Financial Software Engineering
Lecture 1

Co-Pierre Georg
AIFMRM

06 July 2018

Introduction

- 5 Classes + 5 tutorials (details in timetable)
- Marks: 10% class participation; 10% tut participation; 30% exam; 20% group project; 30% develop tutorial;

What we will cover

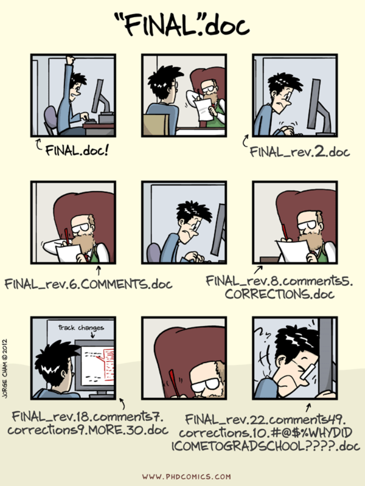
- Version control $\rightarrow$ git and collaborative coding
- Unix Shell and Bash
- Python and object oriented programming
- Front-end web development with HTML, CSS and Bootstrap

Today

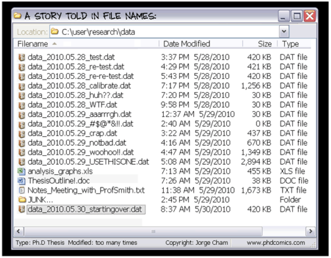
1. Version control, Git & Github
2. Unix Shell and Bash
3. Python 101
4. Python packages

Version control, Git & GitHub

What is version control?



Source: PhD Comics



Source: PhD Comics

What is version control?

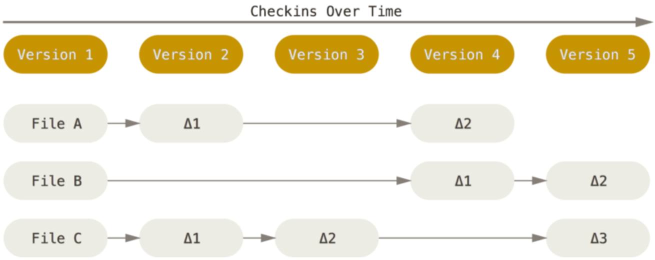
- As important as any codebase itself, is the development of the codebase over time
- What features were added when, who added them, why they were added ...
- We call this process of managing the development of a body of code, **version control** → designed to protect you from yourself
- In it's most rudimentary form, this means having multiple files as our project develops: `my_project_v1.py`, `my_project_v2.py` → `my_project_v100.py`
- This way we can easily revert to previous versions of the codebase if months into the development cycle a specific addition no longer works out

Why is version control important?

- `my_project_v1.py`, `my_project_v2.py` → `my_project_v100.py`
- → This structure is messy, impractical, unorganized and almost always inconsistent across different people
- To better standardize this process, we use a version control system (VCS)
- → a utility designed to implement a consistent framework of version control
- Manages changes to a project <u>without</u> overwriting any part of it
- While many VCS exist, the most popular remains **Git** → offers a snapshot based implementation of VCS

Typical VCS

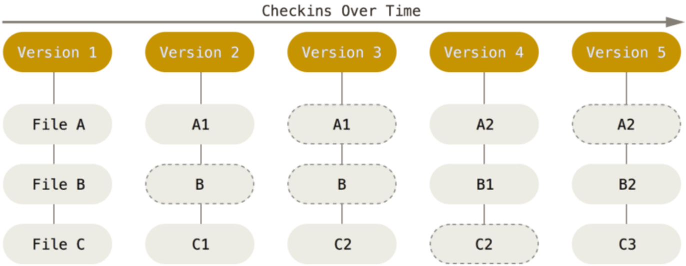
- File based → store information as a set of files and save each file only when changes are made to that file



Source: git website

Git VCS

- Snapshot based → saves a snapshot of the entire system of files once a change is made



Source: git website

Git

- Developed by the Linux development community and released in 2005
- Manages changes to a project <u>without</u> overwriting any part of it
- Important to distinguish between Git and GitHub $\rightarrow$ Git being the software underlying GitHub
- $\rightarrow$ Other implementations of Git include BitBucket, Gogs etc.
- Git's major advantages include - feature branching, distributed development, pull request and community

Getting started with Git

- We install Git and use it to create a local repository (repo) of our development workspace
- **Workspace** → a collection of files associated with a project
- **Repo** → a collection of files associated with a project <u>and</u> a record of all edits to the project
- This local repo stores the master copy of our codebase and all prior versions of it
- When we edit files and save them, we do so in our workspace
- Once happy with the changes we add and commit them to the local repo
- Git lives between the workspace and the repo keeping track of any changes between the two

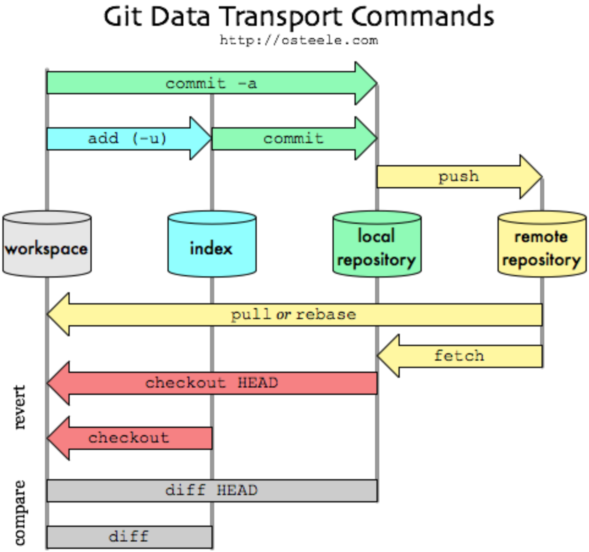
Git and Github

- Importantly however, Git represents a command-line tool which helps us implement version control locally
- Working collaboratively in teams however requires a global version of Git, accessible to all collaborators, where edits and changes made by all team members can be managed
- While many different versions of these implementations of Git exist, the most popular implementation is GitHub
- GitHub is a platform designed to host Git repositories
- Unlike with the local implementation of Git, GitHub hosts a remote repo which now acts as the master version of the codebase
- Github helps to implement version control between developers' own local repo and the projects' remote repo

Git and Github

- The combination of *workspace*, *local repo* and *remote repo* provide a convenient and robust structure to guide and manage development
- Edits are made and saved to the local workspace $\rightarrow$ successful edits are then *added* and *committed* to the local repo where these edits are recorded $\rightarrow$ all collaborators can then *push* the edits made in their local repositories to the remote repo

Git visualized



Source: Git data transport commands

Getting started with Git and GitHub

- To begin interacting with GitHub we have two options
- → Creating a new repo to start adding, committing and pushing files and edits to files to the repo
- → Download or import an existing repo and begin adding, committing, pushing and making pull requests to the repo
- Let's define these commands
- **Add**: stages all files in the workspace to be committed to the local repo
- **Commit**: updates local repo with edits in the staging area and records these changes
- **Push**: updates remote repo with edits made in the local repo

Getting started with Git and GitHub

- When starting new projects, we'll begin by creating a remote repo on GitHub
- We'll then use Git in the command line to *clone* this remote repo
- **Clone**: create a local repo from a remote repo
- We can now go ahead and begin making changes in our workspace, adding and committing these to the local repo and pushing these changes to the remote repo
- We can then invite collaborators who can also clone the remote repo

From cloning to forking

- Importantly, cloning is only available to the owner of a remote repo and other collaborators
- Without this restriction, any person could begin editing any public repo on GitHub
- In some cases, we may be interested in interacting with a remote repo but we are neither an owner, nor invited to join as a collaborator
- $\rightarrow$ finding a useful application on Github and wanting to import this to our local machine to use or contribute to
- In these scenarios we'll make use of the *fork* command

From cloning to forking

- **Fork**: create a local repo from a remote repo without any push functionality
- Forking allows various developers to interact with a codebase without being able to alter the master codebase on the remote repo → think large open source projects
- Forking however allows developers to share or suggest changes to the owners or maintainer of the remote repo by way of a *pull request*
- **Pull request**: ask maintainer of remote repo to pull changes from a local repo into the remote repo
- The maintainer of the repo is notified of the pull request and if they like the changes made, they can pull these changes and update the remote repo

From cloning to forking

- In this way, forking represents one of the key functions of collaborative and open source development
- By opening up your code base to others and allowing them to suggest functionality or improvements, you can leverage off skills in the development community
- Pull requests then help ensure that the integrity of your codebase is intact $\rightarrow$ while others can suggest changes, you decide whether to incorporate these changes

Git workflows

- The next decision you'll be faced with is choosing a workflow
- $\rightarrow$ recommendation or process for how to use Git to accomplish work in a consistent and productive manner
- Workflows dictate how groups of people collaborate using Git by setting the rules and guidelines that structure this collaboration
- The three main workflows include
  - Forking
  - Centralized
  - Feature branching

Forking workflow

- The most used workflow for public and open source projects
- Developers work in their private repo's, submit pull requests and a project maintainer manages changes to the main codebase
- The project maintainer therefore controls write privileges to the codebase
- $\rightarrow$ clear assignment of roles in a team and enforces quality control when it comes to changes to the main codebase

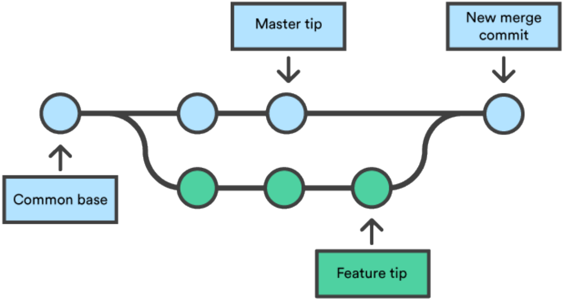
Centralized workflow

- As opposed to the forking workflow, a centralized workflow has no project maintainer
- Instead, each team member is a collaborator, clones the master remote repo locally and can push changes to the codebase directly
- Importantly, a centralized workflow requires that every collaborator's local repo matches the remote repo before any changes can be pushed
- People A and B clone a repo $\rightarrow$ A makes changes and pushes $\rightarrow$ If B wants to push changes after A has pushed, B must first pull the changes made by A
- This ensures that collaborators are able to test their additions against the most current version of the codebase
- $\rightarrow$ ensures that all pushes to the codebase do not break existing functionality

Branching workflow

- Splits the development workflow into branches or independent lines of development
- The main codebase becomes the `master` branch
- Contributors can then create separate branches which mirror the `master` branch and make edits and changes without changing the `master` branch
- Importantly, these additional branches are all stored on the remote repository as well
- Code can also be deployed directly from the new branch
- The the new feature in the branch is successful, it can be merged back into the `master` branch
- If the new feature is not to be incorporated into `master`, everything remains unchanged

Branching visualized



Source: Atlassian

Issues

- The final piece of functionality we'll discuss are issues
- GitHub's native tool to keep track of tasks, improvements and additions, and bugs
- Users can report bugs or project members can list improvements and to-dos
- Each issues has it's own comment thread $\rightarrow$ promotes collaboration and discussion
- Issues can be labelled and attached to a milestone and assigned to a team member
- See the GitHub guide for more

Project structure

- Once your repo is setup, it has to be populated
- More importantly, this repo needs to be structured
- While there are many ways to do this, all project repo's will need certain components

Project structure

```
project
        |- .git
        |- README.md
        |- LICENSE.txt
        |- requirements.txt
        |- documentation
                |- doc1.docx
                |- doc2.pdf
        |- src
                |- myscript.py
        |- tests
                |- testscript.py
        |- setup.py
```

Project structure

- **.git** represents your git structure files
- **requirements.txt** specifies the dependencies or packages required
- **documentation** contains more detailed usage instructions and tutorials/walkthroughs
- **src** contains the source code, or contents of your project
- **tests** contains scripts designed to test your code
- **setup.py** is called to run your project/application

README

- Acts as the 'welcome page' to your project
- A well-written README can affect how users perceive your project and thus usage
- Typically, a README includes
  - A description of your project
  - An installation and usage guide
  - Troubleshooting
  - Acknowledgements
- Some tips here and here and a template here

LICENSE

- Includes full license text and copyright claims $\rightarrow$ NB part of your repo
- Governs how others may use your code
- Without specifying a license, default copyright laws apply $\rightarrow$ you own all rights and no-one may reproduce or share your code
- For open source projects we therefore want to be explicit about the licensing
- Typically, one would use the GNU General Public License or the MIT license
- Choose your license carefully! For help see choosealicense

Git and GitHub resources

- While, we've touched on Git briefly, there is a plethora of online resources
- Some free resources
- $\rightarrow$ Github's official guide
- $\rightarrow$ Atlassian's Git tutorial
- $\rightarrow$ Udemy

Bash Shell

Introduction to the Bash Shell

- So far, we've used the terminal to execute certain commands
- Adding, commiting and pushing in Git and using commands like ls, cd and mkdir to view, navigate and create in our workspace
- The terminal itself however simply represents a way to interact with a PC's macro processor
- We call this macro processor, the **shell**
- We'll focus on the most widely used shell, Bash, or the "Bourne Again Shell"
- $\rightarrow$ default shell for most GNU/Linux distributions
- We can think of the bash shell as representing a command language interpreter for interacting with GNU operating systems

Why would we use the Shell?

- A shell's primary purpose is to read commands and run other programs
- The latter functionality is particularly useful $\rightarrow$ think running .py or .R scripts
- While we could execute these scripts or programs in our specific IDE's, we could use the shell to run these scripts directly
- We could also then begin writing scripts to execute in the shell which fulfill a range of functions, call a variety of scripts in different languages ...
- Importantly, when we begin interacting with servers, which don't have GUI's, we'll make use of the shell to execute all of our operations and actions

Shell scripting

- For the most part, we'll use the shell to run single commands or scripts at a time
- In many cases however, we'll want to run a number of commands or scripts one after the other
- As opposed to issuing one command at a time, shell scripting refers to the activity of running an entire script of commands
- These shell scripts give us the functionality of a programming language and the utility of the shell
- In their most basic form, shell scripts assist us in calling a range of other scripts and actions from a single script in the command line
- These scripts can be called directly from the command-line or can be executed when specific events are triggered: on bootup, daily at 15:00, at log-in etc.

Writing shell scripts

- We can simply write our shell scripts in a text editor, save them with a `.sh` suffix and run these scripts from the terminal
- A first script would look something like this

```bash
#!/bin/bash
echo Hello, World
```

- The first line *#!/bin/bash* is an important one, called a **shebang**
- It tells the shell which program to use to interpret the script, in this case, bash
- Everything we could do in the terminal we can now write into a shell script and have it execute
- Since Bash is a full programming language, we can now execute actions in the terminal using conditional logic - for, if, while etc.

Permissions

- To run our first script, we'll have to first cover permissions
- On a Unix system, each file and directory is assigned access rights
- These can be read (`r`), write (`w`) and/or execute (`x`) $\rightarrow$ determines *who* can do *what*
- We can view the permissions of a file using the `ls - l` command followed by the file or directory name
- This returns something that looks like `-rwx-xr-x`
- $\rightarrow$ every 3 index positions represent access rights for different users in the following order: file owner; group owner; all other users

Permissions

- Typically, files we create won't have execute permissions
- To change the permissions of a file, we make use of the `chmod` command and pass it the desired access rights
- Specified access rights are represented as a combination of octals which represent binary

Octal	Binary	Access
0	000	`---`
1	001	`--x`
2	010	`-w-`
3	011	`-wx`
4	100	`r--`
5	101	`r-x`
6	110	`rw-`
7	111	`rwx`

Permissions

- In this way, we can represent `rwx-rwx-rwx` as 777
- Before we run a shell script we've created we always check permissions and change them if necessary
- To change the permissions of a specific script

```
chmod 777 ./hello_world.sh
```

- Now, to run this script

```
./hello_world.sh
```

Secure shell

- In many applications, you won't be able to run commands locally
- Possibly due to performance reasons, privacy, company guidelines etc.
- In these cases, you'll make use of a server
- Unlike local machines, servers do not make use of a GUI $\rightarrow$ navigating the server, creating files, running scripts all need to be executed via the terminal
- To connect to a server and interact with the shell we'll make use of Secure shell
- Secure shell or **SSH** $\rightarrow$ UNIX-based command interface and protocol for securely getting access to a remote computer or server

Secure shell

- SSH includes three utilities: `slogin`, `ssh`, and `scp`
- `slogin` and `ssh` $\rightarrow$ secure login utilities
- `scp` $\rightarrow$ file transfer utility
- SSH commands are encrypted and secure $\rightarrow$ client/server connections are both authenticated using a digital certificate and passwords are encrypted
- We use SSH to connect to a remote server securely, upload files and scripts and issue commands on the shell
- In this course, we'll use SSH to connect to the AIFMRM servers and run tasks
- You'll cover how to do this in the first tutorial

Python 101

Python

- Simple & powerful programming language
- From the Python website - *"Python is powerful ... and fast; plays well with others; runs everywhere; is friendly and easy to learn; is Open."*
- High level portable language
- Interpreted as opposed to a compiled language
- Extensive packages
- We will be using Python 3 for this course
- Creator Guido van Rossum, named it after the BBC show *Monty Python's Flying Circus*

Variables

- Integers

```
myInteger = 2
```

- Strings

```
myString = 'Hello'
myString[0]
'H'
```

- Booleans

```
myBool = True
```

Control flow

- If/else

```
if i==1:
    word='one'
elif i==2:
    word='two'
else:
    word='big'
```

- For

```
for item in myList:
    print item
```

- While

```
while (count < 5):
    print count
    count = count + 1
```

Data structures

- List
  - Stores an ordered collection of items
- Tuple
  - Similar to lists minus some functionality and with immutability
- Sets
  - Stores an unordered collection of unique items
- Dictionary
  - Unordered mapping for storing objects using key-value pairs

List

- Stores an ordered collection of items

```
myList = [1, 2]
myList[0]
1


myList.append(3)
myList = myList + [4]
myList
[1, 2, 3, 4, 5, 6]


myList[5] = 0 #mutability
myList
[1, 2, 3, 4, 5, 0]
```

Tuple

- Similar to lists minus some functionality and with immutability

```
myTuple = (1, 2)
myTuple[0]
1


myTuple.append(3) #immutability
AttributeError: 'tuple' object has no attribute 'append'


myTuple + [3] #immutability
TypeError: can only concatenate tuple (not "list") to tuple
```

Sets

- Stores an unordered collection of unique items

```
mySet = {1,2,3}
mySet
{1, 2, 3}


myList = [1,1,1,1,2,2,2,3,3,3]
myNewSet = set(myList)
myNewSet # returns the unique items
{1, 2, 3}


myNewSet[1] #unordered, so cannot index
TypeError: 'set' object does not support indexing


myNewSet.add(4)
myNewSet
{1, 2, 3, 4}
```

Dictionary

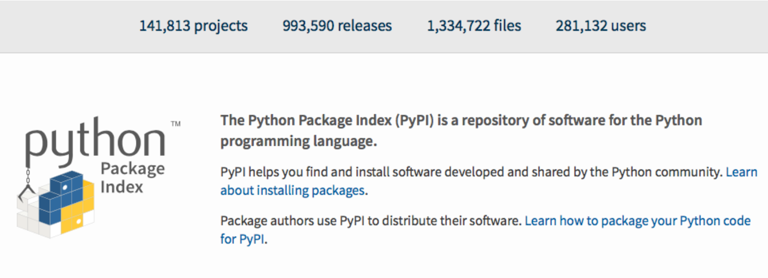
- Unordered mapping for storing objects using key-value pairs

```
price_dict = {'milk':12, 'apples':8}
price_dict['milk']
12
price_dict = {'milk':12, 'apples':{'green':9, 'red':8}}
price_dict['apples']['green'] #nested dictionaries
9
```

Python packages

PyPI: Python packages

- The Python Package Index → repo of third-party Python packages
- Similar to CRAN for R, Packagist for PHP etc.
- Packages can be installed using `pip install`
- `pip` is an easy way to download packages directly from PyPI



- Let's look briefly at 2 of the most used packages, `numpy` and `pandas`

Numpy

- The core package for scientific computing
- Major feature $\rightarrow$ large multidimensionality array objects and tools for working with these objects
- Facilitates element-wise operations
- Important since Python cannot do calculations on lists

```
first_list = [20, 40, 60]
first_list/5
TypeError: unsupported operand type(s) for /: 'list' and 'int'


second_list = [5, 10, 20]
first_list/second_list
TypeError: unsupported operand type(s) for /: 'list' and 'list'
```

Numpy

- Now, using Numpy

```
import numpy as np


np_first_list = np.array(first_list)
np_first_list/5
array([ 4.,  8., 12.])


np_second_list = np.array(second_list)
np_first_list/np_second_list
array([4., 4., 3.])
```

Numpy

- Note: Numpy arrays can only contain items of one type

```
np.array([True, 5, "Hello, World"])
array(['True', '5', 'Hello, World'], dtype='<U21')
```

- Numpy also facilitates subsetting

```
first_list > 20
array([False,  True,  True])


np_first_list[np_first_list > 20]
array([40, 60])
```

Pandas

- Provides data structures for working with relational, tabular and labeled data
- Think SQL tables, Excel spreadsheet, matrices with row and column labels ...
- Provides two primary data structures, the 1 dimensional **series** and the 2 dimensional **dataframe**
- Built on Numpy - but importantly allows for <u>multiple</u> data types
- Any user familiar with the R dataframe structure, will immediately feel comfortable with the pandas dataframe

Pandas

- Allows us to create dataframes from dictionaries

```
import pandas as pd

my_dict = {
"country": ["South Africa", "Namibia", "Botswana"],
"population": [56, 2.5, 2.25] }

country_dataset = pd.DataFrame(my_dict)
country_dataset.index = ["RSA", "NM", "BW"] #adding row labels
country_dataset

         country population
RSA South Africa      56.00
NM   Namibia           2.50
BW   Botswana          2.25
```

Pandas

- Allows us to import data

```
import pandas as pd

country_dataset = pd.read_csv("country_pop.csv", index_col = 0)
country_dataset

         country population
RSA South Africa      56.00
NM  Namibia            2.50
BW  Botswana           2.25
```

Other useful packages

- Scipy - functions typically used in mathematical, scientific and engineering applications (more on this in the tutorial)
- Matplotlib and Seaborn - 2-dimensional plotting
- Bokeh and Plotly - interactive visualizations
- SciKit-Learn - machine learning
- Scrapy - web scraping
- Statsmodels - statistical models and techniques

Resources to get started with this weekend

- Data camp's (free) intro to Python, here
- Codeacademy's (free) introduction to Python, here
- Python basic's are assumed knowledge
- We'll jump into more practical examples and how to set your PCs up for Python development on Monday