Computing for Economists

Software Engineering and Version Control

Toren Fronsdal and Ruby Zhang

Harvard University

Introduction

Introduction

The goal of this first module is to introduce:

- Programming paradigms focusing on Object-Oriented Programming
- General programming principles to follow
- The importance of version control

Table of Contents

Programming Paradigms

Programming Paradigms

Imperative Programming: The programmer provides a sequence of statements that change a program's state.

- "Attach marshmallow to stick, put stick near fire, ..."
- Examples: Python, R, C++, Stata

Declarative Programming: The programmer declares "what" should be achieved rather than specifying "how" to achieve it.

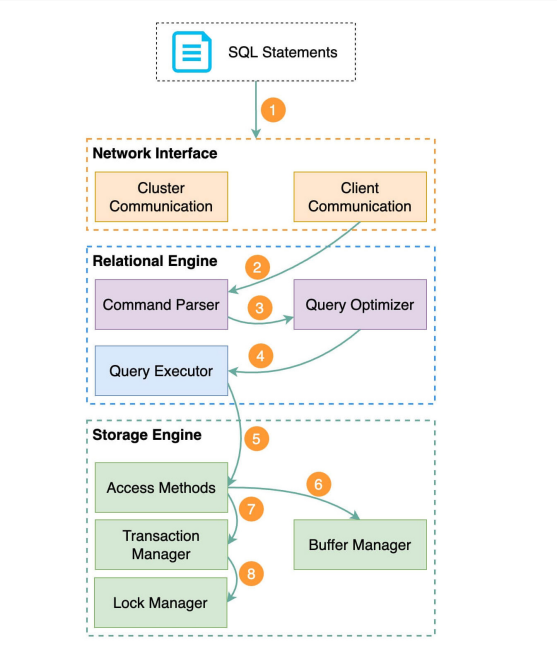
- "Make a golden-brown s'more"
- Examples: SQL (for database queries), HTML/CSS (for web design)

Declarative Example - SQL

```
SELECT *
FROM countries
WHERE country IN ('United States', 'Canada', 'Mexico')
```

Tells SQL **what** you want (all rows that match condition) but not **how** data should be retrieved (how to search, which indexes to use, or the order in which it should access the data).

The database system's query optimizer handles these implementation details.



Imperative Programming

Procedural Programming: Break down task into a collection of variables, data structures, and subroutines.

- Define a sequence of steps to be executed in a specific order
- Stata is a procedural language

Object-Oriented Programming: Break down a task into objects that expose behavior (methods) and data (attributes) using interfaces.

- Python is a multi-paradigm language, but main focus is OOP

Object-Oriented Programming (OOP)

- OOP is a programming paradigm centered around **objects**
- **Classes** define blueprints for creating objects
- **Objects** are *instances* of a class
  - Can have multiple objects of the same class type
  - Objects encapsulate both data (attributes) and behavior (methods)
- OOP is useful for the same reason that abstraction is useful: for recognizing and exploiting the common structures
- Key concepts: **encapsulation**, **inheritance**, and **polymorphism**

Example of a Class

For a given product, suppose we are given its demand $Q(p)$, fixed cost fc, and marginal cost mc. Then, for a given price p, we can then find the total cost as

$$C(p) = fc + mc \cdot Q(p)$$

and firm profits as

$$\pi(p) = p \cdot Q(p) - C(p).$$

We can think about the product as an *object*:

- the scalars fc and mc and the function $Q(p)$ are its attributes
- $C(p)$ and $\pi(p)$ are its methods

Example of a Class

```
class EconomicModel:
    def __init__(self, demand_fn, fixed_cost, marginal_cost):
        # constructor sets up instance variables
        self.Q = demand_fn
        self.fc = fixed_cost
        self.mc = marginal_cost

    def total_cost(self, price):
        # method to compute total costs
        total_cost = self.fc + self.mc * self.Q(price)
        return total_cost

    def profits(self, price):
        # method to compute profits
        profits = price * self.Q(price) - self.total_cost(price)
        return profits

# demand function Q(P)
demand_fn = lambda p: 200 - 10p

# creating an instance of the EconomicModel class
model = EconomicModel(demand_fn, fixed_costs=200, marginal_costs=5)

# call method
model.profits(price=10)
>> 300

# get attribute
model.mc
>> 5
```

Encapsulation

Encapsulation is the principle of bundling **data (attributes)** and the **methods (functions)** that operate on that data into a single unit (an object or class).

- Hides internal object implementation details
- Exposes well-defined interface for interacting with objects
- In the *`EconomicModel`* example, can obtain profits by only knowing price

Inheritance

Inheritance is a mechanism that allows a new class to inherit properties and behaviors (attributes and methods) from an existing class.

- The existing class is called a **superclass** or **base class**
  - Defines common attributes and methods shared by subclasses
- The new class is called a **subclass** or **derived class**
  - Can extend or specialize the functionality of the base class
- Superclasses and subclasses have an “is-a” relationship:
  - A subclass is a more specialized version of the superclass
  - Example: if you have a base class *`Animal`*, a subclass *`Dog`* is an *`Animal`*
  - Example: if you have a base class *`GLM`*, a subclass *`OLS`* is a specialization of *`GLM`*

Inheritance

```
class Model:
    """A statistical model."""

    def __init__(self, data):
        self.data = data

    def fit(self):
        # Placeholder method, to be overridden in derived classes
        pass

class GLM(Model):
    """Generalized Linear Model"""

    def __init__(self, data, link_function):
        self.link_function = link_function
        # Call the constructor of the base class using super()
        super().__init__(data)

    def fit(self, start_params=None, method='newton', ...)
        # Override the fit method for GLM
        ...
    ...

class OLS(GLM):
    """Ordinary Least Squares"""

    def __init__(self, data, ...):
        # OLS is a special case of GLM, with the identity link function
        super().__init__(data, link_function='identity')
    ...
```

Programming Principles

Programming Principles

- Decomposition
- Abstraction
- Reusability
- Documentation
- Testing

Decomposition

General principle:

- A good function does one conceptual thing
- You should know what the thing is from the function name

Code should be written **modularly** so it is

- self-contained
- broken into pieces
- able to be reused
- organized

Decomposition

```
# predict
t = beta @ X1
probs1 = 1 / (1 + np.exp(-t))
preds1 = probs1 > 0.5
# compute eval metrics
accuracy1 = np.mean(y1 == preds1)
tn, fp, fn, tp = confusion_matrix(y2, preds2).
      ravel()
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f1_score1 = 2 * precision * recall / (
      precision + recall)
print("Accuracy:", accuracy1)
print("F1 score:", f1_score1)

# predict
t = beta @ X2
probs2 = 1 / (1 + np.exp(-t))
preds2 = probs2 > 0.5
# compute eval metrics
accuracy2 = np.mean(y2 == preds2)
tn, fp, fn, tp = confusion_matrix(y2, preds2).
      ravel()
precision = tp / (tp + fp)
recall = tp / (tp + fn)
f1_score2 = 2 * precision * recall / (
      precision + recall)
print("Accuracy:", accuracy2)
print("F1-score:", f1_score2)
```

Not modular

```
sigmoid = lambda x: 1 / (1 + np.exp(-x))

def predict_prob(beta, X):
    return sigmoid(beta @ X)

def predict(beta, X):
    probs = predict_prob(beta, X)
    return probs > 0.5

def accuracy(y, preds):
    return np.mean(y == preds)

def f1_score(y, preds):
    tn, fp, fn, tp = confusion_matrix(y, preds
       ).ravel()
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    return 2 * precision * recall / (precision
        + recall)

def print_eval(y, preds):
    print("Accuracy:", accuracy(y, preds))
    print("F1-score:", f1_score(y, preds))

preds1 = predict(beta, X1)
print_eval(y1, preds1)

preds2 = predict(beta, X2)
print_eval(y2, preds2)
```

Modular

Abstraction

Abstraction involves simplifying complex systems by breaking them down into essential components and interactions while hiding unnecessary implementation details.

- For example, classes abstract real-world entities and their behaviors
- Think of it as looking at the bigger picture without getting lost in the minutiae

Guiding principle:

- Abstract to eliminate redundancy
- Abstract to improve clarity
- Otherwise, don’t abstract

Reusability

Your code should be structured such that it can be used in multiple contexts or projects.

Following the principles of abstraction and modularity allows for code reusability.

Why Code Reusability Matters

- **Time Efficiency**: Reusing existing code saves time compared to writing everything from scratch
- **Consistency**: Reused code promotes uniformity and consistent behavior across projects
- **Maintenance Ease**: Updates and bug fixes made in one place benefit all projects using that code
- **Reduced Errors**: Fewer chances of introducing new bugs when using well-tested code

Reusability

```
# Run OLS on first set of covariates
XtX_inv = np.linalg.inv(np.dot(X.T, X))
Xty = np.dot(X.T, y)
coefficient1 = np.dot(XtX_inv, Xty)
predictions1 = np.dot(X, coefficients1)

y_pred = predictions1
sst = np.sum((y - y.mean())**2)
sse = np.sum((y - y_pred)**2)
r_sq1 = 1 - (sse / sst)

# Run OLS on second set of covariates
XtX_inv = np.linalg.inv(np.dot(X2.T, X2))
Xty = np.dot(X2.T, y)
coefficient2 = np.dot(XtX_inv, Xty)
predictions2 = np.dot(X2, coefficients2)

y_pred = predictions2
sst = np.sum((y - y.mean())**2)
sse = np.sum((y - y_pred)**2)
r_sq2 = 1 - (sse / sst)

# Use a different covariate set to predict
predictions3 = np.dot(X3, coefficients2)

y_pred = predictions3
sst = np.sum((y - y.mean())**2)
sse = np.sum((y - y_pred)**2)
r_sq3 = 1 - (sse / sst)
```

Not reusable

```
class OLSRegression:
    def __init__(self):
        self.coeff = None

    def fit(self, X, y):
        # Calculate coefficients
        XtX_inv = np.linalg.inv(X.T @ X)
        Xty = np.dot(X.T, y)
        self.coeff = np.dot(XtX_inv, Xty)

    def predict(self, X):
        return np.dot(X, self.coeff)

    def r_squared(self, X, y):
        y_pred = self.predict(X)
        sst = np.sum((y - y.mean())**2)
        sse = np.sum((y - y_pred)**2)
        return 1 - (sse / sst)

# Run OLS on two covar sets and predict
model = OLSRegression()
model1.fit(X, y)
predictions1 = model1.predict(X)
r_sq1 = model1.r_squared(X, y)

model2.fit(X2, y)
predictions2 = model.predict(X2)
r_sq2 = model2.r_squared(X2, y)
r_sq3 = model2.r_squared(X3, y)
```

Reusable

Documentation

Code should be self-documenting via naming and organization, and the use of comments should not depend on changes in the input or outputs (e.g. “Make sure X is not negative!”). Keep documentation that is automatically revised as a result of code revision.

Guiding principles:

- Name variables, datasets, and functions in a way that is self-explanatory and descriptive
- Organize the file so code sections are read clearly
- Abstraction applies here: do not repeat information
- Use code to assert and check the data, do not comment

Documentation

```
* Load cleaned dataset
* The dataset is unique on county and year
* All wages should be positive, documents
      2005-2020
use "wages.dta", clear

* Winsorize outliers at P99, given by $350K
replace wage = 350000 if wage >= 350000

* Wage growth
sort county year

* Generate wage growth
* Make sure no time skips in the data
bys county: gen lag_wage = l.wage
gen wage_g= = (wage - lag_wage) / lag_wage
```

Not well-documented

```
* Load cleaned dataset
use "wage_by_county_year_2005_2020.dta", clear

*---------------
* Clean data
*---------------
isid county year
assert wage > 0
assert year >= 2005 & year <= 2020

* Winsorize outliers
winsor2 wage, cuts(0 99) replace

*---------------
* Construct wage growth
*---------------
sort county year

* Ensure no time skips
bys county: gen year_gap = year - l.year
assert year_gap <= 1

* Generate wage growth
bys county: gen lag_wage = l.wage
gen wage_growth_rate = (wage - lag_wage) /
      lag_wage
```

Well-documented

Docstrings

A documentation string (docstring) is a string that describes a module, function, class, or method definition.

At a minimum, docstrings should describe the function. Ideally, they should also describe the input parameters and return values and their types.

```
def add(num1, num2):
    """
    Add up two integer numbers.

    This function simply wraps the `+` operator, and does not
    do anything interesting, except for illustrating what
    the docstring of a very simple function looks like.

    Parameters
    ----------
    num1 : int
        First number to add.
    num2 : int
        Second number to add.

    Returns
    -------
    int
        The sum of `num1` and `num2`.
    """
    return num1 + num2
```

Testing

Testing should also be modular, done often, and for each block/function.

- **Unit Testing:** write code to test where possible instead of manually testing (package *`unittest`*)
- Ensure the output is as intended even if there's no error
- Debugging can be tedious, set breakpoints
- Python can handle exceptions with a *`try`* and *`except`* block when you want the code to keep running

Scale documentation and testing with importance of function

- Function often used $\rightarrow$ document and test extensively and well

Testing

Suppose we write the following simple functions to calculate equilibrium price and quantity for a given supply and demand curve.

```
def quantity_supplied(price, supply_slope, supply_intercept):
    return supply_slope * price + supply_intercept

def quantity_demanded(price, demand_slope, demand_intercept):
    return demand_slope * price + demand_intercept

def find_equilibrium(supply_slope, supply_intercept, demand_slope, demand_intercept):
    # Assuming linear functions, solve for price (P) where Qs = Qd
    equilibrium_price = (demand_intercept - supply_intercept) / (supply_slope - demand_slope)
    equilibrium_quantity = quantity_supplied(equilibrium_price, supply_slope, supply_intercept)
    return equilibrium_price, equilibrium_quantity
```

Testing

- If there's an error, which function is it?
- Even if there's no error, do we know if the result is correct?
- How can we manually inspect the code if running a whole script?

```
# Try an example and see what outputs are
equilibrium_price, equilibrium_quantity =
      find_equilibrium(2, 10, -3, 50)

print(equilibrium_price, equilibrium_quantity)
```

Bad testing

```
# Given supply and demand functions: Qs = 2P +
      10 and Qd = 50 - 3P
def test_quantity_supplied():
    assert quantity_supplied(10, 2, 10) == 30
    assert quantity_supplied(0, 2, 10) == 10

def test_quantity_demanded():
    assert quantity_demanded(10, -3, 50) == 20
    assert quantity_demanded(0, -3, 50) == 50

def test_find_equilibrium():
    supply_slope, supply_intercept = 2, 10
    demand_slope, demand_intercept = -3, 50
    equilibrium_price, equilibrium_quantity =
       find_equilibrium(supply_slope,
       supply_intercept, demand_slope,
       demand_intercept)

    assert equilibrium_price == 10
    assert equilibrium_quantity == 30

test_quantity_supplied()
test_quantity_demanded()
test_find_equilibrium()
```

Good testing

A Note on Style

Many languages have recommended sets of conventions (sometimes arbitrary) to ensure consistent code style.

Here are some resources on different coding style guides:

- **PEP 8** is the official code style guide for Python
- **Google's Style Guides** for Python, R, C, etc.

Python Tools:

- *`pycodestyle`* (formerly *`pep8`*): a tool to check Python code against some of the style conventions in PEP 8
- *`flake8`*: a tool that glues together *`pycodestyle`*, *`pyflakes`*, *`mccabe`* to check the style and quality of Python code
- *`black`*: automatically formats Python code in consistent style

Version Control

Why use a version control system?

Large projects will involve many files and collaborators. Version control is the management of code and file changes: the **who**, **when**, **why**, and **how**.

- Documentation and History Tracking
  - All edits of the code (and done by whom) are tracked throughout the entire process
- Transparency and Reproducibility
  - Easily compare and go back and forth between revision changes
  - Changes can be verified and reviewed
- Collaboration
  - Multiple collaborators can locally work on and experiment with different file versions at the same time, then merge the branches
- Organization
  - Projects rarely progress linearly, version control keeps track of not only files, but directories and the entire project structure

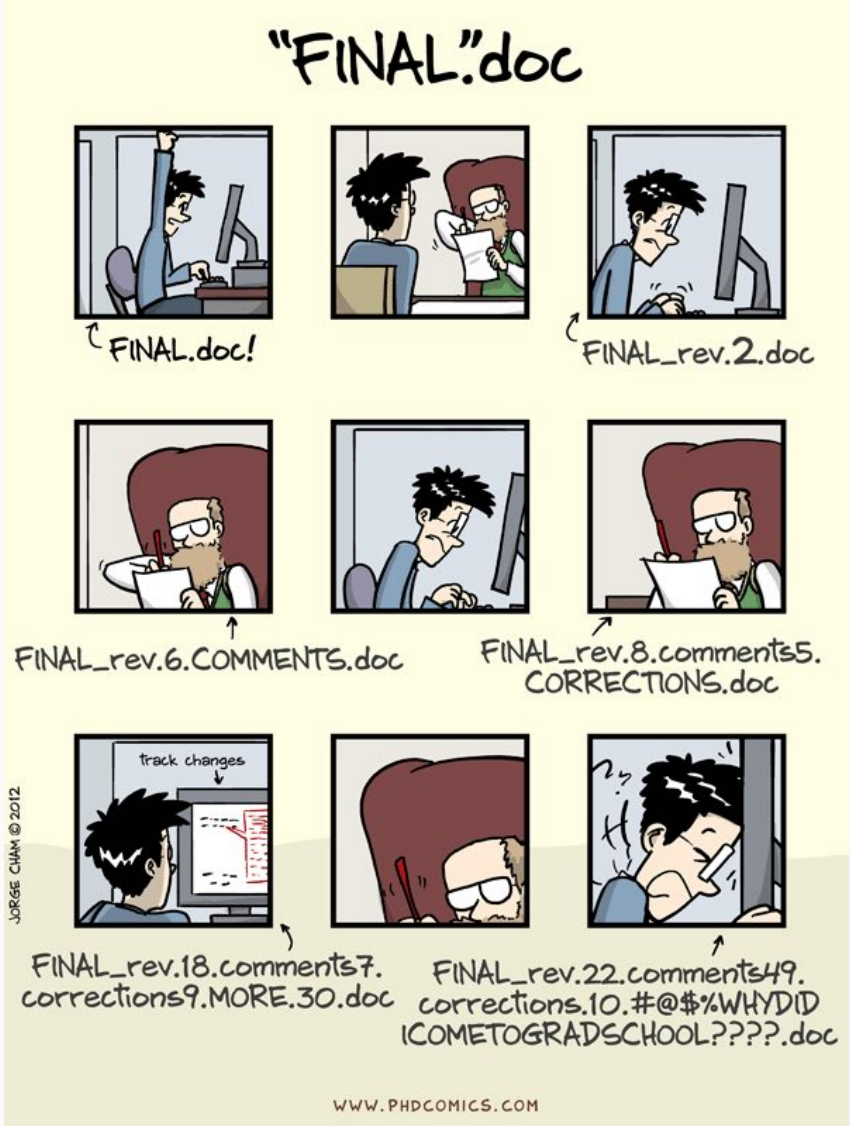
Version Control the Wrong Way

1. **Name and date:** Why is the date modified different between the file name and the computer? Who did which exact changes? Why do we need these changes? Don't do this.

```
reg_table_2023_01_23_v1_RZ.do
reg_table_2023_01_25_v2_RZ.do
reg_table_2023_01_26_v2_edit_TF.do
reg_table_2023_01_27_v3_RZ.do
reg_table_2023_01_28_v3_final_TF.do
reg_table_2023_01_30_v3_final_FINAL_TF.do
```

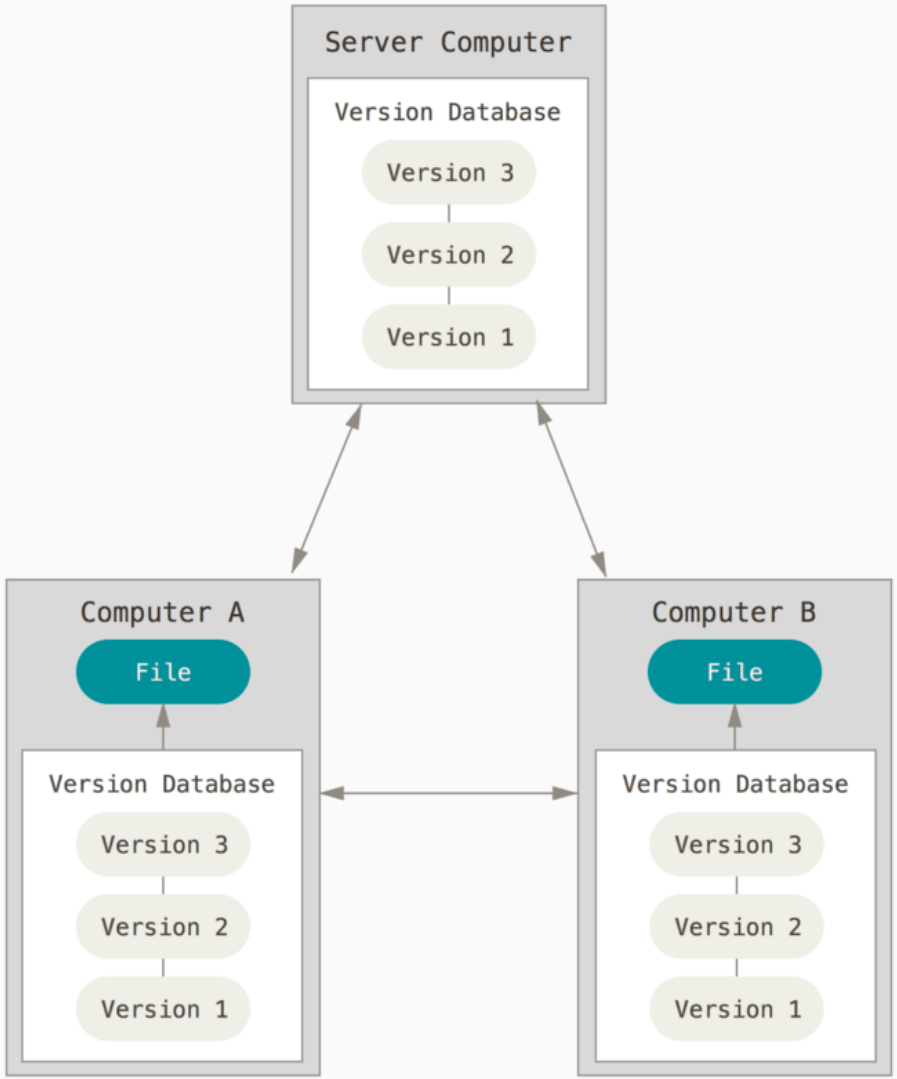
2. **Cloud Storage:** (e.g. Dropbox) Great for giving collaborators access to the same and latest files. Some people store large datasets or PDFs. There is a crude historical back-up that Dropbox stores, that's not enough for code repositories or Tex files!

- How do you see exact changes between new versions? No easy way to see this.
- How can collaborators experiment and create their own versions of the same file? You can't.


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.
CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.
corrections9.MORE.30.doc
FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

“Not one piece of commercial software you have on your PC, your phone, your tablet, your car, or any other modern computing device was written with the ‘date and initial’ method.” (Gentzkow and Shapiro)

Distributed Version Control



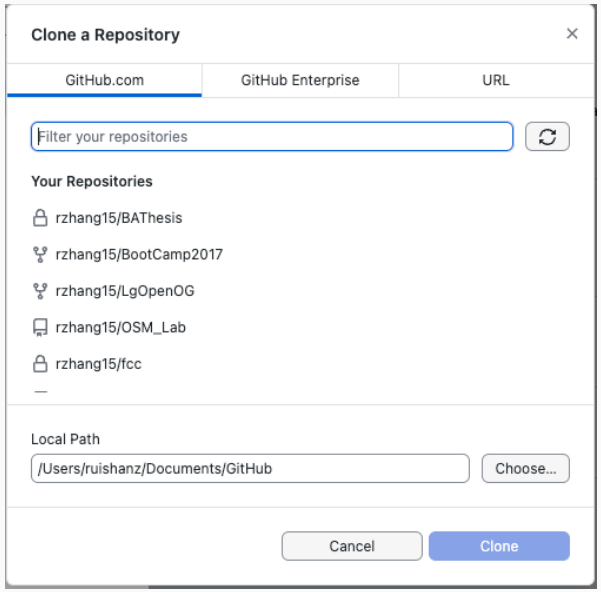
Git and GitHub

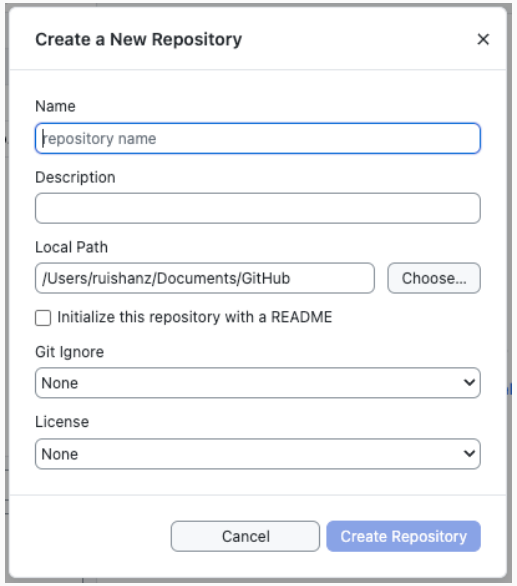
Git is a distributed version control system (DVCS) that allows developers to track changes in their codebase, collaborate with others, and efficiently manage software development projects.

GitHub is a web-based platform that provides hosting for Git repositories. One can use command line or the desktop app.

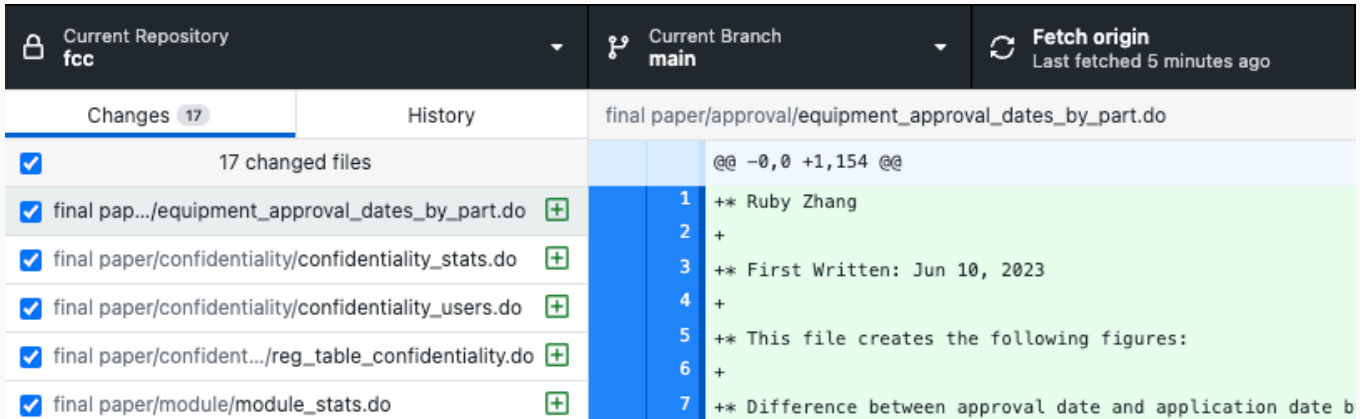
Getting Started with GitHub: Desktop App

1. Clone or create a new repository



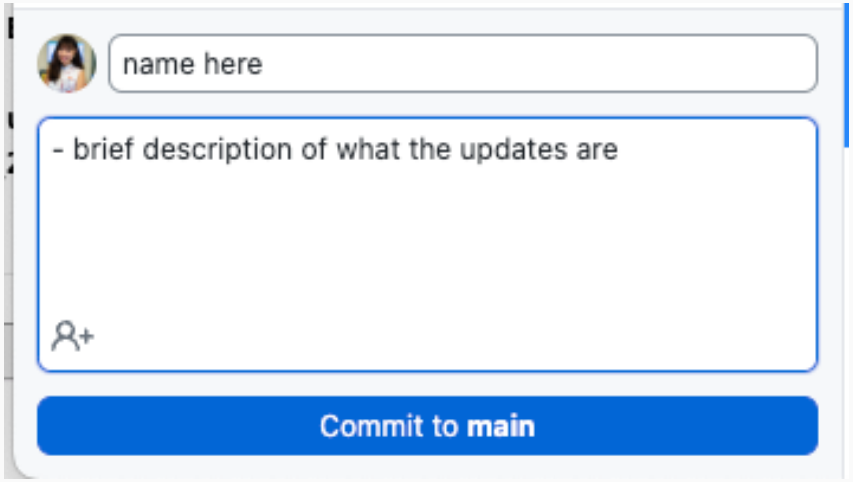


2. Modify files (automatically added to staging)

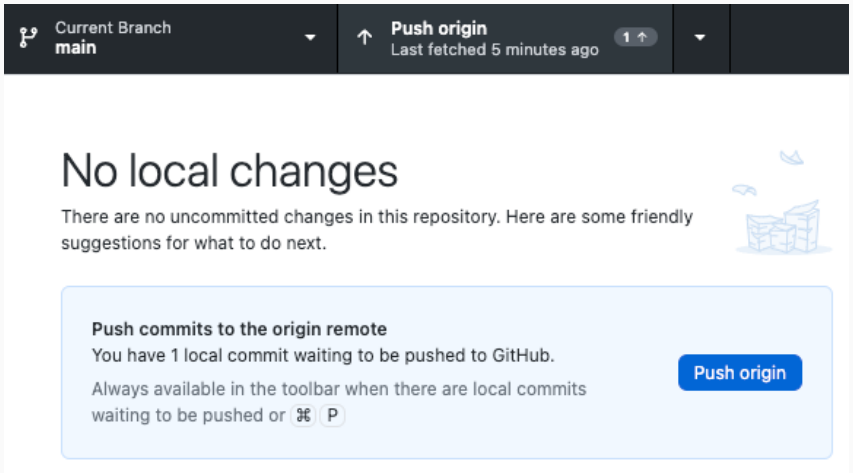


Getting Started with Git: Desktop App

3. Commit files



4. Push changes



Getting Started with GitHub: Terminal Commands

1. Navigate to place in directory you want to place repository

```
cd /home/user/projects
```

2. Clone repository

```
git clone https://github.com/user/my_repo
```

3. Modify files

4. Add files to staging

```
git add .
```

5. Commit files

```
git commit -m "description of changes"
```

6. Repeat 3-5

7. Check status

```
git status
```

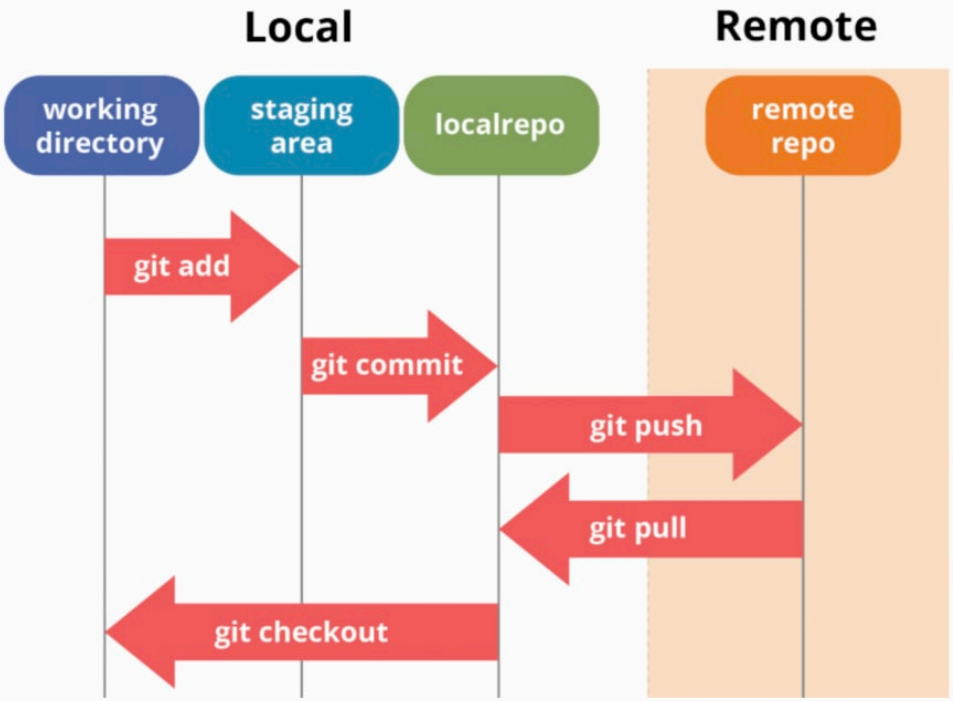
8. Push changes

```
git push
```

Getting Started with GitHub

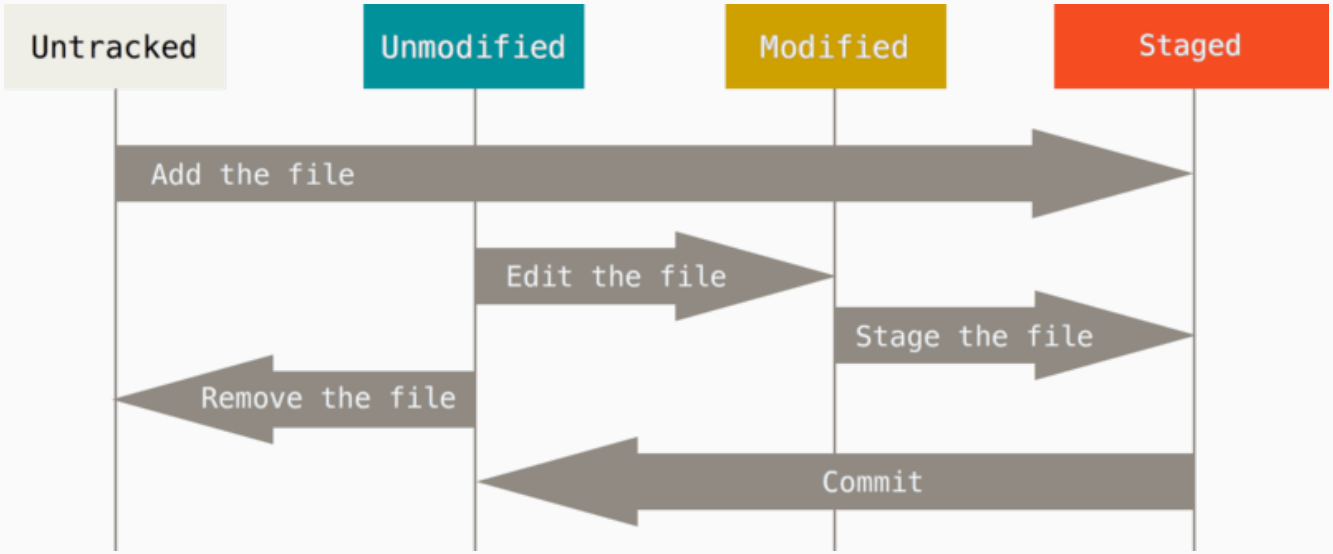
There are three main areas where the file lives when using Git:

1. **Working directory** (outside repository) where changes are made
2. `git add` moves files to the **staging area**
3. `git commit` moves files to the **permanent repository**
4. In the case of GitHub, `git push` then moves changes to the **remote repository**



Getting Started with GitHub

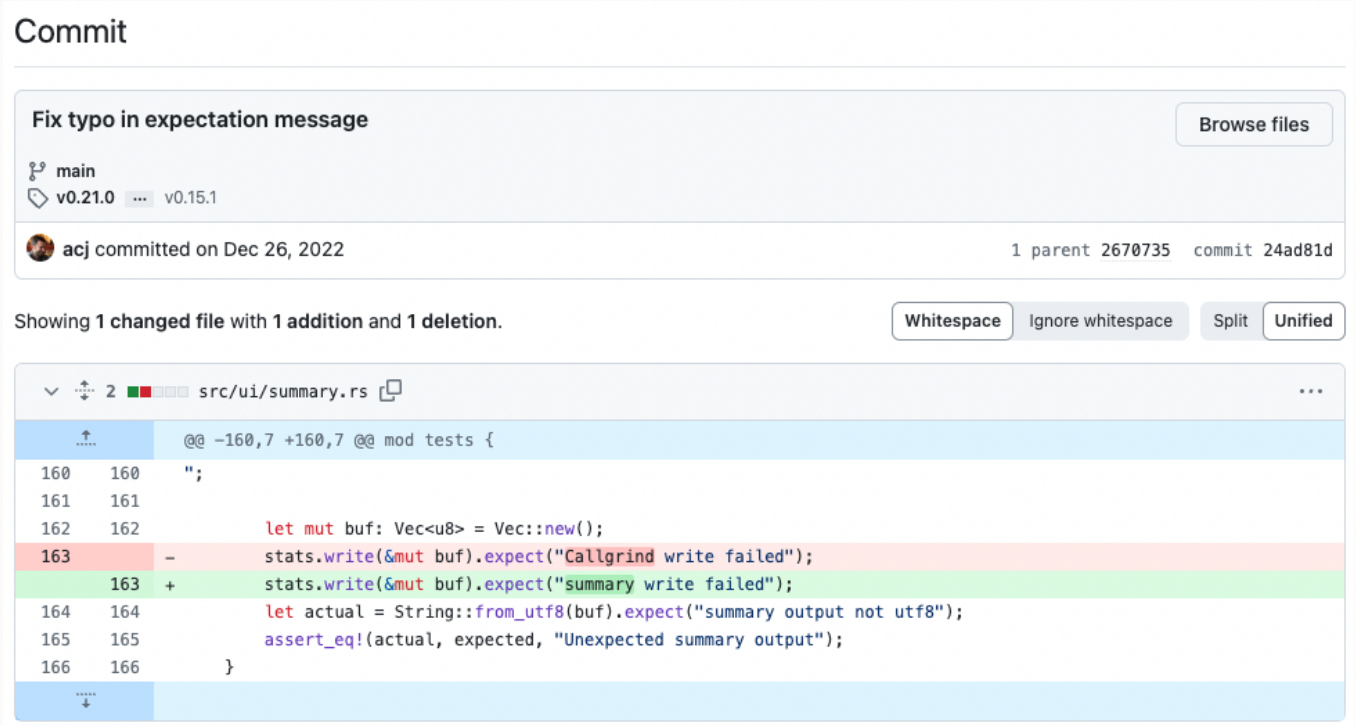
The full life-cycle involves checking out and starting from a commit, then changing, adding and committing the new changes.



Always fetch and pull first to get the latest repository and your collaborators' updates!

Committing as a “Diff”

`git commit` stores the changes to code as “diffs”



Source


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Branches and Merging

- The default is the **main** branch. This means that any committed changes will be part of the permanent repository.
- If you want to experiment, or make a version without disrupting the overall repository, create a new **branch** (see Github flow)

1. Create branch and switch to the branch

```
git checkout -b new_branch
```

2. View branches and which one you're on

```
git branch
```

3. Modify, add, and commit files on the new branch

4. Merge branch back into main if it works out

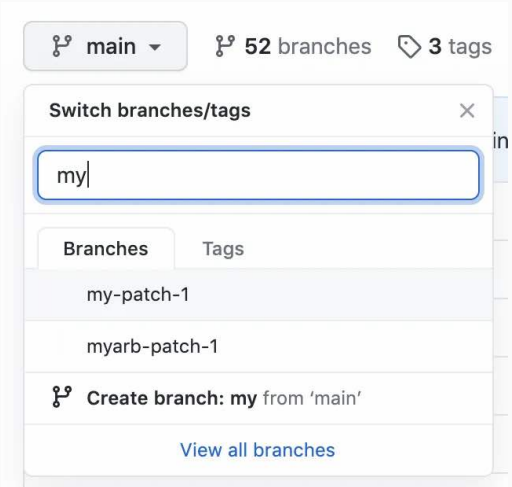
```
git checkout main
git merge new_branch
```

GitHub will tell you when there is a merge conflict – nothing will be overwritten!

Pull Requests

On GitHub, one can create **pull requests**. This requires other collaborators to check the code before allowing a new branch to be merged in, building in a code review component.

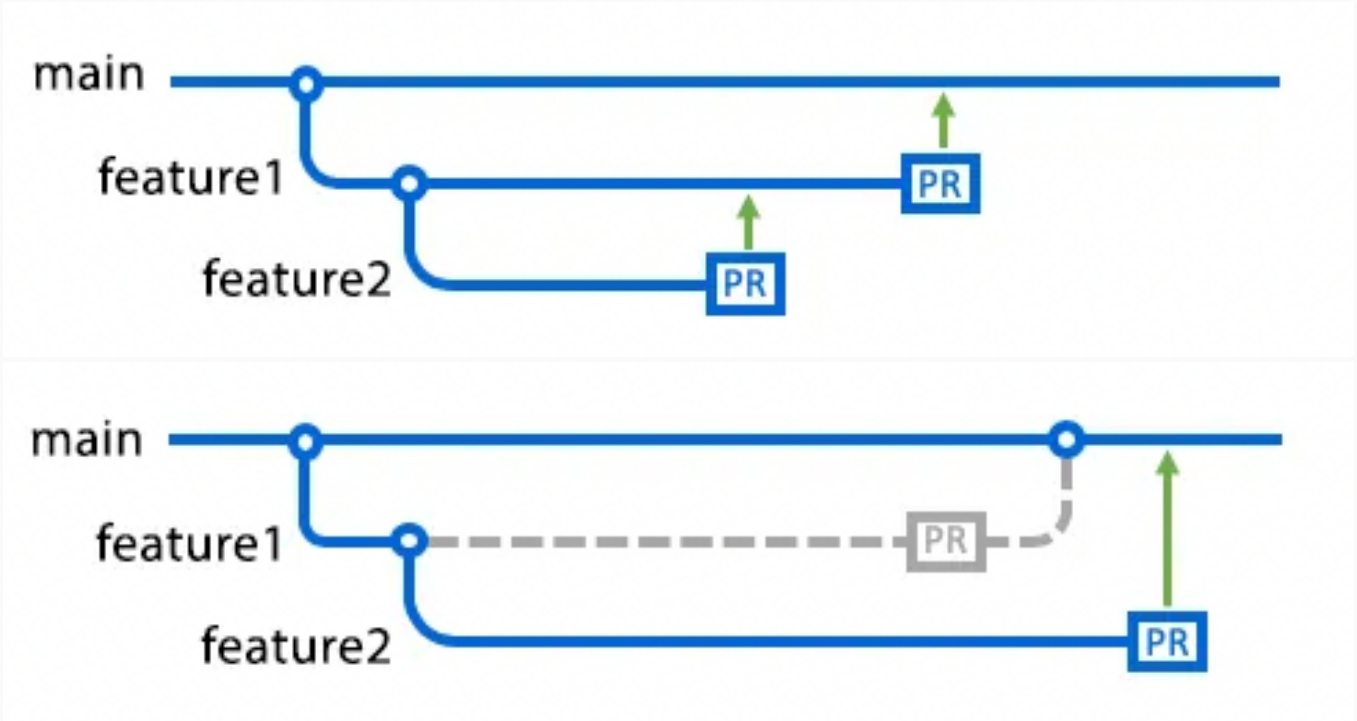
1. On GitHub.com, navigate to the repository and select the branch ready to be committed.



2. A banner will appear to create a pull request.

Branches and Merging

All work done on a branch is isolated prior to merging



Source

Metaprogramming

Metaprogramming / Automation

Suppose you have a number of scripts that must be run to process the data for your project. If you manually run each script one at a time you risk...

- Forgetting to run a script
- Running scripts out of order

Suppose you run your code and then manually port over the output to your paper. In this case, you risk...

- Making a typo when copying over the results
- Forgetting to update results in the paper after re-running your code

Further, in addition to being error-prone, these manual steps are tedious!

Solution: Automate!

Metaprogramming / Automation

One command should compile the entire research pipeline end-to-end:

Raw data → data processing scripts → analysis/estimation → save results → compile paper

For instance, results in your paper should auto-generated from your code:

```python
def save_scalar_tex(val, name, file_name):
    with open(file_name,'w') as file:
        tex = '\newcommand{\\'+name+'}{'+str(val)+'}'
        file.write(tex)

save_scalar_text(
    results[1, 1],
    'PolicyChangeEst',
    '../output/scalars/policy_scalars.tex'
)
results.to_latex('../output/tables/tab1.tex')
```

Python

```latex
As we can see in the table below, in our preferred
estimate, the policy resulted in a \PolicyChangeEst
percentage point increase in productivity.

\begin{center}
\input{../output/tables/tab1.tex}
\end{center}
```

LaTeX

Metaprogramming Tools

A **build system** turns source code and resources into a finished product.

Suppose we had a repo with the following structure:

- analysis
- data
- paper
- processing
- utils
- make.py

In this case, our build system is contained within *`make.py`*. Let us see how this works.

Metaprogramming

For example, we may write the following utility module called *run_scripts.py* in *utils*.

```
import os, sys, subprocess, re
# You can run scripts using this module.

def run_stata(dofile, *params):
    """Run Stata dofile in batch mode, and deletes the log file"""
    cmd = ["stata", "do", dofile]
    for param in params:
        cmd.append(param)
    subprocess.run(cmd)
    os.remove(f"{dofile[0:-3]}.log") # remove log file

def run_python(script):
    """Run Python script"""
    run = subprocess.run(["python", script], stdout=subprocess.PIPE, stderr=subprocess.STDOUT,
       universal_newlines=True)
    print(run.stdout)
    # run.returncode = 0 if code runs successfully.
    if run.returncode != 0:
        # stops when pyfile has an error
        sys.exit(f"Error : {script}")

def run_latex(tex_file):
    """Compile LaTeX to PDF"""
    subprocess.call(["pdflatex", tex_file])
```

Metaprogramming

The *run_scripts* module can be called by *make.py* to execute all of the code for our project:

```
import os, sys
from utils.run_scripts import run_python, run_stata, run_latex

def main():
    processing()
    analysis()
    paper()

def processing():
    os.chdir("processing")
    run_python("clean_data.py")
    run_stata("nearest_neighbor_matching.do")
    run_python("merge_data.py")
    os.chdir("..")

def analysis():
    os.chdir("analysis")
    run_python("summary_stats.py")
    run_python("models.py")
    os.chdir("..")

def paper():
    os.chdir("paper")
    run_latex("paper.tex")
    os.chdir("..")

# Run the main program, i.e. run everything
main()
```

Aside: Implementing Portable Code

Portability

Your code should be able to be run on different computer systems, platforms, or environments without significant modifications.

- Your collaborators may have different operating systems, hardware architectures, compilers, and configurations
- You may need to port your code to a remote server (e.g., the HBS Grid)
- Your research can't be replicated if it is not portable

Portability

Avoiding Hard-Coded Paths:

- Use *relative* paths or configuration files to handle file system differences

```
### Bad:
# can't be run on any other computer or if the folder computing-for-econ-dev is moved
df = pd.read_csv("/Users/tfronsdal/Desktop/computing-for-econ-dev/python-basics/data/iris.csv")
```

```
### Good:
# if Python script is in computing-for-econ-dev/python-basics/scripts
df = pd.read_csv("../data/iris.csv")
```

Documentation:

- Document platform-specific considerations
- Provide clear instructions for setting up and configuring the code on different systems (e.g. necessary environmental variables)

Managing Dependencies:

- Clearly document and manage external dependencies, libraries, and version requirements

Managing Dependencies

Your code may break or not perform as expected if collaborators or other researchers are using different package/language versions.

If versions are not well-documented, your results may not be replicable.

```
print "Estimating model..."
>> Estimating model...
```

Python 2

```
print "Estimating model..."
>> SyntaxError: Missing parentheses in call to
       'print'. Did you mean print(...)?
```

Python 3

```
ser = pd.Series(['13-01-2000', '12-01-2000'])
pd.to_datetime(ser)
>>
0   2000-01-13
1   2000-12-01
dtype: datetime64[ns]
```

pandas 1.5.3

```
ser = pd.Series(['13-01-2000', '12-01-2000'])
pd.to_datetime(ser)
>>
0   2000-01-13
1   2000-01-12
dtype: datetime64[ns]
```

pandas 2.0.0

Managing Dependencies

Solution: use a virtual environment or package manager.

A **virtual environment** is an isolated environment where a Python (or other programming language) interpreter and associated libraries can be installed independently of the system-wide installation.

Python includes a built-in virtual environment tool called *`venv`*.

`virtualenv` and *`conda`* are popular third-party tool for creating virtual environments.

Managing Dependencies

Example: Managing environment with *conda*

1. Create an *environment.yml* file that can be shared across collaborators:

```
name: my_env
channels:
  - defaults
dependencies:
  - python=3.10
  - numpy=1.25.0
  - pandas=2.1.0
  - statsmodels=0.13.5
```

2. Create *conda* environment:

```
conda env create -f environment.yml
```

3. Activate environment:

```
conda activate my_env
```