Introduction to Data Science using Python

A Hands-On Guide

Abhijit Dasgupta, Ph.D.

Last updated: May 22, 2020

Contents

Part I.

Introduction and Python Basics

1. About this guide

This guide will introduce you to different aspects of data science and how they can be implemented using Python

---

This guide will show you how to

-

2. A Python Primer

2.1. Introduction

Python is a popular, general purpose scripting language. The TIOBE index ranks Python as the third most popular programming language after C and Java, while this recent article in IEEE Computer Society says

> "Python can be used for web and desktop applications, GUI-based desktop applications, machine learning, data science, and network servers. The programming language enjoys immense community support and offers several open-source libraries, frameworks, and modules that make application development a cakewalk." (Belani, 2020)

2.1.1. Python is a modular language

Python is not a monolithic language but is comprised of a base programming language and numerous modules or libraries that add functionality to the language. Several of these libraries are installed with Python. The Anaconda Python Distribution adds more libraries that are useful for data science. Some libraries we will use include `numpy`, `pandas`, `seaborn`, `statsmodels` and `scikit-learn`. In the course of this workshop we will learn how to use Python libraries in your workflow.

2.1.2. Python is a scripting language

Using Python requires typing!! You write *code* in Python that is then interpreted by the Python interpreter to make the computer implement your instructions. **Your code is like a recipe that you write for the computer**. Python is a *high-level language* in that the code is English-like and human-readable and understandable, which reduces the time needed for a person to create the recipe. It is a language in that it has nouns (*variables* or *objects*), verbs (*functions*) and a structure or grammar that allows the programmer to write recipes for different functionalities.

One thing that is important to note in Python: **case is important!**. If we have two objects named `data` and `Data`, they will refer to different things.

Scripting can be frustrating in the beginning. You will find that the code you wrote doesn't work "for some reason", though it looks like you wrote it fine. The first things I look for, in order, are

1. Did I spell all the variables and functions correctly
2. Did I close all the brackets I have opened
3. Did I finish all the quotes I started, and paired single- and double-quotes
4. Did I already import the right module for the function I'm trying to use.

These may not make sense right now, but as we go into Python, I hope you will remember these to help debug your code.

2.2. An example

Let's consider the following piece of Python code:

```
# set a splitting point
split_point = 3

# make two empty lists
lower = []; upper = []

# Split numbers from 0 to 9 into two groups,
# one lower or equal to the split point and
# one higher than the split point

for i in range(10):  # count from 0 to 9
    if i <= split_point:
        lower.append(i)
    else:
        upper.append(i)

print("lower:", lower)
print("upper:", upper)
```

First note that any line (or part of a line) starting with # is a **comment** in Python and is ignored by the interpreter. This makes it possible for us to write substantial text to remind us what each piece of our code does

The first piece of code that the Python interpreter actually reads is

```
split_point = 3
```

This takes the number 3 and stores it in the **variable** `split_point`. Variables are just names where some Python object is stored. It really works as an address to some particular part of your computer's memory, telling the Python interpreter to look for the value stored at that particular part of memory. Variable names allow your code to be human-readable since it allows you to write expressive names to remind yourself what you are storing. The rules of variable names are:

1. Variable names must start with a letter or underscore
2. The rest of the name can have letters, numbers or underscores
3. Names are case-sensitive

The next piece of code initializes two **lists**, named `lower` and `upper`.

```
lower = []; upper = []
```

The semi-colon tells Python that, even though written on the same line, a particular instruction ends at the semi-colon, then another piece of instruction is written.

Lists are a catch-all data structure that can store different kinds of things, In this case we'll use them to store numbers.

The next piece of code is a **for-loop** or a loop structure in Python.

```
for i in range(10):  # count from 0 to 9
    if i <= split_point:
        lower.append(i)
    else:
        upper.append(i)
```

It basically works like this:

1. State with the numbers 0-9 (this is achieved in `range(10)`)
2. Loop through each number, naming it `i` each time
    1. Computer programs allow you to over-write a variable with a new value
3. If the number currently stored in `i` is less than or equal to the value of `split_point`, i.e., 3 then add it to the list `lower`. Otherwise add it to the list `upper`

Note the indentation in the code. **This is not by accident**. Python understands the extent of a particular block of code within a for-loop (or within a `if` statement) using the indentations. In this segment there are 3 code blocks:

1. The for-loop as a whole (1st indentation)
2. The `if` statement testing if the number is less than or equal to the split point, telling Python what to do if the test is true
3. The `else` statement stating what to do if the test in the `if` statement is false

Every time a code block starts, the previous line ends in a colon (:). The code block ends when the indentation ends. We'll go into these elements in a bit.

The last bit of code prints out the results

```
print("lower:", lower)
print("upper:", upper)
```

The `print` statement adds some text, and then prints out a representation of the object stored in the variable being printed. In this example, this is a list, and is printed as

```
lower: [0, 1, 2, 3]
upper: [4, 5, 6, 7, 8, 9]
```

We will expand on these concepts in the next few sections.

2.2.1. Some general rules on Python syntax

1. Comments are marked by #
2. A statement is terminated by the end of a line, or by a `;`.
3. Indentation specifies blocks of code within particular structures. Whitespace at the beginning of lines matters. Typically you want to have 2 or 4 spaces to specify indentation, not a tab (^) character. This can be set up in your IDE.
4. Whitespace within lines does not matter, so you can use spaces liberally to make your code more readable
5. Parentheses (`()`) are for grouping pieces of code or for calling functions.

There are several conventions about code styling including the one in PEP8 (PEP = Python Enhancement Proposal) and one proposed by Google. We will typically be using lower case names, with words separated by underscores, in this workshop, basically following PEP8. Other conventions are of course allowed as long as they are within the basic rules stated above.

2.3. Data types in Python

We start with objects in Python. Objects can be of different types, including numbers (integers and floats), strings, arrays (vectors and matrices) and others. Any object can be assigned to a name, so that we can refer to the object in our code. We'll start with the basic types of objects.

2.3.1. Numeric variables

The following is a line of Python code, where we assign the value 1.2 to the variable `a`.

> The act of assigning a name is done using the = sign. This is not equality in the mathematical sense, and has some non-mathematical behavior, as we'll see

```
a = 1.2
```

This is an example of a *floating-point number* or a decimal number, which in Python is called a `float`. We can verify this in Python itself.

```
type(a)
```

Floating point numbers can be entered either as decimals or in scientific notation

```
x = 0.0005
y = 5e-4 # 5 x 10^(-4)
print(x == y)
```

You can also store integers in a variable. Integers are of course numbers, but can be stored more efficiently on your computer. They are stored as an *integer* type, called `int`

```
b = 23
type(b)
```

These are the two primary numerical data types in Python. There are some others that we don't use as often, called `long` (for *long integers*) and `complex` (for *complex numbers*)

2.3.1.1. Operations on numbers

There is an arithmetic and logic available to operate on elemental data types. For example, we do have addition, subtraction , multiplication and division available. For example, for numbers, we can do the following:

Operation	Result
x + y	The sum of x and y
x - y	The difference of x and y
x * y	The product of x and y
x / y	The quotient of x and y
- x	The negative of x
abs(x)	The absolute value of x
x ** y	x raised to the power y
int(x)	Convert a number to integer
float(x)	Convert a number to floating point

Let's see some examples:

```
x = 5
y = 2
```

```
x + y
```

```
x - y
```

```
x * y
```

```
x / y
```

```
x ** y
```

2.3.2. Strings

Strings are how text is represented within Python. It is always represented as a quoted object using either single (`''`) or double (`""`) quotes, as long as the types of quotes are matched. For example:

```
first_name = "Abhijit"
last_name = "Dasgupta"
```

The data type that these are stored in is `str`.

```
type(first_name)
```

2.3.2.1. Operations

Strings also have some "arithmetic" associated with it, which involves, essentially, concatenation and repetition. Let's start by considering two character variables that we've initialized.

```
a = "a"
b = "b"
```

Then we get the following operations and results

Operation	Result
a + b	'ab'
a * 5	'aaaaa'

We can also see if a particular character or character string is part of an exemplar string

```
last_name = "Dasgupta"
"gup" in last_name
```

String manipulation is one of the strong suites of Python. There are several *functions* that apply to strings, that we will look at throughout the workshop, and especially when we look at string manipulation. In particular, there are built-in functions in base Python and powerful *regular expression* capabilities in the `re` module.

2.3.3. Truthiness

Truthiness means evaluating the truth of a statement. This typically results in a Boolean object, which can take values `True` and `False`, but Python has several equivalent representations. The following values are considered the same as False:

> `None`, `False`, zero (`0`, `0L`, `0.0`), any empty sequence (`[]`, `''`, `()`), and a few others

All other values are considered True. Usually we'll denote truth by `True` and the number `1`.

2.3.3.1. Operations

We will typically test for the truth of some comparisons. For example, if we have two numbers stored in x and y, then we can perform the following comparisons

Operation	Result
x < y	x is strictly less than y
x <= y	x is less than or equal to y
x == y	x equals y (note, it's 2 = signs)
x != y	x is not equal to y
x > y	x is strictly greater than y
x >= y	x is greater or equal to y

We can chain these comparisons using Boolean operations

Operation	Result
x | y	Either x is true or y is true or both
x & y	Both x and y are true
not x	if x is true, then false, and vice versa

For example, if we have a number stored in x, and want to find out if it is between 3 and 7, we could write

```
(x >= 3) & (x <= 7)
```

2.3.3.2. A note about variables and types

Some computer languages like C, C++ and Java require you to specify the type of data that will be held in a particular variable. For example,

```
int x = 4;
float y = 3.25;
```

If you try later in the program to assign something of a different type to that variable, you will raise an error. For example, if I did, later in the program, `x = 3.95;`, that would be an error in C.

Python is **dynamically typed**, in that the same variable name can be assigned to different data types in different parts of the program, and the variable will simply be "overwritten". (This is not quite correct. What actually happens is that the variable name now "points" to a different part of the computer's memory where the new data is then stored in appropriate format). So the following is completely fine in Python:

```
x = 4  # An int
x = 3.5  # A float
x = "That's my mother"  # A str
x = True  # A bool
```

> Variables are like individual ingredients in your recipe. It's *mis en place* or setting the table for any operations (*functions*) we want to do to them. In a language context, variables are like *nouns*, which will be acted on by verbs (*functions*). In the next section we'll look at collections of variables. These collections are important in that it allows us to organize our variables with some structure.

2.4. Data structures in Python

Python has several in-built data structures. We'll describe the three most used ones:

1. Lists (`[]`)
2. Tuples (`()`)
3. Dictionaries or dicts (`{}`)

Note that there are three different kinds of brackets being used.

Lists are baskets that can contain different kinds of things. They are ordered, so that there is a first element, and a second element, and a last element, in order. However, the *kinds* of things in a single list doesn't have to be the same type.

Tuples are basically like lists, except that they are *immutable*, i.e., once they are created, individual values can't be changed. They are also ordered, so there is a first element, a second element and so on.

Dictionaries are **unordered** key-value pairs, which are very fast for looking up things. They work almost like hash tables. Dictionaries will be very useful to us as we progress towards the PyData stack. Elements need to be referred to by *key*, not by position.

2.4.1. Lists

```
test_list = ["apple", 3, True, "Harvey", 48205]
test_list
```

There are various operations we can do on lists. First, we can determine the length (or size) of the list

```
len(test_list)
```

The list is a catch-all, but we're usually interested in extracting elements from the list. This can be done by *position*, since lists are *ordered*. We can extract the 1st element of the list using

```
test_list[0]
```

> Wait!! The index is 0?
>
> Yup. Python is based deep underneath on the C language, where counting starts at 0. So the first element has index 0, second has index 1, and so on. So you need to be careful if you're used to counting from 1, or, if you're used to R, which does start counting at 1.

We can also extract a set of consecutive elements from a list, which is often convenient. The typical form is to write the index as `a:b`. The (somewhat confusing) rule is that `a:b` means that you start at index `a`, but continue until **before index b**. So the notation `2:5` means include elements with index 2, 3, and 4. In the Python world, this is called **slicing**.

```
test_list[2:5]
```

If you want to start at the beginning or go to the end, there is a shortcut notation. The same rule holds, though. `:3` does **not** include the element at index 3, but `2:` **does** include the element at index 2.

```
test_list[:3]
```

```
test_list[2:]
```

The important thing here is if you provide an index `a:b`, then `a` is include but `b` **is not**.

You can also count **backwards** from the end. The last element in a Python list has index `-1`.

index	0	1	2	3	4
element	'apple'	3	True	'Harvey'	48205
counting backwards	-5	-4	-3	-2	-1

```
test_list[-1]
```

You can also use negative indices to denote sequences within the list, with the same indexing rule applying. Note that you count from the last element (-1) and go backwards.

```
test_list[:-1]
```

```
test_list[-3:]
```

```
test_list[-3:-1]
```

You can also make a list of lists, or nested lists

```
test_nested_list = [[1, "a", 2, "b"], [3, "c", 4, "d"]]
test_nested_list
```

This will come in useful when we talk about arrays and data frames.

You can also check if something is in the list, i.e. is a member.

```
"Harvey" in test_list
```

Lists have the following properties

- They can be heterogenous (each element can be a different type)
- Lists can hold complex objects (lists, dicts, other objects) in addition to atomic objects (single numbers or words)
- List have an ordering, so you can access list elements by position
- List access can be done counting from the beginning or the end, and consecutive elements can be extracted using slices.

2.4.2. Tuples

Tuples are like lists, except that once you create them, you can't change them. This is why tuples are great if you want to store fixed parameters or entities within your Python code, since they can't be over-written even by mistake. You can extract elements of a tuple, but you can't over-write them. This is called *immutable*.

Note that, like lists, tuples can be heterogenous, which is also useful for coding purposes, as we will see.

```
test_tuple = ("apple", 3, True, "Harvey", 48205)
```

```
test_tuple[:3]
```

```
test_list[0] = "pear"
test_list
```

See what happens in the next bit of code

```
test_tuple[0] = "pear"
test_tuple
```

(I'm not running this since it gives an error)

> Tuples are like lists, but once created, they cannot be changed. They are ordered and can be sliced.

2.4.3. Dictionaries

Dictionaries, or `dict`, are collections of key-value pairs. Each element is referred to by *key*, not by *index*. In a dictionary, the keys can be strings, numbers or tuples, but the values can be any Python object. So you could have a dictionary where one value is a string, another is a number and a third is a DataFrame (essentially a data set, using the pandas library). A simple example might be an entry in a list of contacts

```
contact = {
    "first_name": "Abhijit",
    "last_name": "Dasgupta",
    "Age": 48,
    "address": "124 Main St",
    "Employed": True,
}
```

Note the special syntax. You separate the key-value pairs by colons (`:`), and each key-value pair is separated by commas. If you get a syntax error creating a dict, look at these first.

If you try to get the first name out using an index, you run into an error:

```
contact[0]
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 0
```

You need to extract it by key

```
contact["first_name"]
```

A dictionary is mutable, so you can change the value of any particular element

```
contact["address"] = "123 Main St"
contact["Employed"] = False
contact
```

You can see all the keys and values in a dictionary using extractor functions

```
contact.keys()
```

```
contact.values()
```

It turns out that dictionaries are really fast in terms of retrieving information, without having to count where an element it. So it is quite useful

We'll see that dictionaries are also one way to easily create pandas DataFrame objects on the fly.

There are a couple of other ways to create dict objects. One is using a list of tuples. Each key-value pair is represented by a tuple of length 2, where the 1st element is the key and the second element is the value.

```
A = [('first_name','Abhijit'),('last_name','Dasgupta'),('address', '124 Main St')]
dict(A)
```

This actually can be utilized to create a dict from a pair of lists. There is a really neat function, `zip`, that inputs several lists of the same length and creates a list of tuples, where the i-th element of each tuple comes from the i-th list, in order.

```
A = ['first_name', 'last_name','address']
B = ['Abhijit','Dasgupta','124 Main St']
dict(zip(A, B))
```

> The `zip` function is quite powerful in putting several lists together with corresponding elements of each list into a tuple

On a side note, there is a function `defaultdict` from the `collections` module that is probably better to use. We'll come back to it when we talk about modules.

2.5. Operational structures in Python

2.5.1. Loops and list comprehensions

Loops are a basic construct in computer programming. The basic idea is that you have a recipe that you want to repeatedly run on different entities that you have created. The crude option would be to copy and paste your code several times, changing whatever inputs change across the entities. This is not only error-prone, but inefficient given that loops are a standard element of all programming languages.

You can create a list of these entities, and, using a loop, run your recipe on each entity automatically. For example, you have a data about votes in the presidential election from all 50 states, and you want to figure out what the percent voting for each major party is. So you could write this recipe in pseudocode as

```
Start with a list of datasets, one for each state
for each state
    compute and store fraction of votes that are Republican
    compute and store fraction of votes that are Democratic
```

This is just English, but it can be translated easily into actual code. We'll attempt that at the end of this section.

The basic idea of a list is that there is a list of things you want to iterate over. You create a dummy variable as stand-in for each element of that list. Then you create a for-loop. This works like a conveyor belt and basket, so to speak. You line up elements of the list on the conveyor belt, and as you run the loop, one element of the list is "scooped up" and processed. Once that processing is done, the next element is "scooped up", and so forth. The dummy variable is essentially the basket (so the same basket (variable name) is re-used over and over until the conveyor belt (list) is empty).

In the examples below, we are showing a common use of for loops where we are enumerating the elements of a list as 0, 1, 2, ... using `range(len(test_list))`. So the dummy variable `i` takes values 0, 1, 2, ... until the length of the list is reached. For each value of `i`, this for loop prints the i-th element of `test_list`.

```
for i in range(len(test_list)):
    print(test_list[i])
```

Sometimes using the index number is easier to understand. However, we don't need to do this. We can just send the list itself into the for-loop (u) now is the dummy variable containing the actual element of `test_list`. We'll get the same answer.

```
for u in test_list:
    print(u)
```

The general structure for a `for` loop is:

```
for (element) in (list):
      do some stuff
      do more stuff
```

As a more practical example, let's try and sum a set of numbers using a for-loop (we'll see much better ways of doing this later)

```
test_list2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
mysum = 0
for u in test_list2:
    mysum = mysum + u
print(mysum)
```

There are two things to note here.

1. The code `mysum = mysum + u` is perfectly valid, once you realize that this isn't really math but an assignment or pointer to a location in memory. This code says that you find the current value stored in `mysum`, add the value of `u` to it, and then store it back into the storage that `mysum` points to
2. Indentation matters! Indent the last line and see what happens when you run this code

2.5.1.1. A little deeper

The entity to the right of the `in` in the for-loop can be an **iterator**, which is a generalization of a list. For example, we used `range(len(test_list2))` above. If we just type

```
range(10)
```

nothing really happens. This is an example of an iterator, which is only evaluated when it is called, rather than being stored in memory. This is useful especially when you iterate over large numbers of things, in terms of preserving memory and speed. To see the corresponding list, you would do

```
list(range(10))
```

This `range` iterator is quite flexible:

```
list(range(5, 10))  # range from 5 to 10
```

```
list(range(0, 10, 2))  # range from 0 to 10 by 2
```

Note the rules here are very much like the slicing rules.

Other iterators that are often useful are the `enumerate` iterator and the `zip` iterator.

`enumerate` automatically creates both the index and the value for each element of a list.

```
L = [0, 2, 4, 6, 8]
for i, val in enumerate(L):
    print(i, val)
```

`zip` puts multiple lists together and creates a composite iterator. You can have any number of iterators in zip, and the length of the result is determined by the length of the shortest iterator. We introduced an example of `zip` as a way to create a `dict`.

Technically, `zip` can take multiple *iterators* as inputs, not just lists

```
first = ["Han", "Luke", "Leia", "Anakin"]
last = ["Solo", "Skywalker", "Skywaker", "Skywalker"]
types = ['light','light','light','light/dark/light']

for val1, val2, val3 in zip(first, last, types):
    print(val1, val2, ' : ', val3)
```

2.5.1.2. Controlling loops

There are two statements that can affect how loops run:

- The `break` statement breaks out of the loop
- The `continue` statement skips the rest of the current loop and continues to the next element

For example

```
x = list(range(10))

for u in x:
    if u % 2 == 1: # If u / 2 gives a remainder of 1
        continue
    if u >= 8:
        break
    print(u)
```

In this loop, we don't print the odd numbers, and we stop the loop once it gets to 8.

2.5.2. List comprehensions

List comprehensions are quick ways of generating a list from another list by using some recipe. For example, if we wanted to create a list of the squares of all the numbers in `test_list2`, we could write

```
squares = [u ** 2 for u in test_list2]
squares
```

Similarly, if we wanted to find out what the types of each element of `test_tuple` is, we could use

```
[type(u) for u in test_tuple]
```

Exercise: Can you use a list comprehension to find out the types of each element of the `contact` dict?

We can also use list comprehensions to extract arbitrary sets of elements of lists

```
test = ['a','b','c','d','e','f','g']
test1 = [test[i] for i in [0,2,3,5]]
test1
```

2.5.3. Conditional evaluations

The basic structure for conditional evaluation of code is an **if-then-else** structure.

```
if Condition 1 is true then
    do Recipe 1
else if (elif) Condition 2 is true then
  do Recipe 2
else
  do Recipe 3
```

In Python, this is implemented as a `if-elif-else` structure. Let's take an example where we have a list of numbers, and we want to record whether the number is negative, odd, or even.

```
x = [-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = []  # an empty list

for u in x:
    if u < 0:
        y.append("Negative")
    elif u % 2 == 1:  # what is remainder when dividing by 2
        y.append("Odd")
    else:
        y.append("Even")

print(y)
```

Note here that the indentation (leading whitespace) is crucial to this structure. The `if-elif-else` structure is embedded in a for-loop, so the entire structure in indented. Also, each particular recipe is also indented within the if-elif-else structure.

> The `elif` is optional, in that if you have only 2 conditions, then an `if-else` structure is sufficient. However, you can have multiple `elif`'s if you have more conditions. This kind of structure has to start with an `if`, end with an `else` and can have 0 or more `elif` in the middle.

2.6. Functions

We've already seen some examples of **functions**, such as the `print()` function. For example, if we write `print(y)`, the function name is `print` and the functions *argument* is `y`. So what are functions?

Functions are basically encapsulated recipes. They are groups of code that are given a name and can be called with 0 or more arguments. In a cookbook, you might have a recipe for pasta primavera. This is the name of a recipe that has ingredients and a method to cook. In Python, a similar recipe for the mean might be as follows:

```
def my_mean(x):
    y = 0
    for u in x:
        y += u
    y = y / len(x)
    return y
```

This takes a list of numbers `x`, loops over the elements of `x` to find their sum, and then divides by the length of `x` to compute the mean. It then returns this mean.

> The notation `+=` is a shortcut often used in programming. The statement `y += u` means, take the current value of `y`, add the value of `u` to it, and store it back in to `y`. This is a shorthand for `y = y + u`. In analogous fashion, you can use `-=`, `*=` and `/=` to do subtraction, multiplication and division respectively.

A Python function must start with the keyword `def` followed by the name of the function, the arguments within parentheses, and then a colon. The actual code for the function is indented, just like in for-loops and if-elif-else structures. It ends with a `return` function which specifies the output of the function.

To use the `my_mean` function,

```
x = list(range(10))
my_mean(x)
```

2.6.1. Documenting your functions

Python has an in-built documentation system that allows you to readily document your functions using *docstrings*. Basically, right after the first line with `def`, you can create a (multi-line) string that documents the function and will be printed if the help system is used for that function. You can create a multi-line string by **bounding it with 3 quotation marks on each side**. For example,

```
def my_mean(x):
    """
  A function to compute the mean of a list of numbers.

  INPUTS:
  x : a list containing numbers
```

```
  OUTPUT:
  The arithmetic mean of the list of numbers
  """
    y = 0
    for u in x:
        y = y + u
    y = y / len(x)
    return y
```

```
help(my_mean)
```

2.7. Modules and Packages

Python itself was built with the principle "Batteries included", in that it already comes with useful tools for a wide variety of tasks. On top of that, there is a large ecosystem of third-party tools and packages that can be added on to add more functionality. Almost all the data science functionality in Python comes from third-party packages.

2.7.1. Using modules

The Python standard library as well as third-party packages (which I'll use interchangeably with the term libraries) are structured as modules. In order to use a particular module you have to "activate" it in your Python session using the `import` statement.

```
import math

math.cos(math.pi)
```

In these statements, we have imported the `math` module. This module has many functions, one of which is the cosine or `cos` function. We use the notation `math.cos` to let Python know that we want to use the `cos` function that is in the `math` module. The value of π is also stored in the `math` module as `math.pi`, ie. the element `pi` within the moduel `math`.

Modules can often have long names, so Python caters to our laziness by allowing us to create aliases for modules when we import them. In this workshop we will use the following statements quite often

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

These statements import 3 modules into the current Python session, namely `numpy`, `pandas` and a sub-module of the `matplotlib` module called `pyplot`. In each case, we have provided an alias to the module that is imported. So, in subsequent calls, we can just use the aliases.

```
np.cos(np.pi)
```

If we only want some particular components of a module to be imported, we can specify them using the `from ... import ...` syntax. These imported components will not need the module specification when we subsequently use them.

```
from math import pi, sin, cos

print(sin(pi))
print(cos(pi))
```

We had made reference to the `defaultdict` function from the `collections` module before. Using this instead of `dict` can be advantagous sometimes in data scientific work.

```
from collections import defaultdict

A = defaultdict(list) # Specify each component will be a list
B = {}

s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
for k,v in s: # k = key, v = value
    B[k].append(v)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: 'yellow'
```

```
for k,v in s:
    A[k].append(v)
A
```

The `defaultdict` sees a new key, and adds it to the dict, initializing it with an empty list (since we specified `defaultdict(list)`. The normal dict requires the key to already be in place in the dict for any operations to take place on that key-value pair. So the `default` dict is safer for on-the-fly work and when we don't know beforehand what keys we will encounter when storing data into the dict.

> There is a temptation to use this method to import everything in a module so you don't have to specify the module. This is a **bad practice** generally, both because you clutter up the namespace that Python reads from, and because you may unknowingly over-write and replace a function from one module with one from another module, and you will have a hard time debugging your code.
>
> The code you do **NOT** want to use is
>
> ```
> from math import *
> ```

2.7.2. Useful modules in Python's standard library

Module	Description
`os` and `sys`	Interfacing with the operating system, including files, directories, and executing shell commands
`math` and `cmath`	Mathematical functions
`itertools`	Constructing and using iterators
`random`	Generate random numbers
`collections`	More general collections for objects, beyond lists, tuples and dicts

2.7.3. Installing third-party packages/libraries

The Anaconda Python distribution comes with its own installer and package manager called `conda`. The Anaconda repository contains most of the useful packages for data science, and many come pre-installed with the distribution. However, you can easily install packages using the `conda` manager.

```
conda install pandas
```

would install the `pandas` package into your Python installation. If you wanted a particular version of this package, you could use

```
conda install pandas=0.23
```

to install version 0.23 of the `pandas` package.

Anaconda also provides a repository for user-created packages. For example, to install the Python package RISE which I use for creating slides from Jupyter notebooks, I use

```
conda install -c conda-forge rise
```

Sometimes you may find a Python package that is not part of the Anaconda repositories. Then you can use the more general Python program `pip` to install packages

```
pip install supersmoother
```

This goes looking in the general Python package repository PyPi, which you can also search on a web browser.

2.8. Environments

One of the nice things about Python is that you can set up environments for particular projects, that have all the packages you need for that project, without having to install those packages system-wide. This practice is highly recommended, since it creates a sandbox for you to play in for a project without contaminating the code from another project.

The Anaconda distribution and the `conda` program make this quite easy. There are a couple of ways of doing this.

2.8.1. Command-line/shell

You can open up a command line terminal (any terminal on Mac and Linux, the Anaconda Terminal in Windows) to create a new environment. For example, I have an environment I call **ds** that is my data science environment. This will include the packages `numpy`, `scipy`, `pandas`,`matplotlib`, `seaborn`,`statsmodels` and `scikit-learn` in it. The quick way to do this is

```
conda create -n ds numpy scipy pandas matplotlib seaborn statsmodels scikit-learn
```

To use this environment, at the command line, type

```
conda activate ds
```

Once you're done using it, at the command line, type

```
conda deactivate
```

When your environment is activated, you'll see the name of the environment before the command prompt

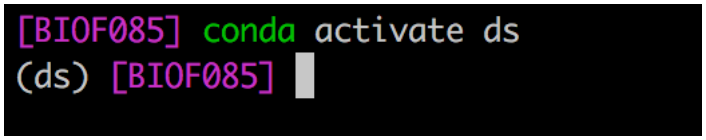


Figure 2.1.: image-20200511003754816

2.8.2. Using Anaconda Navigator

Open the Anaconda Navigator from your start menu or using Spotlight on a Mac. Within the app is a section named "Environments"

At the bottom of the 2nd pane, you can see a "Create" button. Clicking it creates a pop-up window.

I've named this new environment `ds1` since I already have a `ds` environment. Click "Create". You'll have to wait a bit of time for the environment to be created. You can then add/install packages to this environment by clicking on packages on the right panel, making sure you changed the first drop-down menu from "Installed" to "Not installed".

Once you've selected the packages you want to install, click "Appy" on the bottom right of the window.

To activate an environment, you can go to the Home pane for Anaconda Navigator and change the environment on the "Applications on" drop-down menu.

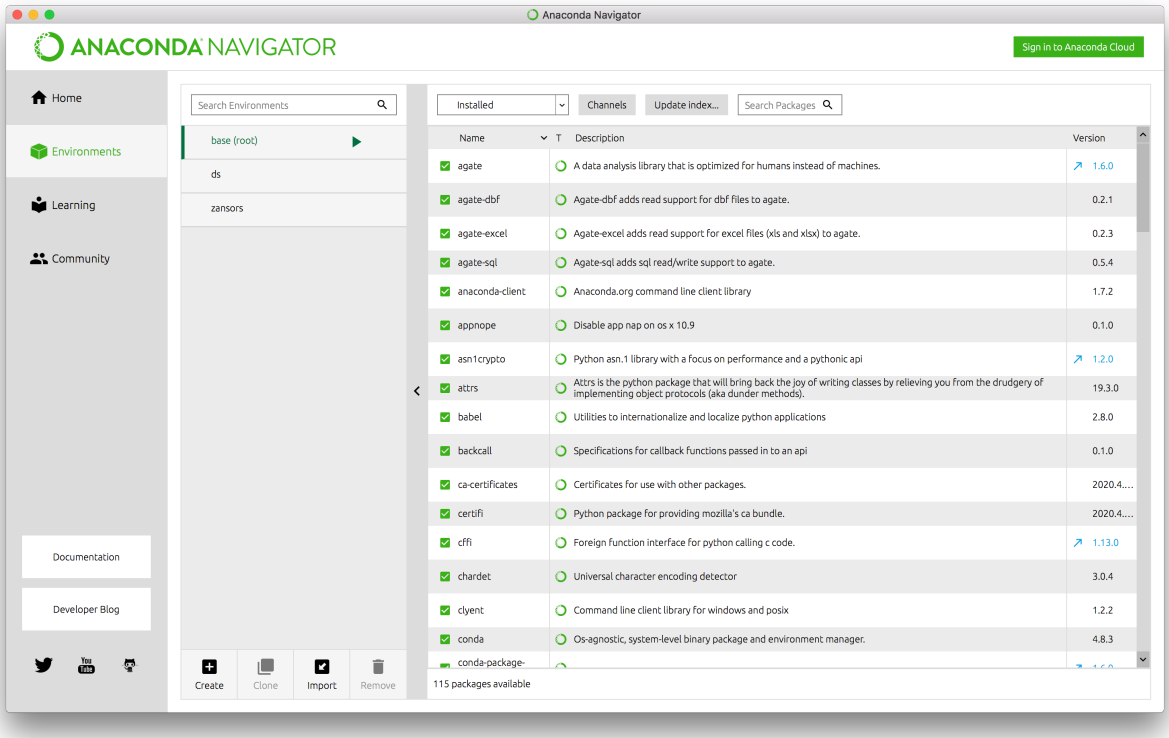


Figure 2.2.: image-20200511004048781

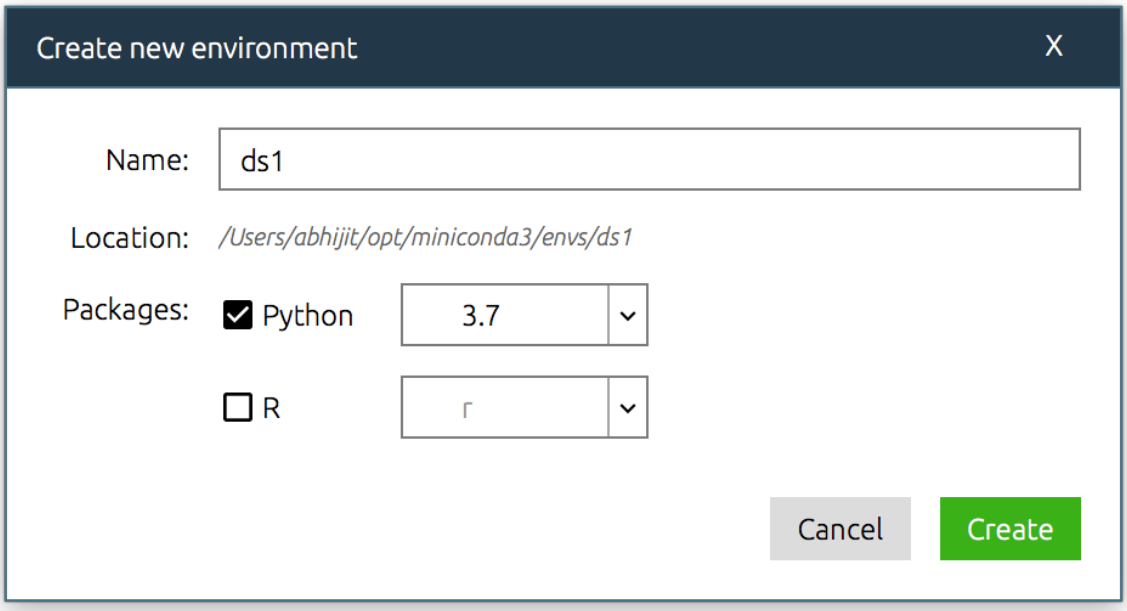


Figure 2.3.: image-20200511004259459

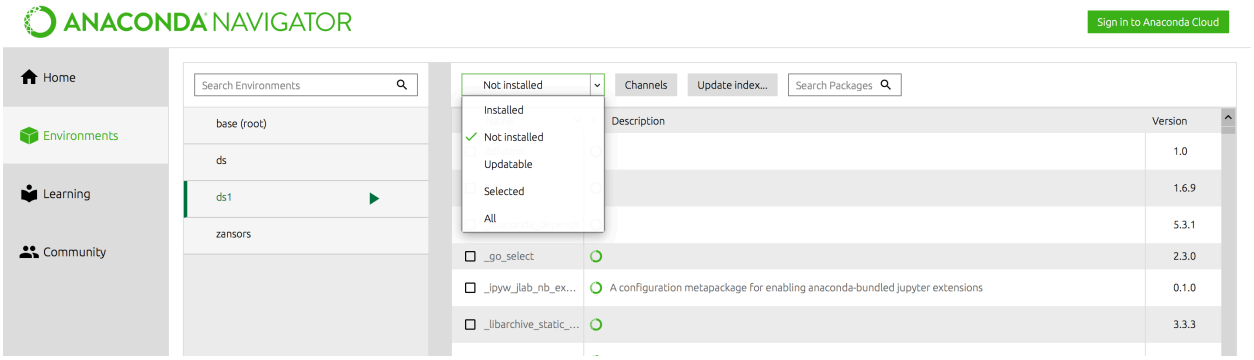


Figure 2.4.: image-20200511004530076

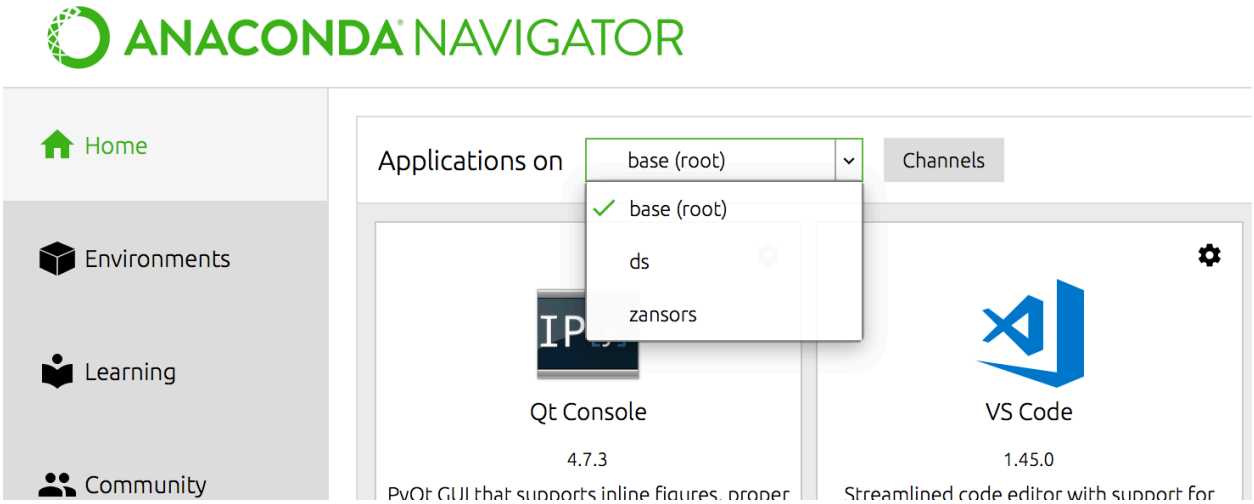


Figure 2.5.: image-20200511004920105

2.8.3. Reproducing environments

Suppose you've got an environment set up the way you like it, and want to clone it on another machine that has Anaconda installed. There is an easy way to do this. You have to use the command line (Anaconda Prompt (Win) or a terminal) for this.

First activate the environment you want to export (I'll use `ds` as an example)

```
conda activate ds
```

Then export the environment specifications which includes all the packages installed in that environment

```
conda env export > environment.yml
```

You can take this `environment.yml` file to a new computer, or e-mail it to a collaborator to install the environment. This environment can be created on the new computer using

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

where the first line of the `environment.yml` file creates the environment name.

You can also create the environment from an `environment.yml` file from Anaconda Navigator by using the Import button rather than the Create button in the instructions above.

> If you are changing operating systems, create the `environment.yml` file using the command
>
> ```
> conda env export --from-history > environment.yml
> ```
>
> This avoids potential issues with dependencies that may not be compatible across operating systems

2.9. Seeking help

Most Python functions have some amount of documentation. As we saw when we created our own function, this documentation is part of the function definition. It can be accessed at the Python console in 2 ways:

```
help(sum)
```

or

```
# sum?
```

You can see the documentation of the `my_sum` function we created earlier in this way, as well.

Other resources that are your friends in the internet age are

1. Stack Overflow: This is a Q & A site. To find Python-related questions, use the tag `python`.
2. Google: Of course.
3. Cross-Validated: A data science oriented Q & A site. Once again, use the tag `python`.

3. Python tools for data science

(last updated 2020-05-18)

3.1. The PyData Stack

The Python Data Stack comprises a set of packages that makes Python a powerful data science language. These include

- Numpy: provides arrays and matrix algebra
- Scipy: provides scientific computing capabilities
- matplotlib: provides graphing capabilities

These were the original stack that was meant to replace Matlab. However, these were meant to tackle purely numerical data, and the kinds of heterogeneous data we regularly face needed more tools. These were added more recently.

- Pandas: provides data analytic structures like the data frame, as well as basic descriptive statistical capabilities
- statsmodels: provides a fairly comprehensive set of statistical functions
- scikit-learn: provides machine learning capabilities

This is the basic stack of packages we will be using in this workshop. Additionally we will use a few packages that add some functionality to the data science process. These include

- seaborn: Better statistical graphs
- plotly: Interactive graphics
- biopython: Python for bioinformatics

We may also introduce the package `rpy2` which allows one to run R from within Python. This can be useful since many bioinformatic pipelines are already implemented in R.

> The PyData stack also includes `sympy`, a symbolic mathematics package emulating Maple

3.2. Numpy (numerical and scientific computing)

We start by importing the Numpy package into Python using the alias `np`.

```
import numpy as np
```

Numpy provides both arrays (vectors, matrices, higher dimensional arrays) and vectorized functions which are very fast. Let's see how this works.

```
z = [1,2,3,4,5,6,7,8,9.3,10.6] # This is a list
z_array = np.array(z)
z_array
```

Now, we have already seen functions in Python earlier. In Numpy, there are functions that are optimized for arrays, that can be accessed directly from the array objects. This is an example of *object-oriented programming* in Python, where functions are provided for particular *classes* of objects, and which can be directly accessed from the objects. We will use several such functions over the course of this workshop, but we won't actually talk about how to do this program development here.

> Numpy functions are often very fast, and are *vectorized*, i.e., they are written to work on vectors of numbers rather than single numbers. This is an advantage in data science since we often want to do the same operation to all elements of a column of data, which is essentially a vector

We apply the functions `sum`, `min` (minimum value) and `max` (maximum value) to `z_array`.

```
z_array.sum()
```

```
z_array.min()
```

```
z_array.max()
```

The versions of these functions in Numpy are optimized for arrays and are quite a bit faster than the corresponding functions available in base Python. When doing data work, these are the preferred functions.

These functions can also be used in the usual function manner:

```
np.max(z_array)
```

Calling `np.max` ensures that we are using the `max` function from numpy, and not the one in base Python.

3.2.1. Numpy data types

Numpy arrays are homogeneous in type.

```
np.array(['a','b','c'])
```

```
np.array([1,2,3,6,8,29])
```

But, what if we provide a heterogeneous list?

```
y = [1,3,'a']
np.array(y)
```

So what's going on here? Upon conversion from a heterogeneous list, numpy converted the numbers into strings. This is necessary since, by definition, numpy arrays can hold data of a single type. When one of the elements is a string, numpy casts all the other entities into strings as well. Think about what would happen if the opposite rule was used. The string 'a' doesn't have a corresponding number, while both numbers 1 and 3 have corresponding string representations, so going from string to numeric would create all sorts of problems.

> The advantage of numpy arrays is that the data is stored in a contiguous section of memory, and you can be very efficient with homogeneous arrays in terms of manipulating them, applying functions, etc. However, `numpy` does provide a "catch-all" `dtype` called `object`, which can be any Python object. This `dtype` essentially is an array of pointers to actual data stored in different parts of the memory. You can get to the actual objects by extracting them. So one could do

```
np.array([1,3,'a'], dtype='object')
```

> which would basically be a valid `numpy` array, but would go back to the actual objects when used, much like a list. We can see this later if we want to transform a heterogeneous `pandas` `DataFrame` into a `numpy` array. It's not particularly useful as is, but it prevents errors from popping up during transformations from `pandas` to `numpy`.

3.2.2. Generating data in numpy

We had seen earlier how we could generate a sequence of numbers in a list using `range`. In numpy, you can generate a sequence of numbers in an array using `arange` (which actually creates the array rather than provide an iterator like `range`).

```
np.arange(10)
```

You can also generate regularly spaced sequences of numbers between particular values

```
np.linspace(start=0, stop=1, num=11) # or np.linspace(0, 1, 11)
```

You can also do this with real numbers rather than integers.

```
np.linspace(start = 0, stop = 2*np.pi, num = 10)
```

More generally, you can transform lists into numpy arrays. We saw this above for vectors. For matrices, you can provide a list of lists. Note the double [in front and back.

```
np.array([[1,3,5,6],[4,3,9,7]])
```

You can generate an array of 0's

```
np.zeros(10)
```

This can easily be extended to a two-dimensional array (a matrix), by specifying the dimension of the matrix as a tuple.

```
np.zeros((10,10))
```

You can also generate a matrix of 1s in a similar manner.

```
np.ones((3,4))
```

In matrix algebra, the identity matrix is important. It is a square matrix with 1's on the diagonal and 0's everywhere else.

```
np.eye(4)
```

You can also create numpy vectors directly from lists, as long as lists are made up of atomic elements of the same type. This means a list of numbers or a list of strings. The elements can't be more composite structures, generally. One exception is a list of lists, where all the lists contain the same type of atomic data, which, as we will see, can be used to create a matrix or 2-dimensional array.

```
a = [1,2,3,4,5,6,7,8]
b = ['a','b','c','d','3']

np.array(a)
```

```
np.array(b)
```

3.2.2.1. Random numbers

Generating random numbers is quite useful in many areas of data science. All computers don't produce truly random numbers but generate *pseudo-random* sequences. These are completely deterministic sequences defined algorithmically that emulate the properties of random numbers. Since these are deterministic, we can set a *seed* or starting value for the sequence, so that we can exactly reproduce this sequence to help debug our code. To actually see how things behave in simulations we will often run several sequences of random numbers starting at different seed values.

The seed is set by the `RandomState` function within the `random` submodule of numpy. Note that all Python names are case-sensitive.

```
rng = np.random.RandomState(35) # set seed
rng.randint(0, 10, (3,4))
```

We have created a 3x4 matrix of random integers between 0 and 10 (in line with slicing rules, this includes 0 but not 10).

We can also create a random sample of numbers between 0 and 1.

```
rng.random_sample((5,2))
```

We'll see later how to generate random numbers from particular probability distributions.

3.2.3. Vectors and matrices

Numpy generates arrays, which can be of arbitrary dimension. However the most useful are vectors (1-d arrays) and matrices (2-d arrays).

In these examples, we will generate samples from the Normal (Gaussian) distribution, with mean 0 and variance 1.

```
A = rng.normal(0,1,(4,5))
```

We can compute some characteristics of this matrix's dimensions. The number of rows and columns are given by `shape`.

```
A.shape
```

The total number of elements are given by `size`.

```
A.size
```

If we want to create a matrix of 0's with the same dimensions as `A`, we don't actually have to compute its dimensions. We can use the `zeros_like` function to figure it out.

```
np.zeros_like(A)
```

We can also create vectors by only providing the number of rows to the random sampling function. The number of columns will be assumed to be 1.

```
B = rng.normal(0, 1, (4,))
B
```

3.2.3.1. Extracting elements from arrays

The syntax for extracting elements from arrays is almost exactly the same as for lists, with the same rules for slices.

Exercise: State what elements of B are extracted by each of the following statements

```
B[:3]
B[:-1]
B[[0,2,4]]
B[[0,2,5]]
```

For matrices, we have two dimensions, so you can slice by rows, or columns or both.

```
A
```

We can extract the first column by specifying `:` (meaning everything) for the rows, and the index for the column (reminder, Python starts counting at 0)

```
A[:,0]
```

Similarly the 4th row can be extracted by putting the row index, and `:` for the column index.

```
A[3,:]
```

All slicing operations work for rows and columns

```
A[:2,:2]
```

3.2.3.2. Array operations

We can do a variety of vector and matrix operations in `numpy`.

First, all usual arithmetic operations work on arrays, like adding or multiplying an array with a scalar.

```
A = rng.randn(3,5)
A
```

```
A + 10
```

We can also add and multiply arrays **element-wise** as long as they are the same shape.

```
B = rng.randint(0,10, (3,5))
B
```

```
A + B
```

```
A * B
```

You can also do **matrix multiplication**. Recall what this is.

If you have a matrix A_{mxn} and another matrix B_{nxp}, as long as the number of columns of A and rows of B are the same, you can multiply them ($C_{mxp} = A_{mxn}B_{nxp}$), with the (i,j)-th element of C being

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}, i = 1, \dots, m; j = 1, \dots, p$$

In numpy the operant for matrix multiplication is `@`.

In the above examples, `A` and `B` cannot be multiplied since they have incompatible dimensions. However, we can take the *transpose* of `B`, i.e. flip the rows and columns, to make it compatible with `A` for matrix multiplication.

```
A @ np.transpose(B)
```

```
np.transpose(A) @ B
```

More generally, you can *reshape* a numpy array into a new shape, provided it is compatible with the number of elements in the original array.

```
D = rng.randint(0,5, (4,4))
D
```

```
D.reshape(8,2)
```

```
D.reshape(1,16)
```

This can also be used to cast a vector into a matrix.

```
e = np.arange(20)
E = e.reshape(5,4)
E
```

One thing to note in all the reshaping operations above is that the new array takes elements of the old array **by row**. See the examples above to convince yourself of that.

3.2.3.3. Statistical operations on arrays

You can sum all the elements of a matrix using `sum`. You can also sum along rows or along columns by adding an argument to the `sum` function.

```
A = rng.normal(0, 1, (4,2))
A
```

```
A.sum()
```

You can sum along rows (i.e., down columns) with the option `axis = 0`

```
A.sum(axis=0)
```

You can sum along columns (i.e., across rows) with `axis = 1`.

```
A.sum(axis=1)
```

Of course, you can use the usual function calls: `np.sum(A, axis = 1)`

We can also find the minimum and maximum values.

```
A.min(axis = 0)
```

```
A.max(axis = 0)
```

We can also find the **position** where the minimum and maximum values occur.

```
A.argmin(axis=0)
```

```
A.argmax(axis=0)
```

We can sort arrays and also find the indices which will result in the sorted array. I'll demonstrate this for a vector, where it is more relevant

```
a = rng.randint(0,10, 8)
a
```

```
np.sort(a)
```

```
np.argsort(a)
```

```
a[np.argsort(a)]
```

`np.argsort` can also help you find the 2nd smallest or 3rd largest value in an array, too.

```
ind_2nd_smallest = np.argsort(a)[1]
a[ind_2nd_smallest]
```

```
ind_3rd_largest = np.argsort(a)[-3]
a[ind_3rd_largest]
```

You can also sort strings in this way.

```
m = np.array(['Aram','Raymond','Elizabeth','Donald','Harold'])
np.sort(m)
```

If you want to sort arrays **in place**, you can use the `sort` function in a different way.

```
m.sort()
m
```

3.2.3.4. Putting arrays together

We can put arrays together by row or column, provided the corresponding axes have compatible lengths.

```
A = rng.randint(0,5, (3,5))
B = rng.randint(0,5, (3,5))
print('A = ', A)
print('B = ', B)
```

```
np.hstack((A,B))
```

```
np.vstack((A,B))
```

Note that both `hstack` and `vstack` take a **tuple** of arrays as input.

3.2.3.5. Logical/Boolean operations

You can query a matrix to see which elements meet some criterion. In this example, we'll see which elements are negative.

```
A < 0
```

This is called **masking**, and is useful in many contexts.

We can extract all the negative elements of A using

```
A[A<0]
```

This forms a 1-d array. You can also count the number of elements that meet the criterion

```
np.sum(A<0)
```

Since the entity `A<0` is a matrix as well, we can do row-wise and column-wise operations as well.

3.2.4. Beware of copies

One has to be a bit careful with copying objects in Python. By default, if you just assign one object to a new name, it does a *shallow copy*, which means that both names point to the same memory. So if you change something in the original, it also changes in the new copy.

```
A[0,:]
```

```
A1 = A
A1[0,0] = 4
A[0,0]
```

To actually create a copy that is not linked back to the original, you have to make a *deep copy*, which creates a new space in memory and a new pointer, and copies the original object to the new memory location

```
A1 = A.copy()
A1[0,0] = 6
A[0,0]
```

You can also replace sub-matrices of a matrix with new data, provided that the dimensions are compatible. (Make sure that the sub-matrix we are replacing below truly has 2 rows and 2 columns, which is what `np.eye(2)` will produce)

```
A[:2,:2] = np.eye(2)
A
```

3.2.4.1. Reducing matrix dimensions

Sometimes the output of some operation ends up being a matrix of one column or one row. We can reduce it to become a vector. There are two functions that can do that, `flatten` and `ravel`.

```
A = rng.randint(0,5, (5,1))
A
```

```
A.flatten()
```

```
A.ravel()
```

So why two functions? I'm not sure, but they do different things behind the scenes. `flatten` creates a **copy**, i.e. a new array disconnected from `A`. `ravel` creates a **view**, so a representation of the original array. If you then changed a value after a `ravel` operation, you would also change it in the original array; if you did this after a `flatten` operation, you would not.

3.2.5. Broadcasting in Python

Python deals with arrays in an interesting way, in terms of matching up dimensions of arrays for arithmetic operations. There are 3 rules:

1. If two arrays differ in the number of dimensions, the shape of the smaller array is padded with 1s on its *left* side
2. If the shape doesn't match in any dimension, the array with shape = 1 in that dimension is stretched to match the others' shape
3. If in any dimension the sizes disagree and none of the sizes are 1, then an error is generated

```
A = rng.normal(0,1,(4,5))
B = rng.normal(0,1,5)
```

```
A.shape
```

```
B.shape
```

```
A - B
```

B is 1-d, A is 2-d, so B's shape is made into (1,5) (added to the left). Then it is repeated into 4 rows to make it's shape (4,5), then the operation is performed. This means that we subtract the first element of B from the first column of A, the second element of B from the second column of A, and so on.

You can be explicit about adding dimensions for broadcasting by using `np.newaxis`.

```
B[np.newaxis,:].shape
```

```
B[:,np.newaxis].shape
```

3.2.5.1. An example (optional, intermediate/advanced))

This can be very useful, since these operations are faster than for loops. For example:

```
d = rng.random_sample((10,2))
d
```

We want to find the Euclidean distance (the sum of squared differences) between the points defined by the rows. This should result in a 10x10 distance matrix

```
d.shape
```

```
d[np.newaxis,:,:]
```

creates a 3-d array with the first dimension being of length 1

```
d[np.newaxis,:,:].shape
```

```
d[:, np.newaxis,:]
```

creates a 3-d array with the 2nd dimension being of length 1

```
d[:,np.newaxis,:].shape
```

Now for the trick, using broadcasting of arrays. These two arrays are incompatible without broadcasting, but with broadcasting, the right things get repeated to make things compatible

```
dist_sq = np.sum((d[:,np.newaxis,:] - d[np.newaxis,:,:]) ** 2)
```

```
dist_sq.shape
```

```
dist_sq
```

Whoops! we wanted a 10x10 matrix, not a scalar.

```
(d[:,np.newaxis,:] - d[np.newaxis,:,:]).shape
```

What we really want is the 10x10 distance matrix.

```
dist_sq = np.sum((d[:,np.newaxis,:] - d[np.newaxis,:,:]) ** 2, axis=2)
```

You can verify what is happening by creating `D = d[:,np.newaxis,:]-d[np.newaxis,:,:]` and then looking at `D[:,:,0]` and `D[:,:,1]`. These are the difference between each combination in the first and second columns of d, respectively. Squaring and summing along the 3rd axis then gives the sum of squared differences.

```
dist_sq
```

```
dist_sq.shape
```

```
dist_sq.diagonal()
```

3.2.6. Conclusions moving forward

It's important to understand numpy and arrays, since most data sets we encounter are rectangular. The notations and operations we saw in numpy will translate to data, except for the fact that data is typically heterogeneous, i.e., of different types. The problem with using numpy for modern data analysis is that if you have mixed data types, it will all be coerced to strings, and then you can't actually do any data analysis.

The solution to this issue (which is also present in Matlab) came about with the `pandas` package, which is the main workhorse of data science in Python

4. Pandas

(last updated 2020-05-18 10:19:20)

4.1. Introduction

`pandas` is the Python Data Analysis package. It allows for data ingestion, transformation and cleaning, and creates objects that can then be passed on to analytic packages like `statsmodels` and `scikit-learn` for modeling and packages like `matplotlib`, `seaborn`, and `plotly` for visualization.

`pandas` is built on top of numpy, so many numpy functions are commonly used in manipulating `pandas` objects.

> `pandas` is a pretty extensive package, and we'll only be able to cover some of its features. For more details, there is free online documentation at pandas.pydata.org. You can also look at the book "Python for Data Analysis (2nd edition)" by Wes McKinney, the original developer of the pandas package, for more details.

4.2. Starting pandas

As with any Python module, you have to "activate" `pandas` by using `import`. The "standard" alias for `pandas` is `pd`. We will also import `numpy`, since `pandas` uses some `numpy` functions in the workflows.

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

4.3. Data import and export

Most data sets you will work with are set up in tables, so are rectangular in shape. Think Excel spreadsheets. In `pandas` the structure that will hold this kind of data is a `DataFrame`. We can read external data into a `DataFrame` using one of many `read_*` functions. We can also write from a `DataFrame` to a variety of formats using `to_*` functions. The most common of these are listed below:

Format type	Description	reader	writer
text	CSV	read_csv	to_csv
	Excel	read_excel	to_excel
text	JSON	read_json	to_json

Format type	Description	reader	writer
binary	Feather	read_feather	to_feather
binary	SAS	read_sas	
SQL	SQL	read_sql	to_sql

We'll start by reading in the `mtcars` dataset stored as a CSV file

```
pd.read_csv('data/mtcars.csv')
```

This just prints out the data, but then it's lost. To use this data, we have to give it a name, so it's stored in Python's memory

```
mtcars = pd.read_csv('data/mtcars.csv')
```

> One of the big differences between a spreadsheet program and a programming language from the data science perspective is that you have to load data into the programming language. It's not "just there" like Excel. This is a good thing, since it allows the common functionality of the programming language to work across multiple data sets, and also keeps the original data set pristine. Excel users can run into problems and corrupt their data if they are not careful.

If we wanted to write this data set back out into an Excel file, say, we could do

```
mtcars.to_excel('data/mtcars.xlsx')
```

> You may get an error if you don't have the `openpyxl` package installed. You can easily install it from the Anaconda prompt using `conda install openpyxl` and following the prompts.

4.4. Exploring a data set

We would like to get some idea about this data set. There are a bunch of functions linked to the `DataFrame` object that help us in this. First we will use `head` to see the first 8 rows of this data set

```
mtcars.head(8)
```

This is our first look into this data. We notice a few things. Each column has a name, and each row has an *index*, starting at 0.

> If you're interested in the last N rows, there is a corresponding `tail` function

Let's look at the data types of each of the columns

```
mtcars.dtypes
```

This tells us that some of the variables, like mpg and disp, are floating point (decimal) numbers, several are integers, and make is an "object". The dtypes function borrows from numpy, where there isn't really a type for character or categorical variables. So most often, when you see "object" in the output of dtypes, you think it's a character or categorical variable.

We can also look at the data structure in a bit more detail.

```
mtcars.info()
```

This tells us that this is indeed a DataFrame, wth 12 columns, each with 32 valid observations. Each row has an index value ranging from 0 to 11. We also get the approximate size of this object in memory.

You can also quickly find the number of rows and columns of a data set by using shape, which is borrowed from numpy.

```
mtcars.shape
```

More generally, we can get a summary of each variable using the describe function

```
mtcars.describe()
```

These are usually the first steps in exploring the data.

4.5. Data structures and types

pandas has two main data types: Series and DataFrame. These are analogous to vectors and matrices, in that a Series is 1-dimensional while a DataFrame is 2-dimensional.

4.5.1. pandas.Series

The Series object holds data from a single input variable, and is required, much like numpy arrays, to be homogeneous in type. You can create Series objects from lists or numpy arrays quite easily

```
s = pd.Series([1,3,5,np.nan, 9, 13])
s
```

```
s2 = pd.Series(np.arange(1,20))
s2
```

You can access elements of a Series much like a dict

```
s2[4]
```

There is no requirement that the index of a Series has to be numeric. It can be any kind of scalar object

```
s3 = pd.Series(np.random.normal(0,1, (5,)), index = ['a','b','c','d','e'])
s3
```

```
s3['d']
```

```
s3['a':'d']
```

Well, slicing worked, but it gave us something different than expected. It gave us both the start **and** end of the slice, which is unlike what we've encountered so far!!

It turns out that in pandas, slicing by index actually does this. It is a discrepancy from numpy and Python in general that we have to be careful about.

You can extract the actual values into a numpy array

```
s3.to_numpy()
```

In fact, you'll see that much of pandas' structures are build on top of numpy arrays. This is a good thing, since you can take advantage of the powerful numpy functions that are built for fast, efficient scientific computing.

Making the point about slicing again,

```
s3.to_numpy()[0:3]
```

This is different from index-based slicing done earlier.

4.5.2. pandas.DataFrame

The `DataFrame` object holds a rectangular data set. Each column of a `DataFrame` is a `Series` object. This means that each column of a `DataFrame` must be comprised of data of the same type, but different columns can hold data of different types. This structure is extremely useful in practical data science. The invention of this structure was, in my opinion, transformative in making Python an effective data science tool.

4.5.2.1. Creating a DataFrame

The `DataFrame` can be created by importing data, as we saw in the previous section. It can also be created by a few methods within Python.

First, it can be created from a 2-dimensional numpy array.

```
rng = np.random.RandomState(25)
d1 = pd.DataFrame(rng.normal(0,1, (4,5)))
d1
```

You will notice that it creates default column names, that are merely the column number, starting from 0. We can also create the column names and row index (similar to the `Series` index we saw earlier) directly during creation.

```
d2 = pd.DataFrame(rng.normal(0,1, (4, 5)),
                  columns = ['A','B','C','D','E'],
                  index = ['a','b','c','d'])
d2
```

> We could also create a `DataFrame` from a list of lists, as long as things line up, just as we showed for numpy arrays. However, to me, other ways, including the `dict` method below, make more sense.

We can change the column names (which can be extracted and replaced with the `columns` attribute) and the index values (using the `index` attribute).

```
d2.columns
```

```
d2.columns = pd.Index(['V'+str(i) for i in range(1,6)]) # Index creates the right object
d2
```

Exercise: Can you explain what I did in the list comprehension above? The key points are understanding `str` and how I constructed the `range`.

```
d2.index = ['o1','o2','o3','o4']
d2
```

You can also extract data from a homogeneous `DataFrame` to a numpy array

```
d1.to_numpy()
```

> It turns out that you can use `to_numpy` for a non-homogeneous `DataFrame` as well. numpy just makes it homogeneous by assigning each column the data type `object`. This also limits what you can do in numpy with the array and may require changing data types using the `astype` function. There is some more detail about the `object` data type in the Python Tools for Data Science (notebook, PDF) document.

The other easy way to create a `DataFrame` is from a `dict` object, where each component object is either a list or a numpy array, and is homogeneous in type. One exception is if a component is of size 1; then it is repeated to meet the needs of the `DataFrame`'s dimensions

```
df = pd.DataFrame({
    'A':3.,
    'B':rng.random_sample(5),
    'C': pd.Timestamp('20200512'),
    'D': np.array([6] * 5),
    'E': pd.Categorical(['yes','no','no','yes','no']),
    'F': 'NIH'})
df
```

```
df.info()
```

We note that C is a date object, E is a category object, and F is a text/string object. pandas has excellent time series capabilities (having origins in FinTech), and the `TimeStamp` function creates datetime objects which can be queried and manipulated in Python. We'll describe category data in the next section.

You can also create a `DataFrame` where each column is composed of composite objects, like lists and dicts, as well. This might have limited value in some settings, but may be useful in others. In particular, this allows capabilities like the *list-column* construct in R tibbles. For example,

```
pd.DataFrame({'list' :[[1,2],[3,4],[5,6]],
              'tuple' : [('a','b'), ('c','d'), ('e','f')],
               'set' : [{'A','B','C'}, {'D','E'}, {'F'}],
             'dicts' : [{'A': [1,2,3]}, {'B':[5,6,8]}, {'C': [3,9]}]})
```

4.5.2.2. Working with a DataFrame

You can extract particular columns of a `DataFrame` by name

```
df['E']
```

```
df['B']
```

There is also a shortcut for accessing single columns, using Python's dot (`.`) notation.

```
df.B
```

This notation can be more convenient if we need to perform operations on a single column. If we want to extract multiple columns, this notation will not work. Also, if we want to create new columns or replace existing columns, we need to use the array notation with the column name in quotes.

Let's look at slicing a `DataFrame`

4.5.2.3. Extracting rows and columns

There are two extractor functions in `pandas`:

- `loc` extracts by label (index label, column label, slice of labels, etc.
- `iloc` extracts by index (integers, slice objects, etc.

```
df2 = pd.DataFrame(rng.randint(0,10, (5,4)),
                   index = ['a','b','c','d','e'],
                   columns = ['one','two','three','four'])
df2
```

First, let's see what naively slicing this `DataFrame` does.

```
df2['one']
```

Ok, that works. It grabs one column from the dataset. How about the dot notation?

```
df2.one
```

Let's see what this produces.

```
type(df2.one)
```

So this is a series, so we can potentially do slicing of this series.

```
df2.one['b']
```

```
df2.one['b':'d']
```

```
df2.one[:3]
```

Ok, so we have all the `Series` slicing available. The problem here is in semantics, in that we are grabbing one column and then slicing the rows. That doesn't quite work with our sense that a `DataFrame` is a rectangle with rows and columns, and we tend to think of rows, then columns.

Let's see if we can do column slicing with this.

```
df2[:'two']
```

That's not what we want, of course. It's giving back the entire data frame. We'll come back to this.

```
df2[['one','three']]
```

That works correctly though. We can give a list of column names. Ok.

How about row slices?

```
#df2['a'] # Doesn't work
df2['a':'c']
```

Ok, that works. It slices rows, but includes the largest index, like a `Series` but unlike `numpy` arrays.

```
df2[0:2]
```

Slices by location work too, but use the `numpy` slicing rules.

This entire extraction method becomes confusing. Let's simplify things for this, and then move on to more consistent ways to extract elements of a `DataFrame`. Let's agree on two things. If we're going the direct extraction route,

1. We will extract single columns of a `DataFrame` with `[]` or `.`, i.e., `df2['one']` or `df2.one`
2. We will extract slices of rows of a `DataFrame` using location only, i.e., `df2[:3]`.

For everything else, we'll use two functions, `loc` and `iloc`.

- `loc` extracts elements like a matrix, using index and columns
- `iloc` extracts elements like a matrix, using location

```
df2.loc[:,'one':'three']
```

```
df2.loc['a':'d',:]
```

```
df2.loc['b', 'three']
```

So `loc` works just like a matrix, but with `pandas` slicing rules (include largest index)

```
df2.iloc[:,1:4]
```

```
df2.iloc[1:3,:]
```

```
df2.iloc[1:3, 1:4]
```

`iloc` slices like a matrix, but uses `numpy` slicing conventions (does **not** include highest index)

4.5.2.3.1. Boolean selection

We can also use tests to extract data from a `DataFrame`. For example, we can extract only rows where column labeled `one` is greater than 3.

```
df2[df2.one > 3]
```

We can also do composite tests. Here we ask for rows where `one` is greater than 3 and `three` is less than 9

```
df2[(df2.one > 3) & (df2.three < 9)]
```

4.5.2.3.2. query

`DataFrame`'s have a `query` method allowing selection using a Python expression

```
n = 10
df = pd.DataFrame(np.random.rand(n, 3), columns = list('abc'))
df
```

```
df[(df.a < df.b) & (df.b < df.c)]
```

We can equivalently write this query as

```
df.query('(a < b) & (b < c)')
```

4.5.3. Categorical data

pandas provides a `Categorical` function and a `category` object type to Python. This type is analogous to the `factor` data type in R. It is meant to address categorical or discrete variables, where we need to use them in analyses. Categorical variables typically take on a small number of unique values, like gender, blood type, country of origin, race, etc.

You can create categorical `Series` in a couple of ways:

```
s = pd.Series(['a','b','c'], dtype='category')
```

```
df = pd.DataFrame({
    'A':3.,
    'B':rng.random_sample(5),
    'C': pd.Timestamp('20200512'),
    'D': np.array([6] * 5),
    'E': pd.Categorical(['yes','no','no','yes','no']),
    'F': 'NIH'})
df['F'].astype('category')
```

You can also create `DataFrame`'s where each column is categorical

```
df = pd.DataFrame({'A': list('abcd'), 'B': list('bdca')})
df_cat = df.astype('category')
df_cat.dtypes
```

You can explore categorical data in a variety of ways

```
df_cat['A'].describe()
```

```
df['A'].value_counts()
```

One issue with categories is that, if a particular level of a category is not seen before, it can create an error. So you can pre-specify the categories you expect

```
df_cat['B'] = pd.Categorical(list('aabb'), categories = ['a','b','c','d'])
df_cat['B'].value_counts()
```

4.5.4. Missing data

Both numpy and pandas allow for missing values, which are a reality in data science. The missing values are coded as `np.nan`. Let's create some data and force some missing values

```
df = pd.DataFrame(np.random.randn(5, 3), index = ['a','c','e', 'f','g'], columns = ['one
df['four'] = 20 # add a column named "four", which will all be 20
df['five'] = df['one'] > 0
df
```

```
df2 = df.reindex(['a','b','c','d','e','f','g'])
df2.style.applymap(lambda x: 'background-color:yellow', subset = pd.IndexSlice[['b','d']
```

The code above is creating new blank rows based on the new index values, some of which are present in the existing data and some of which are missing.

We can create *masks* of the data indicating where missing values reside in a data set.

```
df2.isna()
```

```
df2['one'].notna()
```

We can obtain complete data by dropping any row that has any missing value. This is called *complete case analysis*, and you should be very careful using it. It is *only* valid if we belive that the missingness is missing at random, and not related to some characteristic of the data or the data gathering process.

```
df2.dropna(how='any')
```

You can also fill in, or *impute*, missing values. This can be done using a single value..

```
out1 = df2.fillna(value = 5)

out1.style.applymap(lambda x: 'background-color:yellow', subset = pd.IndexSlice[['b','d'
```

or a computed value like a column mean

```
df3 = df2.copy()
df3 = df3.select_dtypes(exclude=[object])   # remove non-numeric columns
out2 = df3.fillna(df3.mean())  # df3.mean() computes column-wise means

out2.style.applymap(lambda x: 'background-color:yellow', subset = pd.IndexSlice[['b','d'
```

You can also impute based on the principle of *last value carried forward* which is common in time series. This means that the missing value is imputed with the previous recorded value.

```
out3 = df2.fillna(method = 'ffill') # Fill forward

out3.style.applymap(lambda x: 'background-color:yellow', subset = pd.IndexSlice[['b','d'
```

```
out4 = df2.fillna(method = 'bfill') # Fill backward

out4.style.applymap(lambda x: 'background-color:yellow', subset = pd.IndexSlice[['b','d'
```

4.6. Data transformation

4.6.1. Arithmetic operations

If you have a `Series` or `DataFrame` that is all numeric, you can add or multiply single numbers to all the elements together.

```
A = pd.DataFrame(np.random.randn(4,5))
print(A)
```

```
print(A + 6)
```

```
print(A * -10)
```

If you have two compatible (same dimension) numeric `DataFrames`, you can add, subtract, multiply and divide elementwise

```
B = pd.DataFrame(np.random.randn(4,5) + 4)
print(A + B)
```

```
print(A * B)
```

If you have a `Series` with the same number of elements as the number of columns of a `DataFrame`, you can do arithmetic operations, with each element of the `Series` acting upon each column of the `DataFrame`

```
c = pd.Series([1,2,3,4,5])
print(A + c)
```

```
print(A * c)
```

This idea can be used to standardize a dataset, i.e. make each column have mean 0 and standard deviation 1.

```
means = A.mean(axis=0)
stds = A.std(axis = 0)

(A - means)/stds
```

4.6.2. Concatenation of data sets

Let's create some example data sets

```
df1 = pd.DataFrame({'A': ['a'+str(i) for i in range(4)],
    'B': ['b'+str(i) for i in range(4)],
    'C': ['c'+str(i) for i in range(4)],
    'D': ['d'+str(i) for i in range(4)]})

df2 =  pd.DataFrame({'A': ['a'+str(i) for i in range(4,8)],
    'B': ['b'+str(i) for i in range(4,8)],
    'C': ['c'+str(i) for i in range(4,8)],
    'D': ['d'+str(i) for i in range(4,8)]})
df3 =  pd.DataFrame({'A': ['a'+str(i) for i in range(8,12)],
    'B': ['b'+str(i) for i in range(8,12)],
    'C': ['c'+str(i) for i in range(8,12)],
    'D': ['d'+str(i) for i in range(8,12)]})
```

We can concatenate these `DataFrame` objects by row

```
row_concatenate = pd.concat([df1, df2, df3])
print(row_concatenate)
```

This stacks the dataframes together. They are literally stacked, as is evidenced by the index values being repeated.

This same exercise can be done by the `append` function

```
df1.append(df2).append(df3)
```

Suppose we want to append a new row to `df1`. Lets create a new row.

```
new_row = pd.Series(['n1','n2','n3','n4'])
pd.concat([df1, new_row])
```

That's a lot of missing values. The issue is that the we don't have column names in the `new_row`, and the indices are the same, so pandas tries to append it my making a new column. The solution is to make it a `DataFrame`.

```
new_row = pd.DataFrame([['n1','n2','n3','n4']], columns = ['A','B','C','D'])
print(new_row)
```

```
pd.concat([df1, new_row])
```

or

```
df1.append(new_row)
```

4.6.2.1. Adding columns

```
pd.concat([df1,df2,df3], axis = 1)
```

The option `axis=1` ensures that concatenation happens by columns. The default value `axis = 0` concatenates by rows.

Let's play a little game. Let's change the column names of `df2` and `df3` so they are not the same as `df1`.

```
df2.columns = ['E','F','G','H']
df3.columns = ['A','D','F','H']
pd.concat([df1,df2,df3])
```

Now pandas ensures that all column names are represented in the new data frame, but with missing values where the row indices and column indices are mismatched. Some of this can be avoided by only joining on common columns. This is done using the `join` option ir `concat`. The default value is 'outer', which is what you see. above

```
pd.concat([df1, df3], join = 'inner')
```

You can do the same thing when joining by rows, using `axis = 0` and `join="inner"` to only join on rows with matching indices. Reminder that the indices are just labels and happen to be the row numbers by default.

4.6.3. Merging data sets

For this section we'll use a set of data from a survey, also used by Daniel Chen in "Pandas for Everyone"

```
person = pd.read_csv('data/survey_person.csv')
site = pd.read_csv('data/survey_site.csv')
survey = pd.read_csv('data/survey_survey.csv')
visited = pd.read_csv('data/survey_visited.csv')
```

```
print(person)
```

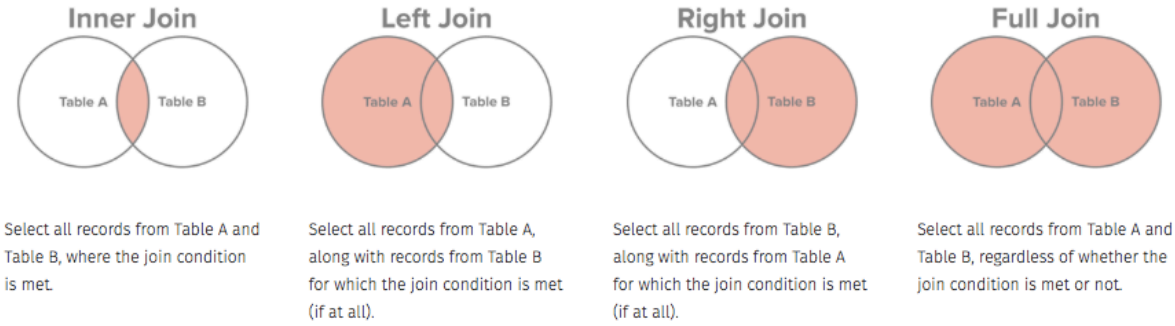
```
print(site)
```

```
print(survey)
```

```
print(visited)
```

There are basically four kinds of joins:

pandas	R	SQL	Description
left	left_join	left outer	keep all rows on left
right	right_join	right outer	keep all rows on right
outer	outer_join	full outer	keep all rows from both
inner	inner_join	inner	keep only rows with common keys



The terms `left` and `right` refer to which data set you call first and second respectively.

We start with an left join

```
s2v_merge = survey.merge(visited, left_on = 'taken',right_on = 'ident', how = 'left')
```

```
print(s2v_merge)
```

Here, the left dataset is `survey` and the right one is `visited`. Since we're doing a left join, we keed all the rows from `survey` and add columns from `visited`, matching on the common key, called "taken" in one dataset and "ident" in the other. Note that the rows of `visited` are repeated as needed to line up with all the rows with common "taken" values.

We can now add location information, where the common key is the site code

```
s2v2loc_merge = s2v_merge.merge(site, how = 'left', left_on = 'site', right_on = 'name')
print(s2v2loc_merge)
```

Lastly, we add the person information to this dataset.

```
merged = s2v2loc_merge.merge(person, how = 'left', left_on = 'person', right_on = 'ident
print(merged.head())
```

You can merge based on multiple columns as long as they match up.

```
ps = person.merge(survey, left_on = 'ident', right_on = 'person')
vs = visited.merge(survey, left_on = 'ident', right_on = 'taken')
print(ps)
```

```
print(vs)
```

```
ps_vs = ps.merge(vs,
                 left_on = ['ident','taken', 'quant','reading'],
                 right_on = ['person','ident','quant','reading']) # The keys need to corr
ps_vs.head()
```

Note that since there are common column names, the merge appends _x and _y to denote which column came from the left and right, respectively.

4.6.4. Tidy data principles and reshaping datasets

The tidy data principle is a principle espoused by Dr. Hadley Wickham, one of the foremost R developers. Tidy data is a structure for datasets to make them more easily analyzed on computers. The basic principles are

- Each row is an observation
- Each column is a variable
- Each type of observational unit forms a table

  Tidy data is tidy in one way. Untidy data can be untidy in many ways

Let's look at some examples.

```
from glob import glob
filenames = sorted(glob('data/table*.csv')) # find files matching pattern. I know there
table1, table2, table3, table4a, table4b, table5 = [pd.read_csv(f) for f in filenames] #
```

This code imports data from 6 files matching a pattern. Python allows multiple assignments on the left of the =, and as each dataset is imported, it gets assigned in order to the variables on the left. In the second line I sort the file names so that they match the order in which I'm storing them in the 3rd line. The function `glob` does pattern-matching of file names.

The following tables refer to the number of TB cases and population in Afghanistan, Brazil and China in 1999 and 2000

```
print(table1)
```

```
print(table2)
```

```
print(table3)
```

```
print(table4a) # cases
```

```
print(table4b) # population
```

```
print(table5)
```

Exercise: Describe why and why not each of these datasets are tidy.

4.6.5. Melting (unpivoting) data

Melting is the operation of collapsing multiple columns into 2 columns, where one column is formed by the old column names, and the other by the corresponding values. Some columns may be kept fixed and their data are repeated to maintain the interrelationships between the variables.

We'll start with loading some data on income and religion in the US from the Pew Research Center.

```
pew = pd.read_csv('data/pew.csv')
print(pew.head())
```

This dataset is considered in "wide" format. There are several issues with it, including the fact that column headers have data. Those column headers are income groups, that should be a column by tidy principles. Our job is to turn this dataset into "long" format with a column for income group.

We will use the function `melt` to achieve this. This takes a few parameters:

- **id_vars** is a list of variables that will remain as is
- **value_vars** is a list of column nmaes that we will melt (or unpivot). By default, it will melt all columns not mentioned in id_vars
- **var_name** is a string giving the name of the new column created by the headers (default: `variable`)
- **value_name** is a string giving the name of the new column created by the values (default: `value`)

```
pew_long = pew.melt(id_vars = ['religion'], var_name = 'income_group', value_name = 'cou
print(pew_long.head())
```

4.6.6. Separating columns containing multiple variables

We will use an Ebola dataset to illustrate this principle

```
ebola = pd.read_csv('data/country_timeseries.csv')
print(ebola.head())
```

Note that for each country we have two columns – one for cases (number infected) and one for deaths. Ideally we want one column for country, one for cases and one for deaths.

The first step will be to melt this data sets so that the column headers in question from a column and the corresponding data forms a second column.

```
ebola_long = ebola.melt(id_vars = ['Date','Day'])
print(ebola_long.head())
```

We now need to split the data in the `variable` column to make two columns. One will contain the country name and the other either Cases or Deaths. We will use some string manipulation functions that we will see later to achieve this.

```
variable_split = ebola_long['variable'].str.split('_', expand=True) # split on the `_` c
print(variable_split[:5])
```

The `expand=True` option forces the creation of an `DataFrame` rather than a list

```
type(variable_split)
```

We can now concatenate this to the original data

```
variable_split.columns = ['status','country']

ebola_parsed = pd.concat([ebola_long, variable_split], axis = 1)

ebola_parsed.drop('variable', axis = 1, inplace=True) # Remove the column named "variabl

print(ebola_parsed.head())
```

4.6.7. Pivot/spread datasets

If we wanted to, we could also make two columns based on cases and deaths, so for each country and date you could easily read off the cases and deaths. This is achieved using the `pivot_table` function.

In the `pivot_table` syntax, `index` refers to the columns we don't want to change, `columns` refers to the column whose values will form the column names of the new columns, and `values` is the name of the column that will form the values in the pivoted dataset.

```
ebola_parsed.pivot_table(index = ['Date','Day', 'country'], columns = 'status', values =
```

This creates something called `MultiIndex` in the `pandas` `DataFrame`. This is useful in some advanced cases, but here, we just want a normal `DataFrame` back. We can achieve that by using the `reset_index` function.

```
ebola_parsed.pivot_table(index = ['Date','Day','country'], columns = 'status', values =
```

Pivoting is a 2-column to many-column operation, with the number of columns formed depending on the number of unique values present in the column of the original data that is entered into the `columns` argument of `pivot_table`

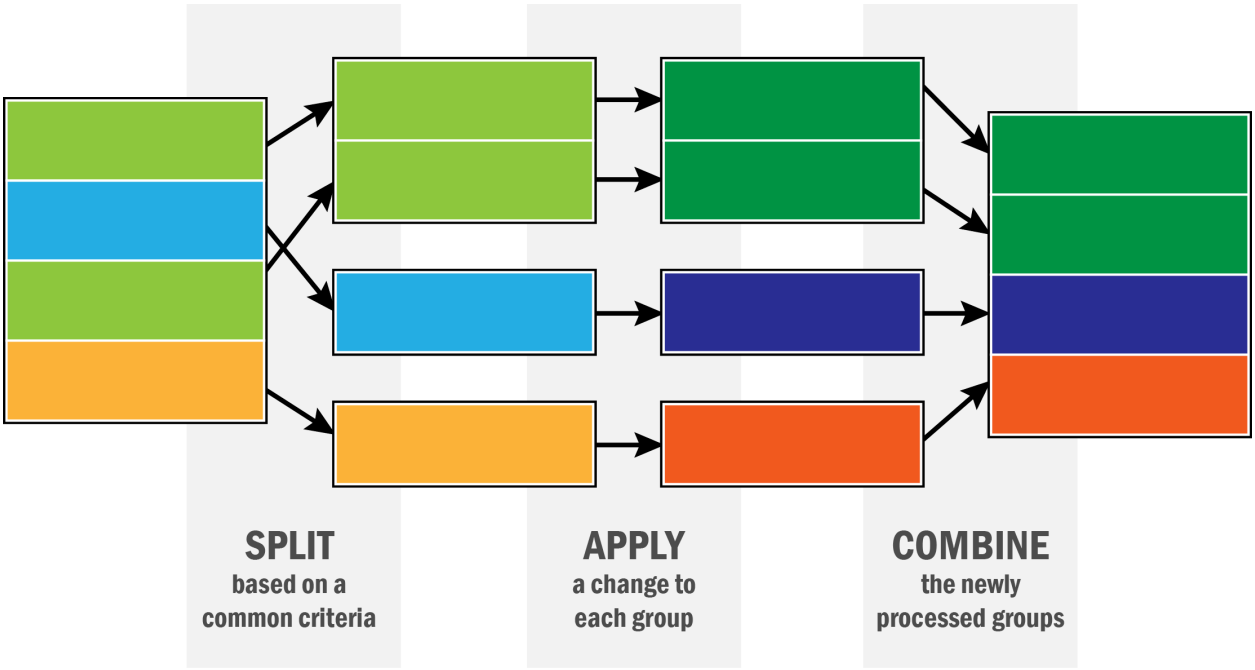
Exercise: Load the file `weather.csv` into Python and work on making it a tidy dataset. It requires melting and pivoting. The dataset comprises of the maximun and minimum temperatures recorded each day in 2010. There are lots of missing value. Ultimately we want columns for days of the month, maximum temperature and minimum tempearture along with the location ID, the year and the month.

4.7. Data aggregation and split-apply-combine

We'll use the Gapminder dataset for this section

```
df = pd.read_csv('data/gapminder.tsv', sep = '\t') # data is tab-separated, so we use `\
```

The paradigm we will be exploring is often called *split-apply-combine* or MapReduce or grouped aggregation. The basic idea is that you split a data set up by some feature, apply a recipe to each piece, compute the result, and then put the results back together into a dataset. This can be described in teh following schematic.



`pandas` is set up for this. It features the `groupby` function that allows the "split" part of the operation. We can then apply a function to each part and put it back together. Let's see how.

```
df.head()
```

```
f"This dataset has {len(df['country'].unique())} countries in it"
```

One of the variables in this dataset is life expectancy at birth, `lifeExp`. Suppose we want to find the average life expectancy of each country over the period of study.

```
df.groupby('country')['lifeExp'].mean()
```

So what's going on here? First, we use the `groupby` function, telling `pandas` to split the dataset up by values of the column `country`.

```
df.groupby('country')
```

`pandas` won't show you the actual data, but will tell you that it is a grouped dataframe object. This means that each element of this object is a `DataFrame` with data from one country.

```
df.groupby('country').ngroups
```

```
df.groupby('country').get_group('United Kingdom')
```

```
type(df.groupby('country').get_group('United Kingdom'))
```

```
avg_lifeexp_country = df.groupby('country').lifeExp.mean()
avg_lifeexp_country['United Kingdom']
```

```
df.groupby('country').get_group('United Kingdom').lifeExp.mean()
```

Let's look at if life expectancy has gone up over time, by continent

```
df.groupby(['continent','year']).lifeExp.mean()
```

```
avg_lifeexp_continent_yr = df.groupby(['continent','year']).lifeExp.mean().reset_index()
avg_lifeexp_continent_yr
```

```
type(avg_lifeexp_continent_yr)
```

The aggregation function, in this case `mean`, does both the "apply" and "combine" parts of the process.

We can do quick aggregations with `pandas`

```
df.groupby('continent').lifeExp.describe()
```

```
df.groupby('continent').nth(10) # Tenth observation in each group
```

You can also use functions from other modules, or your own functions in this aggregation work.

```
df.groupby('continent').lifeExp.agg(np.mean)
```

```
def my_mean(values):
    n = len(values)
    sum = 0
    for value in values:
        sum += value
    return(sum/n)

df.groupby('continent').lifeExp.agg(my_mean)
```

You can do many functions at once

```
df.groupby('year').lifeExp.agg([np.count_nonzero, np.mean, np.std])
```

You can also aggregate on different columns at the same time by passing a `dict` to the agg function

```
df.groupby('year').agg({'lifeExp': np.mean,'pop': np.median,'gdpPercap': np.median}).res
```

4.7.0.1. Transformation

You can do grouped transformations using this same method. We will compute the z-score for each year, i.e. we will substract the average life expectancy and divide by the standard deviation

```
def my_zscore(values):
    m = np.mean(values)
    s = np.std(values)
    return((values - m)/s)
```

```
df.groupby('year').lifeExp.transform(my_zscore)
```

```
df['lifeExp_z'] = df.groupby('year').lifeExp.transform(my_zscore)
```

```
df.groupby('year').lifeExp_z.mean()
```

4.7.0.2. Filter

We can split the dataset by values of one variable, and filter out those splits that fail some criterion. The following code only keeps countries with a population of at least 10 million at some point during the study period

```
df.groupby('country').filter(lambda d: d['pop'].max() > 10000000)
```

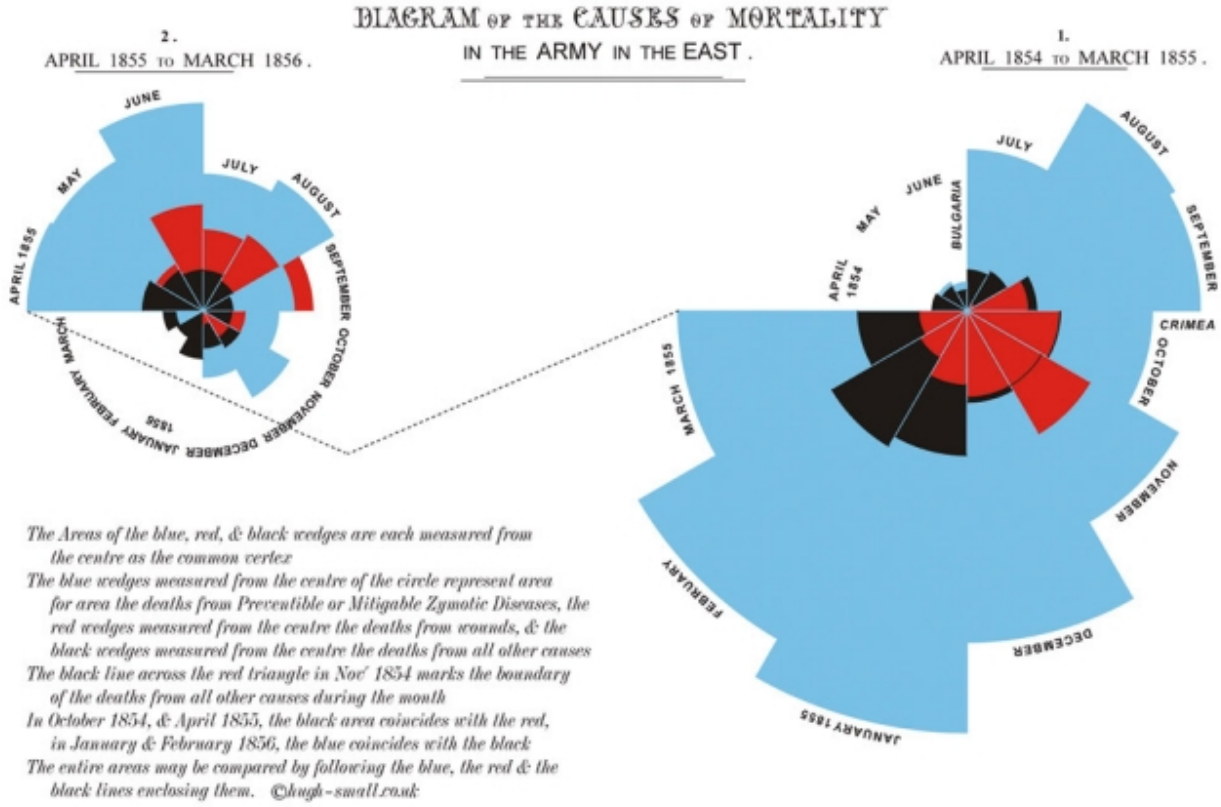
5. Data visualization using Python

5.1. Introduction

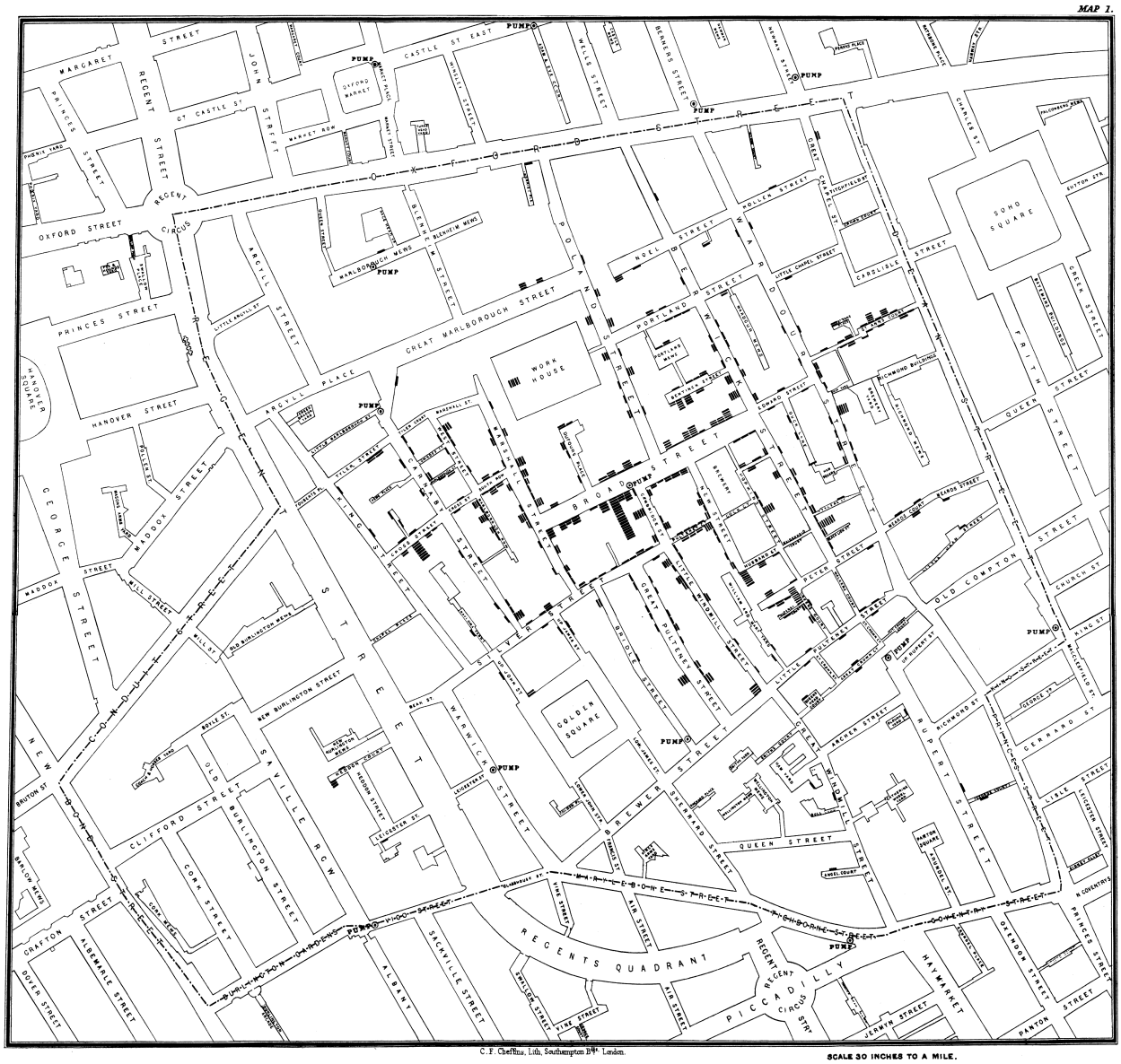
Data visualization is a basic task in data exploration and understanding. Humans are mainly visual creatures, and data visualization provides an opportunity to enhance communication of the story within the data. Often we find that data and the data-generating process is complex, and a visual representation of the data and our innate ability at pattern recognition can help reveal the complexities in a cognitively accessible way.

5.1.1. An example gallery

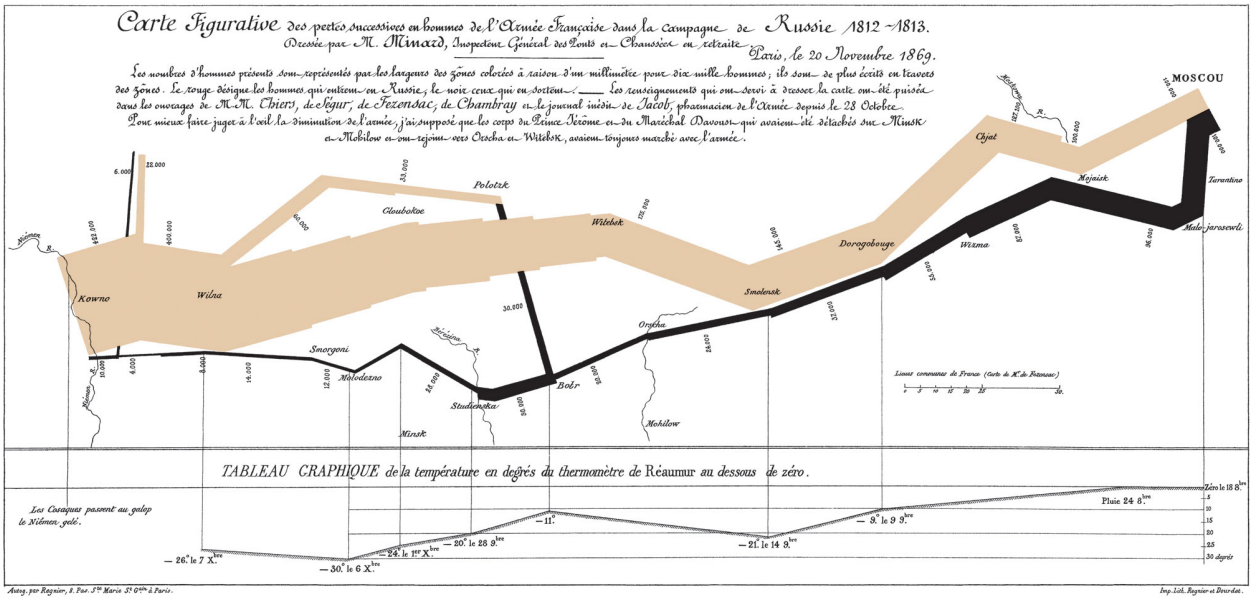
Data visualization has a long and storied history, from Florence Nightangle onwards. Dr. Nightangle was a pioneer in data visualization and developed the *rose plot* to represent causes of death in hospitals during the Crimean War.



John Snow, in 1854, famously visualized the cholera outbreak in London, which showed the geographic proximity of cholera prevalence with particular water wells.



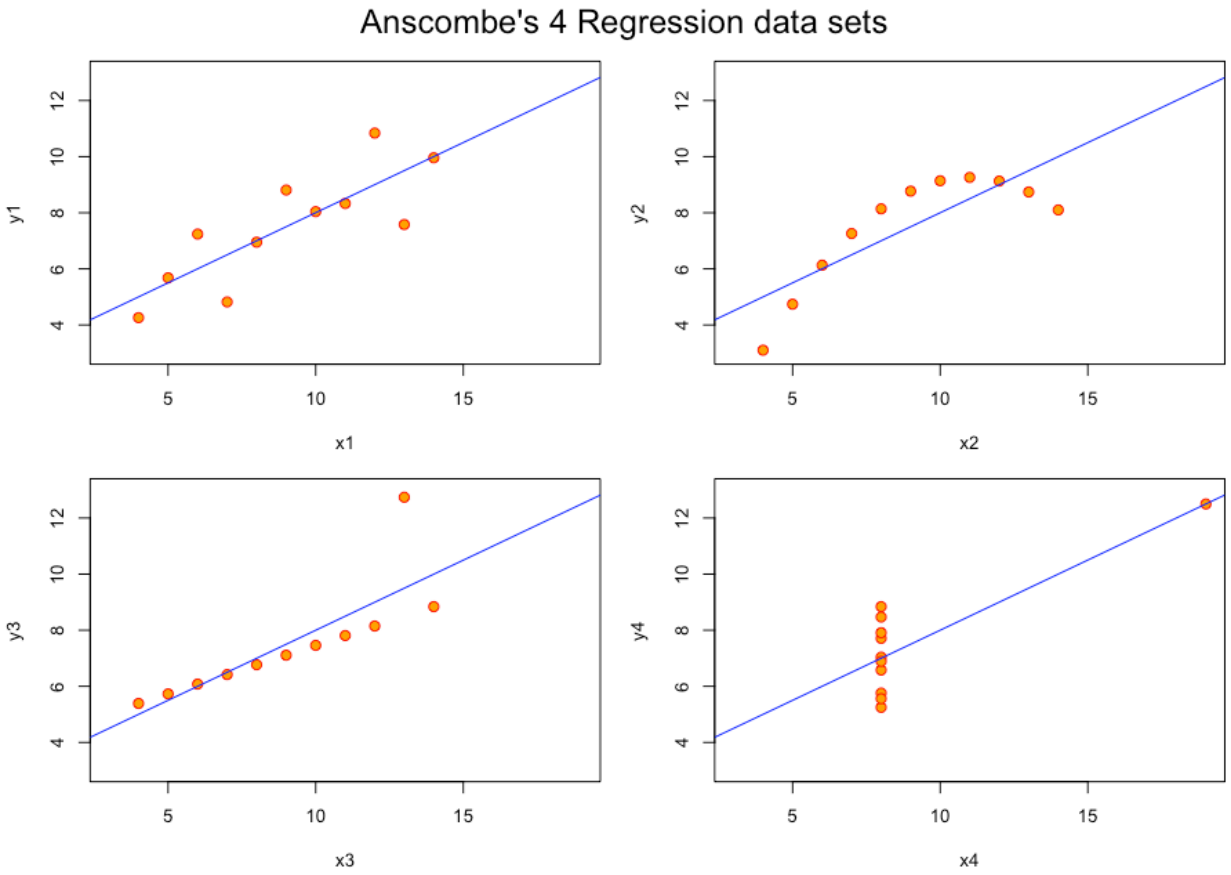
In one of the more famous visualizations, considered by many to be an optimal use of display ink and space, Minard visualized Napoleon's disastrous campaign to Russia



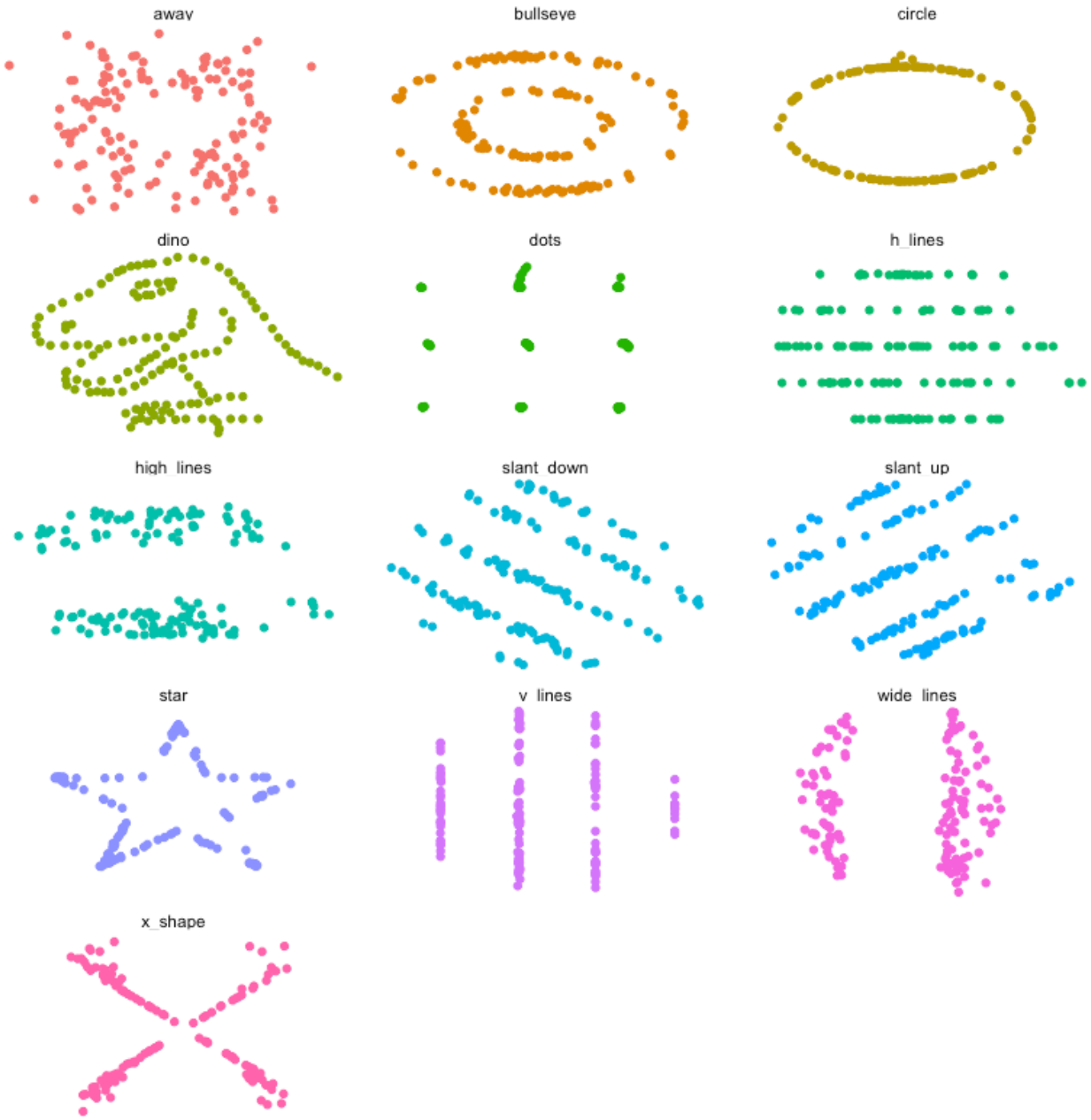
In more recent times, an employee at Facebook visualized all connections between users across the world, which clearly showed geographical associations with particular countries and regions.

5.1.2. Why visualize data?

We often rely on numerical summaries to help understand and distinguish datasets. In 1973, Anscombe published an influential set of 4 datasets, each with two variables and with the means, variances and correlations being identical. When you graphed these data, the differences in the datasets were clearly visible. This set is popularly known as Anscombe's quartet.



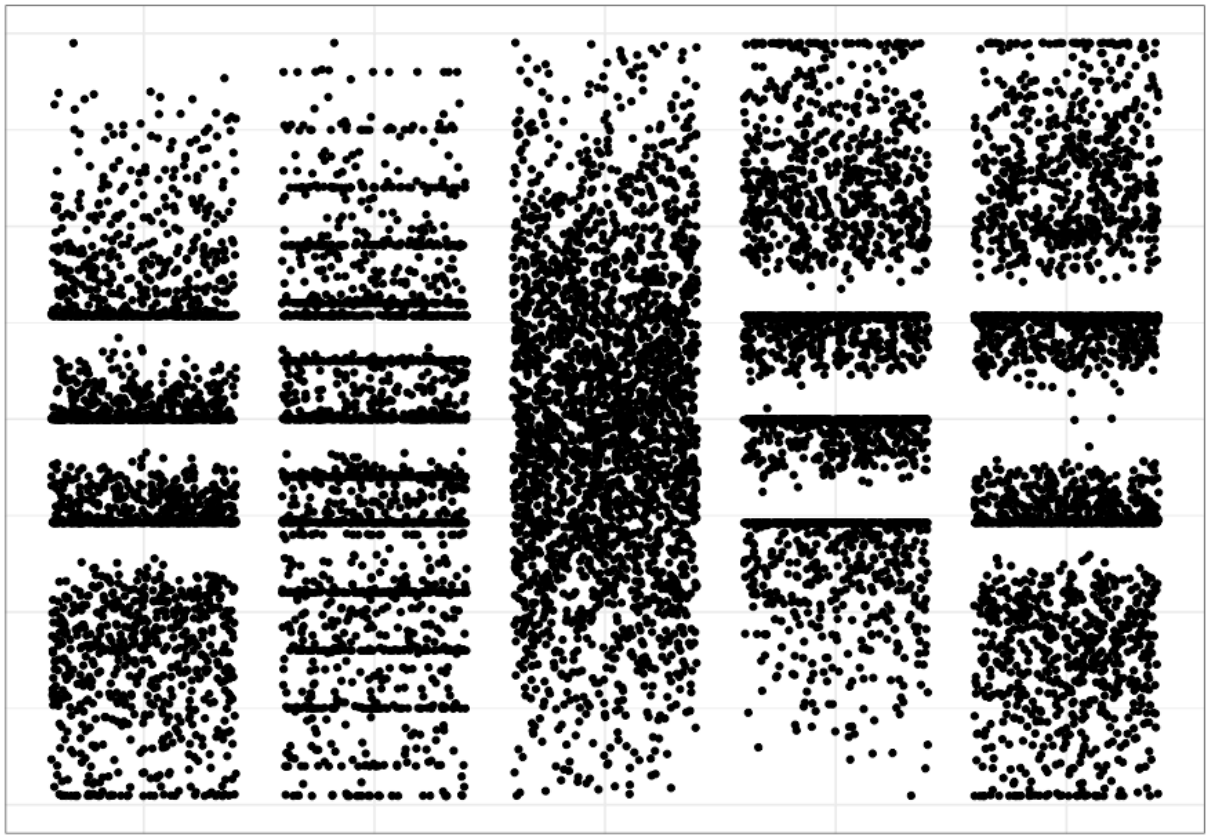
A more recent experiment in data construction by Matejka and Fitzmaurice (2017) started with a representation of a dinosaur and created 10 more bivariate datasets which all shared the same univariate means and variances and the same pairwise correlations.



These examples clarify the need for visualization to better understand relationships between variables.

Even when using statistical visualization techniques, one has to be careful. Not all visualizations can discriminate between statistical characteristics. This was also explored by Matejka and Fitzmaurice.

Strip plot



5.1.3. Conceptual ideas

5.1.3.1. Begin with the consumer in mind

- You have a deep understanding of the data you're presenting
- The person seeing the visualization **doesn't**
- Develop simpler visualizations first that are easier to explain

5.1.3.2. Tell a story

- Make sure the graphic is clear
- Make sure the main point you want to make "pops"

5.1.3.3. A matter of perception

- Color (including awareness of color deficiencies)
- Shape
- Fonts

5.1.3.4. Some principles

1. Data-ink ratio
2. No mental gymnastics
    1. The graphic should be self-evident
    2. Context should be clear
3. Is a graph the wrong choice?
4. Focus on the consumer

    See my slides for some more opinionated ideas

5.2. Plotting in Python

Let's take a very quick tour before we get into the weeds. We'll use the mtcars dataset as an exemplar dataset that we can import using `pandas`

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
sns.set_context('paper')
sns.set_style('white', {'font.family':'Futura', 'text.color':'1'})
```

```
sns.set_context('paper')
sns.set_style('white', {'font.family':'Futura Medium'})
```

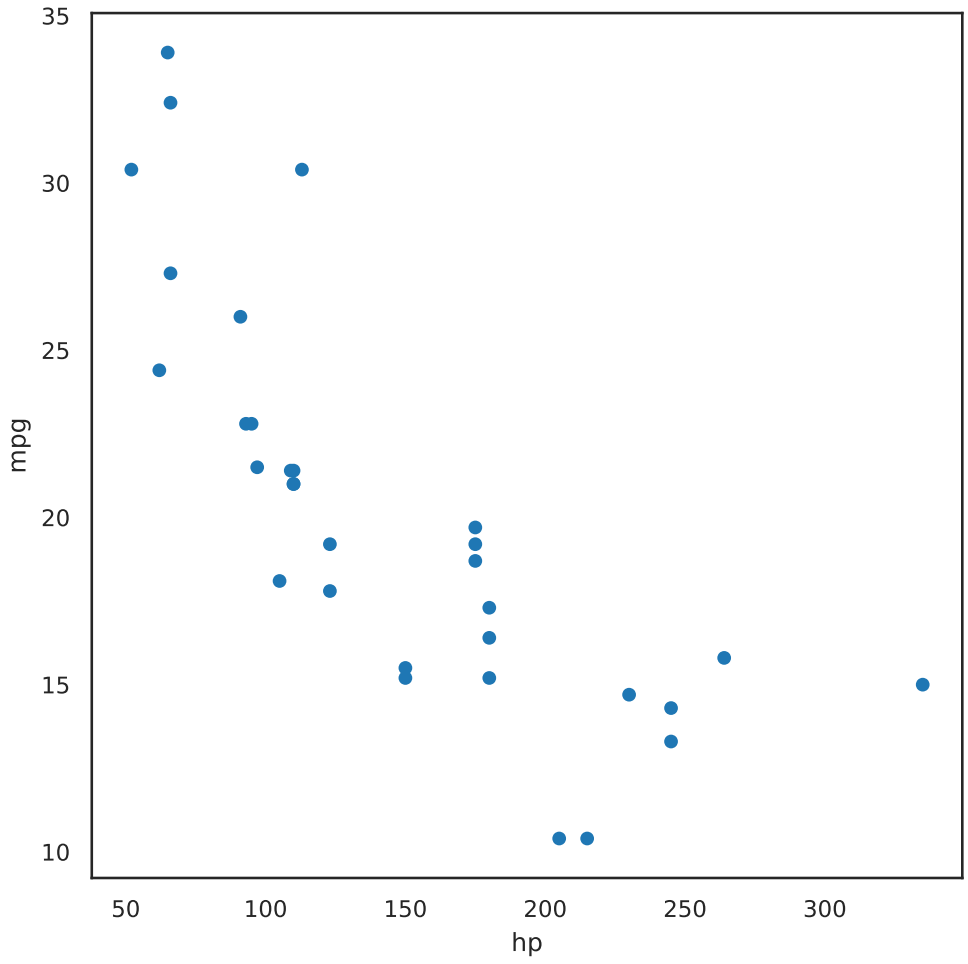
```
mtcars = pd.read_csv('data/mtcars.csv')
```

5.2.1. Static plots

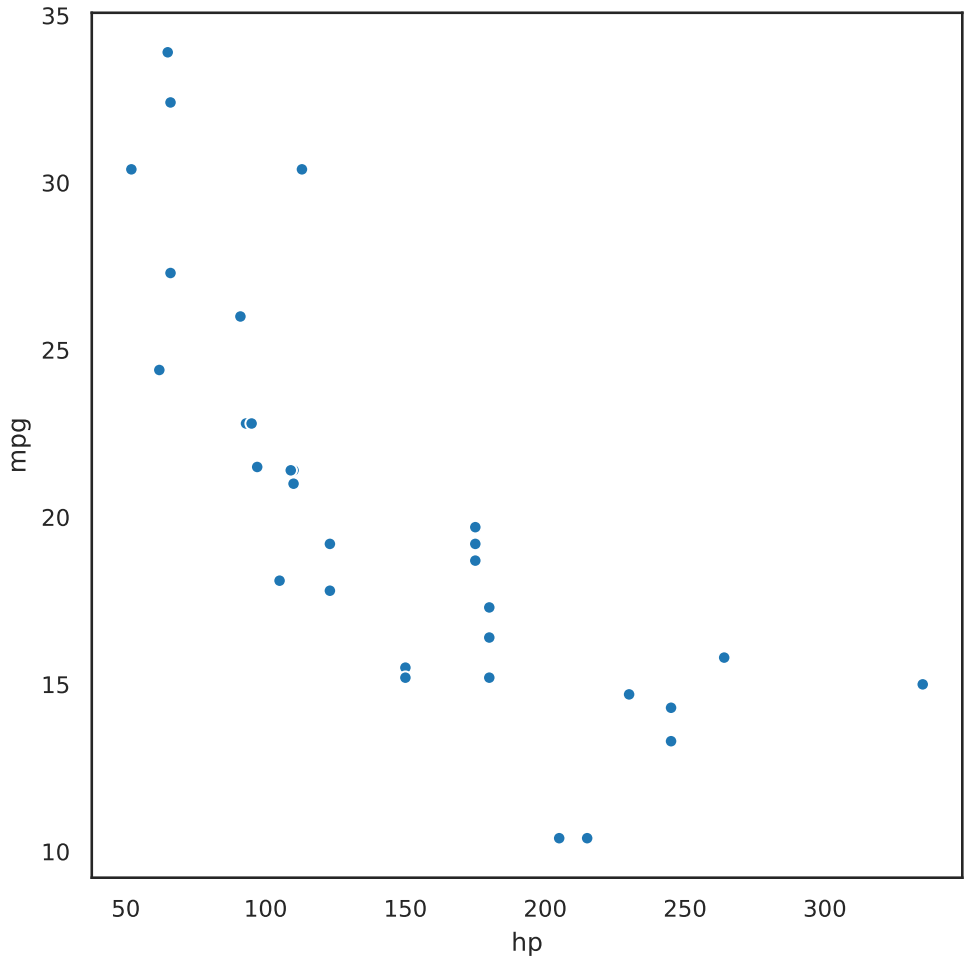
We will demonstrate plotting in what I'll call the `matplotlib` ecosystem. `matplotlib` is the venerable and powerful visualization package that was originally designed to emulate the Matlab plotting paradigm. It has since evolved and as become a bit more user-friendly. It is still quite granular and can facilitate a lot of custom plots once you become familiar with it. However, as a starting point, I think it's a bit much. We'll see a bit of what it can offer later.

We will consider two other options which are built on top of `matplotlib`, but are much more accessible. These are `pandas` and `seaborn`. The two packages have some different approaches, but both wrap `matplotlib` in higher-level code and decent choices so we don't need to get into the `matplotlib` trenches quite so much. We'll still call `matplotlib` in our code, since both these packages need it for some fine tuning. Both packages are also very much aligned to the `DataFrame` construct in `pandas`, so makes plotting a much more seamless experience.

```
mtcars.plot.scatter(x = 'hp', y = 'mpg');
plt.show()
# mtcars.plot(x = 'hp', y = 'mpg', kind = 'scatter');
```




```
sns.scatterplot(data = mtcars, x = 'hp', y = 'mpg');
plt.show()
```

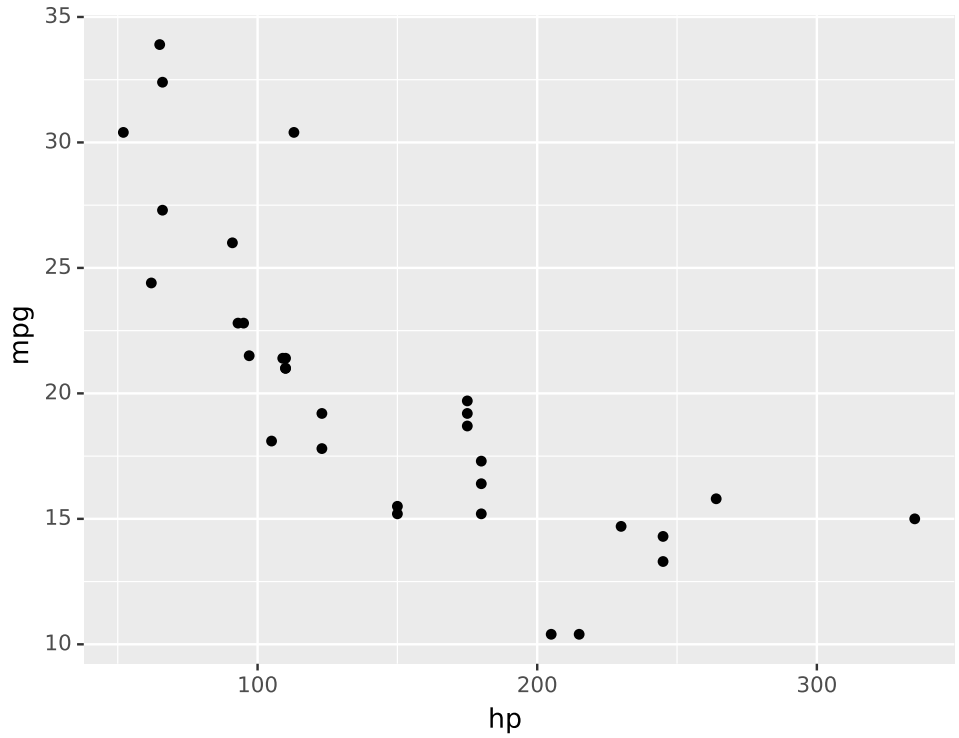


There are of course some other choices based on your background and preferences. For static plots, there are a couple of emulators of the popular R package `ggplot2`. These are `plotnine` and `ggplot`. `plotnine` seems a bit more developed and uses the `ggplot2` semantics of aesthetics and layers, with almost identical code syntax.

You can install `plotnine` using `conda`:

```
conda install -c conda-forge plotnine
```

```
from plotnine import *

(ggplot(mtcars) +
  aes(x = 'hp', y = 'mpg') +
  geom_point())
```



5.2.2. Dynamic or interactive plots

There are several Python packages that wrap around Javascript plotting libraries that are so popular in web-based graphics like D3 and Vega. Three that deserve mention are `plotly`, `bokeh`, and `altair`.

> If you actually want to experience the interactivity of the plots, please use the "Live notebooks" link in Canvas to run these notebooks. Otherwise, you can download the notebooks from the GitHub site and run them on your own computer.

`plotly` is a Python package developed by the company Plot.ly to interface with their interactive Javascript library either locally or via their web service. Plot.ly also develops an R package to interface with their products as well. It provides an intuitive syntax and ease of use, and is probably the more popular package for interactive graphics from both R and Python.

```
import plotly.express as px

fig = px.scatter(mtcars, x = 'hp', y = 'mpg')
fig.show()
```

`bokeh` is an interactive visualization package developed by Anaconda. It is quite powerful, but its code can be rather verbose and granular

```
from bokeh.plotting import figure, output_file
from bokeh.io import output_notebook, show
output_notebook()
p = figure()
p.xaxis.axis_label = 'Horsepower'
p.yaxis.axis_label = 'Miles per gallon'

p.circle(mtcars['hp'], mtcars['mpg'], size=10);

show(p)
```

`altair` that leverages ideas from Javascript plotting libraries and a distinctive code syntax that may appeal to some

```
import altair as alt

alt.Chart(mtcars).mark_point().encode(
    x='hp',
    y='mpg'
).interactive()
```

We won't focus on these dynamic packages in this workshop in the interests of time, but you can avail of several online resources for these.

Package	Resources
plotly	Fundamentals
bokeh	Tutorial
altair	Overview

5.3. Univariate plots

We will be introducing plotting and code from 3 modules: `matplotlib`, `seaborn` and `pandas`. As we go forth, you may ask the question, which one should I learn? Chris Moffitt has the following advice.

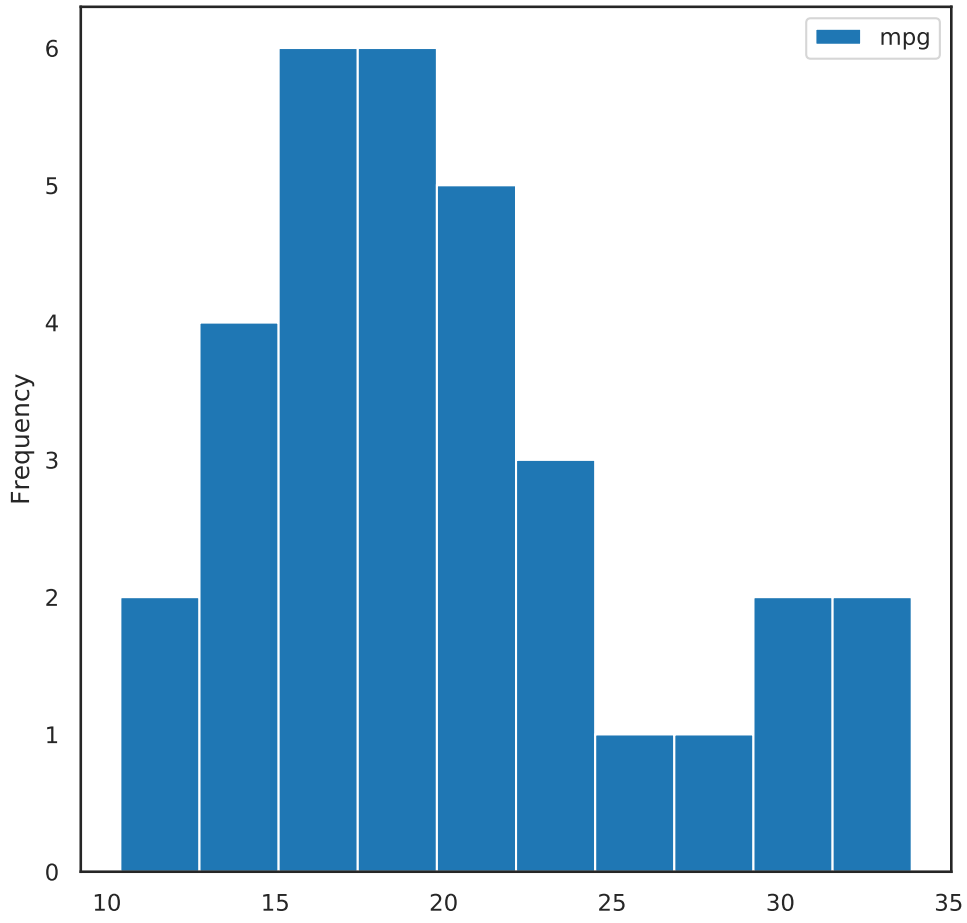
A pathway to learning (Chris Moffit)

1. Learn the basic matplotlib terminology, specifically what is a `Figure` and an `Axes`.
2. Always use the object-oriented interface. Get in the habit of using it from the start of your analysis. (*not really getting into this, but basically don't use the Matlab form I'll show at the end, if you don't have to*)
3. Start your visualizations with basic pandas plotting.
4. Use seaborn for the more complex statistical visualizations.
5. Use matplotlib to customize the pandas or seaborn visualization.

5.3.1. pandas

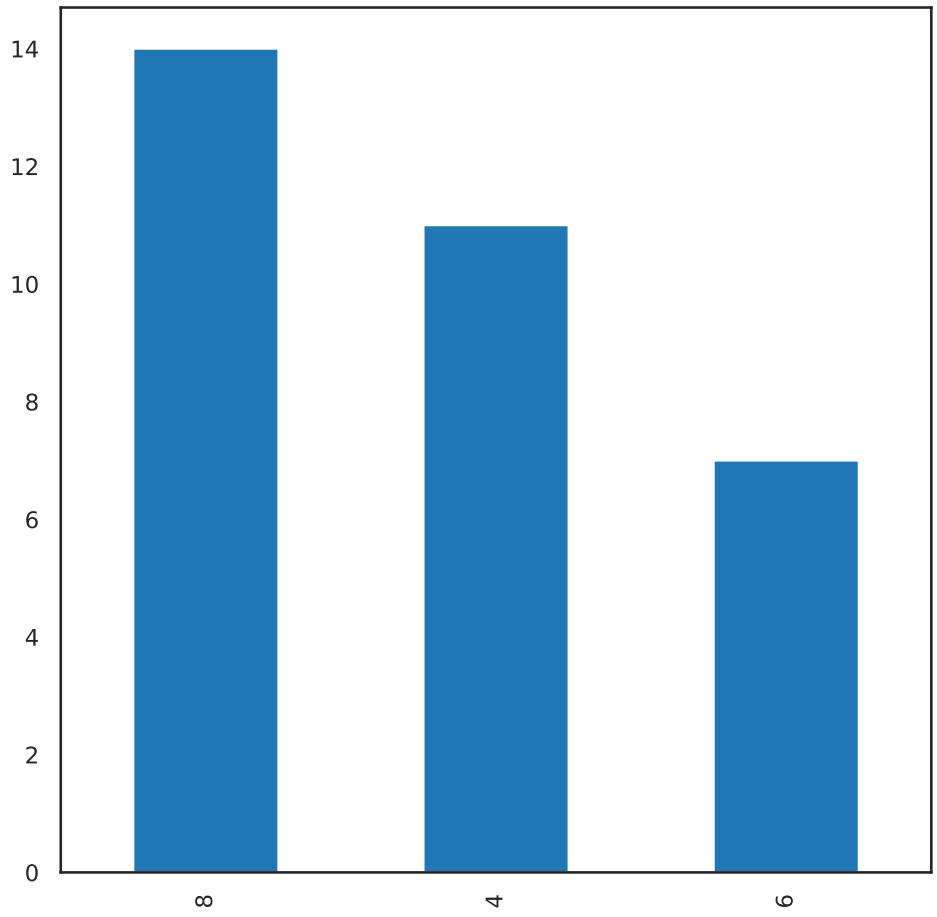
5.3.1.1. Histogram

```
mtcars.plot.hist(y = 'mpg');
plt.show()
# mtcars.plot(y = 'mpg', kind = 'hist')
#mtcars['mpg'].plot(kind = 'hist')
```



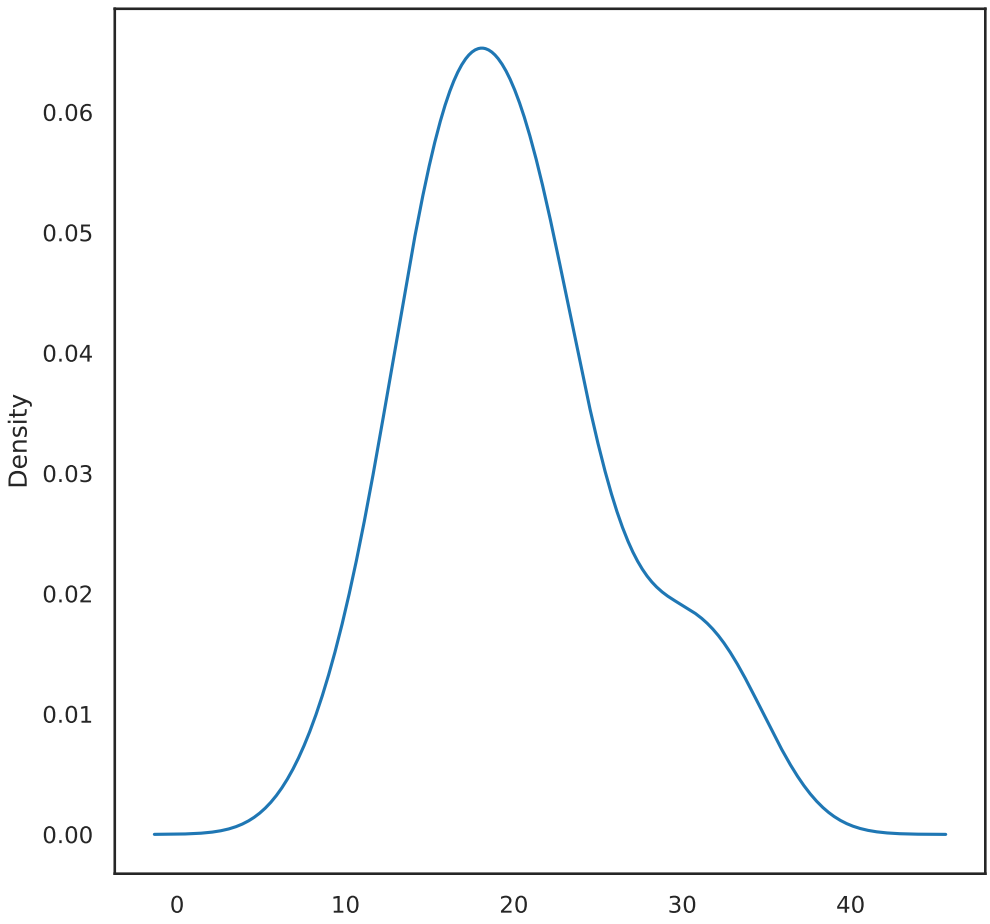
5.3.1.2. Bar plot

```
mtcars['cyl'].value_counts().plot.bar();
plt.show()
```



5.3.1.3. Density plot

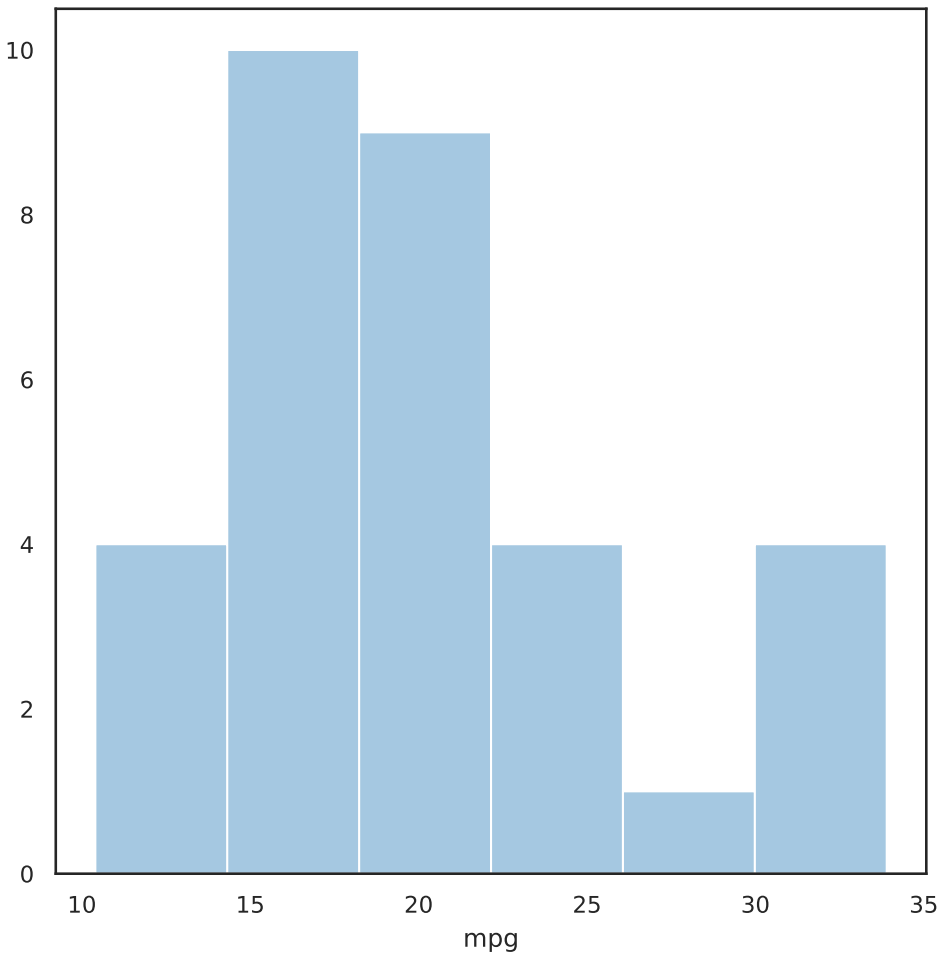
```
mtcars['mpg'].plot( kind = 'density');
plt.show()
```



5.3.2. seaborn

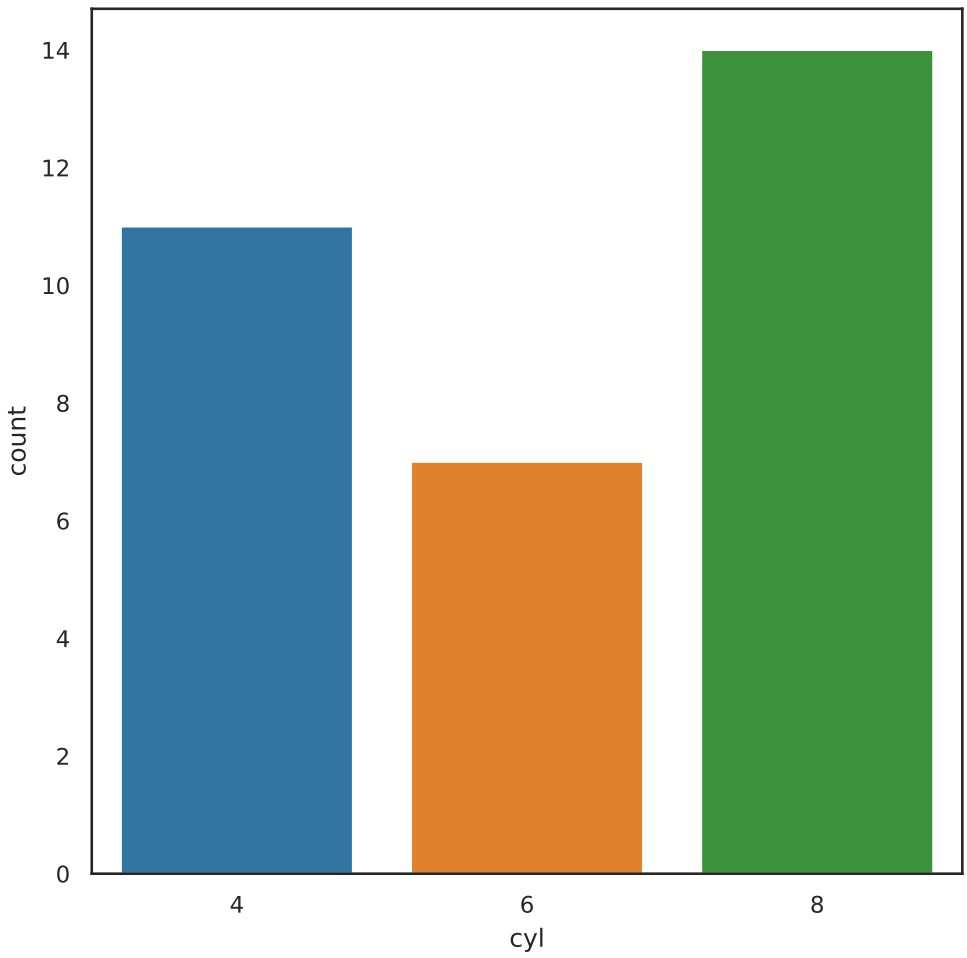
5.3.2.1. Histogram

```
ax = sns.distplot(mtcars['mpg'], kde=False);
plt.show()
```

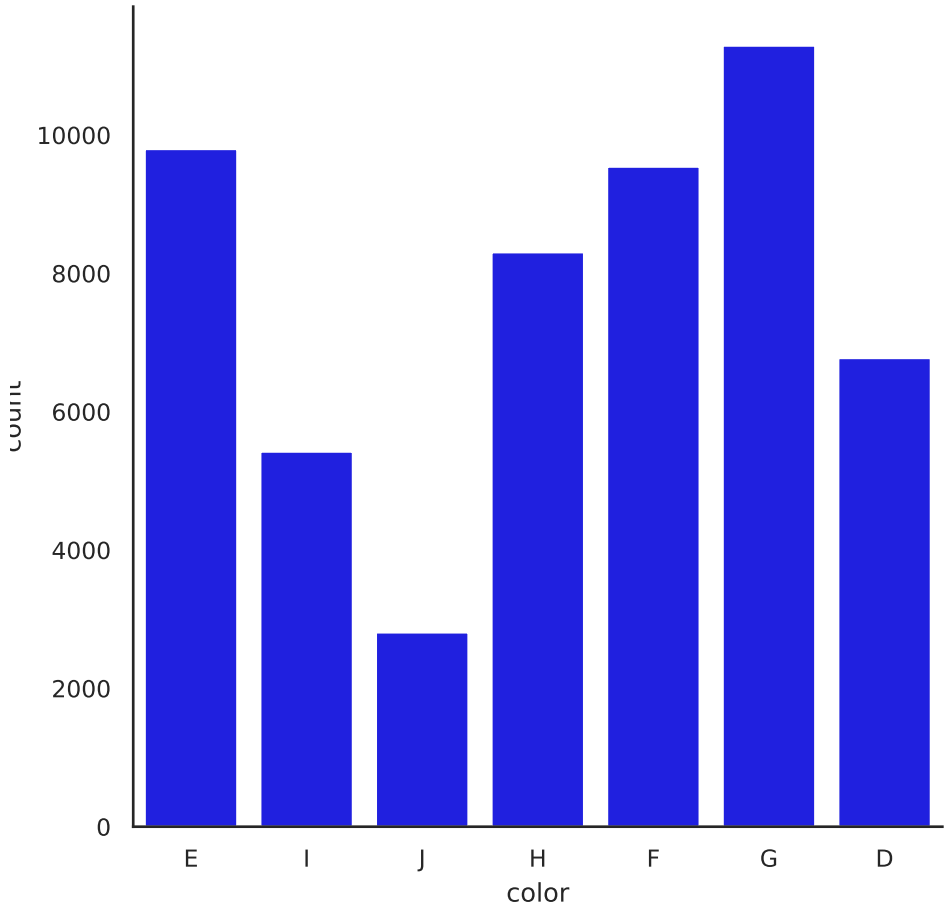


5.3.2.2. Bar plot

```
sns.countplot(data = mtcars, x = 'cyl');
plt.show()
```

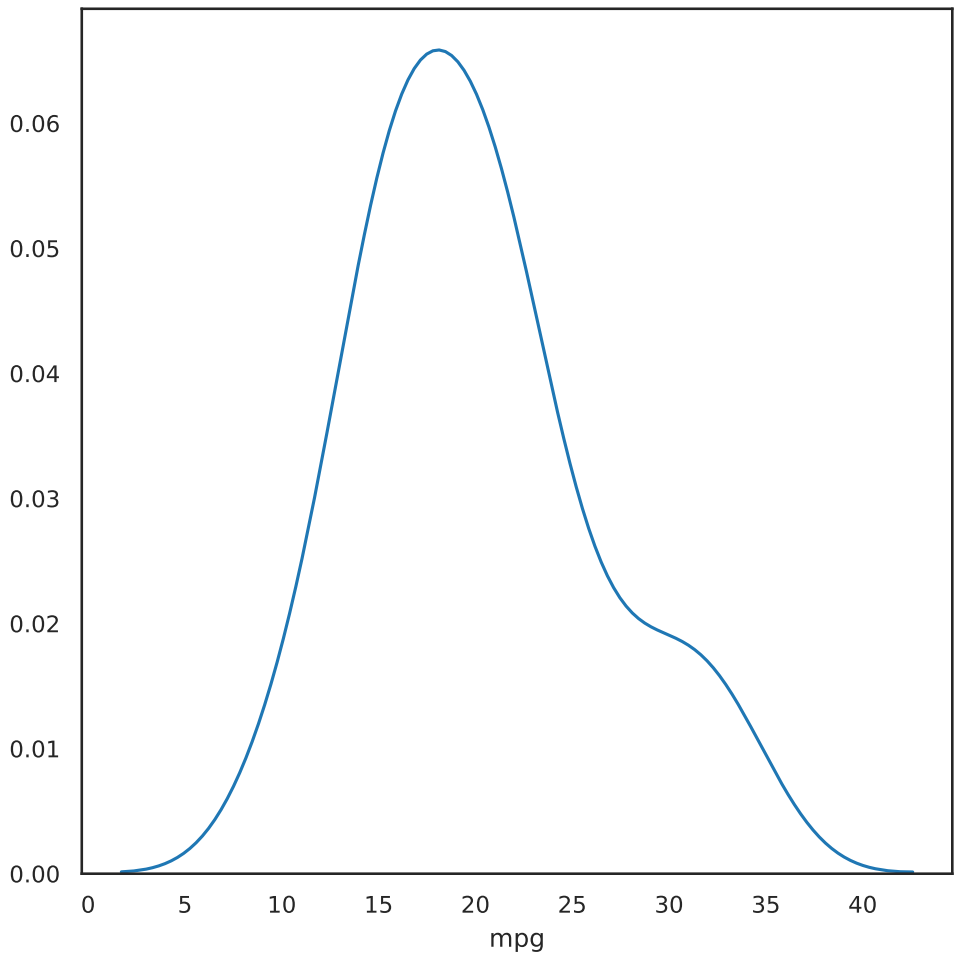



```
diamonds = pd.read_csv('data/diamonds.csv.gz')
ordered_colors = ['E','F','G','H','I','J']
sns.catplot(data = diamonds, x = 'color', kind = 'count', color = 'blue');
plt.show()
```



5.3.2.3. Density plot

```
sns.distplot(mtcars['mpg'], hist=False);
plt.show()
```

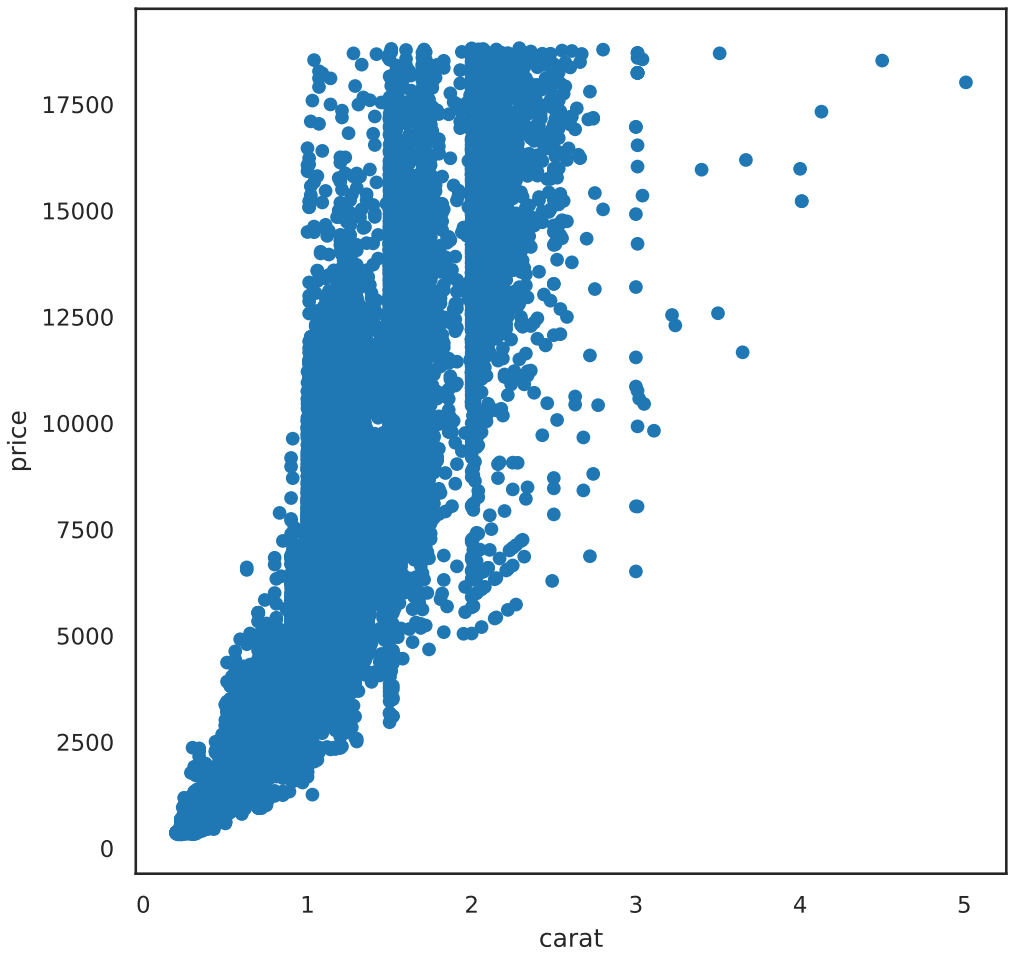


5.4. Bivariate plots

5.4.1. pandas

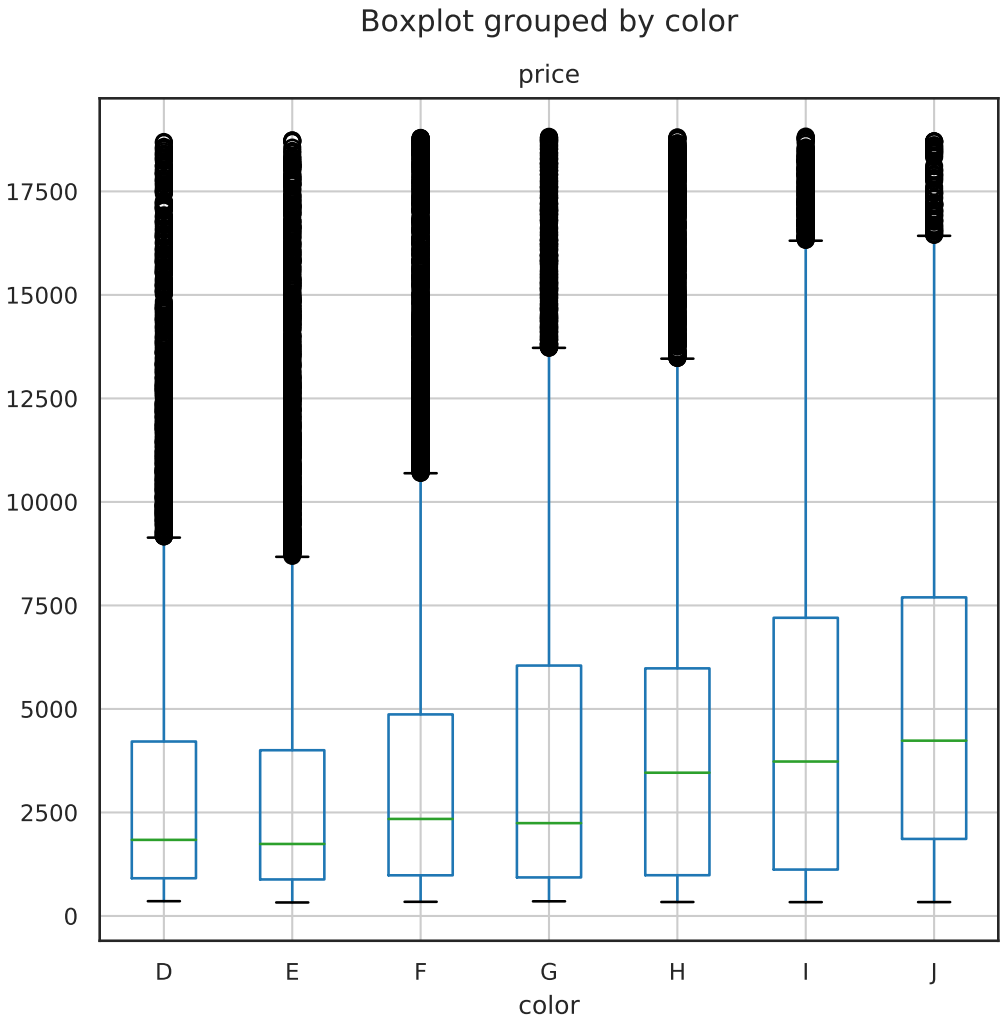
5.4.1.1. Scatter plot

```
diamonds = pd.read_csv('data/diamonds.csv.gz')
diamonds.plot(x = 'carat', y = 'price', kind = 'scatter');
plt.show()
```



5.4.1.2. Box plot

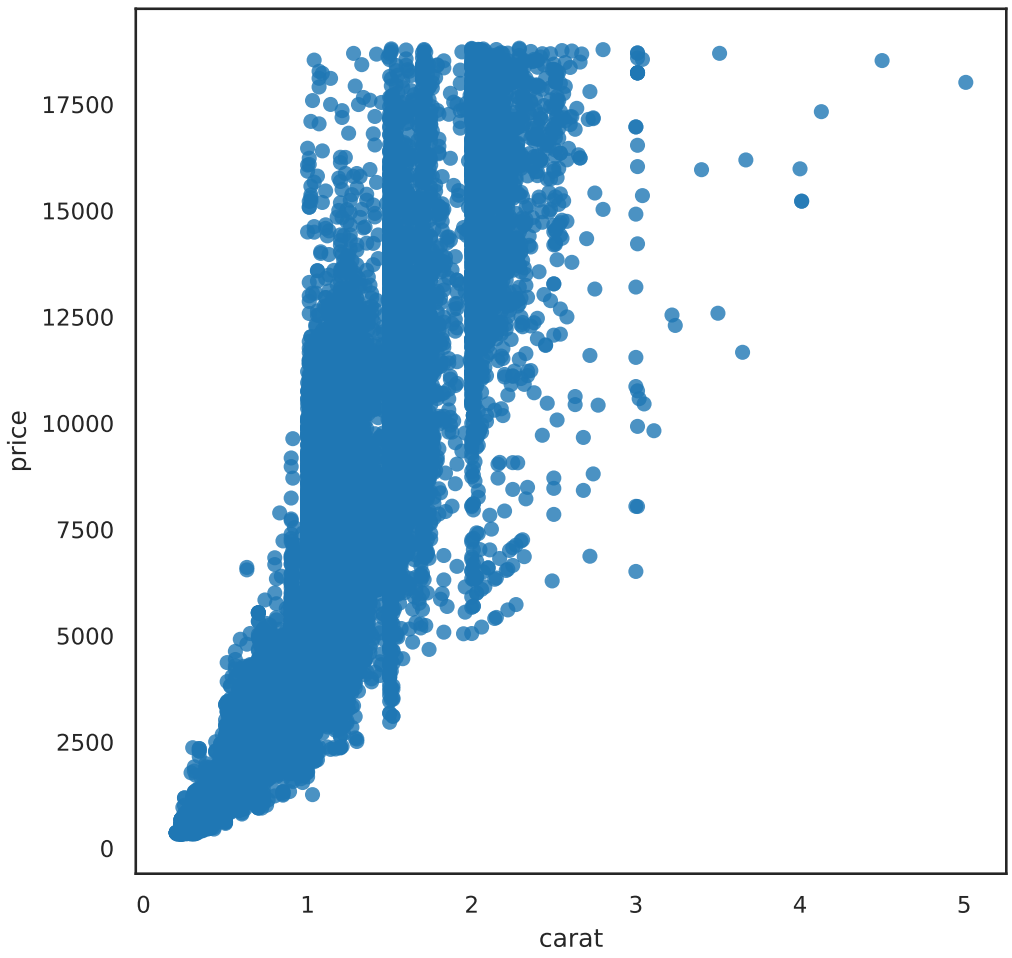
```
diamonds.boxplot(column = 'price', by = 'color');
plt.show()
```



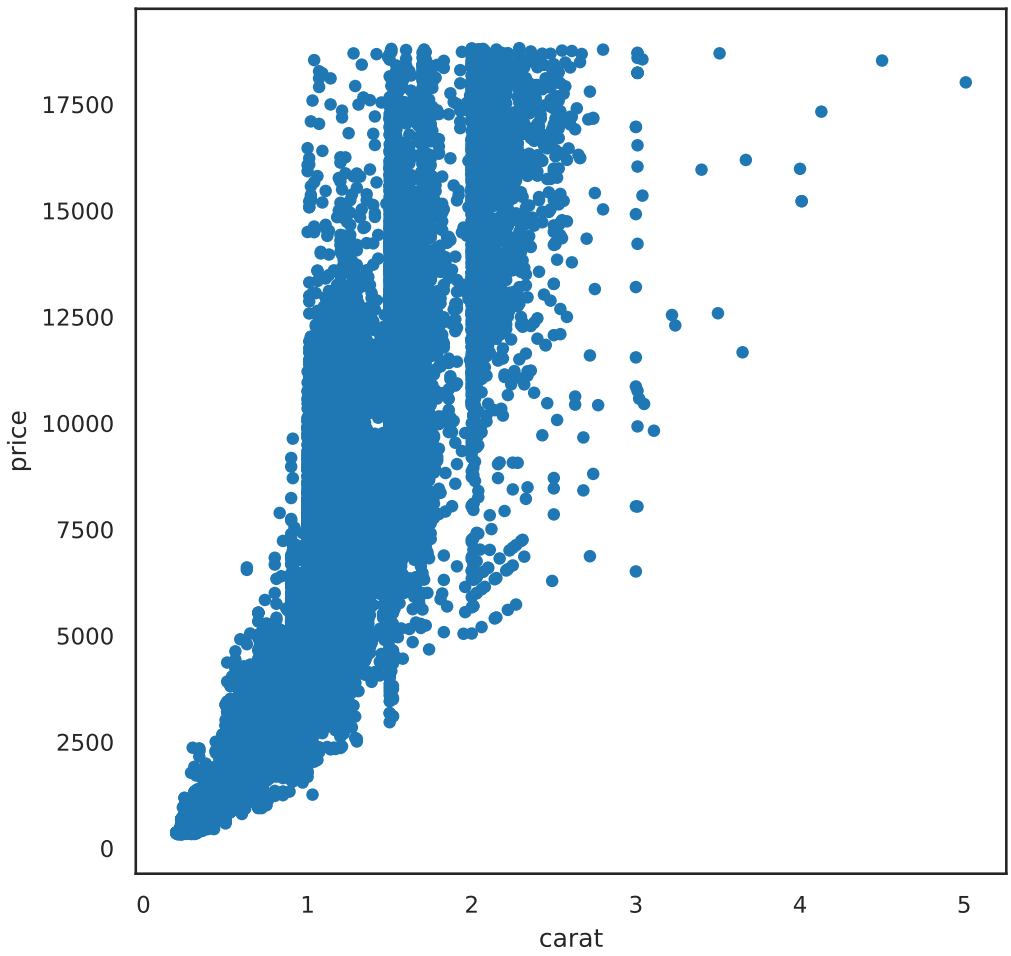
5.4.2. seaborn

5.4.2.1. Scatter plot

```
sns.regplot(data = diamonds, x = 'carat', y = 'price', fit_reg=False);
plt.show()
```

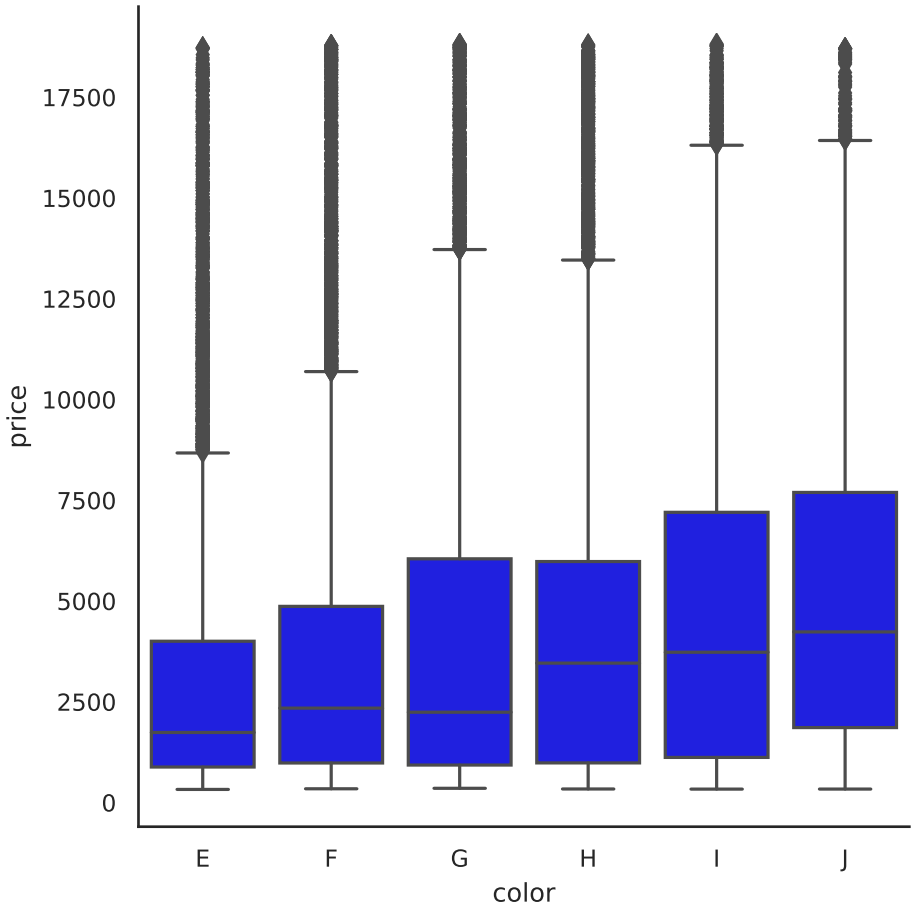



```
sns.scatterplot(data=diamonds, x = 'carat', y = 'price', linewidth=0);
# We set the linewidth to 0, otherwise the lines around the circles
# appear white and wash out the figure. Try with any positive
# value of linewidth
```



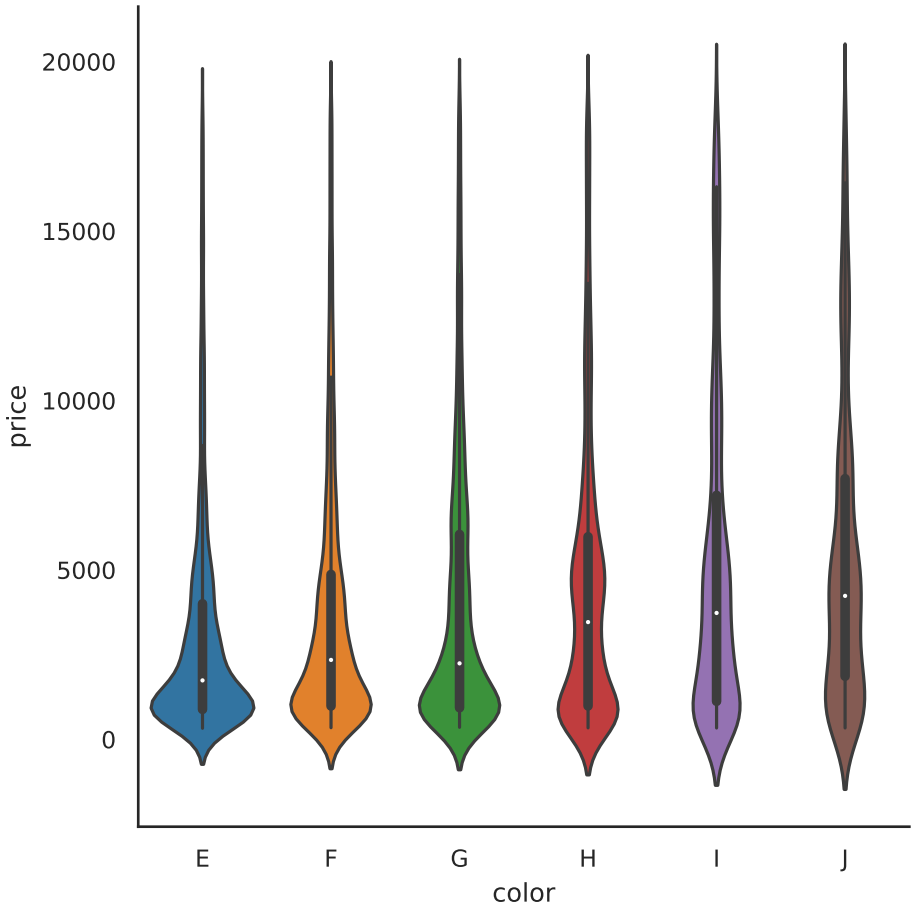
5.4.2.2. Box plot

```
ordered_color = ['E','F','G','H','I','J']
sns.catplot(data = diamonds, x = 'color', y = 'price',
            order = ordered_color, color = 'blue', kind = 'box');
plt.show()
```



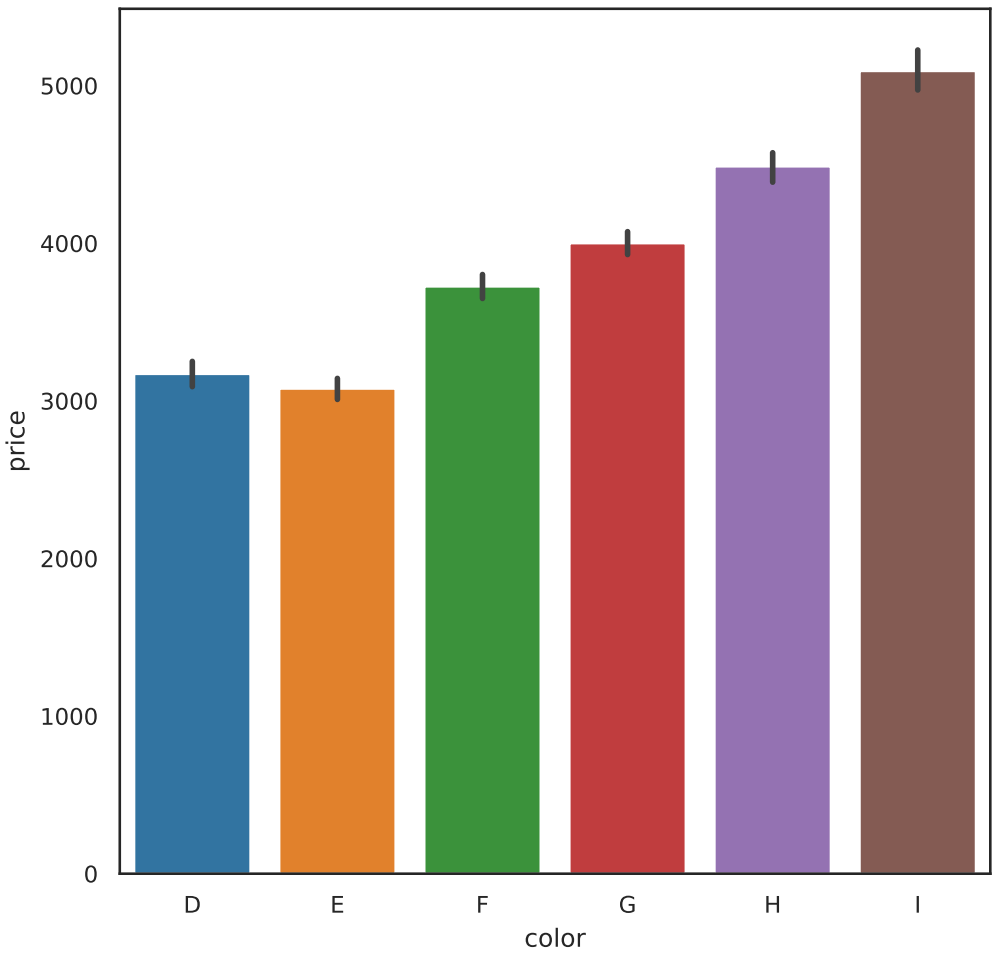
5.4.2.3. Violin plot

```
g = sns.catplot(data = diamonds, x = 'color', y = 'price',
                kind = 'violin', order = ordered_color);
plt.show()
```

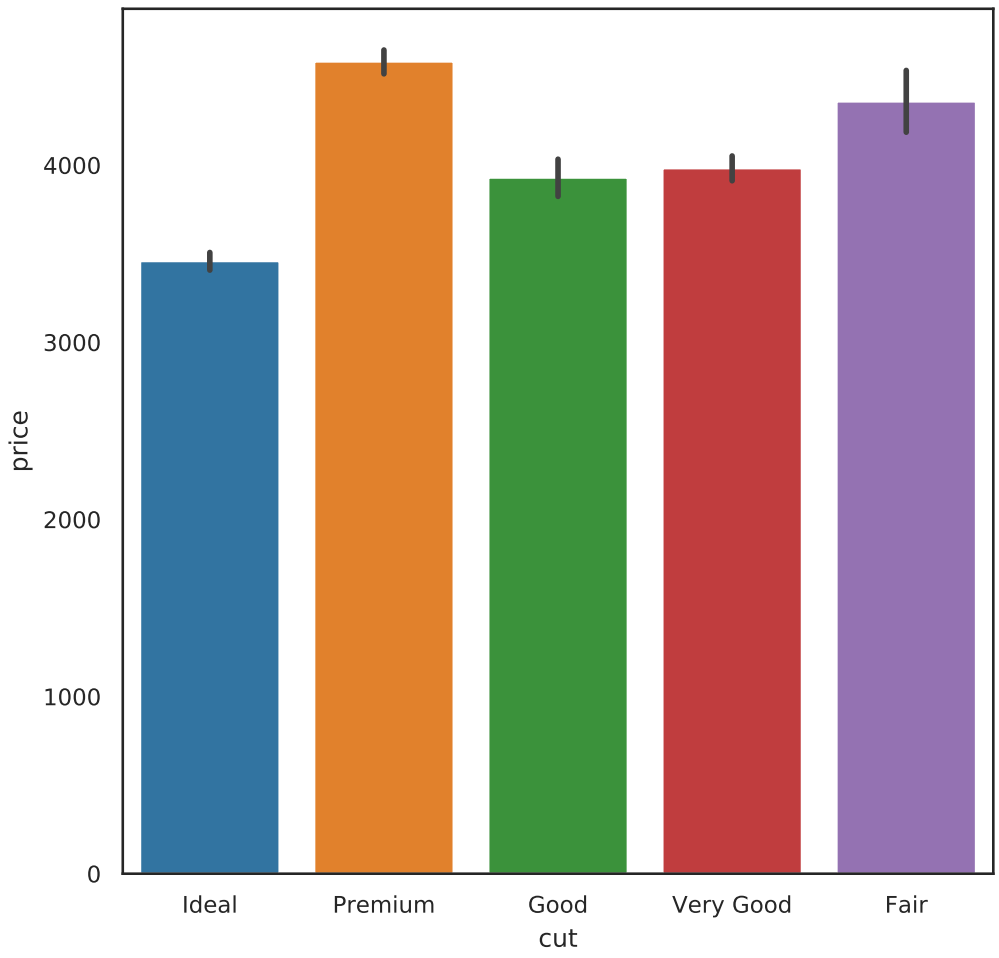


5.4.2.4. Barplot (categorical vs continuous)

```
ordered_colors = ['D','E','F','G','H','I']
sns.barplot(data = diamonds, x = 'color', y = 'price', order = ordered_colors);
plt.show()
```

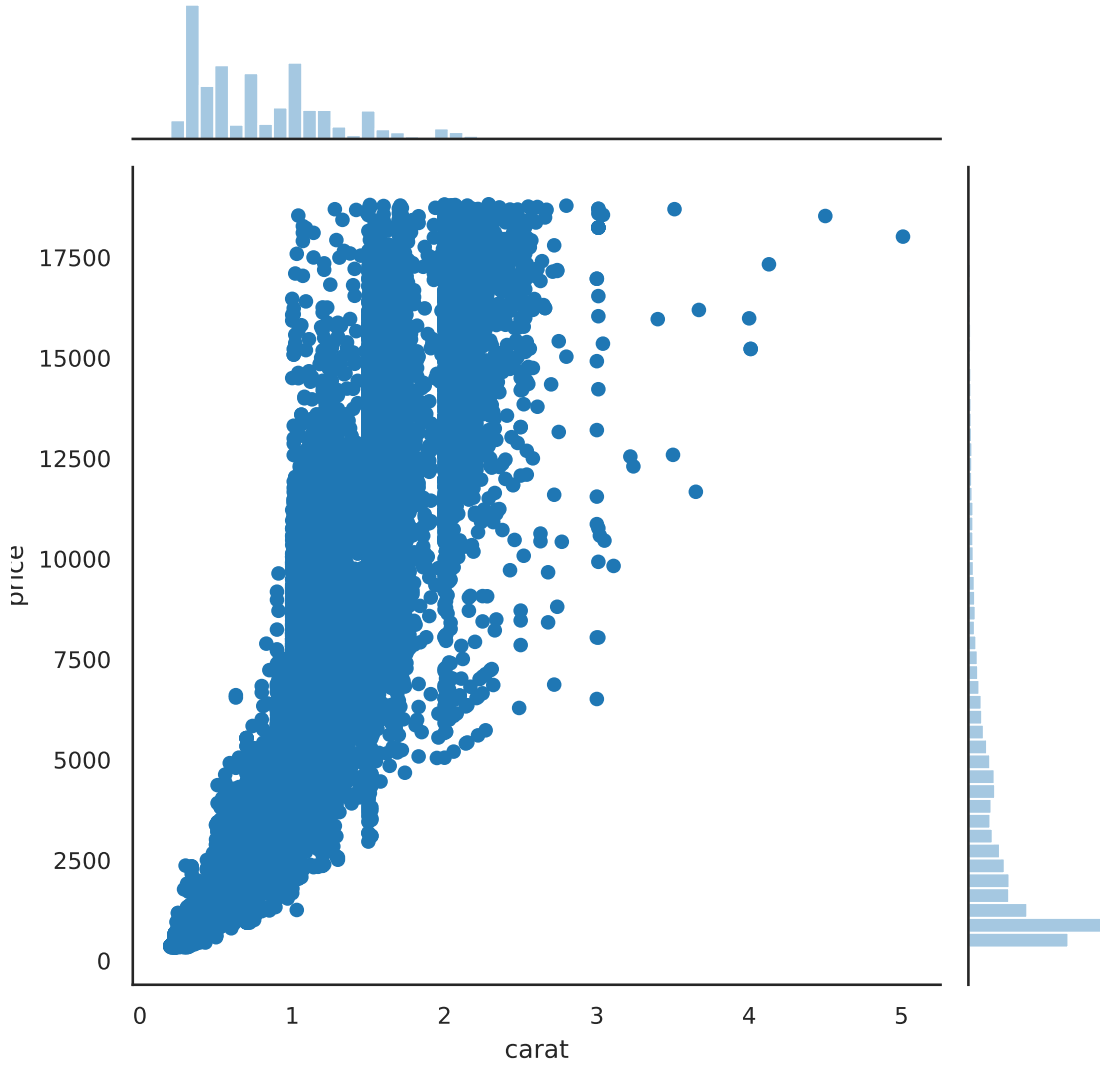



```
sns.barplot(data = diamonds, x = 'cut', y = 'price');
plt.show()
```

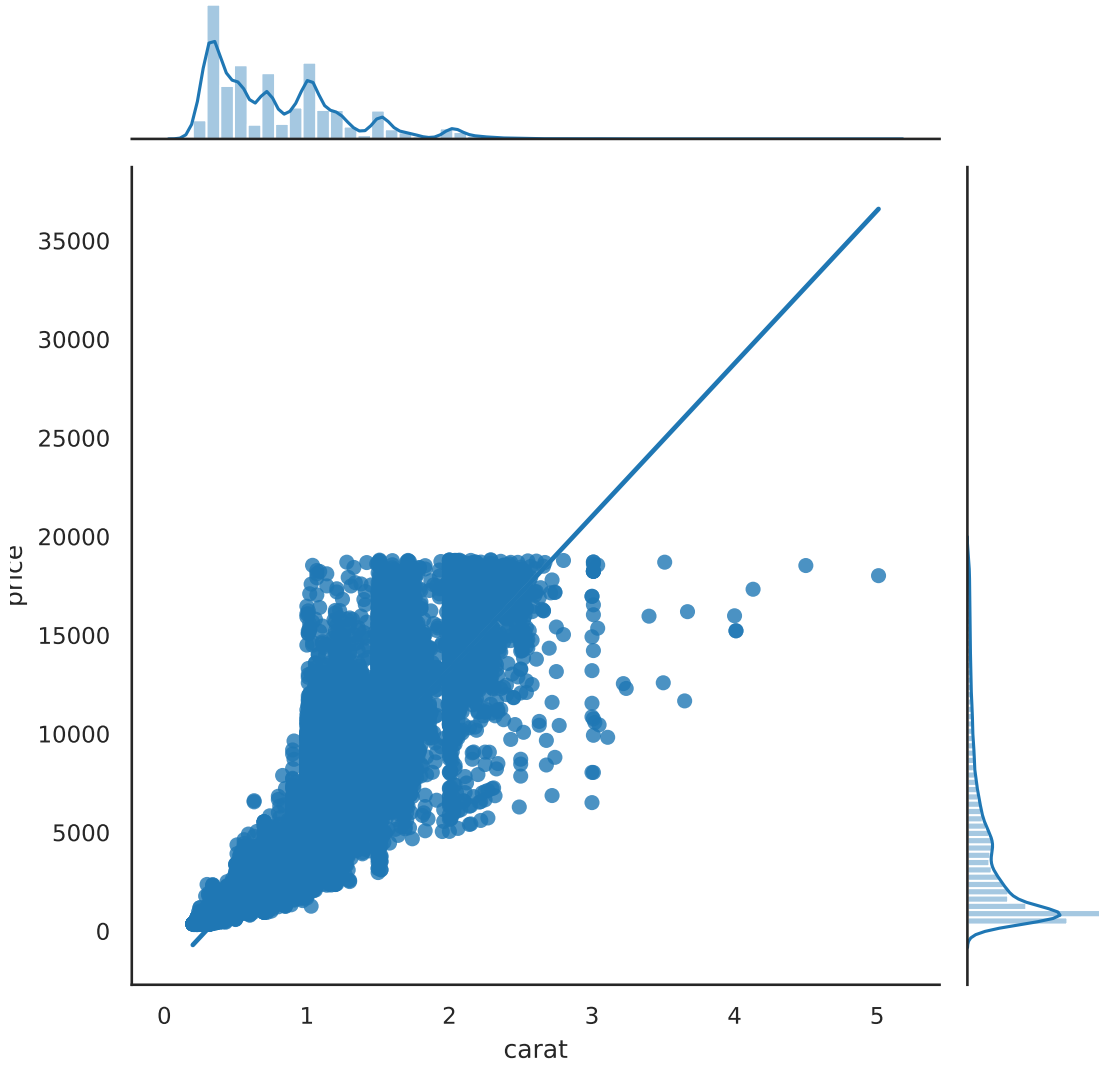


5.4.2.5. Joint plot

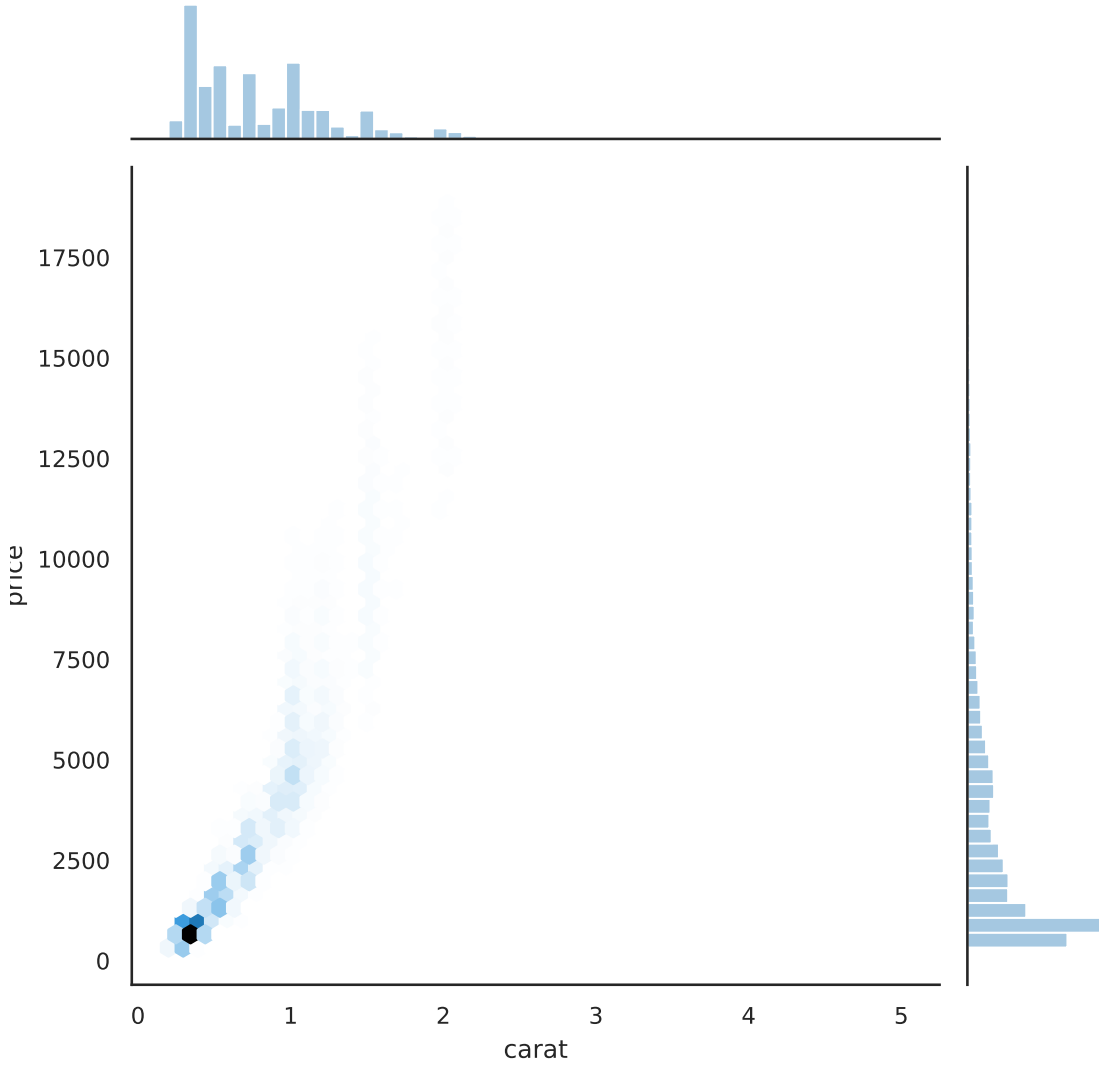
```
sns.jointplot(data = diamonds, x = 'carat', y = 'price');
plt.show()
```




```
sns.jointplot(data = diamonds, x = 'carat', y = 'price', kind = 'reg');
plt.show()
```




```
sns.jointplot(data = diamonds, x = 'carat', y = 'price', kind = 'hex');
plt.show()
```



5.5. Facets and multivariate data

The basic idea in this section is to see how we can visualize more than two variables at a time. We will see two strategies:

1. Put multiple graphs on the same frame, with each graph referring to a level of a 3rd variable
2. Create a grid of separate graphs, with each graph referring to a level of a 3rd variable

This strategy also can work any time we need to visualize the data corresponding to different levels of a variable, say by gender or race or country.

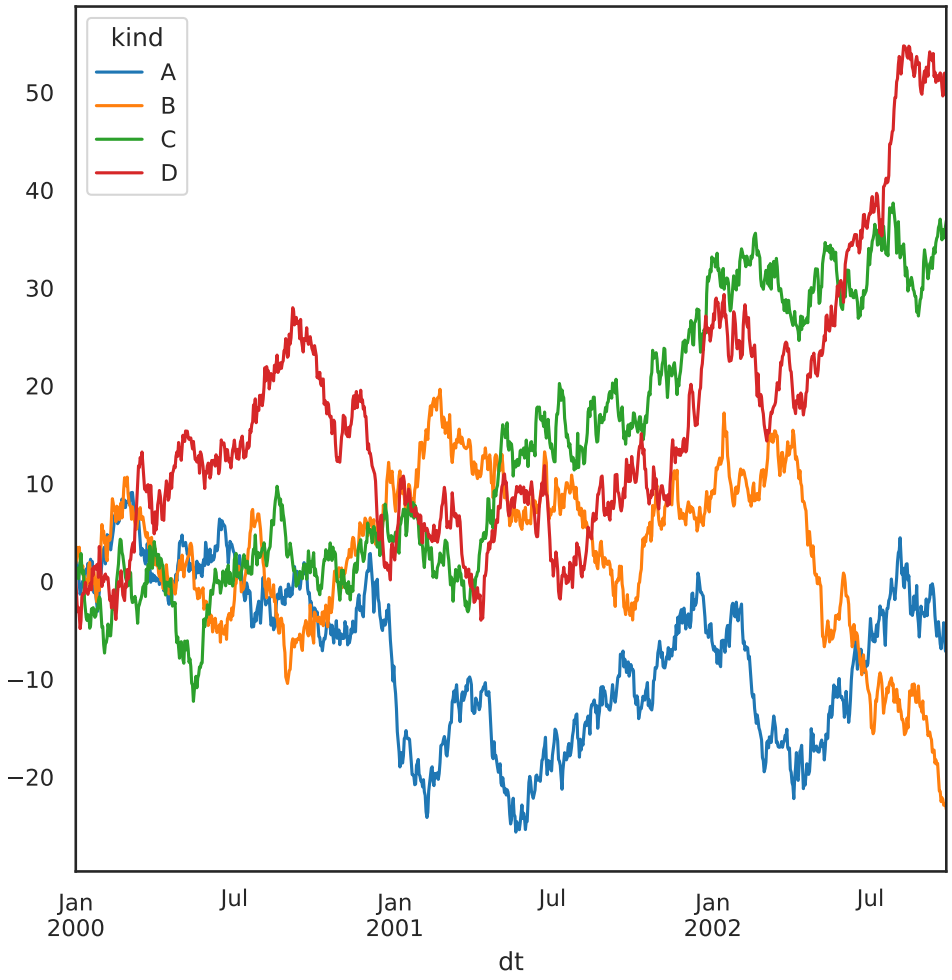
In this example we're going to start with 4 time series, labelled A, B, C, D.

```
ts = pd.read_csv('data/ts.csv')
ts.dt = pd.to_datetime(ts.dt) # convert this column to a datetime object
ts.head()
```

For one strategy we will employ, it is actually a bit easier to change this to a wide data form, using `pivot`.

```
dfp = ts.pivot(index = 'dt', columns = 'kind', values = 'value')
dfp.head()
```

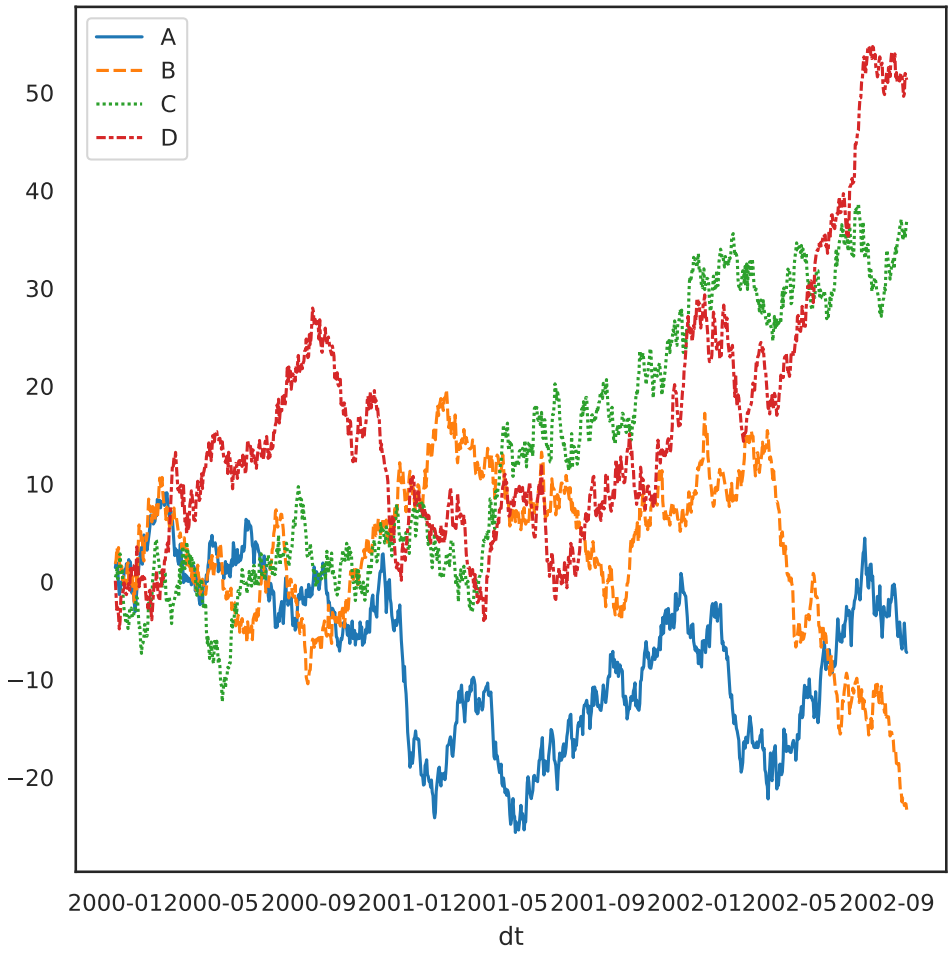
```
fig, ax = plt.subplots()
dfp.plot(ax=ax);
plt.show()
```



This creates 4 separate time series plots, one for each of the columns labeled A, B, C and D. The x-axis is determined by `dfp.index`, which during the pivoting operation, we deemed was the values of `dt` in the original data.

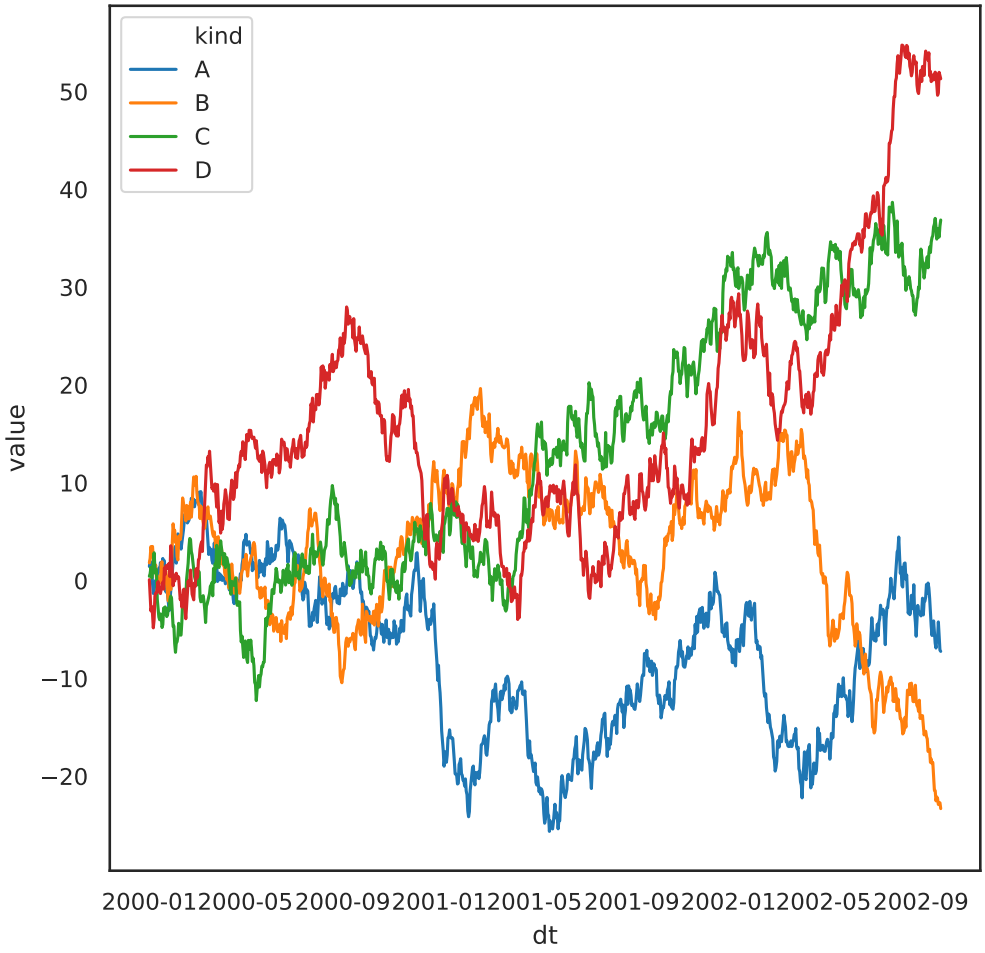
Using `seaborn`...

```
sns.lineplot(data = dfp);
plt.show()
```



However, we can achieve this same plot using the original data, and `seaborn`, in rather short order

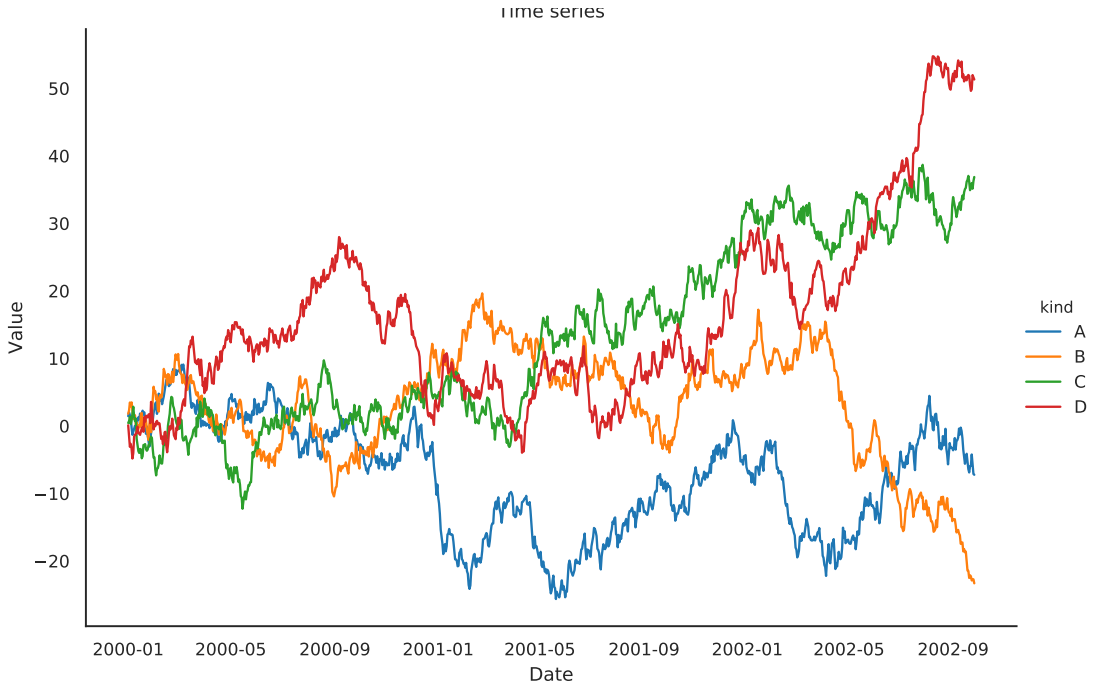
```
sns.lineplot(data = ts, x = 'dt', y = 'value', hue = 'kind');
plt.show()
```



In this plot, assigning a variable to hue tells seaborn to draw lines (in this case) of different hues based on values of that variable.

We can use a bit more granular and explicit code for this as well. This allows us a bit more control of the plot.

```
g = sns.FacetGrid(ts, hue = 'kind', height = 5, aspect = 1.5)
g.map(plt.plot, 'dt', 'value').add_legend()
g.ax.set(xlabel = 'Date',
         ylabel = 'Value',
         title = 'Time series');
plt.show()

## All of this code chunk needs to be run at one time, otherwise you get weird errors. T
## is true for many plotting commands which are composed of multiple commands.
```

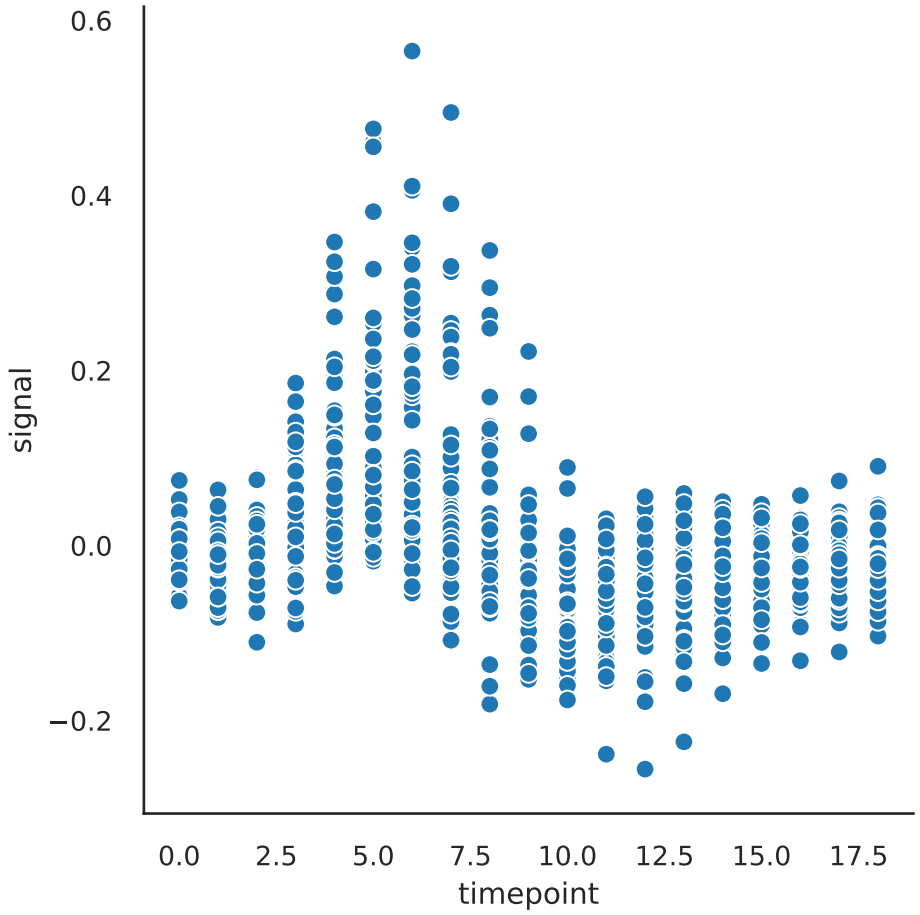


The `FacetGrid` tells `seaborn` that we're going to layer graphs, with layers based on `hue` and the hues being determined by values of `kind`. Notice that we can add a few more details like the aspect ratio of the plot and so on. The documentation for FacetGrid, which we will also use for facets below, may be helpful in finding all the options you can control.

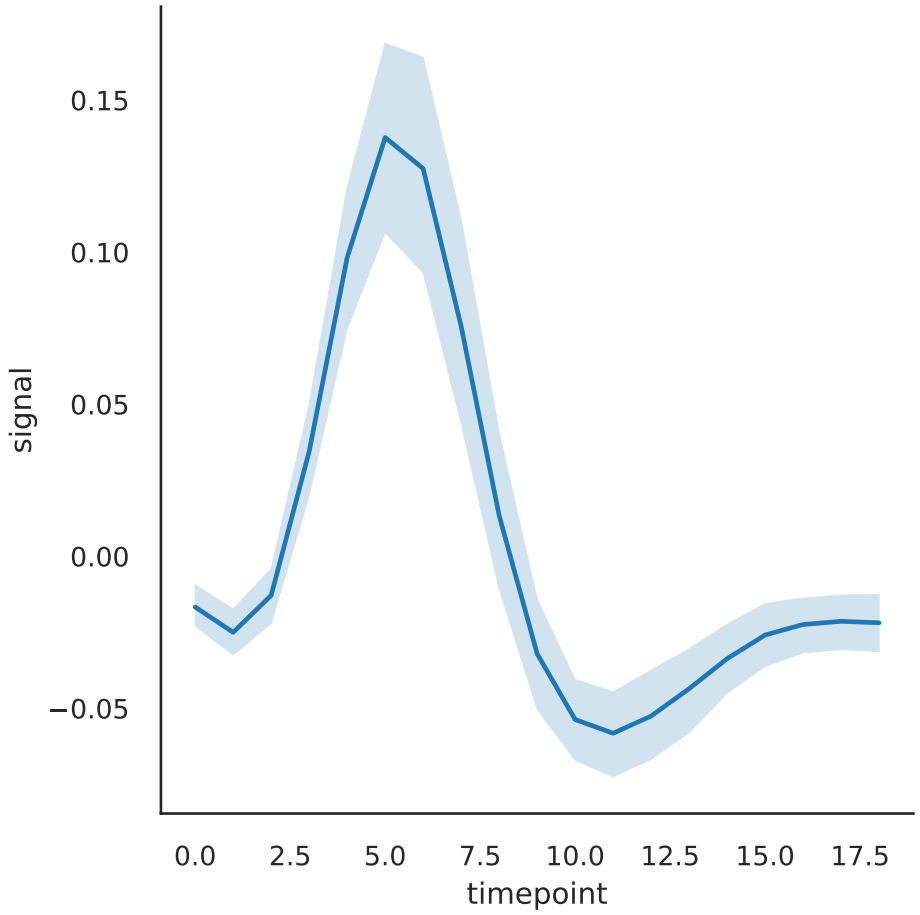
We can also show more than one kind of layer on a single graph

```
fmri = sns.load_dataset('fmri')
```

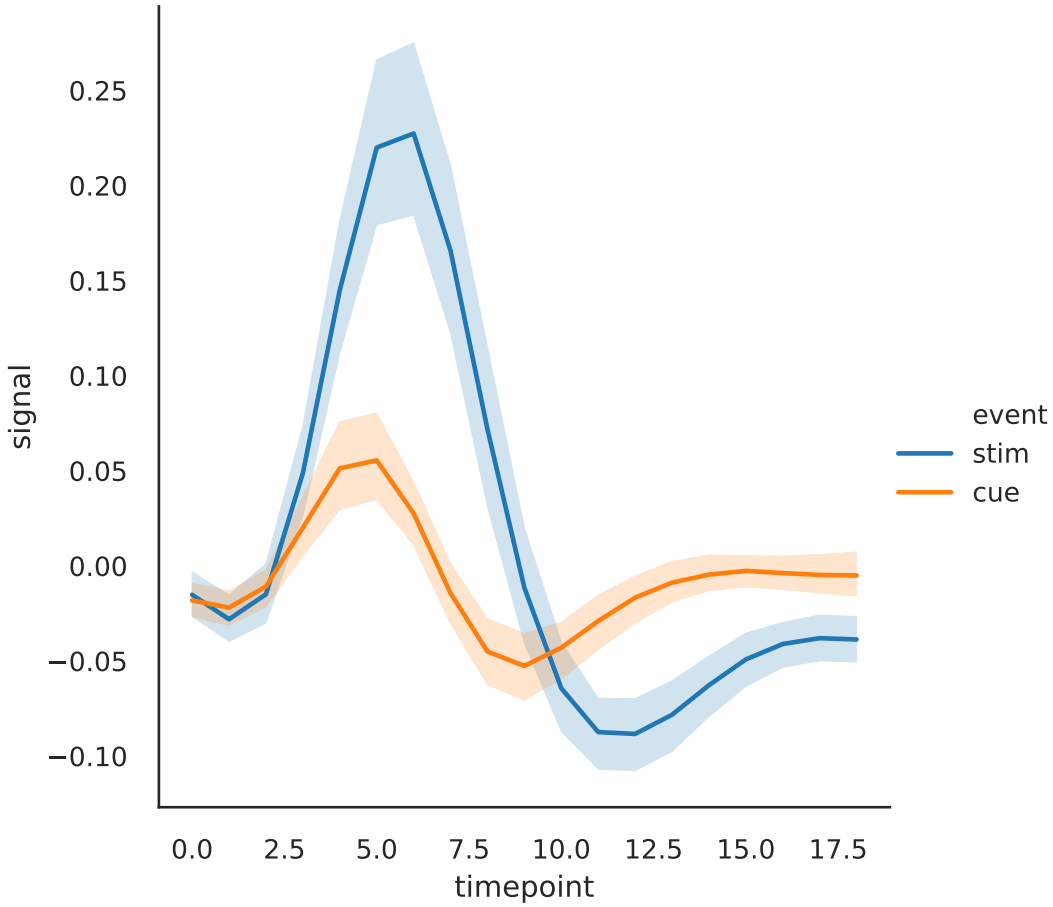
```
plt.style.use('seaborn-notebook')
sns.relplot(x = 'timepoint', y = 'signal', data = fmri);
plt.show()
```



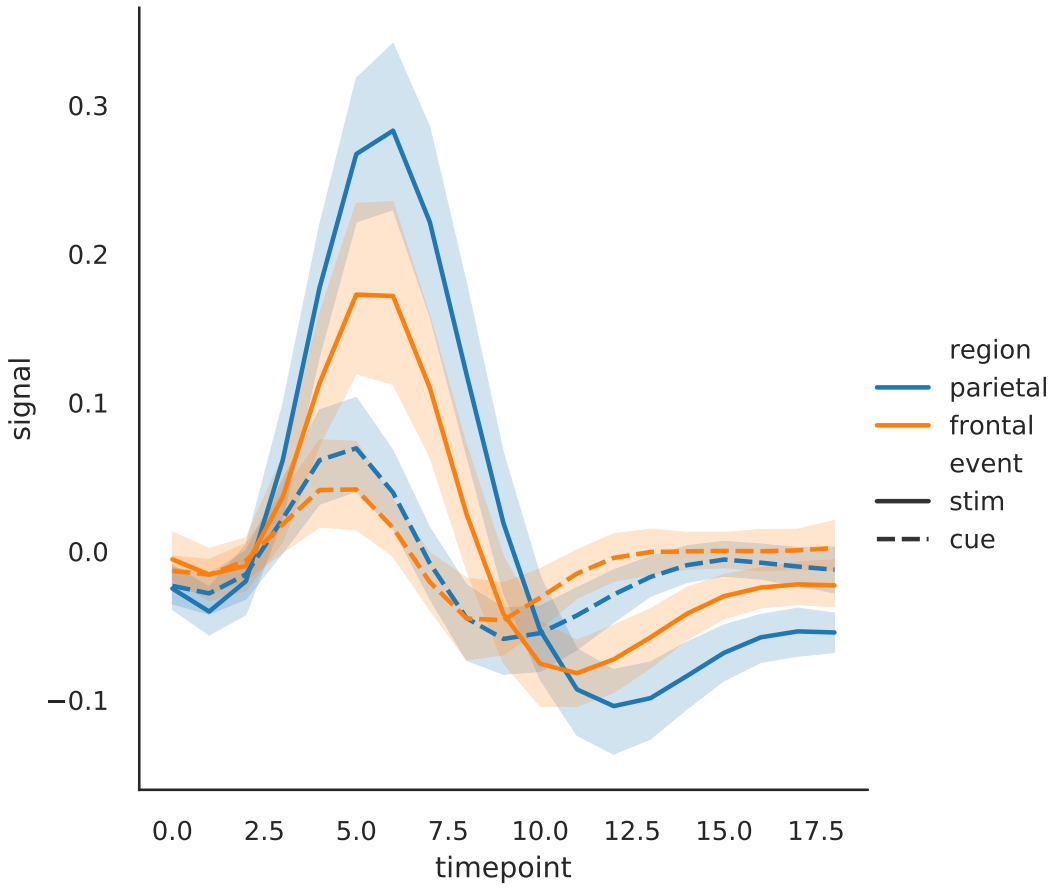

```
sns.relplot(x = 'timepoint', y = 'signal', data = fmri, kind = 'line');
plt.show()
```




```
sns.relplot(x = 'timepoint', y = 'signal', data = fmri, kind = 'line', hue ='event');
plt.show()
```



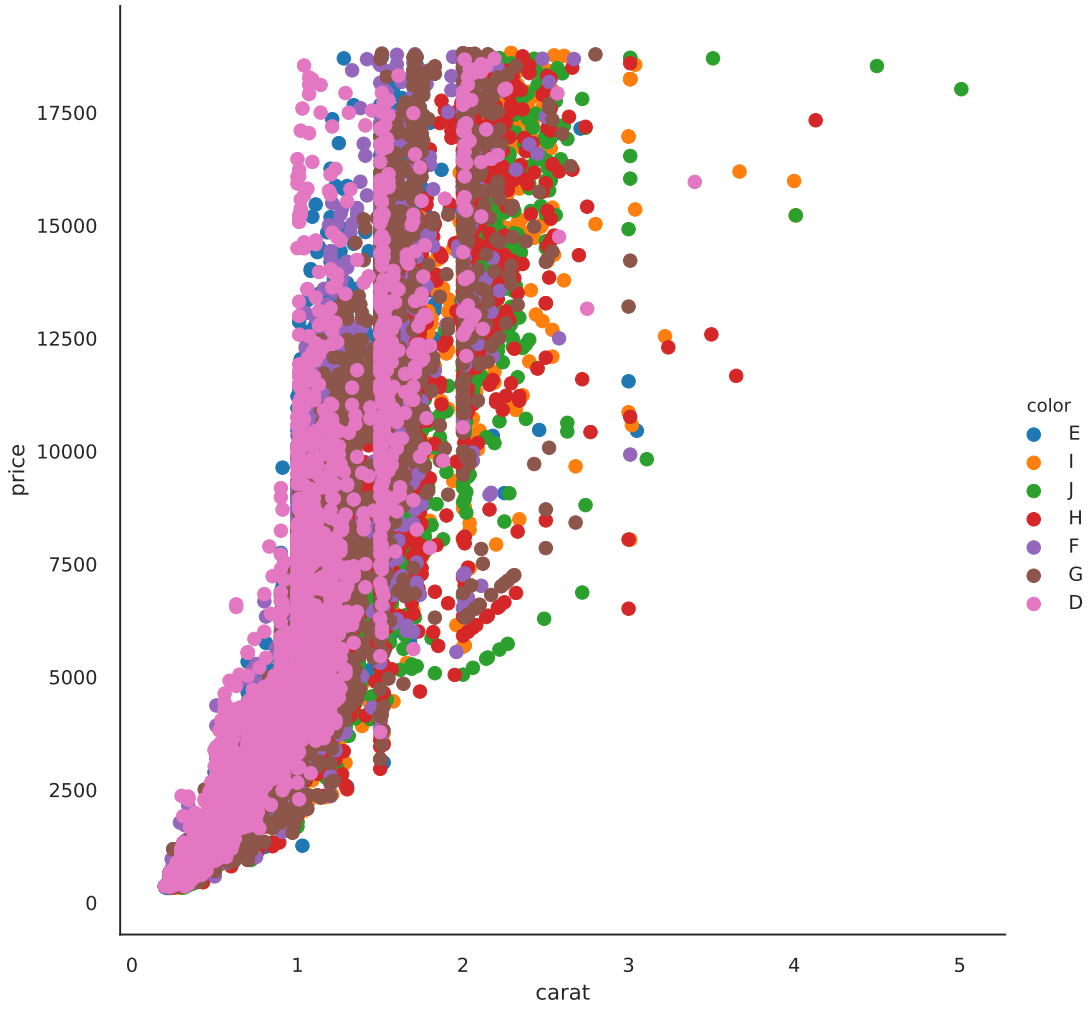

```
sns.relplot(x = 'timepoint', y = 'signal', data = fmri, hue = 'region',
            style = 'event', kind = 'line');
plt.show()
```



Here we use color to show the region, and line style (solid vs dashed) to show the event.

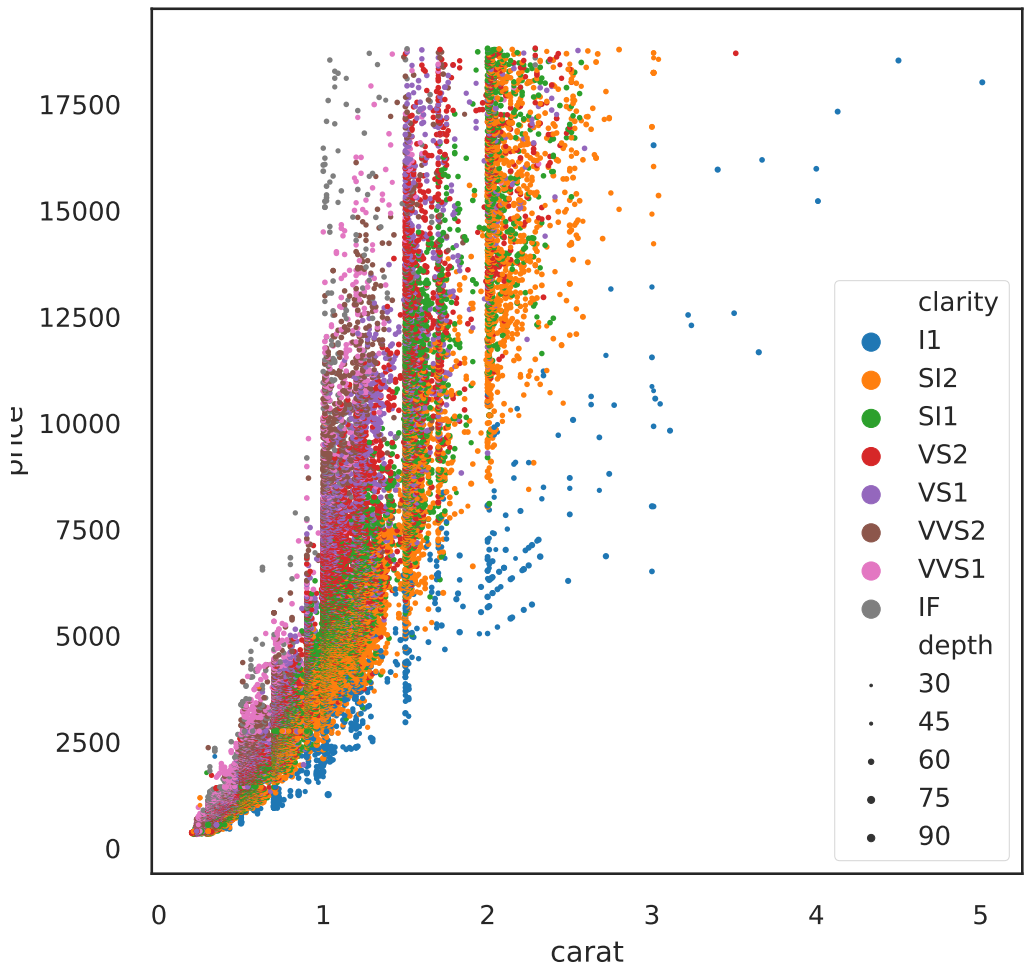
5.5.0.1. Scatter plots by group

```
g = sns.FacetGrid(diamonds, hue = 'color', height = 7.5)
g.map(plt.scatter, 'carat', 'price').add_legend();
plt.show()
```



Notice that this arranges the colors and values for the `color` variable in random order. If we have a preferred order we can impose that using the option `hue_order`.

```
clarity_ranking = ["I1", "SI2", "SI1", "VS2", "VS1", "VVS2", "VVS1", "IF"]
sns.scatterplot(x="carat", y="price",
                hue="clarity", size="depth",
                hue_order=clarity_ranking,
                sizes=(1, 8), linewidth=0,
                data=diamonds);
plt.show()
```



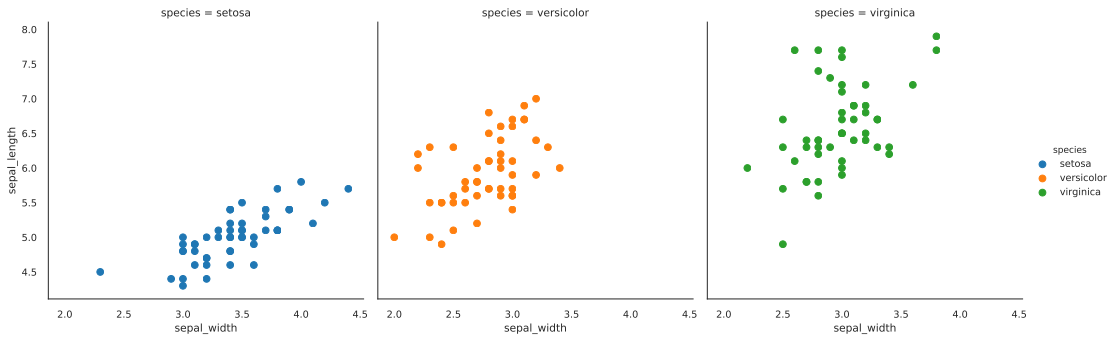
5.5.1. Facets

Facets or trellis graphics is a visualization method where we draw multiple plots in a grid, with each plot corresponding to unique values of a particular variable or combinations of variables. This has also been called *small multiples*.

We'll proceed with an example using the `iris` dataset.

```
iris = pd.read_csv('data/iris.csv')
iris.head()
```

```
g = sns.FacetGrid(iris, col = 'species', hue = 'species', height = 5)
g.map(plt.scatter, 'sepal_width', 'sepal_length').add_legend();
plt.show()
```

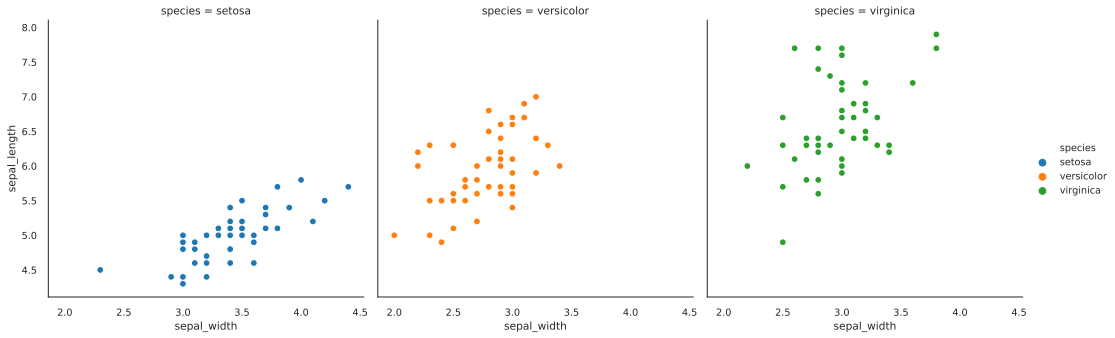


Here we use `FacetGrid` to indicate that we're creating multiple subplots by specifying the option `col` (for column). So this code says we are going to create one plot per level of species, arranged as separate columns (or in effect along one row). You could also specify `row` which would arrange the plots one to a row, or, in effect, in one column.

The `map` function says, take the facets I've defined and stored in g, and in each one, plot a scatter plot with `sepal_width` on the x-axis and `sepal_length` on the y-axis.

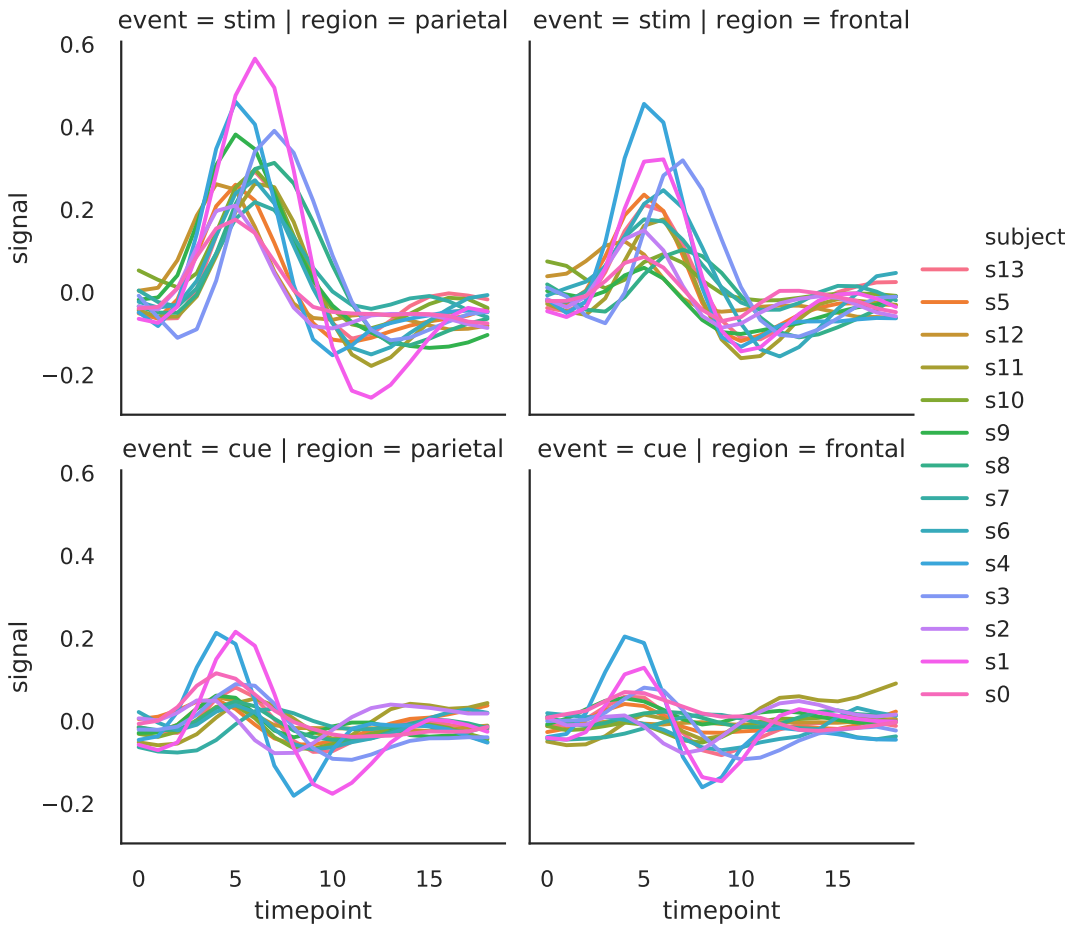
We could also use `relplot` for a more compact solution.

```
sns.relplot(x = 'sepal_width', y = 'sepal_length', data = iris,
            col = 'species', hue = 'species');
plt.show()
```



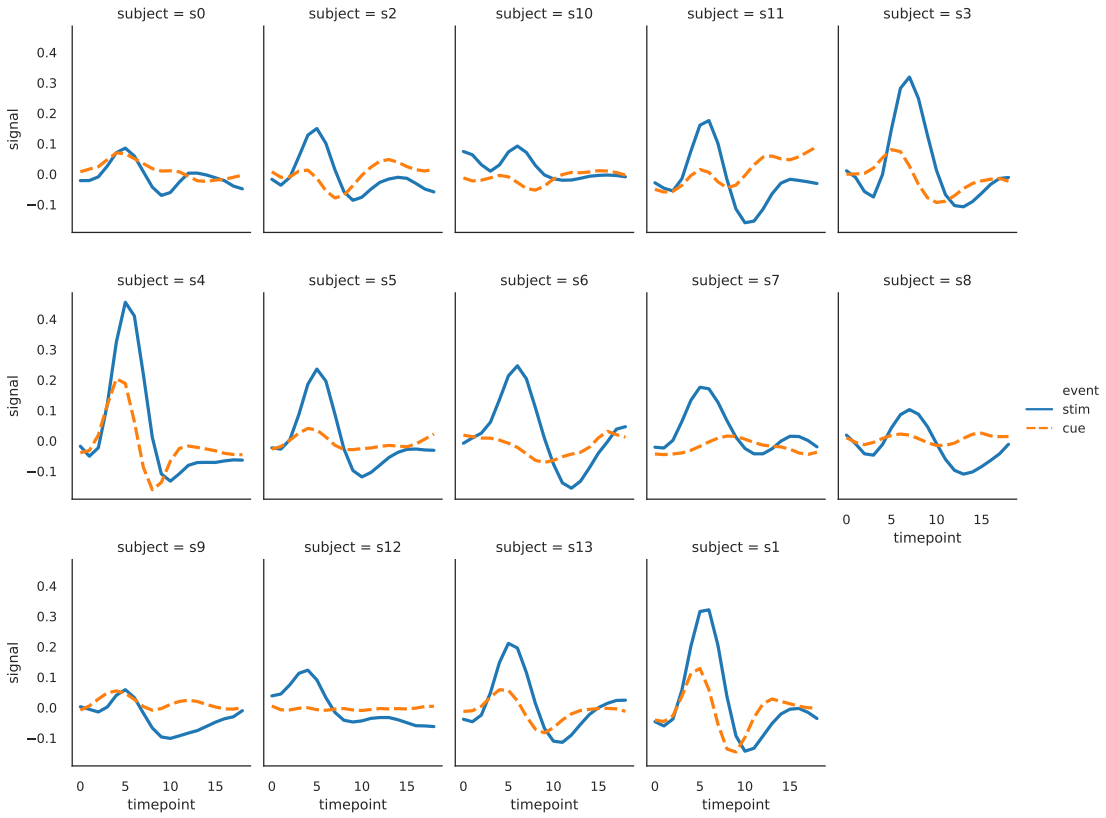
A bit more of a complicated example, using the `fmri` data, where we're coloring lines based on the subject, and creating a 2-d grid, where region of the brain in along columns and event type is along rows.

```
sns.relplot(x="timepoint", y="signal", hue="subject",
            col="region", row="event", height=3,
            kind="line", estimator=None, data=fmri);
plt.show()
```



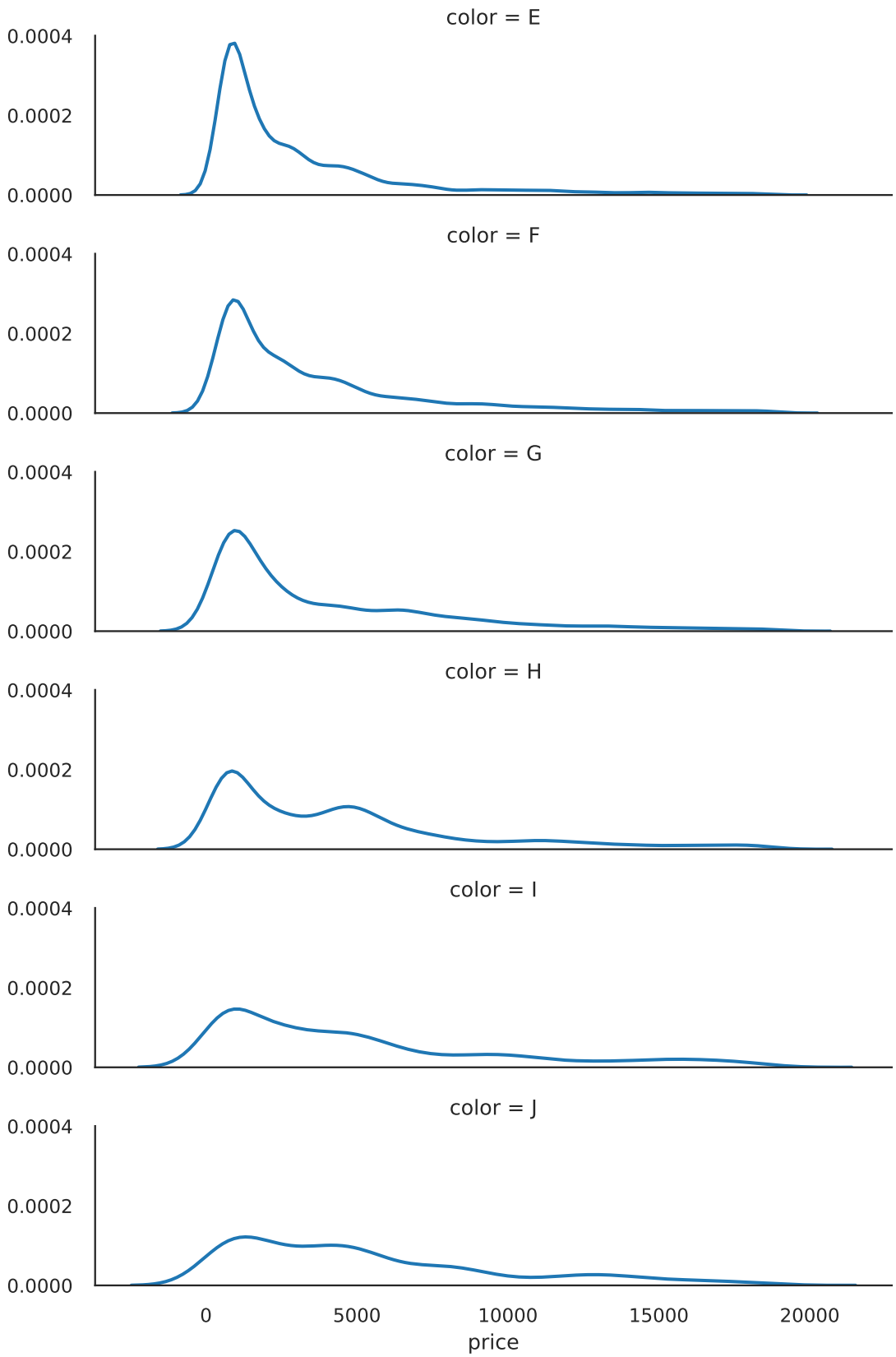
In the following example, we want to show how each subject fares for each of the two events, just within the frontal region. We let `seaborn` figure out the layout, only specifying that we'll be going along rows ("by column") and also saying we'll wrap around to the beginning once we've got to 5 columns. Note we use the `query` function to filter the dataset.

```
sns.relplot(x="timepoint", y="signal", hue="event", style="event",
            col="subject", col_wrap=5,
            height=3, aspect=.75, linewidth=2.5,
            kind="line", data=fmri.query("region == 'frontal'"));
plt.show()
```



In the following example we want to compare the distribution of price from the diamonds dataset by color, and so it makes sense to create density plots of the price distribution and stack them one below the next so we can visually compare them.

```
ordered_colors = ['E','F','G','H','I','J']
g = sns.FacetGrid(data = diamonds, row = 'color', height = 1.7,
                  aspect = 4, row_order = ordered_colors)
g.map(sns.kdeplot, 'price');
plt.show()
```

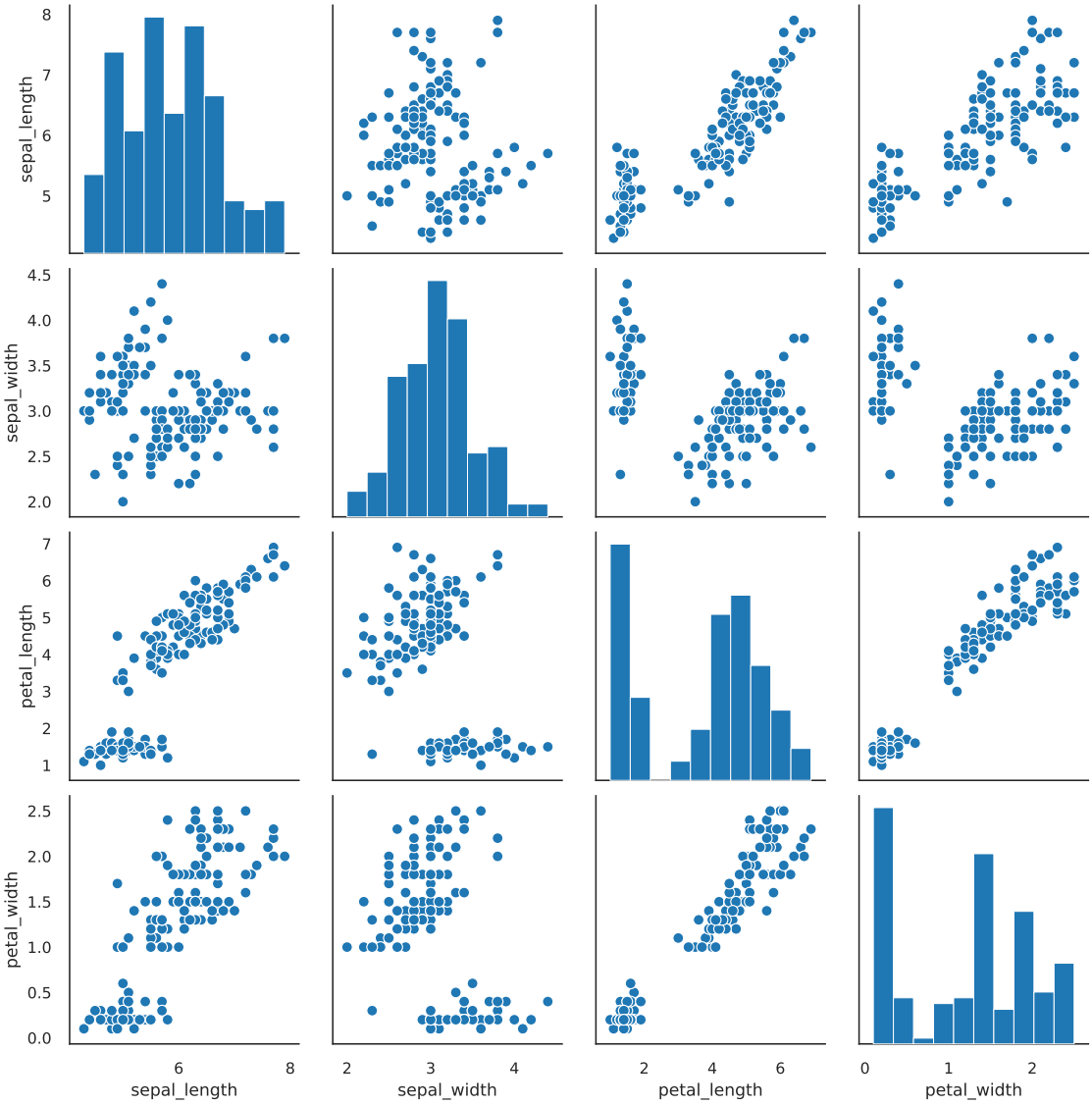

color = E
color = F
color = G
color = H
color = I
color = J
0.0004
0.0002
0.0000
0
5000
10000
15000
20000
price

You need to use `FacetGrid` to create sets of univariate plots since there is no particular method that allows univariate plots over a grid like `relplot` for bivariate plots.

5.5.2. Pairs plots

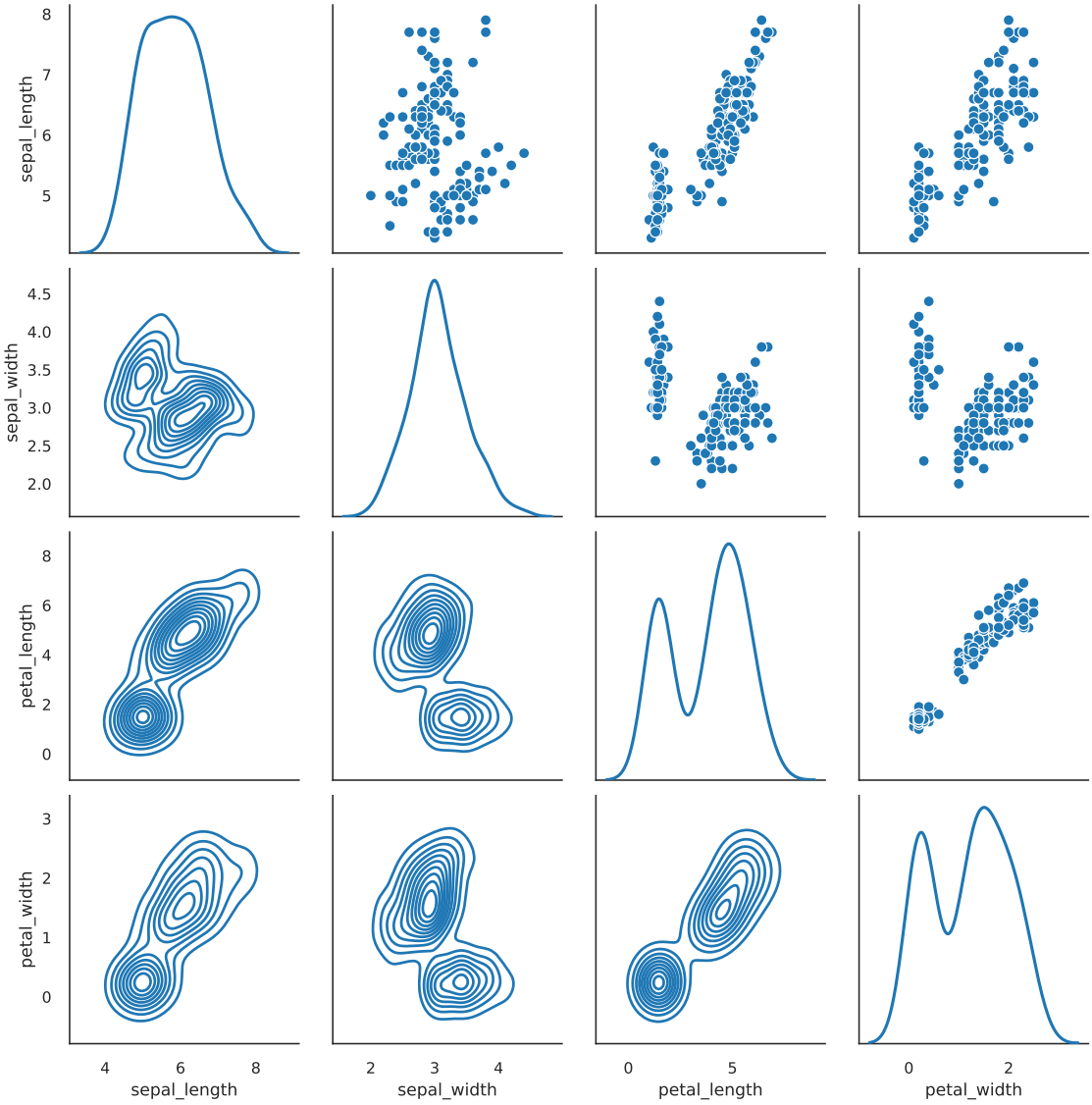
The pairs plot is a quick way to compare every pair of variables in a dataset (or at least, every pair of continuous variables) in a grid. You can specify what kind of univariate plot will be displayed on the diagonal locations on the grid, and which bivariate plots will be displayed on the off-diagonal locations.

```
sns.pairplot(data=iris);
plt.show()
```



You can achieve more customization using `PairGrid`.

```
g = sns.PairGrid(iris, diag_sharey=False);
g.map_upper(sns.scatterplot);
g.map_lower(sns.kdeplot, colors="C0");
g.map_diag(sns.kdeplot, lw=2);
plt.show()
```



5.6. Customizing the look

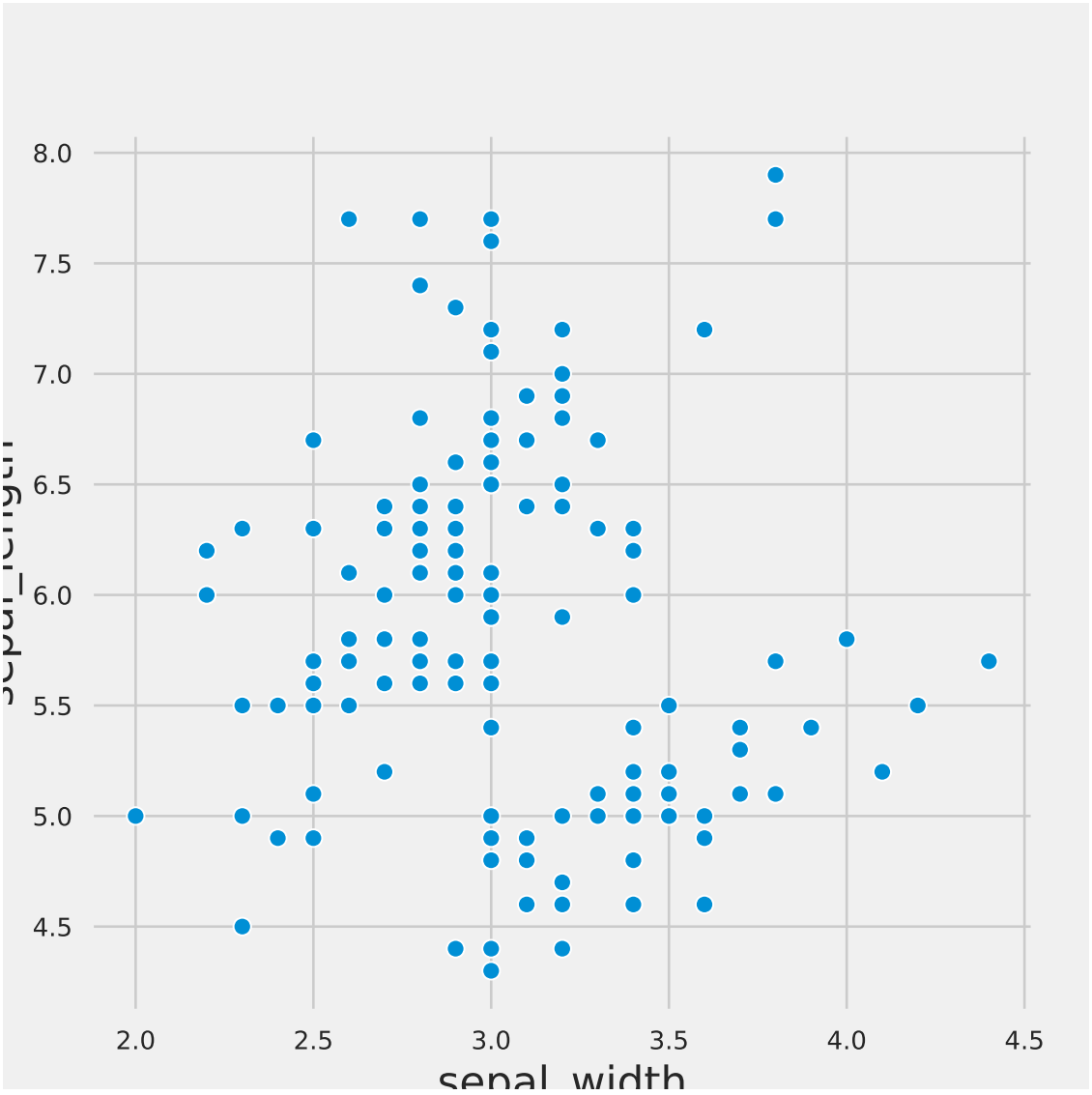
5.6.1. Themes

There are several themes available in the modern `matplotlib`, some of which borrow from `seaborn`. You can see the available themes and play around.

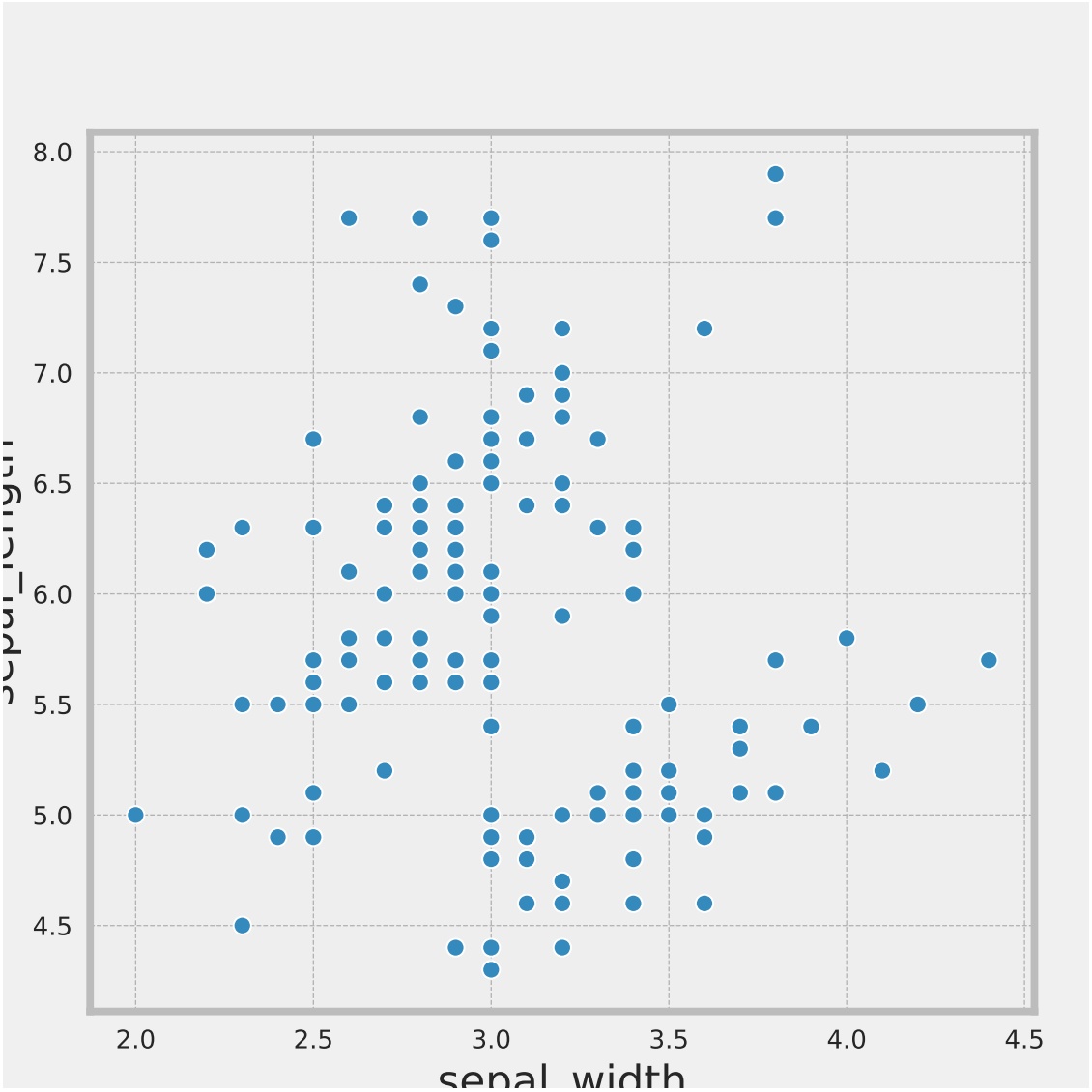
```
plt.style.available
```

See some examples below.

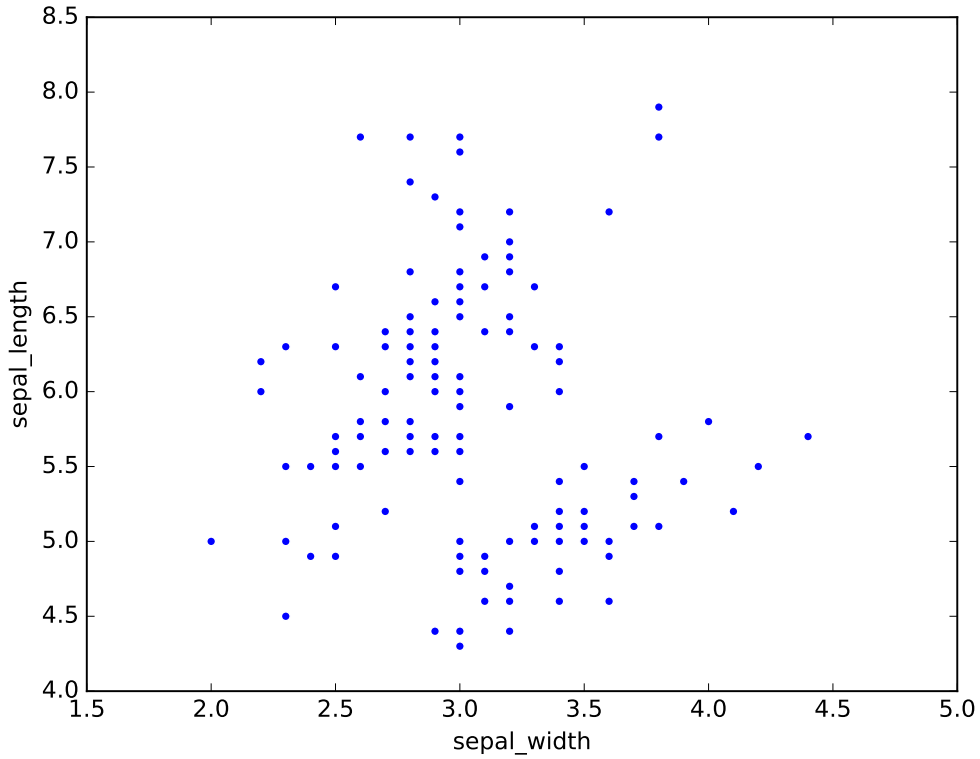
```
plt.style.use('fivethirtyeight')
sns.scatterplot(data = iris, x = 'sepal_width', y = 'sepal_length');
plt.show()
```



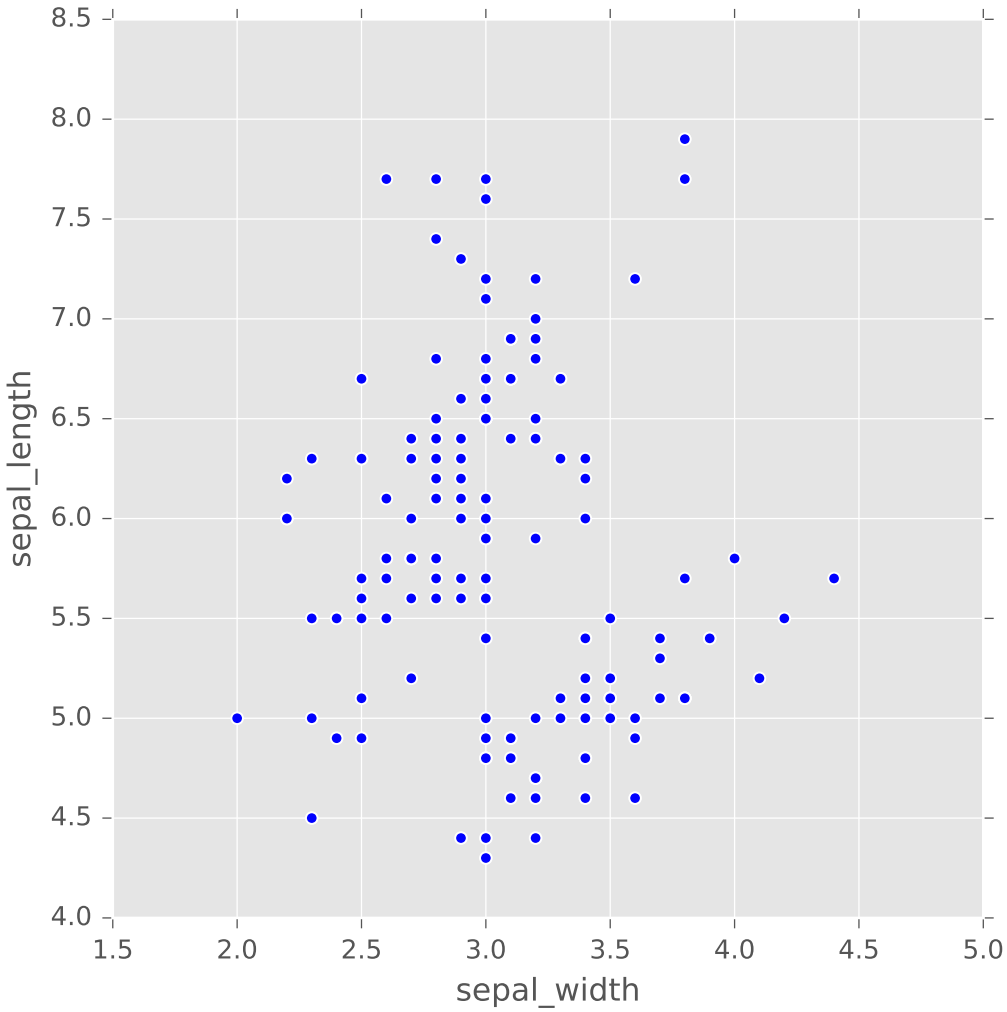

```
plt.style.use('bmh')
sns.scatterplot(data = iris, x = 'sepal_width', y = 'sepal_length');
plt.show()
```



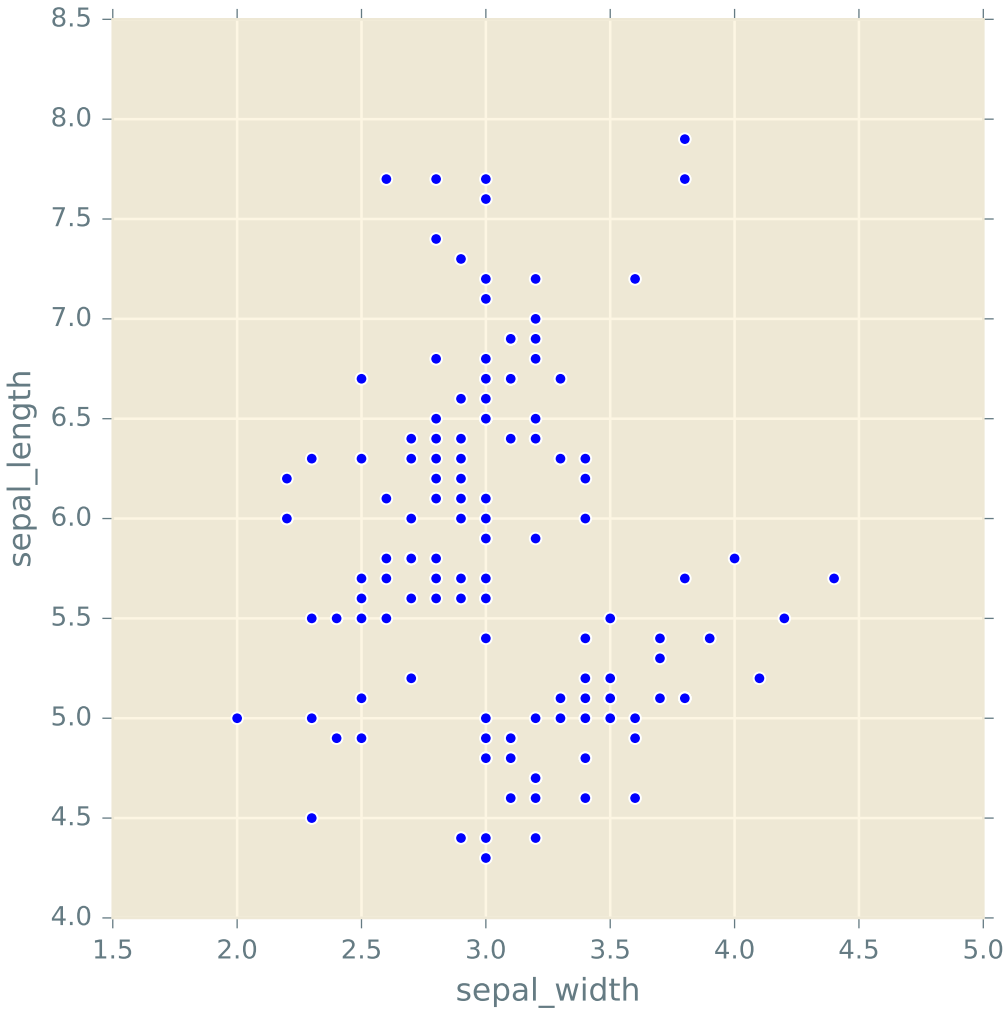

```
plt.style.use('classic')
sns.scatterplot(data = iris, x = 'sepal_width', y = 'sepal_length');
plt.show()
```




```
plt.style.use('ggplot')
sns.scatterplot(data = iris, x = 'sepal_width', y = 'sepal_length');
plt.show()
```




```
plt.style.use('Solarize_Light2')
sns.scatterplot(data = iris, x = 'sepal_width', y = 'sepal_length');
plt.show()
```



One small syntax point. You may have noticed in your own work that you get a little annoying line in the output when you plot. You can prevent that from happening by putting a semi-colon (`;`) after the last plotting command

5.7. Finer control with matplotlib

As you can see from the figure, you can control each aspect of the plot displayed above using `matplotlib`. I won't go into the details, and will leave it to you to look at the `matplotlib` documentation and examples if you need to customize at this level of granularity.

The following is an example using pure `matplotlib`. You can see how you can build up a plot. The crucial part here is that you need to run the code from each chunk together.

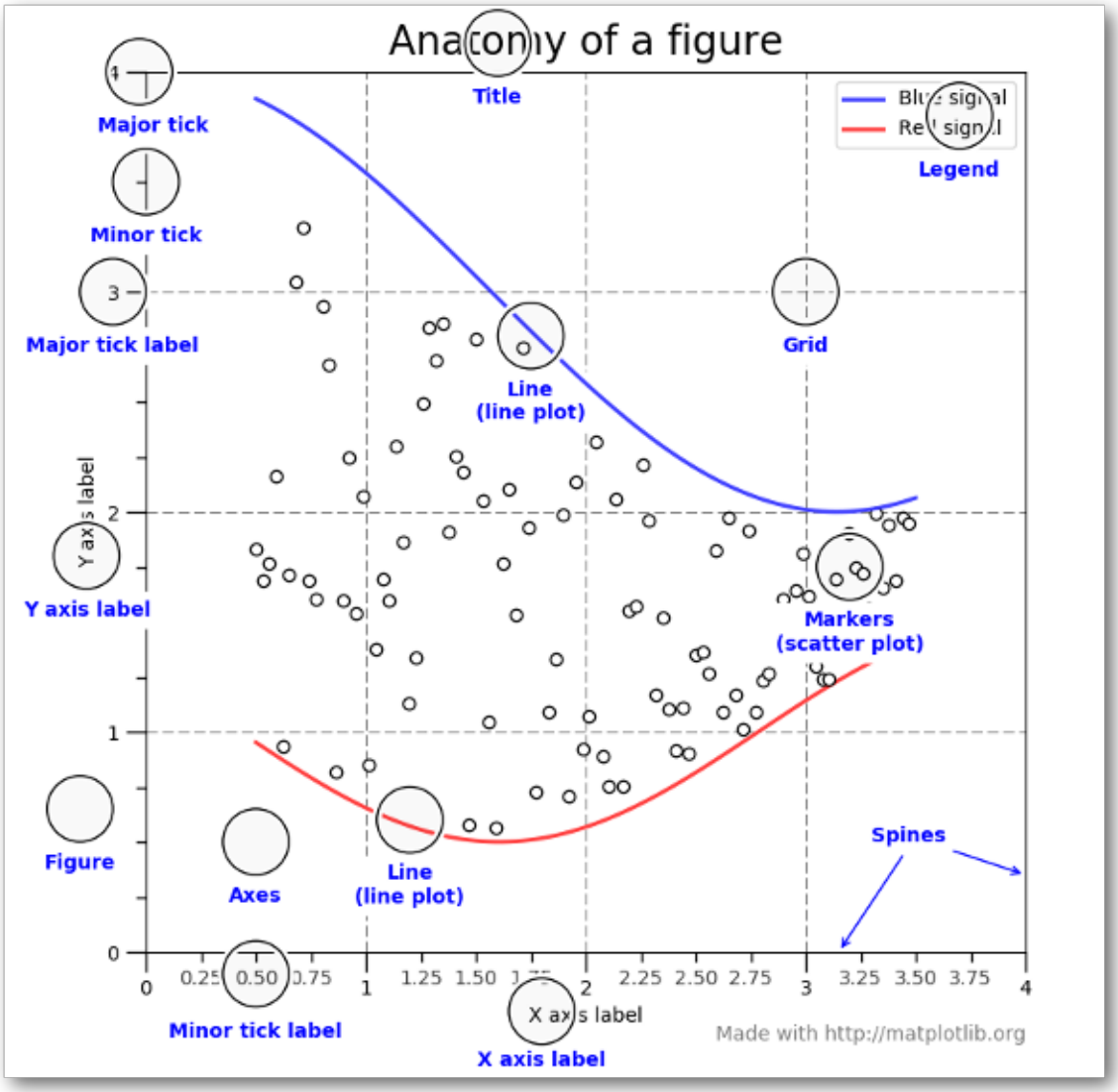
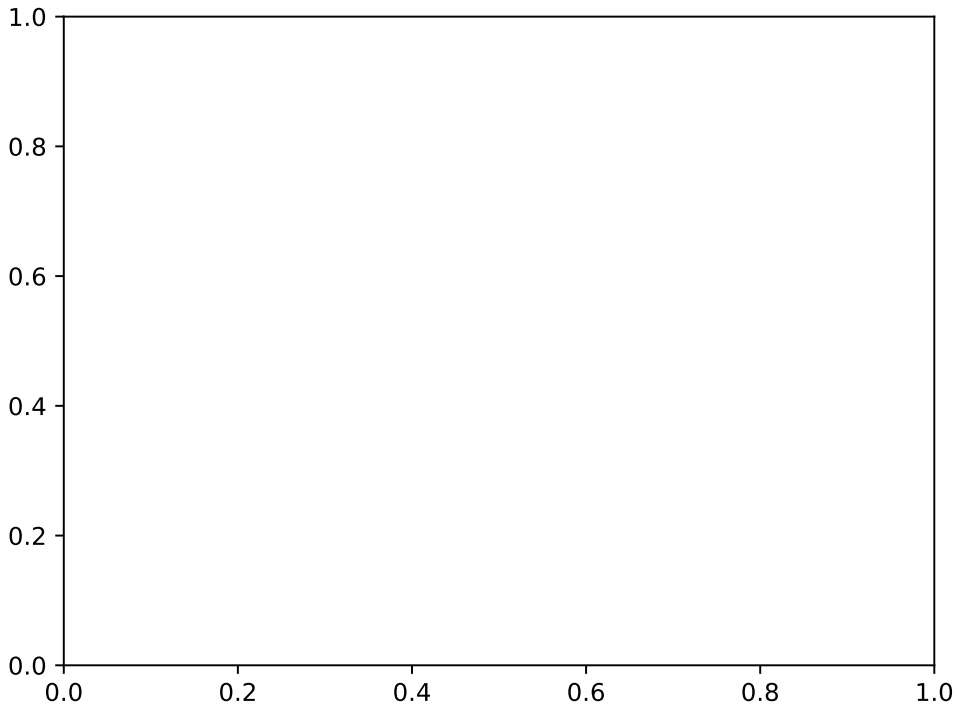


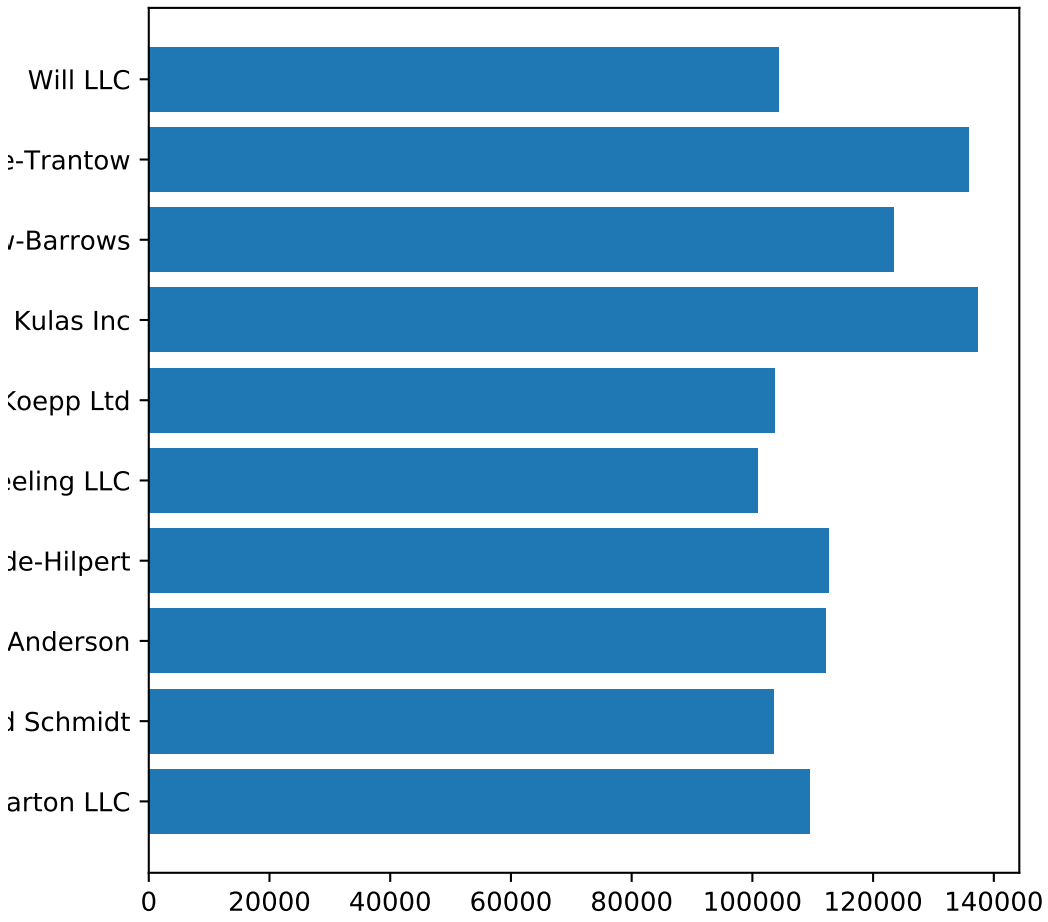
Figure 5.1.: https://matplotlib.org/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py

```
from matplotlib.ticker import FuncFormatter

data = {'Barton LLC': 109438.50,
        'Frami, Hills and Schmidt': 103569.59,
        'Fritsch, Russel and Anderson': 112214.71,
        'Jerde-Hilpert': 112591.43,
        'Keeling LLC': 100934.30,
        'Koepp Ltd': 103660.54,
        'Kulas Inc': 137351.96,
        'Trantow-Barrows': 123381.38,
        'White-Trantow': 135841.99,
        'Will LLC': 104437.60}
group_data = list(data.values())
group_names = list(data.keys())
group_mean = np.mean(group_data)
```

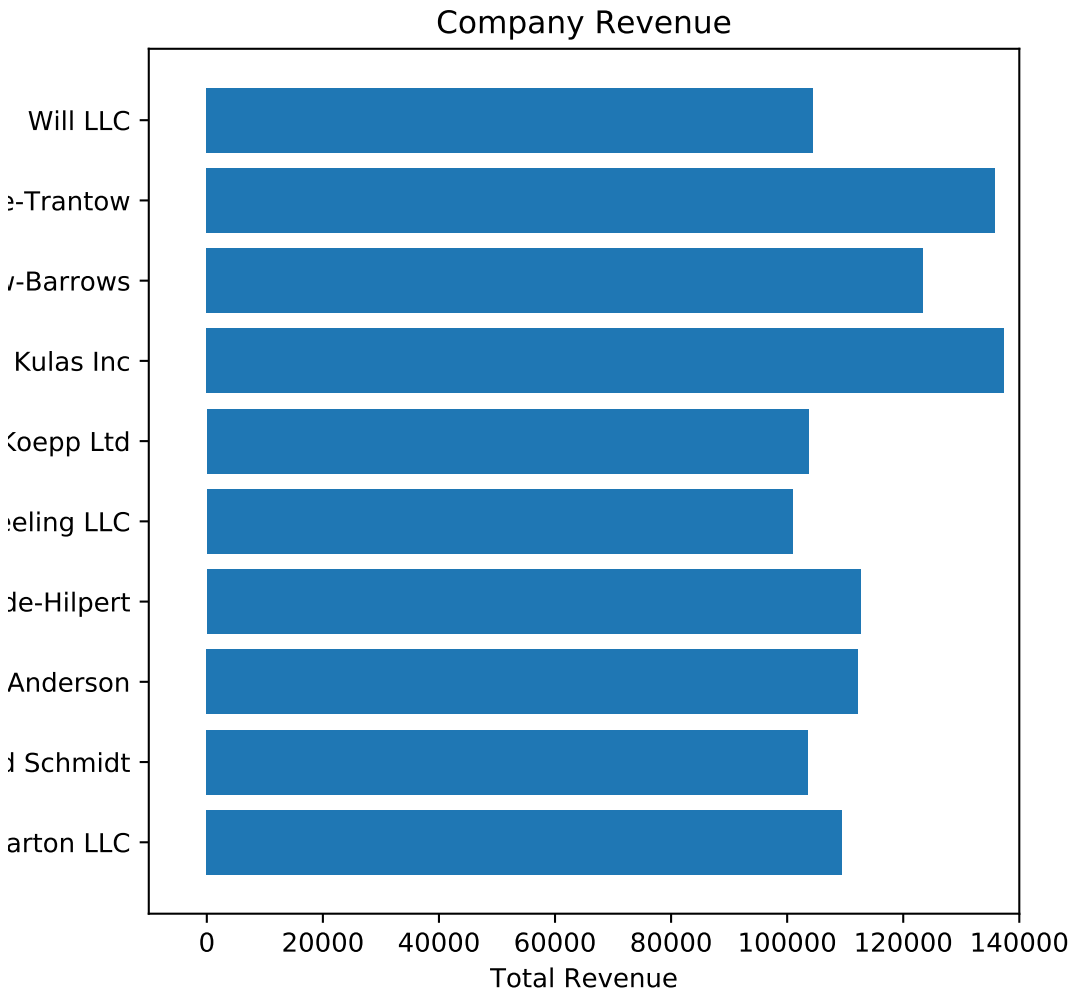
```
plt.style.use('default')
fig, ax = plt.subplots()
plt.show()
```



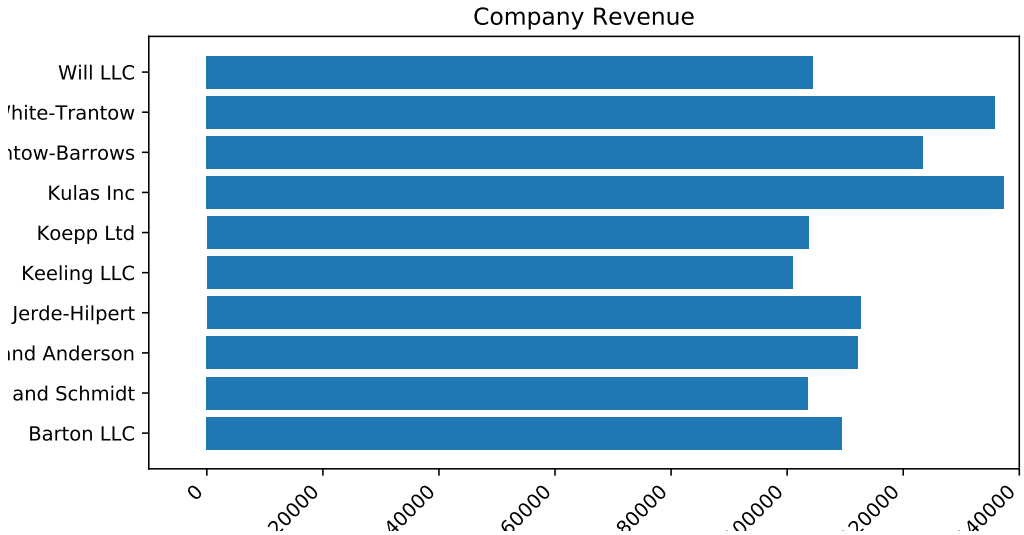
```
fig, ax = plt.subplots()
ax.barh(group_names, group_data);
plt.show()
```




```
fig, ax = plt.subplots()
ax.barh(group_names, group_data)
ax.set(xlim = [-10000, 140000], xlabel = 'Total Revenue', ylabel = 'Company',
       title = 'Company Revenue');
plt.show()
```




```
fig, ax = plt.subplots(figsize=(8, 4))
ax.barh(group_names, group_data)
labels = ax.get_xticklabels()
plt.setp(labels, rotation=45, horizontalalignment='right')
ax.set(xlim=[-10000, 140000], xlabel='Total Revenue', ylabel='Company',
       title='Company Revenue');
plt.show()
```



After you have created your figure, you do need to save it to disk so that you can use it in your Word or Markdown or PowerPoint document. You can see the formats available.

```
fig.canvas.get_supported_filetypes()
```

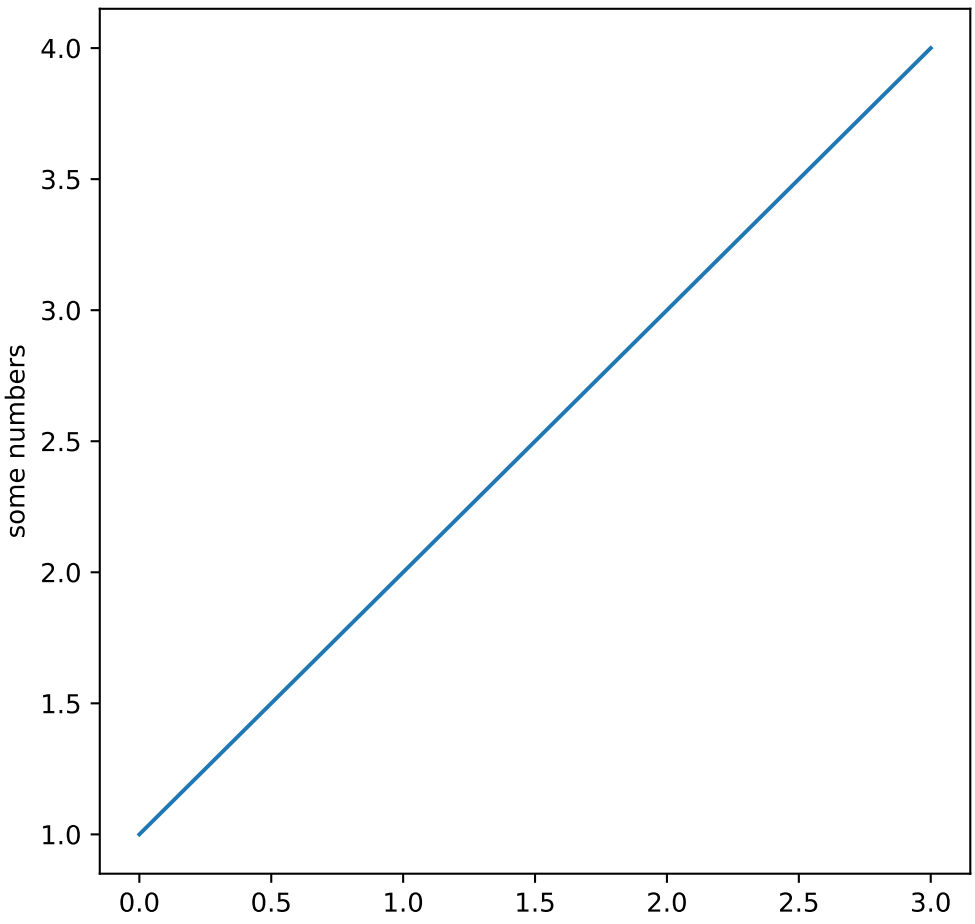
The type will be determined by the ending of the file name. You can add some options depending on the type. I'm showing an example of saving the figure to a PNG file. Typically I'll save figures to a vector graphics format like PDF, and then convert into other formats, since that results in minimal resolution loss. You of course have the option to save to your favorite format.

```
# fig.savefig('sales.png', dpi = 300, bbox_inches = 'tight')
```

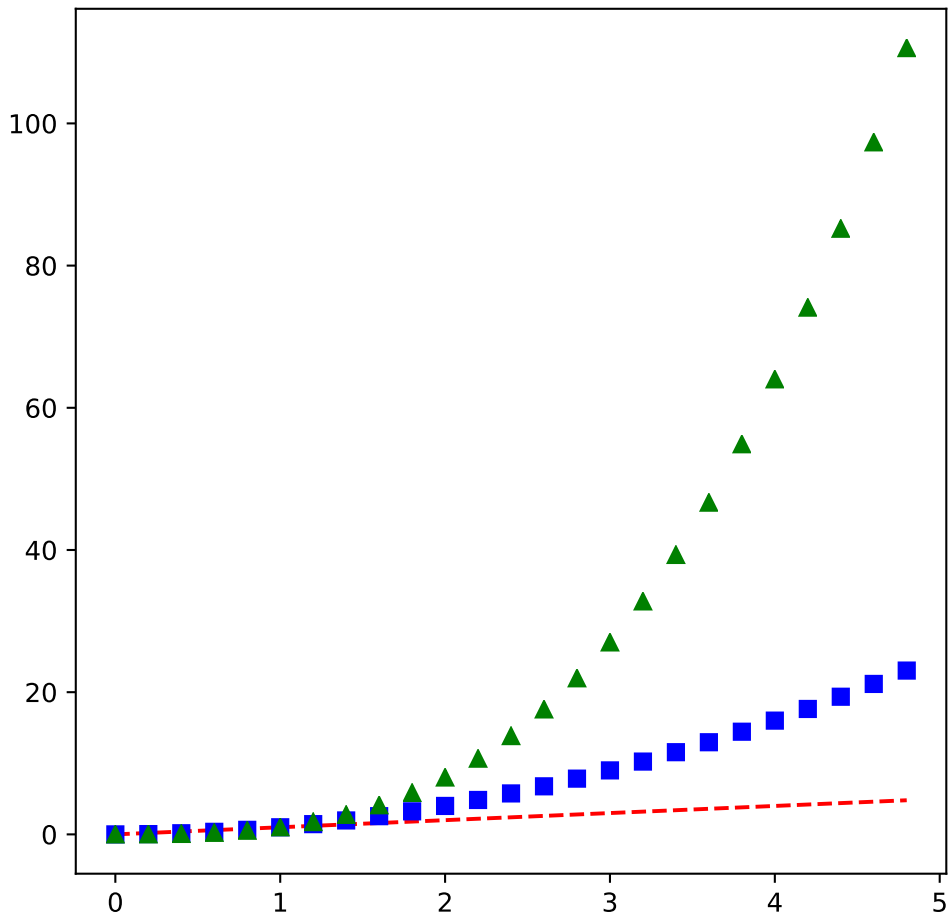
5.7.1. Matlab-like plotting

`matplotlib` was originally developed to emulate Matlab. Though this kind of syntax is no longer recommended, it is still available and may be of use to those coming to Python from Matlab or Octave.

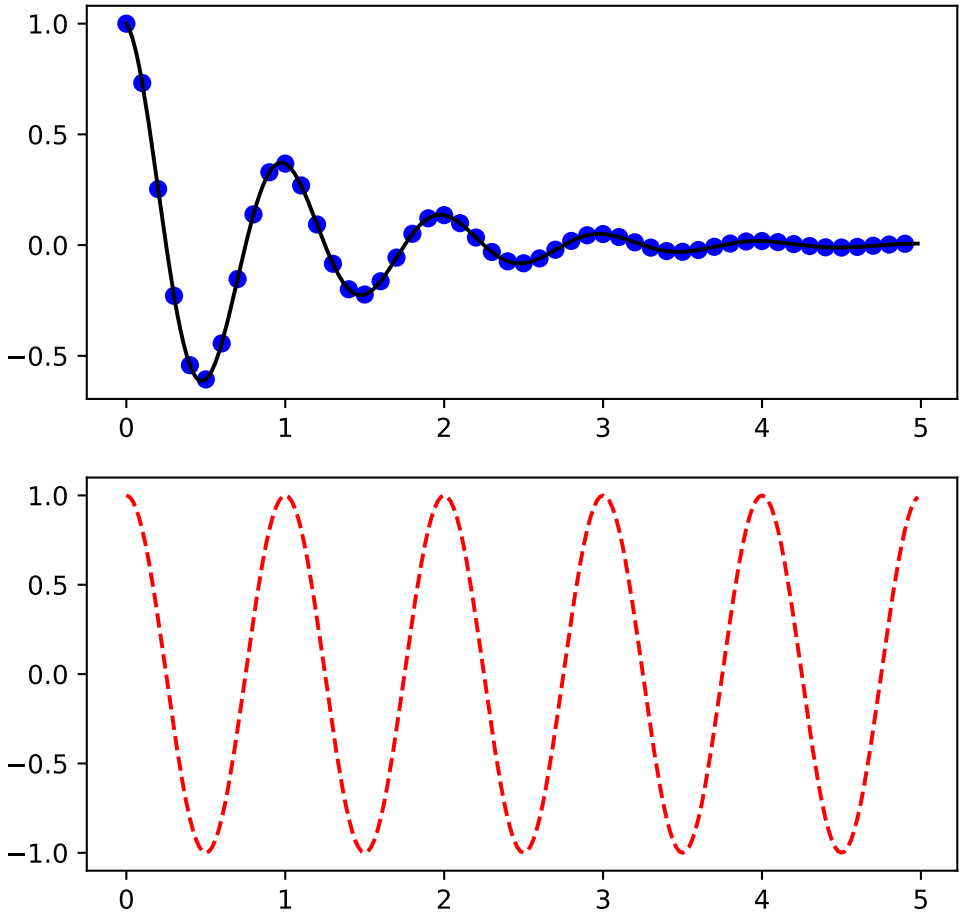
```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4]);
plt.ylabel('some numbers');
plt.show()
```




```
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^');
plt.show()
```




```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure();
plt.subplot(211);
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k');

plt.subplot(212);
plt.plot(t2, np.cos(2*np.pi*t2), 'r--');
plt.show()
```



5.8. Resources

A really nice online resource for learning data visualization in Python is the Python Graph Gallery. This site has many examples of different kinds of plots using `pandas`, `seaborn` and `matplotlib`

6. Statistical analysis

6.1. Introduction

Statistical analysis usually encompasses 3 activities in a data science workflow. These are (a) descriptive analysis, (b) hypothesis testing and (c) statistical modeling. Descriptive analysis refers to a description of the data, which includes computing summary statistics and drawing plots. Hypothesis testing usually refers to statistically seeing if two (or more) groups are different from each other based on some metrics. Modeling refers to fitting a curve to the data to describe the relationship patterns of different variables in a data set.

In terms of Python packages that can address these three tasks:

Task	Packages
Descriptive statistics	pandas, numpy, matplotlib, seaborn
Hypothesis testing	scipy, statsmodels
Modeling	statsmodels, lifelines, scikit-learn

6.2. Descriptive statistics

Descriptive statistics that are often computed are the mean, median, standard deviation, inter-quartile range, pairwise correlations, and the like. Most of these functions are available in `numpy`, and hence are available in `pandas`. We have already seen how we can compute these statistics and have even computed grouped statistics. For example, we will compute these using the diamonds dataset

```
import numpy as np
import scipy as sc
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
diamonds = pd.read_csv('data/diamonds.csv.gz')
```

```
diamonds.groupby('color')['price'].agg([np.mean, np.median, np.std])
```

There were other examples we saw yesterday along these lines. Refer to both the `python_tools_ds` and `python_pandas` documents

6.3. Simulation and inference

Hypothesis testing is one of the areas where statistics is often used. There are functions for a lot of the standard statistical tests in `scipy` and `statsmodels`. However, I'm going to take a little detour to see if we can get some understanding of hypothesis tests using the powerful simulation capabilities of Python. We'll visit the in-built functions available in `scipy` and `statsmodels` as well.

6.3.1. Simulation and hypothesis testing

Question: You have a coin and you flip it 100 times. You get 54 heads. How likely is it that you have a fair coin?

We can simulate this process, which is random, using Python. The process of heads and tails from coin tosses can be modeled as a **binomial** distribution. We can repeat this experiment many many times on our computer, making the assumption that we have a fair coin, and then seeing how likely what we observed is under that assumption.

> Simulation under reasonable assumptions is a great way to understand our data and the underlying data generating processes. In the modern era, it has most famously been used by Nate Silver of ESPN to simulate national elections in the US. There are many examples in engineering where simulations are done to understand a technology and figure out its tolerances and weaknesses, like in aircraft testing. It is also commonly used in epidemic modeling to help understand how an epidemic would spread under different conditions.

```
rng = np.random.RandomState(205) # Seed the random number generator

x = rng.binomial(100, 0.5, 100000) # Simulate 100,000 experiments of tossing a fair coin

sns.distplot(x, kde=True, rug=False)
plt.axvline(54, color = 'r'); # What we observed
plt.xlabel('Number of heads');
```

```
pd.Series(x).describe() # We convert to pd.Series to take advantage of the `describe` fu
```

What we see from the histogram and the description of the data above is the patterns in data we would expect if we repeated this random experiment. We can already make some observations. First, we do see that the average number of heads we expect to get is 50, which validates that our experiment is using a fair coin. Second, we can reasonably get as few as 27 heads and as many as 72 heads even with a fair coin. In fact, we could look at what values we would expect to see 95% of the time.

```
np.quantile(x, [0.025, 0.975])
```

This says that 95% of the time we'll see values between 40 and 60. (This is **not** a confidence interval. This is the actual results of a simulation study. A confidence interval would be computed based on a **single** experiment, assuming a binomial distribution. We'll come to that later).

So how likely would we be to see the 54 heads in 100 tosses assuming a fair coin? This can be computed as the proportion of experiments

```
np.mean(x > 54) # convince yourself of this
```

This is what would be considered the *p-value* for the test that the coin is fair.

> The p-value of a statistical hypothesis test is the likelihood that we would see an outcome at least as extreme as we observed under the assumption that the null hypothesis (H0) that we chose is actually true.
>
> In our case, that null hypothesis is that the coin we're tossing is fair. The p-value **only** gives evidence against the null hypothesis, but does **not** give evidence for the null hypothesis. In other words, if the p-value is small (smaller than some threshold we deem reasonable), then we can claim evidence against the null hypothesis, but if the p-value is large, we cannot say the null hypothesis is true.

What happens if we increase the number of tosses, and we look at the proportion of heads. We observe 54% heads.

```
rng = np.random.RandomState(205)
x = rng.binomial(10000, 0.5, 100000)/10000
sns.distplot(x)
plt.axvline(0.54, color = 'r')
plt.xlabel('Proportion of heads');
```

```
pd.Series(x).describe()
```

Well, that changed the game significantly. If we up the number of coin tosses per experiment to 10,000, so 100-fold increase, then we do not see very much variation in the proportion of tosses that are heads.

> This is expected behavior because of a statistical theorem called the *Law of Large Numbers*, which essentially says that if you do larger and larger sized random experiments with the same experimental setup, your estimate of the true population parameter (in this case the true chance of getting a head, or 0.5 for a fair coin) will become more and more precise.

Now we see that for a fair coin, we should reasonably see between 47.8% and 52% of tosses should be heads. This is quite an improvement from the 27%-72% range we saw with 100 tosses.

We can compute our p-value in the same way as before.

```
np.mean(x > 0.54)
```

So we would never see 54% of our tosses be heads if we tossed a fair coin 10,000 times. Now, with a larger experiment, we would **reject** our null hypothesis H0 that we have a fair coin.

So same observation, but more data, changes our *inference* from not having sufficient evidence to say that the coin isn't fair to saying that it isn't fair quite definitively. This is directly due to the increased precision of our estimates and thus our ability to differentiate between much smaller differences in the truth.

Let's see a bit more about what's going on here. Suppose we assume that the coin's true likelihood of getting a head is really 0.55, so a very small bias towards heads.

> Food for thought: Is the difference between 0.50 and 0.54 worth worrying about? It probably depends.

We're going to compare what we would reasonably see over many repeated experiments given the coin has a 0.50 (fair) and a 0.55 (slightly biased) chance of a head. First, we'll do experiments of 100 tosses of a coin.

```
rng = np.random.RandomState(205)
x11 = rng.binomial(100, 0.5, 100000)/100 # Getting proportion of heads
x12 = rng.binomial(100, 0.55, 100000)/100

sns.distplot(x11, label = 'Fair')
sns.distplot(x12, label = 'Biased')
plt.xlabel('Proportion of heads')
plt.legend();
```

We see that there is a great deal of overlap in the potential outcomes over 100,000 repetitions of these experiments, so we have a lot of uncertainty about which model (fair or biased) is the truth.

Now, if we up our experiment to 10,000 tosses of each coin, and again repeat the experiment 100,000 times,

```
rng = np.random.RandomState(205)
x21 = rng.binomial(10000, 0.5, 100000)/10000
x22 = rng.binomial(10000, 0.55, 100000)/10000

sns.distplot(x21, label = 'Fair')
sns.distplot(x22, label = 'Biased')
plt.xlabel('Proportion of heads')
plt.legend();
```

We now find almost no overlap between the potential outcomes, so we can very easily distinguish the two models. This is part of what gathering more data (number of tosses) buys you.

We typically measure this ability to distinguish between two models using concepts of *statistical power*, which is the likelihood that we would find an observation at least as extreme as what we observed, under the **alternative** model (in this case, the biased coin model). We can calculate the statistical power quite easily for the two sets of simulated experiments. Remember, we observed 54% heads in our one instance of each experiment that we actually observed. By doing simulations, we're "playing God" and seeing what could have happened, but in practice we only do the experiment once (how many clinical trials of an expensive drug would you really want to do?).

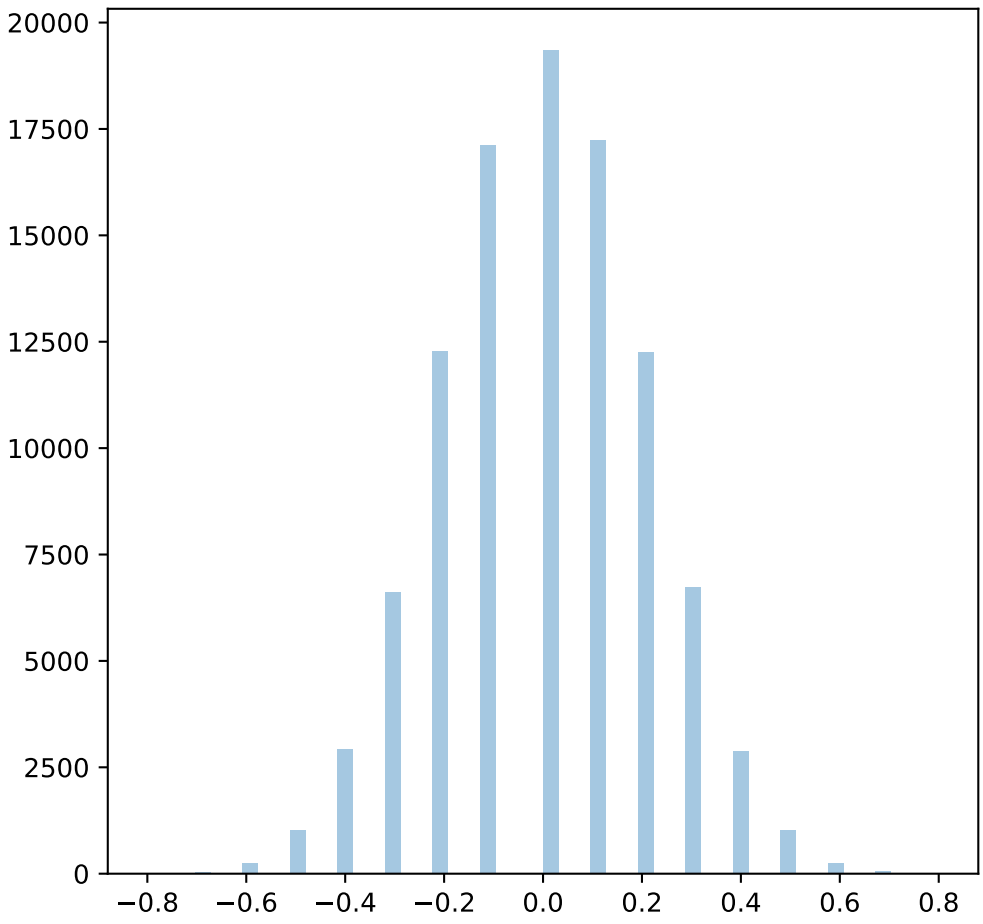
```
pval1 = np.mean(x11 > 0.54)
pval2 = np.mean(x21 > 0.54)

power1 = np.mean(x12 > 0.54)
power2 = np.mean(x22 > 0.54)
```

```
print('The p-value when n=100 is ', np.round(pval1, 2))
print('The p-value when n=10,000 is ', np.round(pval2, 2))
print('Statistical power when n=100 is ', np.round(power1, 2))
print('Statistical power when n=10,000 is ', np.round(power2, 2))
```

So as n goes up, the p-value for the same experimental outcome goes down and the statistical power goes up. This is a general rule with increasing sample size.

This idea can be used to design a two-armed experiment. Suppose we are looking at the difference in proportion of mice who gained weight between a wild-type mouse and a knockout variant. Since mice are expensive, let's limit the number of mice we'll use in each arm to 10. We expect 30% of the wild-type mice to gain weight, and expect a higher proportion of the knockouts will gain weight. This is again the setup for a binomial experiment, with the number of "coin tosses" being 10 for each of the arms. We're going to do two sets of experiments, one for the WT and one for the KO, and see the difference in proportions of weight gain ('heads') between them, and repeat it 100,000 times.

```
rng = np.random.RandomState(304)
N = 10
weight_gain_wt0 = rng.binomial(N, 0.3, 100000)/N # Get proportion
weight_gain_ko0 = rng.binomial(N, 0.3, 100000)/N # Assume first (null hypothesis) that t

diff_weight_gain0 = weight_gain_ko0 - weight_gain_wt0
sns.distplot(diff_weight_gain0, kde=False); # Since we only have 10 mice each, this hist
                                            # No matter!
```



We usually design the actual test by choosing a cutoff in the difference in proportions and stating that we will reject the null hypothesis if our observed difference exceeds this cutoff. We choose the cutoff so that the p-value of the cutoff is some pre-determined error rate, typically 0.05 or 5% (This is not golden or set in stone. We'll discuss this later). Let's find that cutoff from this simulation. This will correspond to the 95th percentile of this simulated distribution.

```
np.round(np.quantile(diff_weight_gain0, 0.95), 2)
```

This means that at least 5% of the values will be 0.3 or bigger. In fact, this proportion is

```
np.mean(diff_weight_gain0 > 0.3)
```

So we'll take 0.3 as the cutoff for our test (It's fine if the Type 1 error is more than 0.05. If we take the next largest value in the simulation, we dip below 0.05). We're basically done specifying the testing rule.

What we (and reviewers) like to know at this point is, what is the difference level for which you might get 80% power. The thinking is that if the true difference was, say, $p > 0$ rather than 0 (under the null hypothesis), we would reject the null hypothesis, i.e., get our observed difference to be more than 0.3, at least 80% of the time. We want to find out how big that value of p is. In other words, what is the level of difference in proportions at which we can be reasonably certain that our test will REJECT H0, given our sample size, when the true difference in proportions is p. Another way of saying this is how big does the difference in true proportions have to be before we would be fairly confident statistically of distinguishing that we have a difference between the two groups given our chosen sample size, i.e., fairly small overlaps in the two competing distributions.

We can also do this using simulation, by keeping the WT group at 0.3, increasing the KO group gradually, simulating the distribution of the difference in proportion and seeing at what point we get to a statistical power of about 80%. Recall, we've already determined that our test will reject H0 when the observed difference is greater than 0.3

```
p1 = np.linspace(0.3, 0.9, 100)
power = np.zeros(len(p1))
for i, p in enumerate(p1):
    weight_gain_wt1 = rng.binomial(N, 0.3, 100000)/N
    weight_gain_ko1 = rng.binomial(N, p, 100000)/N
    diff_weight_gain1 = weight_gain_ko1 - weight_gain_wt1
    power[i] = np.mean(diff_weight_gain1 > 0.3)
```

```
sns.lineplot(p1, power)
plt.axhline(0.8, color = 'black', linestyle = '--');
plt.ylabel('Statistical power')
plt.xlabel('Proportion in KO mice');
```

```
np.round(p1[np.argmin(np.abs(power - 0.8))] - 0.3, 2) # Find the location in the p1 arra
```

So to get to 80% power, we would need the true difference in proportion to be 0.48, or that at least 78% of KO mice should gain weight on average. This is quite a big difference, and its probably not very interesting scientifically to look for such a big difference, since it's quite unlikely.

If we could afford 100 mice per arm, what would this look like?

```
rng = np.random.RandomState(304)
N = 100
weight_gain_wt0 = rng.binomial(N, 0.3, 100000)/N # Get proportion
weight_gain_ko0 = rng.binomial(N, 0.3, 100000)/N # Assume first (null hypothesis) that t

diff_weight_gain0 = weight_gain_ko0 - weight_gain_wt0
cutoff = np.quantile(diff_weight_gain0, 0.95)

p1 = np.linspace(0.3, 0.9, 100)
power = np.zeros(len(p1))
for i, p in enumerate(p1):
```

```
    weight_gain_wt1 = rng.binomial(N, 0.3, 100000)/N
    weight_gain_ko1 = rng.binomial(N, p, 100000)/N
    diff_weight_gain1 = weight_gain_ko1 - weight_gain_wt1
    power[i] = np.mean(diff_weight_gain1 > cutoff)

sns.lineplot(p1, power)
plt.axhline(0.8, color = 'black', linestyle = '--');
plt.ylabel('Statistical power')
plt.xlabel('Proportion in KO mice');
```

```
np.round(p1[np.argmin(np.abs(power - 0.8))] - 0.3, 2)
```

The minimum detectable difference for 80% power is now down to 0.17, so we'd need the KO mice in truth to show weight gain 47% of the time, compared to 30% in WT mice. This is more reasonable scientifically as a query.

6.3.2. A permutation test

A permutation test is a 2-group test that asks whether two groups are different with respect to some metric. Let us look at a breast cancer proteomics experiment to illustrate this. The experimental data contains protein expression for over 12 thousand proteins, along with clinical data. We can ask, for example, whether a particular protein expression differs by ER status.

```
brca = pd.read_csv('data/brca.csv')
brca.head()
```

We will first do the classical t-test, that is available in the `scipy` package.

```
import scipy as sc
import statsmodels as sm
test_probe = 'NP_001193600'

tst = sc.stats.ttest_ind(brca[brca['ER Status']=='Positive'][test_probe], # Need [] sinc
                   brca[brca['ER Status']=='Negative'][test_probe],
                  nan_policy = 'omit')
np.round(tst.pvalue, 3)
```

The idea about a permutation test is that, if there is truly no difference then it shouldn't make a difference if we shuffled the labels of ER status over the study individuals. That's literally what we will do. We will do this several times, and look at the average difference in expression each time. This will form the null distribution under our assumption of no differences by ER status. We'll then see where our observed data falls, and then be able to compute a p-value.

The difference between the simulations we just did and a permutation test is that the permutation test is based only on the observed data. No particular models are assumed and no new data is simulated. All we're doing is shuffling the labels among the subjects, but keeping their actual data intact.

```
nsim = 10000

rng = np.random.RandomState(294)
x = np.where(brca['ER Status']=='Positive', -1, 1)
y = brca[test_probe].to_numpy()

obs_diff = np.nanmean(y[x==1]) - np.nanmean(y[x==-1])

diffs = np.zeros(nsim)
for i in range(nsim):
    x1 = rng.permutation(x)
    diffs[i] = np.nanmean(y[x1==1]) - np.nanmean(y[x1 == -1])
```

```
sns.distplot(diffs)
plt.axvline(x = obs_diff, color ='r');
```

```
pval = np.mean(np.abs(diffs) > np.abs(obs_diff))
f"P-value from permutation test is {pval}"
```

This is pretty close to what we got from the t-test

6.3.3. Testing many proteins

We could do the permutation test all the proteins using the array operations in numpy

```
expr_names = [u for u in list(brca.columns) if u.find('NP') > -1] # Find all column name

exprs = brca[expr_names] # Extract the protein data
```

```
x = np.where(brca['ER Status']=='Positive', -1, 1)
obs_diffs = exprs[x==1].mean(axis=0)-exprs[x==-1].mean(axis=0)
```

```
nsim = 1000
diffs = np.zeros((nsim, exprs.shape[1]))
for i in range(nsim):
    x1 = rng.permutation(x)
    diffs[i,:] =exprs[x1==1].mean(axis=0) - exprs[x1==-1].mean(axis=0)
```

```
pvals = np.zeros(exprs.shape[1])
len(pvals)
```

```
for i in range(len(pvals)):
    pvals[i] = np.mean(diffs[:,i] > obs_diffs.iloc[i])

sns.distplot(pvals);
```

This plot shows that there is probably some proteins which are differentially expressed between ER+ and ER- patients. (If no proteins had any difference, this histogram would be flat, since the p-values would be uniformly distributed). The ideas around Gene Set Enrichment Analysis (GSEA) can also be applied here.

```
exprs_shortlist = [u for i, u in enumerate(list(exprs.columns)) if pvals[i] < 0.0001 ]

len(exprs_shortlist)
```

This means that, if we considered a p-value cutoff for screening at 0.0001, we would select 486 of the 12395 proteins for further study.

6.3.4. Getting a confidence interval using the bootstrap

We can use simulations to obtain a model-free confidence interval for particular parameters of interest based on our observed data. The technique we will demonstrate is called the bootstrap. The idea is that if we sample with replacement from our observed data to get another data set of the same size as the observed data, and compute our statistic of interest, and then repeat this process many times, then the distribution of our statistic that we will obtain this way will be very similar to the true sampling distribution of the statistic if we could "play God". This has strong theoretical foundations from work done by several researchers in the 80s and 90s.

1. Choose the number of simulations `nsim`
2. for each iteration (1,...,nsim)
    - Simulate a dataset with replacement from the original data.
    - compute and store the statistic
3. Compute the 2.5th and 97.5th percential of the distribution of the statistic. This is your confidence interval.

Let's see this in action. Suppose we tossed a coin 100 times. We're going to find a confidence interval for the proportion of heads from this coin.

```
rng = np.random.RandomState(304)
x = rng.binomial(1, 0.7, 100)
x
```

This gives the sequence of heads (1) and tails (0), assuming the true probability of heads is 0.7.

We now create 100000 bootstrap samples from here.

```
nsim = 100000

boots = np.random.choice(x, (len(x), nsim), replace = True) # sample from the data
boot_estimates = boots.mean(axis = 0) # compute mean of each sample, i.e proportion of h

sns.distplot(boot_estimates);
```

```
np.quantile(boot_estimates, (0.025, 0.975)) # Find 2.5 and 97.5-th percentiles
```

So our 95% bootstrap confidence interval is (0.66, 0.83). Our true value of 0.7 certainly falls in it.

6.4. Classical hypothesis testing

Python has the tools to do classic hypothesis testing as well. Several functions are available in the `scipy.stats` module. The commonly used tests that are available are as follows:

Function	Test
`ttest_1samp`	One-sample t-test
`ttest_ind`	Two-sample t-test
`ttest_rel`	Paired t-test
`wilcoxon`	Wilcoxon signed-rank test (nonparametric paired t-test)
`ranksum`	Wilcoxon rank-sum test (nonparametric 2-sample t-test)
`chi2_contingency`	Chi-square test for independence
`fisher_exact`	Fisher's exact test on a 2x2 contingency table
`f_oneway`	One-way ANOVA
`pearsonr`	Testing for correlation

There are also several tests in `statsmodels.stats`

Functions	Tests
`proportions_ztest`	Test for difference in proportions
`mcnemar`	McNemar's test
`sign_test`	Sign test
`multipletests`	p-value correction for multiple tests
`fdrcorrection`	p-value correction by FDR

We've seen an example of these tests earlier. .

6.5. Regression analysis

The regression modeling frameworks in Python are mainly in `statsmodels`, though some of it can be found in `scikit-learn` which we will see tomorrow. We will use the diamonds dataset for demonstration purposes. We will attempt to model the diamond price against several of the other diamond characteristics.

```
import statsmodels.api as sm
import statsmodels.formula.api as smf # Use the formula interface to statsmodels

mod1 = smf.glm('price ~ carat + clarity + depth + cut + color', data = diamonds).fit()
```

```
mod1.summary()
```

This is the basic syntax for modeling in statsmodels. Let's go through and parse it.

One thing you notice is that we've written a formula inside the model

```
mod1 = smf.glm('price ~ carat + clarity + depth + cut + color', data = diamonds).fit()
```

This is based on another Python package, `patsy`, which allows us to write the model like this. This will read as "price depends on carat, clarity, depth, cut and color". Underneath a lot is going on.

1. color, clarity, and cut are all categorical variables. They actually need to be expanded into dummy variables, so we will have one column for each category level, which is 1 when the diamond is of that category and 0 otherwise.
2. An intercept terms is added
3. The dummy variables are concatenated to the continuous variables
4. The model is run

We can see the dummy variables using `pandas`

```
pd.get_dummies(diamonds)
```

7. Machine Learning using Python

7.1. Scikit-learn

Scikit-learn (`sklearn`) is the main Python package for machine learning. It is a widely-used and well-regarded package. However, there are a couple of challenges to using it given the usual `pandas`-based data munging pipeline.

1. `sklearn` requires that all inputs be numeric, and in fact, `numpy` arrays.
2. `sklearn` requires that all categorical variables by replaced by 0/1 dummy variables
3. `sklearn` requires us to separate the predictors from the outcome. We need to have one `X` matrix for the predictors and one `y` vector for the outcome.

The big issue, of course, is the first point. Given we used `pandas` precisely because we wanted to be able to keep heterogenous data. We have to be able to convert non-numeric data to numeric. `pandas` does help us out with this problem. First of all, we know that all `pandas` Series and DataFrame objects can be converted to `numpy` arrays using the `value` or `to_numpy` functions. Second, we can easily extract a single variable from the data set using either the usual extracton methods or the `pop` function. Third, `pandas` gives us a way to convert all categorical values to numeric dummy variables using the `get_dummies` function. This is actually a more desirable solution than what you will see in cyberspace, which is to use the `OneHotEncoder` function from `sklearn`. If the outcome variable is not numeric, we can `LabelEncoder` function from the `sklearn.preprocessing` submodule.

I just threw a bunch of jargon at you. Let's see what this means.

7.1.1. Transforming the outcome/target

```
import numpy as np
import pandas as pd
import sklearn
import matplotlib.pyplot as plt
import seaborn as sns

iris = pd.read_csv('data/iris.csv')
iris.head()
```

Let's hit the first issue first. We need to separate out the outcome (the variable we want to predict) from the predictors (in this case the sepal and petal measurements).

```
y = iris['species']
X = iris.drop('species', axis = 1) # drops column, makes a copy
```

Another way to do this is

```
y = iris.pop('species')
```

If you look at this, `iris` now only has 4 columns. So we could just use `iris` after the pop application, as the predictor set

We still have to update y to become numeric. This is where the `sklearn` functions start to be handy

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
y
```

Let's talk about this code, since it's very typical of the way the `sklearn` code works. First, we import a method (`LabelEncoder`) from the appropriate `sklearn` module. The second line, `le = LabelEncoder()` works to "turn on" the method. This is like taking a power tool off the shelf and plugging it in to a socket. It's now ready to work. The third line does the actual work. The `fit_transform` function transforms the data you input into it based on the method it is then attached to.

> Let's make a quick analogy. You can plug in both a power washer and a jackhammer to get them ready to go. You can then apply each of them to your driveway. They "transform" the driveway in different ways depending on which tool is used. The washer would "transform" the driveway by cleaning it, while the jackhammer would transform the driveway by breaking it.

There's an interesting invisible quirk to the code, though. The object `le` also got transformed during this process. There were pieces added to it during the `fit_transform` process.

```
le = LabelEncoder()
d1 = dir(le)
```

```
y = le.fit_transform( pd.read_csv('data/iris.csv')['species'])
d2 = dir(le)
set(d2).difference(set(d1)) # set of things in d2 but not in d1
```

So we see that there is a new component added, called `classes_`.

```
le.classes_
```

So the original labels aren't destroyed; they are being stored. This can be useful.

```
le.inverse_transform([0,1,1,2,0])
```

So we can transform back from the numeric to the labels. Keep this in hand, since it will prove useful after we have done some predictions using a ML model, which will give numeric predictions.

7.1.2. Transforming the predictors

Let's look at a second example. The `diamonds` dataset has several categorical variables that would need to be transformed.

```
diamonds = pd.read_csv('data/diamonds.csv.gz')

y = diamonds.pop('price').values
X = pd.get_dummies(diamonds)
```

```
type(X)
```

```
X.info()
```

So everything is now numeric!!. Let's take a peek inside.

```
X.columns
```

So, it looks like the continuous variables remain intact, but the categorical variables got exploded out. Each variable name has a level with it, which represents the particular level it is representing. Each of these variables, called dummy variables, are numerical 0/1 variables. For example, `color_F` is 1 for those diamonds which have color F, and 0 otherwise.

```
pd.crosstab(X['color_F'], diamonds['color'])
```

7.2. The methods

We mentioned a bunch of methods in the slides. Let's look at where they are in `sklearn`

ML method	Code to call it
Decision Tree	`sklearn.tree.DecisionTreeClassifier`, `sklearn.tree.DecisionTreeRegressor`
Random Forest	`sklearn.ensemble.RandomForestClassifier`, `sklearn.ensemble.RandomForestRegressor`
Linear Regression	`sklearn.linear_model.LinearRegression`
Logistic Regression	`sklearn.linear_model.LogisticRegression`
Support Vector Machines	`sklearn.svm.LinearSVC`, `sklearn.svm.LinearSVR`

The general method that the code will follow is :

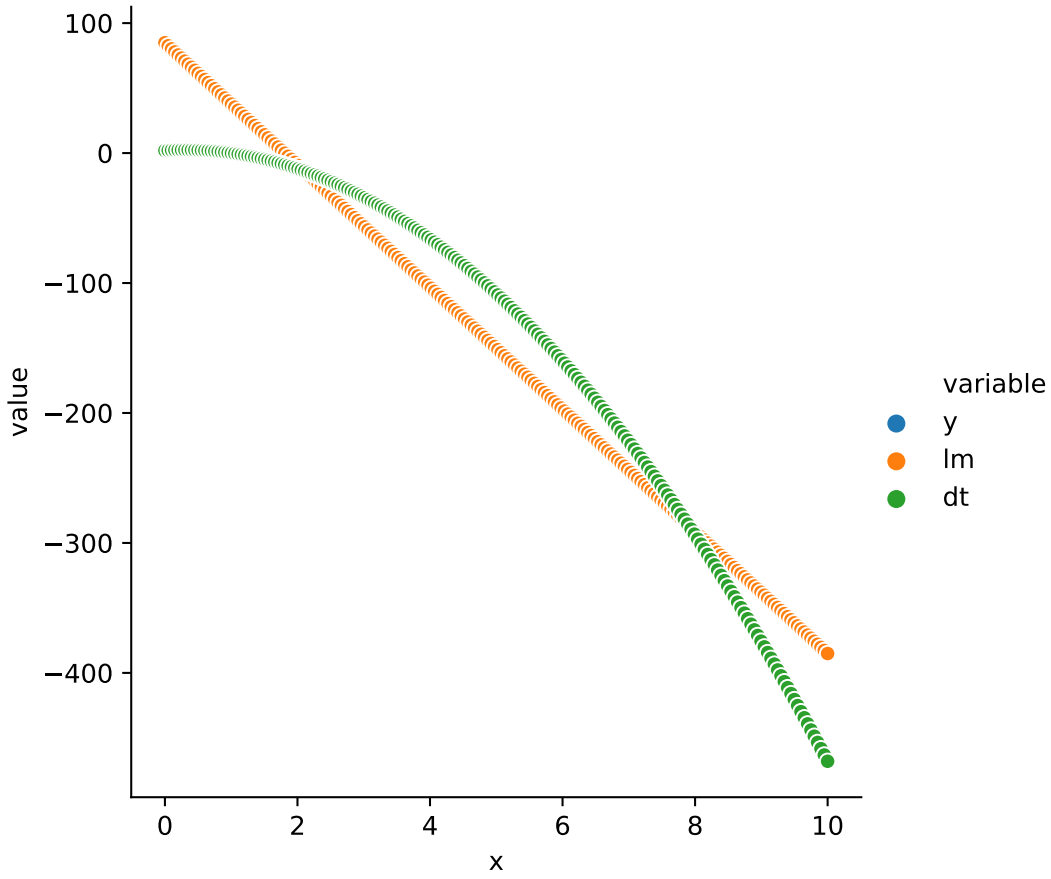
```
from sklearn.... import Machine
machine = Machine(*parameters*)
machine.fit(X, y)
```

7.2.1. A quick example

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

lm = LinearRegression()
dt = DecisionTreeRegressor()
```

Lets manufacture some data

```
x = np.linspace(0, 10, 200)
y = 2 + 3*x - 5*(x**2)
d = pd.DataFrame({'x': x})

lm.fit(d,y)
dt.fit(d, y)

p1 = lm.predict(d)
p2 = dt.predict(d)

d['y'] = y
d['lm'] = p1
d['dt'] = p2

D = pd.melt(d, id_vars = 'x')

sns.relplot(data=D, x = 'x', y = 'value', hue = 'variable')
plt.show()
```



7.3. A data analytic example

```
diamonds = pd.read_csv('data/diamonds.csv.gz')
diamonds.info()
```

First, lets separate out the outcome (price) and the predictors

```
y = diamonds.pop('price')
```

For many machine learning problems, it is useful to scale the numeric predictors so that they have mean 0 and variance 1. First we need to separate out the categorical and numeric variables

```
d1 = diamonds.select_dtypes(include = 'number')
d2 = diamonds.select_dtypes(exclude = 'number')
```

Now let's scale the columns of `d1`

```
from sklearn.preprocessing import scale

bl = scale(d1)
bl
```

Woops!! We get a numpy array, not a `DataFrame`!!

```
bl = pd.DataFrame(scale(d1))
bl.columns = list(d1.columns)
d1 = bl
```

Now, let's recode the categorical variables into dummy variables.

```
d2 = pd.get_dummies(d2)
```

and put them back together

```
X = pd.concat([d1,d2], axis = 1)
```

Next we need to split the data into a training set and a test set. Usually we do this as an 80/20 split. The purpose of the test set is to see how well the model works on an "external" data set. We don't touch the test set until we're done with all our model building in the training set. We usually do the split using random numbers. We'll put 40,000 observations in the training set.

```
ind = list(X.index)
np.random.shuffle(ind)

X_train, y_train = X.loc[ind[:40000],:], y[ind[:40000]]
X_test, y_test = X.loc[ind[40000:],:], y[ind[40000:]]
```

There is another way to do this

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y , test_size = 0.2, random_state
```

Now we will fit our models to the training data. Let's use a decision tree model, a random forest model, and a linear regression.

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor

lm = LinearRegression()
dt = DecisionTreeRegressor()
rf = RandomForestRegressor()
```

Now we will use our training data to fit the models

```
lm.fit(X_train, y_train)
dt.fit(X_train, y_train)
rf.fit(X_train, y_train)
```

We now need to see how well the model fit the data. We'll use the R2 statistic to be our metric of choice to evaluate the model fit.

```
from sklearn.metrics import  r2_score

f"""
Linear regression: {r2_score(y_train, lm.predict(X_train))},
Decision tree: {r2_score(y_train, dt.predict(X_train))},
Random Forest: {r2_score(y_train, rf.predict(X_train))}
"""
```

This is pretty amazing. However, we know that if we try and predict using the same data we used to train the model, we get better than expected results. One way to get a better idea about the true performance of the model when we will try it on external data is to do cross-validation.

In cross-validation, we split the dataset up into 5 equal parts randomly. We then train the model using 4 parts and predict the data on the 5th part. We do for all possible groups of 4 parts. We then consider the overall performance of prediction.

```
from sklearn.model_selection import cross_val_score
cv_score = cross_val_score(dt, X_train, y_train, cv=5)
f"CV error = {np.mean(cv_score)}"
```

If we weren't satisfied with this performance, we could optimize the parameters of the decision tree to see if we could improve performance. The way to do that would be to use `sklearn.model_selection.GridSearchCV`, giving it ranges of the parameters we want to optimize. For a decision tree these would be the maximum depth of the tree, the size of the smallest leaf, and the maximum number of features (predictors) to consider at each split. See `help(DecisionTreeRegressor)` for more details.

So how does this do on the test set?

```
p = dt.predict(X_test)
r2_score(y_test, p)
```

8. String manipulation

String manipulation is one of Python's strong suites. It comes built in with methods for strings, and the `re` module (for *regular expressions*) ups that power many fold.

Strings are objects that we typically see in quotes. We can also check if a variable is a string.

```
a = 'Les Miserable'

type(a)
```

Strings are a little funny. They look like they are one thing, but they can act like lists. In some sense they are really a container of characters. So we can have

```
len(a)
```

```
a[:4]
```

```
a[3:6]
```

The rules are basically the same as lists. To make this explicit, let's consider the word 'bare'. In terms of positions, we can write this out.

index	0	1	2	3
string	b	a	r	e
neg index	-4	-3	-2	-1

We can also slices strings (and lists for that matter) in intervals. So, going back to `a`,

```
a[::2]
```

slices every other character.

Strings come with several methods to manipulate them natively.

```
'White Knight'.capitalize()
"It's just a flesh wound".count('u')
'Almond'.endswith('nd')
'White Knight'.lower()
```

```
'White Knight'.upper()
'flesh wound'.replace('flesh','bullet')
' This is my song    '.strip()
'Hello, hello, hello'.split(',')
```

One of the most powerful string methods is `join`. This allows us to take a list of characters, and then put them together using a particular separator.

```
' '.join(['This','is','my','song'])
```

Also recall that we are allowed "string arithmetic".

```
'g' + 'a' + 'f' + 'f' + 'e'

'a '*5
```

8.0.1. String formatting

In older code, you will see a formal format statement.

```
var = 'horse'
var2 = 'car'

s = 'Get off my {}!'

s.format(var)
s.format(var2)
```

This is great for templates.

```
template_string = """
{country}, our native village
There was a {species} tree.
We used to sleep under it.
"""

print(template_string.format(country='India', species = 'banyan'))
print(template_string.format(country = 'Canada', species = 'maple'))
```

In Python 3.6+, the concept of `f-strings` or formatted strings was introduced. They can be easier to read, faster and have better performance.

```
country = 'USA'
f"This is my {country}!"
```

8.1. Regular expressions

Regular expressions are amazingly powerful tools for string search and manipulation. They are available in pretty much every computer language in some form or the other. I'll provide a short and far from comprehensive introduction here. The website regex101.com is a really good resource to learn and check your regular expressions.

8.1.1. Pattern matching

Syntax	Description
`.`	Matches any one character
`^`	Matches from the beginning of a string
`$`	Matches to the end of a string
`*`	Matches 0 or more repetitions of the previous character
`+`	Matches 1 or more repetitions of the previous character
`?`	Matches 0 or 1 repetitions of the previous character
`{m}`	Matches `m` repetitions of the previous character
`{m,n}`	Matches any number from `m` to `n` of the previous character
`\`	Escape character
`[ ]`	A set of characters (e.g. `[A-Z]` will match any capital letter)
`( )`	Matches the pattern exactly
`|`	OR

9. BioPython

BioPython is a package aimed at bioinformatics work. As with many Python packages, it is opinionated towards the needs of the developers, so might not meet everyone's needs.

You can install BioPython using `conda install biopython`.

We'll do a short example

```
from Bio.Seq import Seq

#create a sequence object
my_seq = Seq("CATGTAGACTAG")

#print out some details about it
print("seq %s is %i bases long" % (my_seq, len(my_seq)))
print("reverse complement is %s" % my_seq.reverse_complement())
print("protein translation is %s" % my_seq.translate())
```

BioPython has capabilities for querying databases like `Entrez`, read sequences, do alignments using FASTA, and the like.