Tips for an Efficient Workflow

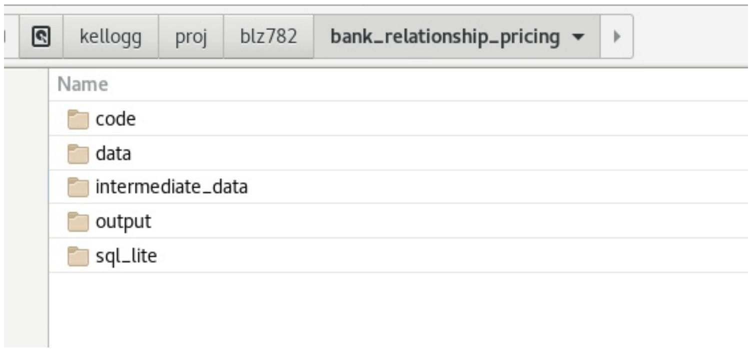
Introduction:

As you're beginning a research project, you should be structuring your research project in an organized way, for your future self's sake, for your coauthor's sake, and for the publication journal's replication RA. Generally, your goal is to create code that takes input files, and generates all output files, with the click of one "run" button.

In this guide, I will first describe the principles that I use in my research workflow, and then will include a second part where I will show you how to incorporate them into using the Kellogg Linux Cluster (for Kellogg affiliates), using my "Relationship Banking" project as an example. I will have all my code available on Github and the one "manual input file" (so you can get all of the data yourself, excluding SDC data, and recreate most of the analysis I have done).

Organizing Folder Structure:

You should set up each project to have a folder structure that can be easily moved (or cloned by a coauthor). For example there should be a "root" directory. Within that, there could be a "code" directory, an "data" folder, and "intermediate_data" directory, and an "output" directory



Tip: Use relative paths when you can! The only path that should be hard coded should be the "root" directory. Everything else should be relative to the root directory.

Organizing Code Structure:

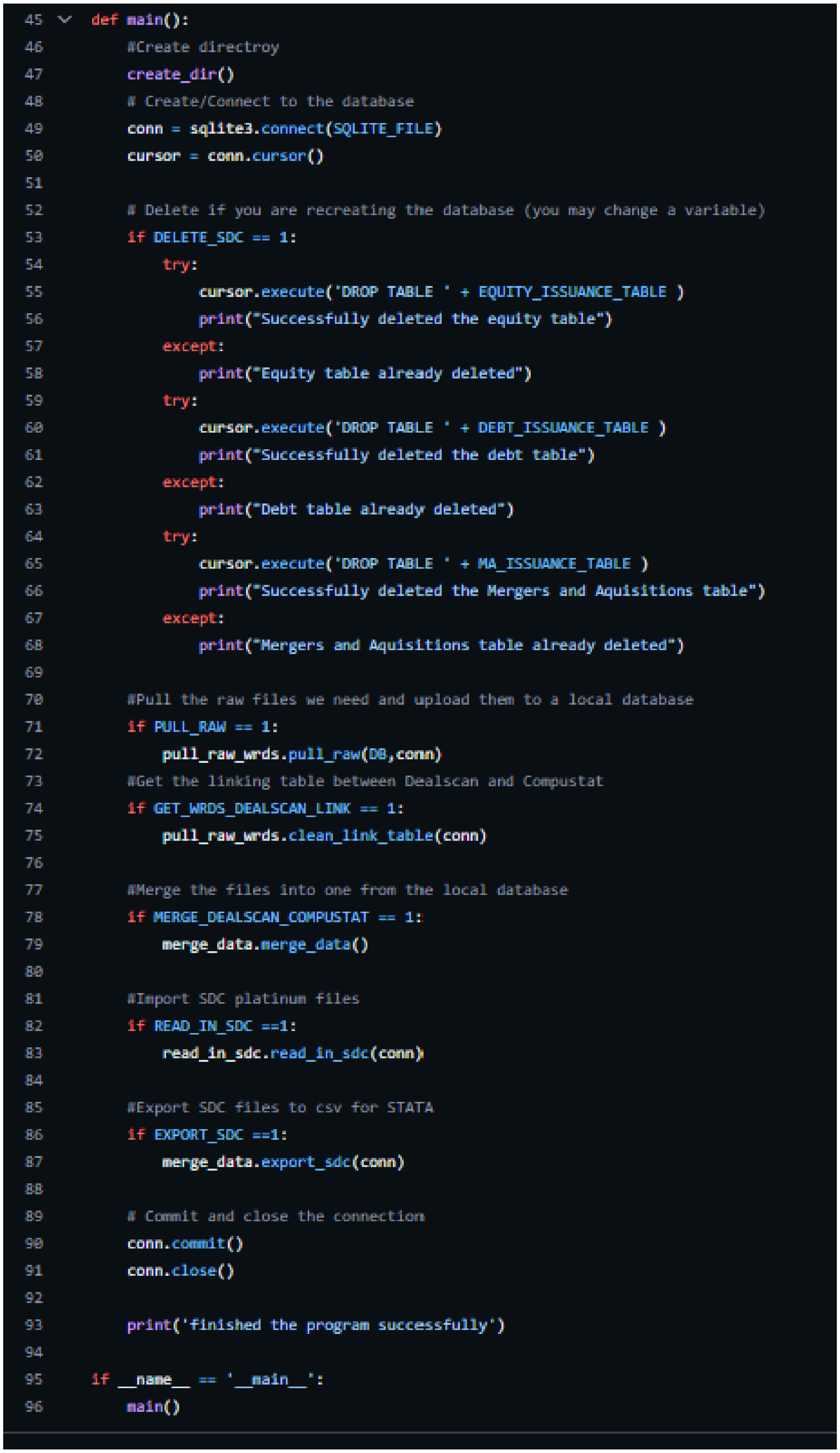
Your code should be organized into several files, that are split up by what they do. No matter what, you should have a "master" file, you may have a "data-gathering" file(s), "data-merging" file(s), data-cleaning file(s), and then analysis file(s). If you have a project that involves structural estimation or solving models, see the appendix:

Master file:

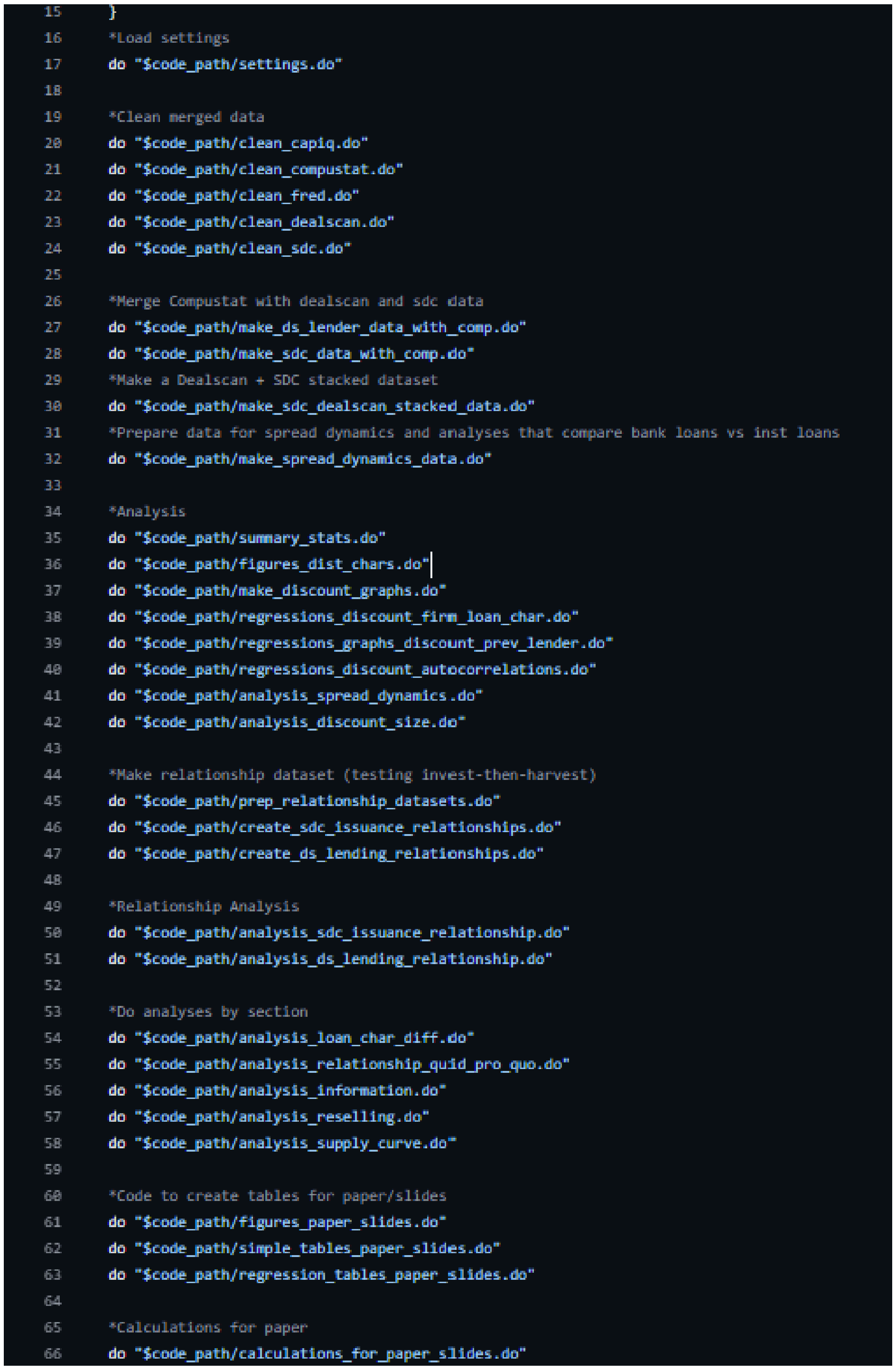
This one master file should be able to run EVERYTHING that you need to run in order to get from data inputs -> outputs. The additional benefit of doing this is that your code is organized. You know exactly what order the code needs to run in. Note that you don’t need to run all code from beginning to end, but you need to be ABLE to do so.

Think about the point of view of someone trying to replicate your file (top Finance journals are requesting this these days). You want them to be able to just click “run” and then it spits out all the results.

Below are the master python file and master Stata file



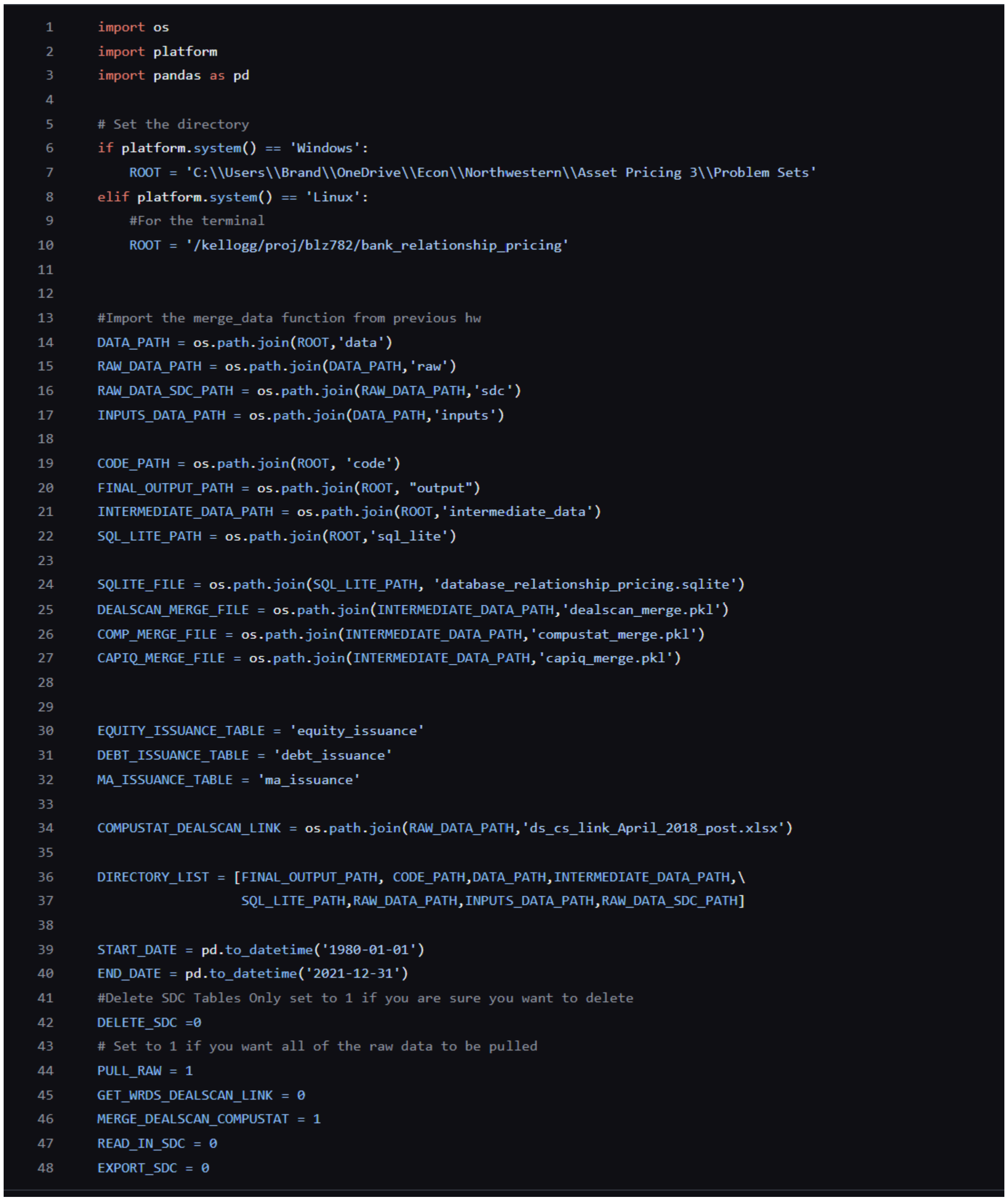
```python
def main():
    #Create directroy
    create_dir()
    # Create/Connect to the database
    conn = sqlite3.connect(SQLITE_FILE)
    cursor = conn.cursor()

    # Delete if you are recreating the database (you may change a variable)
    if DELETE_SDC == 1:
        try:
            cursor.execute('DROP TABLE ' + EQUITY_ISSUANCE_TABLE )
            print("Successfully deleted the equity table")
        except:
            print("Equity table already deleted")
        try:
            cursor.execute('DROP TABLE ' + DEBT_ISSUANCE_TABLE )
            print("Successfully deleted the debt table")
        except:
            print("Debt table already deleted")
        try:
            cursor.execute('DROP TABLE ' + MA_ISSUANCE_TABLE )
            print("Successfully deleted the Mergers and Aquisitions table")
        except:
            print("Mergers and Aquisitions table already deleted")

    #Pull the raw files we need and upload them to a local database
    if PULL_RAW == 1:
        pull_raw_wrds.pull_raw(DB,conn)
    #Get the linking table between Dealscan and Compustat
    if GET_WRDS_DEALSCAN_LINK == 1:
        pull_raw_wrds.clean_link_table(conn)

    #Merge the files into one from the local database
    if MERGE_DEALSCAN_COMPUSTAT == 1:
        merge_data.merge_data()

    #Import SDC platinum files
    if READ_IN_SDC ==1:
        read_in_sdc.read_in_sdc(conn)

    #Export SDC files to csv for STATA
    if EXPORT_SDC ==1:
        merge_data.export_sdc(conn)

    # Commit and close the connection
    conn.commit()
    conn.close()

    print('finished the program successfully')

if __name__ == '__main__':
    main()
```



```
}
*Load settings
do "$code_path/settings.do"

*Clean merged data
do "$code_path/clean_capiq.do"
do "$code_path/clean_compustat.do"
do "$code_path/clean_fred.do"
do "$code_path/clean_dealscan.do"
do "$code_path/clean_sdc.do"

*Merge Compustat with dealscan and sdc data
do "$code_path/make_ds_lender_data_with_comp.do"
do "$code_path/make_sdc_data_with_comp.do"
*Make a Dealscan + SDC stacked dataset
do "$code_path/make_sdc_dealscan_stacked_data.do"
*Prepare data for spread dynamics and analyses that compare bank loans vs inst loans
do "$code_path/make_spread_dynamics_data.do"

*Analysis
do "$code_path/summary_stats.do"
do "$code_path/figures_dist_chars.do"
do "$code_path/make_discount_graphs.do"
do "$code_path/regressions_discount_firm_loan_char.do"
do "$code_path/regressions_graphs_discount_prev_lender.do"
do "$code_path/regressions_discount_autocorrelations.do"
do "$code_path/analysis_spread_dynamics.do"
do "$code_path/analysis_discount_size.do"

*Make relationship dataset (testing invest-then-harvest)
do "$code_path/prep_relationship_datasets.do"
do "$code_path/create_sdc_issuance_relationships.do"
do "$code_path/create_ds_lending_relationships.do"

*Relationship Analysis
do "$code_path/analysis_sdc_issuance_relationship.do"
do "$code_path/analysis_ds_lending_relationship.do"

*Do analyses by section
do "$code_path/analysis_loan_char_diff.do"
do "$code_path/analysis_relationship_quid_pro_quo.do"
do "$code_path/analysis_information.do"
do "$code_path/analysis_reselling.do"
do "$code_path/analysis_supply_curve.do"

*Code to create tables for paper/slides
do "$code_path/figures_paper_slides.do"
do "$code_path/simple_tables_paper_slides.do"
do "$code_path/regression_tables_paper_slides.do"

*Calculations for paper
do "$code_path/calculations_for_paper_slides.do"
```

Settings File:

Have a file where you put down all of your settings. These are like “globals” and are the only things that should be hard coded. I personally like the convention where things that are “outside” of functions are in upper case, so I know that these are global variables that are defined in the settings folder. Generally, my settings folder just sets paths/directories, file locations, and specifies which segments of the code I would like to run.



```python
import os
import platform
import pandas as pd

# Set the directory
if platform.system() == 'Windows':
    ROOT = 'C:\\Users\\Brand\\OneDrive\\Econ\\Northwestern\\Asset Pricing 3\\Problem Sets'
elif platform.system() == 'Linux':
    #For the terminal
    ROOT = '/kellogg/proj/blz782/bank_relationship_pricing'


#Import the merge_data function from previous hw
DATA_PATH = os.path.join(ROOT,'data')
RAW_DATA_PATH = os.path.join(DATA_PATH,'raw')
RAW_DATA_SDC_PATH = os.path.join(RAW_DATA_PATH,'sdc')
INPUTS_DATA_PATH = os.path.join(DATA_PATH,'inputs')

CODE_PATH = os.path.join(ROOT, 'code')
FINAL_OUTPUT_PATH = os.path.join(ROOT, "output")
INTERMEDIATE_DATA_PATH = os.path.join(ROOT,'intermediate_data')
SQL_LITE_PATH = os.path.join(ROOT,'sql_lite')

SQLITE_FILE = os.path.join(SQL_LITE_PATH, 'database_relationship_pricing.sqlite')
DEALSCAN_MERGE_FILE = os.path.join(INTERMEDIATE_DATA_PATH,'dealscan_merge.pkl')
COMP_MERGE_FILE = os.path.join(INTERMEDIATE_DATA_PATH,'compustat_merge.pkl')
CAPIQ_MERGE_FILE = os.path.join(INTERMEDIATE_DATA_PATH,'capiq_merge.pkl')


EQUITY_ISSUANCE_TABLE = 'equity_issuance'
DEBT_ISSUANCE_TABLE = 'debt_issuance'
MA_ISSUANCE_TABLE = 'ma_issuance'

COMPUSTAT_DEALSCAN_LINK = os.path.join(RAW_DATA_PATH,'ds_cs_link_April_2018_post.xlsx')

DIRECTORY_LIST = [FINAL_OUTPUT_PATH, CODE_PATH,DATA_PATH,INTERMEDIATE_DATA_PATH,\
                  SQL_LITE_PATH,RAW_DATA_PATH,INPUTS_DATA_PATH,RAW_DATA_SDC_PATH]

START_DATE = pd.to_datetime('1980-01-01')
END_DATE = pd.to_datetime('2021-12-31')
#Delete SDC Tables Only set to 1 if you are sure you want to delete
DELETE_SDC =0
# Set to 1 if you want all of the raw data to be pulled
PULL_RAW = 1
GET_WRDS_DEALSCAN_LINK = 0
MERGE_DEALSCAN_COMPUSTAT = 1
READ_IN_SDC = 0
EXPORT_SDC = 0
```

Data Gathering File:

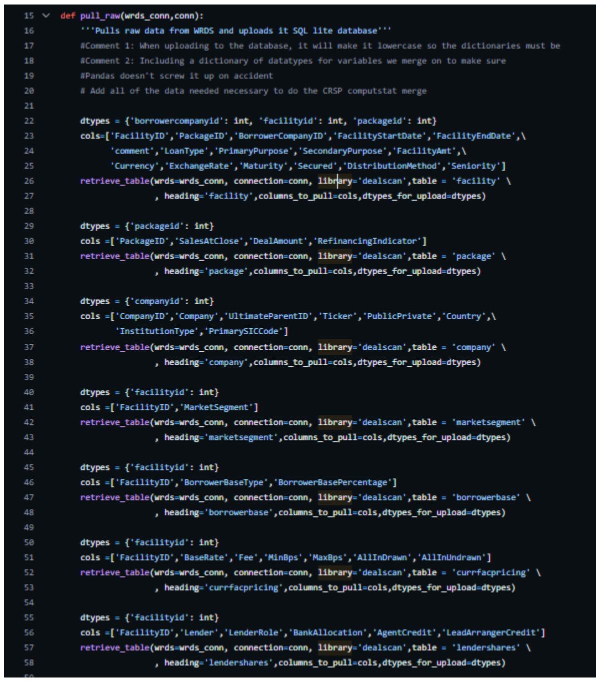
As part of any empirical research project, you will be gathering various datasets and then you will likely be merging them together somehow. I will classify these datasets into two types, those that you are “given/manually collect” and those that your code generates.

Data You Generate:

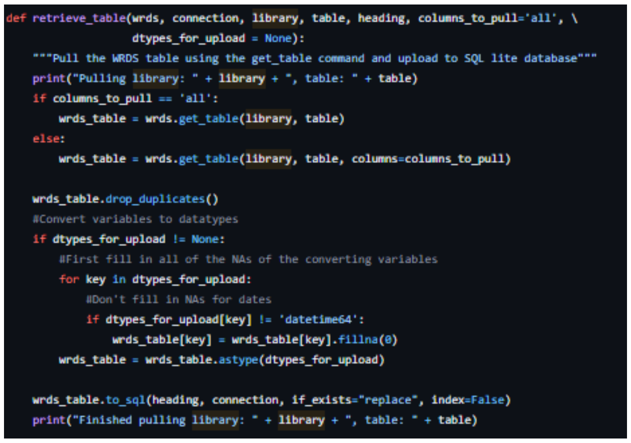
Whenever possible, you should have your code generate your data for you. This will make updating your data/ replication of your code easy. Examples of this type of data include

- Data you web scrape
- Data from FRED
- Downloading data from any WRDS series using their python package
    - If you ever download WRDS data using the user interface by clicking buttons, I will be disappointed with you. Terrible for replicability

Below is a piece of my code that downloads from WRDS and saves them



```python
def pull_raw(wrds_conn,conn):
    '''Pulls raw data from WRDS and uploads it SQL lite database'''
    #Comment 1: When uploading to the database, it will make it lowercase so the dictionaries must be
    #Comment 2: Including a dictionary of datatypes for variables we merge on to make sure
    #Pandas doesn't screw it up on accident
    # Add all of the data needed necessary to do the CRSP computstat merge

    dtypes = {'borrowercompanyid': int, 'facilityid': int, 'packageid': int}
    cols=['FacilityID','PackageID','BorrowerCompanyID','FacilityStartDate','FacilityEndDate',\
          'comment','LoanType','PrimaryPurpose','SecondaryPurpose','FacilityAmt',\
          'Currency','ExchangeRate','Maturity','Secured','DistributionMethod','Seniority']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'facility' \
                   , heading='facility',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'packageid': int}
    cols =['PackageID','SalesAtClose','DealAmount','RefinancingIndicator']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'package' \
                   , heading='package',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'companyid': int}
    cols =['CompanyID','Company','UltimateParentID','Ticker','PublicPrivate','Country',\
           'InstitutionType','PrimarySICCode']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'company' \
                   , heading='company',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'facilityid': int}
    cols =['FacilityID','MarketSegment']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'marketsegment' \
                   , heading='marketsegment',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'facilityid': int}
    cols =['FacilityID','BorrowerBaseType','BorrowerBasePercentage']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'borrowerbase' \
                   , heading='borrowerbase',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'facilityid': int}
    cols =['FacilityID','BaseRate','Fee','MinBps','MaxBps','AllInDrawn','AllInUndrawn']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'currfacpricing' \
                   , heading='currfacpricing',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'facilityid': int}
    cols =['FacilityID','Lender','LenderRole','BankAllocation','AgentCredit','LeadArrangerCredit']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'lendershares' \
                   , heading='lendershares',columns_to_pull=cols,dtypes_for_upload=dtypes)
```



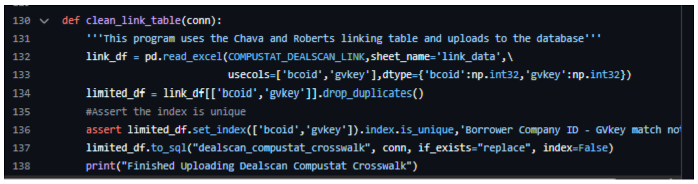
```python
def retrieve_table(wrds, connection, library, table, heading, columns_to_pull='all', \
                   dtypes_for_upload = None):
    """Pull the WRDS table using the get_table command and upload to SQL lite database"""
    print("Pulling library: " + library + ", table: " + table)
    if columns_to_pull == 'all':
        wrds_table = wrds.get_table(library, table)
    else:
        wrds_table = wrds.get_table(library, table, columns=columns_to_pull)

    wrds_table.drop_duplicates()
    #Convert variables to datatypes
    if dtypes_for_upload != None:
        #First fill in all of the NAs of the converting variables
        for key in dtypes_for_upload:
            #Don't fill in NAs for dates
            if dtypes_for_upload[key] != 'datetime64':
                wrds_table[key] = wrds_table[key].fillna(0)
        wrds_table = wrds_table.astype(dtypes_for_upload)

    wrds_table.to_sql(heading, connection, if_exists="replace", index=False)
    print("Finished pulling library: " + library + ", table: " + table)
```

Data You Are Given/Manually Collect:

This type of data are ones that you cannot generate from code. Any dataset that falls in this category is often a huge pain to update. Examples of this type of data include

- Commercial databases without nice python/R interfaces (SDC Platinum, S&P Capital IQ)
- Bloomberg data pulls from a Bloomberg terminal
- Data you are given by a data partner

Here is the version of the Chava and Roberts linking table I used in this project and code used to read it in

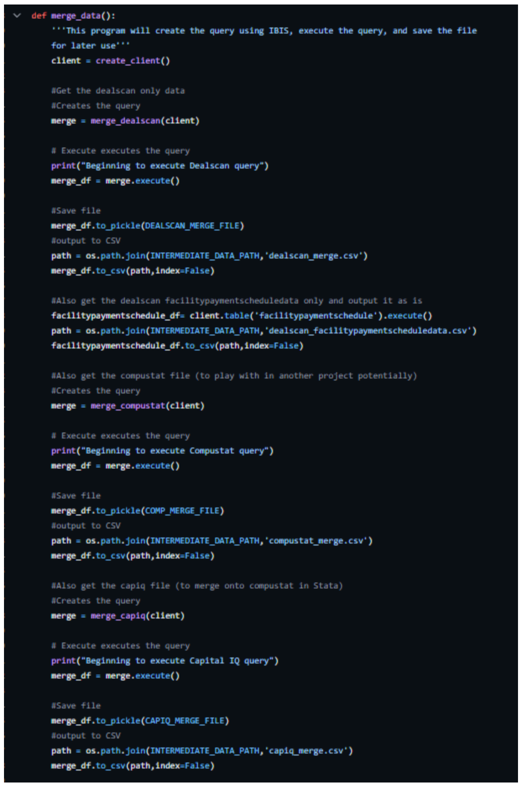


```python
def clean_link_table(conn):
    '''This program uses the Chava and Roberts linking table and uploads to the database'''
    link_df = pd.read_excel(COMPUSTAT_DEALSCAN_LINK,sheet_name='link_data',\
                            usecols=['bcoid','gvkey'],dtype={'bcoid':np.int32,'gvkey':np.int32})
    limited_df = link_df[['bcoid','gvkey']].drop_duplicates()
    #Assert the index is unique
    assert limited_df.set_index(['bcoid','gvkey']).index.is_unique,'Borrower Company ID - GVkey match no
    limited_df.to_sql("dealscan_compustat_crosswalk", conn, if_exists="replace", index=False)
    print("Finished Uploading Dealscan Compustat Crosswalk")
```

Data merging code:

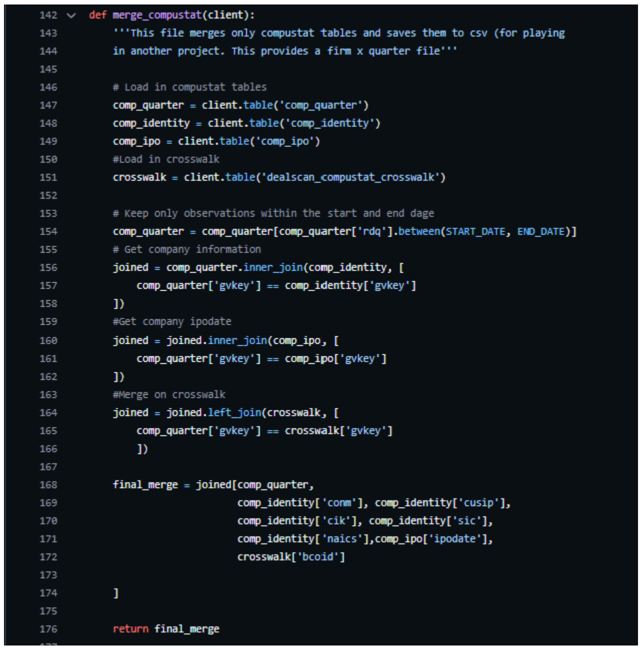
You will likely have various input datasets and now you will need to merge them together. Depending on how many datasets and how big they are, you have several options. When you have few datasets and they are small, you can use any languages merge functionality

Below is the example in Python of joining the Compustat (and Dealscan) files using an incredible package called IBIS. For a complete description of this package and how to do more complex merges in a clean way, and relational databases in general, see appendix.



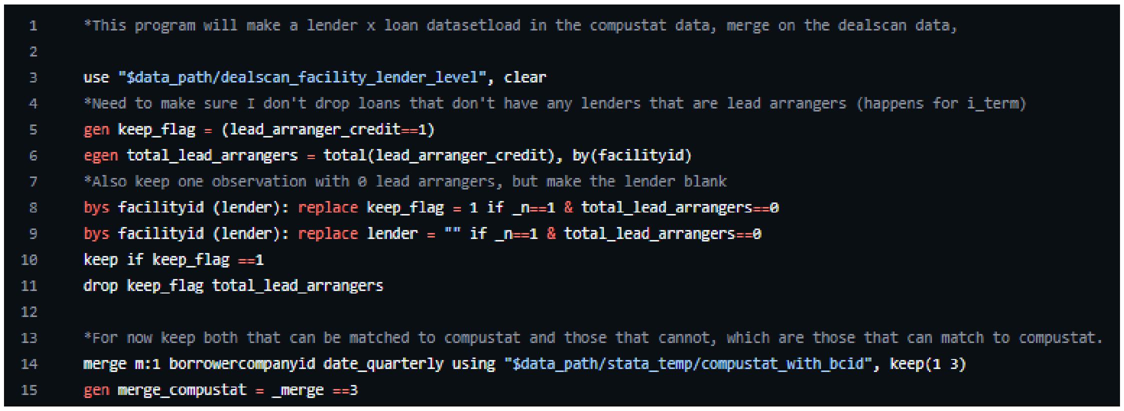
```python
def merge_data():
    '''This program will create the query using IBIS, execute the query, and save the file
    for later use'''
    client = create_client()

    #Get the dealscan only data
    #Creates the query
    merge = merge_dealscan(client)

    # Execute executes the query
    print("Beginning to execute Dealscan query")
    merge_df = merge.execute()

    #Save file
    merge_df.to_pickle(DEALSCAN_MERGE_FILE)
    #output to CSV
    path = os.path.join(INTERMEDIATE_DATA_PATH,'dealscan_merge.csv')
    merge_df.to_csv(path,index=False)

    #Also get the dealscan facilitypaymentscheduledata only and output it as is
    facilitypaymentschedule_df= client.table('facilitypaymentschedule').execute()
    path = os.path.join(INTERMEDIATE_DATA_PATH,'dealscan_facilitypaymentscheduledata.csv')
    facilitypaymentschedule_df.to_csv(path,index=False)

    #Also get the compustat file (to play with in another project potentially)
    #Creates the query
    merge = merge_compustat(client)

    # Execute executes the query
    print("Beginning to execute Compustat query")
    merge_df = merge.execute()

    #Save file
    merge_df.to_pickle(COMP_MERGE_FILE)
    #output to CSV
    path = os.path.join(INTERMEDIATE_DATA_PATH,'compustat_merge.csv')
    merge_df.to_csv(path,index=False)

    #Also get the capiq file (to merge onto compustat in Stata)
    #Creates the query
    merge = merge_capiq(client)

    # Execute executes the query
    print("Beginning to execute Capital IQ query")
    merge_df = merge.execute()

    #Save file
    merge_df.to_pickle(CAPIQ_MERGE_FILE)
    #output to CSV
    path = os.path.join(INTERMEDIATE_DATA_PATH,'capiq_merge.csv')
    merge_df.to_csv(path,index=False)
```

Specifically highlighting the Compustat merge in IBIS (see appendix for a more thorough description):



```python
def merge_compustat(client):
    '''This file merges only compustat tables and saves them to csv (for playing
    in another project. This provides a firm x quarter file'''

    # Load in compustat tables
    comp_quarter = client.table('comp_quarter')
    comp_identity = client.table('comp_identity')
    comp_ipo = client.table('comp_ipo')
    #Load in crosswalk
    crosswalk = client.table('dealscan_compustat_crosswalk')

    # Keep only observations within the start and end dage
    comp_quarter = comp_quarter[comp_quarter['rdq'].between(START_DATE, END_DATE)]
    # Get company information
    joined = comp_quarter.inner_join(comp_identity, [
        comp_quarter['gvkey'] == comp_identity['gvkey']
    ])
    #Get company ipodate
    joined = joined.inner_join(comp_ipo, [
        comp_quarter['gvkey'] == comp_ipo['gvkey']
    ])
    #Merge on crosswalk
    joined = joined.left_join(crosswalk, [
        comp_quarter['gvkey'] == crosswalk['gvkey']
        ])

    final_merge = joined[comp_quarter,
                         comp_identity['conm'], comp_identity['cusip'],
                         comp_identity['cik'], comp_identity['sic'],
                         comp_identity['naics'],comp_ipo['ipodate'],
                         crosswalk['bcoid']

    ]

    return final_merge
```

For an example of simple merge in Stata, see below.



```stata
*This program will make a lender x loan datasetload in the compustat data, merge on the dealscan data,

use "$data_path/dealscan_facility_lender_level", clear
*Need to make sure I don't drop loans that don't have any lenders that are lead arrangers (happens for i_term)
gen keep_flag = (lead_arranger_credit==1)
egen total_lead_arrangers = total(lead_arranger_credit), by(facilityid)
*Also keep one observation with 0 lead arrangers, but make the lender blank
bys facilityid (lender): replace keep_flag = 1 if _n==1 & total_lead_arrangers==0
bys facilityid (lender): replace lender = "" if _n==1 & total_lead_arrangers==0
keep if keep_flag ==1
drop keep_flag total_lead_arrangers

*For now keep both that can be matched to compustat and those that cannot, which are those that can match to compustat.
merge m:1 borrowercompanyid date_quarterly using "$data_path/stata_temp/compustat_with_bcid", keep(1 3)
gen merge_compustat = _merge ==3
```

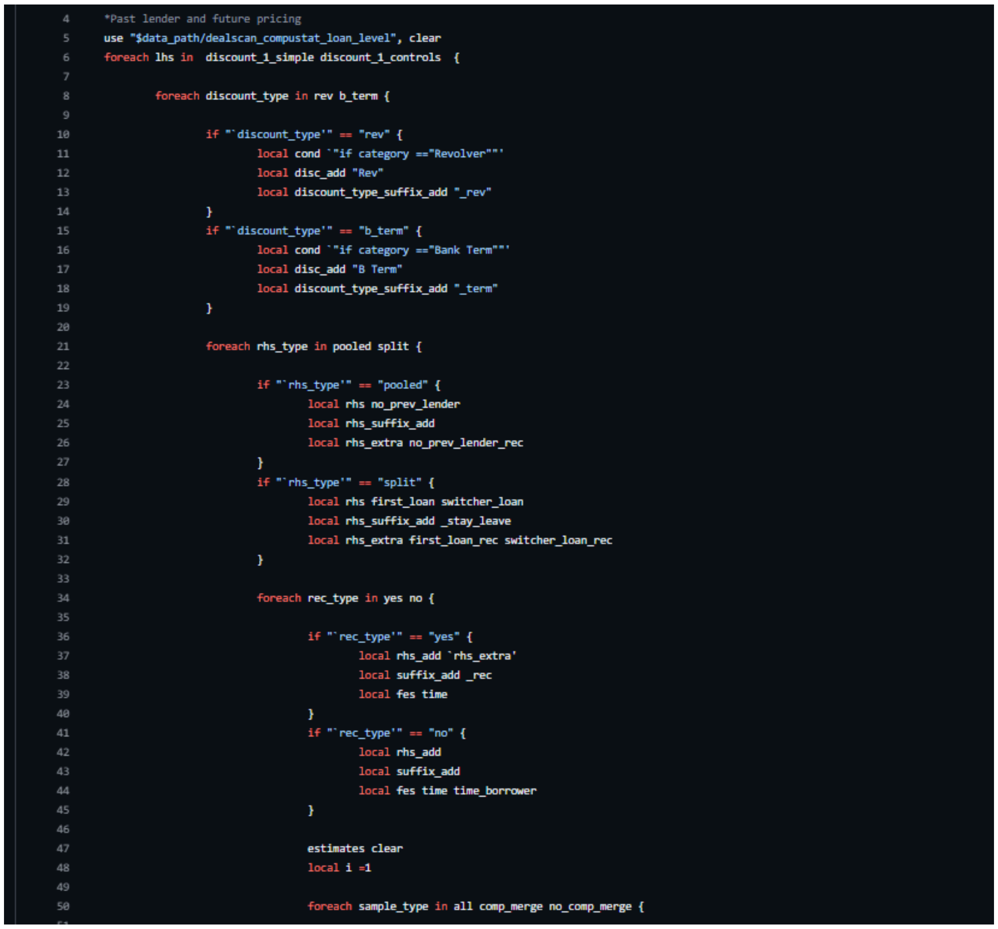
Data cleaning code:

Data cleaning is extremely important and is basically preparing your data to be used in analyses. Here you should be making any transformations to your data (winsorizing), creating any variables necessary for regressions, etc. It is not necessary that this is done after data is merged, it can also be done before data is merged.

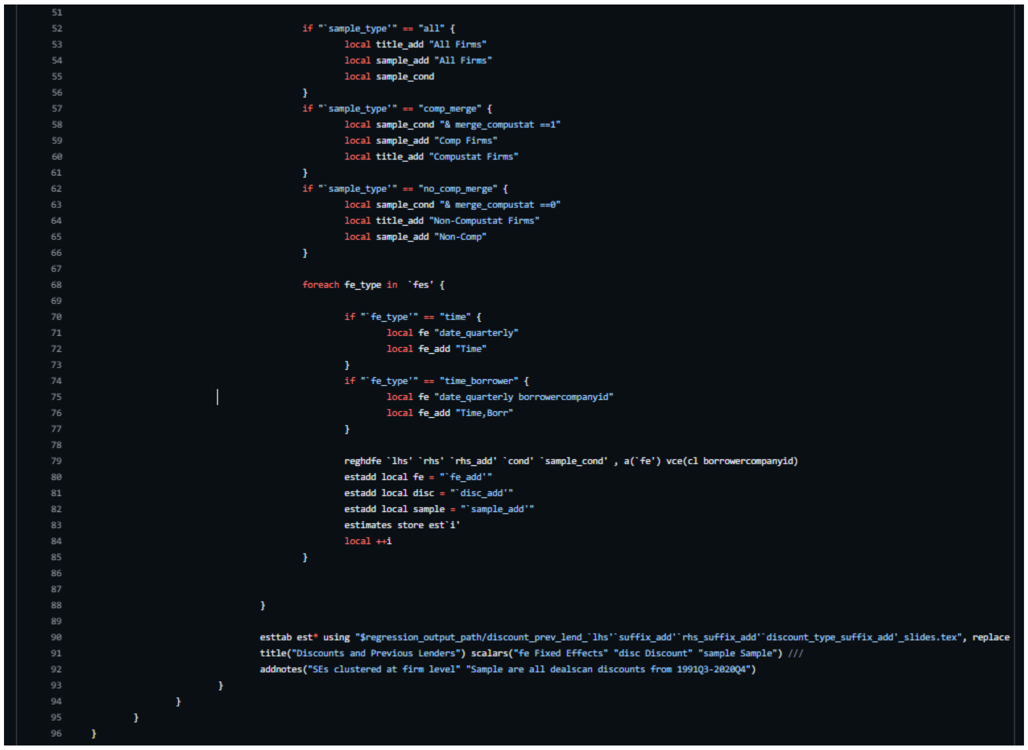
Analysis code:

Your analysis code should ONLY do data analysis. You should NOT be generating new variables but only running analyses/creating figures.

Below is my code that generates the regression tables relating discount size and relationship status



```stata
*Past lender and future pricing
use "$data_path/dealscan_compustat_loan_level", clear
foreach lhs in  discount_1_simple discount_1_controls  {

    foreach discount_type in rev b_term {

        if "`discount_type'" == "rev" {
            local cond `"if category =="Revolver""'
            local disc_add "Rev"
            local discount_type_suffix_add "_rev"
        }
        if "`discount_type'" == "b_term" {
            local cond `"if category =="Bank Term""'
            local disc_add "B Term"
            local discount_type_suffix_add "_term"
        }

        foreach rhs_type in pooled split {

            if "`rhs_type'" == "pooled" {
                local rhs no_prev_lender
                local rhs_suffix_add
                local rhs_extra no_prev_lender_rec
            }
            if "`rhs_type'" == "split" {
                local rhs first_loan switcher_loan
                local rhs_suffix_add _stay_leave
                local rhs_extra first_loan_rec switcher_loan_rec
            }

            foreach rec_type in yes no {

                if "`rec_type'" == "yes" {
                    local rhs_add `rhs_extra'
                    local suffix_add _rec
                    local fes time
                }
                if "`rec_type'" == "no" {
                    local rhs_add
                    local suffix_add
                    local fes time time_borrower
                }

                estimates clear
                local i =1

                foreach sample_type in all comp_merge no_comp_merge {
```



```stata

                if "`sample_type'" == "all" {
                    local title_add "All Firms"
                    local sample_add "All Firms"
                    local sample_cond
                }
                if "`sample_type'" == "comp_merge" {
                    local sample_cond "& merge_compustat ==1"
                    local sample_add "Comp Firms"
                    local title_add "Compustat Firms"
                }
                if "`sample_type'" == "no_comp_merge" {
                    local sample_cond "& merge_compustat ==0"
                    local title_add "Non-Compustat Firms"
                    local sample_add "Non-Comp"
                }

                foreach fe_type in  `fes' {

                    if "`fe_type'" == "time" {
                        local fe "date_quarterly"
                        local fe_add "Time"
                    }
                    if "`fe_type'" == "time_borrower" {
                        local fe "date_quarterly borrowercompanyid"
                        local fe_add "Time,Borr"
                    }

                    reghdfe `lhs' `rhs' `rhs_add' `cond' `sample_cond' , a(`fe') vce(cl borrowercompanyid)
                    estadd local fe = "`fe_add'"
                    estadd local disc = "`disc_add'"
                    estadd local sample = "`sample_add'"
                    estimates store est`i'
                    local ++i
                }


            }

            esttab est* using "$regression_output_path/discount_prev_lend_`lhs'`suffix_add'`rhs_suffix_add'`discount_type_suffix_add'_slides.tex", replace
            title("Discounts and Previous Lenders") scalars("fe Fixed Effects" "disc Discount" "sample Sample") ///
            addnotes("SEs clustered at firm level" "Sample are all dealscan discounts from 1991Q3-2020Q4")
        }
    }
  }
}
```

Exporting to Overleaf/Latex:

(users of RMarkdown, please ignore. RMarkdown is great, I just don't use R).

Generally you want to have as little manual touching of latex files as possible. What you want to do is to have your favorite programming language (Stata for me here) generate figures and .tex file output that you upload to overleaf, and then have overleaf read in those raw files. This allows you to create 100 page pdfs with 400 figures and tables (kidding, not kidding). If you do this, it will allow you to update the figures/tables relatively easily (just run the code and reupload the output).

See the earlier screenshot to see how everything is exported to a .tex file.

Using an IDE:

If you are doing coding in something like Python or R, please use an IDE (integrated developer environment). This will make your life so much easier. See Raul's guide for more tips.

I use PyCharm. Pycharm has a lot of great functionality that help with debugging, in particular the debugger or the "calculation" button. Ultimately do your own research, but please use one. Your future self will thank you!

APPENDIX

Guide to Using Relational Databases:

If you have large datasets or have many datasets to work with, you should use relational databases for organizational purposes and merging. Relational databases will help ensure that your joins/merges are occurring properly, and also will allow you to look at all of the input datasets individually before merging easily. If you have large datasets, it will also allow you to query subsets of your data efficiently.

For anyone doing empirical work where a ready-to-use dataset is not given to you, should consider using relational databases for data management. A relationship database is a set of tables (ideally with some set of identifiers) that you can pull from and merge together using SQL queries. You may have used SQL queries to get data from relational databases, but you SHOULD make your own in your research. The end goal is to make a relatively clean data file with all the relevant information you may need that you can then use in the statistical programming language of choice (note: relational databases are NOT good for fuzzy merges)

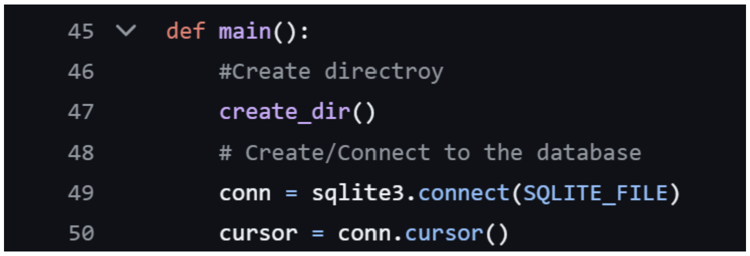
Why should you use relational databases:

- You are creating your own dataset from many individual data files (for example, web scraping, reading in many excel files as inputs)
- Your data is very big (~ over 5gbs) and you want to only get a subset of it for your analysis
- You want to cleanly and transparently join/merge across many different data files
    - This is particularly relevant for many data files from WRDS
        - WRDS uses relational databases

Okay so how do you do it? I will walk you through it with an example from my Python code.

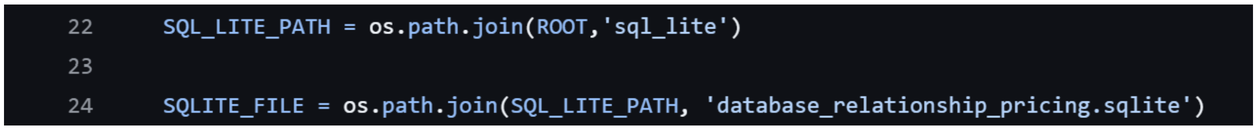
Imagine you want to download Compustat data from WRDS that is spread across many different tables. In addition, you want to merge on a linking table that will allow you eventually merge to Dealscan (from Chava and Roberts). What we will do is to download the data from each individual table, put that data into my own relational database, then use joins to merge all of the relevant information into one data file. (I will omit which python modules you will need to import, but that is easily figured out in a Google search)

First, you will need to create the database file



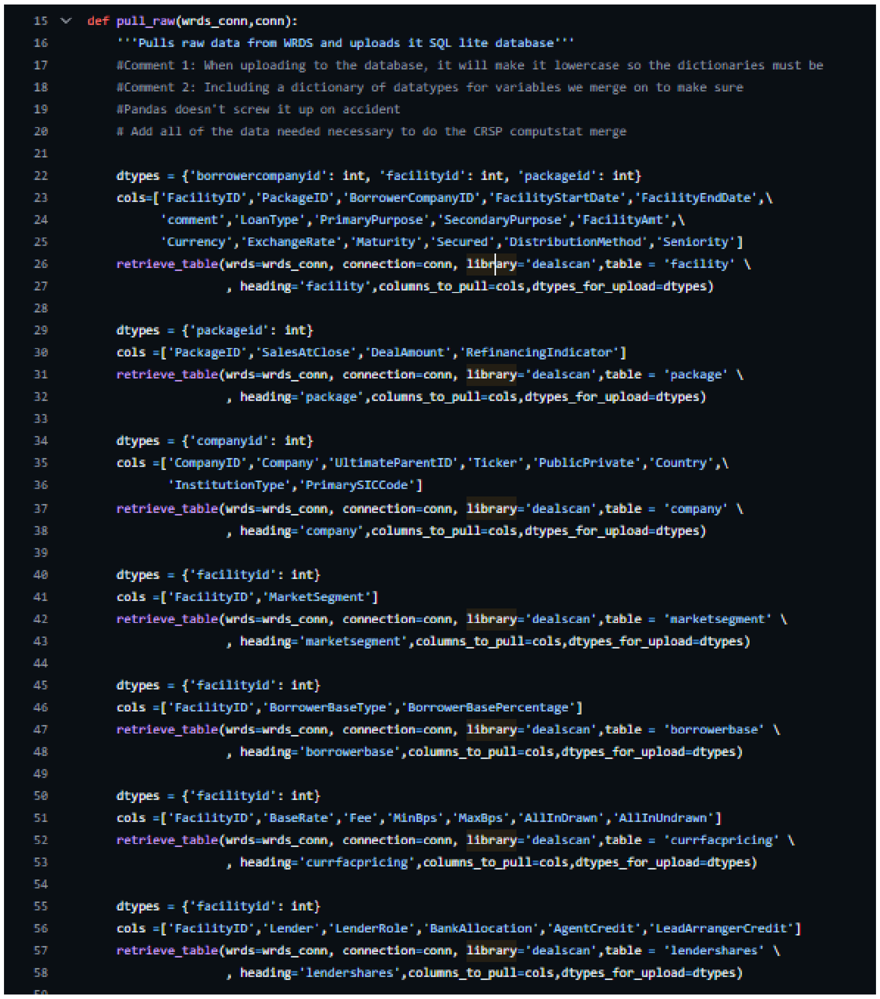
```python
def main():
    #Create directroy
    create_dir()
    # Create/Connect to the database
    conn = sqlite3.connect(SQLITE_FILE)
    cursor = conn.cursor()
```

Where I have defined the name of the SQLITE_FILE previously in “settings.py”.



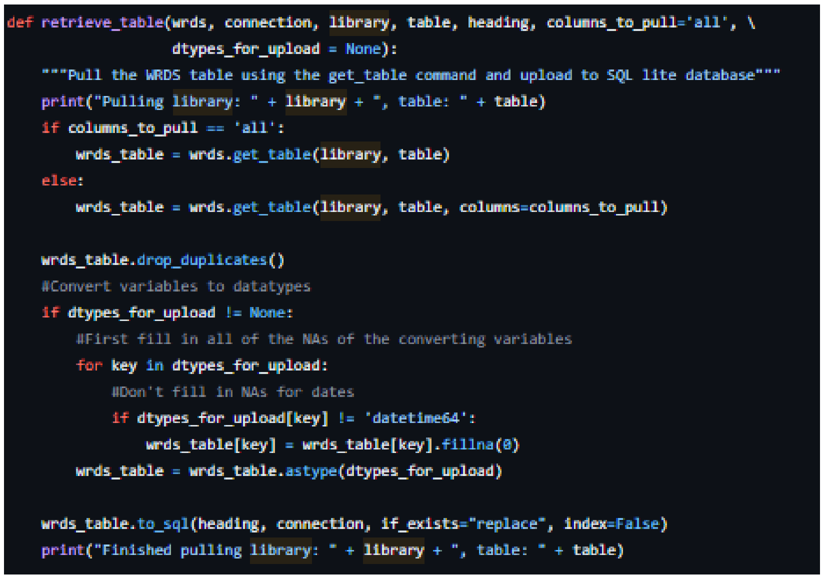
```python
SQL_LITE_PATH = os.path.join(ROOT,'sql_lite')

SQLITE_FILE = os.path.join(SQL_LITE_PATH, 'database_relationship_pricing.sqlite')
```

Now I use WRDS Python module to download three separate tables from WRDS (along with other functions I have defined)



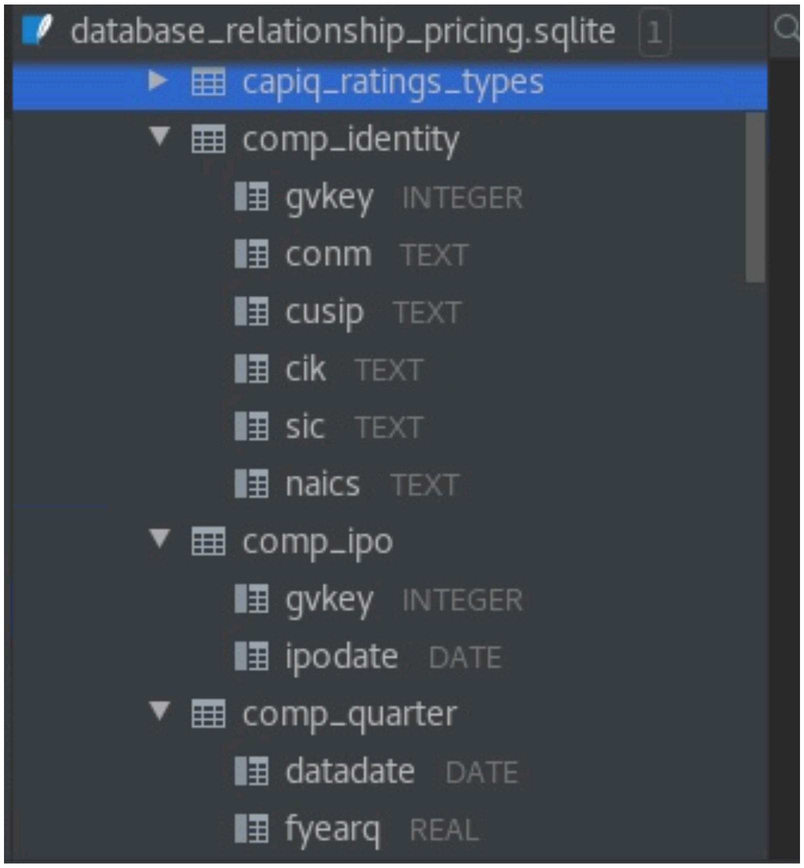
```python
def pull_raw(wrds_conn,conn):
    '''Pulls raw data from WRDS and uploads it SQL lite database'''
    #Comment 1: When uploading to the database, it will make it lowercase so the dictionaries must be
    #Comment 2: Including a dictionary of datatypes for variables we merge on to make sure
    #Pandas doesn't screw it up on accident
    # Add all of the data needed necessary to do the CRSP computstat merge

    dtypes = {'borrowercompanyid': int, 'facilityid': int, 'packageid': int}
    cols=['FacilityID','PackageID','BorrowerCompanyID','FacilityStartDate','FacilityEndDate',\
          'comment','LoanType','PrimaryPurpose','SecondaryPurpose','FacilityAmt',\
          'Currency','ExchangeRate','Maturity','Secured','DistributionMethod','Seniority']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'facility' \
                   , heading='facility',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'packageid': int}
    cols =['PackageID','SalesAtClose','DealAmount','RefinancingIndicator']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'package' \
                   , heading='package',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'companyid': int}
    cols =['CompanyID','Company','UltimateParentID','Ticker','PublicPrivate','Country',\
           'InstitutionType','PrimarySICCode']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'company' \
                   , heading='company',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'facilityid': int}
    cols =['FacilityID','MarketSegment']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'marketsegment' \
                   , heading='marketsegment',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'facilityid': int}
    cols =['FacilityID','BorrowerBaseType','BorrowerBasePercentage']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'borrowerbase' \
                   , heading='borrowerbase',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'facilityid': int}
    cols =['FacilityID','BaseRate','Fee','MinBps','MaxBps','AllInDrawn','AllInUndrawn']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'currfacpricing' \
                   , heading='currfacpricing',columns_to_pull=cols,dtypes_for_upload=dtypes)

    dtypes = {'facilityid': int}
    cols =['FacilityID','Lender','LenderRole','BankAllocation','AgentCredit','LeadArrangerCredit']
    retrieve_table(wrds=wrds_conn, connection=conn, library='dealscan',table = 'lendershares' \
                   , heading='lendershares',columns_to_pull=cols,dtypes_for_upload=dtypes)
```

Where retrieve table is defined below. Each of these “retrieve_table” functions retrieves a table as a dataframe named “wrds_table” from WRDS and uploads them to the database I created using the “to_sql” command.



```python
def retrieve_table(wrds, connection, library, table, heading, columns_to_pull='all', \
                   dtypes_for_upload = None):
    """Pull the WRDS table using the get_table command and upload to SQL lite database"""
    print("Pulling library: " + library + ", table: " + table)
    if columns_to_pull == 'all':
        wrds_table = wrds.get_table(library, table)
    else:
        wrds_table = wrds.get_table(library, table, columns=columns_to_pull)

    wrds_table.drop_duplicates()
    #Convert variables to datatypes
    if dtypes_for_upload != None:
        #First fill in all of the NAs of the converting variables
        for key in dtypes_for_upload:
            #Don't fill in NAs for dates
            if dtypes_for_upload[key] != 'datetime64':
                wrds_table[key] = wrds_table[key].fillna(0)
        wrds_table = wrds_table.astype(dtypes_for_upload)

    wrds_table.to_sql(heading, connection, if_exists="replace", index=False)
    print("Finished pulling library: " + library + ", table: " + table)
```

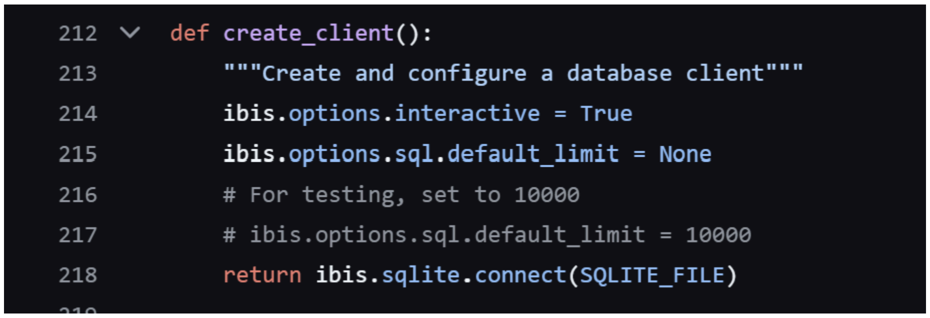
To see how they look in the database, you can view them in a program like DataGrip, which is free for students. These all have an identifier “gvkey” that I will be joining on later.



```
database_relationship_pricing.sqlite  1
  ▶ capiq_ratings_types
  ▼ comp_identity
      gvkey   INTEGER
      conm    TEXT
      cusip   TEXT
      cik     TEXT
      sic     TEXT
      naics   TEXT
  ▼ comp_ipo
      gvkey   INTEGER
      ipodate DATE
  ▼ comp_quarter
      datadate DATE
      fyearq   REAL
```

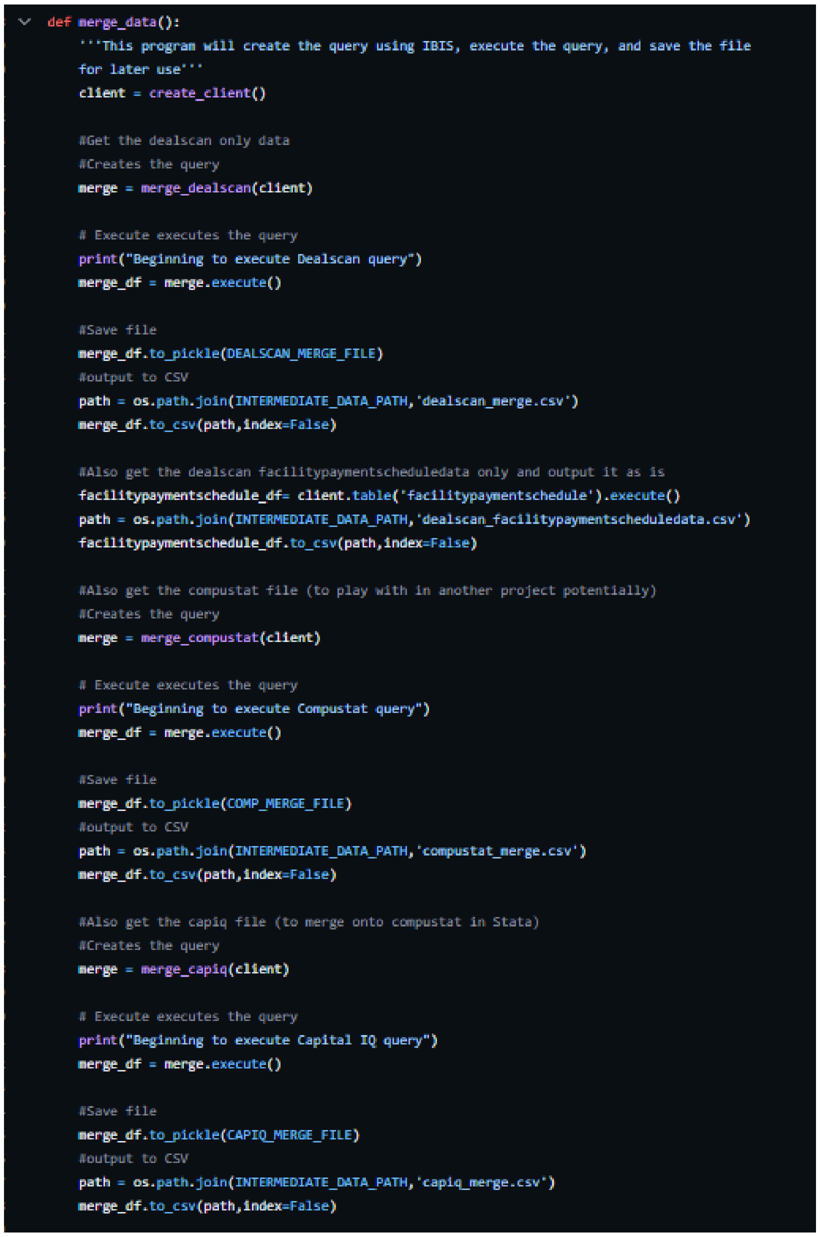
Now let’s merge these together. You can merge these with SQL query syntax, but I find this unpleasant, so I like to use a package in Python called “IBIS”. IBIS is a query building package.

First I create a “client”, or a connection to the SQLITE database.



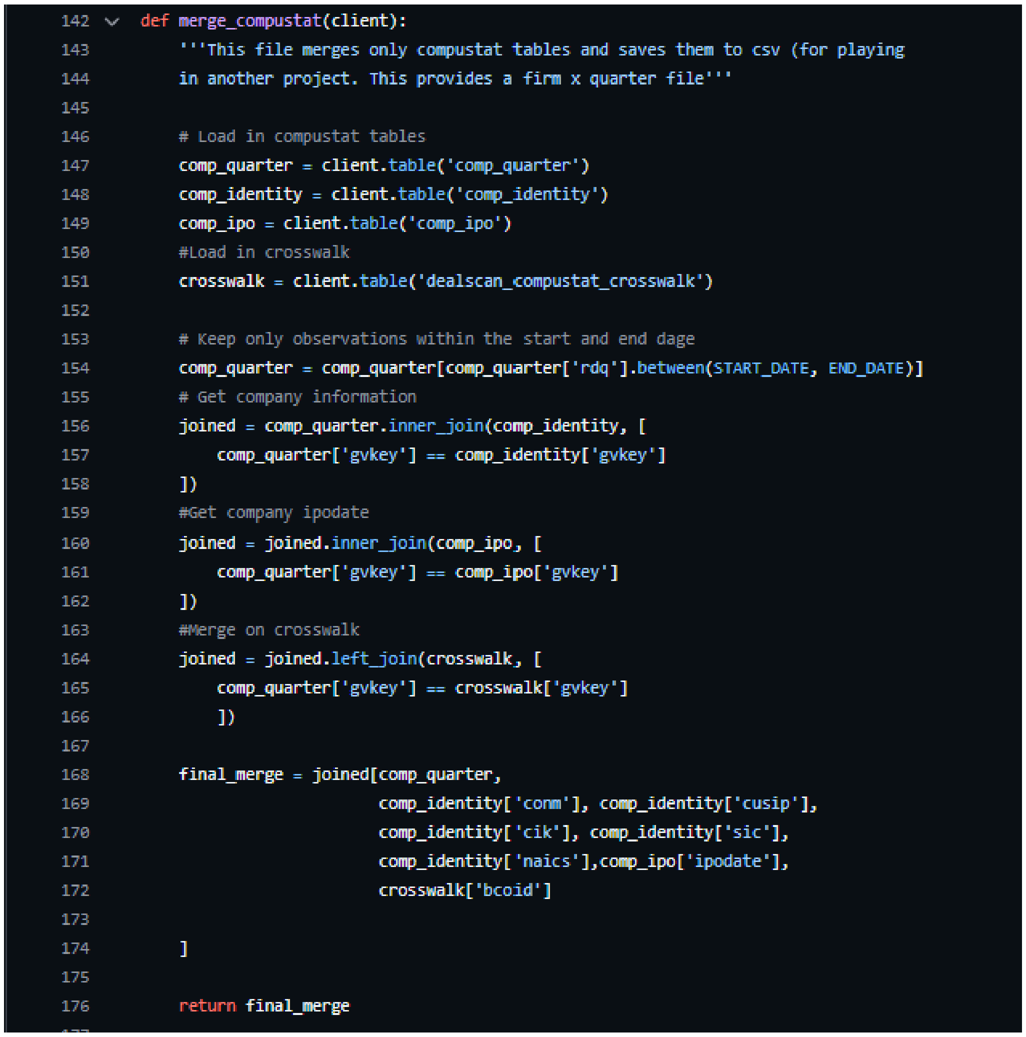
```python
def create_client():
    """Create and configure a database client"""
    ibis.options.interactive = True
    ibis.options.sql.default_limit = None
    # For testing, set to 10000
    # ibis.options.sql.default_limit = 10000
    return ibis.sqlite.connect(SQLITE_FILE)
```

Then I tell this connection to build a query according to my function “merge_compustat”, execute the query to get the merged datafile, and then output it as a .csv, that I will later load into STATA. I actually do three separate merges to make three separate output files (Dealscan, Compustat, and CapitalIQ), but I will only show Compustat because it is sufficient to show the concept.



```python
def merge_data():
    '''This program will create the query using IBIS, execute the query, and save the file
    for later use'''
    client = create_client()

    #Get the dealscan only data
    #Creates the query
    merge = merge_dealscan(client)

    # Execute executes the query
    print("Beginning to execute Dealscan query")
    merge_df = merge.execute()

    #Save file
    merge_df.to_pickle(DEALSCAN_MERGE_FILE)
    #output to CSV
    path = os.path.join(INTERMEDIATE_DATA_PATH,'dealscan_merge.csv')
    merge_df.to_csv(path,index=False)

    #Also get the dealscan facilitypaymentscheduledata only and output it as is
    facilitypaymentschedule_df= client.table('facilitypaymentschedule').execute()
    path = os.path.join(INTERMEDIATE_DATA_PATH,'dealscan_facilitypaymentscheduledata.csv')
    facilitypaymentschedule_df.to_csv(path,index=False)

    #Also get the compustat file (to play with in another project potentially)
    #Creates the query
    merge = merge_compustat(client)

    # Execute executes the query
    print("Beginning to execute Compustat query")
    merge_df = merge.execute()

    #Save file
    merge_df.to_pickle(COMP_MERGE_FILE)
    #output to CSV
    path = os.path.join(INTERMEDIATE_DATA_PATH,'compustat_merge.csv')
    merge_df.to_csv(path,index=False)

    #Also get the capiq file (to merge onto compustat in Stata)
    #Creates the query
    merge = merge_capiq(client)

    # Execute executes the query
    print("Beginning to execute Capital IQ query")
    merge_df = merge.execute()

    #Save file
    merge_df.to_pickle(CAPIQ_MERGE_FILE)
    #output to CSV
    path = os.path.join(INTERMEDIATE_DATA_PATH,'capiq_merge.csv')
    merge_df.to_csv(path,index=False)
```

IBIS is a tool to build a SQL query by using nice Pandas type syntax. To see an example, look at the “merge_compustat” function. This is quite flexible, and it allows you to do many powerful things.



```python
def merge_compustat(client):
    '''This file merges only compustat tables and saves them to csv (for playing
    in another project. This provides a firm x quarter file'''

    # Load in compustat tables
    comp_quarter = client.table('comp_quarter')
    comp_identity = client.table('comp_identity')
    comp_ipo = client.table('comp_ipo')
    #Load in crosswalk
    crosswalk = client.table('dealscan_compustat_crosswalk')

    # Keep only observations within the start and end dage
    comp_quarter = comp_quarter[comp_quarter['rdq'].between(START_DATE, END_DATE)]
    # Get company information
    joined = comp_quarter.inner_join(comp_identity, [
        comp_quarter['gvkey'] == comp_identity['gvkey']
    ])
    #Get company ipodate
    joined = joined.inner_join(comp_ipo, [
        comp_quarter['gvkey'] == comp_ipo['gvkey']
    ])
    #Merge on crosswalk
    joined = joined.left_join(crosswalk, [
        comp_quarter['gvkey'] == crosswalk['gvkey']
        ])

    final_merge = joined[comp_quarter,
                         comp_identity['conm'], comp_identity['cusip'],
                         comp_identity['cik'], comp_identity['sic'],
                         comp_identity['naics'],comp_ipo['ipodate'],
                         crosswalk['bcoid']

    ]

    return final_merge
```

This “loads” in the three tables from the database we uploaded earlier (and the crosswalk). I start the merged file by using “comp_quarter” and keeping only observations within the data range I care about. Then I do an inner join on the “comp_identity” table on the identifier, which is gvkey. I also do the same with the “ipodate” information. And then do a left join to get the crosswalk information (look up SQL documentation to understand joins).

Finally, I need to tell it which variables I want at the end of the day, which are all of the variables from “comp_quarter”, a few variables from “comp_identity”, the “ipodate” from comp_ipo, and a different identifier (“bcoid”) from the crosswalk,which is used to merge with Dealscan later on in my research project (in Stata).

In this particular case, you could do all of things within WRDS SAS studio (except for the Chava Roberts linking table), but it would be much more painful, especially if you want to incorporate data that isn’t inside of WRDS (like I did with the linking table). You could also write your own SQL statement that does the exact same thing as well, but this will be harder (though ChatGPT can help).

Data Exploration Tips:

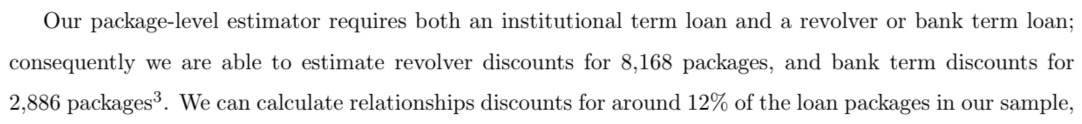
There are many philosophies about how you should pursue a research project. No matter which you follow, at some point you want to really “know the data” and the “stylized facts”. To do this, this will require you to run countless summary statistics and produce countless figures.

One of the first things I will always do when I am “getting to know” my data is to plot all outcomes of interest and all control variables, and often split them up by groups of interest (I personally like kernel densities). This does two things, first it might show some cool empirical facts that will help you know the world better, and it will also reveal to you obvious data issues (say that some of your data has a value multiplied by 100). For example, such exploration led me to see that vertically integrated loans in commercial mortgages are generally lower risk loans with lower rates and have faster securitization speed. Doing this led me to my job market paper, as I found some empirical facts I found interesting and wanted to understand why it was happening.

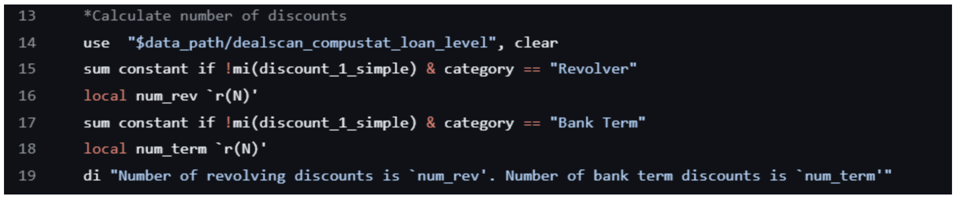
You may be worried about “data mining.” I am not advocating for data mining, but the distinction is not always clear. I genuinely want to know what the data says and what the correlations in the data truly are. And maybe this will lead to cool economics (e.g. my JM paper), maybe it will be uninteresting economics, or maybe it will be spurious. That last question is one you will discuss with your advisors and peers to figure out.

Other Workflow tips:

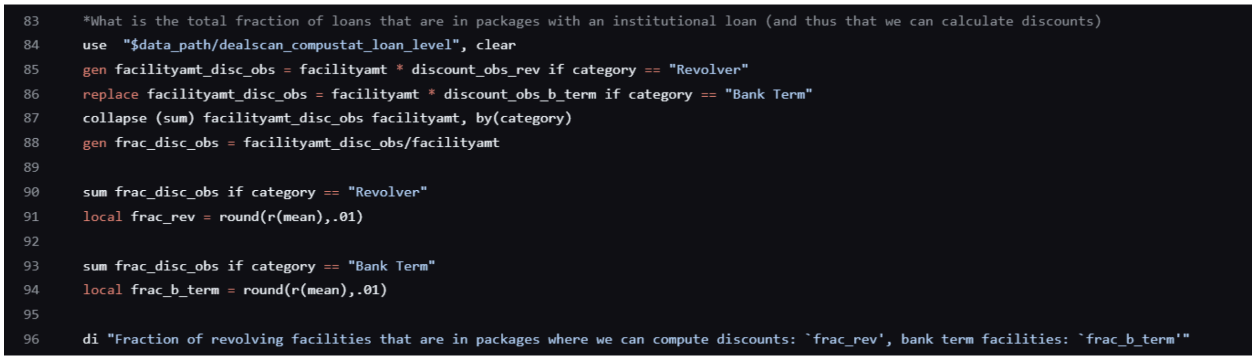
If in your text, you include some summary statistics based off of your data, make sure that you have code that generates as output to the command line. For example, in my paper “Bank Relationships and the Pricing of Loans”, we have the following paragraph


Our package-level estimator requires both an institutional term loan and a revolver or bank term loan; consequently we are able to estimate revolver discounts for 8,168 packages, and bank term discounts for 2,886 packages[3]. We can calculate relationships discounts for around 12% of the loan packages in our sample,

These three numbers come directly from this Stata code



```
*Calculate number of discounts
use  "$data_path/dealscan_compustat_loan_level", clear
sum constant if !mi(discount_1_simple) & category == "Revolver"
local num_rev `r(N)'
sum constant if !mi(discount_1_simple) & category == "Bank Term"
local num_term `r(N)'
di "Number of revolving discounts is `num_rev'. Number of bank term discounts is `num_term'"
```

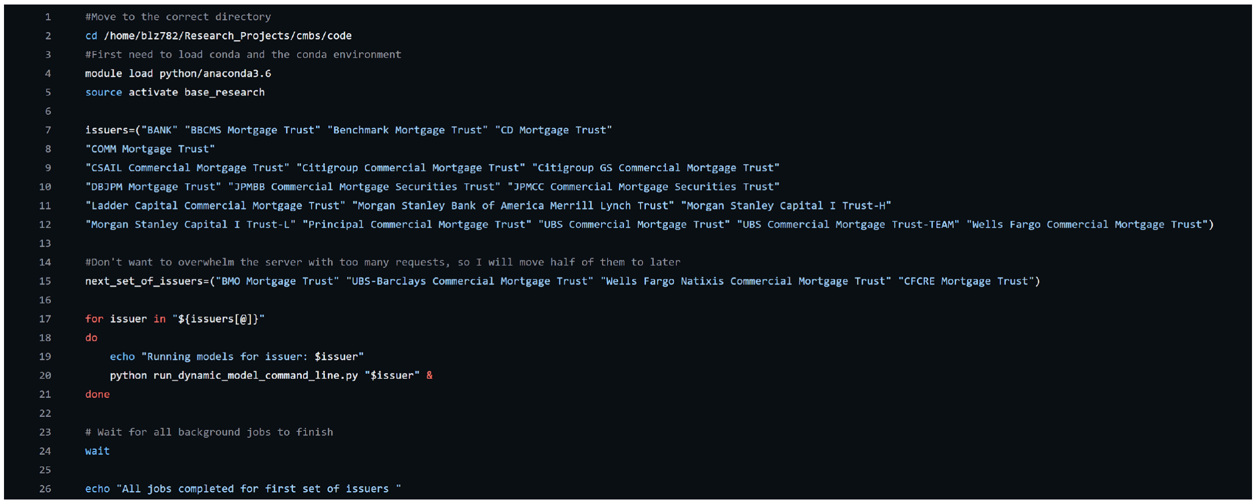


```
*What is the total fraction of loans that are in packages with an institutional loan (and thus that we can calculate discounts)
use  "$data_path/dealscan_compustat_loan_level", clear
gen facilityamt_disc_obs = facilityamt * discount_obs_rev if category == "Revolver"
replace facilityamt_disc_obs = facilityamt * discount_obs_b_term if category == "Bank Term"
collapse (sum) facilityamt_disc_obs facilityamt, by(category)
gen frac_disc_obs = facilityamt_disc_obs/facilityamt

sum frac_disc_obs if category == "Revolver"
local frac_rev = round(r(mean),.01)

sum frac_disc_obs if category == "Bank Term"
local frac_b_term = round(r(mean),.01)

di "Fraction of revolving facilities that are in packages where we can compute discounts: `frac_rev', bank term facilities: `frac_b_term'"
```

Parallelization:

In your research, you may find yourself needing to run the same procedure many times. If you are clever in how you write your code and have access to a Linux cluster, you may be able to use parallelization. Parallelization is the idea of running procedures at the same time, which speeds up your code's runtime. In practice, this is done by submitting many "jobs", each of them can be run independently of each other. A common way to do this is to write a batch script that you will run in the command line, that will submit many jobs.

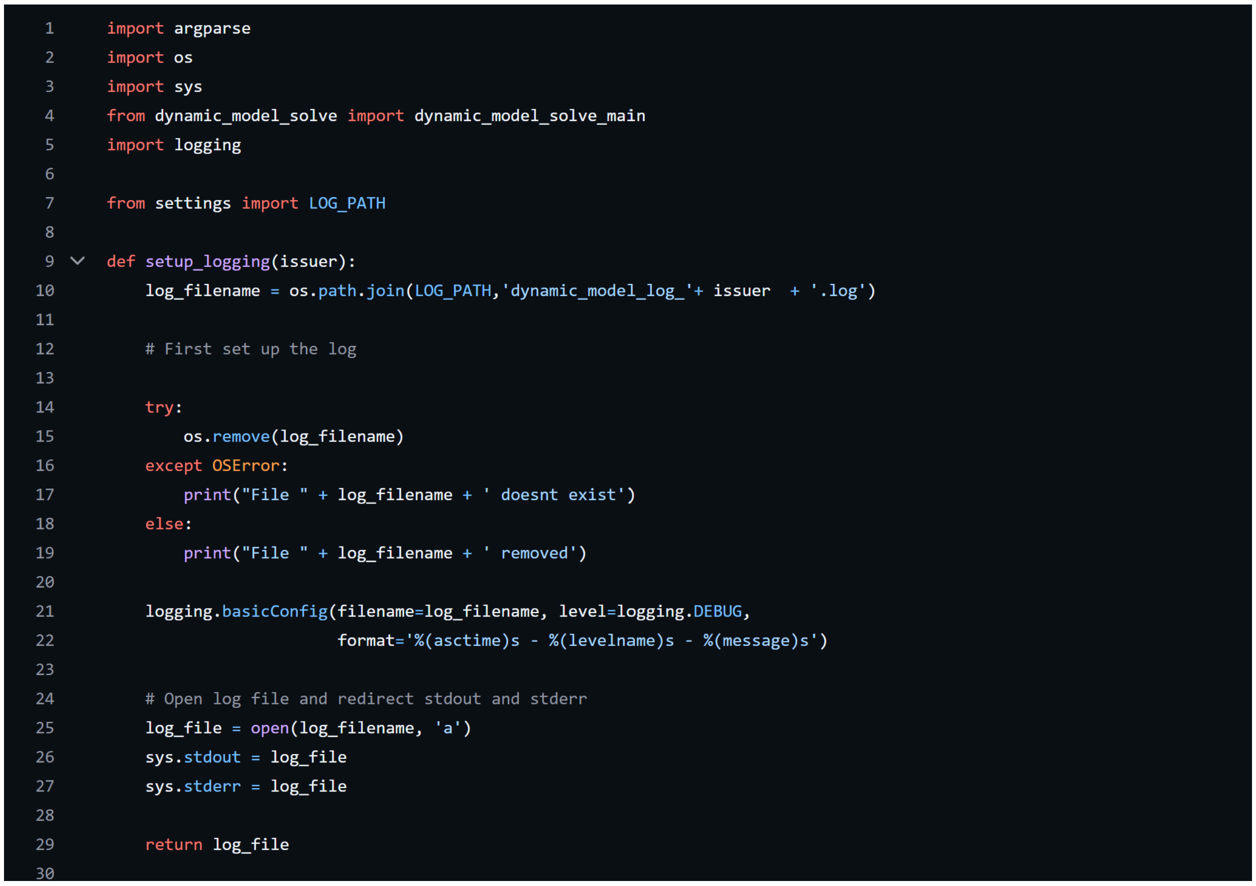
Below is an example from my job market paper, where I need to solve a dynamic model for over 20 issuers of CMBS.



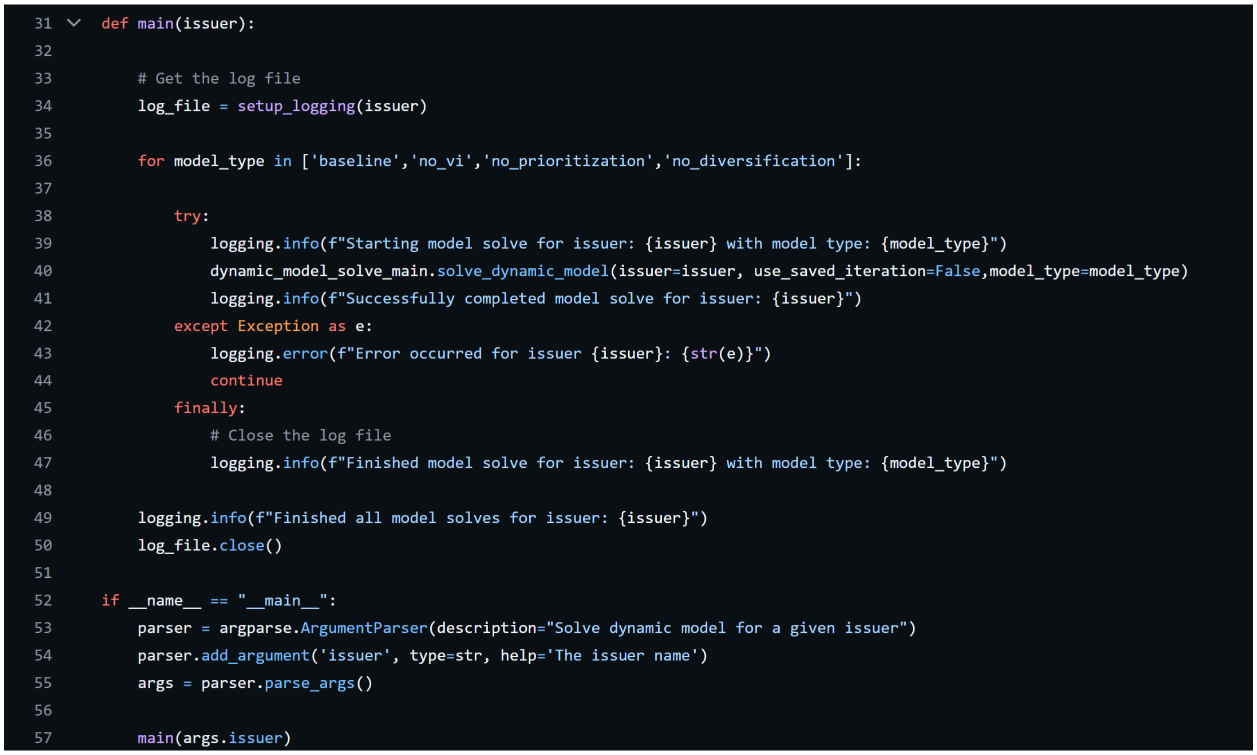
```
#Move to the correct directory
cd /home/blz782/Research_Projects/cmbs/code
#First need to load conda and the conda environment
module load python/anaconda3.6
source activate base_research

issuers=("BANK" "BBCMS Mortgage Trust" "Benchmark Mortgage Trust" "CD Mortgage Trust"
"COMM Mortgage Trust"
"CSAIL Commercial Mortgage Trust" "Citigroup Commercial Mortgage Trust" "Citigroup GS Commercial Mortgage Trust"
"DBJPM Mortgage Trust" "JPMBB Commercial Mortgage Securities Trust" "JPMCC Commercial Mortgage Securities Trust"
"Ladder Capital Commercial Mortgage Trust" "Morgan Stanley Bank of America Merrill Lynch Trust" "Morgan Stanley Capital I Trust-H"
"Morgan Stanley Capital I Trust-L" "Principal Commercial Mortgage Trust" "UBS Commercial Mortgage Trust" "UBS Commercial Mortgage Trust-TEAM" "Wells Fargo Commercial Mortgage Trust")

#Don't want to overwhelm the server with too many requests, so I will move half of them to later
next_set_of_issuers=("BMO Mortgage Trust" "UBS-Barclays Commercial Mortgage Trust" "Wells Fargo Natixis Commercial Mortgage Trust" "CFCRE Mortgage Trust")

for issuer in "${issuers[@]}"
do
    echo "Running models for issuer: $issuer"
    python run_dynamic_model_command_line.py "$issuer" &
done

# Wait for all background jobs to finish
wait

echo "All jobs completed for first set of issuers "
```

This batch script sets the directory to the correct folder, loads python, activates my Conda environment. Then it loops over all of the issuers that I need to solve the dynamic model for, and submits at python job “run_dynamic_model_command_line.py” with the input “$issuer”

The python file “run_dynamic_model_command_line.py” that is called is below:



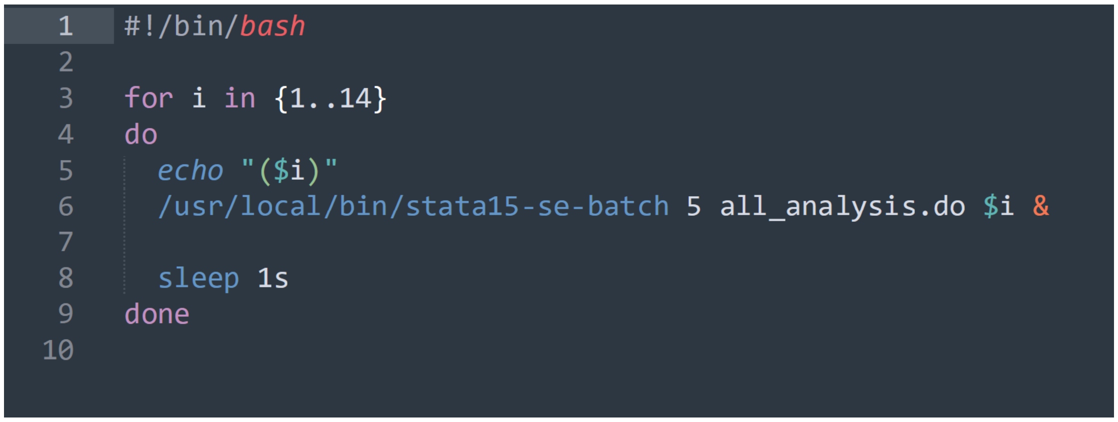
```python
import argparse
import os
import sys
from dynamic_model_solve import dynamic_model_solve_main
import logging

from settings import LOG_PATH

def setup_logging(issuer):
    log_filename = os.path.join(LOG_PATH,'dynamic_model_log_'+ issuer  + '.log')

    # First set up the log

    try:
        os.remove(log_filename)
    except OSError:
        print("File " + log_filename + ' doesnt exist')
    else:
        print("File " + log_filename + ' removed')

    logging.basicConfig(filename=log_filename, level=logging.DEBUG,
                        format='%(asctime)s - %(levelname)s - %(message)s')

    # Open log file and redirect stdout and stderr
    log_file = open(log_filename, 'a')
    sys.stdout = log_file
    sys.stderr = log_file

    return log_file

```



```python
def main(issuer):

    # Get the log file
    log_file = setup_logging(issuer)

    for model_type in ['baseline','no_vi','no_prioritization','no_diversification']:

        try:
            logging.info(f"Starting model solve for issuer: {issuer} with model type: {model_type}")
            dynamic_model_solve_main.solve_dynamic_model(issuer=issuer, use_saved_iteration=False,model_type=model_type)
            logging.info(f"Successfully completed model solve for issuer: {issuer}")
        except Exception as e:
            logging.error(f"Error occurred for issuer {issuer}: {str(e)}")
            continue
        finally:
            # Close the log file
            logging.info(f"Finished model solve for issuer: {issuer} with model type: {model_type}")

    logging.info(f"Finished all model solves for issuer: {issuer}")
    log_file.close()

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Solve dynamic model for a given issuer")
    parser.add_argument('issuer', type=str, help='The issuer name')
    args = parser.parse_args()

    main(args.issuer)
```

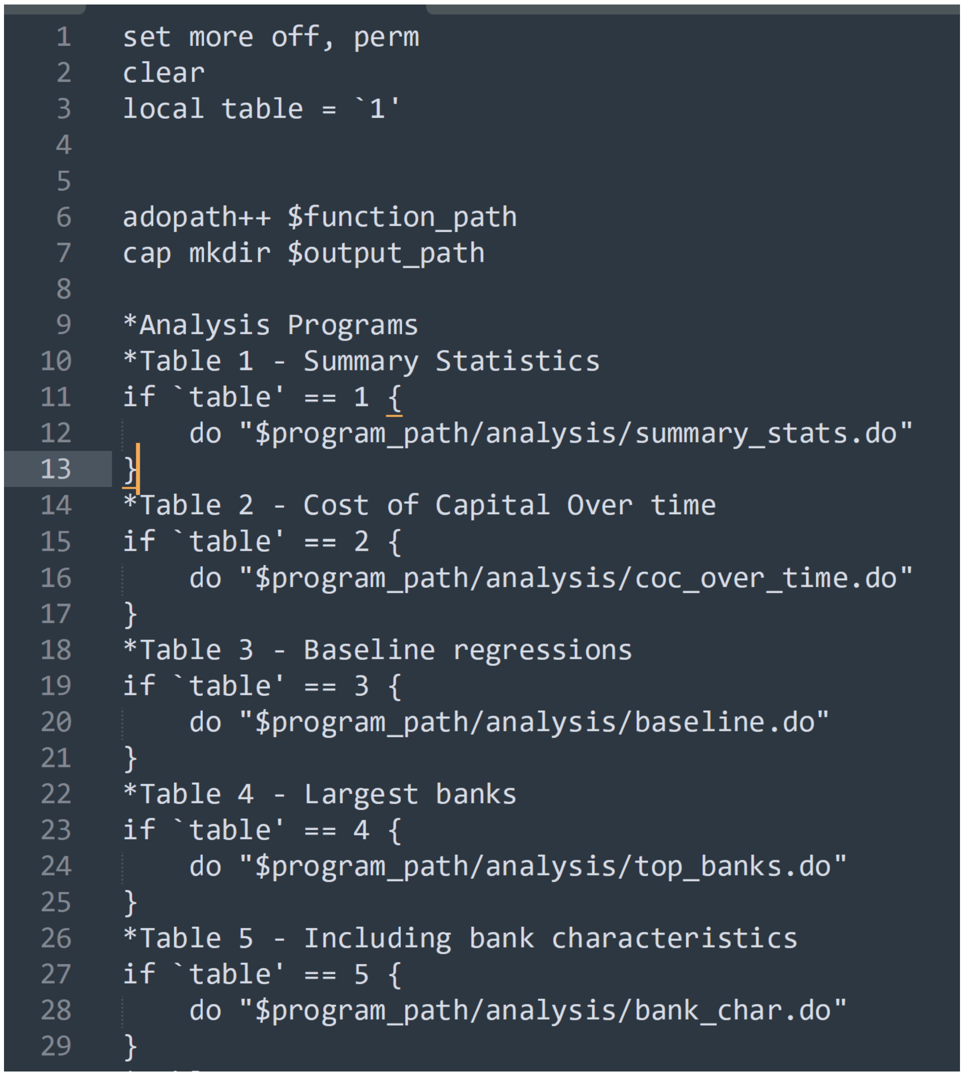
This program takes in an “issuer” as an argument, and then runs the function “dynamic_model_solve_main.solve_dynamic_model” for the given issuer (I solve 4 versions of the model, the baseline and three counterfactuals). The beauty of this is that the dynamic model that one issuer solves is independent of every other issuer, and so I can do them simultaneously. So if I have 24 issuers, and each model takes 1 hour to solve, I can solve all of them in 1 hour, instead of 1 day.

Another example where I do parallelization in Stata is below. I had 14 different analysis that would take a while to run, so instead of just running them one at a time, I had a batch script that would run all 14 at the same time.



```bash
#!/bin/bash

for i in {1..14}
do
  echo "($i)"
  /usr/local/bin/stata15-se-batch 5 all_analysis.do $i &

  sleep 1s
done

```

This code runs the “all_analysis.do” file with an input of “i”, where “i” is a number between 1 and 14. I am only showing 5 of the analyses below.



```
set more off, perm
clear
local table = `1'


adopath++ $function_path
cap mkdir $output_path

*Analysis Programs
*Table 1 - Summary Statistics
if `table' == 1 {
    do "$program_path/analysis/summary_stats.do"
}
*Table 2 - Cost of Capital Over time
if `table' == 2 {
    do "$program_path/analysis/coc_over_time.do"
}
*Table 3 - Baseline regressions
if `table' == 3 {
    do "$program_path/analysis/baseline.do"
}
*Table 4 - Largest banks
if `table' == 4 {
    do "$program_path/analysis/top_banks.do"
}
*Table 5 - Including bank characteristics
if `table' == 5 {
    do "$program_path/analysis/bank_char.do"
}
```

My opinion on coding languages:

I think that depending on your purpose, different coding languages are useful. Below I will summarize my opinion on each of the main ones economists use.

Julia – Good for solving macro style models in fast time. Upgrade over Fortran (don’t learn Fortran), which is also made for solving models fast. I personally don’t use it but have heard good things.

Python: An incredibly powerful object-oriented programming language. Is good webscraping, processing data, and doing “data-science” things and machine learning. Can also do parameter estimation (MLE and GMM) and solve simpler dynamic models.

I use Python extensively, for both data gathering and processing, and for estimation and solving dynamic models.

R: There is a large overlap between what R can do and what Python can do. R is not an object-oriented programming language. But it can do the things I have described above for Python. Additionally, there are a lot of new regression commands that are easy and fast to use, so if you want to have an entire project in just one language, R is probably the way to go. (I personally do not like the syntax of R so I don’t use it, but it is quite powerful)

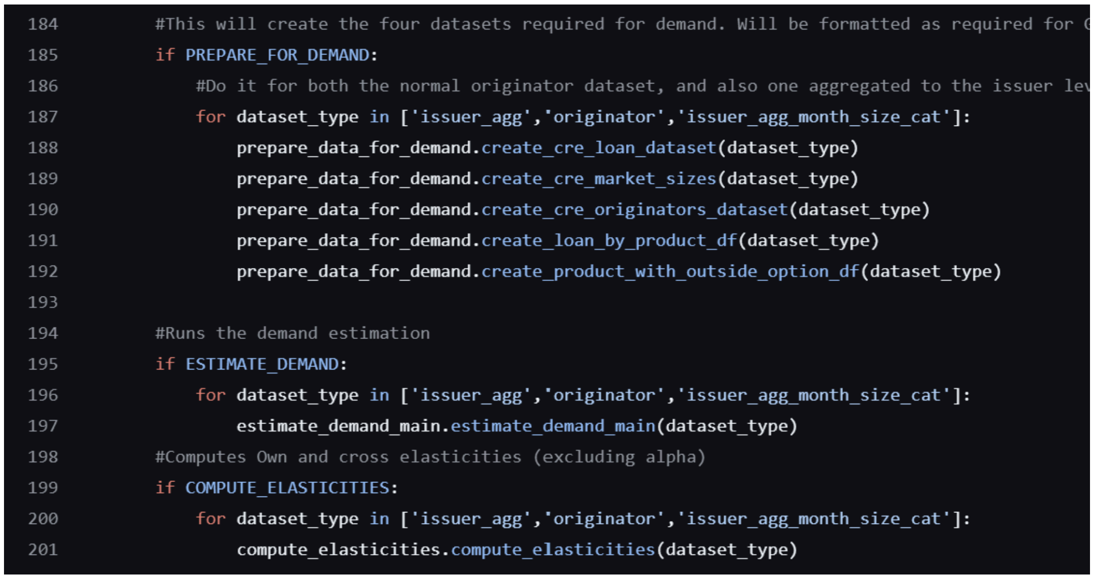
Matlab: A programming language used for computational stuff (e.g. solving a dynamic programming model). This is a bad choice for data analysis. Even merging datasets is hard in Matlab. You shouldn’t use it. If you know matlab, it’s easy to transition to R or Python, which can do the same things and more. I think the reason it is used is that people don’t want to learn new languages.

Stata: Great if you already have a relatively clean dataset, or only requires minor cleaning/merging. Where Stata is excellent is that it has all of the regression and summary stat tools you need to generate the analysis tables and figures you need. I use Stata to do reduced form analyses and explore data. I like the figures and esttab commands much more than Python’s matplotlib. I have heard Stata referred to as the “Excel for Economists” and I think that is accurate. Excel is actually great for simple things, just like Stata is. Please don’t ever try to solve a dynamic model in Stata, or do GMM, or estimate parameters. Only reduced form regression and summary stats.

Estimation or Model Solving Code Organization:

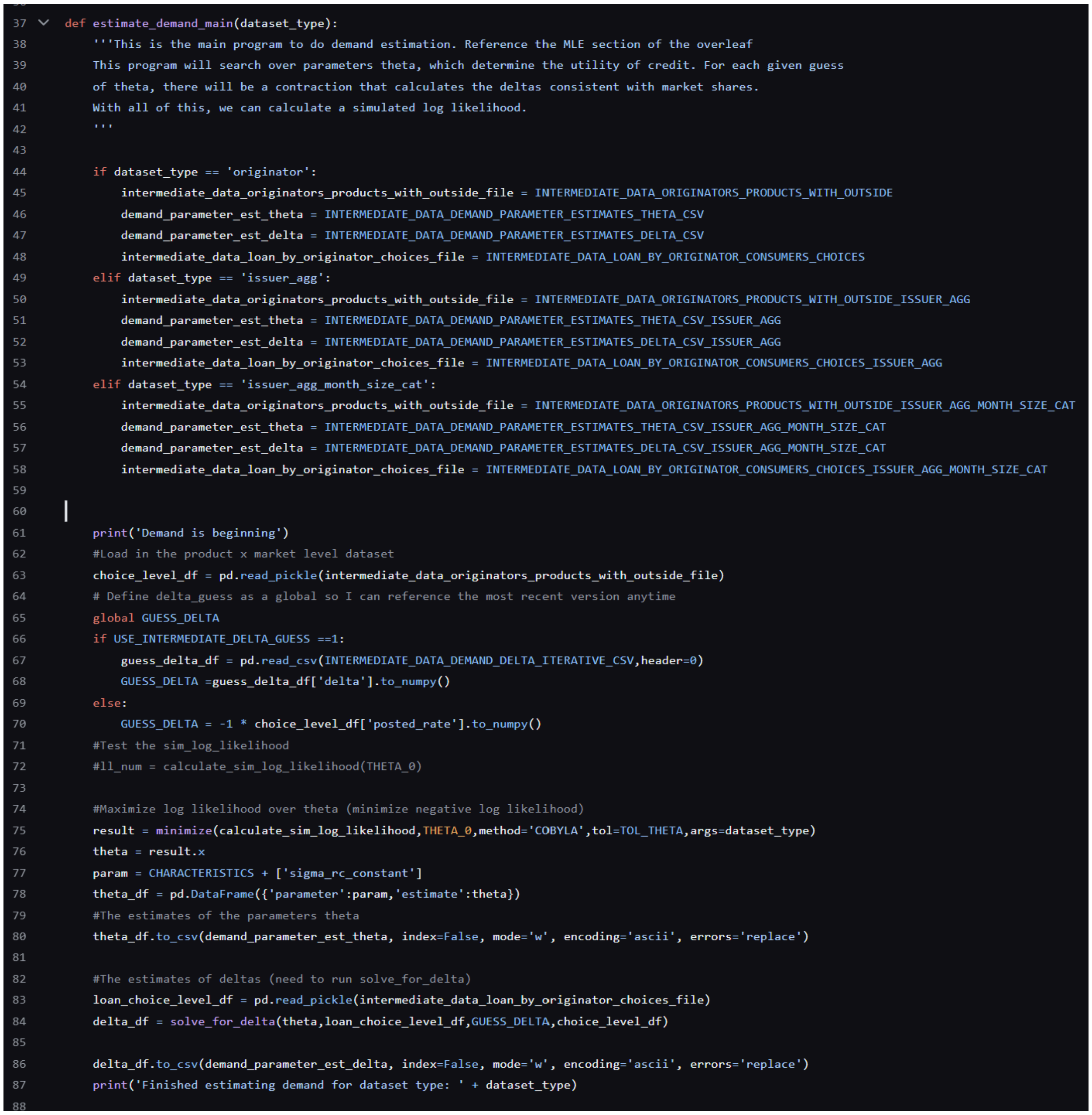
If you are estimating parameters using some sort of a GMM or MLE procedure, you should structure your code in a similar way (have a master file, and then have programs that read in the data, and then other programs compute summary statistics). I will use my CMBS dynamic model code to illustrate.

First you should have code that prepares all of the datasets and inputs you need to actually estimate demand. In addition to looking cleaner, this will speed up your procedure because these things only need to be done once! Not every time during your estimation (see PREPARE_FOR_DEMAND)



```
        #This will create the four datasets required for demand. Will be formatted as required for C
        if PREPARE_FOR_DEMAND:
            #Do it for both the normal originator dataset, and also one aggregated to the issuer lev
            for dataset_type in ['issuer_agg','originator','issuer_agg_month_size_cat']:
                prepare_data_for_demand.create_cre_loan_dataset(dataset_type)
                prepare_data_for_demand.create_cre_market_sizes(dataset_type)
                prepare_data_for_demand.create_cre_originators_dataset(dataset_type)
                prepare_data_for_demand.create_loan_by_product_df(dataset_type)
                prepare_data_for_demand.create_product_with_outside_option_df(dataset_type)

        #Runs the demand estimation
        if ESTIMATE_DEMAND:
            for dataset_type in ['issuer_agg','originator','issuer_agg_month_size_cat']:
                estimate_demand_main.estimate_demand_main(dataset_type)
        #Computes Own and cross elasticities (excluding alpha)
        if COMPUTE_ELASTICITIES:
            for dataset_type in ['issuer_agg','originator','issuer_agg_month_size_cat']:
                compute_elasticities.compute_elasticities(dataset_type)
```

Then, you will have code the runs the estimation (see estimate_demand_main)



```python
def estimate_demand_main(dataset_type):
    '''This is the main program to do demand estimation. Reference the MLE section of the overleaf
    This program will search over parameters theta, which determine the utility of credit. For each given guess
    of theta, there will be a contraction that calculates the deltas consistent with market shares.
    With all of this, we can calculate a simulated log likelihood.
    '''

    if dataset_type == 'originator':
        intermediate_data_originators_products_with_outside_file = INTERMEDIATE_DATA_ORIGINATORS_PRODUCTS_WITH_OUTSIDE
        demand_parameter_est_theta = INTERMEDIATE_DATA_DEMAND_PARAMETER_ESTIMATES_THETA_CSV
        demand_parameter_est_delta = INTERMEDIATE_DATA_DEMAND_PARAMETER_ESTIMATES_DELTA_CSV
        intermediate_data_loan_by_originator_choices_file = INTERMEDIATE_DATA_LOAN_BY_ORIGINATOR_CONSUMERS_CHOICES
    elif dataset_type == 'issuer_agg':
        intermediate_data_originators_products_with_outside_file = INTERMEDIATE_DATA_ORIGINATORS_PRODUCTS_WITH_OUTSIDE_ISSUER_AGG
        demand_parameter_est_theta = INTERMEDIATE_DATA_DEMAND_PARAMETER_ESTIMATES_THETA_CSV_ISSUER_AGG
        demand_parameter_est_delta = INTERMEDIATE_DATA_DEMAND_PARAMETER_ESTIMATES_DELTA_CSV_ISSUER_AGG
        intermediate_data_loan_by_originator_choices_file = INTERMEDIATE_DATA_LOAN_BY_ORIGINATOR_CONSUMERS_CHOICES_ISSUER_AGG
    elif dataset_type == 'issuer_agg_month_size_cat':
        intermediate_data_originators_products_with_outside_file = INTERMEDIATE_DATA_ORIGINATORS_PRODUCTS_WITH_OUTSIDE_ISSUER_AGG_MONTH_SIZE_CAT
        demand_parameter_est_theta = INTERMEDIATE_DATA_DEMAND_PARAMETER_ESTIMATES_THETA_CSV_ISSUER_AGG_MONTH_SIZE_CAT
        demand_parameter_est_delta = INTERMEDIATE_DATA_DEMAND_PARAMETER_ESTIMATES_DELTA_CSV_ISSUER_AGG_MONTH_SIZE_CAT
        intermediate_data_loan_by_originator_choices_file = INTERMEDIATE_DATA_LOAN_BY_ORIGINATOR_CONSUMERS_CHOICES_ISSUER_AGG_MONTH_SIZE_CAT


    print('Demand is beginning')
    #Load in the product x market level dataset
    choice_level_df = pd.read_pickle(intermediate_data_originators_products_with_outside_file)
    # Define delta_guess as a global so I can reference the most recent version anytime
    global GUESS_DELTA
    if USE_INTERMEDIATE_DELTA_GUESS ==1:
        guess_delta_df = pd.read_csv(INTERMEDIATE_DATA_DEMAND_DELTA_ITERATIVE_CSV,header=0)
        GUESS_DELTA =guess_delta_df['delta'].to_numpy()
    else:
        GUESS_DELTA = -1 * choice_level_df['posted_rate'].to_numpy()
    #Test the sim_log_likelihood
    #ll_num = calculate_sim_log_likelihood(THETA_0)

    #Maximize log likelihood over theta (minimize negative log likelihood)
    result = minimize(calculate_sim_log_likelihood,THETA_0,method='COBYLA',tol=TOL_THETA,args=dataset_type)
    theta = result.x
    param = CHARACTERISTICS + ['sigma_rc_constant']
    theta_df = pd.DataFrame({'parameter':param,'estimate':theta})
    #The estimates of the parameters theta
    theta_df.to_csv(demand_parameter_est_theta, index=False, mode='w', encoding='ascii', errors='replace')

    #The estimates of deltas (need to run solve_for_delta)
    loan_choice_level_df = pd.read_pickle(intermediate_data_loan_by_originator_choices_file)
    delta_df = solve_for_delta(theta,loan_choice_level_df,GUESS_DELTA,choice_level_df)

    delta_df.to_csv(demand_parameter_est_delta, index=False, mode='w', encoding='ascii', errors='replace')
    print('Finished estimating demand for dataset type: ' + dataset_type)
```

Finally, you will have code that does something with the estimates (not shown here)

Setting up Git/ Github (relevant files here: [“.bashrc”](), [“.gitconfig”](), and [“.gitignore”]()**):**

You have all heard of Github and you should get in the habit of Git tracking every research project you do. This is essential when working with collaborators but also useful if it is just you. Git is a version tracking software that allows you to periodically “commit” your code to a Git repository. Then if you ever need to go back to an old version, you can ask Git to bring back an old version for you.

Why should you use Git?

- With multiple people working on the same project, you can easily both code at the same time and “push” your changes to a centralized Git repository.
- This includes Git’s fantastic tools for merging in multiple edits and dealing with conflicts in code
- You never have to save files like “regressions_v2”
- You never have to “comment out code just in case I need it later” because you can just get your old code later
- You can have confidence deleting old code
- When you inevitably do something that breaks your code, you can go back to an old version that works (this will happen at some point in your career!)

Okay so how do you do it? You first need to set up a project directory, which should contain a “code” directory. Once you have that code directory, navigate to that directory using the “cd […]” commands. Once there you will initiate the repo by using the command “git init”.

Your git repo has now been created. To see the status of it type “git status”. You will see nothing is being tracked yet. So you need to tell git to start tracking files by typing “git add […]” where […] is the file name you want tracked. If you want git to track everything in the directory you can type “git add -A”.

After you have added everything, type “git status” again and git will tell you exactly what it will start tracking. To start tracking, you will make commits, which are a snapshot of your files you are tracking. To take the snapshot, type ‘git commit -m “message to your future self about this commit” ‘.

If you are not working with anyone but yourself, this is enough (though not the best practice). If you want to work with others or if you want to connect it to Github, you will need to start “branching”, which basically tells Git that there is a “master” version of the code repository somewhere and you/others can access that and contribute to it elsewhere. Github and other sites will have instructions on how to connect your local Git repository to Github/ make branches.

It is important to ONLY track your code directory in Git. Code files (like .m, .do, .py) are tiny in terms of the space they take, so it is fine to store countless versions of them in Git. But you can’t do it with output files and data files. Git is not made for that.

Also, there is a file called a “.gitignore” which you can include in your Git repo (put it directly where your git repo is) that tells git to not track certain files.

Lastly there is a git configuration file called “.gitconfig” which you should include in your HOME directory. This allows you to make customizations that make Git look nicer and that are outside the scope of this document. I would recommend you directly put my ".gitconfig" file in your home directory.

Estimation or Model Solving Tips:

When coding like this, I find it especially important to have cleaned, well commented, and blocked code, because you WILL need to debug issues here. You are doing something quite complex and it's incredibly easy to make mistakes.

Additionally, how you structure your data is very important here. I personally like to use pandas data frames because they are intuitive (they allow you to have many variables in the same object) and also they will ensure you don't have issues where you have different objects with different orderings of the index. It also makes your code easier to read in my opinion. I had to do a SMLE and a pandas dataframe made the coding more manageable.

The other thing about solving models/estimation is there will be problem-specific enhancements that you will include in your code that you will have to put in iteratively. For example, for my job market paper, I had 9 decisions that had to be made by agents, and one of them was much faster to compute than the other 8, so I would iterate on the value function, only recomputing the fast one many times, before ultimately recomputing the other 8 optimal actions. This allowed for much faster convergence.

Please, make sure you are vectorizing your code (pandas will automatically do that for you).

If you are doing a model that takes a "long time", then use timers to figure out how to optimize your code in different sections. For example, if you have a long task, time each subtask and see where the bottle-neck is (which takes more time). Then figure out how to make that segment of the code more efficient.

Additional workflow guides:

See Matthew Gentzkow and Jesse Shapiro's "Code and Data for the Social Sciences: A Practitioner's Guide" https://web.stanford.edu/~gentzkow/research/CodeAndData.pdf

Sean Higgin's Best Practices https://seankhiggins.com/code/