Software Engineering for Social Scientists

Shunsuke Tsuda

ECON 2020 Computing for Economists
Spring 2023

BROWN

Principles of a Productive Research Practice

Use computer wisely, write better code, and organize data nicely:

1. **Portability**: Code should work in any machines without any changes.
2. **Clarity & Maintainability**: Easy, direct, and straightforward to understand code. Easy to maintain and develop further any time.
3. **Accuracy**: Programs do what researchers intend. Design to easily detect if programs do what researchers did not intend.
4. **Efficiency**: Write algorithms that conserve computing resources and save computing time.
5. **Reproducibility**: Automate the whole research process. & Be able to reproduce any stage of research process.

Techniques Covered

1. **Portability**: Directories, Symbolic Links
2. **Clarity & Maintainability**:
   Task management, Documentation, Abstraction
3. **Accuracy**: Debugging, Unit testing, Logging
4. **Efficiency**: Vectorization, Parallelization, HPC
5. **Reproducibility**: Automation, Version control

Techniques Covered

1. **Portability**: Directories, Symbolic Links
2. **Clarity & Maintainability**:
   Task management, Documentation, Abstraction
3. **Accuracy**: Debugging, Unit testing, Logging
4. **Efficiency**: Vectorization, Parallelization, HPC
5. **Reproducibility**: Automation, Version control

Remarks.

- These concepts are **language-agnostic**.
- I am not a computer expert and these slides may not be comprehensive. Based on them, please always try to pursue your best practices.
- You may not have to follow all. Evaluate potential trade-offs such as:
  - Code quality vs. Development time & Frequency of use
  - Running speed vs. Difficulty of checking accuracy/robustness

References

- As the prerequisite assignment, you have read:
  Gentzkow, Matthew and Jesse M. Shapiro. 2014. "Code and Data for the Social Sciences: A Practitioner's Guide."
  (link "GS2014" henceforth)
- The following notes are also valuable:

  Lecture notes by Fernández-Villaverde

  "Coding for Economists" by Ljubica Ristovska

  Note by Benjamin Skrainka
- I deeply thank Jesse Shapiro, and Matt Turner, Bo Yeon Jang, Kohei Kawaguchi, and Masahiro Kubo for their inputs.

E.g.) A Regular Life of an Empirical Economist:

Data Construction for Analysis From Raw Data

Event-Level Data of Conflicts (**raw data**)

	id	side_a	side_a_id	side_b	side_b_id	type_of_violence	event_date
0	1	Gov. of Somalia	95	Al-Shabaab	717	1	20080510
1	2	Gov. of Mali	72	AQIM	539	1	20150705
.	.	.	.	.	.	.	.
11	12	Al-Shabaab	717	Hizbul Islam	1004	2	20091001
12	13	AQIM	539	CMA	1158	2	20160313
.	.	.	.	.	.	.	.
76	77	Al-Shabaab	717	Civilians	1	3	20091025
77	78	Boko Haram	1051	Civilians	1	3	20140710
.	.	.	.	.	.	.	.

Source: Uppsala Conflict Data Program – Georeferenced Event Dataset (UCDP-GED)

Non-State Group-Level Data of Conflicts (**data for analysis**)

	id	group_name	group_id	opponent_name	opponent_id	type_of_violence	event_date
0	1	Al-Shabaab	717	Gov. of Somalia	95	1	20080510
1	2	AQIM	539	Gov. of Mali	72	1	20150705
.	.	.	.	.	.	.	.
11	12	Al-Shabaab	717	Hizbul Islam	1004	2	20091001
12	13	AQIM	539	CMA	1158	2	20160313
13	12	Hizbul Islam	1004	Al-Shabaab	717	2	20091001
14	13	CMA	1158	AQIM	539	2	20160313
.	.	.	.	.	.	.	.
78	77	Al-Shabaab	717	Civilians	1	3	20091025
79	78	Boko Haram	1051	Civilians	1	3	20140710
.	.	.	.	.	.	.	.

Project_EthnicConflicts/codes/build/prep_conflicts_groups.py

```
# This code transforms event-level UCDP-GED into non-state group-level data
# Creator: Masahiro Kubo (masahiro_kubo@brown.edu) 20190702
# Last Modifier: Shunsuke Tsuda (shunsuke_tsuda@brown.edu) 20190926
## INPUT:
# raw_dir + "UCDP\ged181.csv" : UCDP-GED ver 18.1
## OUTPUT:
# "Dropbox\Project_EthnicConflicts\data\process\ged181_Africa_groups_dup.csv"
import numpy as np
import pandas as pd
from pandas import DataFrame
## Please change below into your own path to dataset
data_dir    = r"C:\Users\stsuda\Dropbox\Project_EthnicConflicts\data\"
#data_dir    = r"C:\Users\masah\Dropbox\Project_EthnicConflicts\data\"

## Import UCDP raw data, which contains events during 1945-2017
ged_raw = pd.read_csv(data_dir + "raw\UCDP\ged181.csv", sep=',')
## Keep only events which take pleace in Africa ("keep if" & "rename" in STATA)
ged_Africa = ged_raw[ged_raw.region=='Africa']

## Construct the duplicates identifiers,
## especially for the type2 events (Rebel vs Rebel)
ged_Africa['duplicate_id']=1
## Subset of the data with Non-State Rebel vs Non-State Rebel
temp_type2 = ged_Africa[ged_Africa.type_of_violence==2]
temp_type2['duplicate_id']=2
## Construct the group-level data by "append",
## allowing the duplication of Non-State Rebel groups
ged_Africa_groups = pd.concat([ged_Africa, temp_type2])

## Reset the index after the append (Important process in PYTHON)
ged_Africa_groups.reset_index(drop=True, inplace=True)
## Check the duplicated index (Check if it's zero!)
ged_Africa_groups[ged_Africa_groups.index.duplicated()]
```

(cont'd) Project_EthnicConflicts/codes/build/prep_conflicts_groups.py

```
## Generate unique identifiers: "group_id" & "opponent_id"
## IMPORTANT: these ids are used through the analyses.
# If Conflict Type = Gov vs Group (Type=1)
cdn_type1 = ged_Africa_groups.type_of_violence==1
ged_Africa_groups.loc[cdn_type1, 'group_id'] = ged_Africa_groups.side_b_id
ged_Africa_groups.loc[cdn_type1, 'opponent_id'] = ged_Africa_groups.side_a_id
ged_Africa_groups.loc[cdn_type1, 'group_name'] = ged_Africa_groups.side_b
ged_Africa_groups.loc[cdn_type1, 'opponent_name'] = ged_Africa_groups.side_a
# If Conflict Type = One-Sided Violence against Civilians (Type=3)
cdn_type3 = ged_Africa_groups.type_of_violence==3
ged_Africa_groups.loc[cdn_type3, 'group_id'] = ged_Africa_groups.side_a_id
ged_Africa_groups.loc[cdn_type3, 'opponent_id'] = ged_Africa_groups.side_b_id
ged_Africa_groups.loc[cdn_type3, 'group_name'] = ged_Africa_groups.side_a
ged_Africa_groups.loc[cdn_type3, 'opponent_name'] = ged_Africa_groups.side_b
# If Conflict Type = Group vs Group (Type=2)
cdn_type2_1=(ged_Africa_groups.type_of_violence==2)&(ged_Africa_groups.duplicate_id==1)
ged_Africa_groups.loc[cdn_type2_1, 'group_id'] = ged_Africa_groups.side_a_id
ged_Africa_groups.loc[cdn_type2_1, 'opponent_id'] = ged_Africa_groups.side_b_id
ged_Africa_groups.loc[cdn_type2_1, 'group_name'] = ged_Africa_groups.side_a
ged_Africa_groups.loc[cdn_type2_1, 'opponent_name'] = ged_Africa_groups.side_b
cdn_type2_2=(ged_Africa_groups.type_of_violence==2)&(ged_Africa_groups.duplicate_id==2)
ged_Africa_groups.loc[cdn_type2_2, 'group_id'] = ged_Africa_groups.side_b_id
ged_Africa_groups.loc[cdn_type2_2, 'opponent_id'] = ged_Africa_groups.side_a_id
ged_Africa_groups.loc[cdn_type2_2, 'group_name'] = ged_Africa_groups.side_b
ged_Africa_groups.loc[cdn_type2_2, 'opponent_name'] = ged_Africa_groups.side_a

## Export the final output
ged_Africa_groups.to_csv(data_dir + "process/ged181_Africa_groups_dup.csv")

print('Construct UCDP-GED events by gid, allowing duplicated ones, done')
```

Portability

Portability

- Code and data must be migrated to other machines in various situations:

  Buy a new computer; In a more powerful server at university;

  Collaborators' computers; A journal editor's laptop, etc...

- Be portable: Code should work in any machines without any changes.

Techniques for improving portability:

1. **Fix rules for directory structure and be consistent with it throughout a project and among collaborators.**
2. **Write code which does not depend on a machine-specific path.**

E.g.) Directories for the Ethnic Conflict Project

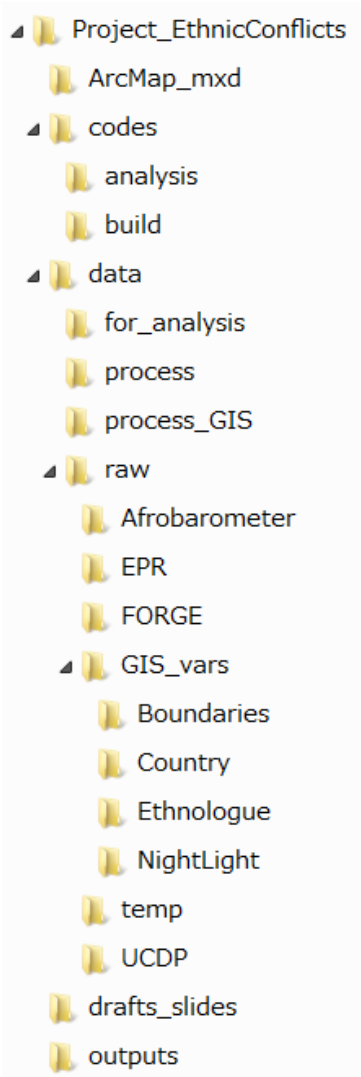
Paper production steps:

`/data/raw`
: All raw data used for this research

(→ `/data/process`; `/data/process_GIS`
: Some intermediate data)

→ `/data/for_analysis`
: The final data used for statistical analyses

→ `/outputs`
: Figures & Tables generated by statistical analyses



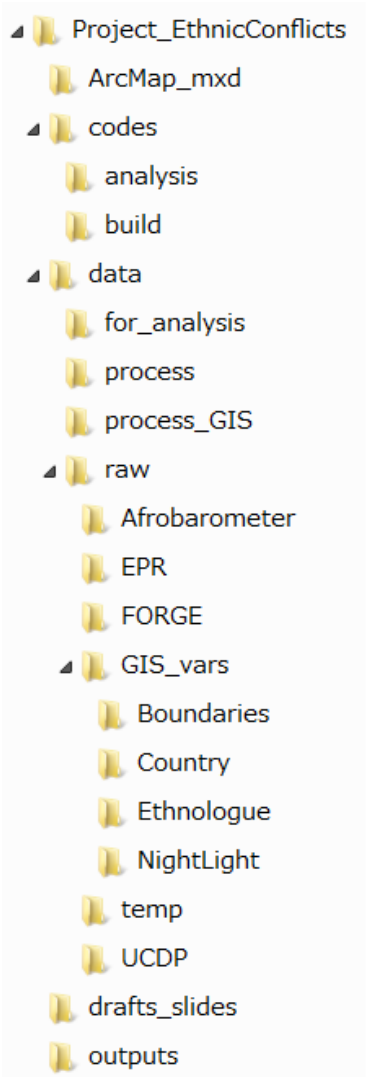
E.g.) Directories for the Ethnic Conflict Project

`/codes/build`
: Code for building the data for analyses from raw data

- Inputs: `/data/raw`
- Outputs: `/data/for_analysis`

`/codes/analysis`
: Code for generating final outputs by analyses

- Inputs: `/data/for_analysis`
- Outputs: `/outputs`



E.g.) Directories for a Larger Scale Project

/code/clean
- Inputs: /data/raw
- Outputs: /data/raw_cleaned

/code/build_1_slow
- Inputs: /data/raw_cleaned
- Outputs: /data/process

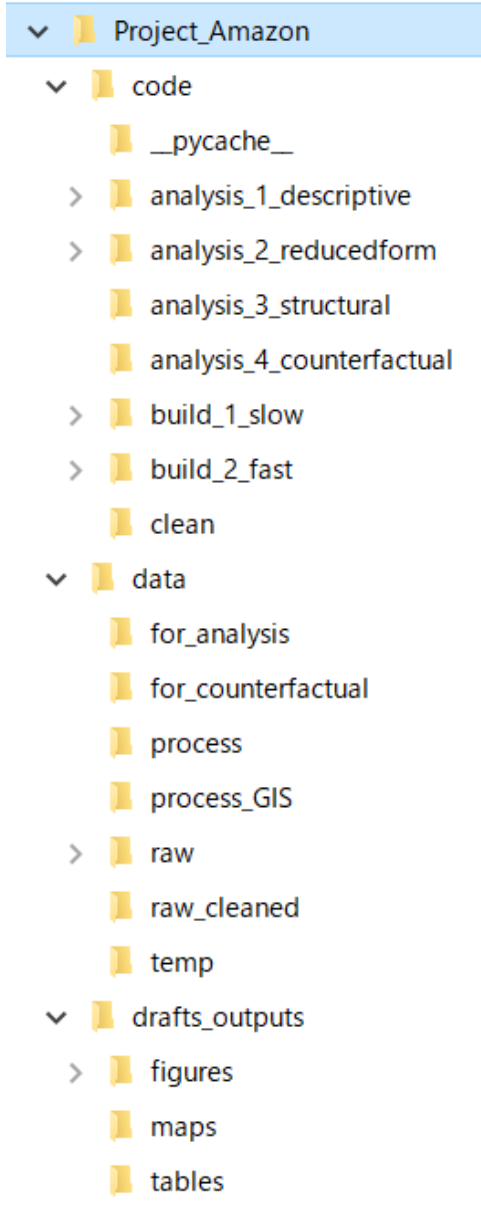
/code/build_2_fast
- Inputs: /data/raw_cleaned & /data/process
- Outputs: /data/for_analysis

/code/analysis_1_descriptive
- Inputs: /data/for_analysis
- Outputs: /drafts_outputs

/code/analysis_2_reducedform:
- Inputs: /data/for_analysis
- Outputs: /drafts_outputs

/code/analysis_3_structural:
- Inputs: /data/for_analysis
- Outputs: /drafts_outputs & /data/for_counterfactual

/code/analysis_4_counterfactual:
- Inputs: /data/for_analysis & /data/for_counterfactual
- Outputs: /drafts_outputs



Directories

- No need to follow the above example precisely.
- No unique golden structure exists:
  - Different researchers employ somewhat different directory structures.
    E.g.) GS2014 proposes a different structure from mine.
  - Optimal directory structures might differ across different research projects.

  But, just follow the key principles:

1. **Separate directories by functions and by steps in a research project**
2. **Separate files into inputs and outputs of each step**

E.g.) Our Code was NOT Portable

Project_EthnicConflicts/codes/build/prep_conflicts_groups.py

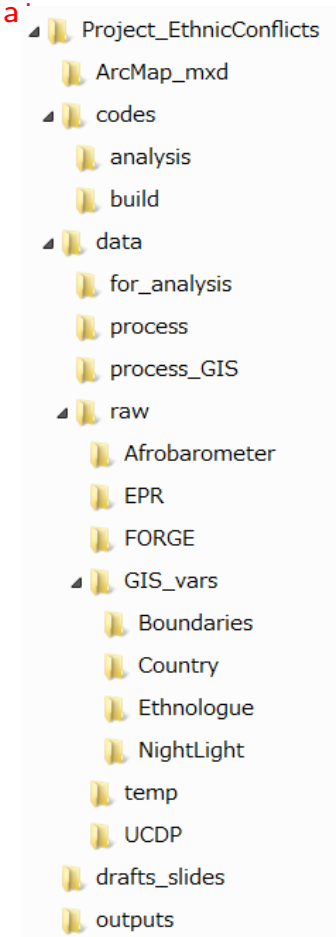
```
# This code transforms event-level UCDP-GED into non-state group-level data
import numpy as np
import pandas as pd
from pandas import DataFrame
## Please change below into your own path to datasets
data_dir    = r"C:\Users\stsuda\Dropbox\Project_EthnicConflicts\data\"
#data_dir    = r"C:\Users\masah\Dropbox\Project_EthnicConflicts\data\"
## Import UCDP raw data, which contains events during 1945-2017
ged_raw = pd.read_csv(data_dir + "raw\UCDP\ged181.csv", sep=',')
⋮
```

- I am working on a joint project with Masa.
- Our (absolute) paths to the data are different.
- The person who edits and runs this code adjusts the above path each time.
- **NO!**

Use a RELATIVE Path for Portability

`Project_EthnicConflicts/codes/build/prep_conflicts_groups.py`

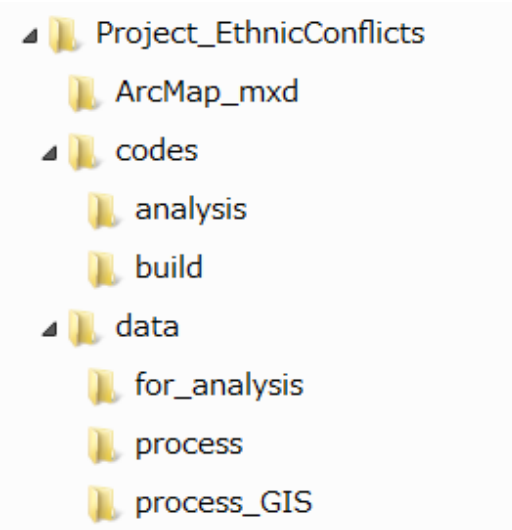
```
# This code transforms event-level UCDP-GED into non-state group-level da
import numpy as np
import pandas as pd
from pandas import DataFrame
data_dir    = r"..\..\data\"    # RELATIVE path to datasets
## Import UCDP raw data, which contains events during 1945-2017
ged_raw = pd.read_csv(data_dir + "raw\UCDP\ged181.csv", sep=',')
⋮
```

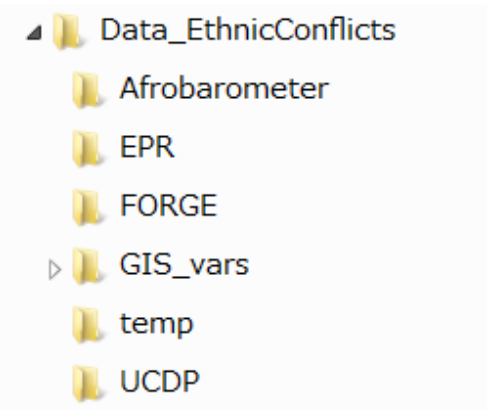


- With the above relative path, any computer can run the code without any changes in the code, as long as the directory structure is fixed.
- Of course, start writing relative paths for the first time after ensuring the internal consistency of directory structure among collaborators.

What if the Project Scale Enlarges Further?

- Raw datasets tend to be heavy in a large scale projcet.
- Some researchers do not want to put them in a main directory which affects machine storage (DropBox, C drive, etc)
- Fine to put them outside (e.g., external hard drive)
- That is also why it is important to separate directories by function:
  - Easy to move only the raw datasets to another place
  - Once in the analysis step, no more frequent use of raw datasets





What if the Project Scale Enlarges Further?

```
data_dir    = r"..\..\data\"   # RELATIVE path to processed datasets
#Set the path to the raw data for each researcher
import getpass
if getpass.getuser() == "stsuda":
    raw_dir    = "Y:/Data_EthnicConflicts/"
elif getpass.getuser() == "masah":
    raw_dir    = "C:/Users/masah/Dropbox/Data_EthnicConflicts/"

## Import UCDP raw data
ged_raw = pd.read_csv(raw_dir + "UCDP/ged181.csv", sep=',')


⋮

## Export the final output
ged_Africa_groups.to_csv(data_dir + "process/ged181_Africa_groups_dup.csv")
```

- In this case, since some collaborators store raw datasets outside, should we rely on absolute paths for calling raw data?
- **NO!** ⇒ Use symbolic links to still keep the main code NOT dependent on machine

Symbolic Links

- **Symbolic links: Shortcuts to a folder**
- Python module: **os.symlink**
  You can also set symbolic links by Command Prompt (Windows) or Terminal (Mac), but I find this Python module more maintainable.
- Run the below code **just once** at the beginning of the project, and use relative paths in the main code even if different researchers put their *actual* raw data in different locations

```
import os, getpass
# Create a "shortcut" to an external raw data folder here accessible by a relative path
input = "../data/raw"

if    getpass.getuser() == "stsuda":
    external_raw = "Y:/Data_EthnicConflicts/"
elif getpass.getuser() == "masah":
    external_raw = "C:/Users/masah/Dropbox/Data_EthnicConflicts/"

def symlink_force(external, link_name):
    try:
        os.symlink(external, link_name)
    except FileExistsError:
        os.remove(link_name)
        os.symlink(external, link_name)

symlink_force(external_raw, input)
```

Clarity & Maintainability

Clarity & Maintainability

Plan ahead for maintenance and future extensions:
(e.g., R&R, Extension of an existing paper for a new project, Share the code with another researcher for his/her project)

- All collaborators should easily understand tasks and follow discussions in the entire research process
- All collaborators should understand your code quickly and straightforwardly without asking any questions and without opening any other documents
- Have no confusions when coming back to your code, data, and discussions after years

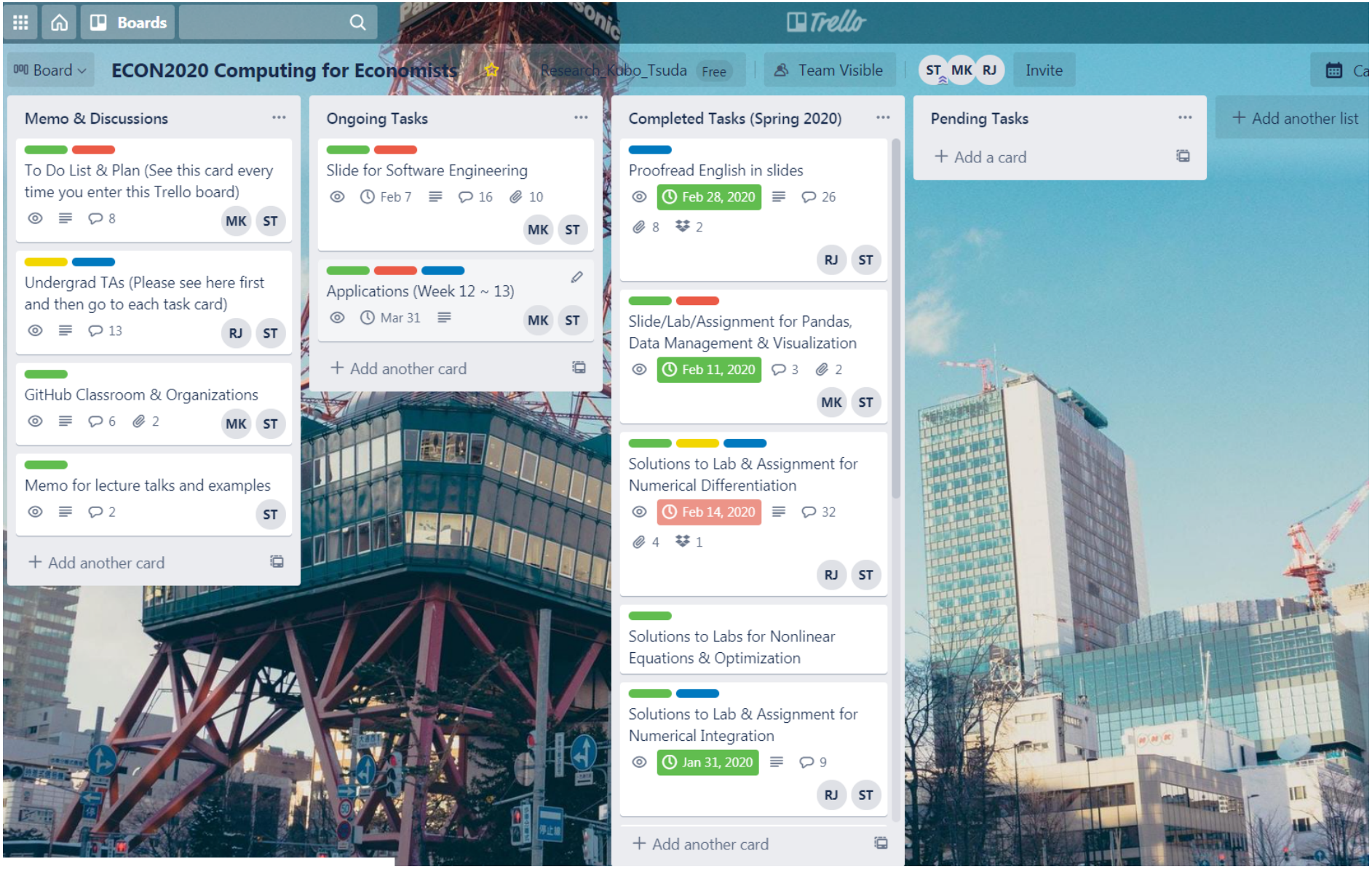
Techniques for improving clarity & maitainability:

1. **Task management system**
2. **Abstraction**
3. **Self- & minimally-documented code**

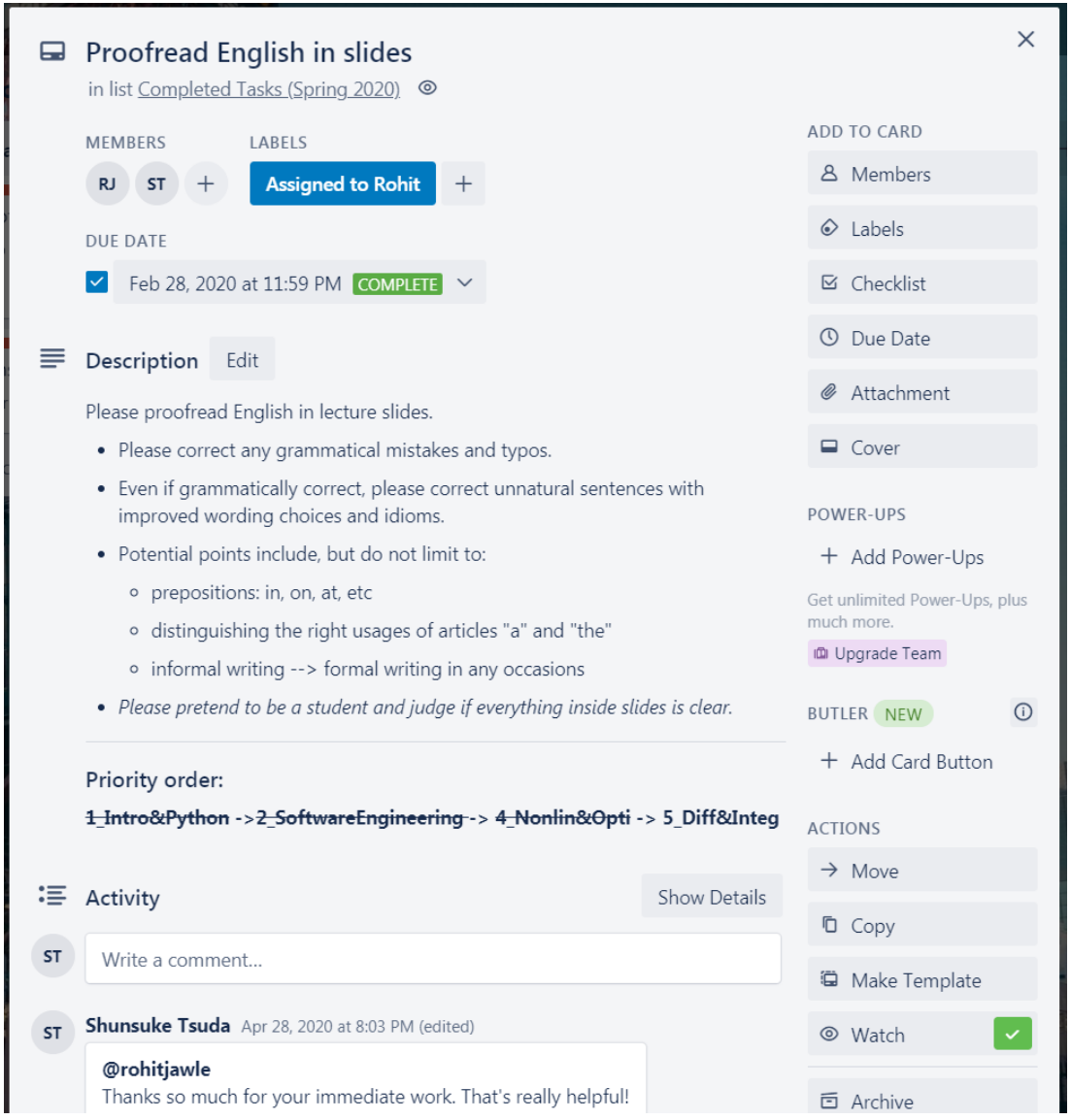
Management of Research Projects

- Do NOT have discussions on your research project that you want to maintain by e-mail
  - It is hard to to track past discussions by emails
  - It also takes time to copy and paste discussions by emails into a memo word file, and its maintainance cost is high
- Rather, manage tasks and accumulate discussions by a task management system
- It also helps a solo research project to accumulate decisions, thoughts, and records
- Some free task management systems:
  - **Trello**
  - GitHub Project
  - ActiveCollab
  - Asana
  - Slack (?)

E.g.) Task Management System (Trello)



E.g.) Task Management System (Trello)



Abstraction

- Oftentimes, we write similar code by copying and pasting...
- Abstraction of code by a general-form funcion helps to:
  - Simplify and shorten code
  - Reduce mistakes
  - Reuse for different purposes with minimum amount of revisions
- More generally, design your code such that you can modify only one visible place in your code when you should modify something in your research process.
- Recall the trade-off between code development time and code quality — Be patient and think dynamically!

Code Example

An example in Chapter 6 of GS2014 transformed to Python:

```
import pandas as pd

# State-level aggregation
df['total_pc_potato'] = df.groupby('state')['pc_potato'].transform('sum')
df['total_obs']       = df.groupby('state')['pc_potato'].transform('count')
df['leaveout_state_pc_potato'] \\
= (df['total_pc_potato'] - df['pc_potato']) / (df['total_obs'] - 1)

# Metroarea-level aggregation
df['total_pc_potato'] = df.groupby('metroarea')['pc_potato'].transform('sum')
df['total_obs']       = df.groupby('state')['pc_potato'].transform('count')
df['leaveout_metro_pc_potato'] \\
= (df['total_pc_potato'] - df['pc_potato']) / (df['total_obs'] - 1)

# What if we have to repeat this more at different levels?
```

Code Example: Abstraction

```
def leaveout_mean(d, invar, byvar):
    '''
    This function returns "leave-out" mean of 'invar' for each group 'byvar'.
    '''

    if __name__ == '__main__':

        #Sum 'invar' by group 'byvar'
        d['tot_invar']    = d.groupby(byvar)[invar].transform('sum')

        #Count total observations by group 'byvar'
        d['count_invar']  = d.groupby(byvar)[invar].transform('count')

        #''leave-out'' mean of 'invar' for each group 'byvar'
        d['outvar'] = (d['tot_invar'] - d[invar]) / (d['count_invar'] - 1)
        return d['outvar']

# With this documentations by """ """,
# help(func) can return the documentations of the function.
help(leaveout_mean)

# State-level aggregation
leaveout_mean(d = df, invar = 'pc_potato', byvar = 'state')

# Metroarea-level aggregation
leaveout_mean(d = df, invar = 'pc_potato', byvar = 'metroarea')
```

Code as Documentation

Code should be self-documenting and document minimally to convey necessary information:

- Naming of files, variables, functions, logical conditions, and error-checkings indicate their meanings straightforwardly
- Keep only documentation that you must maintain by construction of your code, i.e., revising your code should automatically revise your documentation as well

Do NOT write about the same information at multiple locations:

(which entails a risk of having internal inconsistency by forgetting to revise at one location while revising another when the information is updated...)

- Abstraction
- Reduce comment sentences and external "memo" files

That being said, **there are also necessary comment sentences**:

- Key decisions and choices made, and **why** you made them
- Time-invariant things: e.g. formulas, descriptions of built-in packages/commands (if their names are not straightforward)
- Citations and links

Coding Rules as Clear Documentation

- Be consistent within a script, among collaborators, and within a project
- The file name of the code indicates what this code is doing (rather than commenting on a top line inside the code script)
- Import necessary packages on top
- Define inputs and outputs of the code on top (with descriptive names of data files $\Rightarrow$ no need of comment sentenses for describing data sources)
- Define parameters and variables on top (with descritive names)
- Write a main function (a main program in Stata) which describes the entire structure and flow (with descriptive names of inside functions)
- Order functions linearly. Make sub-functions appear immediately after the higher level functions that call them.
- If you have to inevitably write comment sentences, make them not dispersed but concentrated in a clearly visible location

E.g.) Code as Documentation (in STATA)

writingmaps_market_river_access

```
clear all
set more off
*** DIRECTORIES
global data_orig "../../data/raw/1.Data/13.Grid"
global data_w    "../../data/for_analysis"
global output    "../../drafts_outputs"
********************************************************************************
*** INPUTs
********************************************************************************
global input_main_data          "$data_w/Dataset_od_2021feb.dta"
global input_GIS_grids          "$data_orig/Map_element/ShapefileGrids/SmallGrids_Updated"
global input_GIS_river          "$data_orig/Map_element/River/RiverNetwork_Hydrology_UpdatedOrder"

********************************************************************************
*** OUTPUTs
********************************************************************************
global shape_grid               "$data_w/shape_gridcells.dta"
global coord_grid               "$data_w/coordinates_gridcells.dta"
global shape_river              "$data_w/shape_riverline.dta"
global coord_river              "$data_w/coordinates_riverline.dta"
global coord_river_LowerUcayali "$data_w/coordinates_riverline_LowerUcayali.dta"
global coord_river_Pastaza      "$data_w/coordinates_riverline_Pastaza.dta"
global coord_river_Ucayali      "$data_w/coordinates_riverline_Ucayali.dta"
global coord_river_Napo         "$data_w/coordinates_riverline_Napo.dta"
// + Many MAPs are also produced.

********** ROADMAP ************************************************************
capture program drop main
program main
  // Follow the steps described in: https://www.stata.com/support/faqs/graphics/spmap-and-maps/
  /*
  step 1. Install necessary commands
  ssc install spmap
  ssc install shp2dta
  ssc install mif2dta
  ssc install mergepoly
  step 2. prepare .shp & .dbf: See INPUTs above
  */
  step3_translate_shp2dta
  step4_prepare_grid_map
  step5_merge_gridmap_MA_RA
  step6_cut_river_within_basin
  step7_draw_maps
end
********************************************************************************
*** Definitions & Categories
global basins    "LowerUcayali Pastaza Ucayali Napo"
```

4_did_diff_topic_term_congress...

```
/*
ssc install reghdfe
ssc install ftools
ssc install estout
ssc install rwolf
ssc install randcmd
*/
clear all
set more off
*** DIRECTORIES
global data_dir   "../../data"
global result_dir "../../drafts_outputs"
********************************************************************************
*** INPUTs
********************************************************************************
global congressmen_statements   "$data_dir/for_analysis/congressmen_statements_period_term_topics.csv"
global states_fractionalization "$data_dir/for_analysis/states_1951_fractionalization_match.dta"

********************************************************************************
*** OUTPUTs
********************************************************************************
* Many tables

********** ROADMAP ************************************************************
program main
 construct_variables
 sample_selection
 summary_stats
 multiple_hypo
 Diff_topics_pre
 Diff_topics
 Diff_topics_HHI
 DID_topics
 DID_topics_3periods
end

********** DEFINITIONs ********************************************************
*** TABLE OPTIONs
global tab_sum_opt   "replace cells("mean(fmt(a3)) sd min max") nonumber nomtitle label substitute(\_ _)"
global tab_reg_opt1  "replace label se substitute(\_ _) star(* 0.1 ** 0.05 *** 0.01) nomtitles nonotes nc
global tab_reg_opt2  "stats(fe_party fe_state r2 r2_a ymean ysd N, fmt(%9.3f %9.3f %9.3f %9.3f %9.3f %9.3
"State FE" R$^2$ "Adjusted R$^2$" "Mean (Dep. Var.)" "SD (Dep. Var.)" Observations))"
global se_cluster    "cluster state_id"
global tab_note_1_se "Robust standard errors clustered at the state level in parentheses."
 * Because randomization is stratified at the state level

*** KEY OUTCOMES
global topics "trade agriculture industry resource financial infrastructure conflict culture political hu
```

Recall: Programming Paradigms

Procedural: The program moves through instructions linearly

- Simple to write and easy to do line-by-line trial and error

Functional: The program moves from function to function

- Separating tasks into sub-functions make the code readable, scannable, and maintainable
- By looking at the "main" function, easy to understand the entire strucuture of the program
- Easy to run a subset of the entire program

Object Oriented Programming (OOP): a programming paradigm which provides a means of structuring programs so that data and its behaviors are bundled into an individual object (see the previous slides in more detail)

Exercise 1

Recall the Ethnic Conflict Project:

Data Construction for Analysis From Raw Data

Event-Level Data of Conflicts (**raw data**)

	id	side_a	side_a_id	side_b	side_b_id	type_of_violence	event_date
0	1	Gov. of Somalia	95	Al-Shabaab	717	1	20080510
1	2	Gov. of Mali	72	AQIM	539	1	20150705
.	.	.	.	.	.	.	.
11	12	Al-Shabaab	717	Hizbul Islam	1004	2	20091001
12	13	AQIM	539	CMA	1158	2	20160313
.	.	.	.	.	.	.	.
76	77	Al-Shabaab	717	Civilians	1	3	20091025
77	78	Boko Haram	1051	Civilians	1	3	20140710
.	.	.	.	.	.	.	.

Source: Uppsala Conflict Data Program – Georeferenced Event Dataset (UCDP-GED)

Non-State Group-Level Data of Conflicts (**data for analysis**)

	id	group_name	group_id	opponent_name	opponent_id	type_of_violence	event_date
0	1	Al-Shabaab	717	Gov. of Somalia	95	1	20080510
1	2	AQIM	539	Gov. of Mali	72	1	20150705
.	.	.	.	.	.	.	.
11	12	Al-Shabaab	717	Hizbul Islam	1004	2	20091001
12	13	AQIM	539	CMA	1158	2	20160313
13	12	Hizbul Islam	1004	Al-Shabaab	717	2	20091001
14	13	CMA	1158	AQIM	539	2	20160313
.	.	.	.	.	.	.	.
78	77	Al-Shabaab	717	Civilians	1	3	20091025
79	78	Boko Haram	1051	Civilians	1	3	20140710
.	.	.	.	.	.	.	.

EC2020/class_materials/prep_conflicts_groups_bad.py

```
# This code transforms event-level UCDP-GED into non-state group-level data
## INPUT:
# raw_dir + "UCDP\ged181.csv" : UCDP-GED ver 18.1
## OUTPUT:
# "Dropbox\EC2020\data\ged181_Africa_groups_dup.csv"
import pandas as pd
from pandas import DataFrame
data_dir    = r"..\data\"   # RELATIVE path to datasets"

## Import UCDP raw data, which contains events during 1945-2017
ged_raw = pd.read_csv(data_dir + "ged181.csv", sep=',')
## Keep only events which take pleace in Africa ("keep if" & "rename" in STATA)
ged_Africa = ged_raw[ged_raw.region=='Africa']

## Construct the duplicates identifiers,
## especially for the type2 events (Rebel vs Rebel)
ged_Africa['duplicate_id']=1
## Subset of the data with Non-State Rebel vs Non-State Rebel
temp_type2 = ged_Africa[ged_Africa.type_of_violence==2]
temp_type2['duplicate_id']=2

## Construct the group-level data by "append",
## allowing the duplication of Non-State Rebel groups
ged_Africa_groups = pd.concat([ged_Africa, temp_type2])

## Reset the index after the append (Important process in PYTHON)
ged_Africa_groups.reset_index(drop=True, inplace=True)
## Check the duplicated index (Check if it's zero!)
ged_Africa_groups[ged_Africa_groups.index.duplicated()]
```

(cont'd) EC2020/class_materials/prep_conflicts_groups_bad.py

```
## Generate unique identifiers: "group_id" & "opponent_id"
## IMPORTANT: these ids are used through the analyses.
# If Conflict Type = Gov vs Group (Type=1)
cdn_type1 = ged_Africa_groups.type_of_violence==1
ged_Africa_groups.loc[cdn_type1, 'group_id'] = ged_Africa_groups.side_b_id
ged_Africa_groups.loc[cdn_type1, 'opponent_id'] = ged_Africa_groups.side_a_id
ged_Africa_groups.loc[cdn_type1, 'group_name'] = ged_Africa_groups.side_b
ged_Africa_groups.loc[cdn_type1, 'opponent_name'] = ged_Africa_groups.side_a
# If Conflict Type = One-Sided Violence against Civilians (Type=3)
cdn_type3 = ged_Africa_groups.type_of_violence==3
ged_Africa_groups.loc[cdn_type3, 'group_id'] = ged_Africa_groups.side_a_id
ged_Africa_groups.loc[cdn_type3, 'opponent_id'] = ged_Africa_groups.side_b_id
ged_Africa_groups.loc[cdn_type3, 'group_name'] = ged_Africa_groups.side_a
ged_Africa_groups.loc[cdn_type3, 'opponent_name'] = ged_Africa_groups.side_b
# If Conflict Type = Group vs Group (Type=2)
cdn_type2_1=(ged_Africa_groups.type_of_violence==2)&(ged_Africa_groups.duplicate_id==1)
ged_Africa_groups.loc[cdn_type2_1, 'group_id'] = ged_Africa_groups.side_a_id
ged_Africa_groups.loc[cdn_type2_1, 'opponent_id'] = ged_Africa_groups.side_b_id
ged_Africa_groups.loc[cdn_type2_1, 'group_name'] = ged_Africa_groups.side_a
ged_Africa_groups.loc[cdn_type2_1, 'opponent_name'] = ged_Africa_groups.side_b
cdn_type2_2=(ged_Africa_groups.type_of_violence==2)&(ged_Africa_groups.duplicate_id==2)
ged_Africa_groups.loc[cdn_type2_2, 'group_id'] = ged_Africa_groups.side_b_id
ged_Africa_groups.loc[cdn_type2_2, 'opponent_id'] = ged_Africa_groups.side_a_id
ged_Africa_groups.loc[cdn_type2_2, 'group_name'] = ged_Africa_groups.side_b
ged_Africa_groups.loc[cdn_type2_2, 'opponent_name'] = ged_Africa_groups.side_a

## Export the final output
ged_Africa_groups.to_csv(data_dir + "ged181_Africa_groups_dup.csv")

print('Construct UCDP-GED events by gid, allowing duplicated ones, done')
```

Exercise: Code Cleanup

- Notice that the portability is improved in the above code (the same as the distributed code).

Task. Clean up the above code further. Especially care about the principles of **clarity** & **maintainability**.

Hint. You can edit anything in the code (including file name, variable name, etc)

Hint. Any scope for abstraction?

Hint. There are many detailed explanations in the code. Very generous! But, are they really beneficial?

Hint. Conflict event datasets are updated annually by UCDP. Researchers always want to use up-to-date information.

Accuracy

Accuracy: Correctness / Robustness / Verifiability

Imagine the mindset of airplane maintenance

- **Avoid coding errors and if any, find them fast**
- **Be sure that code runs as researchers intended** (even if the code does not return "errors")
- **Test often and modularly**

Techniques for improving accuracy:

1. **Debugging**
2. **"Try" & "Except"** block
3. **Unit testing**
4. **Logging**

Debugging

- Debugging: finding problems/errors in your code and fixing them.
- Not good to make random changes to your code in order to find and fix errors.
- Debugging is more valuable for a larger program, since it is harder to find a problem when you have many loops and your own (user-defined) functions.

Remark. Debugging tools vary across platforms, IDEs and editors. Here, we focus on Spyder.

Debugging Example 1

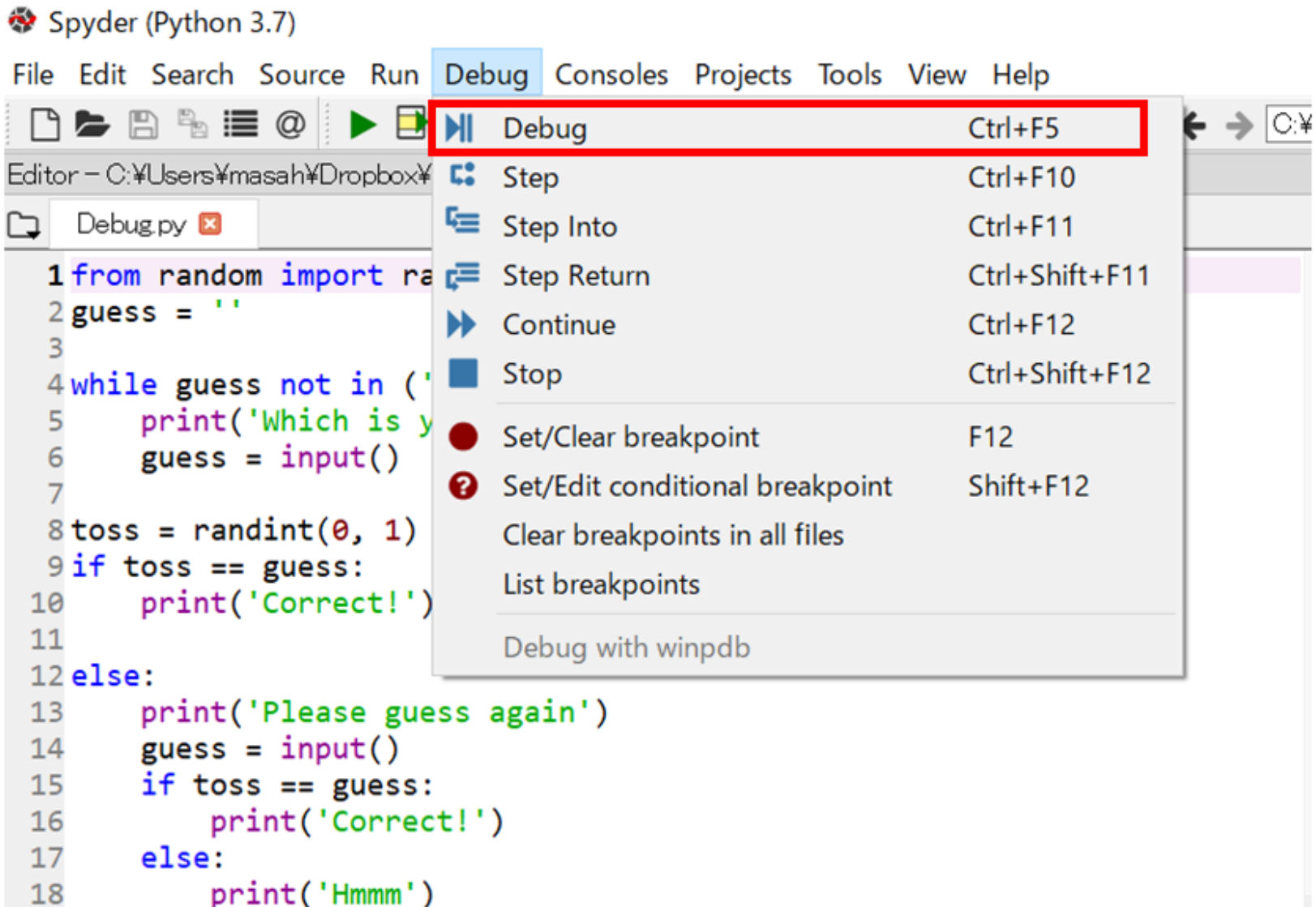
The below code attempts to guess head or tail twice to obtain "`Correct!`" in the end. However, it contains a mistake.

```
# Example: guess either head or tail
In : from random import randint
...: guess = ''
...: while guess not in ('head', 'tail'):
...:     print('Which is your guess, head, or tail:')
...:     guess = input()
...: toss = randint(0, 1) # 0 = tail & 1 = head
...: if toss == guess:
...:     print('Correct!')
...: else:
...:     print('Please guess again')
...:     guess = input()
...:     if toss == guess:
...:         print('Correct!')
...:     else:
...:         print('Hmmm')
Which is your guess, head, or tail:

head

Please guess again

tail
Hmmm
```

Something wrong happened even though we can run code successfully. What is the issue? Let's debug the code.

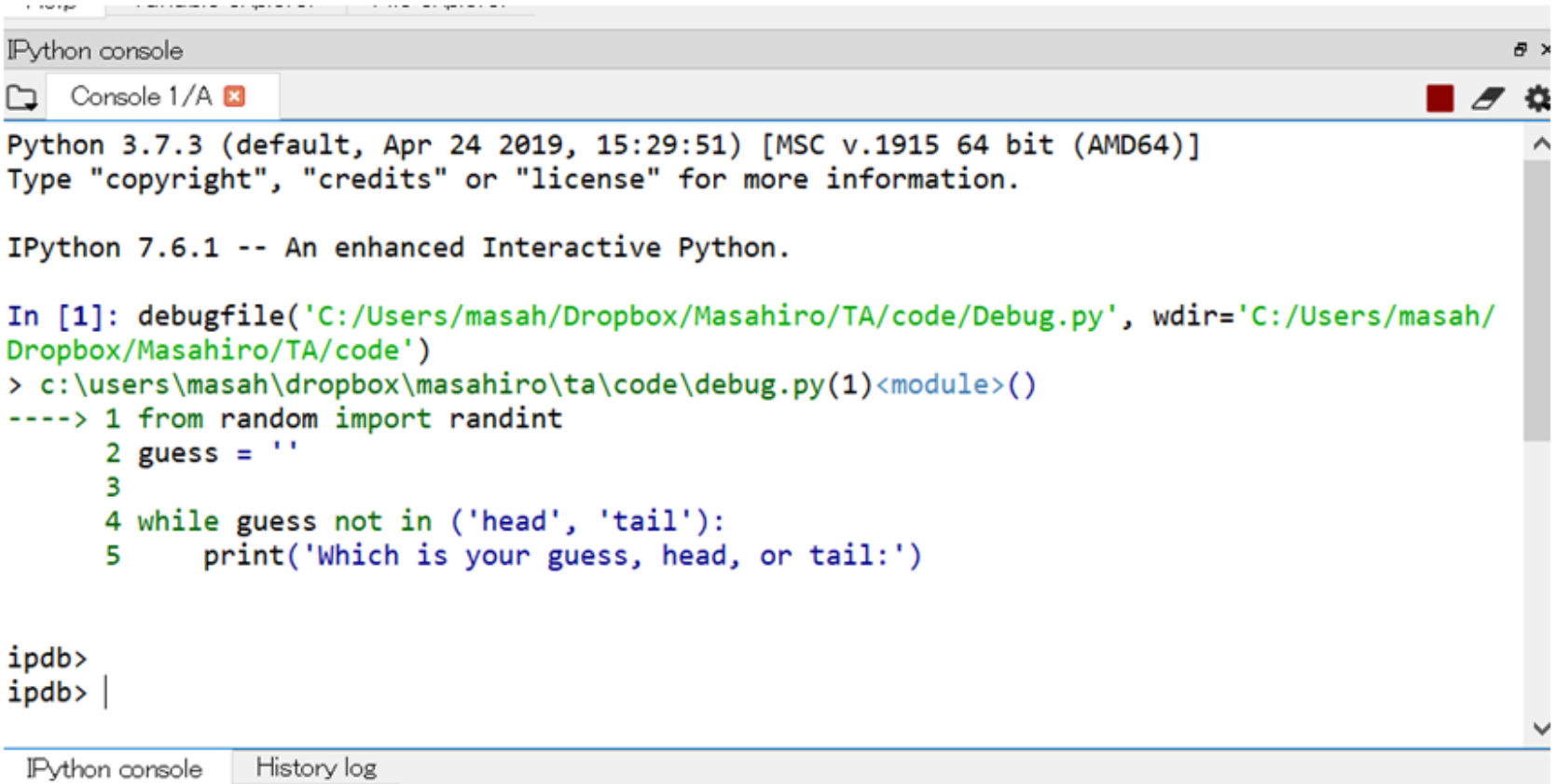
How to Debug

First, click "`Debug`" in the red-lined frame.



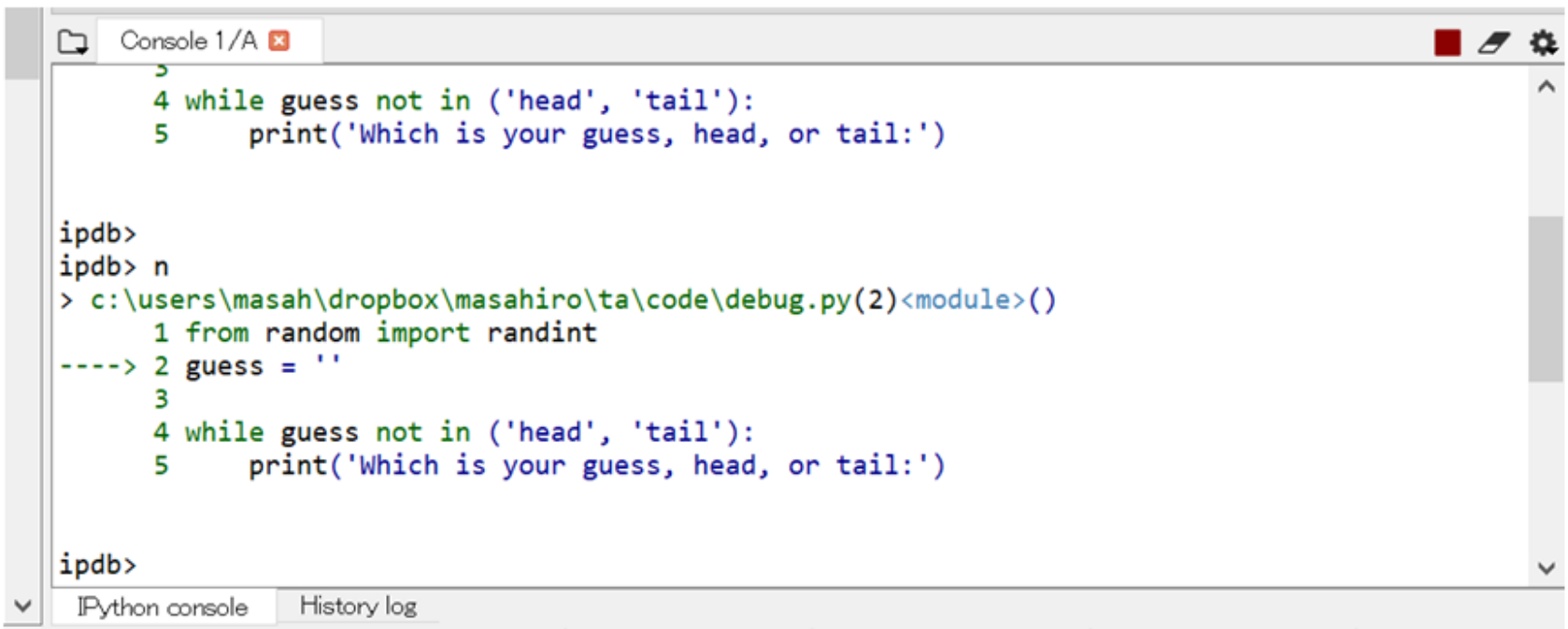
(Cont'd) How to Debug

Your IPython console should look like below. We are in debugging mode now.



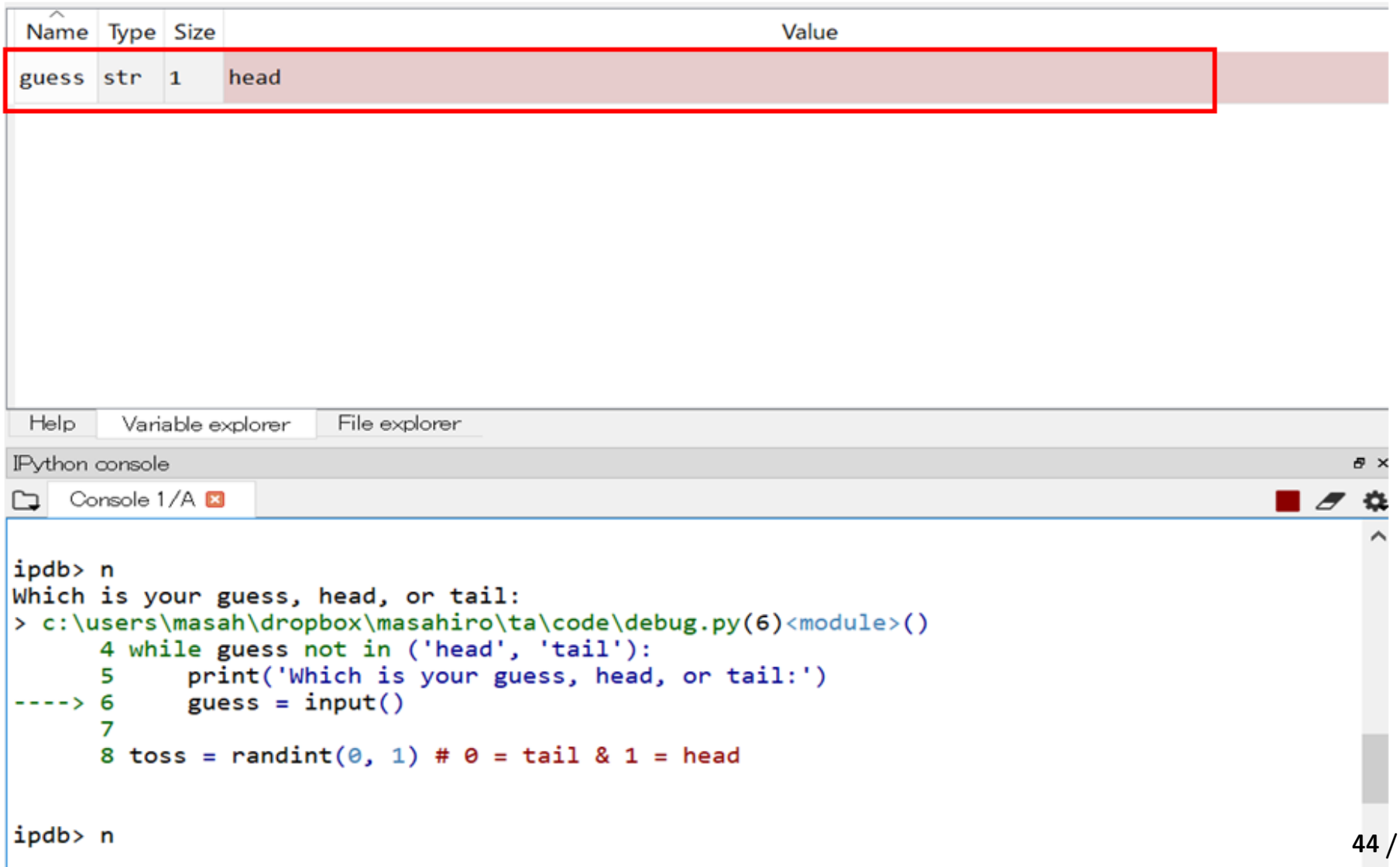
(Cont'd) How to Debug

Type `n` in IPython console or click "**`Step`**" to move to a next line.
The rightwards dotted arrow indicates that we are in the second line.



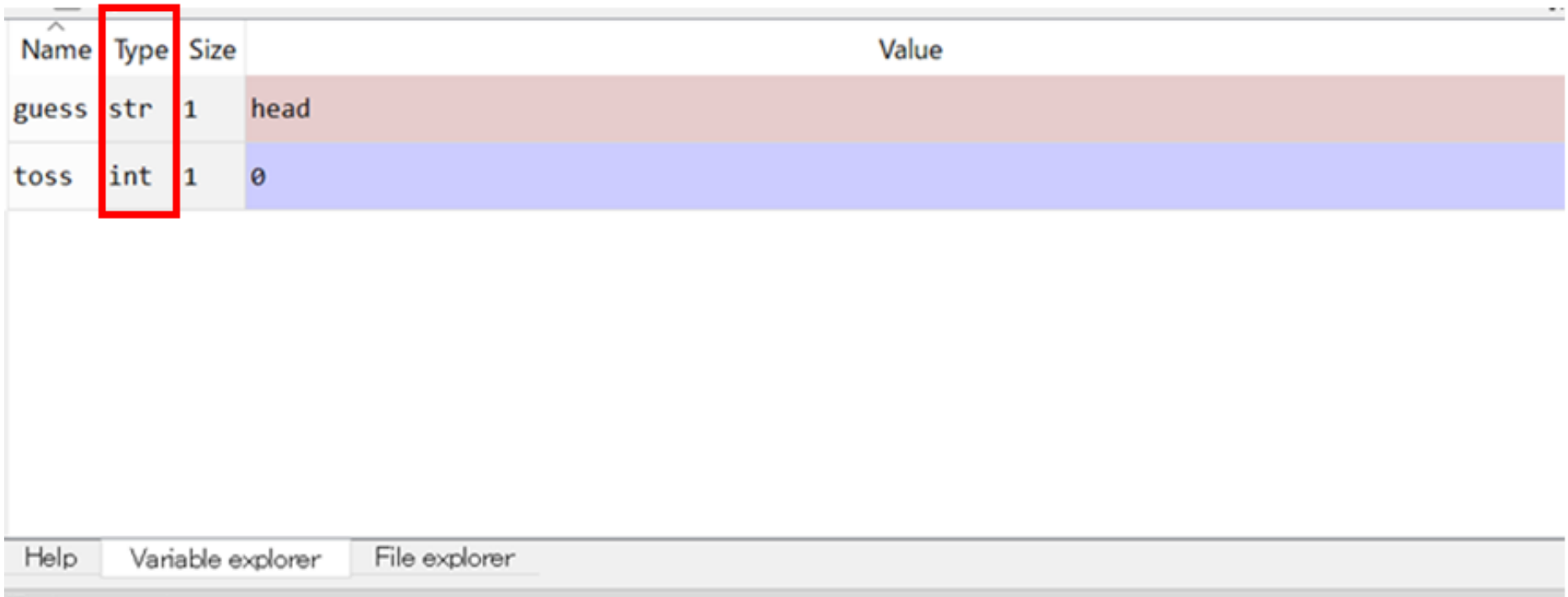
(Cont'd) How to Debug

Once we reach here, let's type **`head`**. You can find the variable **`guess`**, whose value is **`head`** in the variable explorer.



(Cont'd) How to Debug

Move to the line "`toss=randint(0,1)`". Here is the problem: The variables "`guess`" and "`toss`" are different types from each other, therefore, the condition "`toss == guess`" returns false.



Let's end the debugging mode by either:

1. Type `c` (or click `Continue`) to run your whole code in debugging mode and end the debugging mode, or
2. Type `q` (or click `Stop`) to get you out of debugging mode immediately.

Other options

- "**s**" command (or click **Step Intro**): look into function code
- "**r**" command (or click **Step Return**): skip to the end of the function. This is similar with "continue", but this runs to the only end of the function.
- "**l**" command: tell where you are in your whole script.
- "**j <line number>**" command: jump to the specified line.

- In debugging mode, if you type **help** in IPython console, you could know other options. Type **help** ⟨**command**⟩ to know what each command means.

Debugging Example 2

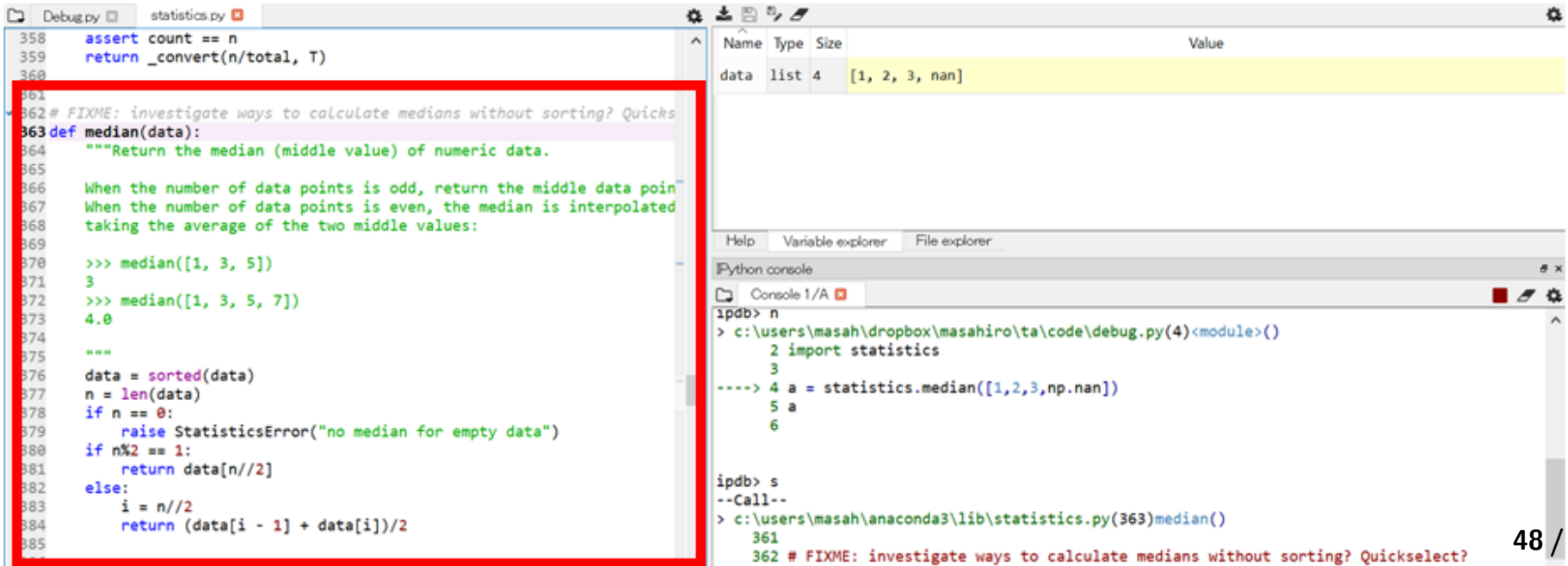
```
In : import numpy as np
...: import statistics
...:
...: a = statistics.median([1, 2, 3, np.nan])
...: a
Out: 2.5 # This should be 2!
```

- The researcher expected to obtain an interger as the median...
- Let's debug the code again. It seems we should look into what the function **`statistics.median`** looks like.
- Let's step to the line of implementing the median function (using **`j <number>`** & **`n`** commands).

```
ipdb> n
> c:\users\masah\dropbox\masahiro\ta\code\debug.py(4)<module>()
      2 import statistics
      3
----> 4 a = statistics.median([1,2,3,np.nan])
      5 a
      6
```

Debugging Example 2

- Type `s` (or click "`Step Into`") at this line, so that the file `statistics.py` which contains the `median` function shows up. (IPython console also starts debugging the function.)
- The `median` function recognizes `nan` as one number and returns a weird result. We found the issue here, so close the debugging mode.



Breakpoint

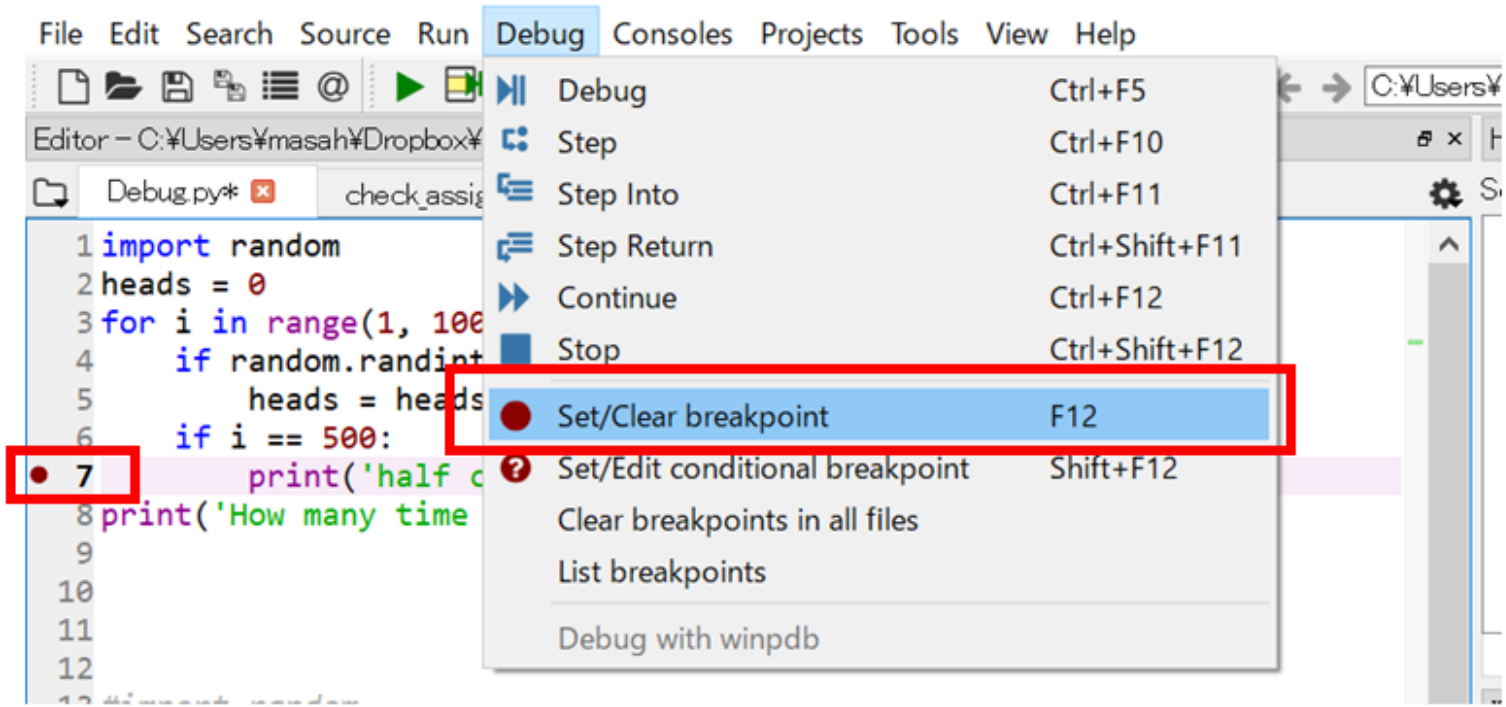
- It may take a lot of time to reach to where you want to debug if you debug line by line.
- Breakpoint would be helpful for you to reach where you want to debug.

Suppose that we want to know how many times we have heads at the half (at the end of the 500th trial in this example). However, if you debug line by line, it takes a lot of time to reach the 500th trial when we have a loop.

```
# One thousand times of coin flips.
import random
heads = 0
for i in range(1, 1001):
    if random.randint(0,1) == 1:
        heads = heads + 1
    if i == 500:
        print('half completed!')
print('How many times we have heads:{0}'.format(heads))
```

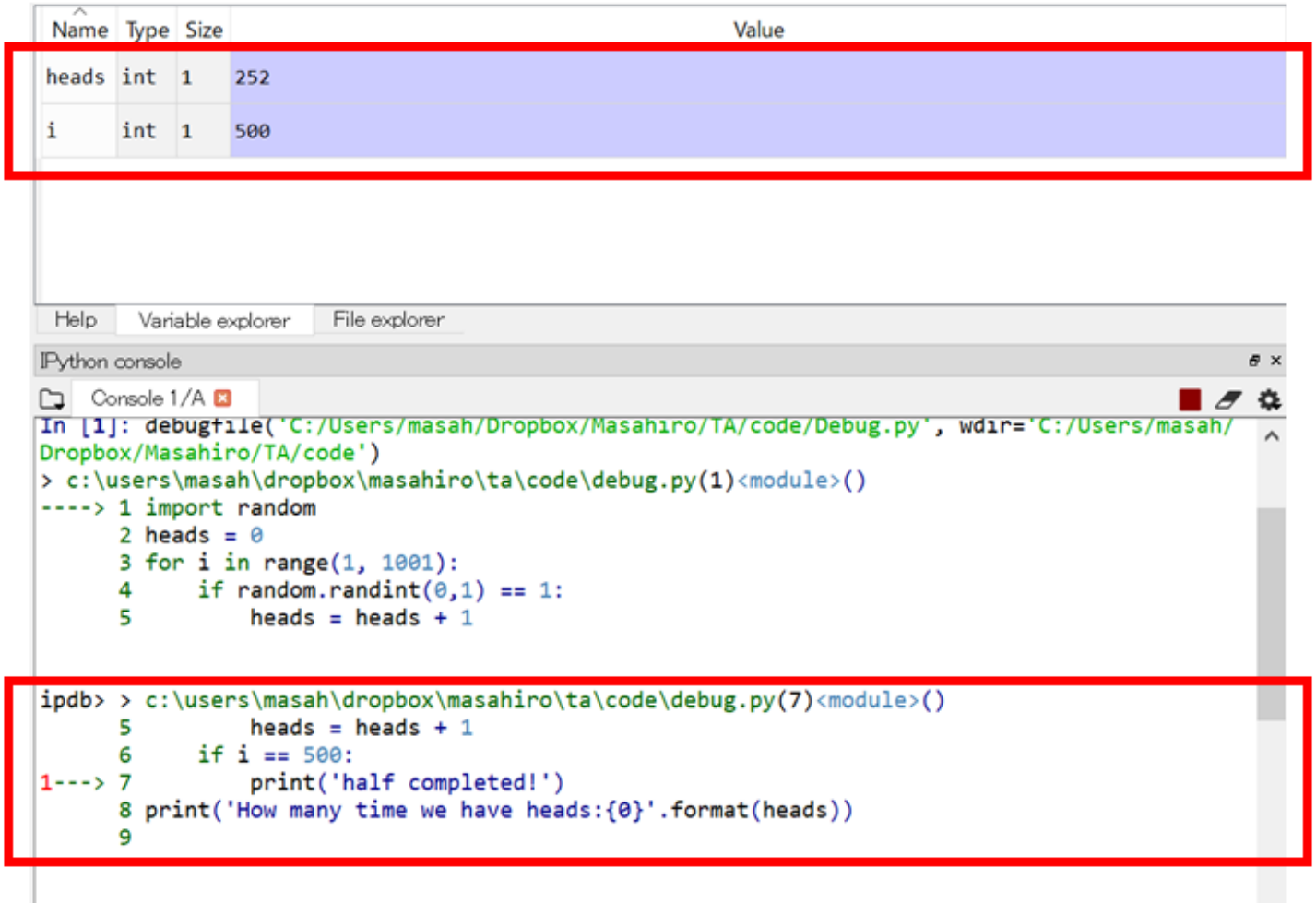
(Cont'd) Breakpoint

Let's set a breakpoint in the 7th line by double-clicking on the point where we have the red point or by clicking "Set/Clear breakpoint" after highlighting the 7th line. Select "Debug".



(Cont'd) Breakpoint

After starting debugging, we immediately jump to "i == 500". As you can see, in the variable explorer, we have an outcome in the end of the 500th trial. (FYI: Explore also the conditional breakpoint method.)



Handling Exceptions

```
import numpy as np
numlist1 = np.random.randint(0, 10, size=[10, 1])

for i in range(0, len(numlist1) + 1):
      print(numlist1[i])
# IndexError: index 10 is out of bounds for axis 0 with size 10
```

Once we get errors, Python stops running the code...

- The **"try" and "except" block** is used to catch and handle exceptions.
- This is a similar block with "capture" and the following "if/else" block in Stata.

Code Example: "try" and "except"

```
import pandas as pd

# Create data frame
Rwanda_income = pd.DataFrame({"year": [1, 2, 3], "income": [100, 200, 300]})
Tanzania_income = pd.DataFrame({"year": [1, 2], "income": [200, 300]})
Uganda_income = pd.DataFrame({"year": [1, 2, 3, 4], "income": [300, 200, 300, 400]})

# Generate "country" column
Rwanda_income['country']   = "Rwanda"
Tanzania_income['country'] = "Tanzania"
Uganda_income['country']   = "Uganda"

# Suppose a researcher recognizes that the data covers 4 years
panel_length = 4

# Create "new_id" by dataframe
# Get error!
for df in [Rwanda_income, Tanzania_income, Uganda_income]:
    df["new_id"] = ""
    for x in range(panel_length):
        df.loc[x, "new_id"] = df["country"][0] + str(df["year"][x])

# Instead, this way can work
for df in [Rwanda_income, Tanzania_income, Uganda_income]:
    df["new_id"] = ""
    for x in range(panel_length):
    #Handling exception errors because of different sizes among dataframes
        try:
            df.loc[x, "new_id"] = df["country"][0] + str(df["year"][x])
        except:
            cntry = df["country"][0]
            print(f"{cntry} : year(index) = {x} data does not exist")
```

Code Example: "try" and "except"

```
# Note that this is a toy example of how to use handling exceptions.
for df in [Rwanda_income, Tanzania_income, Uganda_income]:
    df["new_id"] = ""
    for x in range(panel_length):
    #Handling exception errors because of different sizes among dataframes
        try:
            df.loc[x, "new_id"] = df["country"][0] + str(df["year"][x])
        except:
            cntry = df["country"][0]
            print(f"{cntry} : year(index) = {x} data does not exist")

# This error can be avoided entirely by using the actual panel_length of each
# dataframe rather than assuming it is 4.
for df in [Rwanda_income, Tanzania_income, Uganda_income]:
    df["new_id"] = ""
    for x in range(len(df)):
        df.loc[x, "new_id"] = df["country"][0] + str(df["year"][x])

# This code can be further improved by generating columns without looping through rows.
for df in [Rwanda_income, Tanzania_income, Uganda_income]:
    df["new_id"] = df["country"] + df["year"].astype(str)
```

Handling exceptions can be very useful in practical applications:

- You may want your numerical optimization code to keep going even if the algorithm fails to find a result in one instance.
- In web scraping you may want to try scraping all possible URLs generated from known patterns even if some don't exist.

Unit Test

- Writing a test in your script would take less time than testing manually in the interpreter
- Unit test is to check if the piece of code works as you intend
- How to implement unit test (recall the OOP!):
  1. Import `unittest`
  2. Create a class that inherits `unittest.TestCase`
  3. Within the inherited class, describe test cases using methods `.assertXXX`
  4. Implement the set of tests by `unittest.main()`

Unit Test Example: `statistics.median`

```
import unittest
import numpy as np
import statistics as stats

# Define a class for unit test
class Test(unittest.TestCase):

    # Test for tlist
    def test_list(self):
        tlist  = [1, 2, 3, 4, 5]
        result = stats.median(tlist)   #store the result of stats.median
        self.assertTrue(result == 3)   #test if stats.median returns 3

    # Test for ttuple
    def test_tuple(self):
        ttuple  = (6, 7, 8, 9, 10)
        result  = stats.median(ttuple)
        self.assertTrue(result, int)   #test if this returns an integer

    # Test for Nanlist
    def test_Nan(self):
        Nanlist = [1, 2, 3, 4, 5, np.nan]
        result  = stats.median(Nanlist)
        self.assertTrue(isinstance(result, int))   #test if this returns an integer

# Execute Unit Test
if __name__ == '__main__':
    unittest.main()
```

- Now you get test results similar to below.

```
======================================================================
FAIL: test_Nan (__main__.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "YourFolderPath/UnitTest.py", line 29, in test_Nan
    self.assertTrue(isinstance(result, int))
AssertionError: False is not true

----------------------------------------------------------------------
```

- In this example, if you forget to drop missing values and use statistics module, you get erroneous summary statistics.
- Recall that code should be self-documenting — Instead of adding a comment sentence like "Check that the result of taking median is integer!", implement this test
- It's good to test both functions from standard modules and your own functions.

Robustness & Verifiability

- Robustness is the ability of a computer system to cope with errors during execution.
- Your programs should run correctly or crash when meeting unexpected circumstances.
  - Crashing is much better than reporting something unintended.
  - E.g.) `median{1, 2, 3, nan} = ERROR!` is better than `median{1, 2, 3, nan} = 3`.
- Being able to look at the output later and tell how it was generated, as well as see if it is correct, is important.
- Log and diagnostic files are helpful for this.

Logging

The print command works fine for short code like below.

```
import numpy as np
numlist1 = np.random.randint(0, 10, size=[10, 1])
for i in range(0, len(numlist1) + 1):
    try:
        print(numlist1[i])
    except:
        print("Exception occurred")
```

But, what if we have longer code and more exceptions? $\Rightarrow$ Logging!

- Logging is provided as a standard library module.
- Logging records when your code runs and what it did in a produced log file.
- Logging can set various levels of errors: debug (the lowest level), info, warning, error, and critical (the highest level).
- See Logging facility for Python for more detail.

How to Log

1. Create a log file.
2. Set up log format: what you want Python to write down.
3. Decide a logging level:
   - Python writes down higher logging levels in a log file than the logging level you set.
   - If you set the level debug as your default, then Python writes down all the logging levels in a log file.
4. Name your logger object.
5. Put your logger where you want Python to write down a result in a log file.

Let's look at the above through code example.

```
# Spyder users need to run these three lines every time you use the logging module.
from importlib import reload
import logging # Spyder cannot read logging module properly by only import.
reload(logging) # Tell Spyder to read logging module again.

# Alternately, you can try adding 'force = True' to the basicConfig function in the example
```

Note. See this link for a bug about logging in Spyder.

Logging: Code Example

```
import logging
import numpy as np

# Log File
logfile = r'pathTo\logger.log'  #Use relative path

# Log Format: asctime = time record, name = logger name,
# levelname = logging level for the message (e.g. debug),
# lineno = line number in your script, message =  the logged message
fmt = '%(asctime)s - %(name)s - %(levelname)s - %(lineno)s - %(message)s'

#Set a logging level to debug
logging.basicConfig(filename=logfile, level=logging.DEBUG, format=fmt)

# "w+" means overwriting a log file.
logging.FileHandler(logfile, "w+")

#Logger object: logger name
logger = logging.getLogger("ECON2020")

# Prepare random dataset
numlist1 = np.random.randint(0, 10, size=[10, 1])
for i in range(0, len(numlist1) + 1):
    try:
        print(numlist1[i])
    except IndexError:
        logger.error("Exception occurred", exc_info=True)

logging.FileHandler(logfile).close() # close your log file.
logging.shutdown() # close logging
# See your log file in your path.
```

Logging: Back to In-Class Exercise

- Let's change the below part (*) of in-class exercise code to improve its accuracy.
- We want to check if there is not any duplicated index in (*).
- Rewrite code so that we can check it in a log file after running the whole code.

```
## Suppose we have a logger object already.

## Before
## Check the duplicated index (Check if it's zero!)
ged_Africa_groups[ged_Africa_groups.index.duplicated()]  #(*)

## After
## Log file lets us know when we do no have any duplicated index.
if sum(ged_Africa_groups.index.duplicated() == True) == 0:
    logger.debug("No duplicated index!")
else:
    logger.error("ERROR: duplicated index!")
```

Efficiency

Efficiency

Conserve computing resource and save computing time.

Remark. Make your efforts toward improving efficiency **as far as you can ensure and maintain accuracy**.

Techniques for improving efficiency:

1. **Vectorization**
2. **Parallelization**
3. **High Performance Computing (HPC)**

Other tips for improving efficiency:

- Keep in mind the trade-off between computing time and storage.
- Separate slow and fast code. (Similar motivation to a message at the portability section: Separate directories by function.)
- Check accuracy first without taking a long time. E.g.) Use a subsample of a large dataset to run everything; Solve a GE model in a narrower geographical area than the entire study area.

Vectorization

Vectorize everything you can.

Replace loops and multiple equations with:

- applying a procedure to multiple items
- matrix algebra

In order to do that,

- Know your data structure well
- Manipulate your data structure wisely to facilitate vectorizing

Vectorizing can shorten your code and save computation time.

Vectorization: Example

Imagine a dataframe with three columns of data:

ID	var1	var2	var3
1	$var1_1$	$var2_1$	$var3_1$
2	$var1_2$	$var2_2$	$var3_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Suppose that given constant weights for each variable you now wish to generate a weighted sums column:
$weightedsum_i = var1_i * weight1 + var2_i * weight2 + var3_i * weight3$

- You can of course loop through and calculate this row by row
- This can be sped up by using pandas apply() instead
- But the fastest way would be to vectorize this operation

(Cont'd) Vectorization: Example

To vectorize, note that the weighted sum is equivalent to the diagonal of the variables matrix multiplied by the transpose of the weights matrix:

$$weightedsum = Var * Weights^T$$

where

$$\text{Var} = \begin{bmatrix} var1_1 & var2_1 & var3_1 \\ var1_2 & var2_2 & var3_2 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

and

$$\text{Weights} = \begin{bmatrix} weight1 & weight2 & weight3 \end{bmatrix}$$

Let's test the performance of the three methods with code.

(Cont'd) Vectorization: Example

```
import pandas as pd
import numpy as np
import time

data = pd.DataFrame(np.random.randint(0, 10, size = [10000, 3]),
                    columns = ["var1", "var2", "var3"])
weights = pd.DataFrame(np.random.randint(0, 10, size = [1, 3]),
                    columns = ["weight1", "weight2", "weight3"])
data_iter, data_apply, data_vector = data.copy(), data.copy(), data.copy()

# Iterrows()
start = time.time()
data_iter["weightedsum"] = 0
for ind, row in data_iter.iterrows():
    data_iter.at[ind, "weightedsum"] = row["var1"]*weights["weight1"] + \
    row["var2"]*weights["weight2"] + row["var3"]*weights["weight3"]
print("Iterrows:", time.time() - start)

# Apply()
start = time.time()
data_apply["weightedsum"] = data_apply.apply(lambda row: row["var1"]*weights["weight1"] + \
                            row["var2"]*weights["weight2"] + row["var3"]*weights["weight3"],
print("Apply:", time.time() - start)

# Vectorization
start = time.time()
A = np.array(data_vector[["var1", "var2", "var3"]])
data_vector["weightedsum"] = A@(weights[["weight1", "weight2", "weight3"]]).T
print("Vectorization:", time.time() - start)
```

Parallelization

Multiple processors (CPUs) of a computer execute tasks simultaneously.

When is Parallelization Useful in Economics?

- Simulations
  - Compute equilibrium across independent markets, under different initial conditions, parameter values, shock realizations
- Bootstrapping standard errors
  - Randomly create hypothetical data set ("bootstrapped sample") many times, obtain the estimator from each bootstrapped sample, and obtain its empirical distribution
- Dynamic programming (with large state space)
  - Evaluate value functions under many different states within k-th step of value function iteration

Note. Not all tasks are parallelizable: We cannot parallelize across tasks/steps/iterations if the current one requires a previous one's output as an input

Parallelization: Remark

Parallelization does not always save computation time. Care about **granularity** and **scalability** of the problem when implementing parallel processing.

- Granularity: The communication cost outweighs when the division of tasks is too fine. That is, you may not benefit from parallelizing when working with small datasets.
- Scalability: Scalable problems are easy to parallelize.
- Also, care about independence across tasks that are parallelized.

Quiz: Sequential vs. Parallel Processing

- Which code runs faster: Code A or Code B?
- Both of them will do the same job. The only difference is whether using parallel processing or not.

```
######## Code A ##########
import time, math, itertools
import numpy as np
#Define a function: calculate distance between two points
def dist(inputs):
    x, y = inputs
    distance = math.sqrt((x[0]-y[0])**2 + (x[1]-y[1])**2)
    return distance
if __name__ == '__main__':
    # Prepare data
    np.random.seed(1)
    loc1 = np.random.randint(0, 10, size=[10, 2])     #Generate data: [10,2]
    np.random.seed(2)
    loc2 = np.random.randint(0, 10, size=[20000, 2]) #Generate data: [20000,2]
    # Generate all pairs in loc1 & loc2
    pair = [(coor1, coor2) for coor1, coor2 in itertools.product(loc1, loc2)]
    result =[] # store results
    start = time.time()
    for i in pair:
        result.append(dist(i)) #Store results from the function
    end = time.time()
    print(end - start)
```

(Cont'd) Quiz: Sequential vs. Parallel Processing

```
######## Code B ##########
import time, math, itertools
import numpy as np
from multiprocessing import Pool
#Define a function: calculate distance between two points
def dist(inputs):
    x, y = inputs
    distance = math.sqrt((x[0]-y[0])**2 + (x[1]-y[1])**2)
    return distance
if __name__ == '__main__':
    np.random.seed(1)
    loc1 = np.random.randint(0, 10, size=[10, 2])
    np.random.seed(2)
    loc2 = np.random.randint(0, 10, size=[20000, 2])
    pair = [(coor1, coor2) for coor1, coor2 in itertools.product(loc1, loc2)]
    result = [] # store results
    start = time.time()
    pool = Pool(processes=4) # start 4 processes(cannot exceed the maximum processors).
    result = pool.map(dist, pair)  # synchronous processing
    pool.close() #Prevents any more tasks from being submitted to the pool
    pool.join()   #Wait for the worker processes to exit
    end = time.time()
    print(end - start)
```

Note. `os.cpu_count()` or `multiprocessing.cpu_count()` shows the number of processors in your computer.

Note. `if __name__ == '__main__':` should be written in your script when you use multiprocessing. This clarifies what the current main program is and prevents Python from running unintended function files. Windows users in particular must write this when you use multiprocessing since, for example, newly spawned processes try to spawn other new processes. See this discussion.

(Cont'd) Quiz: Sequential vs. Parallel Processing

- Python runs Code B slower than Code A, despite using parallel processing in Code B!
- In this example, we use synchronous execution, in which your computer blocks next processing until a result from each process is ready. A main processor communicates with other processors after each processing during execution.
- Recall the granularity and scalability:

  In this example, once you increase the scale of the task of computing distances (say, from 200,000 pairs to 20,000,000 pairs), you can easily see that the computation becomes much faster with multiprocessing (i.e., the benefit from dividing tasks outweighs the communication cost).

E.g.) Sequential vs. Parallel Processing

- One more example where parallelization makes it faster
- The program calculates distance between a pair of random points. Every time it calculates distance, it calculates 5000 inverse matrices.
- `pool.map()` enables multiple processors to work at the same time by sharing the whole task. That is, they start executing their given tasks such as running loops for 5000 inverse matrices at the same time, which reduces the computation time significantly.

Note. Another example where multiprocess execution really beneficial could be the case when you use sleep function. Scraping usually requires sleep function to move to a next task. Intuitively, one process with 60 sec sleep in total can divide into 30 sec sleep for each of two processes.

(Cont'd) E.g.) Sequential vs. Parallel Processing

```
from multiprocessing import Pool
import os, time, itertools, math
import numpy as np
## This func calculates distance between two points and
## in the process, it calculates 5000 inverse matrices.
def dist(inputs):
    x, y = inputs
    distance = math.sqrt((x[0] - y[0])**2 + (x[1] - y[1])**2)
    ## Calculate 5000 inverse matrices
    for i in range(0, 5000):
        np.random.seed(i)
        np.linalg.inv(np.random.randint(1,10, size=[100,100]))
    return distance

if __name__ == "__main__":
    ## Generate random pairs of two points
    np.random.seed(1)
    loc1 = np.random.randint(0, 10, size = [5, 2])
    np.random.seed(2)
    loc2 = np.random.randint(0, 10, size = [5, 2])
    pair = [(coor1, coor2) for coor1, coor2 in itertools.product(loc1, loc2)]
    ## Sequential Execution
    start_time = time.time()
    result_seq = []
    for item in pair:
        result_seq.append(dist(item))
    print ("Sequential execution in " + str(time.time() - start_time), "seconds")
    ## Synchronous Execution (pool.map)
    start_time_map = time.time()
    pool = Pool(processes = os.cpu_count())
    result_map = pool.map(dist, pair)
    pool.close()
    pool.join()
    print ("Process pool map execution in " + str(time.time() - start_time_map), "seconds")
```

Choice of Synchronization

Synchronous execution:

- The master processor gives out a set of commands to the other processors and waits for all of them to report before issuing the next set of commands.
- The processes are completed in the same order in which execution was started. A returned result is ordered.
- This is used in the previous quiz and example.

Asynchronous execution:

- A processor can be given a new task independent of the progress the other processors have made with their old tasks.
- The order of results *could* get mixed up, but computation gets done quickly in some cases.
- `pool.apply_async` enables a processor to work for a next task without waiting for the other processors to finish previous tasks.

E.g.) Asynchronous Execution

The program computes the sum, $0^2 + 1^2 + 2^2 + ... + 19^2$. Every time it calculates a square, it calculates 5000 inverse matrices.

```
import multiprocessing as mp
import numpy as np
import os, time
# This func just returns x, but in the process, it calculates 5000 inverse matrices.
def identity(x):
#  Calculate 5000 inverse matrices
    for k in range(0, 5000):
        np.random.seed(k)
        np.linalg.inv(np.random.randint(1,10, size=[100,100]))
    return x

# This func returns the sum from 0**n + 1**n + ... + end_num**n
def sum_power_n_list(n, end_num):
#   Here a = 0 does not work.
    a = [0]
#   This is a callback func to add each term.
#   Without this, pool.apply_async returns nothing.
    def foo(x):
        a[0] += x ** n

    pool = mp.Pool(processes = os.cpu_count())
    for m in range(end_num):
        pool.apply_async(identity, (m,), callback = foo)
    pool.close()
    pool.join()
    print(a[0])

if __name__ == "__main__":
    start_time_apply_async = time.time()
    sum_power_n_list(2, 20) # n = 2 & end_num = 20
    print ("Apply Async execution in "+str(time.time() - start_time_apply_async),"seconds")
```

Callback Function

- In asynchronous execution (e.g., `pool.apply_async()`, `pool.map_async()`), we can set a callback function.
- A callback function helps you with storing returned results as you like. As in the below code example, you can use a list to store the result. You could also store it in a dictionary (e.g. with the function name as a key and returned result as its value) by using a callback function.

```
result = [] # store a result
def callback(values):
    result.extend(values) # store the result in the list ''result'' above.
start = time.time()   # Start time
pool = Pool(processes=4)              # Start 4 processes
pool.map_async(dist, pair, callback = callback)
pool.close()
pool.join()
```

- See What is a callback?

Note. You can also get results with methods (e.g., `get()`) instead of callback function. See this for details.

FYI: Chunksize

- Chunksize is workload assigned to each worker at a time. It is hard to say how to choose optimized chunksize in every case.
- See the discussions: Pool's chunksize-algorithm and Data and chunk sizes matter.

Parallel Processing: Further Readings

We have covered only very basics of parallel processing so far. For more detail, see for example:

- *Judd, Chap 2.2.*
- `multiprocessing` — Process-based parallelism (documentation)
- Multiprocessing to Make Python Code Faster
- Parallel Processing in Python
- Slides by Fernández-Villaverde, Guerrón, and Zarruk Valencia
- Practical guide by Fernández-Villaverde and Zarruk Valencia

High Performance Computing (HPC)

- Even if you have written your code nicely to save computing time, many tasks will take too long using affordable laptops
- In such situations, learn how to use HPC clusters, which should be available at many research universities
    - Oscar: Brown University's Supercomputer
- Also, useful to know how to run code from Shell/Terminal rather than in a GUI (graphical user interface)

How to Use Oscar (for Brown Faculty/Students)

- Create a new account for Oscar (from Quickstart Guide)
- Log in using Secure Shell (SSH):
    - Mac/Linux: use Terminal
    - Windows: use PuTTY, a free Secure Shell (SSH) client
- Open OnDemand (OOD): You can also use a web portal to run GUI-based applications (e.g., Stata, Matlab, Rstudio)
- You can choose either interactive jobs or batch jobs depending on job types. See the below links for details.
    - Interactive Jobs : Small jobs with short run times and jobs that require the use of a GUI. If your connection to the system is interrupted, the job will abort.
    - Batch Jobs : Your program can run for long periods of time in the background, so you do not need to be connected to Oscar.

Note. Do not run CPU-intense or long-running programs directly on the login nodes! The login nodes are shared by many users, and you will interrupt other users' work. To get to the compute nodes from the login nodes you can either start an interactive session on a compute node, or submit a batch job.

Note. For batch jobs, Windows users should see this for writing a batch sctipt.

(Cont'd) How to Use Oscar (for Brown Faculty/Students)

- List all available modules in Oscar:

```
$ module avail
```

  To run code for Python 3 using Anaconda, need to type:

```
$ module load python/3.7.4
$ module load anaconda/2020.02
$ conda init bash
```

  To create a new environment and/or to install packages using conda, see this

- Create a new directory on Oscar (~ means **/users/username**) and change current directory to **/users/username/code**:

```
$ mkdir ~/code
$ cd ~/code
```

- Several ways to transfer files (code, data, outputs) between your machine and Oscar. I use FileZilla, a GUI program for Windows/Mac/Linux. Inside FileZilla, we can also manage directories on Oscar.

- Run interactive jobs (load python/3.7.4 etc after reaching the interactive session)

```
$ interact
$ python yourfile.py
```

- Run batch jobs. See this for how to write and run a batch file.

```
$ sbatch yourfile.bat
```

Reproducibility

Reproducibility

So far, we have learnt how to improve research productivity in fine steps. But, how to make the research **reproducible**? i.e.,

Q. How can we handle thousands of processes from raw data into a final result in a manageable way?

Q. How can we track back thousands of revisions of our work?

Techniques for improving reproducibility:

1. **Automation**
2. **Version control**

Automation: Motivation

Q. How can we handle thousands of processes from raw data into a final result in a decent way?

- E.g.) Recall our ethnic conflict research project introduced in the portability section
- Let's consider the process from raw data into the dataset for econometric analysis
- Code in the "build" folder execute this process

Automation: Motivation

Q. How can we handle thousands of processes from raw data into a final result in a decent way?

- E.g.) Recall our ethnic conflict research project introduced in the portability section
- Let's consider the process from raw data into the dataset for econometric analysis
- Code in the "build" folder execute this process

```
================================================================================
Project_EthnicConflicts/codes/build/
 conflicts_event_to_group.py
   (Construct a rebel group-level conflict event dataset from the UCDP conflict event dataset)
 conflicts_event_to_group_match.py
   (Combine rebel-level conflict event with matching status and construct some GIS baseline data)
 match_ethnologue_Islam_groups.do
   (Construct matching results of ethnic groups corresponding to rebels from by-hand matching)
 rebel_homeland.py
   (Construct GIS data of rebel groups' homelands from matching results and Ethnologue map)
 rebel_cell_time_vars.py
   (Construct all rebel-cell-time-level variables used for econometric analysis)
================================================================================
```

Automation: Problem

```
================================================================================================
Project_EthnicConflicts/codes/build/
 conflicts_event_to_group.py
   (Construct a rebel group-level conflict event dataset from the UCDP conflict event dataset)
 conflicts_event_to_group_match.py
   (Combine rebel-level conflict event with matching status and construct some GIS baseline data)
 match_ethnologue_Islam_groups.do
   (Construct matching results of ethnic groups corresponding to rebels from by-hand matching)
 rebel_homeland.py
   (Construct GIS data of rebel groups' homelands from matching results and Ethnologue map)
 rebel_cell_time_vars.py
   (Construct all rebel-cell-time-level variables used for econometric analysis)
================================================================================================
```

- Messy. It will be messier as the project develops further.
- What if we revise one code of them, but forget running others?
- Or, what if we revise one code, but forget the order of running others? Results might be affected by running these code in a mistaken order...

Automation: Some Improvement

```
==========================================================================================
Project_EthnicConflicts/codes/build/
 1_conflicts_event_to_group.py
 2_match_ethnologue_Islam_groups.do
 3_conflicts_event_to_group_match.py
 4_rebel_homeland.py
 5_rebel_cell_time_vars.py
==========================================================================================
```

- Recall clarity: File names themselves should indicate their means and objectives as much as possible.
  The above naming makes the order clear...

  Whenever we revise one code, we know which code should also be rerun and their order...

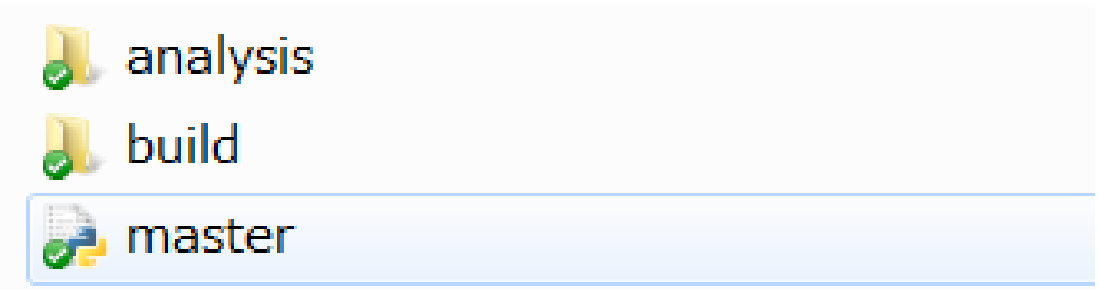
- Still a little messy: run the revised code, wait until it finishes running, run the next code, go and back between Python and Stata, etc... Is that an inevitable process?

- **No!** There's still a much better way!

Automation: Solution

```
================================================================================
Project_EthnicConflicts/codes/build/
 conflicts_event_to_group.py
 conflicts_event_to_group_match.py
 match_ethnologue_Islam_groups.do
 rebel_homeland.py
 rebel_cell_time_vars.py
================================================================================
```

We can avoid the above messy process if **`master.py`** can run everything from **`conflicts_event_to_group.py`** to **`rebel_cell_time_vars.py`** (including Stata **`.do`** file) and all the code in the analysis folder as well at once!



Master Code Example: run_software.py

```
import os, sys,subprocess, re
### You can run stata, python, and arcgis (python 2) by using this module.

def stata(path_stata, script):
    """Run stata dofile in batch mode, and deletes the log file"""
    subprocess.run([path_stata, "/e", "do", script])
    os.remove("{}.log".format(script[0:-3]))  #remove log file
    # Explore by yourself to show line-by-line command in ipython console

def python(path_python, script):
    """Run Python script without arcpy"""
    run = subprocess.run([path_python, script], stdout=subprocess.PIPE,\\
                         stderr=subprocess.STDOUT, universal_newlines=True)
    print(run.stdout)
    # returncode = 0 if code runs successfully.
    if run.returncode != 0:
        # stops when pyfile has an error
        sys.exit("Error : {script} ".format(script = script))

def arcgis(path_python2, script):
    """Run Python script with arcpy"""
    run = subprocess.run([path_python2, script], stdout=subprocess.PIPE,\\
                         stderr=subprocess.STDOUT, universal_newlines=True)
    print(run.stdout)
    if run.returncode != 0:
        sys.exit("Error : {script} ".format(script = script))
```

Master Code Example: `master.py`

```
import glob, os, sys
import run_software # Created a module that enables us to run stata and python2 as well.
path_Stata   = glob.glob("C:\*\Stata*\Stata*.exe")[0]
path_Python2 = glob.glob("C:\Python27\ArcGIS10.*\pythonw.exe")[0]

def main():
    build()
    analysis()

def build():
    os.chdir("build")
    print("Construct a rebel-level conflict event datasets from the UCDP")
    run_software.python(sys.executable, "conflicts_event_to_group.py")

    print("Construct by-hand matching results of ethnic groups corresponding to rebels.")
    run_software.stata(path_Stata, "match_ethnologue_Islam_groups.do")

    print("Combine rebel-level conflict event with matching status.")
    run_software.python(sys.executable, "conflicts_event_to_group_match.py")

    print("Construct polygons for each rebel's homeland using matching results.")
    run_software.arcgis(path_Python2, "rebel_homeland.py")

    print("Construct all rebel-cell-time-level vars for analysis.")
    run_software.arcgis(path_Python2, "rebel_vars.py")

    os.chdir("..")

def analysis():
    os.chdir("analysis")
    # Again, list up code for econometric analyses
    os.chdir("..")

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

Automation

- One button to produce a paper from raw data — Write a single master code that executes all code from beginning to end.
- Run this master code after any revisions of any code.
- The master code can also work well as a guide for the whole research process.
- Python can do this on both Windows and Linux systems. (See GS2014 Chap 2. for Windows/Linux shell script.)

Note. `master.do` in STATA, which calls python code as well, can also work. It seems comfortable with STATA 16. See this. `master.bat` (a batch file that we have learnt at the HPC part) could also work as a master code.

Note. Printing some (but again, minimum amount of) notes might be helpful to check which code inside the master code is running. That also works as a documentation about the entire research structure and flow. At the same time, putting all the relevant information together inside the master code would eliminate internal inconsistency.

Version Control: Motivation

Q. How can we track back thousands of revisions of our works? What can we do if...

- After employing a new empirical strategy, we notice a flaw in the new one and want to go back to the previous strategy...
- A coauthor made code revisions. I want to check which parts have been revised exactly to understand what drove changes in the empirical results...
- After deleting a paragraph in the introduction of a paper, I notice the deleted paragraph is better than the current version of the introduction...

Version Control: Motivation

- Should we keep all revised code after some important revisions and remember which version is up-to-date?
- Do we care who in the research team made the revision?
- Have we revised the master code accordingly?

```
================================================================================
Project_EthnicConflicts/codes/build/
 conflicts_event_to_group_20190321ST.py
 conflicts_event_to_group_20190321ST_rev1.py
 conflicts_event_to_group_20190321ST_rev2.py
 conflicts_event_to_group_20190323MK.py
 conflicts_event_to_group_20190323MK_STCheked.py
 conflicts_event_to_group_match.py
 master_build.py
 master_build_rev.py
 master_build_almostfinal.py
 match_ethnologue_Islam_groups.do
 rebel_homeland.py
 rebel_cell_time_vars.py
 rebel_cell_time_vars_ST.py
 rebel_cell_time_vars_ST_MK.py
================================================================================
```

Dangerous!

Version Control

- Multiple versions of files confuse you and your collaborators and make it difficult to replicate results
- Version control system records and tracks all edits of code (and texts, documents, data, outputs, etc) that you or your collaborators have made through the entire research process
- It provides an organized revision history
- It facilitates going back and forth between multiple versions of the same file, with easy comparisons between them
- It facilitates experimentations for editing any files without fear
- It facilitates collaboration:
    - Collaborators can work on the same project by using their own local directories
    - They share changes they make on their local machines via a web-based repository
    - Collaborators can *simultaneously* work on multiple versions of the same file with different branches and merge them

Git & GitHub

- **Git** : a version control system that stores all the previous versions of files within a project
- **GitHub**: a web-based service to host remote repositories for collaborations that uses the Git system to save the complete history of a project in a cloud

cf.) There are also other such services (e.g., GitLab/Bitbucket), but GitHub is most popular

Setting up Git & GitHub (Again)

1. Create your GitHub account from here. Request the student free plan from here, which allows you to create a free private repository.
2. Install Git from here.
3. Open GitBash (in Windows) or the Terminal (in Mac).
4. When we use Git on a new computer for the first time, we need to configure a few settings, which will be used globally (i.e., for all projects).

   Link Git to your GitHub account: Set your username and email same as in your GitHub account.

```
$ git config --global user.name "stsuda"
$ git config --global user.email "shunsuke_tsuda@brown.edu"
```

Remark. This course uses a command line interface for operating Git.
There are also some graphical interfaces:
Sourcetree/GitKraken/GitHub Desktop/RStudio (for R projects)

Starting a (Collaborative) Research Project

1. Create a remote repository online in GitHub (private):
   "Project_EthnicConflicts"
   Invite collaborators from (Settings → Collaborators).

2. Click on 'Create new file' to create **`.gitignore`** in a main folder:
   - GitHub cannot track a file with >100MB.
   - Specify files/folders which we configure Git to ignore.
   - See here for a collection of useful `.gitignore` templates.
   - Look also into Git Large File Storage for sharing large datasets

3. Each member creates a local copy of this repository:
   Decide its location and then clone it.

```
$ cd d:/Research      (Local Path of Each Research Term Member)
$ git clone https://github.com/stsuda/Project_EthnicConflicts
```

   Go to the directory of the project and check the list of files inside it:

```
$ cd Project_EthnicConflicts
$ ls
```

4. Now, ready to edit anything in this repository. Go to the next slide.

Remark. Do NOT use Git inside a Dropbox shared folder! The remote repository instead plays a role for sharing.

Workflow in a Typical Work Day

1. Guide to your local path to the project:

   ```
   $ cd d:/Research/Project_EthnicConflicts (Local Path to Your Project)
   ```

2. "Pull": Update your local repo by downloading your collaborators' changes from GitHub.

   ```
   $ git pull
   ```

3. Work in your local repo. Make your changes and stage them.

   ```
   $ git add -A
   ```

4. Commit your changes with a comment

   ```
   $ git commit -m "I did something for better visibility"
   ```

   Check if you are forgetting something to commit by:

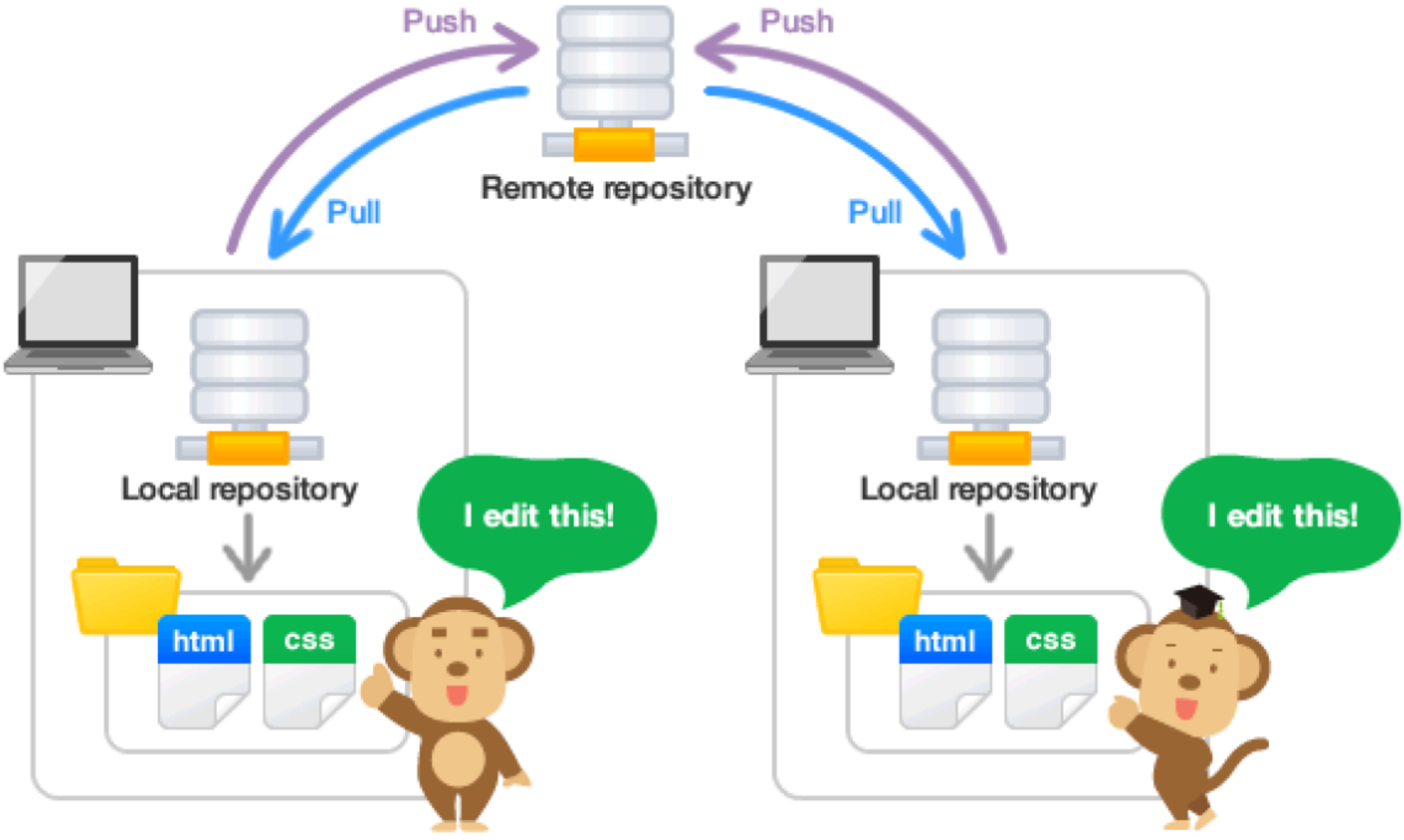
   ```
   $ git status
   ```

   Iterate 2. and 3. until you have done your work on the day.

5. Upload the changes to GitHub.

   ```
   $ git push
   ```

Remark. Commit & comment often for easier tracking! Write a reason of your decision, too.



Source: https://backlog.com/git-tutorial/

Exercise 2

Collaboration Exercise with Git & GitHub

Form groups, each consisting of 2 students. Nominate a group leader.

Aim. Experience how a joint research project goes with Git & GitHub.

0. The leader creates a (private) repository called "`Project_EC2020`" and inivites your group member.

1. All the members clone "`Project_EC2020`" into your local directories and navigate to that directory by GitBash (Windows) or the Terminal (Mac). Unless otherwise noted, use the master branch (by default).

2. In your local directory, each of you generates the text file "`intro_yourfirstname.txt`". Write about yourself (that you can share with your friend) inside it.

3. Stage your changes, commit with a message, and try to push it. If you cannot push it (because another collaborator pushed his/her commit earlier), pull first and then push your commit.

4. Once pulling, check that in your local directory, you can see your friend's self-intro! Feel free to make a subfolder like "`Project_EC2020/texts`" (again, push this change if you did so).

Collaboration Exercise (cont'd)

5. Edit a text file generated by another student in your group by writing your image of the friend. Stage your changes, commit with a message, and try to push it. Pull when necessary.

6. Pull. Check that the text file you generated has been revised by your friend.

Now, in each group, nominate one student.

7. All students in each group edit the text file generated by the nominated student. Stage your changes, commit with a comment, and try to push it.

8. Experience conflict. Resolve it together by discussing on which revision to adopt in your group.

9. Track the history. Check revisions made so far.

It works better than having a "conflicted copy" in Dropbox. Still it makes the process complicated. Important to divide tasks between collaborators to avoid conflicts.

Branches & Merging

Aim. Keep track of multiple versions of the same file with different branches. Enable easy experimentations. Easy to compare across experimentations and choose the optimal one by merging.

- Switch to a new branch and list all local branches:

```
$ git checkout -b temp_shunsuke
$ git branch
```

(Drop `-b` when switching to a branch that already exists)

- Push the current branch and set the remote as upstream for sharing it with collaborators:

```
$ git push --set-upstream origin temp_shunsuke
```

- Switch to the master branch and merge the changes made in the new branch back into it:

```
$ git checkout main
$ git merge temp_shunsuke
```

- Delete a local branch & Delete a remote branch:

```
$ git branch -d temp_shunsuke
$ git push --delete origin temp_shunsuke
```

Collaboration Exercise (cont'd)

Now, in order to avoid the messy conflict, use branches and merging.

10. Each student creates the local branch called `temp_firstname`. Push it to the remote repository.

11. Each member of a group edits the text file of the nominated student in each corresponding branch. After editing it, stage your changes, commit with a comment, and push it.

12. Issue pull requests, assigning your collaborator as the reviewer. Compare all members' versions in the online repository. (Discuss in the group and decide which version to adopt.)

13. Comment on texts.
    (Anyone in each group) Merge the adopted branch into the master branch.
    Finally, all members switch back to the master branch and pull.

Pull Request

- Not a function of Git itself, but invented by GitHub first
- A pull request lets you notify your collaborators of changes you've pushed to a branch in a remote repository on GitHub.
- Once a pull request is opened, you can discuss and review the potential changes with your collaborators, before your changes are merged into the master branch.
- Reviewers can provide line-by-line comments on code.
- Just make a pull request at the remote repository on GitHub on the web.

cf.) hub commands: send a pull request to GitHub by command lines

Some Useful Commands

- Check the current directory & Climb up in the folder hierarchy:

```
$ pwd
$ cd ..
```

- Check the commit history (e.g., 7 recent commits):

```
$ git log -7
```

  Go back to the command line after `git log` by pressing `q`.

- Go back to the previous version of commit in the local directory (not in the remote!)

```
$ git reset --hard [commit id]
```

- Check the differences in a specific file with commit messages for the latest commit and the commit two before:

```
$ git show HEAD intro_shunsuke.txt
$ git show HEAD~2 intro_shunsuke.txt
```

- Git Cheat Sheet by GitHub Education

  Git Cheat Sheet by GitHub Training

Git & GitHub: References

For more on practical usages, see:

- Jesús Fernández-Villaverde's note (more on command lines)
- Heitor Pellegrina's note
- Frank Pinter's note (illustrations with GitKraken)
- Benjamin Skrainka's note

For more detail, see:

- Software Carpentry. *Version Control with Git.* (MUST)
- Pro Git Second Edition (Detail documentations)

Assignment 2

Assignment 2: Code Cleanup

- In a journal publication process, authors also submit their code and data.
- However, quality of code does not affect the journal's acceptance decision.
- Code of most papers published in top 5 economics journals are actually messy.
- Let's experience how the techniques we've learnt so far help their improvements!
- Important to be capable of applying the principles we learnt not only to Python but also to other programming languages.

Choose one of the following papers for this assignment:

1. Michalopoulos and Papaioannou (2016 AER) "The Long-Run Effects of the Scramble for Africa" **Use STATA.**

2. Henderson et al. (2018 QJE) "The global distribution of economic activity: Nature, history, and the role of trade" **Use STATA.**

1. Michalopoulos and Papaioannou (2016 AER)

Q. How does border artificiality (w.r.t ethnic partitioning) contribute to conflicts in Africa?

- Download replication files from here.

Task. Clean and shorten the code to replicate Table1, Table2-A and Table2-B.

- Use abstraction nicely.
- Feel free to write a master code in whichever Python or Stata and/or shift to the functional paradigm.

Michalopoulos and Papaioannou (2016 AER)

2. Henderson et al. (2018 QJE)

Q. **What is the role of natural characteristics in determining the worldwide spatial distribution of economic activity?**

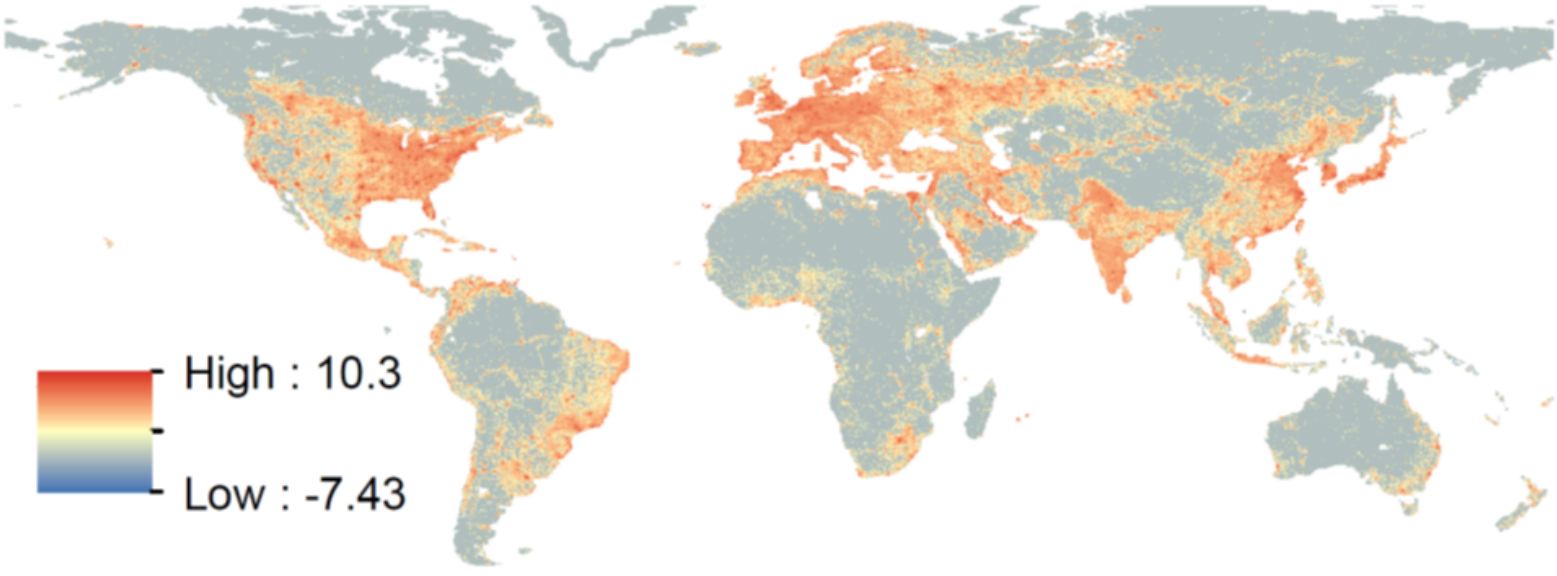
- Download replication files from Adam Storeygard's web.

Task. Separate the code for the part Table I, Table II and Table III into multiple code or program, each of which implements a separate task.

  - Use abstraction nicely.
  - Improve its clarity and maintainability by well-organized (self-)documentations.
  - Feel free to write a master code whichever in Python or Stata and/or shift to the functional paradigm.

Henderson et al. (2018 QJE)

Figure I. Demeaned ln(lights)



FYI: ArcGIS in Economics Research

- Geographic Information System (GIS) is useful for wide fields of economics research.
- Sophisticated use of it can increase the range of questions that you can answer.
- The two papers in this assignment are good examples.
- Historical maps also expand research possibilities.
- Python is also valuable here in that it can automate managing geo-spatial data (ArcPy, GeoPandas).
- Take GIS Institute organized by S4 (Spatial Structures in the Social Sciences) at Brown after your core exam if interested (check the application deadline)!