Coding Primer: A Reference Guide

July 30, 2020

Note: this tutorial is work in progress. If you find typos or have suggestions please email me at benjaminvatterj@u.northwestern.edu

1 Preliminaries: Coding Environment, Projects Structure and Versioning

Here we discuss how to setup your coding environment, organize your code and keep track of the versions of your projects. Although the majority of this handout consists of references to other sources and exercises, this chapter is mostly self contained. It is worth stating that the ideas expressed here are our personal views on the subject, although other guides have been written on the subject. For other references see Gentzkow and Shapiro's (2014) "Code and data for the social sciences: a practitioner's guide".

1.1 Pre-preliminary

This guide is set up to work with Python 3 and Git. Installing and managing these tools requires some prior knowledge of the terminal. We recommend that you begin by reading the appendix of "Python the Hard Way" (PHW) called "Command Line Crash Course". Once you have done this, go ahead and install these before doing anything else. How you go about installing it depends on your OS, but some references are provided in chapter 0 of PHW.

A couple of warnings regarding PHW. First, be careful when looking for a version of PHW online, because the first three editions where written for Python 2, which is now deprecated. Second, PHW is known as one of the best books for learning how to program in python, however it is oriented to a very general audience. This implies that it will assume that you know basically nothing about coding, which is likely to not be the case if you are a graduate student. This might make the exercises a bit slow for some of you, so feel free to skip the things that you deem obvious. Finally, PHW recommends using Atom which is very similar to Sublime and suggests avoiding VIM. Starting with Atom/Sublime is of course great if you are completely inexperienced. If, however, you have coded before it might worth to make the investment and learn how to use something more sophisticated.

1.2 Coding Environment

Having a good coding environment is crucial for an efficient work-flow. Many economics graduate students are used to working in integrated development settings such as the ones provided by Stata or Matlab. However, the majority of coding actually takes place in non-integrated environments, and for good reasons: they are agnostic to the language, they do not hang when you run your code (unlike Stata), and they are often customizable to your liking. In general, we would like our editor to have a series of features:

1. File management: often a side bar that can be toggled on and off to create, delete, move files without having to leave the editor.

2. Tag/Jump-to capabilities: Codes tend to become large. Being able to quickly view the list of functions that are defined in a side panel, and jump to them through a click, is extremely useful. Some even let you define your own tags as you work, to move around easily.

3. Linters: Linters provide integrated code-checking to your editor (if you are familiar with Matlab, this when matlab draws a red or yellow line while you code). There are linters for almost all languages and a good editor should recognize the language you are working in and activate the corresponding linter (if installed).

4. Git gutter: In the next subsection we will talk about versioning. For now, think of a Git gutter as something that marks what part of your code is new relative to the last time you worked on it. This is really useful when the code become long.

5. Search/Replace: now this sounds obvious, but wait until you learn how good a **good** search and replace can be.

6. Split screen: Split screens are very useful when you are debugging and trying to connect the different parts of your code and remember how they interact.

There are many (MANY) alternatives out there, but here I will just focus on two alternatives. I would recommend you pick one of the two and try it for a bit, and after you feel comfortable with it, look around for the things you think it is missing. Disclaimer: yes, you can run Stata and Matlab code directly from external editors.

Sublime Text (the easy way): Sublime is a very popular editor. It comes with very few capabilities, but you can install plugins that do the rest for you. There are many tutorials on how to setup Sublime Text (here is one). In general what you want to have in Sublime is the following:

1. Install subl, package control

2. Install some cool theme that makes it easy for you to read the code (e.g Molokai, Bad Wolf)

3. Install plugins: SidebBarEnhancements, SublimeLinter (and install a linter for the python 3 lenguage, such as pyflakes. Also install a linter for the proper coding style in python like pycodestyle), GitGutter, FTPSync, and Anaconda (if you want autocomplete while you type in Python).

4. Learn the keyboard shortcuts for: jump-to, toggeling the side bar, splitting the screen.

5. Extra credit: setup shortcuts to execute your code from the editor. Add CTags or other plugin that gives you a function declaration sidebar.

VIM (the hard way): The VIM editor has been around since the 90s, and it is still one of the most popular editors among programmers. Like sublime, it comes with almost nothing on it's own but it can be extended with plugins to get all the required feature. It has, however, three key distinctions that makes it a popular choice [1]:

1. Portability: VIM is available on all systems, and it is preinstalled in almost all Linux systems. Additionally, all your configuration is held on a single file. These two features mean that you can have your favorite setup on every computer you connect to, edit remotely on servers as if it was on your own system.

2. Programmatic Macros and Search/Replace: In VIM, you interact with the editor through commands which can be recorded and repeated. The search/replace option has a powerful Regular Expression that allows for conditionals, groups and multiple matches.

3. Lightweight: Finally, VIM uses very few resources. Unlike popular editors such as Sublime and Atom, VIM doesn't eat into you RAM as much.

The downside of VIM however is it's difficulty (The stack overflow page on how to exit VIM has over a million views!). The learning curve is steep but it's worth it. If you want to suffer a little bit, I have left my configuration file in this chapter's folder. How you install VIM depends on your platform and it is likely to already be installed. Before using my ".vimrc" file you should install vim-plug. Some tutorials on vim you should check:

- Tutorial by Daniel Miessler
- Interactive Guide
- There's even a game!

1.3 Versioning: GIT

Have you been keeping "my_script_v1.do" to "my_script_v231.do" in a messy folder in which you can no longer remember what was the difference between version 2 and 100? then versioning is what you are doing wrong. Vim is the supreme-ruler of versioning software.

[1]Another great tool in VIM is GUNDO, which is an undo feature that keeps track of your edits in a tree structure. If you have ever done a ctrl+z then small typo an then tried to revert without luck, then you will appreciate GUNDO.

It allows you to keep your folder clean by maintaining versions of your code behind the scene. Everyone should use it (or other versioning tool) as it is useful both for coding and for writing in latex. Enough said, here are some tutorials:

1. A learning-by-doing tutorial. Recommended by Github
2. The Bitbucket tutorial
3. The Vogella tutorial, form basic to really advanced.

1.4 Project Structure

This is the most personal part of this chapter. The way we organize code in folders makes a big difference in the long run. In general, I recommend you always code knowing that you will forget why and how you did things. Keep things clean and clear, and things should work out fine. Here I present the way I do things and why.

Broadly speaking, we can classify codes in economics into two categories, which for lack of better names we will call: *Sequential* and *Modular*. This is a separation that often follows a chronological order. Sequential codes are common in the first steps of a project, which involves writing a **sequence** of scripts that clean, merge and provide descriptive representations of your data. The most important part of this type of codes is the sequence in which things run. Additionally, it is often a stage that is very project-specific with little scope for reuse of previous scripts or functionalities. Another characteristic of this sort of stage is that once the data-stage of your project is done, you will never run these scripts again and review them very rarely. Therefore, *Sequential* parts should be clear about their sequence of execution. The simplest way of doing it is by naming files properly. For example, in this project the Data folder contains the following scripts:

```
Data/
|--- c0_parse_data.py
|--- c1_create_estimation_data.py
|--- c2_create_control_data.py
+--- README.txt
```

The README file provides a description of the purpose of the codes in this folder, while the three scripts provide a clear name and a sequence of operations. In this way, we can always remember that we should first create the estimation data and then the control, and not the other way around.

The second type of codes are what I have called *Modular*. These codes often define the key logic of the project: utility functions, demand systems, numerical optimizer, estimators, etc. This type of code has two key characteristics that merits a different structure:

- Reusable structure: It is composed of components that will be used by many parts of the code. For example, to avoid repetition and error, we want to define the utility functions once and then use the definition whenever appropriate.
- Exploration: Part of research is that we are not sure on what will be the best way of doing things. This requires a project structure that is more flexible, where we can change one component and propagate automatically to all our code (e.g, changing the utility function) and where we can change the orders of operations easily (e.g, adding a pre-estimation or a counterfactual exercise).

The preferred structure for this sort of code is modular: one "main" file handles the calls to each component and a "package" directory encapsulate the different operations coded. Late on, when you get to the Object Orientation part of the guide, you will see how easy it is to keep things clean and modular in Python. In the example project of Illanes and Padi (2019), the (summarized) file structure looks like:

```
Project/
├── README.txt
├── install.py
├── main.py
├── annuity_model/
│       ├── README.txt
│       ├── integrate/
│       │       ├── IntegratorError.py
│       │       ├── Integrator.py
│       ├── simulate/
│       │       ├── Consumer.py
│       │       ├── SimulatorObject.py
│       └── utility/
│               ├── AnnuitySolver.py
│               ├── MixedSolver.py
│               ├── PWSolver.py
│               ├── UtilityObject.py
├── data/
├── data_creation/
│       ├── c0_parse_data.py
│       ├── c1_create_estimation_data.py
│       ├── c2_create_control_data.py
│       └── README.txt
├── docs/
├── output/
├── scripts/
```

In this project,there is always a README.txt file that describes the purpose of codes within each folder. The *install.py* script installs all necessary packages and verifies that all the necessary data are available. This enables portability of the project across systems and easy installation. The *main.py* is the key file that handles the calls to all the code parts. It provides the core-access to all the functions of the project and allows for exploration and testing. The *annuity_model* folder is actually a Python module! it include three submodules each encapsulating specific logic. For example the *utility* module includes all functionalities necessary to compute the utility of agents in the model. The *simulate* module creates simulated consumers, each of which knows how to compute utilities using the *utility* module. Finally, *integrate* encapsulate specific mathematical tools for numerical integration which are used by the *utility* module and others. Below the main package we see that *data* folder where data is stored, the *data_creation* folder with our *sequential* logic, the *output* folder to store results and the *docs* folder where the paper is written. Finally, *scripts* includes other scripts that use the *annuity_model* module but that require longer execution and specific logic that falls out of the scope of the *main.py* file.

Hopefully, the usefulness of proper structuring is apparent. The idea is to always code defensively: be prepared for errors and for you own forgetfulness. A nice principle to enforce is the DRY (Don't Repeat Yourself). If you are repeating code over and over, it should be encapsulated and a good project structure is a necessary condition to do so.

1.5 Key Exercise

By the end of this section you should have an editor with a Python linter enabled. To test your new skills, migrate an old project or homework to a new folder. Structure properly according to the principles above and initialize a git repository in the folder. Start a debug branch, switch branch, and edit the code. Then return to the main branch and merge the debug branch.

2 Python Basics

Teaching how to code properly in any language is a difficult task. For this section we recommend that you read and work through chapters 1 to 40 of PHW. Although it sounds like a lot, it is actually a fast tutorial through all the basics of Python. We recommend that you actually type and try most (if not all) the scripts presented in PHW.

2.1 Key Exercise: A Scaffolding Script

Now that you know the basics, you are ready to create a simple *scaffolding* script. This script will be used to create new empty projects in your *projects folder*. To do so we will require some useful built-in libraries of Python 3 which PHW doesn't cover. In particular, we will use the *argparse* module to parse inputs provided to the script and automatically create a help text. We will use the *datetime* module to get the current date, and the *os* module to manage the directory. Finally, we will use the *subprocess* library to initialize a git repository in the newly created folder. How you decide to go about creating the script is up to you, but it should satisfy the following requirements

1. The script should be executable from the command line (hint: check "chmod +x" in linux/mac).
2. The script should take two arguments: a full project title (title) and a short name (codename).
3. Using these arguments, the script should:
   (a) Check that a folder named as the codename doesn't exists, and if it does abort the operation.
   (b) If the folder doesn't exist, create it together with a code, docs and data folder within it (hint: check os.path).
   (c) Create a README file in folder that contains the title of the project, the date of creation and a line for inserting the idea behind the project.
   (d) (optional) It should open the README in a text editor such that the user can fill the idea section comfortably.
   (e) Initialize a git repository in the folder and commit the current structure. (hint: it is easier to use subprocess.run with the shell=True argument)
4. The script should also contains a useful help message regarding the purpose of the script, displayed by running "$ ./new_project.py - -help"

For comparison only, we have left an example script in the section 2 resources which does the things above. Finally, if you have managed to make this script: make it your own. For example, you can add options to include coauthors, to initialize remote git repositories, to establish links with remote servers, etc.

3 Math in Python

Many economists are trained to do computational mathematics in Matlab. This chapter is dedicated to importing the tools you might be familiar with in Matlab to Python.

The two main libraries used in Python for computational mathematics and statistics are: NumPy and SciPy. The first contains the core functions, such as multidimensional arrays (or matrices), standard matrix algebra and operations, while the second is provides more advanced linear algebra, numerical integration and statistics. The following tutorials discuss the basics of NumPy:

1. Basic NumPy tutorial
2. Numpy for Matlab users

Here are some things you should look out for in NumPy and get familiar with:

1. The convention is to import NumPy as `np` (i.e `import numpy as np`). So you will often see python code that just uses `np` without showing the import row. For example something like `np.sum(x)` is using the `sum` function on some object `x`.

2. NumPy arrays are by default row-oriented (which is the C standard, instead of the column-orientation Fortran standard used by Matlab). This implies two important things regarding coding in NumPy versus Matlab. First, one-dimensional arrays are by default row-vectors. Second, operation that traverse an array row-wise are sometimes more efficient, as elements are stored in memory in row-order.

3. NumPy arrays are multidimensional. Particularly, one dimensional arrays have no second dimension to transpose on. To test this, create a random vector (`x = np.random.rand(500)`) and check its transposed shape (`x.T.shape`). You should see that it is still a row vector despite the transposition. The most efficient solution is to rewrite your algebra such that row vectors are in fact what you need. Alternatively, reshape to a column vector: `x.reshape(-1, 1)`.

4. As in Matlab, NumPy matrix operations benefit from powerful boolean and integer indexing. Often, messy and expensive loops over matrix rows and columns can be replaced by proper indexing.

5. NumPy offers automatic *broadcasting* capabilities which makes it's usage very efficient. This implies that if you want to add an $(N,)$ row-vector v to each row of an (M,N) matrix X, it suffices to write `X + v`. NumPy, will automatically broadcast v to the appropriate shape for the summation.

6. Importantly, by this point you should be familiar with the 0-based indexing of Python (instead of the 1-based indexing of Matlab). Numpy arrays are no different and 0 indicates the first element. Additionally, indexing in numpy (and python in general) can go backwards such that `x[-2]` fetches the penultimate element in the array `x`.

7. Make use of "out" arguments to gain efficiency. Most algebraic operations need to create new objects when returning results. This often comes at significant cost because the system needs to allocate the additional memory for this new result object. One very common example is the multiplication of two (N,N) matrices x and y and their storage back in x, i.e `x = x.dot(y)` or `x = np.matmul(x, y)` or `x = x @ y`.[2]

[2] The "at" (@) notation for matrix multiplication was only added starting Python 3.5. It is still common to find people using older versions of Python 3, so it might be good idea to avoid the "@" for the time being.

When doing this product, NumPy will first create a new matrix to store the results of the product in, and then copy the results back to x. This is clearly inefficient, as dot products can be computed in-place as `np.matmul(x,y, out=x)`. This saves the system the creation of temporary storage for the results which can sometimes make the difference between having the memory to execute an operation or not.

8. Make use of numpy's `np.any` and `np.all` functions for checking the truth status of arrays properly. For example, the two following examples check whether the equal-shaped vector x and y are equal, but the second is significantly more efficient: 1)`assert np.max(np.abs(x-y)) == 0`; 2) `assert np.all(x == y)`.

9. For more flavors of matrix multiplication check this question on Stack overflow. For more tips on efficient use of copying and broadcasting check this post.

10. Despite being dynamically typed, NumPy arrays have explicit datatypes. When memory is a constraint, setting the optimal data type can reduce memory usage significantly.

The SciPy library contains more advanced mathematical operation. In section 7 we will focus specifically on the numerical optimization methods included in SciPy. In addition to optimization, SciPy includes numerical integration, interpolation, and linear algebra capabilities among others. We recommend that you have a brief overview of the numerical integration, statistics and linear algebra tutorials:

1. Numerical Integration

2. Statistics

3. Linear Algebra

Here are some things you should look out for in SciPy and get familiar with:

1. Many of the linear algebra routines (eigen values, ranks, etc.) of SciPy are directly available in NumPy under `np.linalg`.

2. Be careful with the definition of particular distributions in the statistics packages. They are sometimes transformations of the expression used in Economics and require slight adjustments.

3. Numerical integration in SciPy is basic but efficient. For more advanced usage see QuadPy.

4. Scipy gives access to low-level BLAS function that given finer control of linear algebra function (link). With few exceptions, using these function directly is an overkill but experienced programmers might find it comforting to control all the parameters involved.

3.1 Key Exercise: Logit choice probabilities

We will now start to build towards the end goal of this coding tutorial: a functional and useful individual-level Logit demand estimation module. We will begin by creating the key functions that computes the likelihood of a series of choices given a set of covariates and parameters. We recommend that you begin by reviewing the chapter on Logit demand in Kenneth Train's book *Discrete Choice Method with Simulation* (link).

Begin by creating a new project called *coding_tutorial* using the script of the previous section. In the codes directory create a new folder called *logit_demand* and inside of it create an empty script called *logit_demand.py* and one called *test.py*. Before starting, here is an overview of what you will create. Within *logit_demand.py* you will create two functions:

1. `choice_probability(I, X, beta)`. The arguments for this function are first a vector of consumer indexes (I), second a matrix of consumer-choice attributes (X), and the coefficient vector (β). The vector I has a length equal to the number of individual-products in the data, while X should have the same number of rows as the length of I and the same number of columns as the length of β. This function should return a vector of the same shape as I which contains the probability with which each choice is made, assuming the outside option is normalized to 0 and not included among the choices.

2. `loglikelihood(Y, I, X, beta)`. This function should use the previous function to compute the **negative** log-likelihood of a binary-vector of choices Y. The function should return a scalar. We need to compute the negative of the log-likelihood because later we will aim to minimize this objective.

In the *test.py* you will create a simulation data set to test your script with.

Now, to the actual coding. We recommend that throughout you keep an IPython shell open to test your code as you make it. As always, this exercise involves a lot of researching functions on your own. All should be available within NumPy, Scipy or be part of the Python core. If they are not, then you are expected to code them yourself.

1. Begin by creating the two functions in *logit_demand.py*. Define them with empty bodies, returning some arbitrary value.

2. In *test.py* import the functions from *logit_demand.py* and create the following

   (a) Define at the top, following the imports and the definition of the script (Always start scripts with a multiline string describing what the script does), a settings section. There, define the following: number of individual `N=1000`; number of attributes `M=6`; range of products for each consumer `C=[3,6]`; total number of products `TC=100`. Additionally, fix the random number generator seed at this point.

(b) Following the settings section, create an increasing sequence of N integers indicating consumers ids. We will refer to this vector as `consumers`

(c) Create a vector of $M-1$ elements drawn independently and uniformly from $[-50, 50]$, which we will call `locs`. Additionally, draw a vector of $M-1$ independent log-normals with standard deviation equal to 5, called `stds`. Draw TC samples out of a joint-normal of $M-1$ of mean `locs` and variance matrix of diagonal equal to `stds` squared and cross-correlations equal to zero. We will refer to this matrix as the product attributes `p_attrs`. Draw a log-normal vector of mean 5 and standard deviation 3 and size TC. This is the price vector `p`. Draw a uniform binary matrix of size $N \times \lfloor (M-1)/2 \rfloor$, which we will call the consumer characteristics `c_attrs`.

(d) Using efficient indexing operations, for each consumer draw a number of offered products c uniformly from C, then draw uniformly without replacement c products out of the list of TC product indexes. Use the row draws to expand the `consumers` to match the consumers-products size; call this new variable I (note that by construction different consumers might have different number of products). Use the product draws of each consumer to select the appropriate rows out of `p_attrs` and the `p` variables (the row index drawn from `p_attrs` and `p` should match). Concatenate these two array to form the matrix X of offer attributes, with the first column being the prices. Expand the `c_attrs` to match the size of X such that each consumer's row in `c_attrs` is matched with it's offer attributes in X. Replace the last $\lfloor (M-1)/2 \rfloor$ columns of X with their Hadamard (element-wise) product with `c_attrs`. This concludes the creation of X, which now contains plenty of product- and individual-level variation.

(e) Draw a vector β from your preferred distribution. The first element of β should be negative as it correspond to the consumers' preferences for price.

(f) To construct the the choice of each consumer, create a score $S = X\beta$ and sample a product for each consumer, with a probability proportional to its score. Form a binary vector Y of choices, where each row is either a zero if a product is chosen and 0 otherwise. Each consumer should have one, and only one, choice.

(g) Now proceed to implement the functions in `logit_demand.py`. Implement them in the order shown above and test them after each execution. Train's chapter provide further guidance on how to compute the choice probabilities and log-likelihoods. Some hints:

   - The logit function includes exponentials which grow very quickly and due to the computer's limited precision, are fairly unstable. This often leads to infinities or to products being predicted to be chosen with probability 1 or 0. A workaround is to normalize each consumer's product utilities with respect to the largest one. Using indexing and broadcasting, that can be done quite efficiently. If you chose to do so, remember to also adjust the normalized outside option utility.

- The logit function requires deviding the exponentiated utility of each product by the sum of all products offered to the consumer. In a dataset where different consumers have different number of products, this can be tricky. Looping over consumer will result in a very slow code as direct loops in Python are expensive. One, of many, efficient way around this is to use a cummulative sum over the entire data, and then use the consumer index vector I to fetch the correct sum. The less challenging route is using numpy's `bincount` function.
- Note that the likelihood function depends only on the probability of the chosen products.

4 Data Management

One of the nice features of Python is that we can do both numeric optimization and data processing in a single language. Although it is likely you will resort to Stata for simple data operations, Python has a great advantage when processing complex datasets and merging multiple datasets. You will likely find cases where Stata's single-dataset orientation will result in painfully long codes and running times. The go-to data management library in Python right now is Pandas.

We recommend you cover the official Pandas tutorial. Within each section of the tutorial you will see links to more in depth discussion of subjects. We highly recommend you review at least the data structures tutorial, the basics tutorial, the merging tutorial and the group-by tutorial.
Here are some things you should look out for in Pandas and get familiar with:

1. The convention is to import pandas as `pd`, i.e `import pandas as pd`. Therefore you will often see code starting with something like `pd.concat([x, y])` without the import line.

2. Pandas has an important distinction between a DataFrame (2-dimensional) and a Series (1-dimensional). When you select a column of a DataFrame, you get a Series which has different attributes and methods. When searching online for specific functions, be careful to specify whether it is a DataFrame or a Series function.

3. One of Pandas key source of efficiency is it's ability to return *views* (which is the default for most indexing operations). This means that the object returned does not possess a new address in the memory, but instead consists of pointers to (or a view of) the original data. This implies that if you are not careful, you might alter the original data. To try this do the following:

```python
import numpy as np
import pandas as pd

X = pd.DataFrame(np.random.rand(10, 3), columns=list('abc'))
Y = X.iloc[:3, :]
Y[:] = -1
print(X)
```

You should first see a warning from Pandas and then see that X was modified. With few exceptions, if you want to modify a DataFrame do it on the original using indexing. If you instead want a copy, be explicit and use the `DataFrame.copy()` method.

4. Pandas DataFrames always have at least one row index. Indices are tricky but powerful ways to select variables. For example, suppose we have two dataframes, one containing consumers, states and income (called `df`), and the other containing state and taxes (called `tax`.) The following are two ways of computing taxed income, with the latter using indices.

```python
# First method using merge:
df = df.merge(taxes, on='state')
df['taxed_income'] = df['income'] * df['tax']

# Second method using indexing
taxes.set_index('state', drop=True, inplace=True)
df['taxed_income'] = df['income'] * taxes.reindex(df['state'])['tax'].values
```

In fact, the indexing method takes a seventh of the time as the merge code!

5. Pandas `groupby` functions is enormously powerful. If you are familiar with Stata, groupby combines both the "`bysort ...: egen/gen`" logic and the "`collapse ..., by(...)`" logic. The following example illustrates some particular uses you might have missed

```python
# Get the households median yearly spending
df['median_spending'] = df.groupby(['household', 'year'])['spending'].transform('median')
# You can even apply fancier custom function
df['normalized_spending'] = df.groupby(['household', 'year'])['spending'].apply(
        lambda x: (x - x.mean())/ x.std()
)
# Use agg to apply different operations on different columns and
# even multiple operations on a single columns
```

```
collapsed = df.groupby(['household', 'year'], as_index=False).agg(
        {'spending': ['min', 'median', 'max'], 'member': 'nunique'}
)
```

be careful though: when grouping over columns, the resulting object will have these columns as an index unless the `as_index=False` option is set.

6. Memory allocations are often very expensive when dealing with large datasets. To avoid allocating memory unnecessarily, many of Pandas arguments have an *inplace* argument. Using `inplace=True` can save a lot of computing time and memory and is highly encouraged when possible. Consider the following example

```
# == Both of these lines do the same thing ==

# This one creates a new dataframe without the columns
# and then assigns it to df
df = df.drop(['column1', 'column2'], axis=1)
# This one avoids creating a new dataframe
df.drop(['column1', 'column2', axis=1, inplace=True)
```

7. Pandas also has a nice feature for exporting to latex. Additionally, as column names can have spaces and latex code in them, you can automate the creation of tables easily. For example the following code creates the table below

```
# Create a fake data set
X = pd.DataFrame(np.random.rand(10000, 3), columns=list('abc'))
X['year'] = np.random.randint(2017, 2019, 10000)
# Create the table
tbl = X.groupby(['year']).agg(
        {'a': ['min', 'max'],
        'b': ['median', 'mean'],
        'c': lambda x: np.mean(np.log((x - x.mean())**2))}
)
# Make a nice header
tbl.columns = pd.MultiIndex.from_tuples([
        ('Column (a)', 'min($x$)'),
        ('Column (a)', 'max($x$)'),
        ('Column (b)', 'median($x$)'),
        ('Column (b)', r'$\bar{x}$'),
        ('Column (c)',  r'$\theta(x)$')])
# save to tex
tbl.to_latex('~/Downloads/tbl.tex', multicolumn=True,
        encoding='utf-8', escape=False, float_format='%.03f')
```

Resulting table:

	Column (a)		Column (b)		Column (c)
	min(x)	max(x)	median(x)	$\bar{x}$	$\theta(x)$
year					
2017	0.0	1.0	0.511	0.505	-3.367
2018	0.0	1.0	0.496	0.497	-3.391

8. Pandas DataFrame are wrappers around NumPy arrays. It is often useful to transforms dataframe to arrays to avoid the Pandas overhead, which can be done simply by accessing the *values* attributes of the DataFrame.

9. Pandas has quite a few hidden functions. One that turns out to be quite useful is one to create unique identifiers for groups (as the *egen group* function in Stata). To create unique identifier the easiest way is to do

```
df['group_id'] = df.groupby(group_cols).grouper.group_info[0]
```

10. Another useful thing in Pandas is that it can read data from many origins. For example, `pd.read_stata()` reads `.dta` files. Matlab files are more tricky and require the `scipy.io` module. Pandas also includes the ability to store and read HDF5 files which are a very powerful storage format which allows compression, querying and hierarchical structures. See the IO reference page for more details.

11. As should be clear by now, the tutorials pointed above cover only the surface of Pandas capabilities. Check the Pandas Cookbook for more tricks.

4.1 Key Exercise: processing data

In this exercise you will process a sample data set created for this exercise. You will start by following a sequential pattern of codes to clean-up and prepare the data for it's usage in the logit code.

In the resource folder for chapter 4 you will find three datasets in Stata format: *choices.dta*, *price_scales.dta* and *providers.dta*. This data corresponds to individual level data on the demand for cancer treatment from Cuesta, Noton and Vatter (2019). The data has been scrambled to protect the anonymity of individuals and providers. However, it has the *messiness* of true data. The *choices* data set contains the ids of each consumer, the year id, the insurer id of each consumer, her/his age, female gender indicator, number of dependents, and the id of the chosen provider. The *providers* data set contains a row for each provider-year together with the average price for cancer treatment as well as a public-hospital indicator. Finally, the *prices* data contains for each insurer-year-provider the fraction of the average price the insurer has negotiated to pay. You will combine these datasets to form a more useful dataset to use with the logit demand.

1. Copy the dataset to the Data folder of your coding_tutorial project.

2. Create a new folder in the Codes directory of the logit_demand project called *data_processing*. In this folder create a script called `c0\_combine.py`.

3. In `c0_combine.py` create a script that merges together the three dataset in to a single one. Create a single price for each consumer by multiplying the price scale of each insurer with the average price of each provider.

    - Clean-up the data: make sure that no consumers become younger as the years increase. Correct ages starting from the minimum if necessary. Make sure that the gender doesn't change for the patient (transgender and other gender definitions where removed from this extract for simplicity), and correct if necessary.
    - Create an automated summary statistics extract to latex: a formatted table containing the average age of patients at each hospital, the average and median payment at each hospital and the market share of each hospital. For each year, create a table containing the average number of cases, average price paid and the share of demand going to public hospitals. The outcome should be stored in the Output folder of the project.
    - Store the resulting cleaned data set using an HDF5 format, compressed at level 2.

4. Create a new script called `c1_expand.py`. In this script, use the data you just cleaned and expand the choice sets of each consumer. To do so, use the insurer of each consumer and the choices of all patients in that insurer. If a provider was never selected in a given year by any consumer in a given insurer, do not include it in the choice set of consumers in that insurer. Create a column that for each consumer-hospital-year indicates whether the age of the patient is above or below the median age of patients of the given hospital in the previous year. Drop the first year. Store the resulting panel in the same HDF5 storage as before in a different key.

5. In the *test.py* script created in the previous section create a new setting called `use_data` such that if it is set to true, the code will load the actual data and use that to test the logit module. Otherwise, it should run the simulation code created in the previous section.

6. Add to *test.py* the lines necessary to transform the expanded data into the format used by the logit code and test it.

5 Object Orientation

The same way functions allow us to avoid repetition and encapsulate sequential logic, objects allow us to encapsulate groups of functions into classes that share common purposes. This can be enormously useful for programming in research project because it creates a natural organization of codes, and allows reusing logic across projects. As an introduction to object orientation, please read chapters 40 to 44 in Python the Hard Way. We also recommend that you read the second chapter of Data Structure and Algorithms in Python by Goodrich, Tamassia and Goldwasser (found in the resources folder) which provides more depth about design patterns and method overloading.

Objects and inheritance are powerful tools that can make the difference between writing a horrible spaghetti code that will take you years to debug, or writing one that can teach you something new about what you are researching. There is a lot of creativity and ingenuity in designing code, and there is no single right way. If you search around, you will find many design patterns that are adapted to different situations, and it will be up to you to choose or make the one that fits your problem the best. Here are a few pointers we have found to be useful in the context of programming in economics:

1. Make your code more abstract than your model.

    - It is likely that your model will change as your research progresses, and you want to make your code to adapt as easily as possible. One way to do so is to use configuration files and inheritance properly.
    - For example, suppose you are working on a model of spatial competition among rideshare drivers. You might be tempted to create separate objects for full-time and part-time drivers if your model treats them differently. However, it might be significantly more robust to first create a "Driver" class which establishes all the common methods (e.g, utility function, optimal location finder) and attributes (e.g, car make, years of experience, home location). After this, if needed, you can create a Part-time class that inherits from Driver all the methods and overloads anything that requires adjustment. This way, if you change your utility function down the road, there is a single change to be made in the Driver class which will propagate to all the types of drivers.

2. Leverage Factory design patterns.

    - We are often interested in simulating agents based on data or a list of characteristics. Factory classes can be useful for creating multiple instances of agent objects based on data in an organized fashion.
    - In the rideshare example above, suppose that you have data on the car make, experience and home location of each driver. If you want to simulate the behavior

of the drivers using your code, you can load the data and iterate on the rows to create instances of the Driver class to form a collection and operate on them. A factory class would encapsulate this logic providing two important advantages: i) all scripts creating drivers from data will use the same code such that changes will have to be coded once; ii) testing the creation of drivers becomes a matter of testing a single class instead of multiple scripts.

3. Use Python's magic methods to make your code more readable and simple.

   - Magic methods prescribe the behavior of objects when used in certain ways that are not direct calls to methods or attributes. For example, when summing two objects, Python secretly calls their `__add__` method. This can turn out to be quite useful for defining lists of abstract objects or computing attributes dynamically.
   - For example, we can set the `__str__` method of the Driver class such that when we execute `print(driver)` we get useful information about the driver. A more advanced usage could be to make the Drivers factory class an iterator by setting the `__iter__`, `__len__`, `__next__` and `__getitem__` such that we can operate on a set of drivers as if they were a list.

4. Use configuration files and `__getattr__` to control global configuration of the project.

   - It is common for economic models to have global assumptions held fixed. It seems ideal to have these assumptions available throughout the code but have a single place to modify them. Even better, you might want to create multiple instances of your code with different specifications of these assumption. Having the assumptions directly coded in a parent class will accomplish the first but not the second. Configuration files makes it easy to have a single file where all global assumptions are specified. Moreover, using the magic method `__getattr__` makes it easy to add new assumption without altering the code. See this post for more details on how to setup and use configuration files. See this post for more on dynamic attributes in classes.

5.1 Key Exercise: Logit Class

In this exercise you will convert your logit demand estimation code into a class and extend it. The idea is to leverage the class as interface in which the data is shared instead of being copied over function calls. Proceed by doing the following

1. Move your `test.py` file out of the `logit_demand` folder and rename it as `main.py`. Add an empty `__init__.py` file to the `logit_demand` folder to convert it to a module. Rename `logit_demand.py` to `LogitDemand.py`. Adjust the imports in the `main.py` folder.

2. Within `LogitDemand.py` create a class called LogitDemand and set the functions your have previously coded as methods. Create an initializer that accepts a DataFrame containing an expanded panel of choices (as your cleaned data) and extract the X, I and Y matrices as NumPy arrays. Modify your `choice_probability` and `loglikelihood` method to use the data shared in the class.[3]

3. Add a method called `plot_fit(self, beta)`. This method should take a guess of the coefficient parameter β and create a plot of the distribution of the logit choice probability of each effective choice in the data. To do the plot explore the libraries matplotlib and seaborn. The plot should be automatically saved in the output folder.

4. Create a configuration file within the `code` folder next to the `main.py` script. In this configuration file define the default name for storing the fit plot together with two boolean options: `female_only` and `under_65`. Adjust your `__init__` method to read the configuration and keep only females if `female_only` is set to true, and keep only individuals under 65 if `under_65` is set to true.

5. Modify the *main.py* to load the data, simulate some β, evaluate the loglikelihood at that point and then plot the fit.

6 Data Structures and Algorithms

In this very ambitious chapter we will attempt to guide you towards learning the basics of a very large and important part of a computer science major. We will be using the book Data Structure and Algorithms in Python (DSAp henceforth) by Goodrich et al. (found in the resources folder). We will focus on only a few chapters here, but we encourage the motivated reader to dig further in the book. Importantly, because we will be jumping chapters, it is likely that you will find references to things in the book you haven't encountered so far.

6.1 Complexity Analysis

Analyzing the complexity of code is very important for many projects and something that is done in a very unsystematic manner in economics in general. Many methodological papers in economics do a meager job of providing accurate complexity analysis that can be used to judge their usability and compare them to alternatives. To learn more on how computer scientists evaluate the complexity of code, we suggest you read chapter 3 of DSAp.

6.2 Algorithms: recursion, sorting and selecting

With complexity in mind, the importance of designing proper algorithms for certain tasks becomes apparent. For example, in the analysis of dynamic games in Industrial Organiza-

[3]hint: the signature of the methods should be as `choice_probability(self, beta)`.

tion we currently face many computational challenges where the complexity of algorithms impedes us from applying the methods to interesting problems. These sorts of problems are part of the history of algorithm design and the evolution of empirical methods. Read chapter 4 of DSAp to find out more on the use of recursions in algorithms and their impact on complexity. Read chapter 12 of DSAp to learn more about sorting and searching, which are classic example of the impact that more efficient algorithms can have on usability, and how sometimes "best" is not a uniquely defined concept in algorithms.

6.3 Data structures

The choice of algorithm often depends on the structure of the data. Therefore, the way we organize data in a code can have important consequences on the complexity of the algorithms we write and the overall running time. To learn more about the importance of data structures we suggest you read chapters 5 and 8 of DSAp.

6.4 Key Exercise: Grid Search Optimization

Using your new knowledge of algorithms and data structures, you will now make the first step towards making your logit demand usable. In particular, you will implement a grid search optimizer. How you design the grid search is up to you, but the overall guidelines are:

1. Add a method to the `LogitDemand` class called `grid_estimate(self, min_beta, max_beta, points)` which takes an two vectors `min_beta` and `max_beta` which mark the highest and lower boundary of the grid, and `points` denotes the maximum number of points to test.

2. Test the code by implementing an additional method to create "true" logit demand choices given some parameter value and see whether your grid search algorithm gets close.

3. Evaluate the complexity of your code. Search online for the complexity of optimization algorithms such as the Simplex method or Newton's Method.

4. Create an new class called TimeTester that initializes LogitDemand objects given a data set. Add a method called `run_test(self, points, min_size, max_size)` which draws `points` growing linearly between `min_size` and `max_size`. These points will be the size of the grid to search over. Holding a simulation value of β fixed, for each of these points run your grid search and time the execution. Plot the size-time curve of your algorithm and compare to your assessment of your algorithm's complexity.

Some tips for the implementation

1. Limit the data to some small sub sample.

2. Start by searching in a single dimension and applying lessons from DSAp on search algorithms.
3. Experiment with the starting point in the grid.
4. For improved performance, add a method computing the analytic gradient of the loglikelihood function. Use that information to determine the direction of movement more efficiently.

7 Numerical Optimization

Despite your best efforts, it is unlikely that your grid search method is particularly efficient. Thankfully, many researchers have dedicated their careers to developing good numerical optimization algorithms. For a quick guide to using SciPy for optimization we recommend that you read these scipy lecture notes.

It is often when doing numerical optimization that we become conscious of the efficiency of our code. Knowing where to dig for extra speed is particularly important, which is what *profilers* are for. To learn more about profiling and how to optimize code read these scipy lecture notes. Additionally, Snakeviz provides a nice GUI to Python's profiler.

Finally, a common source of code inefficiency is due to poor memory management. To learn more about how the computer's memory is structured and how garbage collection works in languages that dynamically allocate memory (like Python), read chapter 15 of DSAp.

7.1 Key Exercise: Maximum Log-likelihood

In this exercise you will finally implement an efficient optimizer for the loglikelihood and obtain reliable estimates of the demand preferences. As by now you are more experienced in Python and programming, we will provide only rough guidance on the implementation and leave the rest to you.

- Add a method called `fit`. This method should be called from the `main.py` with a starting point for the optimization and return the demand estimates which maximize the likelihood of the observed choices.
- Your new method should be flexible enough to run with and without a negativity constraint on the price coefficient.
- Your new method should be flexible enough to switch between optimizer easily, and run with and without gradient information.

- Compare the running time and accuracy of different constrained and unconstrained optimizers. Take the opportunity to test tuning parameters such as step-sizes and tolerances to see their effect on results.

8 Parallel Computing

Most modern systems, and particularly research clusters, come equipped with multiple processors. However, all that we have coded here so far has been reliant on a single processor.[4] Hopefully, we could use the additional resources our computers have to improve the performance of our code. Whether we can do it or not will depend largely on the type of problem we are trying to solve.

Parallel computing is a large field on it's own which we wont attempt to cover. We will instead discuss only the very basics that are often used in our line of work. We refer the interested reader that wishes to learn more formally about these subjects to one of the many books on the subject or these lecture notes from the University of Illinois on Designing and Building Applications for Extreme Scale Systems.

Sadly, to our knowledge, there is no single good tutorial to parallel computing in Python that covers all the subjects that you should be aware of. Instead, we recommend you review all of the following:

- This tutorial discusses the very basics of the multiprocessing library.
- This one discusses the usage of queues and subprocess to tackle parallel computing problems.
- This one discusses the distinctions between concurrent and parallel programming.
- This one delves more on the differences between multithreading and multiprocessing.
- This one discusses more about the Python GIL.

Here are some things you should keep mind when thinking about parallelizing code:

- Not all code can be parallelized. In effect, many things we routinely do in economics can not be parallelized because each round of iteration depends on the previous round's results. For example, you can not parallelize the iterations of your maximum likelihood estimator because the best guess for the next value of β depends on the previous value of the loglikelihood. If you were to create multiple processes to run different iterations on different processors, each processor would have to wait for the other one to finish.

[4]At least explicitly, because NumPy will automatically parallelize some algebraic operations in systems with the capability to do so.

- Parallelizing has an overhead cost due to the creation of new instances and the communication across processes. This creates a threshold for the complexity of processes to parallelize, under which the overhead cost is higher than the benefit of parallelizing the code. Therefore, parallelizing a sequence of short tasks might results in slower rather than faster results.

- By default Python's multiprocessing is a shared-memory parallelization scheme. This means two important things: first, you can not spawn processes across multiple nodes in a cluster; second, the processes you run in parallel share memory and therefore each eats into the resources of the others. This implies that often parallelizing code will increase the memory usage of your code. This also means that if your code is memory-constrained (i.e using almost all of the RAM available), than you wont be able to parallelize it effectively. There are, however, ways around these restrictions using the multiprocessing library differently or using MPI.

- Creating and destroying parallel processes has a cost. If your code has sections that can be parallelized and sections that can not, you can use the multiprocessing Pool class to keep alive a series of parallel processes and use them only when needed. If doing so, be very careful to close the pools that you are not using before spawning new ones.

- Python's multiprocessing Pool class has a hard time parallelizing class methods. A common workaround is to declare an auxiliary function outside of the class. See this question on Stack Overflow for more details.

8.1 Key Exercise: Bootstrapped Standard Errors

In this exercise you will use parallel computing to compute standard errors via bootstrapped simulations. If you are unfamiliar with the bootstrap and its use for computing standard errors, we suggest reading this review article by Joel Horowitz.

Given that you have designed your logit estimator in object orientation, bootstrapping standard errors is very simple:

1. Add a `_bootstrap_stderr(self)` to your method which should be called by your `fit` method after convergence is achieved.[5] This class should handle the computation of standard errors via simulation in parallel.

2. Add a configuration field to your configuration file that controls the default number of bootstrap draws to use and the sample size to draw.

[5]Note that the method name starts with an underscore. This is a convention for *private* methods, which should only be called within the same class or it's subclasses, but not in the public scope. This logic comes from the Desgin by Contract approach to software design.

3. Modify the output of your `fit` method to automatically print a result table with the point estimates and standard errors.

Some tips:

1. Create an auxiliary function outside of your class but within the same script to handle the multiprocessing calls.
2. Use `Pool.map` and the `with` clause for a simple parallelization scheme.
3. Control the *chunksize* parameter of map for added performance.
4. Do not spawn more processes than processors; you will be paying the overhead without any of the benefits.

9 Commenting, Debugging, Testing and Final Thoughts

You now have a fully functional logit demand estimator. Although some things were added just for pedagogical purposes (like the configuration file), the class is mostly portable and could be used in any project that has individual-level logit demand. However, it might be a year until you have to use this code again and by then you might have completely forgotten how to use the class. In general, when programming large projects, it is likely you and your coauthors will forget what things do. That's why commenting and code styling is so important!

Thankfully, styling is not all arbitrary. The Python community has standardized rules about how your code should look which are known as the PEP8 standard. This is a particularly nice thing about Python, which means that if you dig into the source code of libraries such as Pandas or NumPy, things will look mostly familiar in organization. However, the way functions are commented (known as *docstring*s) varies quite a bit, as this post on Stack Overflows details. Whatever pattern you choose to follow, stick to it and document every method you write. A nice thing about modern coding editors is that they make it easy to stick to styling and commenting styles. Install a flake8 linter in your editor to automatically detect styling errors (vim, sublime, atom) and a automatic PEP8 corrector (sublime, atom, vim). Finally, add automatic snippet support to your editor such that function declarations will be automatically created with docstrings (atom, sublime, vim).

It is estimated that about half of the time programmers work is spent on debugging (see this article for references). Therefore, improving the way you debug code and detect errors can make significant differences in the time spent coding. One tool frequently used by programmers is *unit testing*. Unit tests are a series of checks designed to test the integrity of pieces of codes, such that you are always aware of what is working and what isn't. One case where this is particularly helpful is if you accidentally brake a functionality while working on another. Without testing, these sorts of errors can remain hidden for a long time, making finding the source of the mistake very challenging. We recommend that you read this post to learn more about unit testing in Python and complement with this short guide on testing guides and alternative tools for testing.

Finally, making the execution of your code informative can make a big difference in the time spent debugging. Using the python logging module you can setup different levels of information to be displayed depending if you are debugging a new piece of code or running one that you already know to work. Creating custom exception classes is done very easily by extending the Exception class. This is a very common approach to control the informativeness of your errors and distinguish exceptions handled by your code form those thrown by other libraries. We recommend that you code to *fail frequently and loudly*, meaning that your code should contain several clauses that test inputs, intermediary steps, and output, and throw exceptions that are abundant in information. The Python error traceback is usually highly informative and will help you locate the origin of mistakes easily.

Some final things to keep in mind regarding styling, debugging and testing:

- It is normal for codes to be more than 50% comments. It might seems like a lot of extra work, but it will save you a lot of time down the road. Make the effort to maintain docstrings as you modify functions and make your coauthors comply with styling standards.
- Write unit tests after you complete significant chunks of your code. For example, if you just finished the code that estimates your demand system and about to move towards a structural simulation, write a unit test for your demand estimation. If results start to look odd down the road, you will appreciate the certainty of knowing that your demand estimation routine hasn't been damaged by some odd mistake.
- When you have some unit tests written, you can automate your editor to run the tests every time you start working on your project. This way you will know immediately what is broken when you start working.
- Use the logging class instead of prints to provide runtime information. Limit prints to things like output table and communications with the user. Define the logging format and level in the `main.py` file, not at the class level. You can also use the logging system to automatically store a log file during execution.
- Create custom exceptions that are aware of your custom definitions. For example, if you defined a `__str__` representation of your class, make your exceptions aware of that to present useful information to users.

- When confronting a bug in an external library, do not be scared from digging into the source code. The styling uniformity makes most source code in Python relatively easy to read and it often contains more information not provided in the reference page.
- When dealing with exceptions using Pandas DataFrames, remember to print the data types of your columns, as they can often lead to conflicts.
- Finally, a lot of what you have implemented in this guide could have been pre-packaged more generically. The Statsmodels package tries to do exactly that by providing a general class of Maximum Likelihood models, which you can extend by subclassing and overloading key methods. By doing so, your classes will automatically have standard error computation and output tables. Statsmodels also provides nice linear regressions capabilities, much in the flavor of R. However, Statsmodels is still a young package and many things (like the GMM class) are still work in progress. Hopefully, some of you will contribute to make this package better by fixing bugs and suggesting new methods (github makes contributing to these sorts of projects quite easy).

9.1 Key Exercise: Unit Testing The Logit Class

In this final exercise you will make your logit demand portable and create unit tests to validate the integrity of your code.

1. Remove your configuration file and turn it's elements to optional arguments of your methods. Configuration files are very useful for many purposes but not for portable tools like an estimator. Adjust your `main.py` file to account for this new setup.
2. Add docstrings to all your methods detailing their objective and their arguments.
3. Create a unit test for your logit class. Test that your class detects invalid dataframes containing anything but a single choice for a consumers (or choice-event). Test that your class detects non-numeric, missing or infinities in the estimation data. Store a simple "true logit" case and test that your class recovers parameters reasonably close to the truth (make this as simple and fast as possible). Test that your bootstrap method is not creating more processes than cpus.
4. Add a testing script next to your `main.py` script, that runs all the tests you wrote. Add a documentation script for your library (there are some automatic generators for this if you commented your class properly).