Notes on using Python

John Gardner

Table of contents

Intro . . . 1
Preliminaries . . . 2
- Installing and interacting with Python . . . 2
- Getting Positron/VSCode up and running . . . 2
- Packages . . . 2

Language basics . . . 3
- Data types . . . 3
- Functions . . . 5
- Loops and conditionals . . . 6

NumPy . . . 7
Pandas . . . 10
- Descriptive statistics . . . 11
- Subsetting . . . 15
- Creating and recoding variables . . . 16
- Merging and reshaping data . . . 18

Data visualization . . . 20
- Matplotlib . . . 20
- Seaborn . . . 26

Numba . . . 29
Statistics and econometrics . . . 30
Basic programming examples . . . 35

Intro

This is a quick introduction to Python. Since Python is a full programming language with many scientific computing libraries, a short introduction can only scratch the surface. These notes will help you get started, and you can use the web and the references below for even more.

Some nice references are:

- The QuantEcon tutorials
- Python for Data Analysis
- The documentation for individual packages, which usually include a tutorial in addition to more detailed references (I have linked to the webpage for each package referenced below)

Preliminaries

Installing and interacting with Python

There are many ways to install and run Python. You can install Python using the official installer or the Anaconda distribution. You can run Python directly from the terminal (macOS) or command line (Windows), using an IDE like VSCode or Positron, using JupyterLab Notebooks in a browser-based interface, or online using services like Google Collab.

Getting Positron/VSCode up and running

I'll briefly describe how to get set up with the official installer in Positron (the process for VSCode is similar, since Positron is based on VSCode). Download and install both. Now you can create or open either a Python (`.py`) or Jupyter Notebook (`.ipynb`) file. When you do this, you'll need to select a Python interpreter from the menu in the upper right corner, just above the document, or by typing `cmd+shift+p` (`ctrl+shift+p` on Windows) to open the command palette, then typing to search for `Python: Select Interpreter`. My system also has another version of Python installed, which I don't want to use or modify, so I select the version that says Python 3.15.0 (saved in `/usr/local/bin/python3`), which came from the official installer. It's also possible to set up *virtual environments*, which are more advanced, but useful for working with projects that may have conflicting dependencies.

To work with Jupyter Notebook file, click `+Code` or `+Markdown` to add a code or a text cell. To work with a Python script, you can run a line or selection by hitting `cmd+enter` (`ctrl+enter`), and the entire script by clicking on the arrow in the upper left of the document window (in VSCode, you will need to type `Jupyter: Create Interactive Window` first, and the shortcut for running a line is `shift+enter`).

Packages

Packages add functionality to Python, and can be installed using the package manager `pip`. You can install a package by clicking on the *Terminal* tab in the bottom window, then typing `path/python3 -m pip install packagename`, replacing `path` with the path to your Python

distribution (mine is in `/usr/local/bin/`) and `packagename` with the name of your package (you can skip the `path` part if Python has been added to your path, but you might want to use it just to be safe in case you have multiple Python installations). In VSCode, the process is the same, but you may need to go to `Terminal > New terminal` in the menu (or using the command palette) if it isn't already open.

Alternatively, once you have a Python environment running, you can use `%pip install packagename` in a Jupyter Notebook code cell, or `pip install packagename` in a Python script:

```
%pip install scipy
```

```
Requirement already satisfied: scipy in /Library/Frameworks/Python.framework/Versions/3.15/li
packages (1.18.1)
Requirement already satisfied: numpy<2.8,>=2.0.0 in /Library/Frameworks/Python.framework/Vers
packages (from scipy) (2.5.4)
Note: you may need to restart the kernel to use updated packages.
```

Once a package is installed, you can use `import` to load it. For example, we can import the `numpy` package using

```
import numpy as np
```

Language basics

Data types

Python is an object-oriented language, which means that objects are associated with *attributes* and *methods*. You can create an *integer* or *string*, and check their *type* using (note that the `print` statement is necessary in a notebook but not an interactive script):

```
# define objects
a = 1
b = 'foo'

# print them
print(type(a))
print(type(b))
```

```
<class 'int'>
<class 'str'>
```

Note that we use ' (or ") to enclose strings.

You can use `del` to delete an object:

```
del b
```

Anything after `#` is a comment. It's a good idea to use comments to document and structure your code.

Lists are another useful type of object. They can contain numbers, strings, or a mix of the two:

```
c = [1, 2, 3]
d = ['foo', 'bar', 'baz', 4]
print(d)
print(type(d))
```

```
['foo', 'bar', 'baz', 4]
<class 'list'>
```

You can use *slice* notation to access elements of lists:

```
print(d[0])
print(d[1:3])
print(d[3:])
print(d[-1])
```

```
foo
['bar', 'baz']
[4]
4
```

Python starts counting at zero, so `d[0]` is the first element. `d[1:3]` gives the sublist starting at the second element (element 1…) and ending *before* the fourth element (element 3). `d[-1]` returns the last element.

List comprehensions give us a way to automatically generate lists (similar to the slice notation, the `range` function gives us numbers stopping *before* the specified point):

```
e = [i**2 for i in range(4)]
print(e)
```

```
[0, 1, 4, 9]
```

Tuples are similar to lists, but they are specified with parentheses instead of brackets, and they are *immutable*, which means that their elements can't be modified after they are assigned.

One thing to watch out for with *mutable* objects like lists is that changes to their elements propagate to objects that are created from them. If we want these new objects to be independent, we have to use `.copy()`, which is our first example of a *method* attached to an object:

```
f = e
g = e.copy()
print(f)
e[0] = 10
print(f)
print(g)
```

```
[0, 1, 4, 9]
[10, 1, 4, 9]
[0, 1, 4, 9]
```

You can get help on a function by typing `help(function)` or `function?`. You can see all of the attributes and methods associated with an object by typing `dir(object)`, and you can get help on attributes/methods using `object.method?`.

Functions

Functions allow you create outputs from inputs. Here are two simple examples:

```
def fct(x):
    return x**2 - 16
print(fct(5))

def fcts(x, y):
    '''
    This function returns x^2 - 16 and x^2 - 8
    '''
    f1 = x**2 - 16
    f2 = x**2 - 8
    return f1, f2

h, i = fcts(5, 6)
```

```
print(h, i)
j = fcts(5, 6)
print(type(j))
```

```
9
9 17
<class 'tuple'>
```

Notes:

- Python uses indentation (rather than brackets or semicolons) to structure code
- Python uses ** rather than ^ for exponentiation
- A function can optionally include a description using '''
- If we assign the output of a function with multiple returns to a single object, the result is a tuple

Loops and conditionals

Loops are useful for doing things iteratively. Here are some basic examples:

```
for i in c:
    print(i)

sum = 0
for i in range(len(c)):
    sum += c[i]
print(sum)
```

```
1
2
3
6
```

The syntax `sum += c[i]` is shorthand for `sum = sum + c[i]`.

Two additional looping patterns are `zip` and `enumerate`:

```
for i, j in zip(c, d):
    print(f"i={i}, j={j}")
```

```
i=1, j=foo
i=2, j=bar
i=3, j=baz
```

```
for i, j in enumerate(d):
    print(f"i={i}, j={j}")
```

```
i=0, j=foo
i=1, j=bar
i=2, j=baz
i=3, j=4
```

We can also use *while loops* to iterate while a condition holds, and `if`, `else` and `elseif` to check whether conditions hold. The following illustrates this by implementing the bisection method to find the positive root of $x^2 - 16$:

```
def bisect(f):
    eps, low, high = [1, 0, 5]
    i = 1
    while eps>(10**(-9)):
        mid = .5*(low + high)
        eps = abs(f(mid))
        if f(low)* f(mid) < 0:
            high = mid
        else:
            low = mid
        i += 1
    return mid, i

result = bisect(fct)
print(result)
```

```
(3.9999999999417923, 35)
```

NumPy

The NumPy package provides numerical computing functionality, including linear algebra.

Matrices and vectors are types of *arrays*:

```
import numpy as np

x = np.array([1, 2, 3])
print(x)
print(x.shape)
y = np.array([[1, 2, 3],
              [2, 1, 3],
              [1, 2, 5]])
print(y)
print(y.shape)
```

```
[1 2 3]
(3,)
[[1 2 3]
 [2 1 3]
 [1 2 5]]
(3, 3)
```

`x` is a flat array (it only has one dimension), while `y` is a 3x3 matrix. `shape` is an example of an *attribute*. If we want, we can turn `x` into a column or row vector using `x.reshape(3, 1)` or `x.reshape(1, 3)`.

You can create an n-vector of zeros using `np.zeros(n)` and an $n \times k$ matrix of zeros using `np.zeros([n, k])`. Replacing `zeros` with `ones` creates a vector/matrix of ones. You can create an an n-dimensional identity matrix using `np.eye(n)`.

We can use `@` for matrix multiplication and `+` for matrix addition. Python intelligently treats `x` as a row vector when we premultiply `y` by `x`, but a column vector when we postmultiply `y` by `x`:

```
print(x @ y)
print(y @ x)
```

```
[ 8 10 24]
[14 13 20]
```

You can also slice arrays. For example:

```
print(y[:, 0])
print(y[2, 0:2])
```

```
[1 2 1]
[1 2]
```

Other useful matrix algebra tools are the `.T` method for transpose and `np.linalg.inv` for inversion:

```
print(y.T)
print(np.linalg.inv(y))
```

```
[[1 2 1]
 [2 1 2]
 [3 3 5]]
[[ 0.16666667  0.66666667 -0.5       ]
 [ 1.16666667 -0.33333333 -0.5       ]
 [-0.5         0.          0.5       ]]
```

Two useful functions for array creation are `np.linspace`, which creates a specified *number* of points, and `np.arange`, which creates points at a specified *distance*:

```
z1 = np.linspace(0, 1, 10)
print(z1)
z2 = np.arange(0, 1, .2)
print(z2)
```

```
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
[0.  0.2 0.4 0.6 0.8]
```

NumPy is also useful for generating random numbers. First, we have to choose a random number generator (we'll use the default) and seed it (to ensure reproducible results):

```
rng = np.random.default_rng(12345)

z = rng.normal(0, 1, 10)
print(z)
```

```
[-1.42382504  1.26372846 -0.87066174 -0.25917323 -0.07534331 -0.74088465
 -1.3677927   0.6488928   0.36105811 -1.95286306]
```

Pandas

Pandas is a library for working with rectangular data (you may also be interested in Polars if you work with very large datasets).

We can create an example *DataFrame* using

```
import pandas as pd

data = pd.DataFrame()

for i in range(1,3):
    data[f"y{i}"] = rng.normal(0, 1, 100)
    data[f"x{i}"] = rng.normal(0, 1, 100)
    data[f"z{i}"] = rng.choice([1, 2, 3], 100)

print(data.head())
```

```
         y1        x1  z1        y2        x2  z2
0  2.347410  1.848289   3  2.471313  1.170736   2
1  0.968497  1.899953   2 -1.019885 -0.937843   2
2 -0.759387 -0.098425   2  0.018753  0.343267   3
3  0.902198  0.813445   2 -1.894264  0.050898   2
4 -0.466953  0.392494   3 -0.755002  0.517560   3
```

Notes:

- Here I used `range(1, 3)` to ensure that the variable names start at 1
- The syntax `f"y{i}"` is an example of *f-strings*, which are useful for looping

You can create a DataFrame from a csv or Stata file using

```
mydataframe = read_csv("path/file.csv")
mydataframe = read_stata("path/file.dta")
```

and write a DataFrame to csv using

```
data.to_csv('filename.csv', index=False)
```

As far as DataFrame hygiene goes, you can use `rename` to change column names and `sort_values` to re-order rows. Below, `columns` gives the column names of a DataFrame, and we create a test DataFrame using Python's *dictionary* syntax:

```python
# rename y1 to var1
data_newnames = data.rename(columns={'y1': 'var1'})

# make all variables names upper case
data_newnames = data.rename(str.upper, axis='columns')
print(data_newnames.columns)

test_data = pd.DataFrame(
    {'x': [1, 2, 2, 2, 1],
     'y': [5, 2, 3, 4, 1]}
)
test_data = test_data.sort_values(['x', 'y'])
print(test_data)
```

```
Index(['Y1', 'X1', 'Z1', 'Y2', 'X2', 'Z2'], dtype='str')
   x  y
4  1  1
0  1  5
1  2  2
2  2  3
3  2  4
```

Descriptive statistics

You can get basic descriptive statistics using

```python
print(data.describe())
```

```
                y1          x1          z1          y2          x2          z2
count   100.000000  100.000000  100.000000  100.000000  100.000000  100.000000
mean     -0.014297    0.074054    2.100000   -0.074694   -0.023050    2.040000
std       0.936453    0.975812    0.771984    1.032687    1.003658    0.803025
min      -1.995857   -2.722416    1.000000   -3.119609   -2.269948    1.000000
25%      -0.751330   -0.599598    1.750000   -0.759370   -0.866335    1.000000
50%      -0.029952    0.177323    2.000000   -0.114577    0.012604    2.000000
75%       0.626376    0.761904    3.000000    0.639594    0.677688    3.000000
max       2.618159    2.150310    3.000000    2.471313    2.402473    3.000000
```

Alternatively, you can get custom statistics, or describe individual variables:

```
print(data.mean())
print(data['y1'].describe())
```

```
y1   -0.014297
x1    0.074054
z1    2.100000
y2   -0.074694
x2   -0.023050
z2    2.040000
dtype: float64
count    100.000000
mean      -0.014297
std        0.936453
min       -1.995857
25%       -0.751330
50%       -0.029952
75%        0.626376
max        2.618159
Name: y1, dtype: float64
```

`.groupby` allows you to do things within groups:

```
print(data.groupby('z1').mean())
print(data.groupby(['z1', 'z2']).mean())
```

```
          y1        x1        y2        x2        z2
z1
1   0.026668  0.041968 -0.026110  0.124452  2.120000
2   0.118519  0.009419 -0.065162  0.013387  2.050000
3  -0.195348  0.170842 -0.120291 -0.170049  1.971429
             y1        x1        y2        x2
z1 z2
1  1   0.018018  0.079362 -0.152361 -0.424251
   2   0.145449 -0.171271 -0.473805  0.068818
   3  -0.062301  0.186383  0.420421  0.553051
2  1   0.213321 -0.134062 -0.347297 -0.105567
   2   0.416014 -0.036246 -0.089101 -0.351128
   3  -0.260234  0.178068  0.200606  0.479861
3  1  -0.243168  0.077125 -0.337453 -0.285162
   2  -0.020378  0.215688  0.254339  0.047078
   3  -0.387702  0.211145 -0.405895 -0.347404
```

You can use `.agg` to get multiple summary statistics at once:

```
summary = data.groupby('z1')[['y1', 'y2']].agg(['std', 'count'])
print(summary)
```

```
          y1              y2
         std count       std count
z1
1   0.914602    25  1.204293    25
2   0.911319    40  0.984362    40
3   0.977048    35  0.982866    35
```

This results in a DataFrame with *MultiIndex* columns (the columns are indexed by variable name, then statistic type). We can *flatten* the columns using:

```
print(summary.columns)
summary.columns = [f"{a}_{b}" for a, b in summary.columns]
print(summary.head())
```

```
MultiIndex([('y1',   'std'),
            ('y1', 'count'),
            ('y2',   'std'),
            ('y2', 'count')],
           )
      y1_std  y1_count    y2_std  y2_count
z1
1   0.914602        25  1.204293        25
2   0.911319        40  0.984362        40
3   0.977048        35  0.982866        35
```

`.transform` allows you to add summary statistics to a DataFrame (you could combine this with a function to perform even more complex transformations)

```
data2 = data.copy()
data2[['y1_mean', 'y2_mean']] = data.groupby('z1')[['y1', 'y2']].transform('mean')
print(data2.head())
```

```
         y1        x1  z1        y2        x2  z2   y1_mean   y2_mean
0  2.347410  1.848289   3  2.471313  1.170736   2 -0.195348 -0.120291
1  0.968497  1.899953   2 -1.019885 -0.937843   2  0.118519 -0.065162
```

```
2 -0.759387 -0.098425   2  0.018753  0.343267   3  0.118519 -0.065162
3  0.902198  0.813445   2 -1.894264  0.050898   2  0.118519 -0.065162
4 -0.466953  0.392494   3 -0.755002  0.517560   3 -0.195348 -0.120291
```

You could use a loop to do this for many variables:

```
for i in range(1, 3):
    data2[f"x{i}_mean"] = data.groupby('z1')[f"x{i}"].transform('mean')
print(data2.head())
```

```
         y1        x1  z1        y2        x2  z2   y1_mean   y2_mean  \
0  2.347410  1.848289   3  2.471313  1.170736   2 -0.195348 -0.120291
1  0.968497  1.899953   2 -1.019885 -0.937843   2  0.118519 -0.065162
2 -0.759387 -0.098425   2  0.018753  0.343267   3  0.118519 -0.065162
3  0.902198  0.813445   2 -1.894264  0.050898   2  0.118519 -0.065162
4 -0.466953  0.392494   3 -0.755002  0.517560   3 -0.195348 -0.120291

    x1_mean   x2_mean
0  0.170842 -0.170049
1  0.009419  0.013387
2  0.009419  0.013387
3  0.009419  0.013387
4  0.170842 -0.170049
```

You can use `.value_counts()` to get distributions of discrete variables, and `crosstabs` for frequency tables. To get proportions, we can use `normalize='index'` for column percentages and `normalize='columns'` for row percentages:

```
print(data['z1'].value_counts())
print(pd.crosstab(data['z1'], data['z2'], margins=True, margins_name='Total'))
print(pd.crosstab(data['z1'], data['z2'], margins=True, normalize='index'))
```

```
z1
2    40
3    35
1    25
Name: count, dtype: int64
z2      1   2   3  Total
z1
1       7   8  10     25
2      12  14  14     40
```

```
3      11  14  10     35
Total  30  36  34    100
z2          1     2         3
z1
1    0.280000  0.32  0.400000
2    0.300000  0.35  0.350000
3    0.314286  0.40  0.285714
All  0.300000  0.36  0.340000
```

Subsetting

You can subset to specific columns using syntax like

```
print(data[['y1', 'y2']].head())
```

```
         y1        y2
0  2.347410  2.471313
1  0.968497 -1.019885
2 -0.759387  0.018753
3  0.902198 -1.894264
4 -0.466953 -0.755002
```

All of the following allow you to choose rows according to conditions:

```
data.loc[data['y1']<0].head()
data.loc[data['z1'].isin([1, 2])].head()
data.query('y1 < 0 and z1 in [1,2]')[['y1', 'z1']].head()
print(data.query('y1<0 and z1 in [1,2]')[['y1', 'z1']].head())
```

```
          y1  z1
2  -0.759387   2
5  -0.060690   1
11 -0.299699   2
13 -1.621583   2
17 -0.081688   2
```

You can use `.index` to select rows by number and columns by name:

```
print(data.loc[data.index[5:10], ['y1', 'y2']].head())
```

```
         y1        y2
5 -0.060690  0.756198
6  0.788844 -1.042462
7 -1.256668 -0.034258
8  0.575858 -0.355168
9  1.398979 -0.378428
```

Creating and recoding variables

It's straightforward to create new variables. For example,

```
# manual generation
data['x_bin'] = np.nan
data.loc[data['x1']>0, 'x_bin'] = 1
data.loc[data['x1']<=0, 'x_bin'] = 0
print(data.head())
```

```
         y1        x1  z1        y2        x2  z2  x_bin
0  2.347410  1.848289   3  2.471313  1.170736   2    1.0
1  0.968497  1.899953   2 -1.019885 -0.937843   2    1.0
2 -0.759387 -0.098425   2  0.018753  0.343267   3    0.0
3  0.902198  0.813445   2 -1.894264  0.050898   2    1.0
4 -0.466953  0.392494   3 -0.755002  0.517560   3    1.0
```

`np.nan` is how to create missing data, stored as `NaN`. We can use `replace` to recode values (also see `map`):

```
print(data['z1'].value_counts())
data3 = data.copy()
data3['z1'] = data3['z1'].replace([1, 2, 3], [0, 0, 1])
print(data3['z1'].value_counts())
```

```
z1
2    40
3    35
1    25
Name: count, dtype: int64
```

```
z1
0    65
1    35
Name: count, dtype: int64
```

We can use `apply` (or `transform`) to change multiple variables without using a loop:

```
data3[['y1', 'y2']] = data3[['y1', 'y2']].apply(lambda x: np.log(x))
print(data3.head())
```

```
         y1        x1  z1        y2        x2  z2  x_bin
0  0.853312  1.848289   1  0.904750  1.170736   2    1.0
1 -0.032010  1.899953   0       NaN -0.937843   2    1.0
2       NaN -0.098425   0 -3.976422  0.343267   3    0.0
3 -0.102921  0.813445   0       NaN  0.050898   2    1.0
4       NaN  0.392494   1       NaN  0.517560   3    1.0
```

```
/Library/Frameworks/Python.framework/Versions/3.15/lib/python3.15/site-
packages/pandas/core/arraylike.py:402: RuntimeWarning: invalid value encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

The syntax `lambda x: np.log(x)` is a *lambda function*, which allows you to define a function inline (without using `def`). Note that taking logs produces missing values.

We can combine this with a string trick to add new variables by transforming existing ones, although a loop might be more intuitive:

```
vars = data3.columns[data3.columns.str.startswith('x')]
data3 = data3[vars + '_sq'] = data3[vars].apply(lambda x: x**2)
print(data3.head())
```

```
         x1        x2  x_bin     x1_sq     x2_sq  x_bin_sq
0  3.416172  1.370623    1.0  3.416172  1.370623       1.0
1  3.609821  0.879549    1.0  3.609821  0.879549       1.0
2  0.009687  0.117832    0.0  0.009687  0.117832       0.0
3  0.661693  0.002591    1.0  0.661693  0.002591       1.0
4  0.154052  0.267868    1.0  0.154052  0.267868       1.0
```

You can also use `apply` across columns:

```
data3['x_rowmax'] = data3[['x1', 'x2']].apply('max', axis=1)
print(data3.head())
```

```
         x1        x2  x_bin     x1_sq     x2_sq  x_bin_sq  x_rowmax
0  3.416172  1.370623    1.0  3.416172  1.370623       1.0  3.416172
1  3.609821  0.879549    1.0  3.609821  0.879549       1.0  3.609821
2  0.009687  0.117832    0.0  0.009687  0.117832       0.0  0.117832
3  0.661693  0.002591    1.0  0.661693  0.002591       1.0  0.661693
4  0.154052  0.267868    1.0  0.154052  0.267868       1.0  0.267868
```

Merging and reshaping data

We can use `merge` to combine two DataFrames. For example, instead of using `transform` to add summary statistics, we could merge the statistics we created using `agg`:

```
data3 = pd.merge(data, summary, on='z1', how='left')
print(data3.head())
```

```
         y1        x1  z1        y2        x2  z2  x_bin    y1_std  y1_count  \
0  2.347410  1.848289   3  2.471313  1.170736   2    1.0  0.977048        35
1  0.968497  1.899953   2 -1.019885 -0.937843   2    1.0  0.911319        40
2 -0.759387 -0.098425   2  0.018753  0.343267   3    0.0  0.911319        40
3  0.902198  0.813445   2 -1.894264  0.050898   2    1.0  0.911319        40
4 -0.466953  0.392494   3 -0.755002  0.517560   3    1.0  0.977048        35

     y2_std  y2_count
0  0.982866        35
1  0.984362        40
2  0.984362        40
3  0.984362        40
4  0.982866        35
```

`how='left'` performs a left merge (keep all observations from the first dataset, drop observations from the second that don't match to the first). There is also `right` (keep all observations from the second dataset), `inner` (only keep observations in both), and `outer` (keep all).

Suppose `x1` and `x2` represent the same variable at different points in time. To go from a *wide* form (time periods in columns) to a *long* form (time periods in rows), we can use:

```
# copy the data so we don't mess it up
data4 = data.copy()

# add the row index as an "id" variable
data4 = data4.reset_index()
print(data4.head())

# reshape
data_long = pd.wide_to_long(data4, ['x', 'y', 'z'], i='index', j='time')
data_long
print(data_long.head())
```

```
   index        y1        x1  z1        y2        x2  z2  x_bin
0      0  2.347410  1.848289   3  2.471313  1.170736   2    1.0
1      1  0.968497  1.899953   2 -1.019885 -0.937843   2    1.0
2      2 -0.759387 -0.098425   2  0.018753  0.343267   3    0.0
3      3  0.902198  0.813445   2 -1.894264  0.050898   2    1.0
4      4 -0.466953  0.392494   3 -0.755002  0.517560   3    1.0
            x_bin         x         y  z
index time
0     1       1.0  1.848289  2.347410  3
1     1       1.0  1.899953  0.968497  2
2     1       0.0 -0.098425 -0.759387  2
3     1       1.0  0.813445  0.902198  2
4     1       1.0  0.392494 -0.466953  3
```

To reshape back to wide form, we can use:

```
data_long = data_long.reset_index()
print(data_long.head())
data_wide = data_long.pivot(index='index', columns='time', values=['x', 'y', 'z'])
print(data_wide.head())
```

```
   index  time  x_bin         x         y  z
0      0     1    1.0  1.848289  2.347410  3
1      1     1    1.0  1.899953  0.968497  2
2      2     1    0.0 -0.098425 -0.759387  2
3      3     1    1.0  0.813445  0.902198  2
4      4     1    1.0  0.392494 -0.466953  3
              x                   y              z
time          1         2         1         2    1    2
```

```
index
0      1.848289  1.170736  2.347410  2.471313  3.0  2.0
1      1.899953 -0.937843  0.968497 -1.019885  2.0  2.0
2     -0.098425  0.343267 -0.759387  0.018753  2.0  3.0
3      0.813445  0.050898  0.902198 -1.894264  2.0  2.0
4      0.392494  0.517560 -0.466953 -0.755002  3.0  3.0
```

We could use the method discussed previously to flatten the column indices if we wanted, or we can just work with the hierarchical indices. We can also accomplish the above using `unstack` (or we could go back again using the inverse, `stack`), as long as we keep the indices set:

```
data_long = data_long.set_index(['index', 'time'])
print(data_long.unstack('time').head())
```

```
      x_bin              x                   y            z
time      1    2         1         2         1         2  1  2
index
0       1.0  1.0  1.848289  1.170736  2.347410  2.471313  3  2
1       1.0  1.0  1.899953 -0.937843  0.968497 -1.019885  2  2
2       0.0  0.0 -0.098425  0.343267 -0.759387  0.018753  2  3
3       1.0  1.0  0.813445  0.050898  0.902198 -1.894264  2  2
4       1.0  1.0  0.392494  0.517560 -0.466953 -0.755002  3  3
```

Data visualization

Matplotlib

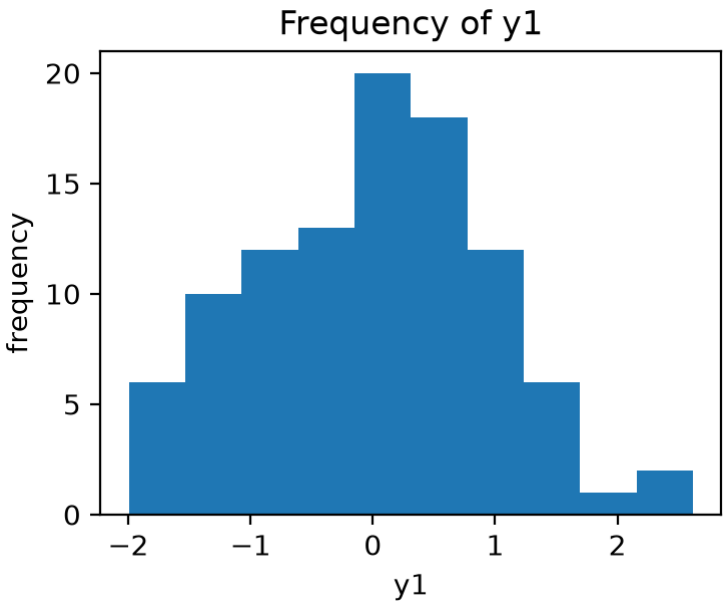
Matplotlib is Python's basic plotting package.

It's easy to make simple plots:

```
import matplotlib.pyplot as plt

# use default theme/make figures smaller
plt.style.use('default')
plt.rcParams['figure.figsize'] = (4, 3)

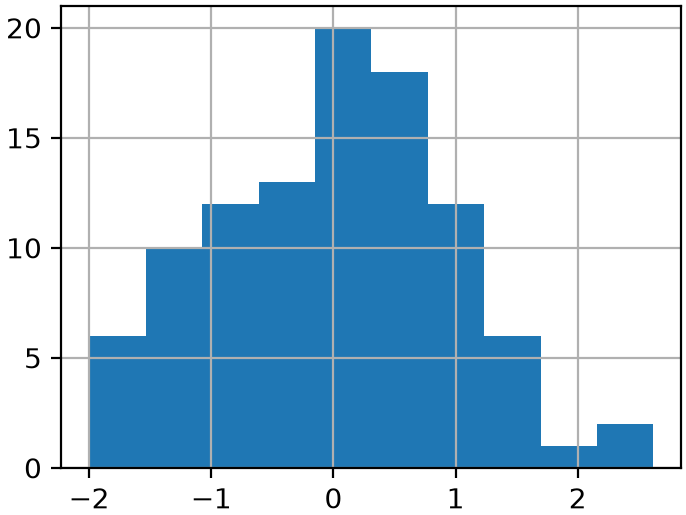
plt.hist(data["y1"])
plt.xlabel('y1')
plt.ylabel('frequency')
plt.title('Frequency of y1')
```



In a Python script, we can use `plt.figure()` to open a new figure, clearing the existing one.

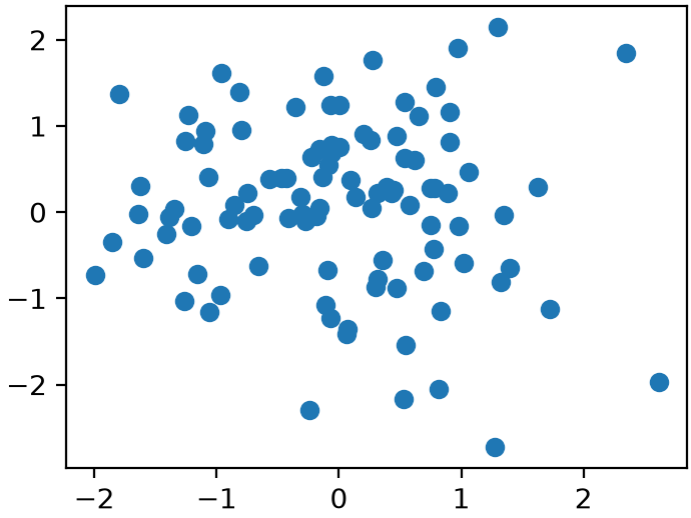
Matplotlib integrates with pandas, so we can also just type:

```
data['y1'].hist()
```



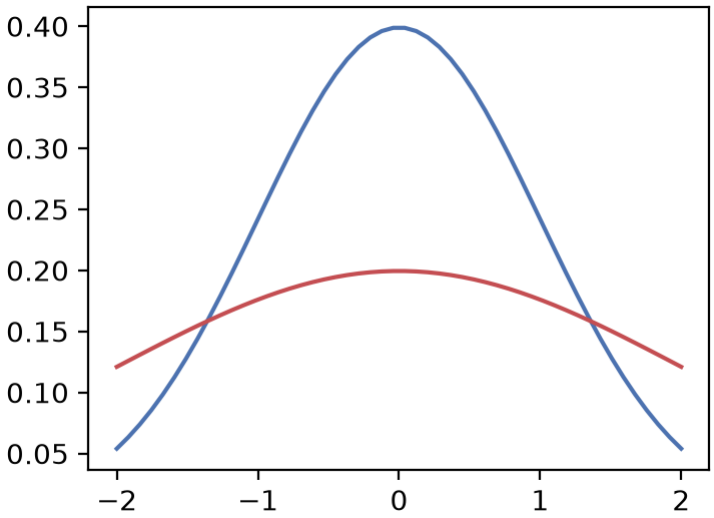
To create a scatter plot, we can use

```
plt.scatter(data['y1'], data['x1'])
```



We can also create line plots. For example, to plot a normal cdf, we can use `norm` from the SciPy package, which provides scientific computing tools for statistics, optimization, numerical integration, and more:

```
from scipy.stats import norm
x = np.linspace(-2, 2, 50)
y = norm.pdf(x, 0, 1)
y2 = norm.pdf(x, 0, 2)
plt.plot(x, y, 'b')
plt.plot(x, y2, 'r-')
```



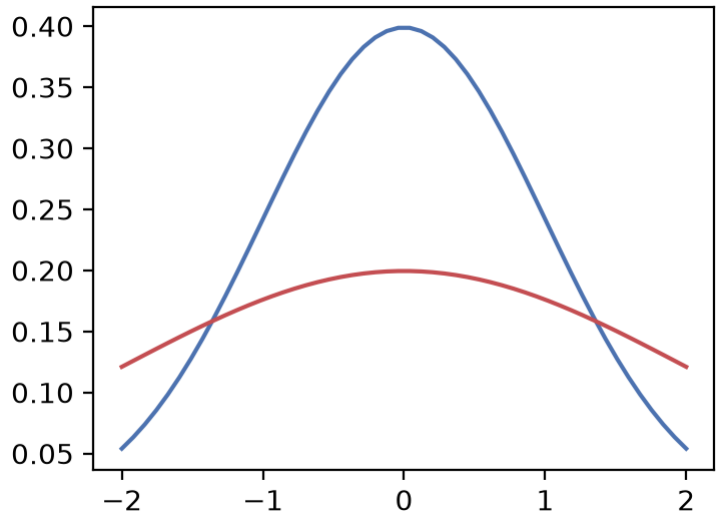
Note that additional plots are added to the existing figure (if we want a new plot, we can use `plt.clf()` or `plt.figure()`).

You can save a plot using (the type of file is inferred from the extension)

```
plt.savefig('name.pdf')
```

Matplotlib also has an *object-oriented API* which is useful for more complex graphs. We could recreate the above using

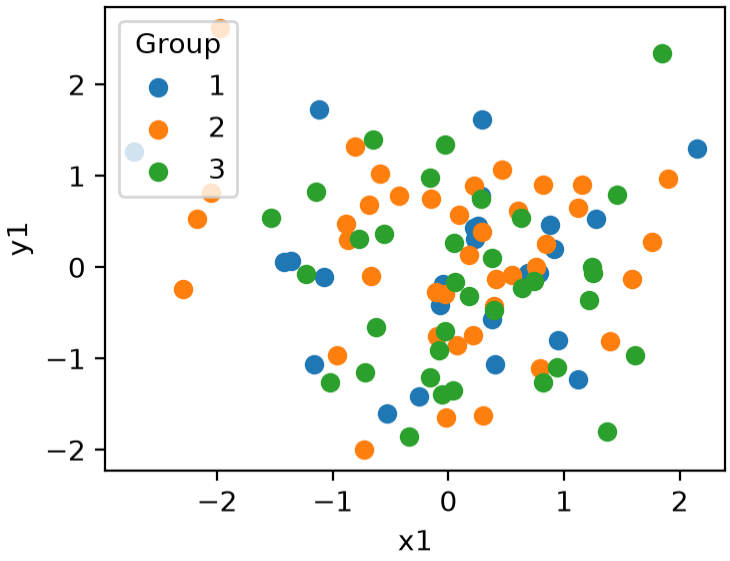
```
fig, ax = plt.subplots()
ax.plot(x, y, 'b-')
ax.plot(x, y2, 'r-')
```



`fig, ax` sets up a *figure* (the figure itself) and an *axes* (the area where the plot goes). Then each `ax.plot` statement draws a picture on the axes. "axes" isn't a typo, it's matplotlib's terminology for the collection of objects that make up a plot.

To make a scatter plot with points colored by a group variable, we can use the following (the loop works because `data.groupby` provides a group name and the associated data):

```
fig, ax = plt.subplots()
for gp_name, gp_data in data.groupby('z1'):
    ax.scatter(
        gp_data['x1'],
        gp_data['y1'],
        label = gp_name,
    )
    ax.set_xlabel('x1')
    ax.set_ylabel('y1')
ax.legend(title='Group', loc='upper left')
```

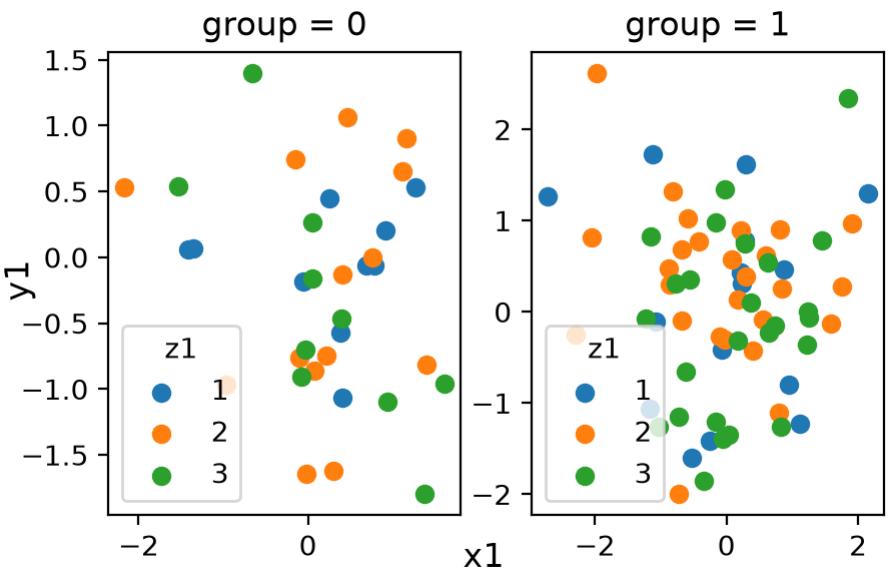


For multiple plots side by side (or in a grid), we can use

```
# binary group indicator
data['gp'] = 1*(data['z2']<3)

# gen list of groups and number of groups
gps = sorted(data['gp'].unique())
num_gps = len(gps)

# iterate axes across groups and colors across z1 values
fig, axes = plt.subplots(1, num_gps, figsize=(5, 3))
for gp in gps:
    for z1_gp_name, z1_gp_data in data[data['gp']==gp].groupby('z1'):
        axes[gp].scatter(z1_gp_data['x1'],
                         z1_gp_data['y1'],
                         label = z1_gp_name,
        )
        axes[gp].set_title(f"group = {gp}")
        axes[gp].legend(title='z1', loc='lower left')
fig.supxlabel('x1')
fig.supylabel('y1')
```



Seaborn

Seaborn is another plotting package, built on top of matplotlib, which makes it easier to produce certain more complicated graphs.

For example, seaborn makes it easy to get a scatter plot with points colored by group, a scatter plot with a regression line, or side-by-side plots for different groups:

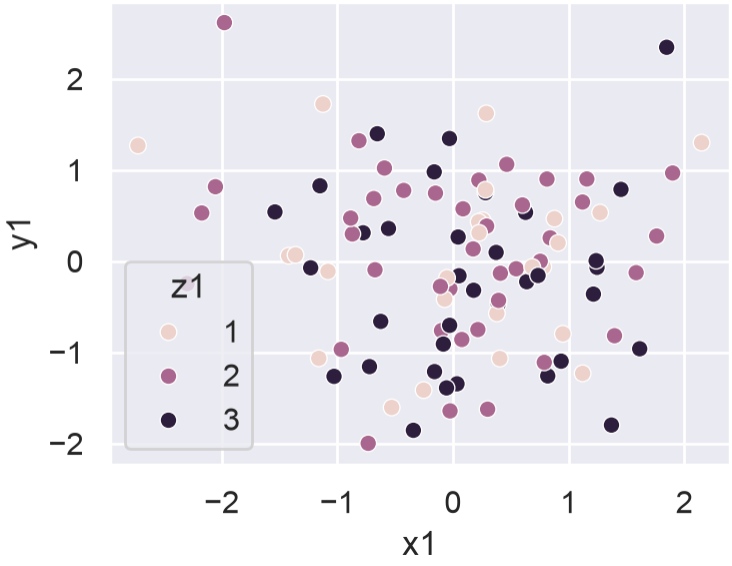
```
import seaborn as sns

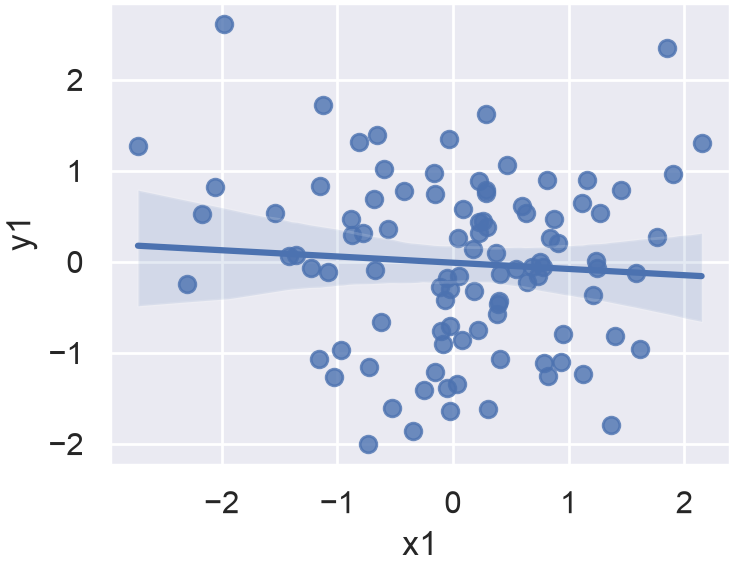
# use the default theme (instead of inheriting it from the previous graph)
sns.set_theme()

sns.scatterplot(data = data, x="x1", y="y1", hue="z1", )

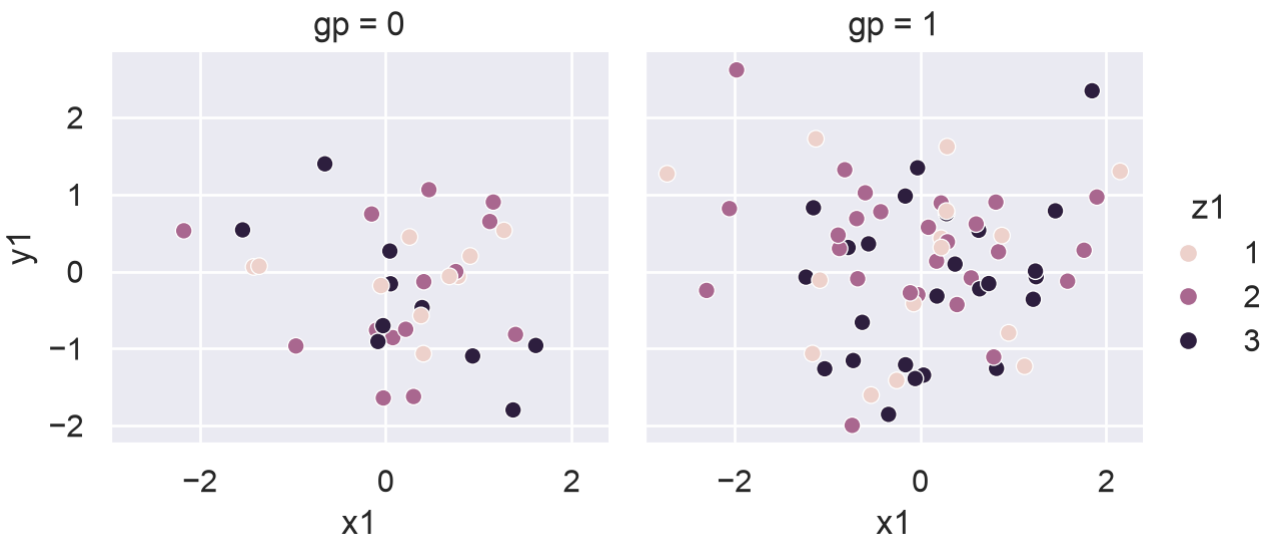
plt.figure()
sns.regplot(data, x='x1', y='y1')

plt.figure()
sns.relplot(data=data, x="x1", y="y1", hue="z1", col="gp", height=3)
```





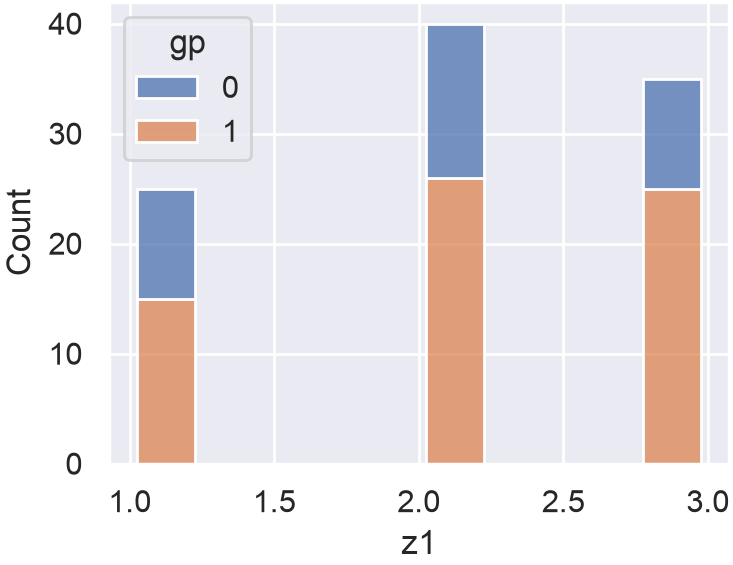

```
<Figure size 800x600 with 0 Axes>
```

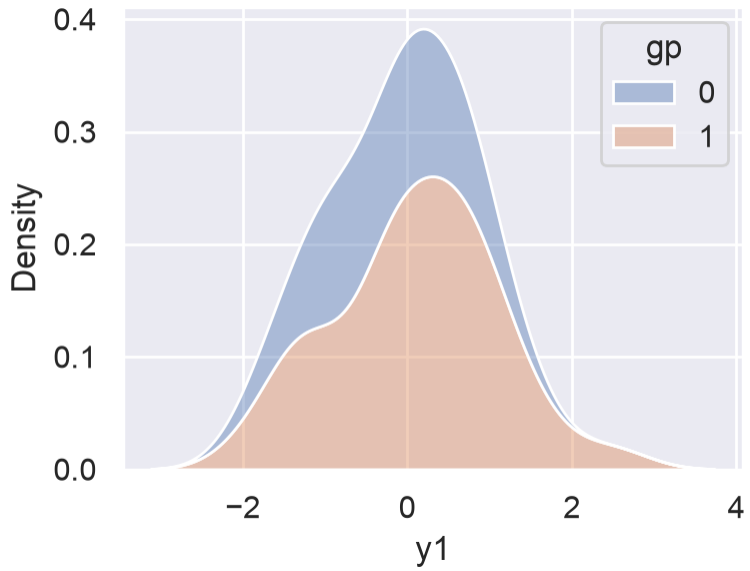


We can also get stacked bar charts, or overlapping density plots:

```
sns.histplot(data, x="z1", hue="gp", multiple="stack", shrink=0.8)
plt.figure()

sns.kdeplot(data = data, x = "y1", hue = "gp", multiple = "stack", alpha = .4)
```





Numba

For some kinds of computing, speed matters. Python is an interpreted language, which means that it has to translate commands before it runs them, and this can make it slow. One way to avoid this is to use precompiled and vectorized functions, which don't have to be translated, and can operate on entire vectors at once.

For example, it's much slower to create the log of every element in a vector using a loop than it is to use NumPy's vectorize `np.log` function:

```
import time
x = np.arange(2, 1_000_000)

# create logs in loop, record start and end time
t0 = time.perf_counter()

def slow_log(z):
  log_x_1 = np.zeros(len(x))
  for i in range(len(x)):
    log_x_1[i] = np.log(x[i])
  return log_x_1
%timeit -r 5 -n 1 slow_log(x)

# create logs with numpy
%timeit -r 5 -n 1 np.log(x)
```

```
632 ms ± 5.46 ms per loop (mean ± std. dev. of 5 runs, 1 loop each)
3.66 ms ± 170 µs per loop (mean ± std. dev. of 5 runs, 1 loop each)
```

Some algorithms can't be vectorized, however, which is where Numba comes in. Numba lets you compile a function, so it doesn't have to be interpreted before it's run (this is called *just-in-time compilation*).

Here's how it works in the example above:

```
from numba import njit
@njit
def new_log(z):
    log_z = np.zeros(len(z))
    for i in range(len(z)):
        log_z[i] = np.log(z[i])
    return log_z
x = np.arange(2, 1_000_000)

# run once to warm up
test = new_log(x)

%timeit -r 5 -n 1 new_log(x)
```

```
4.15 ms ± 377 µs per loop (mean ± std. dev. of 5 runs, 1 loop each)
```

The version compiled with numba is much faster, and all we had to do was add `@njit` before it (in a Python script, rather than the Jupyter notebook used to create this document, it is nearly as fast as the NumPy function).

Notes:

- The function won't be compiled until it's run, which may make the initial run slow (this won't add much to overall time on functions that are called repeatedly)
- Numba only works with a subset of Python, although this subset includes the key numerical computing commands

Statistics and econometrics

In addition to letting you do your own programming, Python also has packages that run common statistical and econometric procedures. One of the main packages is statsmodels.

Here is how you can run a basic regression:

```python
import statsmodels.api as sm

exog = sm.add_constant(data[['x1', 'x2']])
exog2 = sm.add_constant(data[['x1', 'x2', 'z1']])

m1 = sm.OLS(data['y1'], exog).fit()
print(m1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     y1   R-squared:                       0.006
Model:                            OLS   Adj. R-squared:                 -
0.015
Method:                 Least Squares   F-statistic:                    0.2788
Date:                Mon, 02 Nov 2026   Prob (F-statistic):              0.757
Time:                        21:34:06   Log-Likelihood:                -
134.54
No. Observations:                 100   AIC:                             275.1
Df Residuals:                      97   BIC:                             282.9
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0099      0.095     -0.105      0.917      -0.198       0.178
x1            -0.0663      0.098     -0.679      0.499      -0.260       0.127
x2            -0.0233      0.095     -0.246      0.807      -0.212       0.165
==============================================================================
Omnibus:                        0.129   Durbin-Watson:                   1.753
Prob(Omnibus):                  0.938   Jarque-Bera (JB):                0.286
Skew:                           0.059   Prob(JB):                        0.867
Kurtosis:                       2.766   Cond. No.                         1.13
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

If you prefer, you can use Patsy to use R-style formulas. You can use `C()` to include categorical variables, `I()` to include transformations of variables, `*` to include main effects and interactions, and `:` to get interactions:

```python
import statsmodels.formula.api as smf

m2 = smf.ols('y1 ~ x1 + x2', data).fit()
print(m2.summary())
m3 = smf.ols('y1 ~ x1 + x2 + C(z1)', data).fit()
m4 = smf.ols('y1 ~ x1 + I(x1**2)', data).fit()
m5 = smf.ols('y1 ~ x1*x2', data).fit()
m6 = smf.ols('y1 ~ x1:x2', data).fit()
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                     y1   R-squared:                       0.006
Model:                            OLS   Adj. R-squared:                 -
0.015
Method:                 Least Squares   F-statistic:                    0.2788
Date:                Mon, 02 Nov 2026   Prob (F-statistic):             0.757
Time:                        21:34:06   Log-Likelihood:                -
134.54
No. Observations:                 100   AIC:                             275.1
Df Residuals:                      97   BIC:                             282.9
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.0099      0.095     -0.105      0.917      -0.198       0.178
x1            -0.0663      0.098     -0.679      0.499      -0.260       0.127
x2            -0.0233      0.095     -0.246      0.807      -0.212       0.165
==============================================================================
Omnibus:                        0.129   Durbin-Watson:                   1.753
Prob(Omnibus):                  0.938   Jarque-Bera (JB):                0.286
Skew:                           0.059   Prob(JB):                        0.867
Kurtosis:                       2.766   Cond. No.                         1.13
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

You can also get a summary table and export it as LaTeX or save it as a CSV file:

```python
from statsmodels.iolib.summary2 import summary_col
results = summary_col([m2, m3, m4, m5, m6])
print(results)
print(results.as_latex())
output = results.tables[0]
output.to_csv("regression_summary.csv")
```

```
=============================================================
                 y1 I    y1 II    y1 III  y1 IIII  y1 IIIII
-------------------------------------------------------------
Intercept      -0.0099  0.0336   -0.2161  -0.0242  -0.0308
               (0.0947) (0.1890) (0.1121) (0.0929) (0.0923)
x1             -0.0663  -0.0545  -0.0145  -0.1061
               (0.0976) (0.0979) (0.0943) (0.0971)
x2             -0.0233  -0.0373           0.0158
               (0.0949) (0.0956)          (0.0945)
C(z1)[T.2]              0.0859
                        (0.2406)
C(z1)[T.3]              -0.2260
                        (0.2490)
I(x1 ** 2)                       0.2139
                                 (0.0687)
x1:x2                                     0.2052   0.1869
                                          (0.0902) (0.0872)
R-squared      0.0057   0.0271   0.0955   0.0565   0.0448
R-squared Adj. -0.0148  -0.0139  0.0768   0.0271   0.0351
=============================================================
Standard errors in parentheses.
\begin{table}
\caption{}
\label{}
\begin{center}
\begin{tabular}{llllll}
\hline
               & y1 I     & y1 II    & y1 III   & y1 IIII  & y1 IIIII  \\
\hline
Intercept      & -0.0099  & 0.0336   & -0.2161  & -0.0242  & -0.0308   \\
               & (0.0947) & (0.1890) & (0.1121) & (0.0929) & (0.0923)  \\
x1             & -0.0663  & -0.0545  & -0.0145  & -0.1061  &           \\
               & (0.0976) & (0.0979) & (0.0943) & (0.0971) &           \\
x2             & -0.0233  & -0.0373  &          & 0.0158   &           \\
```

```
                & (0.0949) & (0.0956) &          & (0.0945) &          \\
C(z1)[T.2]      &          & 0.0859   &          &          &          \\
                &          & (0.2406) &          &          &          \\
C(z1)[T.3]      &          & -0.2260  &          &          &          \\
                &          & (0.2490) &          &          &          \\
I(x1 ** 2)      &          &          & 0.2139   &          &          \\
                &          &          & (0.0687) &          &          \\
x1:x2           &          &          &          & 0.2052   & 0.1869   \\
                &          &          &          & (0.0902) & (0.0872) \\
R-squared       & 0.0057   & 0.0271   & 0.0955   & 0.0565   & 0.0448   \\
R-squared Adj. & -0.0148  & -0.0139  & 0.0768   & 0.0271   & 0.0351   \\
\hline
\end{tabular}
\end{center}
\end{table}
\bigskip
Standard errors in parentheses.
```

You can also get robust and clustered standard errors:

```
m7 = smf.ols('y1 ~ x1 + x2', data).fit(cov_type='HC1')
m8 = smf.ols('y1 ~ x1 + x2', data).fit(cov_type='cluster',
                                       cov_kwds={"groups": data['z1']})
```

Statsmodels supports hypothesis tests on model parameters:

```
print(m2.f_test("x1 = 0, x2 = 0"))
```

```
<F test: F=0.2788208579402281, p=0.7572798112559005, df_denom=97, df_num=2>
```

We can also estimate a probit, and obtain marginal effects:

```
data['y_bin'] = 1*(data['y1'] > 0)
probit = smf.probit("y_bin ~ x1 + x2", data=data).fit()
print(probit.summary())
margeff = probit.get_margeff()
print(margeff.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.687504
```

```
         Iterations 4
                           Probit Regression Results
==============================================================================
Dep. Variable:                  y_bin   No. Observations:                  100
Model:                         Probit   Df Residuals:                       97
Method:                           MLE   Df Model:                            2
Date:                Mon, 02 Nov 2026   Pseudo R-squ.:                0.008142
Time:                        21:34:06   Log-Likelihood:                 -
68.750
converged:                       True   LL-Null:                        -
69.315
Covariance Type:            nonrobust   LLR p-value:                    0.5687
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.0098      0.126      0.078      0.938      -0.238       0.257
x1            -0.0975      0.131     -0.744      0.457      -0.354       0.159
x2             0.1045      0.127      0.821      0.411      -0.145       0.354
==============================================================================
       Probit Marginal Effects
=====================================
Dep. Variable:                  y_bin
Method:                          dydx
At:                           overall
==============================================================================
                dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
x1            -0.0386      0.051     -0.751      0.453      -0.139       0.062
x2             0.0413      0.050      0.830      0.406      -0.056       0.139
==============================================================================
```

Statsmodels supports many more estimators, including binary choice models, count data models, and survival models. The linearmodels package supports panel data models, instrumental variables, and GMM estimation. The marginaleffects package provides marginal effects for many different types of models.

Basic programming examples

In addition to pre-programmed estimators, Python makes it (relatively) easy to program your own estimator.

As a simple example, here's how we can get the OLS estimate "by hand":

```
# data matrices
X = data[['x1', 'x2']].to_numpy()
ones = np.ones(100)
X = np.column_stack((ones, X))
y = data['y1'].to_numpy()

# estimator
beta_hat = np.linalg.inv(X.T @ X) @ X.T @ y
print("beta_hat: ", beta_hat)

# standard errors
e = y - X @ beta_hat
n = X.shape[0]
k = X.shape[1]
var_beta_hat = np.linalg.inv(X.T @ X) * (e.T @ e) / (n - k)
print("standard errors:\n", np.sqrt(np.diag(var_beta_hat)))

# compare to canned function

print("canned params:\n", m2.params)
print("canned variance:\n", np.sqrt(np.diag(m2.cov_params())))
```

```
beta_hat:  [-0.00992435 -0.06629607 -0.0232951 ]
standard errors:
 [0.09465218 0.09758196 0.09487458]
canned params:
 Intercept   -0.009924
x1          -0.066296
x2          -0.023295
dtype: float64
canned variance:
 [0.09465218 0.09758196 0.09487458]
```

And here's how we can estimate a probit:

```
from scipy.stats import norm
from scipy.optimize import minimize

# binary y
data['y_bin'] = 1*(data['y1'] > 0)
y_bin = data['y_bin'].to_numpy()
```

```
# likelihood function
def llf(b):
    loglik = y_bin * np.log(norm.cdf(X @ b)) +\
        (1 - y_bin) * np.log(1 - norm.cdf(X @ b))
    return(-np.sum(loglik))

llf([0, 0, 0])

# estimate parameters
prob = minimize(llf, [0, 0, 0])
print("results:\n", prob)

# standard errors
print("standard errors:\n", np.sqrt(np.diag(prob.hess_inv)))

# compare to built-in function
print("canned params:\n", probit.params)
print("canned standard errors:\n", np.sqrt(np.diag(probit.cov_params())))
```

```
results:
   message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 68.75038753899568
        x: [ 9.801e-03 -9.754e-02  1.045e-01]
      nit: 6
      jac: [ 0.000e+00 -3.815e-06 -2.861e-06]
 hess_inv: [[ 1.593e-02 -1.425e-03  4.655e-04]
            [-1.425e-03  1.717e-02 -1.674e-03]
            [ 4.655e-04 -1.674e-03  1.619e-02]]
     nfev: 40
     njev: 10
standard errors:
 [0.12620291 0.13103613 0.12722576]
canned params:
 Intercept    0.009801
x1          -0.097539
x2           0.104525
dtype: float64
canned standard errors:
 [0.12623574 0.13103926 0.12724128]
```