INTRODUCTION TO DATA SCIENCE

Elias Gonzalez

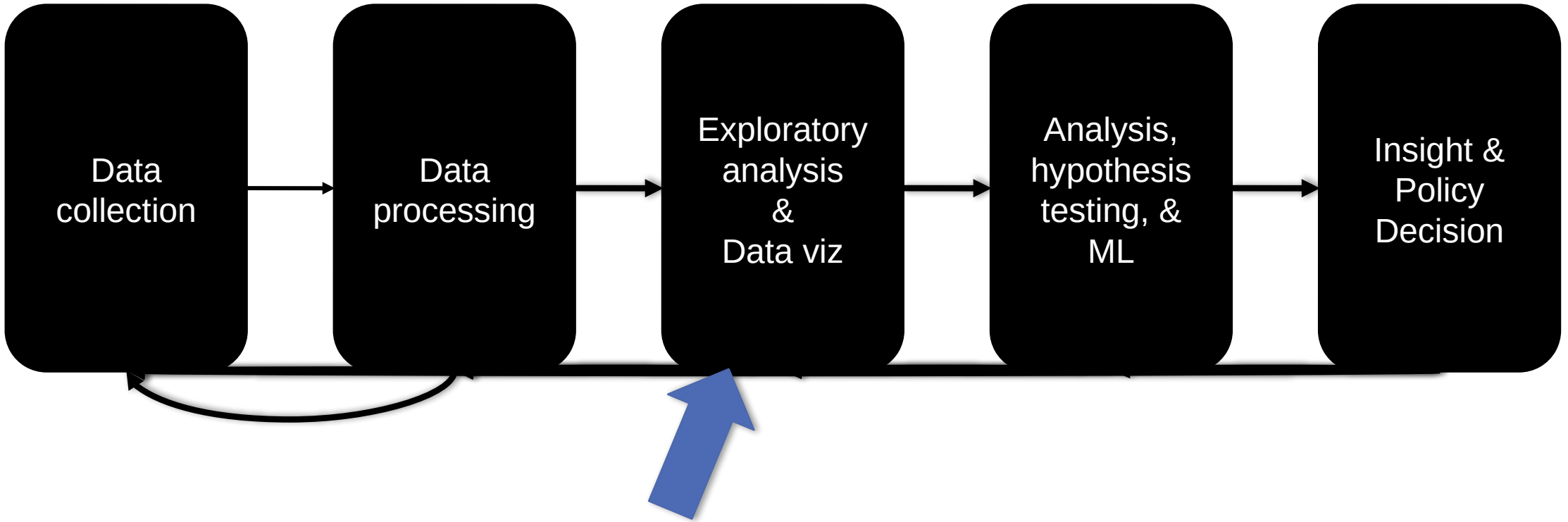
Lecture #8 – 02/16/2022

CMSC320
Monday & Wednesday
3:30pm – 4:45pm

https://cmsc320.github.io/

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

THE DATA LIFECYCLE

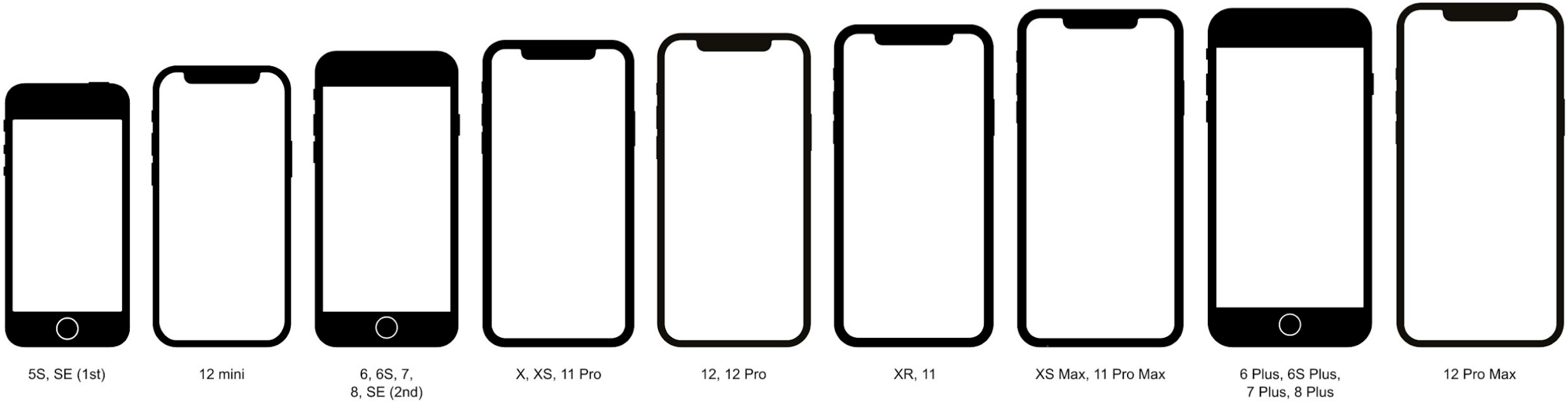


Version control for tracking code/data and for managing collaboration.

TODAY'S LECTURE

By popular request …

- **Version control primer!**
- **Specifically, git via GitHub and GitLab**
- **Thanks: Mark Groves (Microsoft), Ilan Biala & Aaron Perley (CMU), Sharif U., & the HJCB Senior Design Team!**



WHAT IS VERSION CONTROL?

```
Aaron@HELIOS ~/112_term_project
$ ls
termproject_actually_final  termproject_v10  termproject_v3
termproject_final           termproject_v11  termproject_v4
termproject_handin          termproject_v12  termproject_v5
termproject_old_idea        termproject_v13  termproject_v6
termproject_superfrogger    termproject_v14  termproject_v7
termproject_temp            termproject_v15  termproject_v8
termproject_this_one_works  termproject_v16  termproject_v9
termproject_v1              termproject_v2
```

DEVELOPMENT TOOL

When working with a team, the need for a central repository is essential

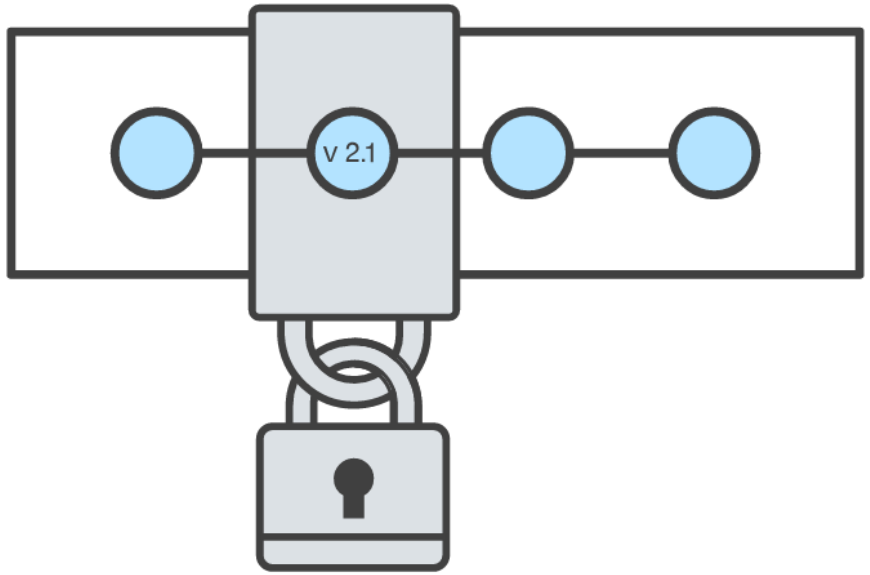
- Need a system to allow versioning, and a way to acquire the latest edition of the code
- A system to track and manage bugs was also needed

GOALS OF VERSION CONTROL

Be able to search through revision history and retrieve previous versions of any file in a project

Be able to share changes with collaborators on a project

Be able to confidently make large changes to existing files



atlassian.com/git/tutorials/what-is-version-control

NAMED FOLDERS APPROACH

Can be hard to track

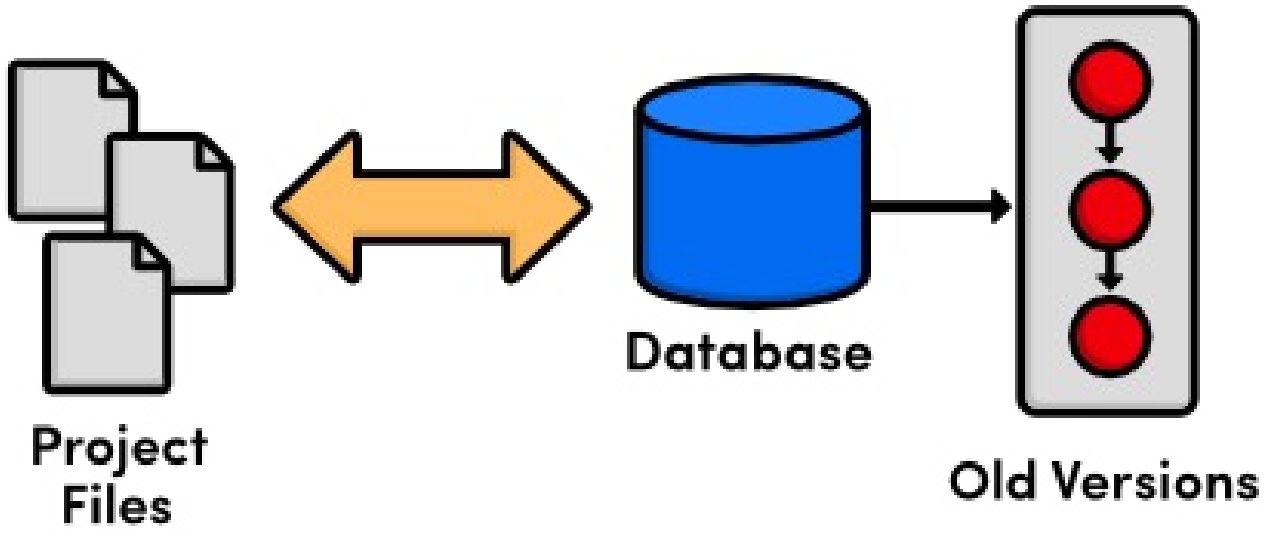
Memory-intensive

Can be slow

Hard to share

No record of authorship

LOCAL DATABASE OF VERSIONS APPROACH



Provides an abstraction over finding the right versions of files and replacing them in the project

Records who changes what, but hard to parse that

Can’t share with collaborators

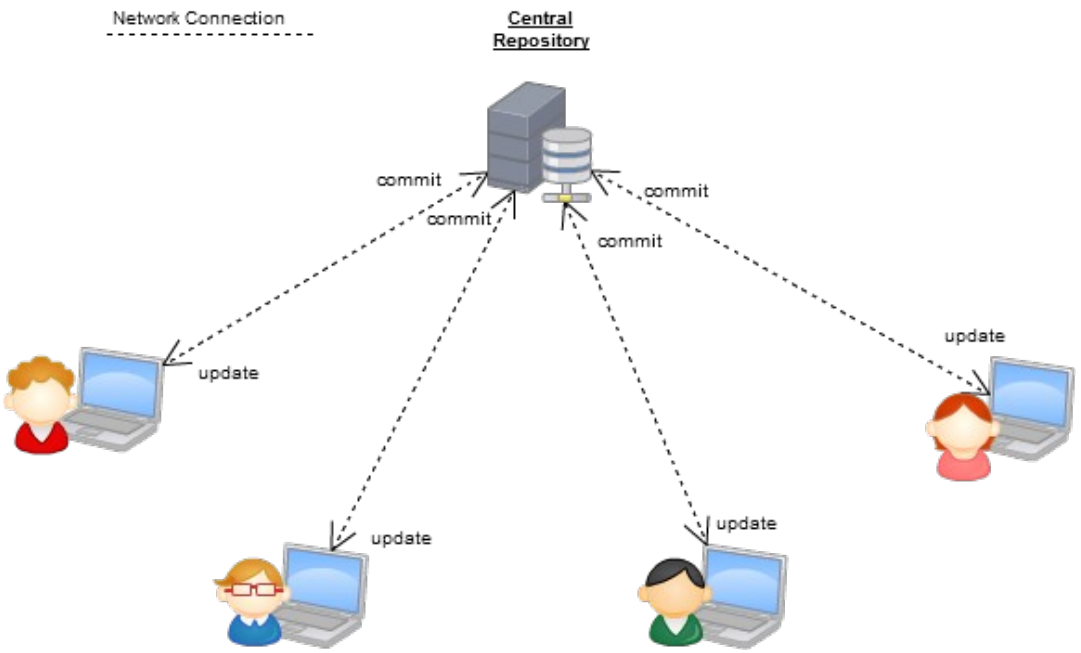
CENTRALIZED VERSION CONTROL SYSTEMS

A central, trusted repository determines the order of commits (“versions” of the project)

Collaborators “push” changes (commits) to this repository.

Any new commits must be compatible with the most recent commit. If it isn’t, somebody must “merge” it in.

Examples: SVN, CVS, Perforce

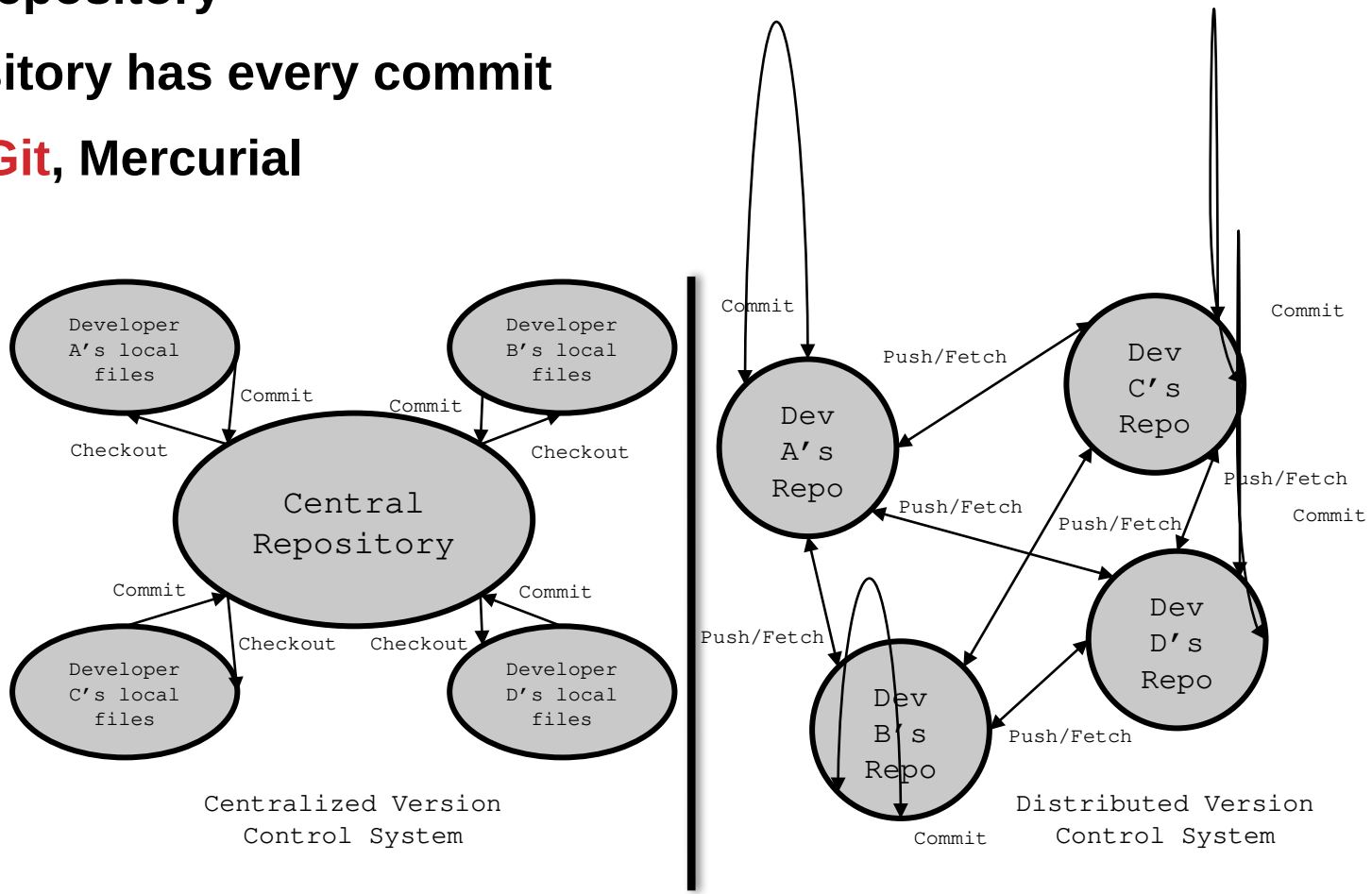


DISTRIBUTED VERSION CONTROL SYSTEMS (DVCS)

No central repository

Every repository has every commit

Examples: Git, Mercurial



WHAT IS GIT

Git is a version control system

Developed as a repository system for both local and remote changes

Allows teammates to work simultaneously on a project

Tracks each commit, allowing for a detailed documentation of the project along every step

Allows for advanced merging and branching operations

A SHORT HISTORY OF GIT

Linux kernel development

1991-2002

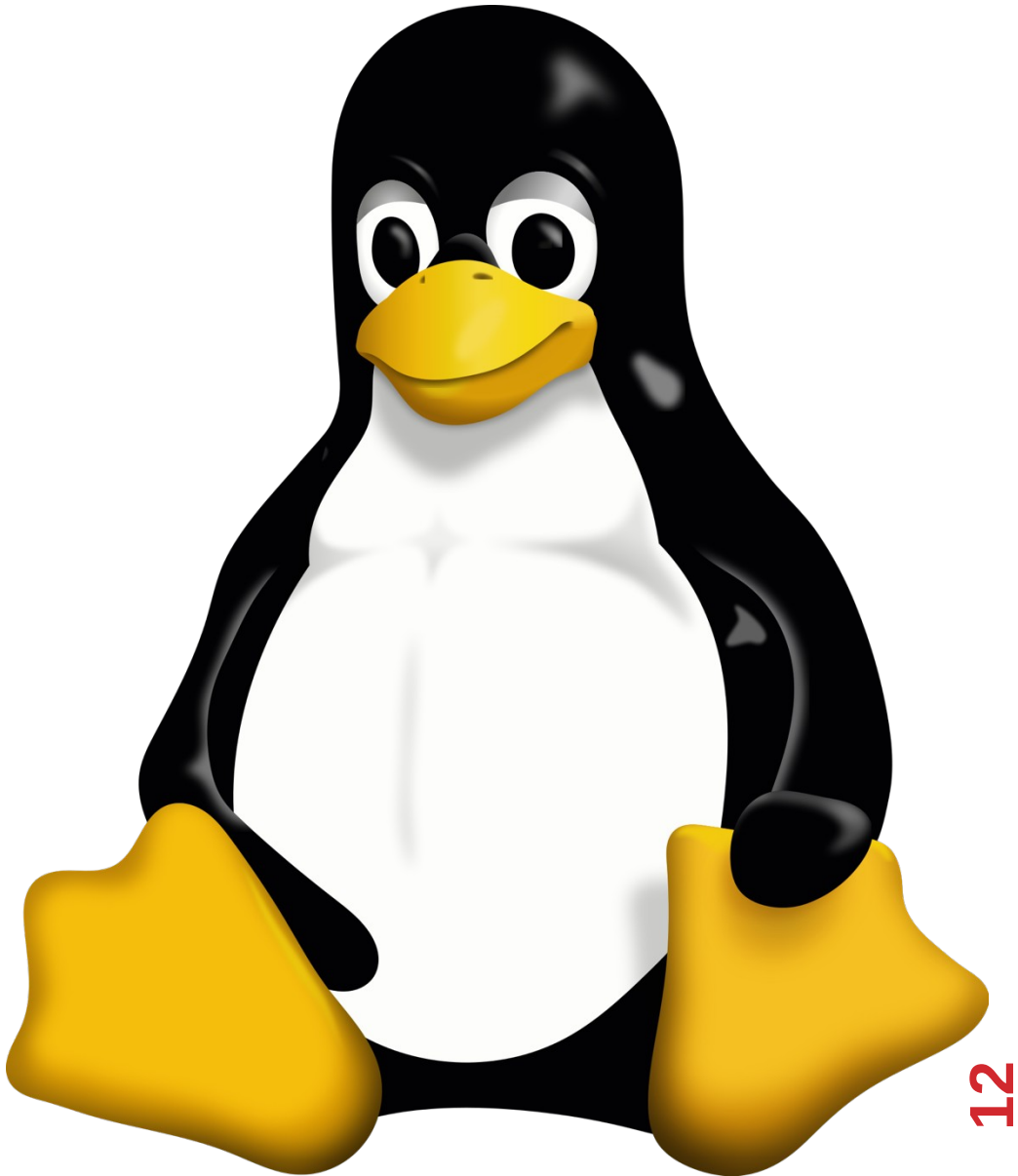
Changes passed around as archived file

2002-2005

Using a DVCS called BitKeeper

2005

Relationship broke down between two communities (BitKeeper licensing issues)



A SHORT HISTORY OF GIT

Goals:

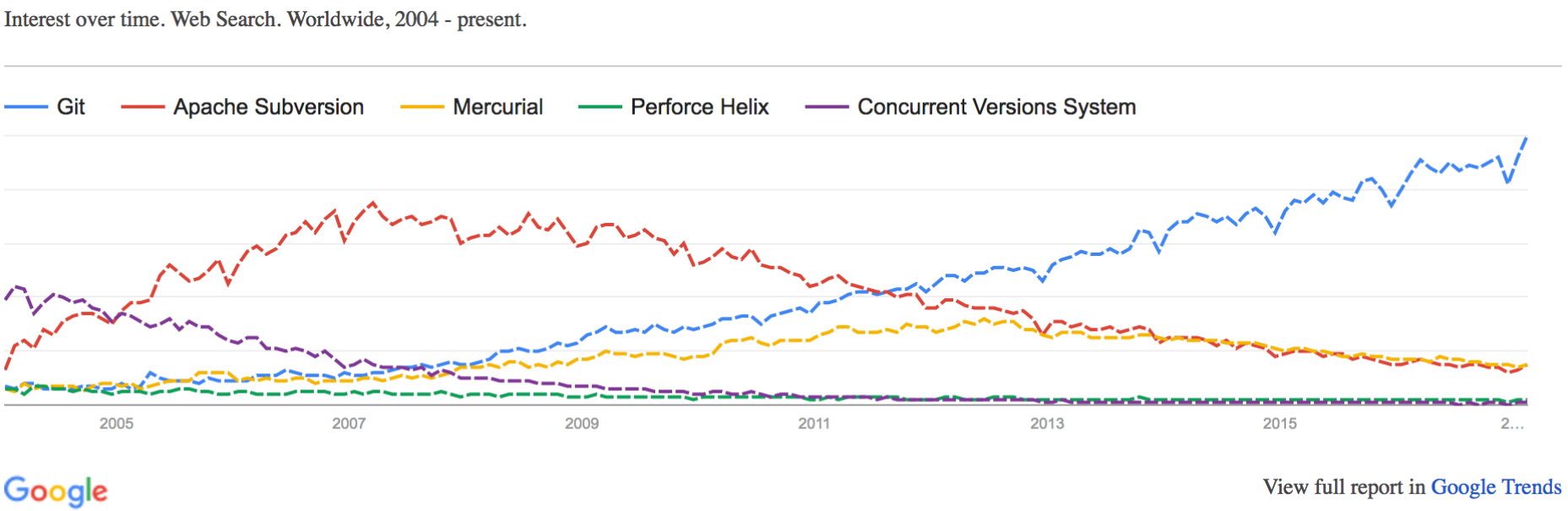
- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed – not a requirement, can be centralized
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

A SHORT HISTORY OF GIT

Popularity:

- Git is now the most widely used source code management tool
- 50% of professional software developers use Git (often through GitHub) as their primary source control system

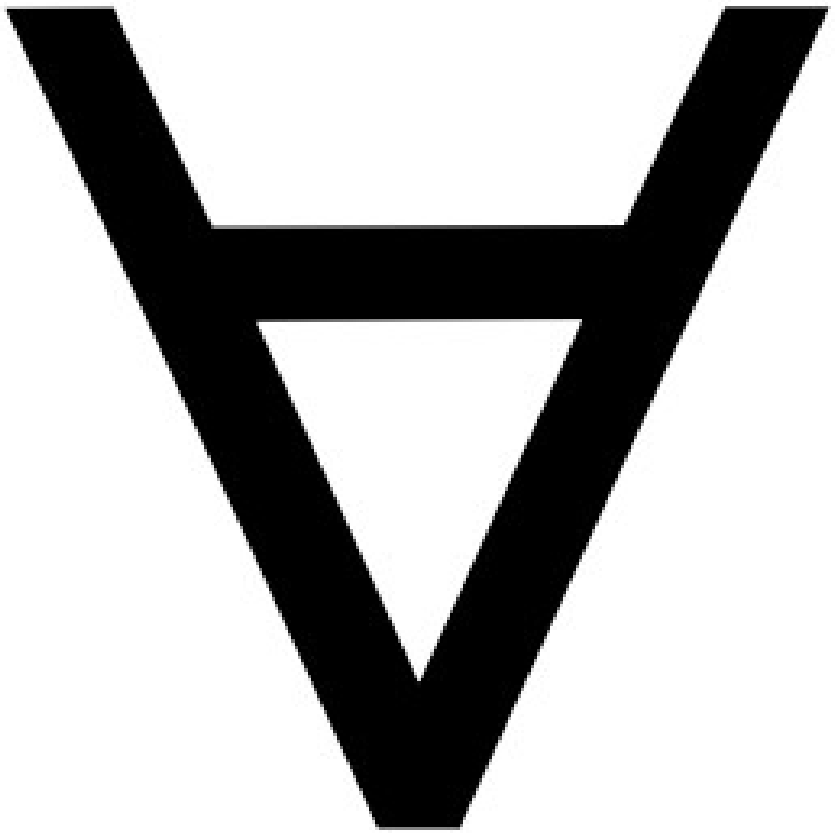
[citation needed, IMO much more ☺]



GIT IN INDUSTRY

Companies and projects currently using Git

- Google
- Android
- Facebook
- GitHub
- Microsoft
- Netflix
- Linux
- Ruby on Rails
- Gnome
- KDE
- Eclipse
- X.org



GIT BASICS

Snapshots, not changes

- A picture of what all your files look like at that moment
- If a file has not changed, store a reference

Nearly every operation is local

- Browsing the history of project
- See changes between two versions

WHY GIT IS BETTER

Git tracks the content rather than the files

Branches are lightweight, and merging is a simple process

Allows for a more streamlined offline development process

Repositories are smaller in size and are stored in a single .git directory

Allows for advanced staging operations, and the use of stashing when working through troublesome sections

Important Command Line Commands!

cd

ls

git

sudo

code

Important Command Line Commands!

cd- Change directory, must be followed by a folder to enter

ls - List files and folders in current directory

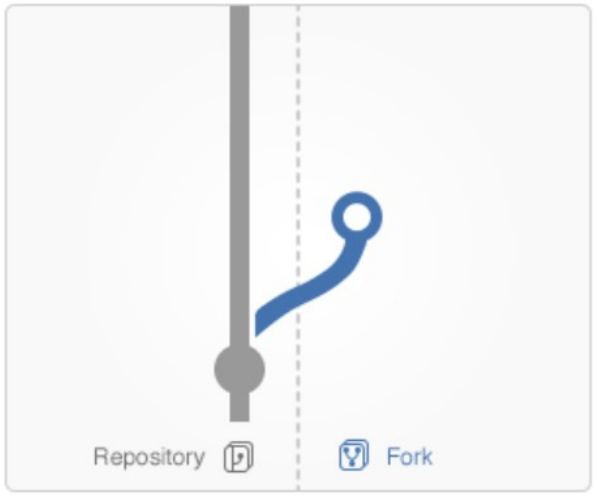
git- Must precede any git commands

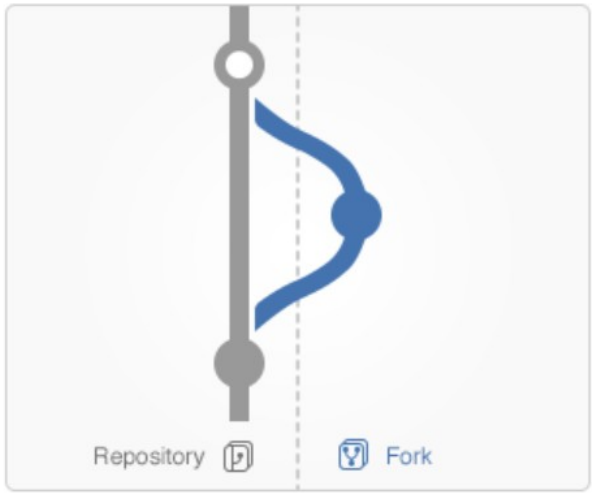
Sudo-
On linux machines, used whenever permission is needed to execute command

code- Followed by a folder or file, opens it in VScode

Other Vocab

Fork- Making a copy of a branch and allowing you to make changes freely
Branch- The stream you create by forking
Master/Main- Main Branch
Merge- Merges the stream you forked back into the master





Other Vocab

git status → To see if there are any changes to commit

Other Vocab

git status → To see if there are any changes to commit

git fetch [remote]→ See if there is anything to pull

git pull → Actually pulling from upstream

GIT VS {CVS, SVN, …}

Why you should care:

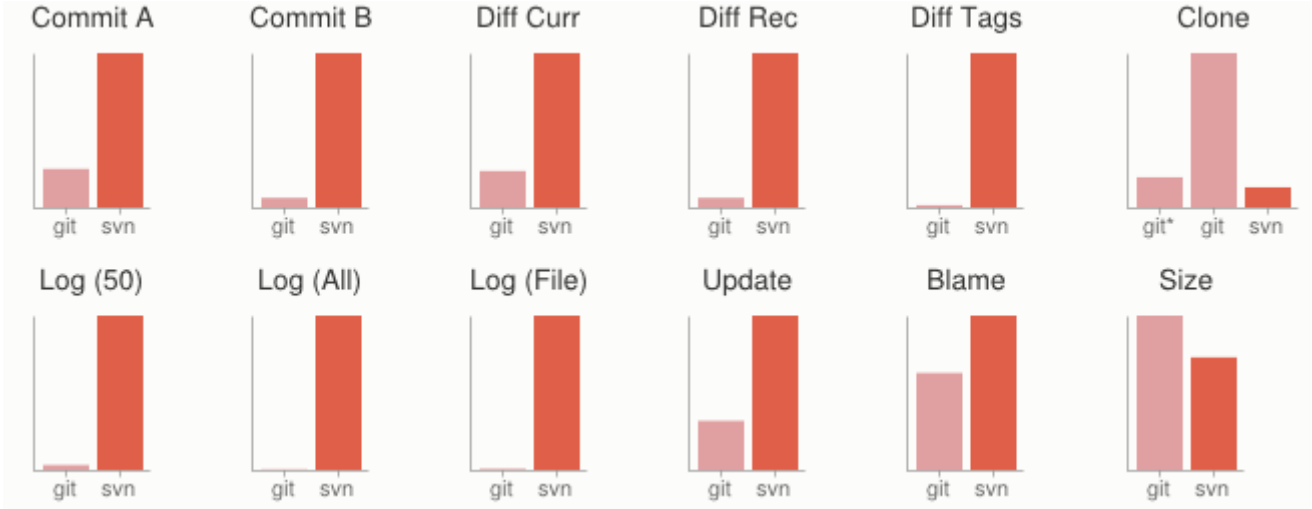
- Many places use legacy systems that will cause problems in the future – be the change you believe in!

Git is much faster than SVN:

- Coded in C, which allows for a great amount of optimization
- Accomplishes much of the logic client side, thereby reducing time needed for communication
- Developed to work on the Linux kernel, so that large project manipulation is at the forefront of the benchmarks

GIT VS {CVS, SVN, …}

Speed benchmarks:



Benchmarks performed by http://git-scm.com/about/small-and-fast

GIT VS {CVS, SVN, …}

Git is significantly smaller than SVN

- All files are contained in a small decentralized .git file
- In the case of Mozilla's projects, a Git repository was 30 times smaller than an identical SVN repository
- Entire Linux kernel with 5 years of versioning contained in a single 1 GB .git file
- SVN carries two complete copies of each file, while Git maintains a simple and separate 100 bytes of data per file, noting changes and supporting operations

Nice because you can (and do!) store the whole thing locally

GIT VS {CVS, SVN, …}

Git is more secure than SVN

- All commits are uniquely hashed for both security and indexing purposes
- Commits can be authenticated through numerous means
  - In the case of SSH commits, a key may be provided by both the client and server to guarantee authenticity and prevent against unauthorized access

GIT VS {CVS, SVN, …}

Git is decentralized:

- Each user contains an individual repository and can check commits against itself, allowing for detailed local revisioning
- Being decentralized allows for easy replication and deployment
- In this case, SVN relies on a single centralized repository and is unusable without

GIT VS {CVS, SVN, …}

Git is flexible:

- Due to it’s decentralized nature, git commits can be stored locally, or committed through HTTP, SSH, FTP, or even by Email
- No need for a centralized repository
- Developed as a command line utility, which allows a large amount of features to be built and customized on top of it

GIT VS {CVS, SVN, …}

Data assurance: a checksum is performed on both upload and download to ensure sure that the file hasn't been corrupted.

Commit IDs are generated upon each commit:

- Linked list style of commits
- Each commit is linked to the next, so that if something in the history was changed, each following commit will be rebranded to indicate the modification

GIT VS {CVS, SVN, …}

Branching:

- Git allows the usage of advanced branching mechanisms and procedures
- Individual divisions of the code can be separated and developed separately within separate branches of the code
- Branches can allow for the separation of work between developers, or even for disposable experimentation
- Branching is a precursor and a component of the merging process

Will give an example shortly.

GIT VS {CVS, SVN, …}

Merging

- The process of merging is directly related to the process of branching
- Individual branches may be merged together, solving code conflicts, back into the default or master branch of the project
- Merges are usually done automatically, unless a conflict is presented, in which case the user is presented with several options with which to handle the conflict

Will give an example shortly.

GIT VS {CVS, SVN, …}

Merging: content of the files is tracked rather than the file itself:

- This allows for a greater element of tracking and a smarter and more automated process of merging
- SVN is unable to accomplish this, and will throw a conflict if, e.g., a file name is changed and differs from the name in the central repository
- Git is able to solve this problem with its use of managing a local repository and tracking individual changes to the code

INITIALIZATION OF A GIT REPOSITORY

```
C:\> mkdir CoolProject
C:\> cd CoolProject
C:\CoolProject > git init
Initialized empty Git repository in
C:/CoolProject/.git
C:\CoolProject > notepad README.txt
C:\CoolProject > git add .
C:\CoolProject > git commit –m 'my first commit'
[master (root-commit) 7106a52] my first commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.txt
```

UNIX®

A Standard of The Open Group®

-based systems such as MacOS (more related to FreeBSD) and Linux (e.g., the Ubuntu distros many of you are running)

```
john@Johns-MBP ~ % mkdir cmsc320 && cd cmsc320
john@Johns-MBP cmsc320 % git init
Initialized empty Git repository in /Users/john/cmsc320/.git/
john@Johns-MBP cmsc320 % touch README.md
john@Johns-MBP cmsc320 % git add .
john@Johns-MBP cmsc320 % git commit -m "First commit!"
```

GIT BASICS I

The three (or four) states of a file:

- **Modified:**
  - File has changed but not committed
- **Staged:**
  - Marked to go to next commit snapshot
- **Committed:**
  - Safely stored in local database
- **Untracked!**
  - Newly added or removed files

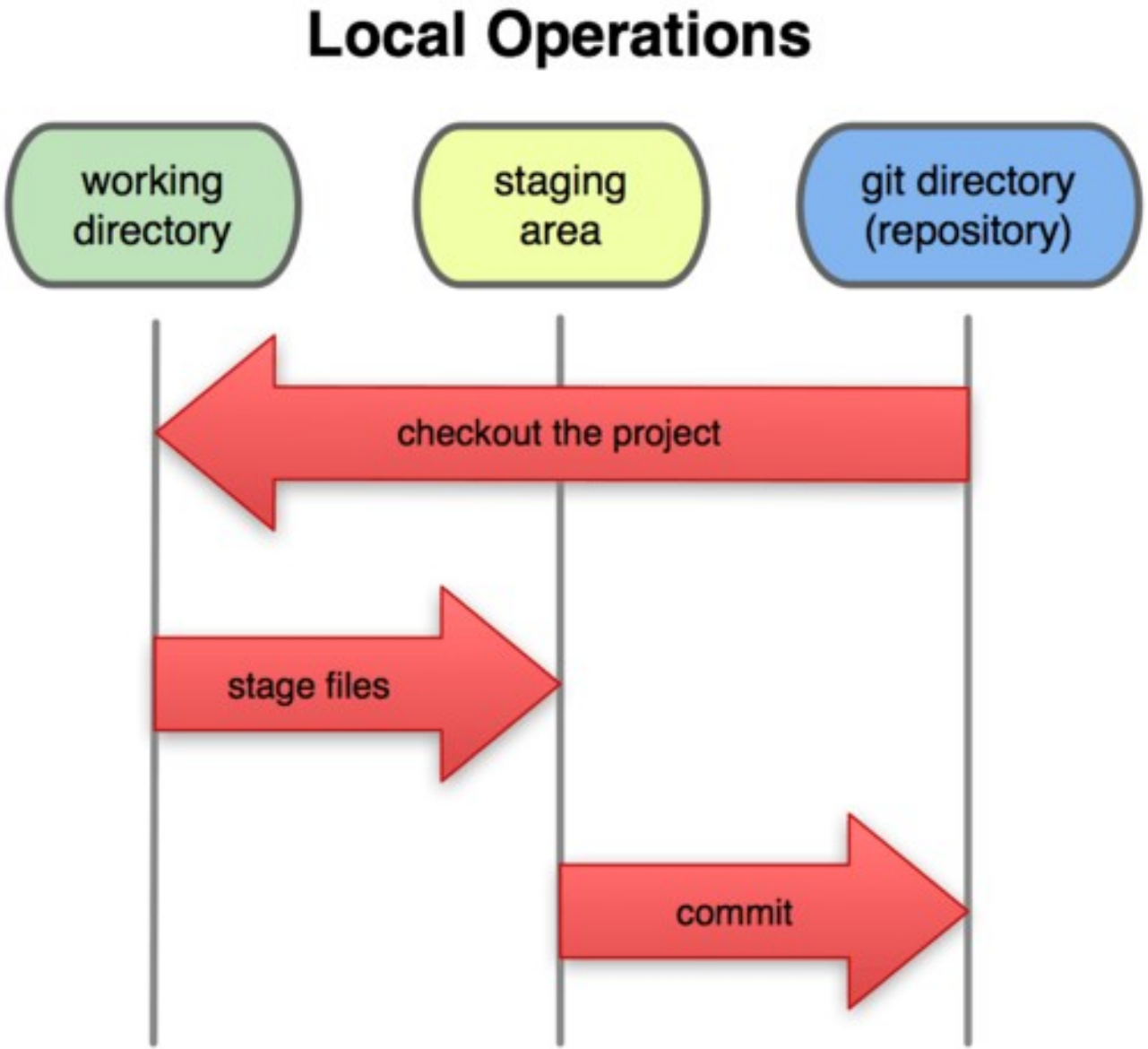
GIT BASICS II

Three main areas of a git project:

- **Working directory**
  - Single checkout of one version of the project.
- **Staging area**
  - Simple file storing information about what will go into your next commit
- **Git directory**
  - What is copied when cloning a repository

GIT BASICS III

Three main areas of a git project:

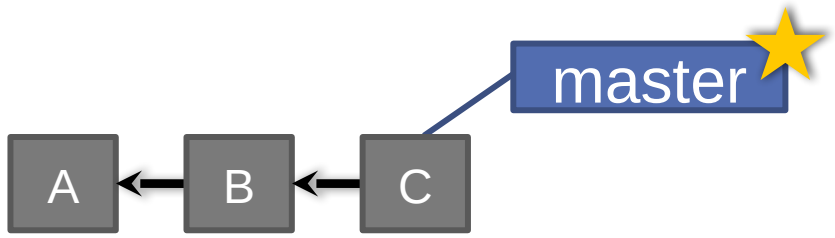


BRANCHES ILLUSTRATED

(Default branch is called “master”; your first commit will be on this branch. Starting October 1, 2020, this will be called “main” on GitHub.)

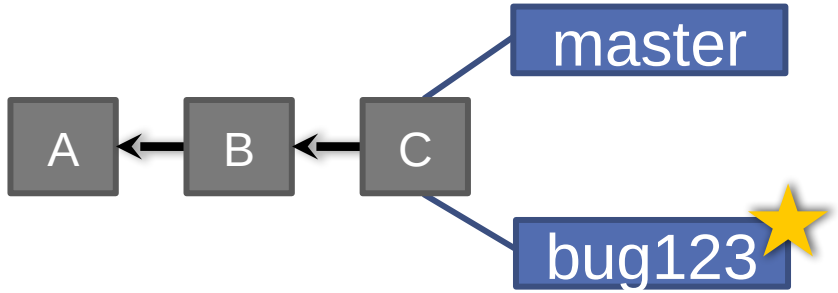
```
> git commit –m ‘my first commit’
```

BRANCHES ILLUSTRATED



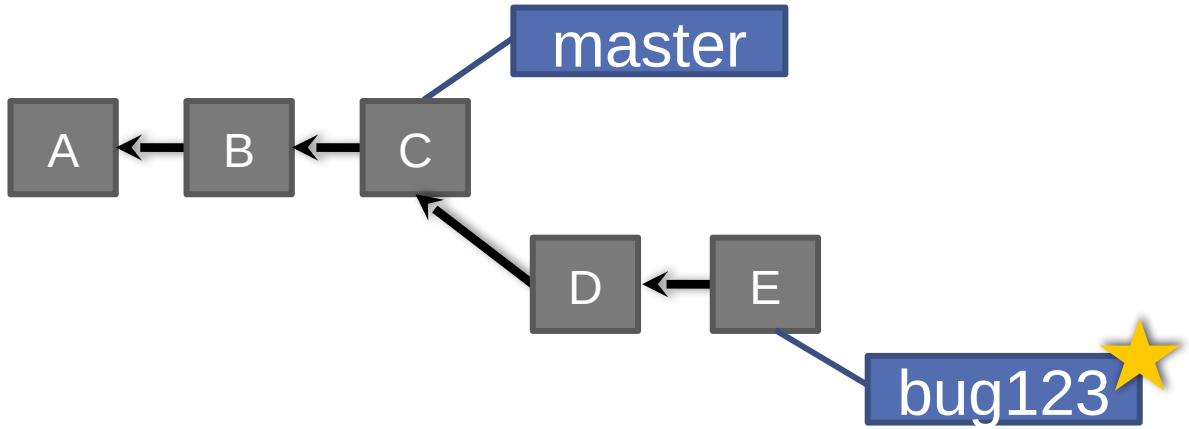

```
> git commit (x2)
```

BRANCHES ILLUSTRATED



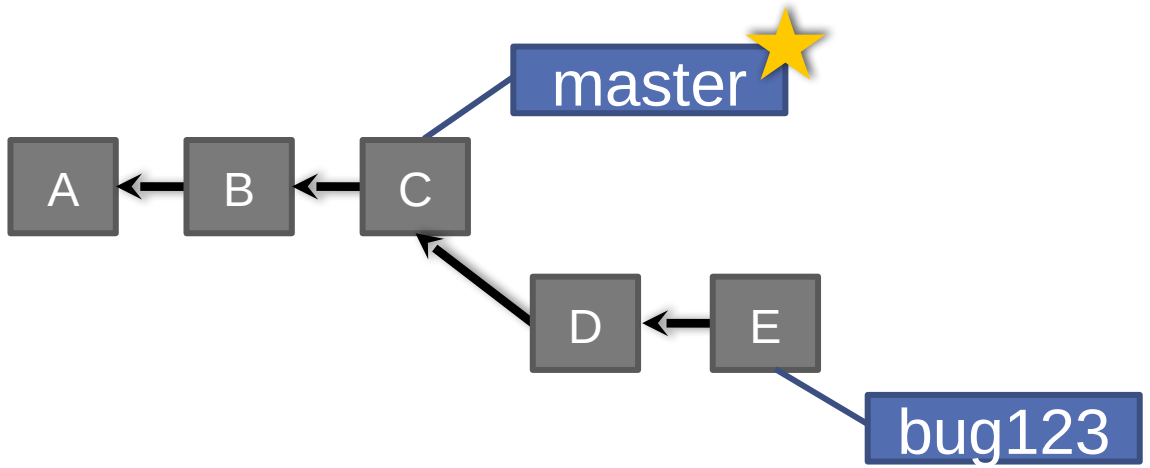

```
> git checkout -b bug123
```

BRANCHES ILLUSTRATED



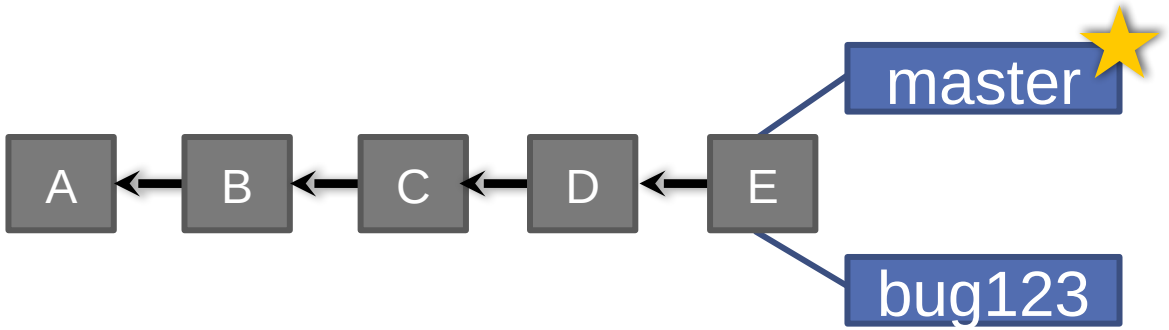

```
> git commit (x2)
```

BRANCHES ILLUSTRATED



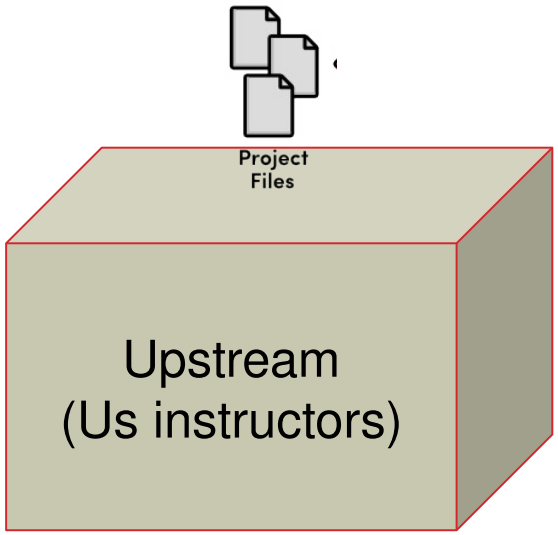

```
> git checkout master
```

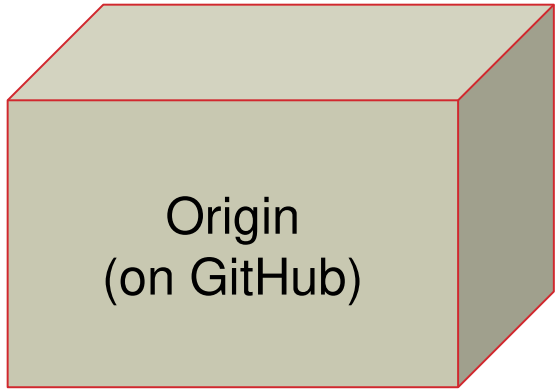
BRANCHES ILLUSTRATED

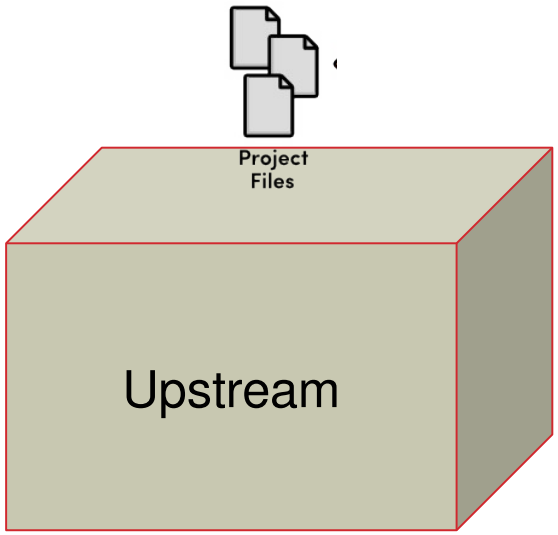


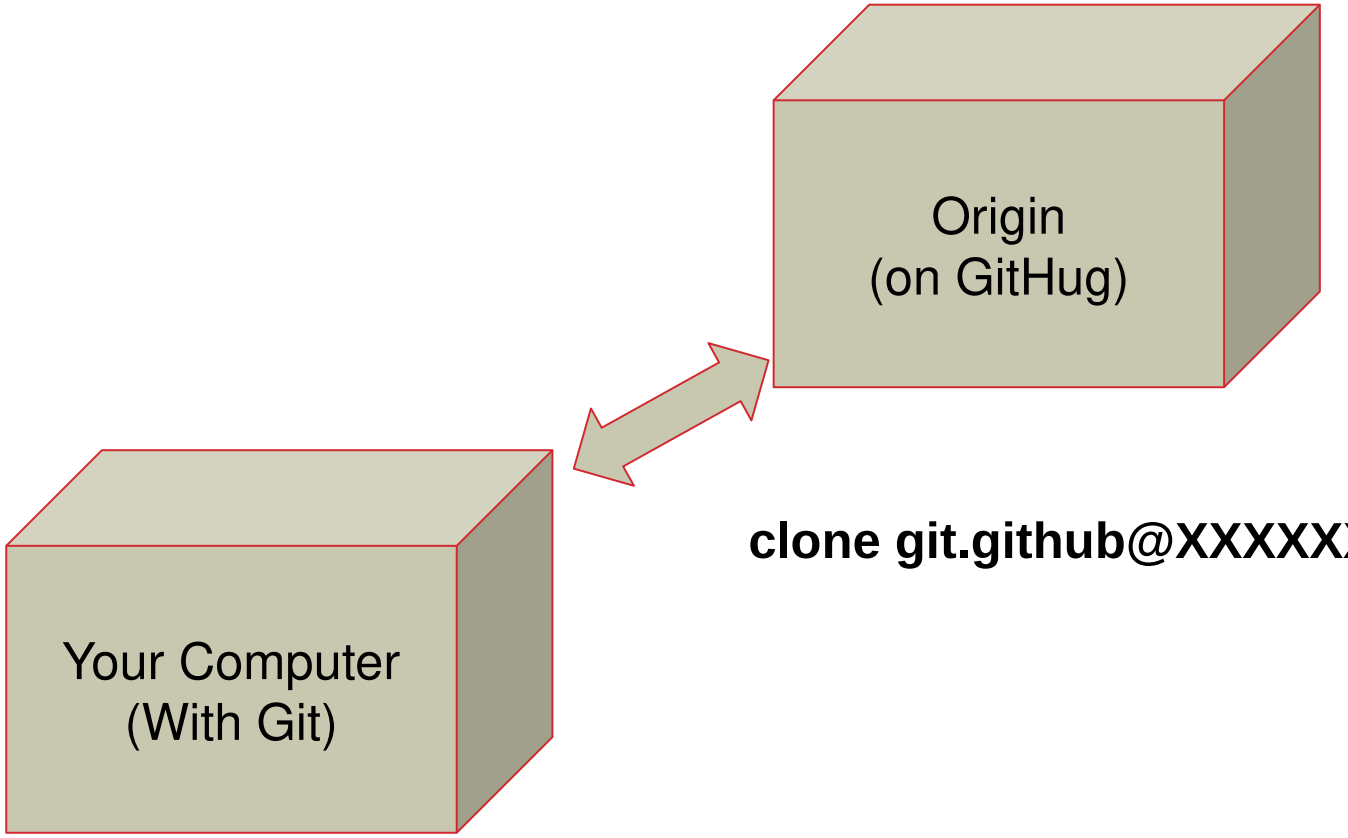

```
> git merge bug123
```

Typical Git Version Control

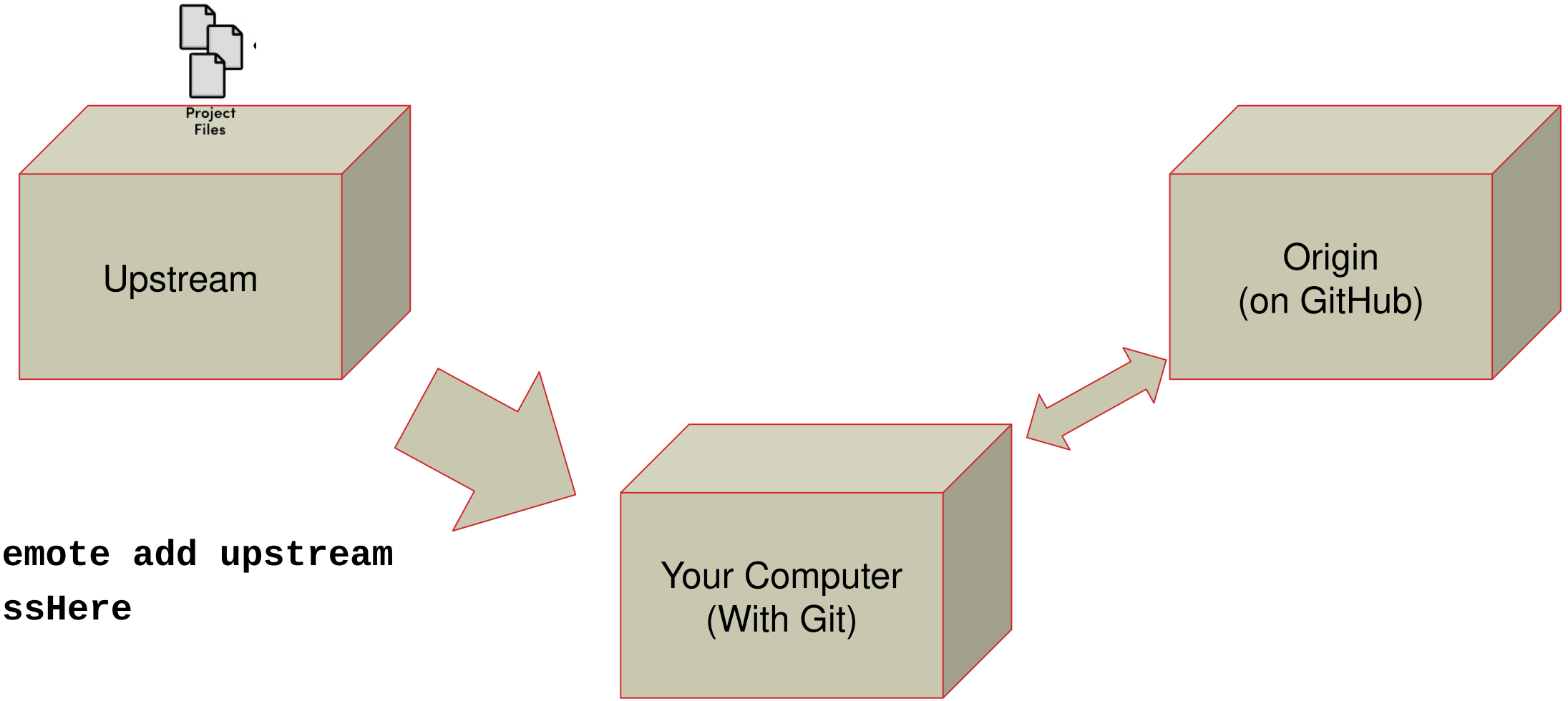




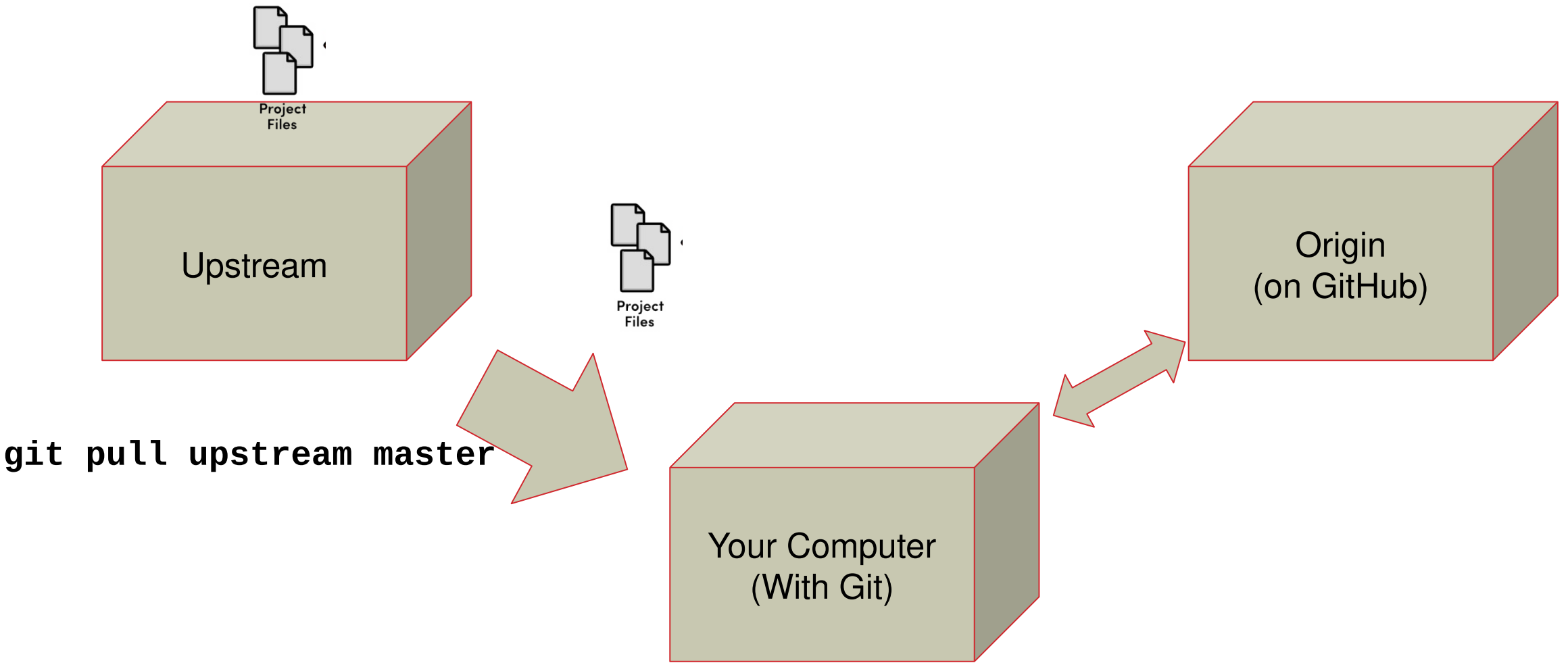


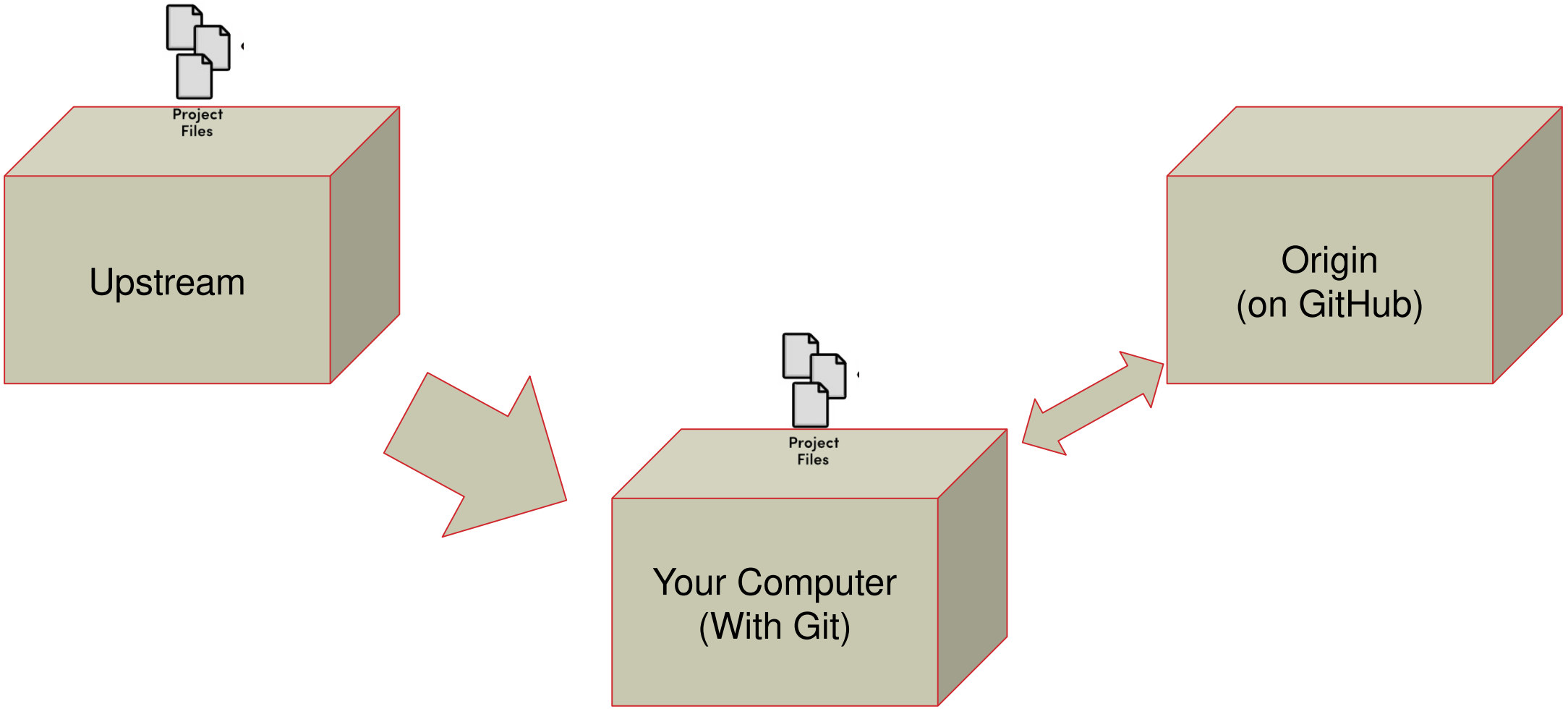


clone git.github@XXXXXXX

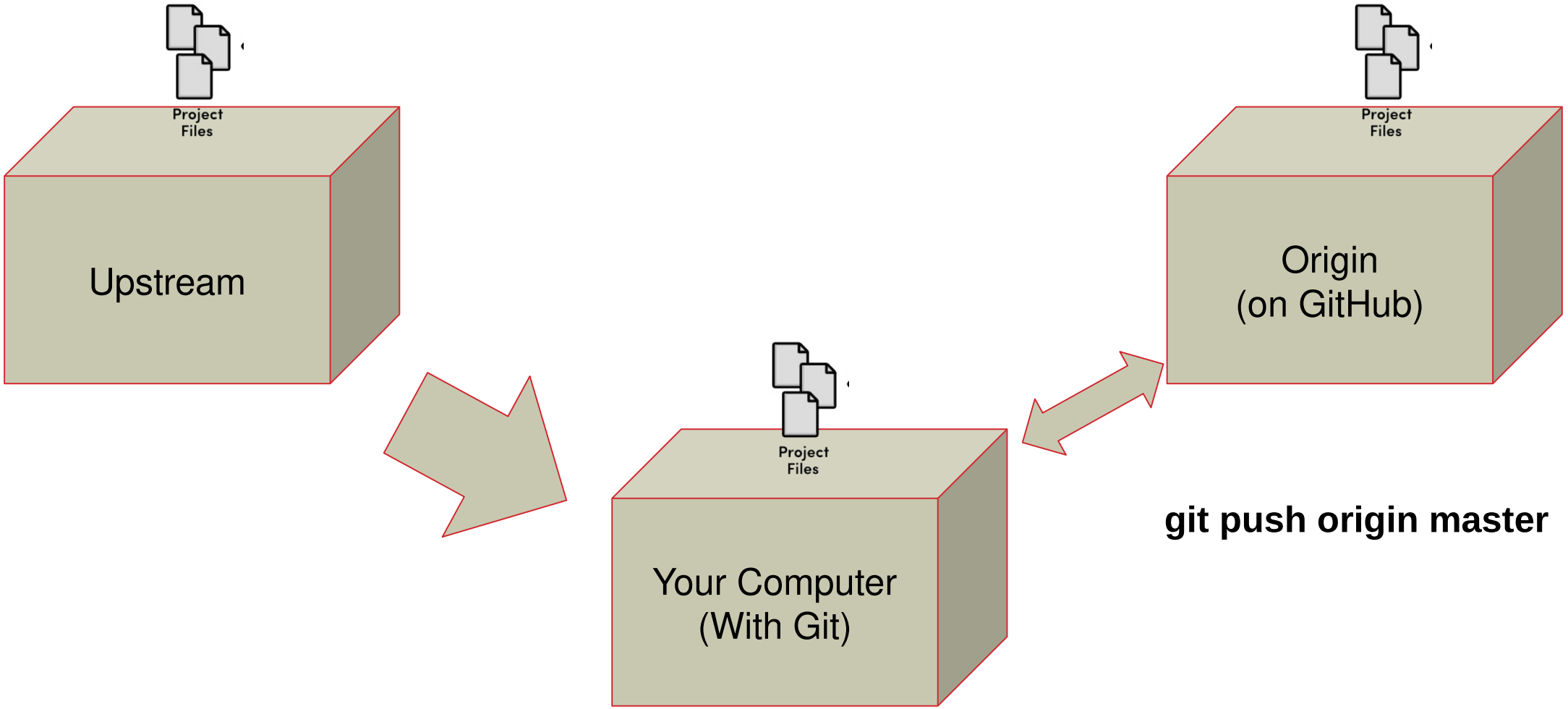


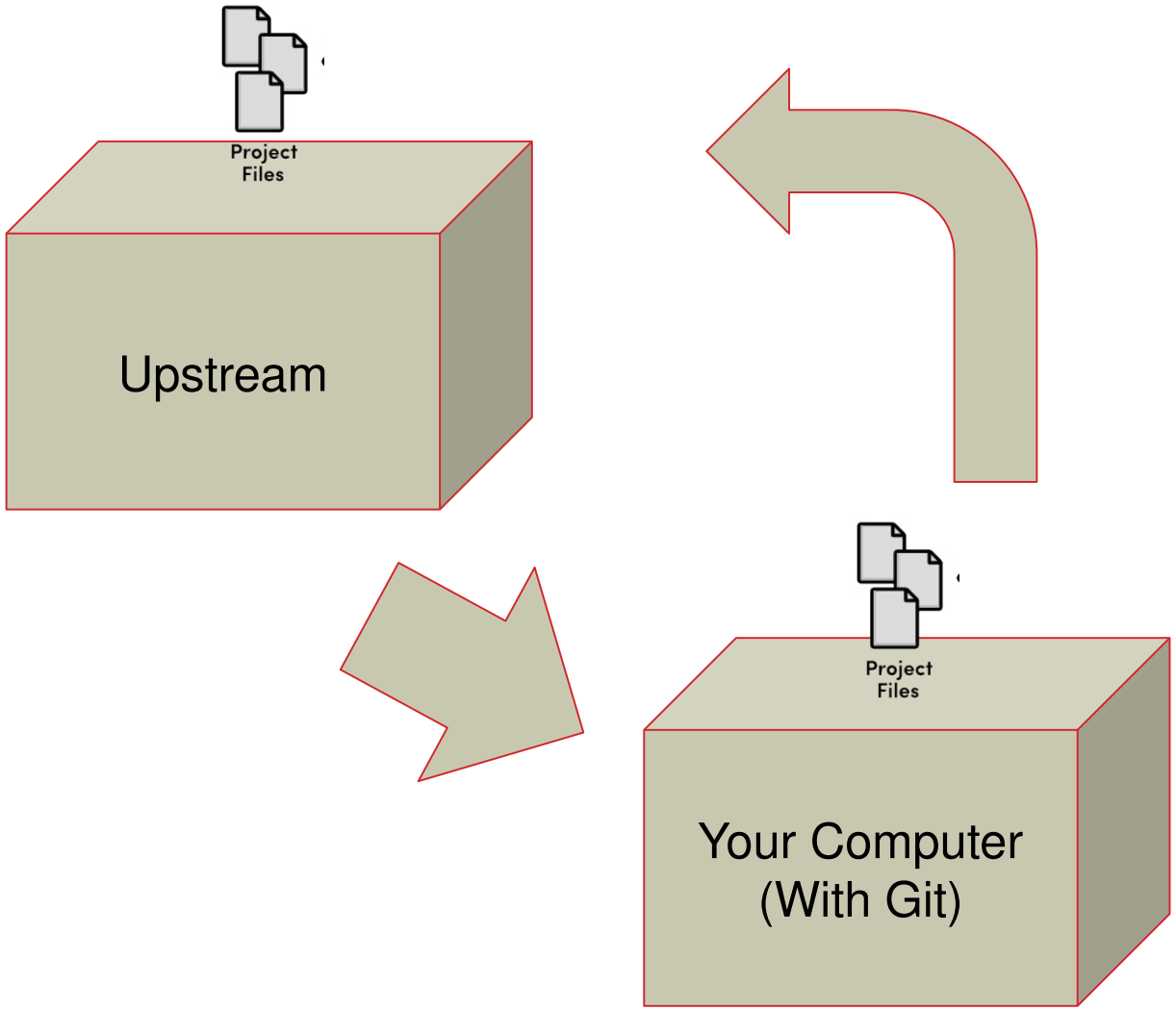

```
git remote add upstream
addressHere
```


Project Files
Upstream
Project Files
Origin (on GitHub)
git pull upstream master
Your Computer (With Git)



Make your changes!


Project
Files
Upstream
Project
Files
Your Computer
(With Git)
Project
Files
Origin
(on GitHub)
git push origin master


Project
Files
Upstream
Project
Files
Your Computer
(With Git)

WHEN TO BRANCH?

General rule of thumb:

- **Anything in the master branch is always deployable.**

Local branching is very lightweight!

- New feature? Branch!
- Experiment that you won't ever deploy? Branch!

Good habits:

- Name your branch something descriptive (`add-like-button`, `refactor-jobs`, `create-ai-singularity`)
- Make your commit messages descriptive, too!

SO YOU WANT SOMEBODY ELSE TO HOST THIS FOR YOU …

Git: general distributed version control system

GitHub / BitBucket / GitLab / …: hosting services for git repositories

In general, GitHub is the most popular:

Lots of big projects (e.g., Python, Bootstrap, Angular, D3, node, Django, Visual Studio)

Lots of ridiculously awesome/“awesome” projects (e.g., https://github.com/maxbbraun/trump2cash)

There are reasons to use the competitors (e.g., private repositories, access control)

REVIEW: HOW TO USE

Git commands for everyday usage are relatively simple

- **git pull**
  - Get the latest changes to the code
- **git add .**
  - Add any newly created files to the repository for tracking
- **git add –u**
  - Remove any deleted files from tracking and the repository
- **git commit –m ‘Changes’**
  - Make a version of changes you have made
- **git push**
  - Deploy the latest changes to the central repository

Make a repo on GitHub and clone it to your machine:

- https://guides.github.com/activities/hello-world/

STUFF TO CLICK ON

Git

- http://git-scm.com/

GitHub

- https://github.com/
- https://guides.github.com/activities/hello-world/
- **^-- Just do this one. You may need it for your tutorial ☺.**

GitLab

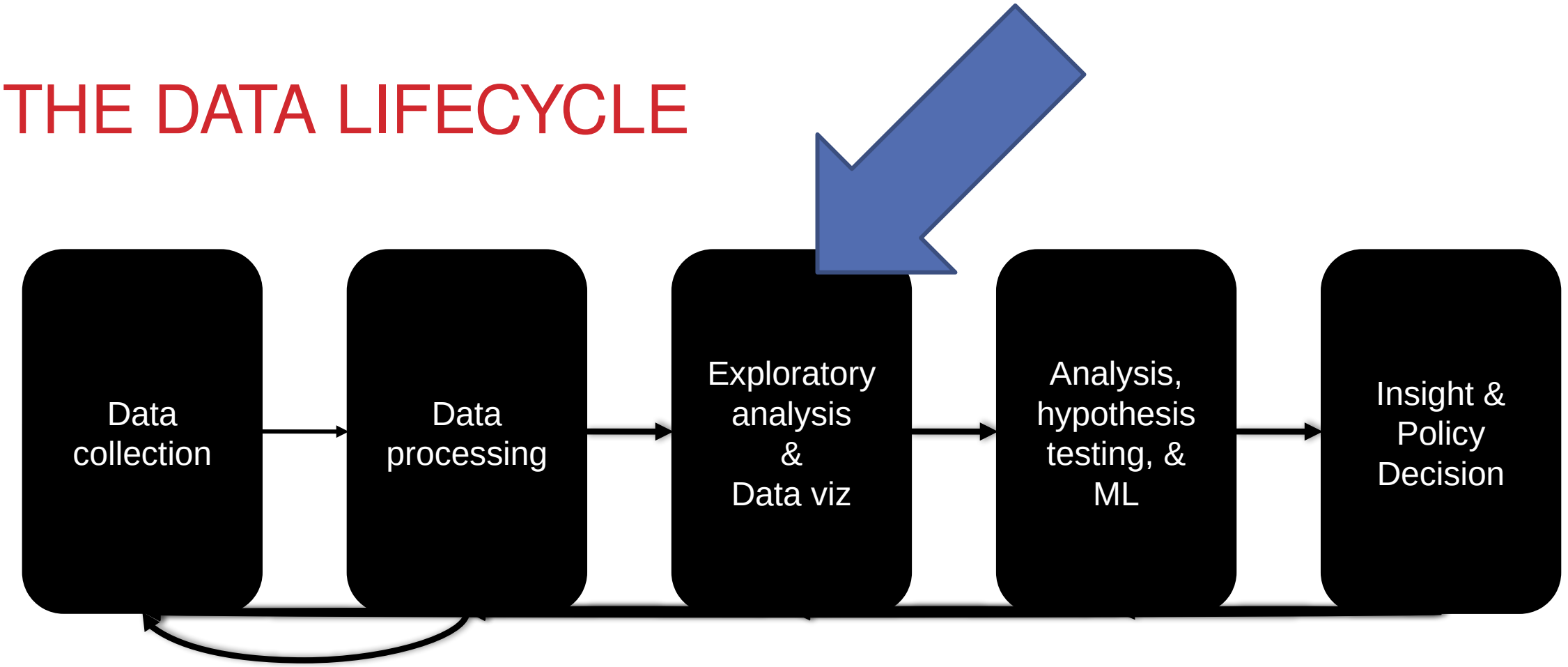
- http://gitlab.org/
- https://gitlab.cs.umd.edu/

Git and SVN Comparison

- https://git.wiki.kernel.org/index.php/GitSvnComparison

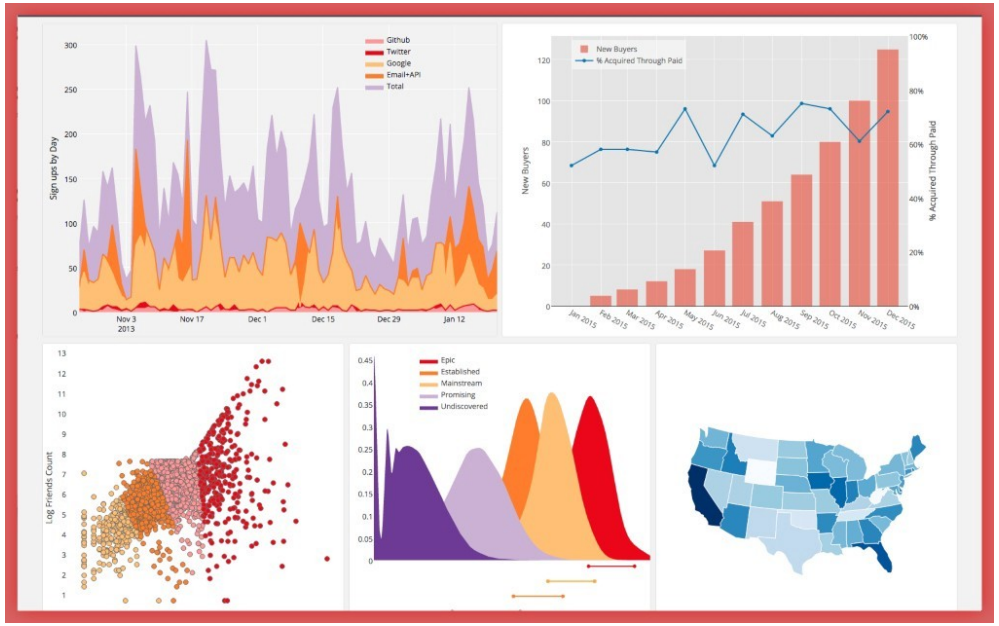
Plotting

THE DATA LIFECYCLE



Visualization

Source



Two primary purposes:

- **To *explore* our data**
- **To *communicate* our data**

There are a million different ways to visualize our data, but how do we do it effectively?

Matplotlib

Included with our Docker image and through Anaconda

```
# Installing Independently
python -m pip install matplotlib
```

Import into your notebook using:

```
# Importing, don't forget to use numpy too!
from matplotlib import pyplot as plt
```

Developed in 2002 by John Hunter to have Matlab-style plotting from the python command line.

Core Matplotlib

- **Bar Charts**
- **Line Charts**
- **Scatterplots**

Core Matplotlib

Matplotlib use a notion of a 'Figure' as its main abstraction. This abstraction is presented as an object (in the object-oriented sense), so manipulating your figures and plots means interacting with your Figure object (and others, as we'll see). You can create a figure with the plt.subplots() function.

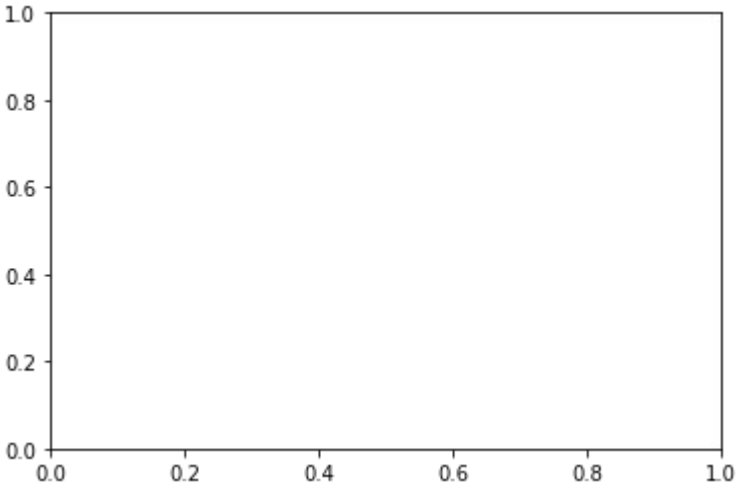
For our purposes today, we just need to know that with plt.subplots() we can create a single figure with one or more plots. If we pass it no arguments, we get a Figure with a single subplot, otherwise we can provide it with the dimensions of how we would like our subplots to divide the figure.

Example

```
# Create some x values that would give us two periods of a standard
sin wave
xs = np.arange(0,4*np.pi,0.1)

# y values for the standard sin
sin1 = np.sin(xs)

# y values for an amplified sin
sin2 = 2 * np.sin(xs)

# y values for an attenuated sin
sin3 = 0.5 * np.sin(xs)

# y values for a squashed sin
sin4 = np.sin(2*xs)
```

On to the Plotting

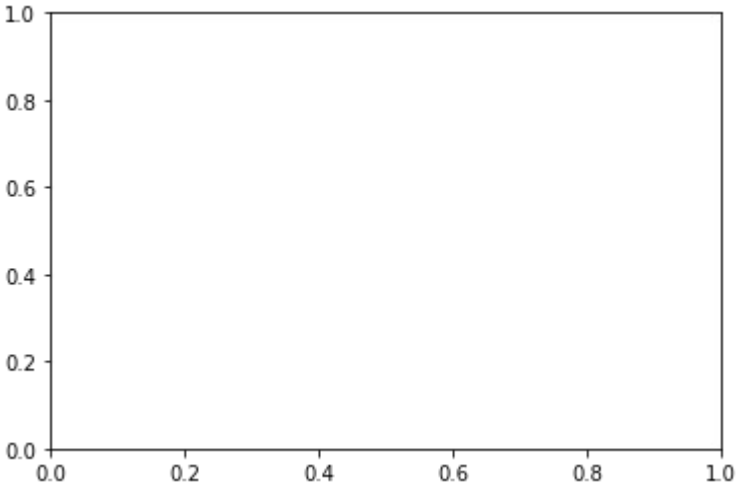
```
## create a single figure with a single subplot:
# fig <- The figure that contains all of our subplots
# ax  <- the subplot (if there's only one)
#        or the list of subplots (if there's more than
one)
fig, ax = plt.subplots()
```



???

On to the Plotting

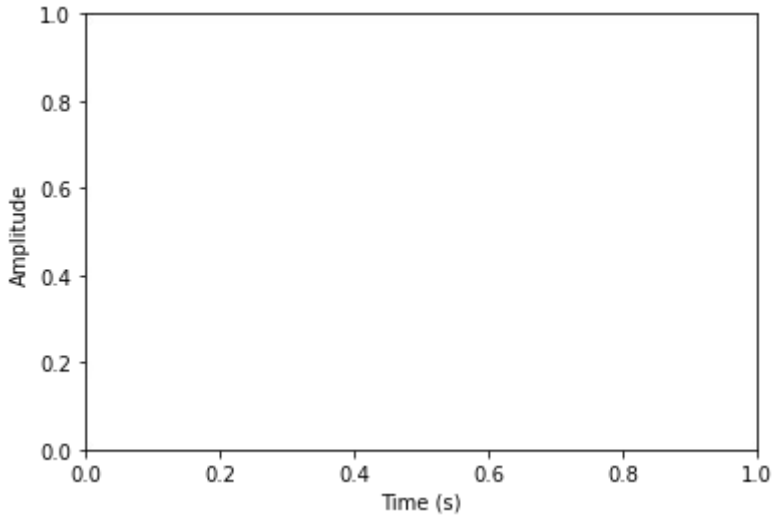
```
## create a single figure with a single subplot:
# fig <- The figure that contains all of our subplots
# ax  <- the subplot (if there's only one)
#        or the list of subplots (if there's more than
one)
fig, ax = plt.subplots()
```



???

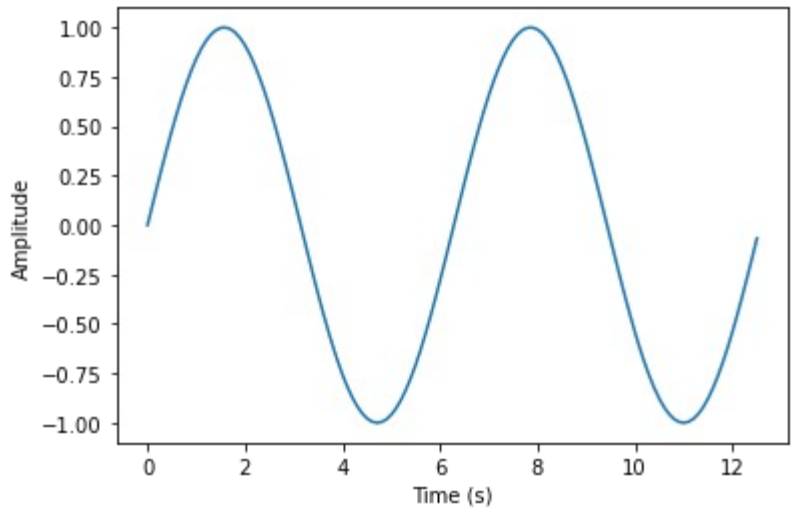
And my Ax[e]

```
#Setting up my labels
ax.set_xlabel('Time (s)')
ax.set_ylabel('Amplitude')
fig
```

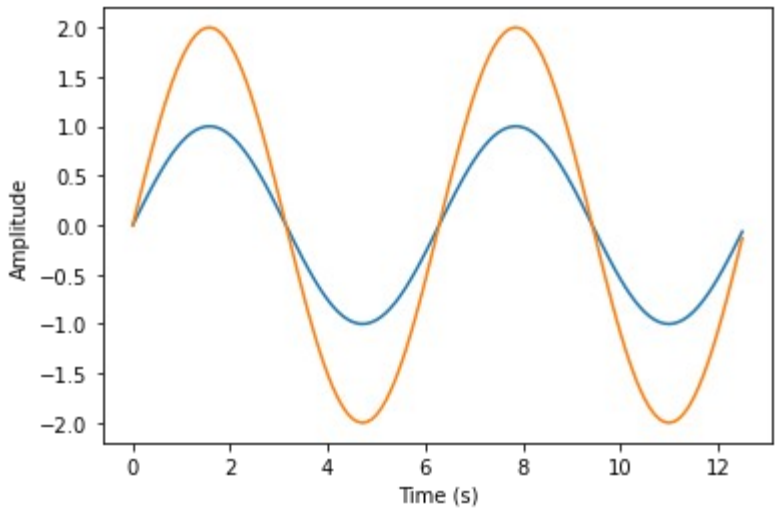


Finally!

```
#Setting up my labels
ax.plot(xs,sin1)
fig
```

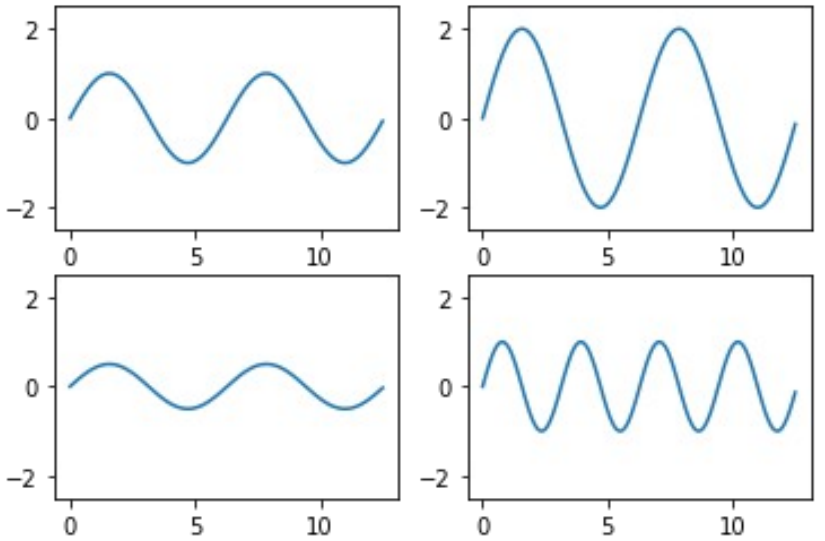



```
#Setting up my labels
ax.plot(xs,sin2)
fig
```



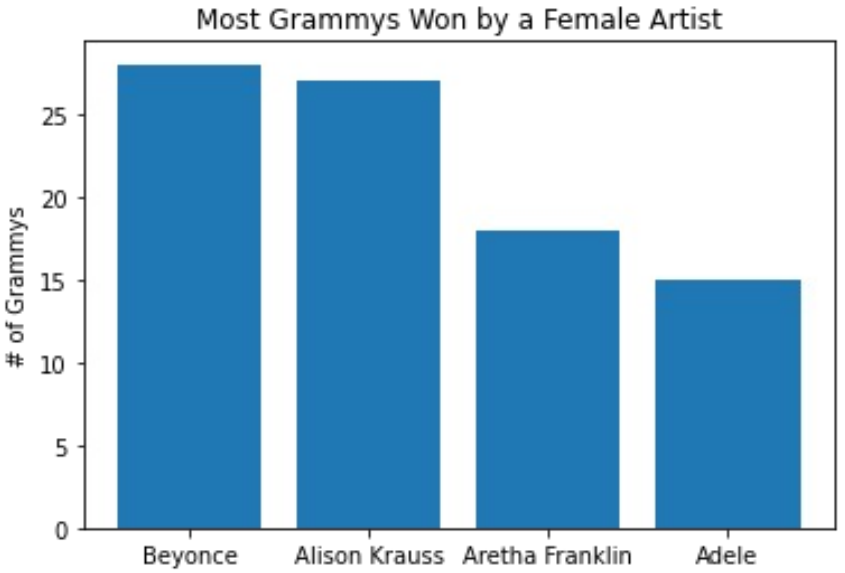
Plotting on Separate Subplots

```
#Setting up my labels
data_set = [sin1, sin2, sin3, sin4]

fig, axes = plt.subplots(nrows=num_rows,
ncols=num_cols)

#Plotting using a loop
i = 0
for x in range(0,num_rows):
    for y in range (0, num_cols):
        axes[x,y].plot(xs,data_set[i])
        i += 1
#Setting Axis Limits
for ax in axes.reshape(-1):
    ax.set_ylim([-2.5,2.5])
```



Bar Charts

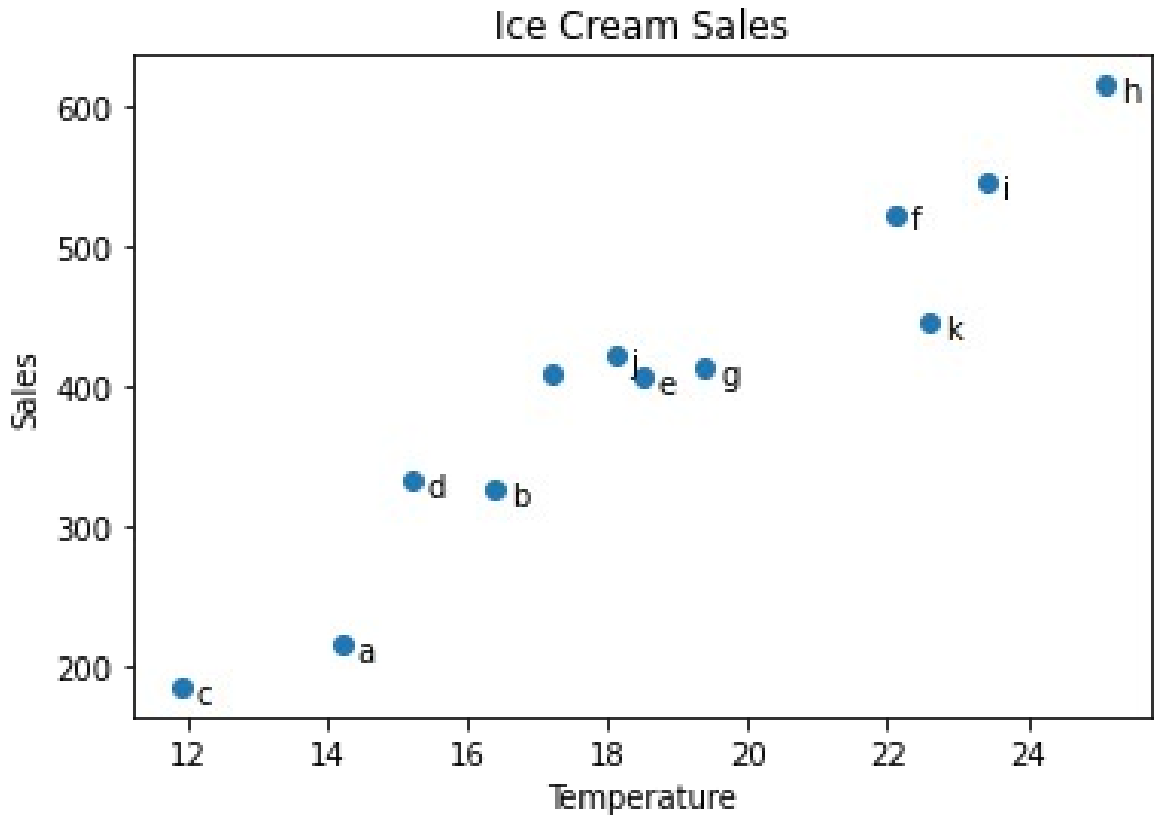
```
#Setting up my Data
artists = ["Beyonce", "Alison Krauss", "Aretha
Franklin", "Adele"]
num_grammys = [28, 27, 18, 15]

plt.bar(range(len(artists)), num_grammys)

#Plotting the bar graph
plt.title("Most Grammys Won by a Female Artist")
plt.ylabel("# of Grammys")
plt.xticks(range(len(artists)), artists)
plt.show()
```



Scatterplots

```
#Setting up my labels
ice_cream_sales = [215, 325, 185, 332, 406, 522, 412, 614, 544,
421, 445, 408]

temp = [14.2, 16.4, 11.9, 15.2, 18.5, 22.1, 19.4, 25.1, 23.4, 18.1,
22.6 17.2]
labels = [‘a’,’b’,’c’,’d’,’e’,’f’,’g’,’h’,’i’,’j’,’k’]

plt.scatter(temp, ice_cream_sales)
#Labeling using a loop
for label, temp_count, sales_count in zip(labels, temp,
ice_cream_sales):
    plt.annotate(label,
            xy=(temp_count, sales_count),
            xytext=(5, -5),
            textcoords=’offset points’)
plt.show
```

Scatterplots



MORE COMPLICATED EXAMPLE

```
# Formatting
df["week"] = df['week'].str.extract('(\d+)',
                                    expand=False).astype(int)
df["rank"] = df["rank"].astype(int)
```

```
[…, "x2nd.week", 63.0]    […, 2, 63]
```

```
# Cleaning out unnecessary rows
df = df.dropna()

# Create "date" columns
df['date'] = pd.to_datetime(
                df['date.entered']) +
                pd.to_timedelta(df['week'], unit='w') -
                pd.DateOffset(weeks=1)
```

NEXT CLASS:

EXPLORATORY ANALYSIS