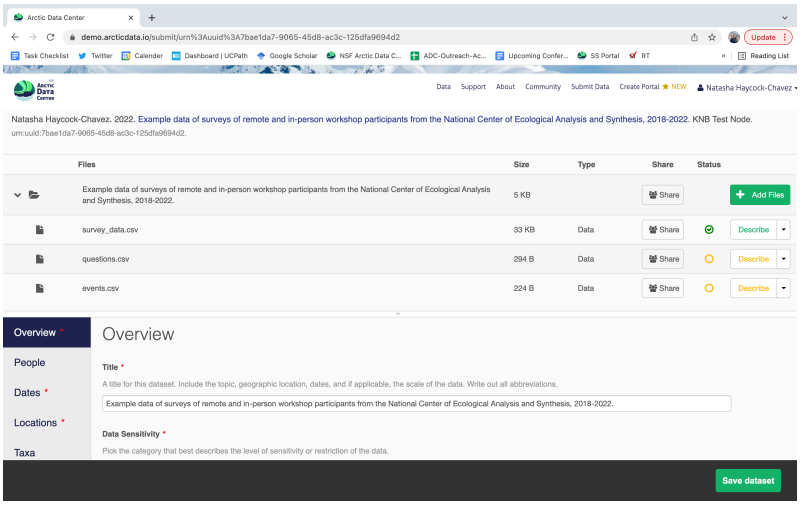


- units & code definitions



Users will need to add these definitions for every variable (column) in the file. When done, click **Done**. Now, the list of data files will show a green checkbox indicating that you have fully described that file's internal structure. Proceed with documenting other unique file types, and then click **Save Dataset** to save all of these changes.

> **Tip**
>
> When dealing with large datasets, the Data Team can help users fill out metadata for similar file types with identical internal structures. An example of this will be discussed in the Best Practices section.



After you get the big green success message, you can visit your dataset and review all of the information that you provided. If you find any errors, simply click **Edit** again to make changes.

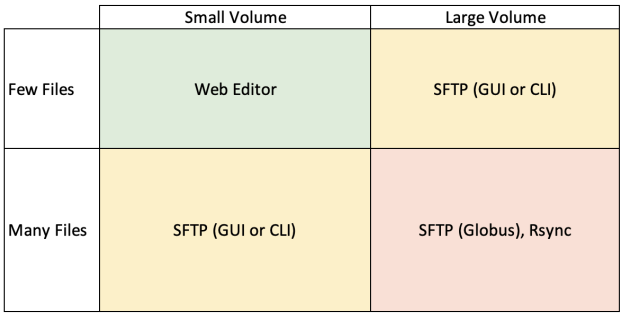
5.4.3 Step 3: Uploading Large Data

In order to submit your large data files to the Arctic Data Center repository, we encourage users to directly upload their data to the Data Team's servers using a secure file transfer protocol (SFTP). There are a number of GUI driven and command line programs out there that all work well. For a GUI program, our team uses and recommends the free program Cyberduck. We will discuss command line programs in Session 18 in more detail, including `rsync` and Globus.

Before we begin, let's answer the following question: Why would a user want to upload their data through a separate process, rather than the web form when they submit their metadata?

Depending on your internet connection, the number of files you have, and the cumulative size of the data, users may experience difficulty uploading their data through the submission form. These difficulties are most often significantly reduced upload speeds and submission malfunctions. As such, it is best for all parties for large quantities of data to be uploaded directly to our server through an FTP.

The second question is, under what circumstances should I consider uploading data directly to the server, versus the webform? Although server uploads are more efficient for large datasets, for smaller datasets it is much more efficient to go through the webform. So, what constitutes a large dataset and what tools should you consider? Here is a helpful diagram:



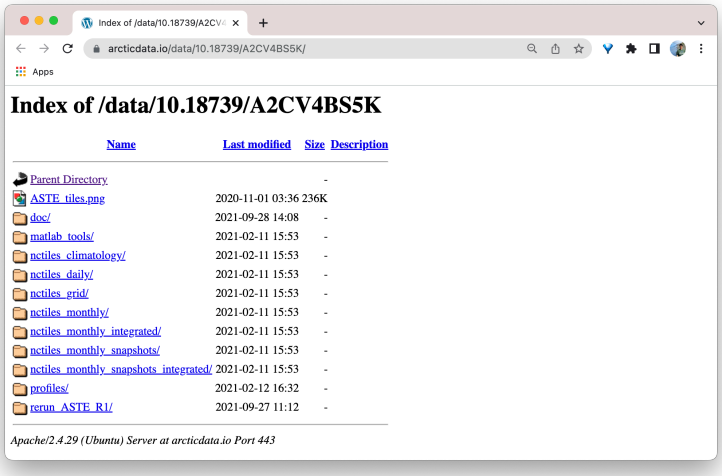
- Web Editor: The web editor is most often used by those with less than a hundred files and a small cumulative file size (0-5 GBs). Overall, this option is best for those who have less than 250 files with a small cumulative file size.
- SFTP: For users that expect to upload more than 250 files and have a medium cumulative file size (10-100 GBs), uploading data to our servers via SFTP is the recommended method. This can be done through the command line, or a program like Cyberduck. If you find yourself considering uploading a zip file through the web editor, you should instead upload your files using this method.
- GridFTP: For users that expect to upload hundreds or thousands of files, with a cumulative file size of hundreds of GB to TBs, you will likely want to make use of GridFTP through Globus. Jeanette will be talking about data transfers in more depth on Thursday.

Before you can upload your data to the Data Team's server, make sure to email us at `support@arcticdata.io` to retrieve the login password. Once you have that, you can proceed through the following steps.

> **Tip**
>
> If you know that you will need to use this process for more than one dataset, we suggest creating folders with the **same name** as the associated dataset's title. This way, it will be clear as to which submission the data should be associated with.

Once you have finished uploading your data to our servers, please let the Data Team know via email so that we can continue associate your uploaded data with your metadata submission.

As mentioned in Step 1: The Narrative Metadata Submission section above, when the data package is finalized and made public, there will be a sentence in the abstract that directs users to a separate page where your data will live. The following image is an example of where the data from this dataset live.



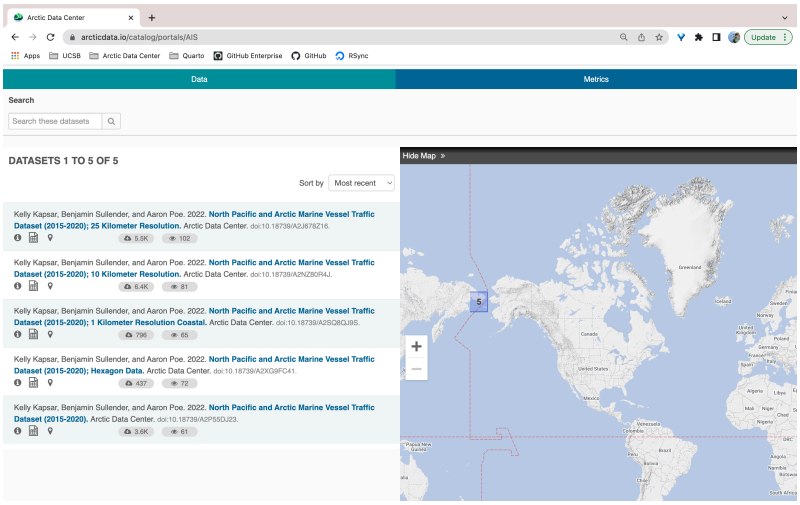
5.5 Best Practices for Submitting Large Data

5.5.1 Data Organization

The data archival process is much smoother when users already have a well structured and organized file management system. We strongly recommend starting to work with the Arctic Data Center towards the beginning of your project, instead of the very end. Our curation team can help you publish your data faster, and more smoothly, the earlier you talk to us. Regardless of where you are in your project, however, considering your organization strategy is best done ***before*** data upload.

Take a look at this recently made data portal. Originally, this user had only submitted one dataset with all of her files (over 700, at varying resolutions). The data contain rasters and shapefiles depicting vessel paths in the North Pacific at many different resolutions. Upon review, the submitter and curation team together determined that it would be best if other users could download the data at specific resolution scales. As a solution, we created separate datasets according to resolution and data type. With the addition of the data portal, the user was able to house their related datasets at a single URL, which allows for search and discovery within only the datasets for that project.

> **Note**
>
> To find out more about portals, visit https://arcticdata.io/data-portals/. You can view our catalog of portals here. We recommend reviewing the Distributed Biological Observatory's portal to see what a more fleshed out portal can look like.



Tip

This example also illustrates that users can create multiple datasets for a single NSF award. It's understandable that you may think you have to submit all your data for a given award under one dataset. However, we think breaking up data into separate datasets (in logical, structured ways) increases the discoverability and usability of the data. This is especially true when users also create a data portal for their project.

When uploading your data files using SFTP, either through the command line or a GUI, we recommend creating folders with the same name as your dataset's title so that it is easier to associate the data with their associated metadata submission.

For more complex folder structures, it is wise to include a README file that explicitly walks users through the structure and provides a basic understanding of what is available. Generally speaking, we recommend structuring your data in an easy to understand way such that a README isn't completely necessary.

5.6 Summary

In this lesson we learned about metadata and the structure of a data package; how to submit narrative metadata; how to upload data and when to use different services; how to document file and attribute-level metadata; and best practices for data organization and management. As mentioned earlier, more in-depth information on uploading data to our servers using Globus and RSync can be found in Session 18.

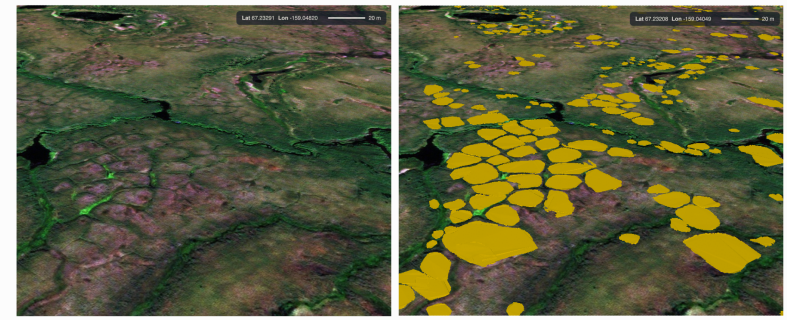
6 Group Project: Staging and Preprocessing

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

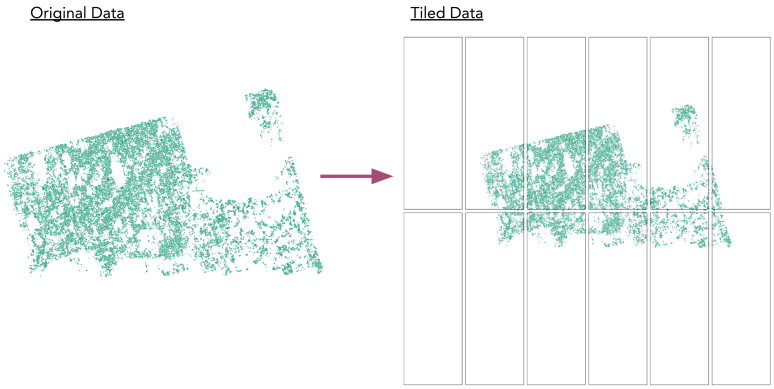
6.1 Introduction

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.

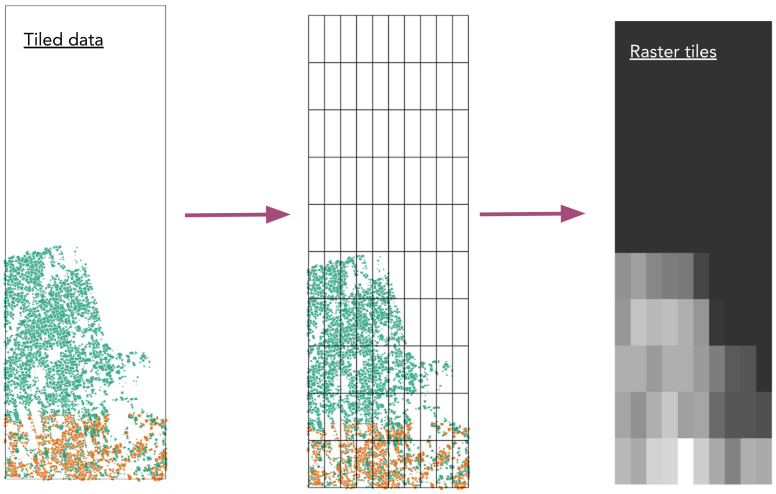
Ice wedges form as a result of thermal contraction during melt/freeze cycles of permafrost. They can form very distinctive geometries clearly visible in satellite images. The PDG is using advanced analysis and computational tools to take high resolution satellite imagery and automatically dectect where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



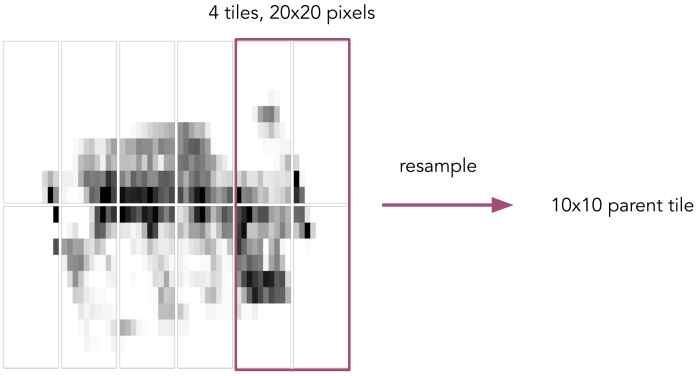
In the group project, we are going to use a subset of the high resolution dataset of these detected ice wedge polygons in order to learn some of the reproducible, scalable techniques that will allow us to process it. Our workflow will start with a set of large geopackage files that contain the detected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



6.2 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.

7 Software Design I

8 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`

8.1 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.

8.2 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.

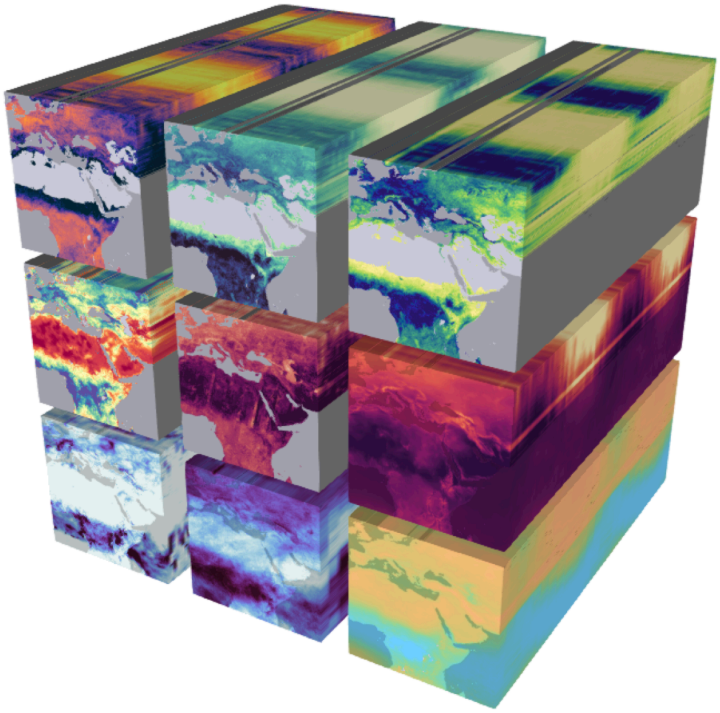


Figure 8.1: Mahecha et al. 2020 . *Visualization of the implemented Earth system data cube. The figure shows from the top left to bottom right the variables sensible heat (H), latent heat (LE), gross primary production (GPP), surface moisture (SM), land surface temperature (LST), air temperature (Tair), 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.

8.2.1 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.

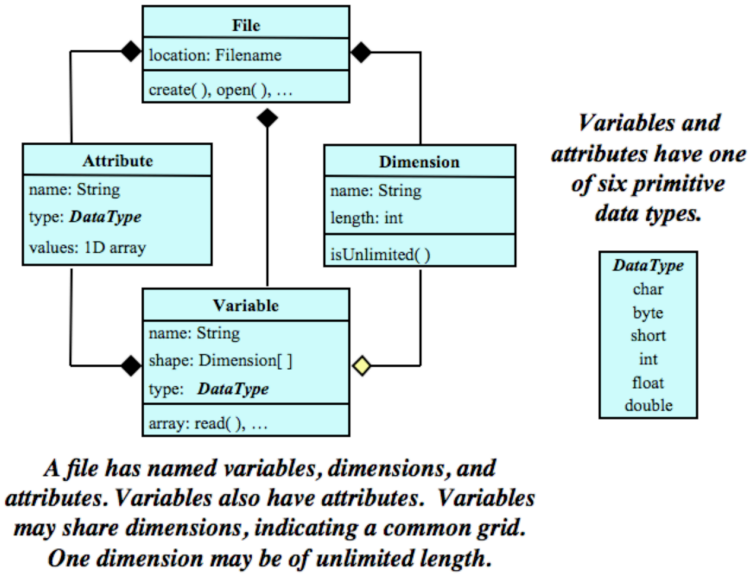


Figure 8.2: Classic NetCDF Data Model (NetCDF documentation)

8.2.2 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.

8.2.3 Exercise

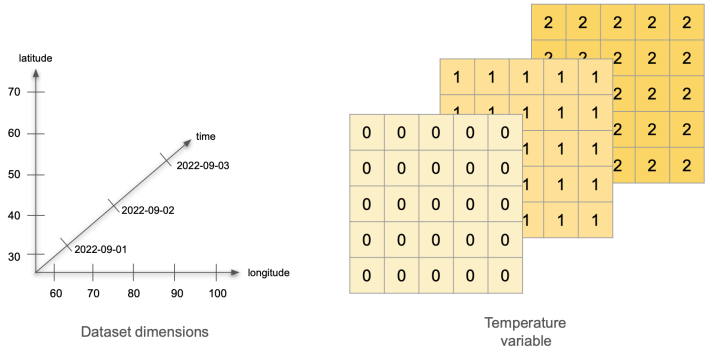
Let's do a short practice now that we have reviewed the classic NetCDF model and know a bit about metadata best practices.

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

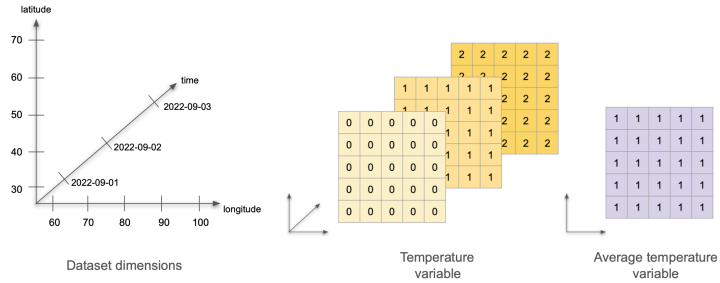
Part 2

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.

8.3 `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 8.3: `xarray`'s development portal

8.3.1 `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.

8.3.1.1 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 urllib
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.

8.3.1.2 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.

8.3.1.3 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.

8.3.2 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.

8.3.2.1 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
```

8.3.2.2 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
```

8.3.3 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

msg = urllib.request.urlretrieve(url, "FW_data_CESM_LW_2006_2100.nc")
```

```
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.

8.4 Tabular Data and NetCDF

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`.

8.4.1 Tabular to NetCDF

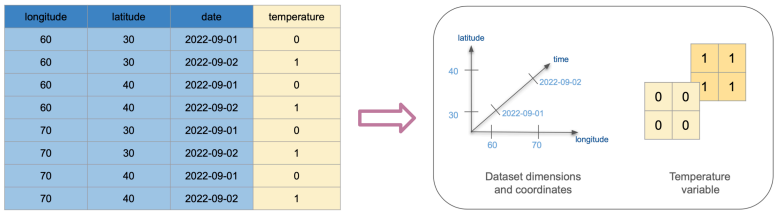
We assume our starting point is tabular data that meets the criteria for **tidy data**, which means:

- 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 an intrinsic 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.

8.4.2 `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

9 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

9.1 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.

9.2 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** compute tasks and store and return computations results. Workers can be threads, processes, or separate machines in a cluster. We won't go into how to choose between threads and processes, but there are best practices to select the right one. 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.

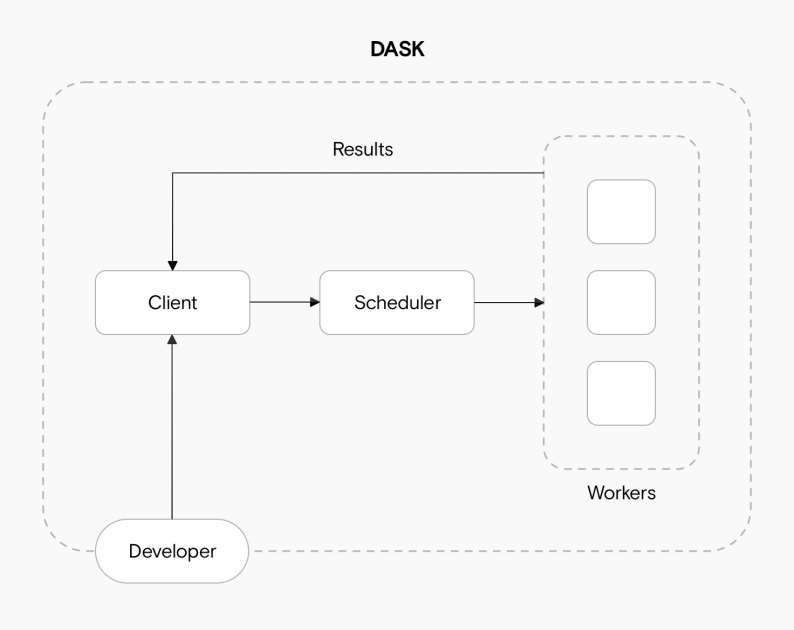


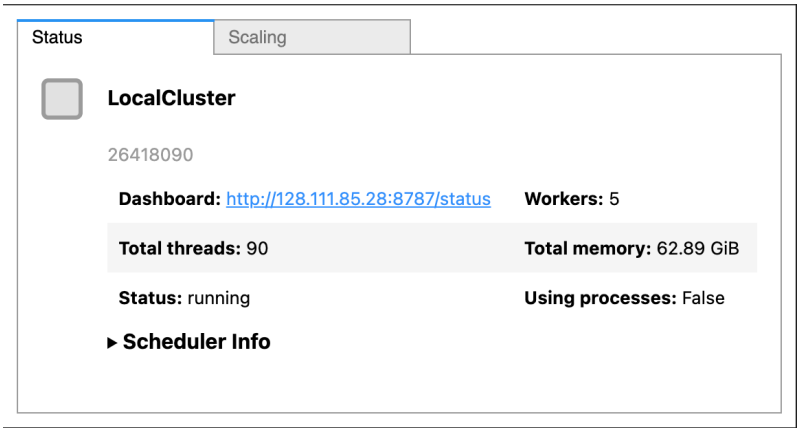
Figure 9.1: M. Schmitt, *Understanding Dask Architecture: The Client, Scheduler and Workers*

9.2.1 Setting up a Local Cluster

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

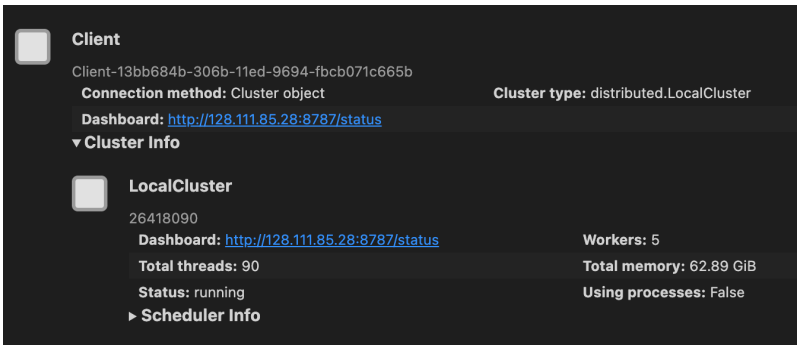
```
from dask.distributed import LocalCluster, Client
```

```
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.

9.2.2 Dask Dashboard

We chose to use the Dask cluster in this lesson instead of the default Dask scheduler to take advantage of the **cluster dashboard**, 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 9.2: A Dask dashboard.

9.3 `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`.

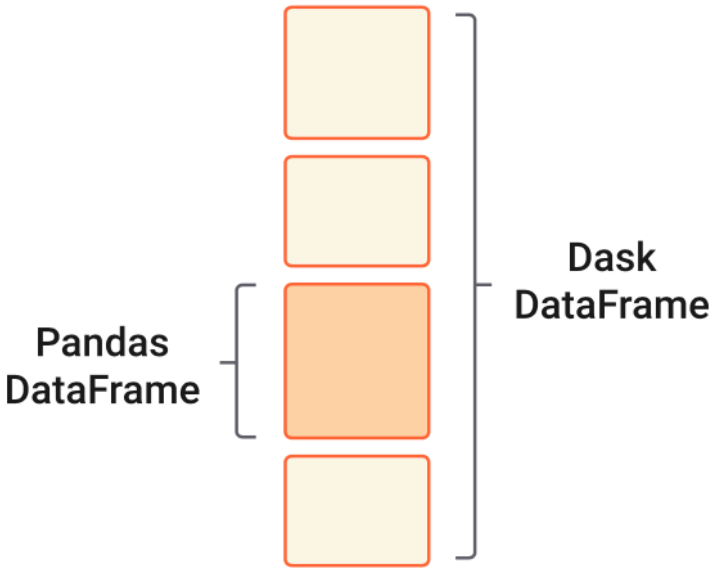


Figure 9.3: Dask Array design (dask documentation)

9.3.1 Reading a csv

To get familiar with `dask.dataframes`, we will use tabular data of soil moisture measurements at six forest stands in north-eastern 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 urllib

import dask.dataframe as dd
```

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

msg = urllib.request.urlretrieve(url, "dg_soil_moisture.csv")
```

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)
```

9.3.2 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()
```

9.4 `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.

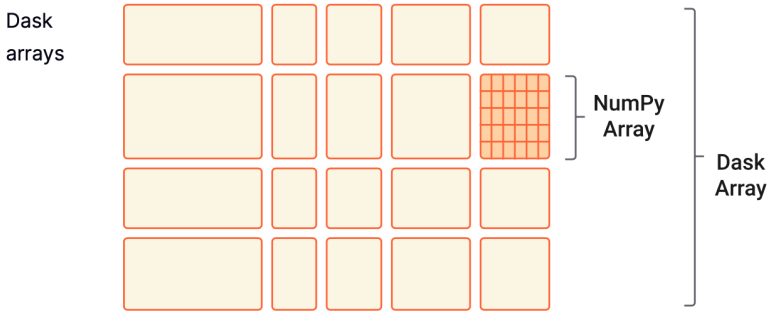


Figure 9.4: Dask Array design (dask documentation)

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()
```

9.5 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 and its extender `rioxarray` together with Dask. In the previous lesson, *Data Structures and Formats for Large Data*, we explore how to use the `xarray` package to work with labelled arrays. `rioxarray` extends `xarray` with the *rio* accessor, which stands for "raster input and output".

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]).

9.5.1 Open .tif file

As an example, let's do a Normalized Difference Vegetation Index (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.). The NDVI is an index commonly used to check if an area has live green vegetation or not. It can also show the difference between water, plants, bare soil, and human-made structures, among other things.

The NDVI is calculated using the near-infrared and red bands of the satellite image. The formula is

$$NDVI = \frac{NIR - Red}{NIR + Red}.$$

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
msg = urllib.request.urlretrieve(url, "RU_ANS_TR2_FL005M_red.tif")


# download nir band
url = 'https://arcticdata.io/metacat/d1/mn/v2/object/urn%3Auuid%3A1762205e-c505-450d-90ed-d4

msg = urllib.request.urlretrieve(url, "RU_ANS_TR2_FL005M_nir.tif")
```

Because these are .tif files and have geospatial metadata, we will use `rioxarray` to read them. 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
```

```
# 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')

#squeeze
nir = nir.squeeze()
```

9.5.2 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)
```

9.6 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

10 Spatial and Image Data Using GeoPandas

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

10.1 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

10.2 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."

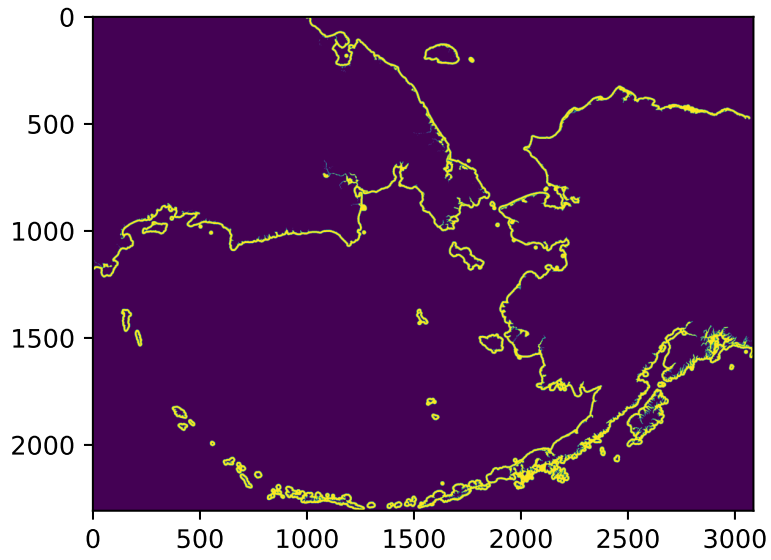
```
import urllib

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

```
msg = urllib.request.urlretrieve(url, "Coastal_2020_08.tif")
```

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.

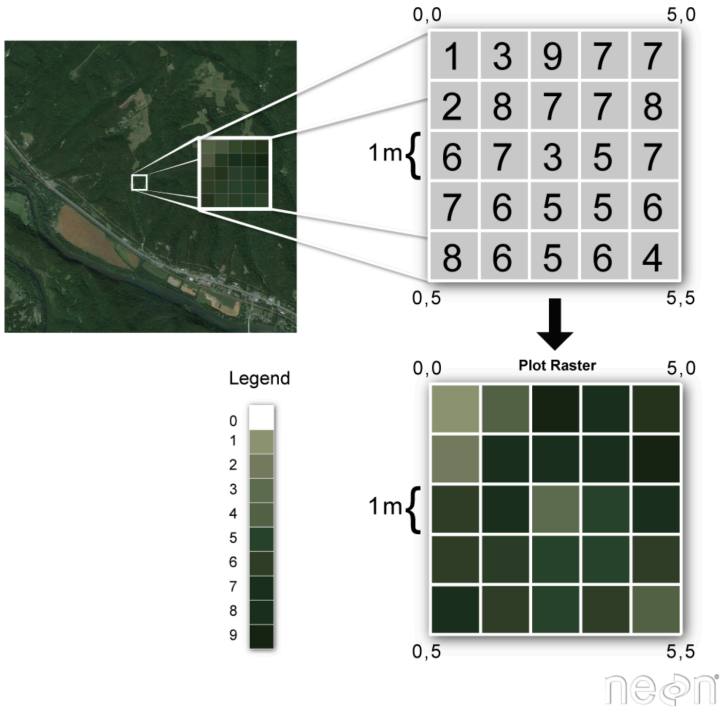
```
type(ships)
```

```
numpy.ndarray
```

```
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, you can see 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 an n-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, column) 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.

10.3 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

msg = urllib.request.urlretrieve(url, "Alaska_Commercial_Salmon_Boundaries.gpkg")
```

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:

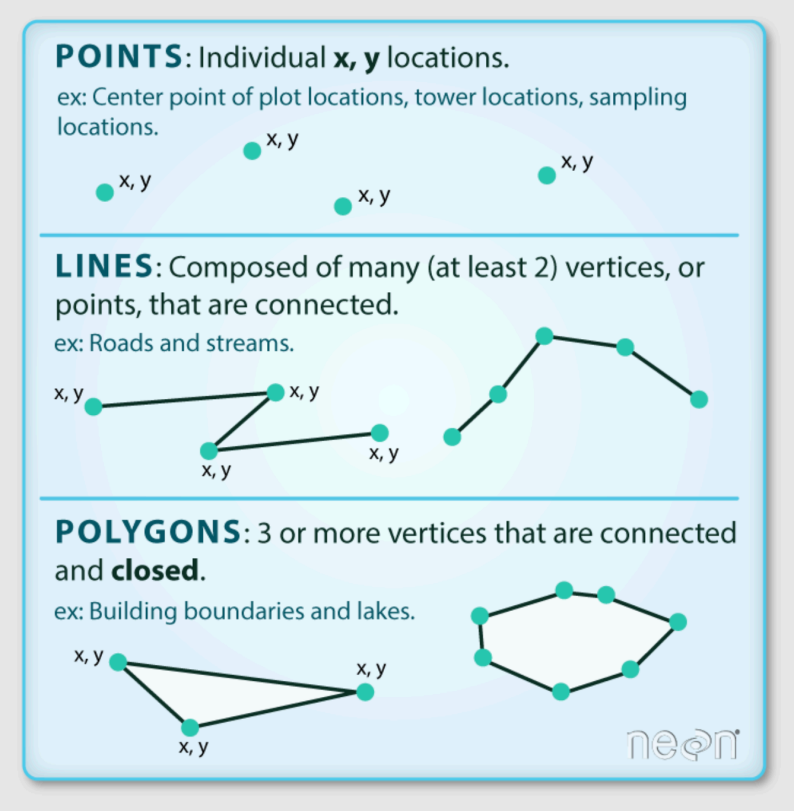
```
comm.head
```

```
<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:

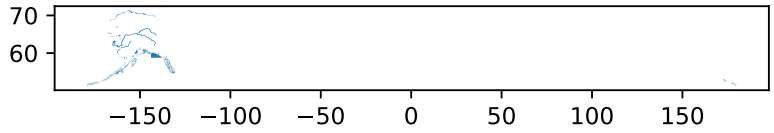
```
comm.crs
```

```
<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 analyze. First, we can make a plot of it just to see what we have.

```
comm.plot()
```

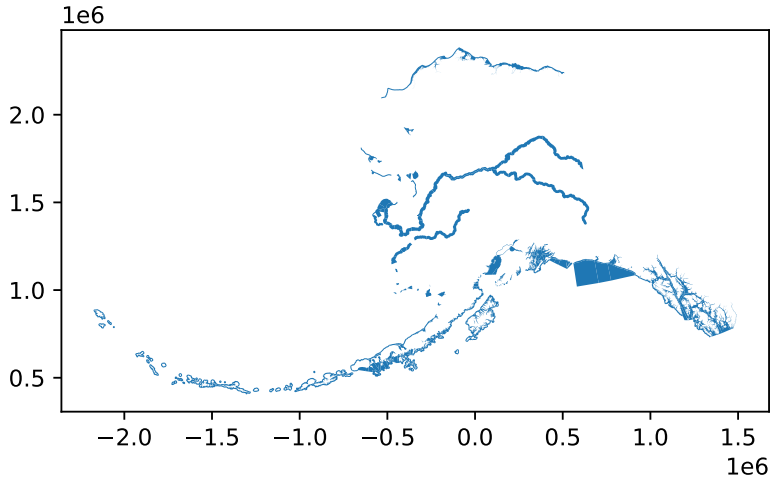
```
<AxesSubplot:>
```



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 are 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!

10.4 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 finally 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, because the `mask` function requires as its first `dataset` argument a dataset object opened in `r` mode.

```
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=True)
    shipc_meta = ship_con.meta
    # select just the 2-D array (by default a 3-D array is returned even though we only have
    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.

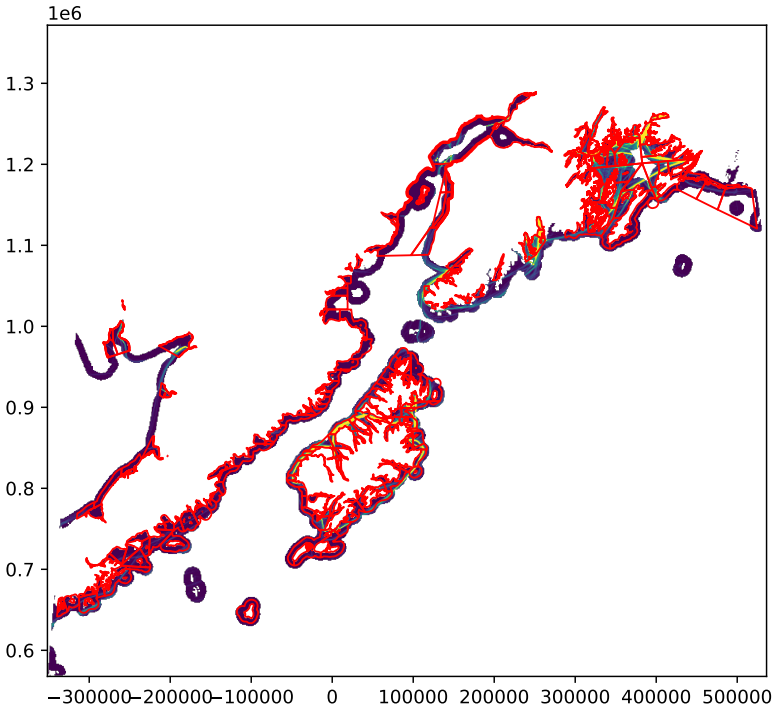
```
comm_clip = gpd.sjoin(comm_3338,
                      coord_box_df,
                      how='inner',
                      predicate='within')
```

10.4.1 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=(7, 7))
# 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='red')
```

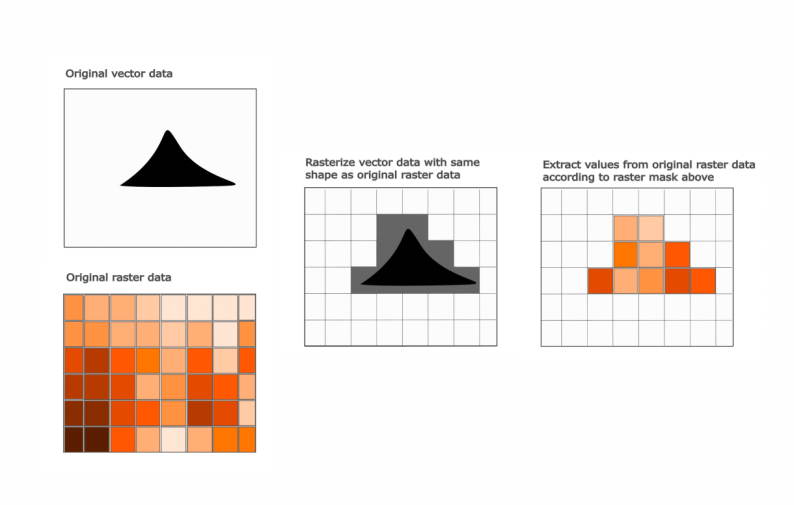
```
<AxesSubplot:>
```



10.5 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:



10.5.0.1 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), and the shape and transform the output raster will take. 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.

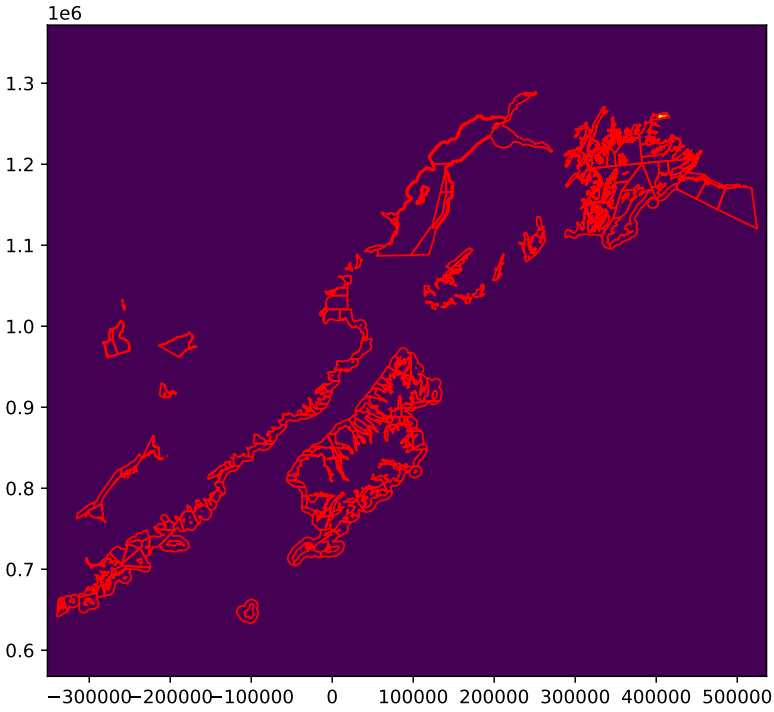
```
from rasterio import features

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 showing the rasterized district (with pixel values of 1) and any area not in the district has a pixel value of 0.

```
# set up plot
fig, ax = plt.subplots(figsize=(7, 7))
```

```
# 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='red')
```

```
<AxesSubplot:>
```



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

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

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

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

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

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

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

```
np.nansum(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.

10.5.0.2 Zonal statistics over all polygons

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

For each district (with `geometry` and `id`), we run the `features.rasterize` function. Then 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)

    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 and convert distance to k
distance_df['id'] = distance_df.index
distance_df['distance'] = distance_df['distance']/1000
```

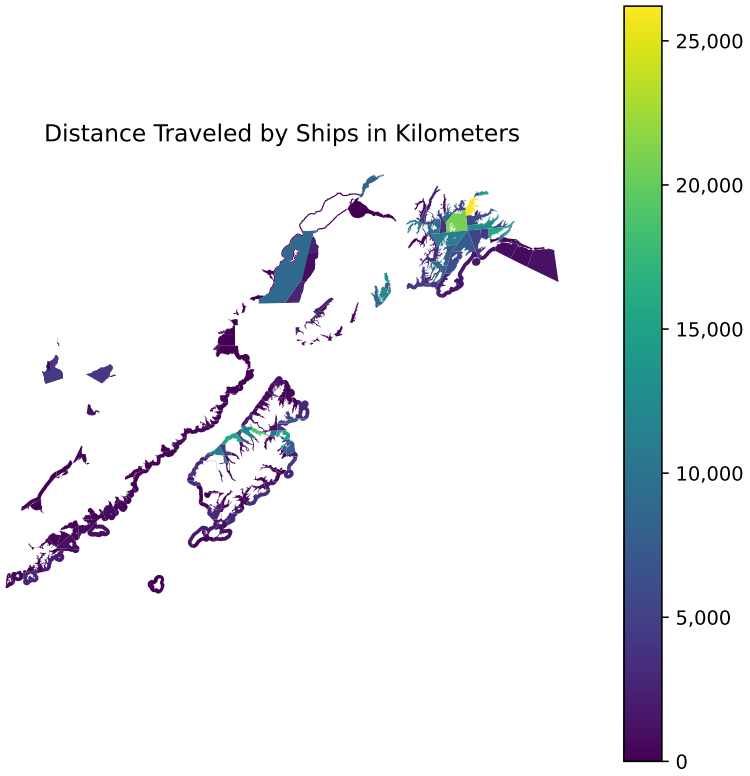
Now we join the result to the original geodataframe.

```
# 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()
```

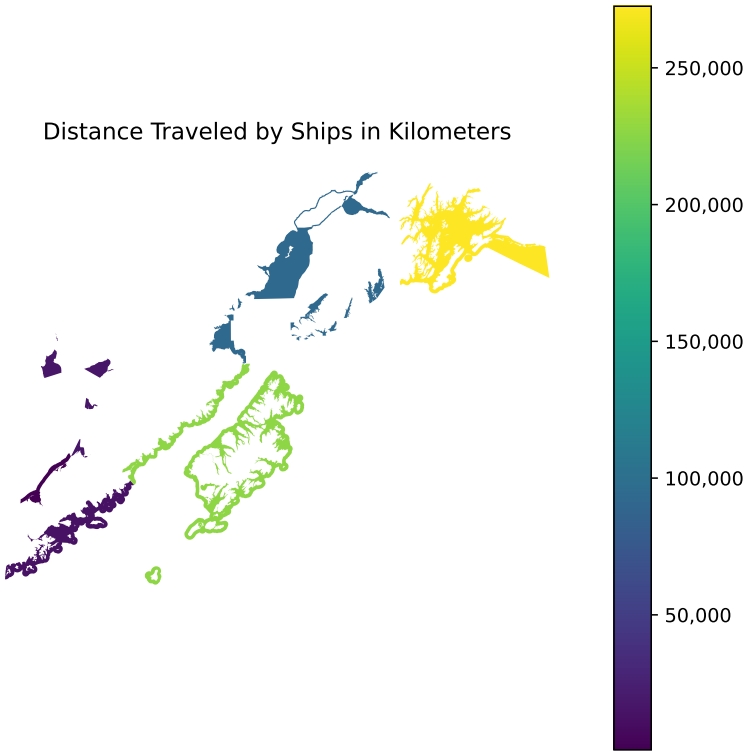


From here we can do any additional `geopandas` operations we might be interested in. For example, what if we want to calculate the total distance by registration area (a superset of fishing district). We can do that using `dissolve` from geopandas.

```
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()
```



10.6 Summary

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

- Raster data consist of regularly gridded values, and can be represented in python as an array
- Vector data consist 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.

11 Parquet and Arrow

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

11.1 Introduction

Paralleization is great, and can greatly help you in working with large data. However, it might not help you with every processing problem. Like we talked about with Dask, sometimes your data are too large to be read into memory, or you have I/O limitations. Parquet and `pyarrow` are newer, powerful tools that are designed to help overcome some of these problems. `pyarrow` and Parquet are newer technologies, and are a bit on the 'bleeding edge', but there is a lot of excitement about the possibility these tools provide.

Before jumping into those tools, however, first let's discuss system calls. These 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. 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. Finally, close closes the connection to the file.

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? My brand new M1 MacBook Pro has 16 GB of memory, 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.

11.2 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.

11.2.1 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 \n 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 (utilizing the seek system call). 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:

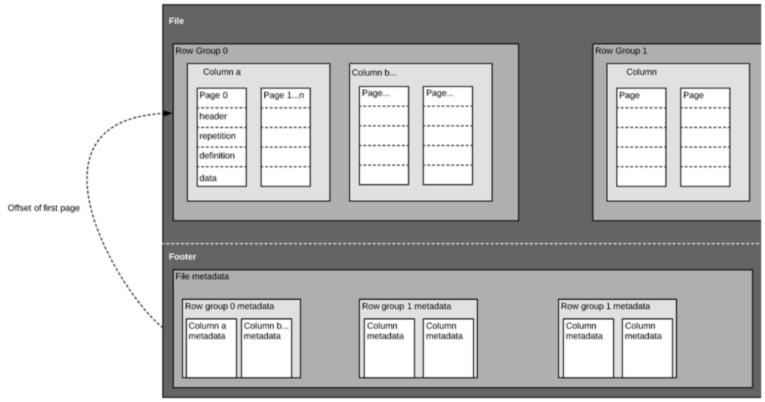
```
Name: John, Mariah, Allison
Location: Washington, Texas, Oregon
Age: 40, 21, 57
```

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.

11.3 Parquet

Parquet is an open-source binary 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.

11.4 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.

11.5 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 them 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.

11.6 Synopsis

In this lesson we learned:

- the difference between row major and column major formats
- under what circumstances a column major data format can improve memory efficiency
- how `pyarrow` can interact with Parquet files to analyze data

12 Software Design II

13 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.

14 Data Ethics for Scalable Computing

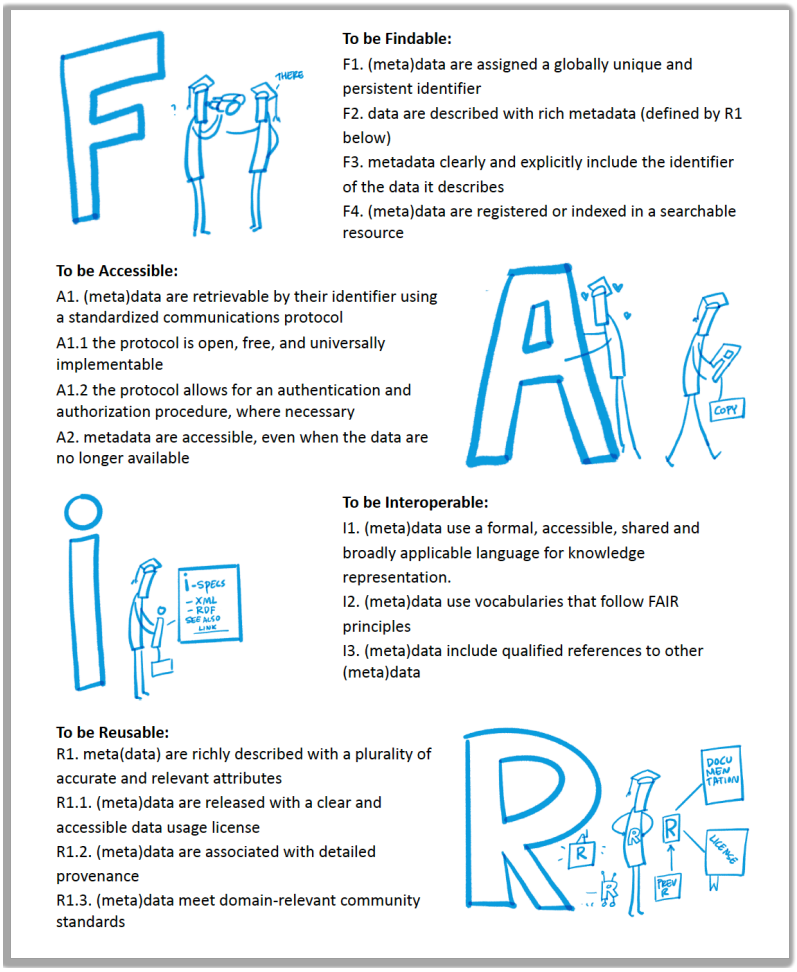
- Review FAIR and CARE Principles, and their relevance to data ethics
- Examine how ethical considerations are shared and considered at the Arctic Data Center
- Discuss ethical considerations in machine learning

14.1 Intro to Data Ethics

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).

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.

14.2 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.

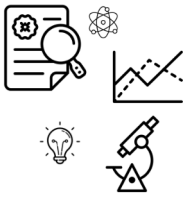
Inspired by CARE Principles for Indigenous Data Governance (Collective Benefit, Authority to Control, Responsibility, Ethics) and FAIR Principles (Findable, Accessible, Interoperable, Reproducible), we include a space in the data submission process for researchers to describe their ethical research

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.

14.2.1 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

14.2.2 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?

14.2.3 Human Participation and Sensitive Data

Research Planning

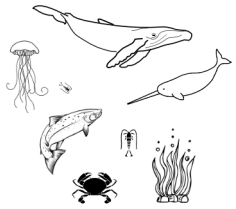
1. Please describe the institutional IRB approval that was required for this research.
2. Was any knowledge provided by community members?

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?

14.2.4 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.

14.2.5 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?

14.2.6 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?

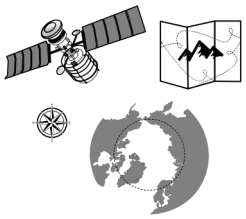
Data Collection

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?

14.2.7 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?

14.2.8 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?

14.3 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.

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

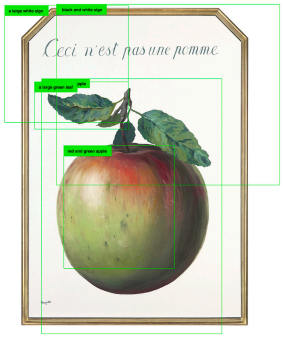


Image source: Kate Crawford and Trevor Paglen, "Excavating AI: The Politics of Training Sets for Machine Learning" (September 19, 2019).

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).

14.3.1.1 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?

14.4 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

15 Google Earth Engine

- Understand what Google Earth Engine provides and its applications
- Learn how to search for, import, manipulate, and visualize Google Earth Engine Data
- Learn about some real-world applications of Google Earth Engine in the geosciences ## Introduction

Google Earth Engine (GEE) is a geospatial processing platform powered by Google Cloud Platform. It contains over 30 years (and multiple 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 (Earth Engine is free to use for research, education, and non-profit use).

Image Source: Earth Engine Data Catalog

GEE is just one of a number of cloud platform solutions developed for climate and geoscience research. Others include Microsoft Planetary Computer, Pangeo, & Amazon Sustainability Data Initiative (ADSI)

So what's so exciting about platforms like GEE? Ryan Abernathey frames this nicely in his blogpost Closed Platform vs. Open Architectures for Cloud-Native Earth System Analytics...

- as Earth System data have gotten larger, the typical download-data-work-locally workflow is no longer always feasible
- those data are also produced and distributed by lots of different organizations (e.g. NASA, NOAA, Copernicus)
- researchers often need to apply a wide range of analytical methods to those data, ranging from simple stats to machine learning approaches

GEE offers web access (i.e. no need to download data to your computer) to an extensive catalog of analysis-ready geospatial

data (from many different organizations) 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).

In this lesson, we'll first get some hands-on practice connecting to and using Google Earth Engine to visualize global precicpation data. We'll then walk through a demonstration using GEE to visualize and analyze fire dynamics in the Arctic.

15.1 Exercise 1: An introductory lesson on using Google Earth Engine

15.1.1 Part i. Setup

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

- Please refer back to the Preface to find instructions on creating a GEE account.

2. Load libraries

```
import ee
import geemap
```

3. Authenticate your GEE account

- In order to begin using GEE, you'll need to connect your environment (`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).

```
ee.Authenticate() # triggers the authentication process
```

- 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 into the VS Code command palette (at the top of the IDE). 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. The browser-based authentication steps will look something like this:

a. **Notebook Authenticator:** choose an active Google account and Cloud Project (you may have to create one if this is your first time authenticating) and click "Generate Token"

b. **Choose an account:** if prompted, select the same Google account as above
c. **Google hasn't verified this app:** You may be temped to click the blue "Back to saftey" button, but don't! Click "Continue"

d. **Select what Earth Engine Notebook Client can access:** click both check boxes, then "Continue"

e. **Copy your authorization code** to your clipboard to paste into the VS Code command palette

4. Lastly, intialize. This verifies that valid credentials have been created and populates the Python client library with methods that the backend server supports.

```
ee.Initialize()
```

If successful, you're now ready to begin working with Earth Engine data!

15.1.2 Part ii. Explore the ERA5 Daily Aggregates Data

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.

ERA5 Daily Aggregates dataset, available via the Earth Engine Data Catelog

Reanalysis combines observation data with model data to provide the most complete picture of past weather and climate. To read more about reanalyses, check out the EWCMWF website.

Take a few moments to explore the metadata record for this dataset. You'll notice that it includes a bunch of important information, including:

- **Dataset Availability:** the date range
- **Dataset Provider:** where the data come from
- **Earth Engine Snippet:** a code snippet used for loading the dataset
- **Description (tab):** get to know a bit about the data
- **Bands (tab):** the variables present in the dataset; each band has its own name, data type, scale, mask and projection
- **Image Properties:** metadata available for each image band
- **Example Code:** a script to load and visualize ERA5 climate reanalysis parameters in Google Earth Engine (JavaScript)

15.1.3 Part iii. Visualize global precipitation using ERA5 Daily Aggregate data

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. 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
```

2. Load the ERA5 Image Collection from GEE

- Next, we need to tell GEE what data we want to layer on top of our basemap. 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.

> Earth Engine Snippets make importing ImageCollections easy!
>
> To import an ImageCollection, copy and paste the Earth Engine Snippet for your dataset of interest. For example, the Earth Enginge Snippet to import the ERA5 daily aggregates data can be found on the dataset page.

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

3. 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 – explore which variables the dataset contains under the Bands tab. We'll use the `select` method to choose the parameter(s) we're interested in from our `weatherData` object. Let's select the `total_precipitation` band.

```
# select desired bands (total_preciptation)
precip = weatherData.select("total_precipitation")
```

Plotting an **Image** will produce a static visualization (e.g. total precipitation at a particular point in time, or the average precipitation over a specified date range), while an **ImageCollection** can be visualized as either an animation or as a series of thumbnails (aka a "filmstrip"), such as this animation showing a three-day progression of Atlantic hurricanes in September, 2017 (source: Google Earth Engine).

- We can look at our `precip` object metadata using the `print` method to see that it's still an *ImageCollection*.

```
print(precip)
```

> **Note**
>
> You may see a message in the VS Code Interactive pane that says, **"Output exceeds the size limit. Open the full output data in a text editor"** when printing your image object. Click here to see the entire output, which includes date range information.

- Let's say that we want to look at data for a particular time of interest – e.g. January 1, 2019 - December 31, 2019. We can apply the `filterDate` method to our selected `total_precipitation` parameter to filter for data from our chosen date range. We can also apply the `mean` method, which takes whatever precedes it and calculates the average.

```
# initial date of interest (inclusive)
i_date = '2019-01-01'

# final data of interest (exclusive)
f_date = '2020-01-01'

# select apporpriate bands (total_preciptation), dates, and calculate mean precipitation acr
precip = weatherData.select("total_precipitation").filterDate(i_date, f_date).mean()
```

- Use the `print` method again to check out your new `precip` object – notice that it's now an `ee.Image` (rather than `ee.ImageCollection`) and the start and end date values are as we specified.

```
print(precip)
```

4. Add the precipitation Image to the basemap

- We can first use the `setCenter` method to tell the map where to center itself (here, we center it over Cook Inlet, Alaska). It takes the longitude and latitude as the first two arguments, followed by the zoom level.

```
myMap.setCenter(lat = 60, lon = -151, zoom = 4)
myMap
```

- Next, set a color palette to use when plotting the data layer. The following is a palette specified for ERA5 precipitation data (scroll down to the example code, available on the landing page for the ERA5 metadata in the Earth Engine Data Catelog). Here, we adjusted the max value to change the range of pixel values to which the palette should be applied – this will make our colors stand out a bit more when we layer our precipitation data on our basemap, below.

```
precip_palette = {
    'min':0,
    'max':0.01,
    'palette': ['#FFFFFF', '#00FFFF', '#0080FF', '#DA00FF', '#FFA400', '#FF0000']
}
```

Learn more about GEE color palettes and Image visualization here.

> **Note**
>
> GEE has lots of pre-defined color palettes to choose from based on the type of data you want to visualize. Also check out Crameri et al. 2020 for recommended best practices when choosing color gradients that accurately represent data and are readable by those with color vision deficiencies.

- Finally, plot our filtered data, `precip`, on top of our basemap using the `addLayer` method. We'll also 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).

```
myMap.addLayer(precip, precip_palette, 'total precipitation', opacity = 0.3)
myMap
```

In just five lines of code (and mere seconds of execution time), we've applied and visualized a global precipitation model. We can zoom in/out across our interactive map while Google Earth Engine recalculates and revisualizes our model in near real-time.

> **!** TODO
>
> While GEE does have its limitations as a closed platform service, it provides researchers with quick and easy access many analysis-ready geospatial datasets...

15.2 Exercise 2: Visualize and analyze fire dynamics in the Arctic using GEE

Landscape changes in Arctic permafrost regions can occur as the result of a variety of processes including slope processes (thaw slumps, landslides), wildfires, hydrological changes, and coastal dynamics.

In this exercise, we'll explore fire dynamics Batagay, Russia, the location of the largest permafrost crater in the world – the Batagaika crater.

Top image source: https://www.science.org/do/10.1126/science.abe0581/full/main_batagai_1280p2.jpg. Bottom image source: https://worldview.earthdata.nasa.gov/

Using the following code, we'll demonstrate how to:

- load and visualize image data
- calculate and visualize indices
- work with ImageCollections

- extract time-series
- clustering / unsupervized classification

15.2.1 Part i. Setup

1. Import libraries

```
import ee
import geemap
import eemont
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

2. Authenticate & Initialize

```
ee.Authenticate()
ee.Initialize()
```

2. Create basemap displaying the Batagay region

```
# site coordinates
xy = [134.6, 67.66]
poi = ee.Geometry.Point(xy)
```

```
# create map
Map = geemap.Map()
Map.addLayer(poi)
Map.centerObject(poi, zoom=3)
Map
```

15.2.2 Part ii. Visualize NDVI and NBF spectral indices

We'll plot *Image* data using two different multispectral indices:
(1) **Normalized Difference Vegetation Index (NDVI)** for

visualizing vegegation, using RGB (Red - Green - Blue), and (2) **Normalized Burn Ratio (NBF)** for visualizing burned areas, using Color-Infrared (NIR - Red - Green).

1. Instantiate map

```
Map = geemap.Map()
```

2. Load map

> ! TODO: code below this point erroring out

```
# load image
S2_Image = (ee.Image('COPERNICUS/S2_SR/20210805T030551_20210805T030545_T53WMR')
            .scaleAndOffset()
            .spectralIndices(['NDVI', 'NBR']) # Calculate multispectral indices
           )

# Show
Map.addLayer(S2_Image, dict(min=0.0, max=0.15, gamma=1.5, bands=['B2', 'B3', 'B2']), 'RGB')
Map.addLayer(S2_Image, dict(min=0.0, max=0.3, gamma=1.5, bands=['B8', 'B4', 'B3']), 'Color-I

RdYlGn = ['#d7191c','#fdae61','#ffffbf','#a6d96a','#1a9641']
Map.addLayer(S2_Image, dict(min=0.0, max=0.8, bands=['NDVI'], palette=RdYlGn), 'NDVI')
Map.addLayer(S2_Image, dict(min=0.0, max=0.8, bands=['NBR'], palette=RdYlGn), 'NBR')

Map.centerObject(S2_Image, zoom=8)

Map
```

3. Select point on the map above and pull location as an `ee.Feature`

```
point_selected = Map.draw_last_feature
point_selected

# or alternatively, use this code:
point_fire = ee.Geometry.Point([134.6339, 68.0397])
```

15.2.3 Part iii. Load more than one image (i.e. an ImageCollection) to analyze

You can do the same analysis with an *ImageCollection* (stacks of images of the same type).

1. Here, we'll load the full Sentinel-2 (S2) series and filter for our data of interest (Location: Batagay & Dates: months June - September). We'll also filter out cloud cover, which can obstruct our spectral indices analysis.:

```
# Image Collection
ds = 'COPERNICUS/S2_SR'

S2 = (ee.ImageCollection(ds)
      ### Filters ###
      .filterBounds(poi) # only images, which intersect our point/region
      .filter(ee.Filter.calendarRange(6,9,'month')) # only June through September, Summer on
      .filter(ee.Filter.lt('CLOUDY_PIXEL_PERCENTAGE', 10)) # only images with less that 10 %
      #.filterDate('2022-07-01','2022-12-31')
      ### PreProcessing ###
      .scaleAndOffset()
      .spectralIndices(['NDVI', 'NBR'])
      .maskClouds()
      .sort('CLOUDY_PIXEL_PERCENTAGE')
     )#.first()
# calculate indices (e.g. NDVI, NBR)
print(f'Number of images {S2.size().getInfo()}') # Print number of images
```

2. Visualize satellite data

```
#Map = geemap.Map()
Map.addLayer(S2, dict(min=0.0, max=0.15, gamma=1.5, bands=['B2', 'B3', 'B2']), 'RGB_Collecti
Map.addLayer(S2, dict(min=0.0, max=0.3, gamma=1.5, bands=['B8', 'B4', 'B3']), 'Color-Infrare
RdYlGn = ['#d7191c','#fdae61','#ffffbf','#a6d96a','#1a9641']

# Show NDVI
#Map.addLayer(S2_Image, dict(min=0.0, max=0.8, bands=['NDVI'], palette=RdYlGn), 'NDVI')
# Show NDVI
#Map.addLayer(S2_Image, dict(min=0.0, max=0.8, bands=['NBR'], palette=RdYlGn), 'NBR')
```

```
Map.centerObject(S2_Image, zoom=8)
Map
```

15.2.4 Part iv. Make a timeseries

*Example taken from https://eemont.readthedocs.io/en/0.2.0/guide/timeSeries.html.

1. Create timeseries & clean data

```
def load_and_prepare_Landsat(collection_name):
    collection = (ee.ImageCollection(collection_name)
      ### Filters ###
      .filterBounds(poi) # only images, which intersect our point/region
      .filter(ee.Filter.calendarRange(7,8,'month')) # only June through September, Summer on
      .filter(ee.Filter.lt('CLOUD_COVER_LAND', 10)) # only images with less that 10 % clouds
      #.filterDate('2022-07-01','2022-12-31')
      ### PreProcessing ###
      .scaleAndOffset()
      .spectralIndices(['NDVI', 'NBR'])
      .maskClouds()
     )
    return collection
```

```
L9 = load_and_prepare_Landsat("LANDSAT/LC09/C02/T1_L2")
L8 = load_and_prepare_Landsat("LANDSAT/LC08/C02/T1_L2")
L7 = load_and_prepare_Landsat("LANDSAT/LE07/C02/T1_L2")
L5 = load_and_prepare_Landsat("LANDSAT/LT05/C02/T1_L2")
LS = L9.merge(L8).merge(L7).merge(L5)
```

```
ts = LS.getTimeSeriesByRegion(reducer = ee.Reducer.mean(),
                              geometry = point_fire,
                              bands = ['NBR','NDVI'],
                              scale = 10)
```

```
tsPandas = geemap.ee_to_pandas(ts)
tsPandas
```

```
# a little more cleaning is necessary
tsPandas[tsPandas == -9999] = np.nan
tsPandas['date'] = pd.to_datetime(tsPandas['date'],infer_datetime_format = True)

tsPandas = tsPandas.sort_values(by='date').dropna(axis=0).reset_index()

tsPandas
```

2. Visualize output (example using plotly, but can use library of your choice e.g. matplotlib, seaborn, etc.)

```
# scatterplot
px.scatter(tsPandas, x='date', y=['NDVI', 'NBR'])
```

```
# line plot
px.line(tsPandas, x='date', y=['NDVI', 'NBR'])
```

15.2.5 Part v. Clustering

Example taken from https://geemap.org/notebooks/31_unsupervised_classification/#make-training-dataset.

1. Group individual images into several "similar" objects

```
image = S2_Image
```

```
# Make the training dataset.
training = image.sample(
    **{
        #     'region': region,
        'scale': 10,
        'numPixels': 5000,
        'seed': 0,
        'geometries': True,  # Set this to False to ignore geometries
    }
)
```

```
# Instantiate the clusterer and train it.
n_clusters = 5
clusterer = ee.Clusterer.wekaKMeans(n_clusters).train(training)
```

```
# Cluster the input using the trained clusterer.
result = image.cluster(clusterer)

# # Display the clusters with random colors.
Map.addLayer(result.randomVisualizer(), {}, 'clusters')
```

```
legend_keys = ['One', 'Two', 'Three', 'Four', 'ect']
legend_colors = ['#8DD3C7', '#FFFFB3', '#BEBADA', '#FB8072', '#80B1D3']

# Reclassify the map
result = result.remap([0, 1, 2, 3, 4], [1, 2, 3, 4, 5])

Map.addLayer(
    result, {'min': 1, 'max': 5, 'palette': legend_colors}, 'Labelled clusters'
)
Map.add_legend(
    legend_keys=legend_keys, legend_colors=legend_colors, position='bottomright'
)
```

```
Map
```

15.3 Other Resources

Tools:
- GEE Code Editor, a web-based IDE for using GEE (JavaScript)

Data:
- Earth Engine Data Catalog, the main resource for "official" GEE Datasets
- awesome-gee-community-datasets, the main resource for "community" GEE Datasets

Documentation, Tutorials, & Help:

- `earthengine-api` installation instructions
- Creating and managing Google Cloud projects
- Troubleshooting authentication issues
- An Intro to the Earth Engine Python API
- geemap documentation
- Qiusheng Wu's YouTube channel for GEE & geemap Python tutorials - GEE on StackExhange

16 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.

17 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

17.1 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.

17.2 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.

17.2.0.0.1 * `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.

17.2.0.0.2 * `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!

17.2.0.0.3 * 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.

17.2.0.1 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.

17.2.0.2 `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.

17.3 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:

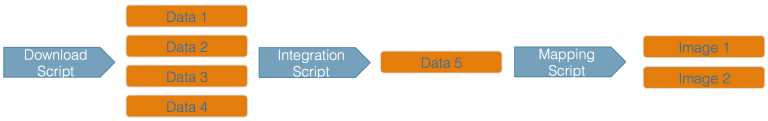
- use self documenting file formats, for metadata efficiency
  - netcdf
  - geotiff, geopackage
- regular, parseable filenames and consistent file formatting
- communicate early and often with the Arctic Data Center staff, preferably before you start a submission and well before your final report is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

17.4 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.

17.4.1 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.

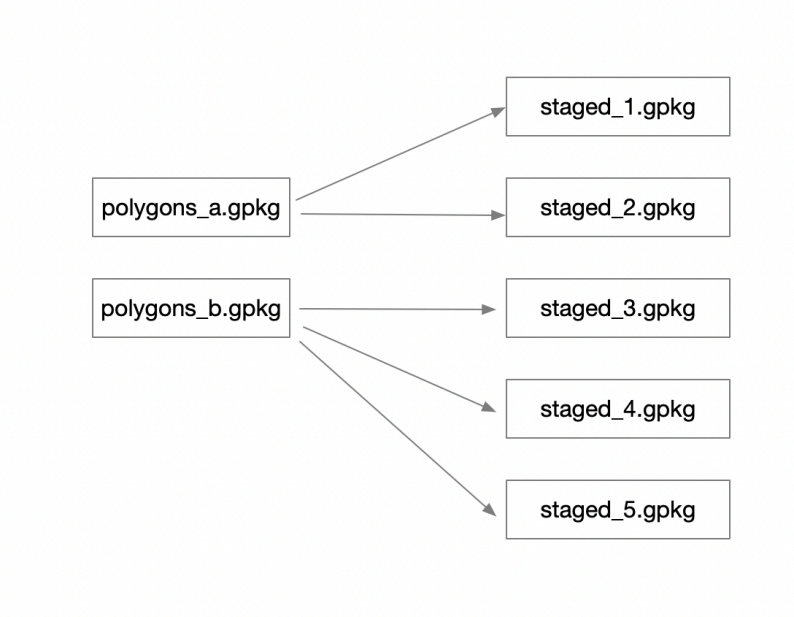
Workflows can be implemented in many ways, with various benefits:

- 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…

17.4.2 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

18 What is Cloud Computing Anyways?

19 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

19.1 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)

19.2 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.