Introduction to Data Science in Python

Prof. Dr. Markus Löcher

05/28/2023

Table of contents

Preface

Python workshop taught by Professor Markus Loecher (HWR Berlin). The workshop will run in-person June 6 - 9 in room VC 14-280 (from 10am to 1pm and 2 to 5pm).

About this python workshop

Goals

This 4 day workshop is intended to introduce participants to the python language. It is designed to provide the solid foundation needed to conduct data analysis and visualization for data science. While no previous experience is required, some basic programming or data science experience is helpful.

I will lean heavily on the book Python for Data Analysis (as well as the Python Data Science Handbook).

The first day will focus on the fundamentals of data types and flow structures while the ultimate goal of the course will be to introduce you to *statistical thinking, data literacy and modeling.*

DataCamp

DataCamp is a pretty good resource for students to learn coding and data analysis skills. By completing the DataCamp courses listed below we would be able to significantly shorten the time we spend on basics and open up more space for data science concepts.

- Introduction to Python
- Intermediate Python

If you have extra time: * Data Manipulation with pandas

And much more advanced and totally optional:

- Python Data Science Toolbox (Part 1)
- Python Data Science Toolbox (Part 2)

- Statistical Thinking in Python (Part 1)
- Statistical Thinking in Python (Part 2)

Coding Environment

The most convenient environment for you to code in might be Google Colab, for which you probably need a gmail account. It does not hurt to look at the 2-minute intro video. If you prefer a real IDE, I would recommend Visual Studio or PyCharm. (I will not be able to help much with the latter though)

Agenda

Day 1: Basic python programming

- basic data types: lists, tuples, dictionaries, strings
- control structures (for, if else, while)
- functions
- numpy arrays: slicing and subsetting, axis
- Probabilistic Simulations
- basic plots

Day 2: pandas and visualization

- pandas Data Frames: slicing and subsetting
- Counting and Summary Statistics
- Handling Files
- Grouped Operations
- plotting with pandas
- Contingency Tables as models

Day 3: Statistical Modeling

- A/B Testing and sampling distributions
- Hypothesis Testing
  - parametric
  - permutation
  - the bootstrap

- regression
  - simple and multiple
  - logistic
  - categorical variables and interactions
  - regularization

Day 4: Machine Learning

- Basic ML tools
  - Cross Validation
  - sklearn
  - Data Cleaning
- Classification and Regression Trees
- Random Forests and Boosting
- Exlainable ML
  - Partial dependence plots
  - SHAP values

If time permits

- Word embeddings such as word2vec

- Sentiment analysis
- Internet scraping
- Topic models

Lecturer

Prof. Dr. Markus Loecher

Professor for Mathematics and Statistics

Berlin School of Economics and Law

linkedin

researchgate

my blog

1 Data Types

Knowing about data types in Python is crucial for several reasons:

- **Correctness and reliability**: Understanding data types helps ensure that your code operates correctly and produces reliable results. By explicitly defining and handling data types, you can avoid unexpected errors or inconsistencies in your code.
- **Memory optimization**: Different data types have varying memory requirements. By choosing appropriate data types, you can optimize memory usage and improve the performance of your code. For example, using integers instead of floating-point numbers can save memory if decimal precision is not necessary.
- **Data manipulation and operations**: Each data type in Python has its own set of operations and methods. Understanding data types allows you to perform specific operations and manipulate data effectively. For example, you can concatenate strings, perform arithmetic calculations with numbers, or iterate over elements in a list.
- **Input validation and error handling**: When receiving input from users or external sources, it is important to validate and handle the data appropriately. Knowing the expected data types allows you to validate inputs, handle errors gracefully, and provide meaningful feedback to users.
- **Interoperability and integration**: Python integrates with various libraries, frameworks, and external systems. Understanding data types helps you exchange data seamlessly between different components of your code or integrate with external systems. For example, when interacting with a database, you need to understand how Python data types map to the database's data types.
- **Code readability and maintainability**: Explicitly defining data types in your code improves readability and makes it easier for other developers to understand and maintain your code. It helps convey your intentions and makes the code self-documenting, reducing the chances of misinterpretation or confusion.

Overall, having knowledge of data types in Python enables you to write robust, efficient, and understandable code that operates correctly with the data it handles. It empowers you to make informed decisions about memory usage, data manipulation, input validation, and code integration, leading to better overall programming proficiency.

1.1 Scalar Types

Python has a small set of built-in types for handling numerical data, strings, Boolean (True or False) values, and dates and time. These "single value" types are sometimes called scalar types, and we refer to them in this book as scalars . See Standard Python scalar types for a list of the main scalar types. Date and time handling will be discussed separately, as these are provided by the datetime module in the standard library.

```
x = 5
type(x)
```

```
int
```

```
y=7.5
type(y)
```

```
float
```

```
type(x+y)
```

```
float
```

1.1.1 Numeric Type

The primary Python types for numbers are int and float. An int can store arbitrarily large numbers:

```
ival = 17239871
ival ** 6
```

```
26254519291092456596965462913230729701102721
```

1.1.2 Booleans

can take only two values: `True` and `False`.

Table 1.1: **?(caption)**

Standard Python scalar types

Type

Description

None

The Python “null” value (only one instance of the None object exists)

str

String type; holds Unicode strings

bytes

Raw binary data

float

Double-precision floating-point number (note there is no separate double type)

bool

```
print(4 == 5)
print(4 < 5)
b = 4 != 5
print(b)
print(int(b))
```

```
False
True
True
1
```

1.2 Strings

Many people use Python for its built-in string handling capabilities. You can write string literals using either single quotes ' or double quotes " (double quotes are generally favored):

```
a = 'one way of writing a string'
b = "another way"

type(a)
```

```
str
```

For multiline strings with line breaks, you can use triple quotes, either `'''` or `"""`:

```
c = """
This is a longer string that
spans multiple lines
"""
```

```
c.count("\n")
```

```
3
```

1.2.0.1 Strings built-in *methods*

Note: All string methods return new values. They do not change the original string.

Method

Description

capitalize()

Converts the first character to upper case

casefold()

Converts string into lower case

center()

Returns a centered string

count()

Returns the number of times a specified value occurs in a string

encode()

Returns an encoded version of the string

endswith()

Returns true if the string ends with the specified value

expandtabs()

Sets the tab size of the string

find()

Searches the string for a specified value and returns the position of where it was found

format()

Formats specified values in a string

format_map()

Formats specified values in a string

index()

Searches the string for a specified value and returns the position of where it was found

isalnum()

Returns True if all characters in the string are alphanumeric

isalpha()

Returns True if all characters in the string are in the alphabet

isascii()

Returns True if all characters in the string are ascii characters

isdecimal()

Returns True if all characters in the string are decimals

isdigit()

Returns True if all characters in the string are digits

isidentifier()

Returns True if the string is an identifier

islower()

Returns True if all characters in the string are lower case

isnumeric()

Returns True if all characters in the string are numeric

isprintable()

Returns True if all characters in the string are printable

isspace()

Returns True if all characters in the string are whitespaces

istitle()

Returns True if the string follows the rules of a title

isupper()

Returns True if all characters in the string are upper case

join()

Converts the elements of an iterable into a string

ljust()

Returns a left justified version of the string

lower()

Converts a string into lower case

lstrip()

Returns a left trim version of the string

maketrans()

Returns a translation table to be used in translations

partition()

Returns a tuple where the string is parted into three parts

replace()

Returns a string where a specified value is replaced with a specified value

rfind()

Searches the string for a specified value and returns the last position of where it was found

rindex()

Searches the string for a specified value and returns the last position of where it was found

rjust()

Returns a right justified version of the string

rpartition()

Returns a tuple where the string is parted into three parts

rsplit()

Splits the string at the specified separator, and returns a list

rstrip()

Returns a right trim version of the string

split()

Splits the string at the specified separator, and returns a list

splitlines()

Splits the string at line breaks and returns a list

startswith()

Returns true if the string starts with the specified value

strip()

Returns a trimmed version of the string

swapcase()

Swaps cases, lower case becomes upper case and vice versa

title()

Converts the first character of each word to upper case

translate()

Returns a translated string

upper()

Converts a string into upper case

zfill()

Fills the string with a specified number of 0 values at the beginning

1.2.1 Methods

Many operations/functions in python are specific to the data type even though we use the same syntax:

```
print(x+y)
print(a+b)
```

```
12.5
one way of writing a stringanother way
```

1.2.1.1 Type conversion

We can often convert from one type to another if it makes sense:

```
str(x)
```

```
'5'
```

```
float(x)
```

```
5.0
```

1.3 Immutable Objects

Mutable objects are those that allow you to change their value or data in place without affecting the object's identity. In contrast, immutable objects don't allow this kind of operation. You'll just have the option of creating new objects of the same type with different values.

In Python, mutability is a characteristic that may profoundly influence your decision when choosing which data type to use in solving a given programming problem. Therefore, you need to know how mutable and immutable objects work in Python.

In Python, variables don't have an associated type or size, as they're labels attached to objects in memory. They point to the memory position where concrete objects live. In other words, a Python variable is a name that refers to or holds a reference to a concrete object. In contrast, Python objects are concrete pieces of information that live in specific memory positions on your computer.

The main takeaway here is that variables and objects are two different animals in Python:

- **Variables** hold references to objects.
- **Objects** live in concrete memory positions.

Read more about this topic

Strings and tuples are immutable:

```
a = "this is a string"

a[10] = "f"
```

```
TypeError: 'str' object does not support item assignment
```

1.4 Tuples

A tuple is a fixed-length, immutable sequence of Python objects which, once assigned, cannot be changed. The easiest way to create one is with a comma-separated sequence of values wrapped in parentheses:

```
tup = (4, 5, 6)
print(tup)
tup = (4, "Ray", 6)
print(tup)
#In many contexts, the parentheses can be omitted
tup = 4, "Ray", 6
```

```
print(tup)
```

```
(4, 5, 6)
(4, 'Ray', 6)
(4, 'Ray', 6)
```

Elements can be accessed with square brackets [] as with most other sequence types. As in C, C++, Java, and many other languages, sequences are 0-indexed in Python:

```
tup[0]
```

```
4
```

```
#but you cannot change the value:
tup[0] = 3
```

```
TypeError: 'tuple' object does not support item assignment
```

You can concatenate tuples using the + operator to produce longer tuples:

```
(4, None, 'foo') + (6, 0) + ('bar',)
```

```
(4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating that many copies of the tuple:

```
('foo', 'bar') * 4
```

```
('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

1.4.0.1 Unpacking tuples

If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the righthand side of the equals sign:

```
tup = (4, 5, 6)
a, b, c = tup
```

2 Lists and Loops

In this lesson we will get to know and become experts in:

1. Lists
    - DataCamp, Introduction to Python, Chap 2
2. Loops
    - DataCamp, Intermediate Python, Chap 4
3. Conditions
    - DataCamp, Intermediate Python, Chap 3

2.0.1 List

In contrast with tuples, lists are variable length and their contents can be modified in place. Lists are mutable. You can define them using square brackets `[]` or using the `list` type function:

```
fam = [1.73, 1.68, 1.71, 1.89]
fam = list((1.73, 1.68, 1.71, 1.89))
fam
```

```
[1.73, 1.68, 1.71, 1.89]
```

You can `sort`, `append`,`insert`, concatenate, ...

```
#sorting is in place!
fam.sort()
fam
```

```
fam.append(2.05)
fam
```

```
fam + fam
```

```
[1.73, 1.68, 1.71, 1.89, 1.73, 1.68, 1.71, 1.89]
```

Lists can

- Contain any type
- Contain different types

```
fam2 = ["liz", 1.73, "emma", 1.68, "mom", 1.71, "dad", 1.89]
fam2
```

```
['liz', 1.73, 'emma', 1.68, 'mom', 1.71, 'dad', 1.89]
```

2.0.2 List of lists

```
fam3 = [["liz", 1.73],
          ["emma", 1.68],
          ["mom", 1.71],
          ["dad", 1.89]]
fam3
```

```
[['liz', 1.73], ['emma', 1.68], ['mom', 1.71], ['dad', 1.89]]
```

2.0.3 Slicing

You can select sections of most sequence types by using slice notation, which in its basic form consists of start:stop passed to the indexing operator []:

```
fam[1:3]
```

```
[1.68, 1.71]
```

While the element at the start index is included, the stop index is not included, so that the number of elements in the result is stop - start.

Either the start or stop can be omitted, in which case they default to the start of the sequence and the end of the sequence, respectively:

```
print(fam[1:])
print(fam[:3])
```

```
[1.68, 1.71, 1.89]
[1.73, 1.68, 1.71]
```

Negative indices slice the sequence relative to the end:

```
fam[-4:]
```

```
[1.73, 1.68, 1.71, 1.89]
```

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See this helpful illustration of slicing with positive and negative integers. In the figure, the indices are shown at the "bin edges" to help show where the slice selections start and stop using positive or negative indices.

Illustration of Python slicing conventions

2.0.4 Manipulating lists of lists

The following list of lists contains names of sections in a house and their area.

1. Extract the area corresponding to kitchen
2. String Tasks:
    - Extract the first letters of each string
    - Capitalize all strings
    - Replace all occurrences of "room" with "rm"
    - count the number of "l" in "hallway"
3. Insert a "home office" with area 10.75 after living room
4. Append the total area to the end of the list
5. **Boolean** operations:
    - Generate one True and one False by comparing areas
    - Generate one True and one False by comparing names

```
house = [['hallway', 11.25],
 ['kitchen', 18.0],
 ['living room', 20.0],
 ['bedroom', 10.75],
 ['bathroom', 9.5]]
```

2.0.5 Automation by iterating

for loops are a powerful way of automating MANY otherwise tedious tasks that repeat.

The syntax template is (watch the indentation!):

```
for var in seq :
     expression
```

````
::: {.cell execution_count=23}
``` {.python .cell-code}
#We can use lists:
for f in fam:
    print(f)
```

::: {.cell-output .cell-output-stdout}
```
1.73
1.68
1.71
1.89
```
:::
:::


::: {.cell execution_count=24}
``` {.python .cell-code}
#or iterators
for i in range(len(fam)):
    print(fam[i])
```
````

````
::: {.cell-output .cell-output-stdout}
```
1.73
1.68
1.71
1.89
```
:::
:::


::: {.cell execution_count=26}
``` {.python .cell-code}
#or enumerate
for i,f in enumerate(fam):
    print(fam[i], f)

```


::: {.cell-output .cell-output-stdout}
```
1.73 1.73
1.68 1.68
1.71 1.71
1.89 1.89
```
:::
:::


### Iterators (Optional)

An iterator is an object that contains a countable number of values.

An iterator is an object that can be iterated upon, meaning that you can traverse through al

Technically, in Python, an iterator is an object which implements the iterator protocol, whi

::: {.cell execution_count=5}
``` {.python .cell-code}
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)
````
````

````
print(next(myit))
print(next(myit))

```

::: {.cell-output .cell-output-stdout}
```
apple
banana
```
:::
:::


::: {.cell execution_count=6}
``` {.python .cell-code}
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
```

::: {.cell-output .cell-output-stdout}
```
b
a
```
:::
:::


Strings are also iterable objects, containing a sequence of characters:

::: {.cell execution_count=4}
``` {.python .cell-code}
#funny iterators
print(range(5))
list(range(5))
```

::: {.cell-output .cell-output-stdout}
```
````

```
range(0, 5)
```
:::

::: {.cell-output .cell-output-display execution_count=4}
```
[0, 1, 2, 3, 4]
```
:::
:::

1. Repeat the tasks 2 and 4 from above by using a for loop
    - using `enumerate`
    - using `range`
2. Create two separates new lists which contain only the names and areas separately
3. [Clever Carl](https://nrich.maths.org/2478#:~:text=Gauss%20added%20the%20rows%20pairwise,

$$
\sum_{i=1}^{100}{i}
$$

::: {.cell execution_count=27}
``` {.python .cell-code}

```


::: {.cell-output .cell-output-display execution_count=27}
```
[0, 1, 2, 3, 4]
```
:::
:::

Conditions

The syntax template is (watch the indentation!):

if condition: expression

```
for f in fam:
    if f > 1.8:
        print(f)
```

```
1.89
```

1. Find the **max** of the areas by using `if` inside a for loop
2. Print those elements of the list with
    - area > 15
    - strings that contain "room" (or "rm" after your substitution)

3 Dictionaries and Functions

In this lesson we will get to know and become experts in:

1. Functions
    - DataCamp, Introduction to Python, Chap 3
2. Dictionaries
    - DataCamp, Intermediate Python, Chap 2
3. Introduction to numpy
    - DataCamp, Introduction to Python, Chap 4

3.0.1 Functions

Functions are essential building blocks to **reuse code** and to **modularize code**.

We have already seen and used many **built-in functions/methods** such as `print()`, `len()`, `max()`, `round()`, `index()`, `capitalize()`, etc..

```
areas = [11.25, 18.0, 20.0, 10.75, 10.75, 9.5]
print(max(areas))
print(len(areas))
print(round(10.75,1))
print(areas.index(18.0))
```

```
20.0
6
10.8
1
```

But of course we want to define our own functions as well ! As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function. Functions can also help make your code more readable by giving a name to a group of Python statements.

For example, we computed the BMI previously as follows:

```
height = 1.79
weight = 68.7
bmi = weight/height**2
print(bmi)
```

```
21.44127836209856
```

Functions are declared with the `def` keyword. A function contains a block of code with an optional use of the `return` keyword:

```
def compute_bmi(height, weight):
    return weight/height**2

compute_bmi(1.79, 68.7)
```

```
21.44127836209856
```

Each function can have *positional* arguments and *keyword* arguments. Keyword arguments are most commonly used to specify default values or optional arguments. For example:

```
def compute_bmi(height, weight, ndigits=2):
    return round(weight/height**2, ndigits)

print(compute_bmi(1.79, 68.7))
print(compute_bmi(1.79, 68.7,4))
```

```
21.44
21.4413
```

3.0.1.1 Multiple Return Values

are easily possible in python:

```
def compute_bmi(height, weight, ndigits=2):
    bmi = round(weight/height**2, ndigits)
    #https://www.cdc.gov/healthyweight/assessing/index.html#:~:text=If%20your%20BMI%20is%2
    if bmi < 18.5:
```

```
        status="underweight"
    elif bmi <= 24.9:
        status="healthy"
    elif bmi <= 29.9:
        status="underweight"
    elif bmi >= 30:#note that a simple else would suffice here!
        status="obese"
    return bmi, status

print(compute_bmi(1.79, 68.7))
print(compute_bmi(1.79, 55))
```

```
(21.44, 'healthy')
(17.17, 'underweight')
```

Recall from the previous lab how we

1. found the largest room,
2. computed the sum of integers from 1 to 100

```
#find the maximum area:
areas = [11.25, 18.0, 20.0, 10.75, 10.75, 9.5]
currentMax = areas[0] # initialize to the first area seen

for a in areas:
  if a > currentMax:
    currentMax = a

print("The max is:", currentMax)
```

```
The max is: 20.0
```

```
#Clever IDB students: Compute the sum from 1 to 100:
Total =0

for i in range(101):#strictly speaking we are adding the first  0
  Total = Total + i
  #Total += i

print(Total)
```

3.0.1.2 Tasks

Write your own function

1. to find the min and max of a list
2. to compute the Gauss sum with defaukt values $m = 1, n = 100$

$$\sum_{i=m}^{n} i$$

3.0.1.3 Namespaces and Scope

Functions seem straightforward. But one of the more confusing aspects in the beginning is the concept that we can have **multiple instances** of the same variable!

Functions can access variables created inside the function as well as those outside the function in higher (or even global) scopes. An alternative and more descriptive name describing a variable scope in Python is a *namespace*. Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and is immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed.

Examples:

```
height = 1.79
weight = 68.7
bmi = weight/height**2
#print("height, weight, bmi OUTSIDE the function:",height, weight,bmi)

def compute_bmi(h, w):
    height = h
    weight = w
    bmi = round(weight/height**2,2)
    status="healthy"
    print("height, weight, bmi INSIDE the function:",height, weight,bmi)
    print("status:", status)
    return bmi

compute_bmi(1.55, 50)

print("height, weight, bmi OUTSIDE the function:",height, weight,bmi)
#print(status)
```

```
height, weight, bmi INSIDE the function: 1.55 50 20.81
status: healthy
height, weight, bmi OUTSIDE the function: 1.79 68.7 21.44127836209856
```

3.0.2 Dictionaries

A dictionary is basically a **lookup table**. It stores a collection of key-value pairs, where key and value are Python objects. Each key is associated with a value so that a value can be conveniently retrieved, inserted, modified, or deleted given a particular key.

The dictionary or `dict` may be the most important built-in Python data structure. In other programming languages, dictionaries are sometimes called *hash maps* or *associative arrays*.

```
#This was the house defined as a list of lists:
house = [['hallway', 11.25],
 ['kitchen', 18.0],
 ['living room', 20.0],
 ['bedroom', 10.75],
 ['bathroom', 9.5]]

#Remember all the disadvantages of accessing elements

#Better as a lookup table:
house = {'hallway': 11.25,
    'kitchen': 18.0,
    'living room': 20.0,
    'bedroom': 10.75,
    'bathroom': 9.5}
```

```
europe = {'spain':'madrid', 'france' : 'paris'}
print(europe["spain"])
print("france" in europe)
print("paris" in europe)#only checks the keys!
europe["germany"] = "berlin"
print(europe.keys())
print(europe.values())
```

```
madrid
True
False
dict_keys(['spain', 'france', 'germany'])
dict_values(['madrid', 'paris', 'berlin'])
```

3.0.3 Dictionaries from lists

How would we convert two lists into a key: value pair dictionary?

Method 1: using `zip`

```
rooms=['hallway', 'kitchen', 'living room', 'bedroom', 'bathroom']
areas=[11.25, 18.0, 20.0, 10.75, 9.5]
#create list of tuples
list(zip(rooms, areas))
```

```
[('hallway', 11.25),
 ('kitchen', 18.0),
 ('living room', 20.0),
 ('bedroom', 10.75),
 ('bathroom', 9.5)]
```

```
dict(zip(rooms, areas))
```

```
{'hallway': 11.25,
 'kitchen': 18.0,
 'living room': 20.0,
 'bedroom': 10.75,
 'bathroom': 9.5}
```

If you need to iterate over both the keys and values, you can use the `items` method to iterate over the keys and values as 2-tuples:

```
#print(list(europe.items()))

for country, capital in europe.items():
    print(capital, "is the capital of", country)
```

```
madrid is the capital of spain
paris is the capital of france
berlin is the capital of germany
```

Note: You can use integers as keys as well. However -unlike in lists- one should not think of them as positional indices!

```
#Assume you have a basement:
house[0] = 21.5
house
```

```
{'hallway': 11.25,
 'kitchen': 18.0,
 'living room': 20.0,
 'bedroom': 10.75,
 'bathroom': 9.5,
 0: 21.5}
```

```
#And there is a difference between the string and the integer index!
house["0"] = 30.5
house
```

```
{'hallway': 11.25,
 'kitchen': 18.0,
 'living room': 20.0,
 'bedroom': 10.75,
 'bathroom': 9.5,
 0: 21.5}
```

Categorize a list of words by their first letters as a dictionary of lists:

```
words = ["apple", "bat", "bar", "atom", "book"]

by_letter = {}

for word in words:
     letter = word[0]
     if letter not in by_letter:
        by_letter[letter] = [word]
     else:
         by_letter[letter].append(word)
```

```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

3.0.3.1 Tasks

1. Find the maximum of the areas of the houses
2. Remove the two last entries.
3. Write a function named word_count that takes a string as input and returns a dictionary with each word in the string as a key and the number of times it appears as the value.

3.0.4 Introduction to numpy

NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.

1. Vectorized, fast mathematical operations.
2. Key features of NumPy is its N-dimensional array object, or ndarray

```
height = [1.79, 1.85, 1.95, 1.55]
weight = [70, 80, 85, 65]

#bmi = weight/height**2
```

```
import numpy as np

height = np.array([1.79, 1.85, 1.95, 1.55])
weight = np.array([70, 80, 85, 65])

bmi = weight/height**2
np.round(bmi,2)
```

```
array([21.84700852, 23.37472608, 22.35371466, 27.05515088])
```

4 Intro to numpy

In this lecture we will get to know and become experts in:

1. Introduction to numpy
    - DataCamp, Introduction to Python, Chap 4
    - Multiple Dimensions
    - Data Summaries in numpy
2. Introduction to **Simulating Probabilistic Events**
    - Generating Data in numpy

```
import numpy as np
from numpy.random import default_rng
```

4.1 Introduction to numpy

NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.

1. Vectorized, fast mathematical operations.
2. Key features of NumPy is its N-dimensional array object, or ndarray

```
height = [1.79, 1.85, 1.95, 1.55]
weight = [70, 80, 85, 65]

#bmi = weight/height**2
```

```
height = np.array([1.79, 1.85, 1.95, 1.55])
weight = np.array([70, 80, 85, 65])

bmi = weight/height**2
bmi
```

```
array([21.84700852, 23.37472608, 22.35371466, 27.05515088])
```

4.1.1 Multiple Dimensions

are handled naturally by numpy, e.g.

```
hw1 = np.array([height, weight])
print(hw1)
print(hw1.shape)
hw2 = hw1.transpose()
print(hw2)
print(hw2.shape)
```

```
[[ 1.79  1.85  1.95  1.55]
 [70.   80.   85.   65.  ]]
(2, 4)
[[ 1.79 70.  ]
 [ 1.85 80.  ]
 [ 1.95 85.  ]
 [ 1.55 65.  ]]
(4, 2)
```

4.1.2 Accessing array elements

is similar to lists but allows for multidimensional index:

```
print(hw2[0,1])
```

```
70.0
```

```
print(hw2[:,0])
```

```
[1.79 1.85 1.95 1.55]
```

```
print(hw2[0])
#equivalent to
print(hw2[0,:])
#shape:
print(hw2[0].shape)
```

```
[ 1.79 70.  ]
[ 1.79 70.  ]
(2,)
```

To select a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
print(hw2[[2,0,1]])
```

```
[[ 1.95 85.  ]
 [ 1.79 70.  ]
 [ 1.85 80.  ]]
```

Negative indices

```
print(hw2)
print("Using negative indices selects rows from the end:")
print(hw2[[-2,-1]])
```

```
[[ 1.79 70.  ]
 [ 1.85 80.  ]
 [ 1.95 85.  ]
 [ 1.55 65.  ]]
Using negative indices selects rows from the end:
[[ 1.95 85.  ]
 [ 1.55 65.  ]]
```

You can pass multiple slices just like you can pass multiple indexes:

```
hw2[:2,:1]
```

```
array([[1.79],
       [1.85]])
```

4.1.2.1 Reshaping

```
np.arange(32).reshape((8, 4))
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

4.1.2.2 Boolean indexing

```
height_gt_185 = hw2[:,0]>1.85
print(height_gt_185)
print(hw2[height_gt_185,1])
```

```
[False False  True False]
[85.]
```

numpy arrays cannot contain elements with different types. If you try to build such a list, some of the elements' types are changed to end up with a homogeneous list. This is known as **type coercion**.

```
print(np.array([True, 1, 2]))
print(np.array(["True", 1, 2]))
print(np.array([1.3, 1, 2]))
```

```
[1 1 2]
['True' '1' '2']
[1.3 1.  2. ]
```

Lots of extra useful functions!

```
np.zeros((2,3))
#np.ones((2,3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
np.column_stack([height, weight])
```

```
array([[ 1.79, 70.  ],
       [ 1.85, 80.  ],
       [ 1.95, 85.  ],
       [ 1.55, 65.  ]])
```

4.2 Data Summaries in numpy

We can compute simple statistics:

```
print(np.mean(hw2))
print(np.mean(hw2, axis=0))
```

```
38.3925
[ 1.785 75.   ]
```

```
print(np.unique([1,1,2,1,2,3,2,2,3]))

print(np.unique([1,1,2,1,2,3,2,2,3], return_counts=True))
```

```
[1 2 3]
(array([1, 2, 3]), array([3, 4, 2], dtype=int64))
```

4.3 Introduction to Simulating Probabilistic Events

4.3.1 Generating Data in numpy

Meet your friends:

1. `np.random.permutation`: Return a random permutation of a sequence, or return a permuted range
2. `np.random.integers`: Draw random integers from a given low-to-high range
3. `np.random.choice`: Generates a random sample from a given 1-D array

```
# Do this (new version)
from numpy.random import default_rng
rng = default_rng()

x= np.arange(10)
print(x)
print(rng.permutation(x))
print(rng.permutation(list('intelligence')))
```

```
[0 1 2 3 4 5 6 7 8 9]
[6 7 9 4 1 0 3 8 2 5]
['t' 'c' 'n' 'l' 'e' 'n' 'i' 'e' 'e' 'l' 'i' 'g']
```

```
print(rng.integers(0,10,5))
print(rng.integers(0,10,(5,2)))
```

```
[7 9 7 9 4]
[[9 0]
 [8 6]
 [6 7]
 [0 5]
 [1 5]]
```

```
rng.choice(x,4)
```

```
array([8, 5, 1, 4])
```

4.3.2 Examples:

- Spotify playlist
- Movie List

```
movies_list = ['The Godfather', 'The Wizard of Oz', 'Citizen Kane', 'The Shawshank Redempt

# pick a random choice from a list of strings.
movie = rng.choice(movies_list,2)
print(movie)
```

```
['The Shawshank Redemption' 'The Godfather']
```

4.4 Birthday "Paradox"

Please enter your birthday on google drive https://forms.gle/CeqyRZ4QzWRmJFvs9

How many people do you think will share a birthday? Would that be a rare, highly unusual event?

How can we find out how likely it is that across n folks in a room at least two share a birthday?

Hint: can we put our random number generators to task ?

```
# Can you simulate 25 birthdays?
from numpy.random import default_rng
rng = default_rng()


#initialize it to be the empty list:
shardBday = []

n = 40

PossibleBDays = np.arange(1,366)
#now "draw" 25 random bDays:
for i in range(1000):# is the 1000 an important number ??
#no it only determines the precision of my estimate !!
  ran25Bdays = rng.choice(PossibleBDays, n, replace = True)
  #it is of utmost importance to allow for same birthdays !!
  #rng.choice(PossibleBDays, 366, replace = False)
```

```
    x , cts = np.unique(ran25Bdays ,return_counts=True)
    shardBday = np.append(shardBday, np.sum(cts>1))#keep this !!
    #shardBday = np.sum(cts>1)
```

```
  #np.sum(shardBday>0)/1000
  np.mean(shardBday > 0)

  #shardBday = 2
```

```
0.893
```

```
  5 != 3 #not equal
```

```
True
```

```
  #Boolean indexing !!
  x[cts > 1]
```

```
array([ 71, 192])
```

```
  x[23]
```

```
182
```

```
  #can you design a coin flip with an arbitary probability p = 0.25
  #simulate 365 days with a 1/4 chance of being sunny

  #fair coin
  coins = np.random.randint(0,2,365)

  np.unique(coins, return_counts=True)
```

```
(array([0, 1]), array([189, 176], dtype=int64))
```

4.4.1 Tossing dice and coins

Let us toss many dice or coins to find out: - the average value of a six-faced die - the variation around the mean when averaging - the probability of various "common hands" in the game Liar's Dice: * Full house: e.g., 66111 * Three of a kind: e.g., 44432 * Two pair: e.g., 22551 * Pair: e.g., 66532

Some real world problems:

1. **Overbooking flights**: airlines
2. **Home Office** days: planning office capacities and minimizing social isolation

5 Intro to pandas

In this Introduction to pandas we will get to know and become experts in:

1. Data Frames
2. Slicing
3. Counting and Summary Statistics
4. Handling Files

Relevant DataCamp lessons:

- Data manipulation with pandas, Chaps 1-4
- Matplotlib, Chap 1

```
import pandas as pd # hopefully colab has this preinstalled
import numpy as np
```

5.1 Introduction to pandas

While numpy offers a lot of powerful numerical capabilities it lacks some of the necessary convenience and natural of handling data as we encounter them. For example, we would typically like to - mix data types (strings, numbers, categories, Boolean, …) - refer to columns and rows by names - summarize and visualize data in efficient pivot style manners

All of the above (and more) can be achieved easily by extending the concept of an array (or a matrix) to a so called **dataframe**.

There are many ways to construct a DataFrame, though one of the most common is from a dictionary of equal-length lists or NumPy arrays:

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)# creates a dataframe out of the data given!
frame.head(3)
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6

```
frame[2,1]#too bad
```

```
#to get the full row: use the .iloc method
frame.iloc[2]
frame.iloc[2,1]
```

```
2002
```

5.1.1 Subsetting/Slicing

We first need to understand the attributes **index** (=rownames) and **columns** (= column names):

```
frame.index
```

```
RangeIndex(start=0, stop=6, step=1)
```

```
#We can set a column as an index:
frame2 = frame.set_index("year")
print(frame2)
#
```

```
       state  pop
year
2000    Ohio  1.5
2001    Ohio  1.7
2002    Ohio  3.6
2001  Nevada  2.4
2002  Nevada  2.9
2003  Nevada  3.2
```

```
#it would be nice to access elements in the same fashion as numpy
#frame2[1,1]
frame["pop"]
```

```
0    1.5
1    1.7
2    3.6
3    2.4
4    2.9
5    3.2
Name: pop, dtype: float64
```

```
frame.pop
```

```
<bound method DataFrame.pop of     state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2>
```

5.1.2 Asking for rows

Unfortunately, we cannot use the simple `[row,col]` notation that we are used to from numpy arrays. (Try asking for `frame[0,1]`)

Instead, row subsetting can be achieved with either the `.loc()` or the `.iloc()` methods. The latter takes integers, the former indices:

```
frame2.loc[2001] #note that I am not using quotes !!
#at first glance this looks like I am asking for the row number 2001 !!
```

year	state	pop
2001	Ohio	1.7
2001	Nevada	2.4

```
frame2.loc[2001,"state"]
```

```
year
2001      Ohio
2001    Nevada
Name: state, dtype: object
```

```
frame.iloc[0]#first row
```

```
state    Ohio
year     2000
pop       1.5
Name: 0, dtype: object
```

```
frame3 = frame.set_index("state", drop=False)
print(frame3)
```

```
         state  year  pop
state
Ohio      Ohio  2000  1.5
Ohio      Ohio  2001  1.7
Ohio      Ohio  2002  3.6
Nevada  Nevada  2001  2.4
Nevada  Nevada  2002  2.9
Nevada  Nevada  2003  3.2
```

```
frame3.loc["Ohio"]
```

state	year	pop
Ohio	2000	1.5
Ohio	2001	1.7
Ohio	2002	3.6

```
frame.iloc[2001]# this does not work because we do not have 2001 rows !
```

```
frame.iloc[0,1]
```

2000

5.1.3 Asking for columns

```
#The columns are also an index:
frame.columns
```

```
Index(['state', 'year', 'pop'], dtype='object')
```

A column in a DataFrame can be retrieved MUCH easier: as a Series either by dictionary-like notation or by using the dot attribute notation:

```
frame["state"]
```

```
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object
```

```
frame.year#equivalent to frame["year"]
```

```
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

5.1.4 Summary Stats

Just like in numpy you can compute sums, means, counts and many other summaries along rows and columns, by specifying the `axis` argument:

```
height = np.array([1.79, 1.85, 1.95, 1.55])
weight = np.array([70, 80, 85, 65])
hw = np.array([height, weight]).transpose()

hw
```

```
array([[ 1.79, 70.  ],
       [ 1.85, 80.  ],
       [ 1.95, 85.  ],
       [ 1.55, 65.  ]])
```

```
df = pd.DataFrame(hw, columns = ["height", "weight"])
print(df)
```

```
   height  weight
0    1.79    70.0
1    1.85    80.0
2    1.95    85.0
3    1.55    65.0
```

```
df = pd.DataFrame(hw , columns = ["height", "weight"],
                  index = ["Peter", "Matilda", "Bee", "Tom"])
print(df)
```

```
         height  weight
Peter      1.79    70.0
Matilda    1.85    80.0
Bee        1.95    85.0
Tom        1.55    65.0
```

Can you extract:

0. All weights
1. Peter's height
2. Bee's full info
3. the average height
4. get all persons with height greater than 180cm

```
#see Lab5
```

```
print(df.mean(axis=0))
print(df.mean(axis=1))# are these averages sensible ?
```

```
height     1.785
weight    75.000
dtype: float64
Peter      35.895
Matilda    40.925
Bee        43.475
Bee        33.275
dtype: float64
```

Some methods are neither reductions nor accumulations. `describe` is one such example, producing multiple summary statistics in one shot:

```
df.describe()
```

	height	weight
count	4.000	4.000000
mean	1.785	75.000000
std	0.170	9.128709
min	1.550	65.000000
25%	1.730	68.750000
50%	1.820	75.000000
75%	1.875	81.250000
max	1.950	85.000000

5.2 Built in data sets

5.3 Gapminder Data

https://www.gapminder.org/fw/world-health-chart/

https://www.ted.com/talks/hans_rosling_the_best_stats_you_ve_ever_seen#t-241405

> You've never seen data presented like this. With the drama and urgency of a sportscaster, statistics guru Hans Rosling debunks myths about the so-called "developing world."

```
!pip install gapminder
#!conda install gapminder
from gapminder import gapminder
#gapminder.to_csv("../datasets/gapminder.csv")
```

```
gapminder
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

```
#find the unique years

#get the years:
gapminder["year"]
np.unique(gapminder.year)
```

```
array([1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, 2002,
       2007])
```

```
#get all rows with year 1952:
#Hint:
#either use Boolean subsetting
gapminder["year"] == 1952
gapminder[gapminder["year"] == 1952]
#or use an index !!
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
12	Albania	Europe	1952	55.230	1282697	1601.056136
24	Algeria	Africa	1952	43.077	9279525	2449.008185
36	Angola	Africa	1952	30.015	4232095	3520.610273
48	Argentina	Americas	1952	62.485	17876956	5911.315053
...	...	...	...	...	...	...
1644	Vietnam	Asia	1952	40.412	26246839	605.066492
1656	West Bank and Gaza	Asia	1952	43.160	1030585	1515.592329
1668	Yemen, Rep.	Asia	1952	32.548	4963829	781.717576
1680	Zambia	Africa	1952	42.038	2672000	1147.388831
1692	Zimbabwe	Africa	1952	48.451	3080907	406.884115

5.4 Handling Files

Get to know your friends

- `pd.read_csv`
- `pd.read_table`
- `pd.read_excel`

```
'''url = "https://drive.google.com/file/d/1oIvCdN15UEwt4dCyjkArekHnTrivN43v/view?usp=share
url='https://drive.google.com/uc?id=' + url.split('/')[-2]
gapminder = pd.read_csv(url, index_col=0)
gapminder.head()'''
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

```
gapminder.sort_values(by="year").head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314

	country	continent	year	lifeExp	pop	gdpPercap
528	France	Europe	1952	67.410	42459667	7029.809327
540	Gabon	Africa	1952	37.003	420702	4293.476475
1656	West Bank and Gaza	Asia	1952	43.160	1030585	1515.592329
552	Gambia	Africa	1952	30.000	284320	485.230659

```
#How many countries?
CtryCts = gapminder["country"].value_counts()
CtryCts
#note the similarity with np.unique(..., return_counts=True)
```

```
Afghanistan          12
Pakistan             12
New Zealand          12
Nicaragua            12
Niger                12
                     ..
Eritrea              12
Equatorial Guinea    12
El Salvador          12
Egypt                12
Zimbabwe             12
Name: country, Length: 142, dtype: int64
```

```
from numpy.random import default_rng
rng = default_rng()
rng.choice(gapminder["country"].unique(),2)
gapminder["year"].unique()
```

```
array([1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, 1997, 2002,
       2007])
```

```
#How meaningful are the column stats?
print(gapminder.mean(axis=0))
gapminder.describe()
```

```
year         1.979500e+03
```

```
lifeExp      5.947444e+01
pop          2.960121e+07
gdpPercap    7.215327e+03
dtype: float64

/var/folders/h4/k73g68ds6xj791sf8cpmlxlc0000gn/T/ipykernel_33611/633466148.py:2: FutureWarni
  print(gapminder.mean(axis=0))
```

	year	lifeExp	pop	gdpPercap
count	1704.00000	1704.000000	1.704000e+03	1704.000000
mean	1979.50000	59.474439	2.960121e+07	7215.327081
std	17.26533	12.917107	1.061579e+08	9857.454543
min	1952.00000	23.599000	6.001100e+04	241.165876
25%	1965.75000	48.198000	2.793664e+06	1202.060309
50%	1979.50000	60.712500	7.023596e+06	3531.846988
75%	1993.25000	70.845500	1.958522e+07	9325.462346
max	2007.00000	82.603000	1.318683e+09	113523.132900

Sort the index before you slice!!

Choose a time range and specific countries

```
gapminder2 = gapminder.set_index("year").sort_index()
gap1982_92 = gapminder2.loc[1982:1992].reset_index()
gap1982_92 = gap1982_92.set_index("country").sort_index()
gap1982_92.loc["Afghanistan":"Albania"]
```

country	year	continent	lifeExp	pop	gdpPercap
Afghanistan	1982	Asia	39.854	12881816	978.011439
Afghanistan	1987	Asia	40.822	13867957	852.395945
Afghanistan	1992	Asia	41.674	16317921	649.341395
Albania	1992	Europe	71.581	3326498	2497.437901
Albania	1987	Europe	72.000	3075321	3738.932735
Albania	1982	Europe	70.420	2780097	3630.880722

```
gap1982_92.loc["Afghanistan":"Albania","lifeExp"].mean()
```

```
56.0585
```

6 Data Summaries

In this lecture we will get to know and become experts in: 1. Data Manipulation with pandas * Handling Files * Counting and Summary Statistics * Grouped Operations 2. Plotting * matplotlib * pandas

And if you want to delve deeper, look at the Advanced topics

Relevant DataCamp lessons:

- Data manipulation with pandas, Chaps 2 and 4
- Matplotlib, Chap 1

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

6.1 Data Manipulation with pandas

While we have seen panda's ability to (i) mix data types (strings, numbers, categories, Boolean, ...) and (ii) refer to columns and rows by names, this library offers a lot more powerful tools for efficiently gaining insights from data, e.g.

- summarize/aggregate data in efficient pivot style manners
- handling missing values
- visualize/plot data

```
!pip install gapminder
from gapminder import gapminder
```

6.2 Handling Files

Get to know your friends

- `pd.read_csv`
- `pd.read_table`

- `pd.read_excel`

But before that we need to connect to our Google drives ! (more instructions can be found here)

```
"Sam" + " Altman"
```

```
'Sam Altman'
```

Counting and Summary Statistics

```
gapminder.sort_values(by="year").head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
528	France	Europe	1952	67.410	42459667	7029.809327
540	Gabon	Africa	1952	37.003	420702	4293.476475
1656	West Bank and Gaza	Asia	1952	43.160	1030585	1515.592329
552	Gambia	Africa	1952	30.000	284320	485.230659

```
#How many countries?
CtryCts = gapminder["country"].value_counts()
CtryCts
#note the similarity with np.unique(..., return_counts=True)
```

```
Afghanistan          12
Pakistan             12
New Zealand          12
Nicaragua            12
Niger                12
                     ..
Eritrea              12
Equatorial Guinea    12
El Salvador          12
Egypt                12
Zimbabwe             12
Name: country, Length: 142, dtype: int64
```

6.3 Grouped Operations

The gapminder data is a good example for wanting to apply functions to subsets to data that correspond to categories, e.g. * by year * by country * by continent

The powerful pandas `.groupby()` method enables exactly this goal rather elegantly and efficiently.

First, think how you could possibly compute the average GDP seprataley for each continent. The `numpy.mean(..., axis=...)` will not help you.

Instead you will have to manually find all continents and then use Boolean logic:

```
continents =np.unique(gapminder["continent"])
continents
```

```
array(['Africa', 'Americas', 'Asia', 'Europe', 'Oceania'], dtype=object)
```

```
AfricaRows = gapminder["continent"]=="Africa"
gapminder[AfricaRows]["gdpPercap"].mean()
```

```
#you could use a for loop instead, of course
gapminder[gapminder["continent"]=="Africa"]["gdpPercap"].mean()
gapminder[gapminder["continent"]=="Americas"]["gdpPercap"].mean()
gapminder[gapminder["continent"]=="Asia"]["gdpPercap"].mean()
gapminder[gapminder["continent"]=="Europe"]["gdpPercap"].mean()
gapminder[gapminder["continent"]=="Oceania"]["gdpPercap"].mean()
```

```
18621.609223333333
```

Instead, we should embrace the concept of **grouping by a variable**

```
gapminder.mean()
```

```
FutureWarning: The default value of numeric_only in DataFrame.mean is deprecated. In a future
  gapminder.mean()
```

```
year         1.979500e+03
lifeExp      5.947444e+01
pop          2.960121e+07
```

```
gdpPercap    7.215327e+03
dtype: float64
```

```
byContinent = gapminder.groupby("continent")
byContinent.mean()
```

```
FutureWarning: The default value of numeric_only in DataFrameGroupBy.mean is deprecated. In a
  byContinent.mean()
```

continent	year	lifeExp	pop	gdpPercap
Africa	1979.5	48.865330	9.916003e+06	2193.754578
Americas	1979.5	64.658737	2.450479e+07	7136.110356
Asia	1979.5	60.064903	7.703872e+07	7902.150428
Europe	1979.5	71.903686	1.716976e+07	14469.475533
Oceania	1979.5	74.326208	8.874672e+06	18621.609223

```
#only lifeExp:
byContinent["lifeExp"].max()
#maybe there is more to life than the mean
```

```
76.442
```

```
byContinent["gdpPercap"].agg([min,max, np.mean])
```

continent	min	max	mean
Africa	241.165876	21951.21176	2193.754578
Americas	1201.637154	42951.65309	7136.110356
Asia	331.000000	113523.13290	7902.150428
Europe	973.533195	49357.19017	14469.475533
Oceania	10039.595640	34435.36744	18621.609223

```
#multiple aggregating functions (no built in function mean)
gapminder.groupby("continent")["gdpPercap"].agg([min,max, np.mean])
```

continent	min	max	mean
Africa	241.165876	21951.21176	2193.754578
Americas	1201.637154	42951.65309	7136.110356
Asia	331.000000	113523.13290	7902.150428
Europe	973.533195	49357.19017	14469.475533
Oceania	10039.595640	34435.36744	18621.609223

```
byContinentYear = gapminder.groupby(["continent", "year"])["gdpPercap"]
byContinentYear.mean()
```

```
#multiple keys
gapminder["past1990"] = gapminder["year"] > 1990
byContinentYear = gapminder.groupby(["continent", "past1990"])["gdpPercap"]
byContinentYear.mean()
```

```
continent  past1990
Africa     False        1997.008411
           True         2587.246913
Americas   False        6051.047533
           True         9306.236000
Asia       False        6713.113041
           True        10280.225202
Europe     False       11341.142807
           True        20726.140986
Oceania    False       15224.015414
           True        25416.796842
Name: gdpPercap, dtype: float64
```

6.3.1 Titanic data

```
# Since pandas does not have any built in data, I am going to "cheat" and
# make use of the `seaborn` library
import seaborn as sns

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

```
titanic["3rdClass"] = titanic["pclass"]==3
titanic["male"] = titanic["sex"]=="male"

titanic
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True
...	...	...	...	...	...	...	...	...	...	...	...
886	0	2	male	27.0	0	0	13.0000	S	Second	man	True
887	1	1	female	19.0	0	0	30.0000	S	First	woman	False
888	0	3	female	NaN	1	2	23.4500	S	Third	woman	False
889	1	1	male	26.0	0	0	30.0000	C	First	man	True
890	0	3	male	32.0	0	0	7.7500	Q	Third	man	True

```
#overall survival rate
titanic.survived.mean()
```

```
0.3838383838383838
```

Tasks:

- compute the proportion of survived separately for
  - male/female
  - the three classes
  - Pclass and sex
- compute the mean age separately for male/female

```
#I would like to compute the mean survical seprately for each group
bySex = titanic.groupby("sex")
#here I am specifically asking for the mean
bySex["survived"].mean()
#if you want multiple summaries, you can list them all inside the agg():
bySex["survived"].agg([min, max, np.mean ])
```

sex	min	max	mean
female	0	1	0.742038
male	0	1	0.188908

```
#I would like to compute the mean survical seprately for each group
bySexPclass = titanic.groupby(["pclass", "sex"])
#here I am specifically asking for the mean
bySexPclass["survived"].mean()
```

```
pclass  sex
1       female    0.968085
        male      0.368852
2       female    0.921053
        male      0.157407
3       female    0.500000
        male      0.135447
Name: survived, dtype: float64
```

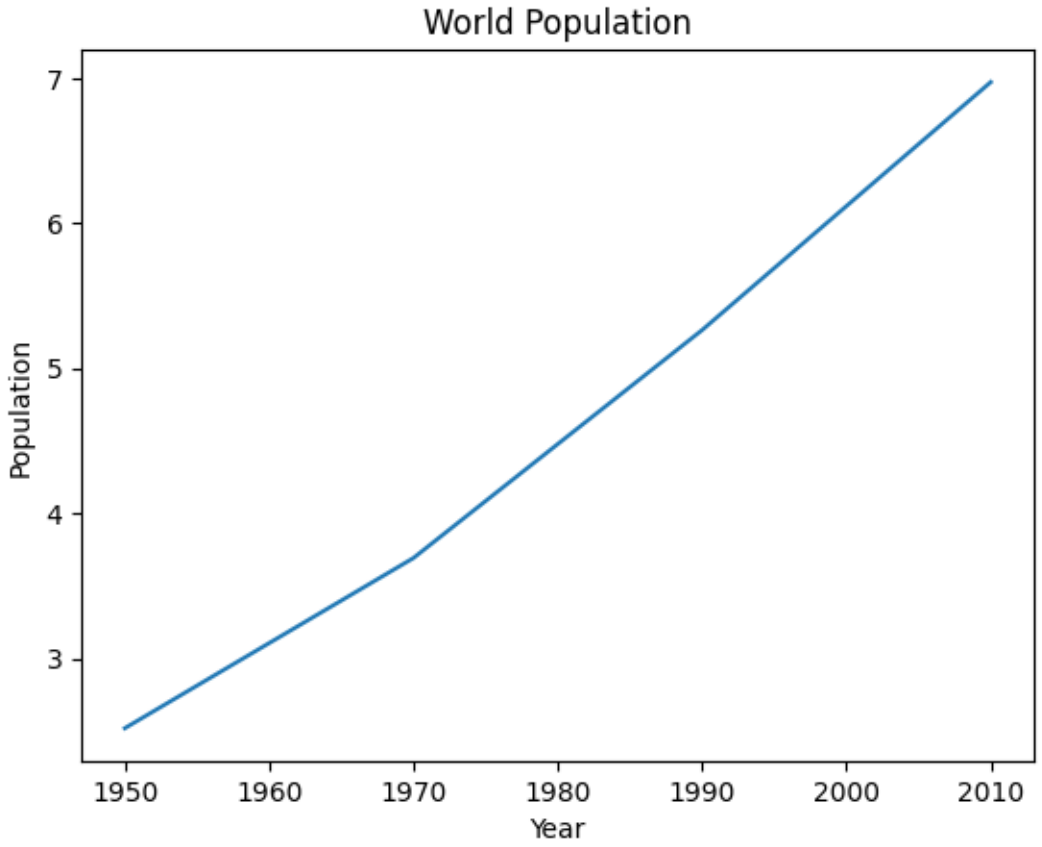
```
bySex = titanic.groupby("sex")
#here I am specifically asking for the mean
bySex["survived"].mean()
```

6.4 Plotting

We will not spend much time with basic plots in matplotlib but instead move quickly toward the pandas versions of these functions.

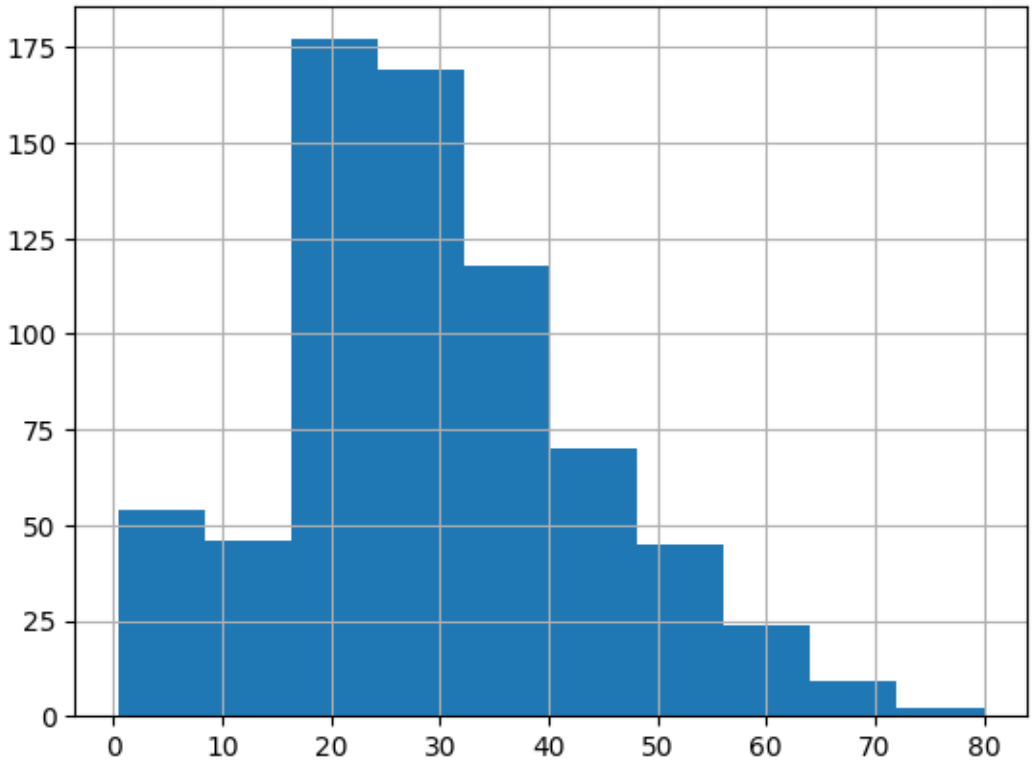
```
#%matplotlib inline
import matplotlib.pyplot as plt

#plt.rcParams['figure.dpi'] = 800
year = [1950, 1970, 1990, 2010]
pop = [2.519, 3.692, 5.263, 6.972]
plt.plot(year, pop)
#plt.bar(year, pop)
#plt.scatter(year, pop)
plt.xlabel('Year')
```

```
plt.ylabel('Population')
plt.title('World Population')
x = 1
#plt.show()
```



pandas offers plots directly from its objects

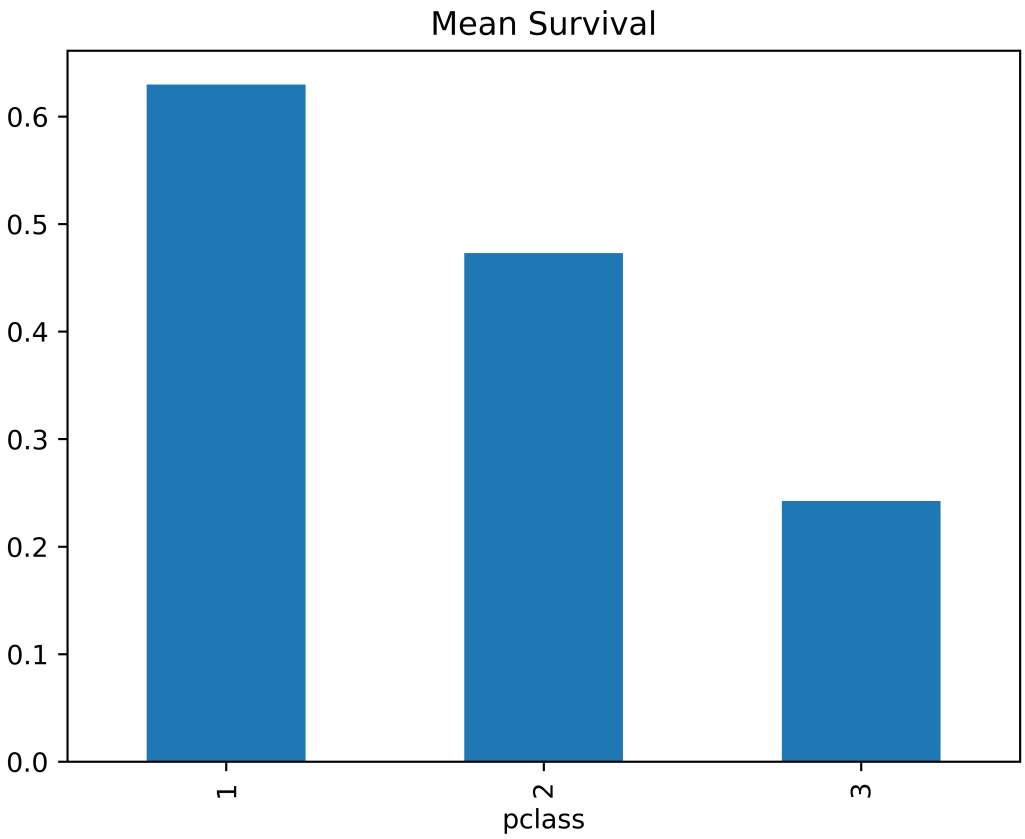
```
titanic.age.hist()
plt.show()
```



And often the axis labels are taken care of

```
#titanic.groupby("pclass").survived.mean().plot.bar()
SurvByPclass = titanic.groupby("pclass").survived.mean()

SurvByPclass.plot(kind="bar", title = "Mean Survival");
```

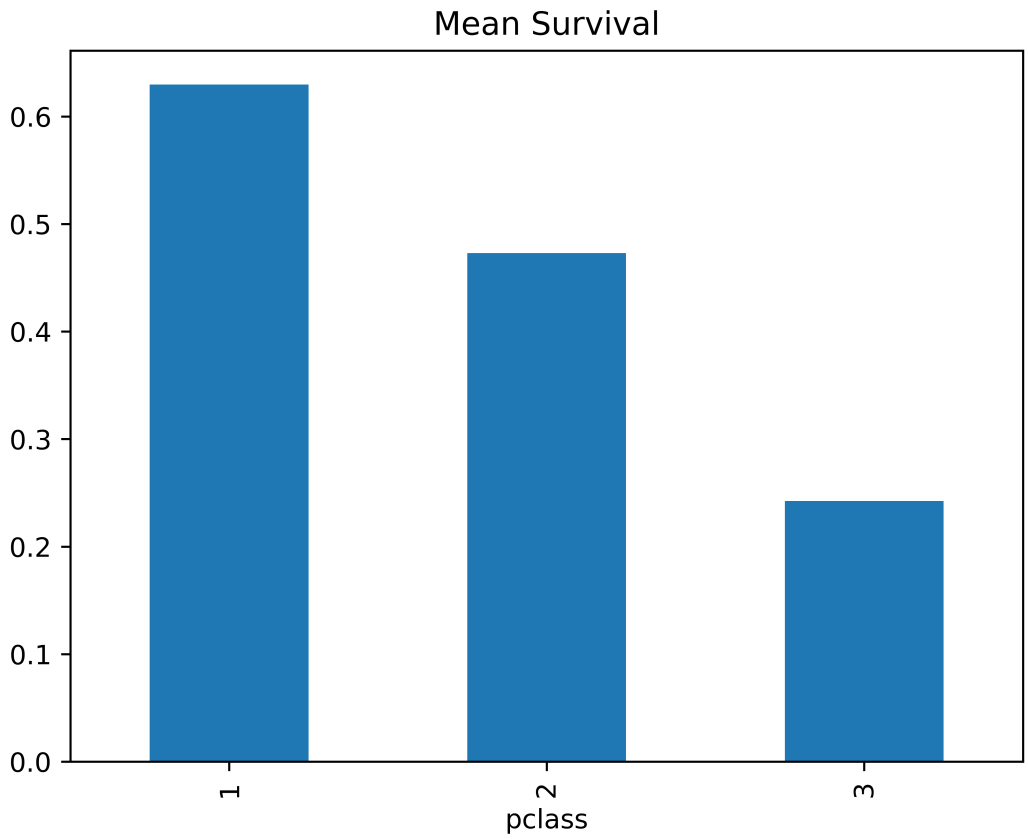
```
<Axes: title={'center': 'Mean Survival'}, xlabel='pclass'>
```



But you can customize each plot as you wish:

```
SurvByPclass.plot(kind="bar", x = "Passenger Class", y = "Survived", title = "Mean Surviva
```

```
<Axes: title={'center': 'Mean Survival'}, xlabel='pclass'>
```

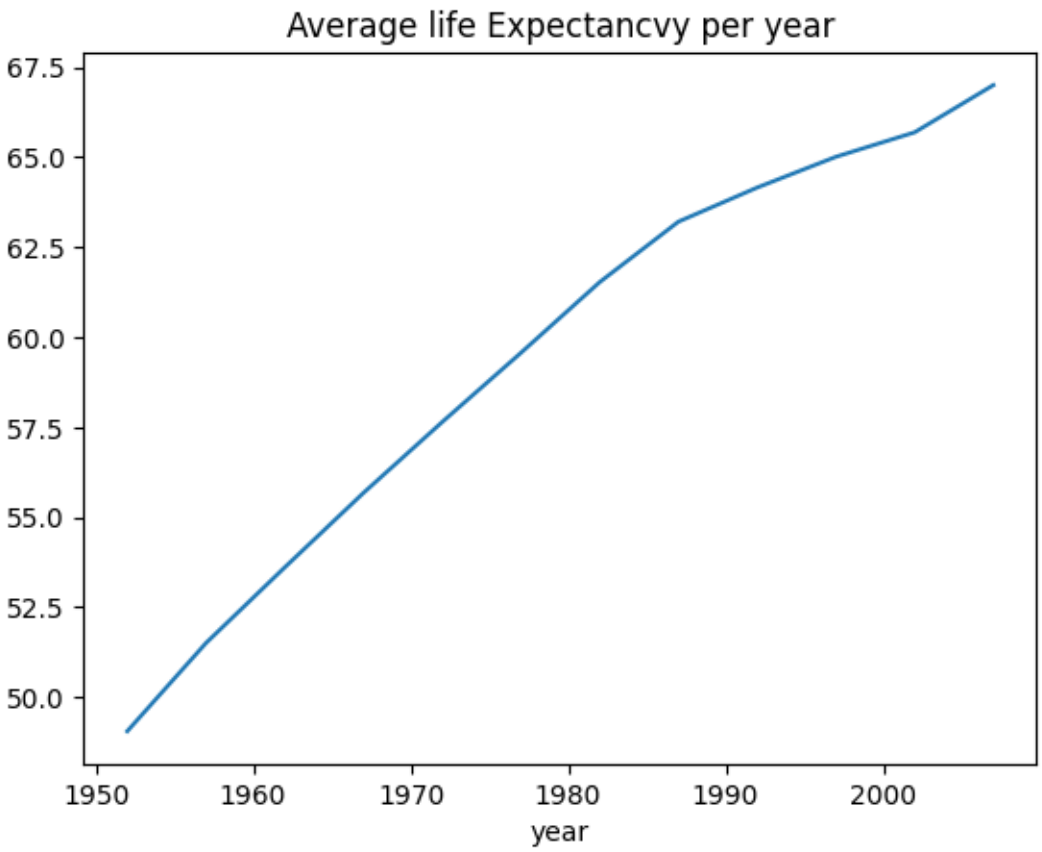


Tasks:

- Compute the avg. life expectancy in the gapminder data for each year
- Plot this as a line plot and give meaningful x and y labels and a title

```
lifeExpbyYear = gapminder.groupby("year")["lifeExp"].mean()

lifeExpbyYear.plot(y= "avg. life Exp", title = "Average life Expectancvy per year");
```

```
<Axes: title={'center': 'Average life Expectancvy per year'}, xlabel='year'>
```



6.5 Advanced topics

6.5.1 Creating Dataframes

1. Zip
2. From list of dicts

6.5.2 Indexing:

1. multilevel indexes
2. sorting
3. asking for ranges

6.6 Types of columns

1. categorical
2. dates

```
# Creating Dataframes
#using zip
# List1
Name = ['tom', 'krish', 'nick', 'juli']

# List2
Age = [25, 30, 26, 22]

# get the list of tuples from two lists.
# and merge them by using zip().
list_of_tuples = list(zip(Name, Age))
list_of_tuples = zip(Name, Age)
# Assign data to tuples.
#print(list_of_tuples)


# Converting lists of tuples into
# pandas Dataframe.
df = pd.DataFrame(list_of_tuples,
                  columns=['Name', 'Age'])

# Print data.
df
```

	Name	Age
0	tom	25
1	krish	30
2	nick	26
3	juli	22

```
#from list of dicts
data = [{'a': 1, 'b': 2, 'c': 3},
        {'a': 10, 'b': 20, 'c': 30}]

# Creates DataFrame.
```

```
df = pd.DataFrame(data)

df
```

	a	b	c
0	1	2	3
1	10	20	30

```
# Indexing:

advLesson = True
if advLesson:
    frame2 = frame.set_index(["year", "state"])
    print(frame2)
    frame3 = frame2.sort_index()
    print(frame3)
    print(frame.loc[:,"state":"year"])
```

```
            pop
year state
2000 Ohio    1.5
2001 Ohio    1.7
2002 Ohio    3.6
2001 Nevada  2.4
2002 Nevada  2.9
2003 Nevada  3.2
            pop
year state
2000 Ohio    1.5
2001 Nevada  2.4
     Ohio    1.7
2002 Nevada  2.9
     Ohio    3.6
2003 Nevada  3.2
    state  year
0    Ohio  2000
1    Ohio  2001
2    Ohio  2002
3  Nevada  2001
4  Nevada  2002
```

```
5  Nevada  2003
```

6.6.1 Inplace

Note that I reassigned the objects in the code above. That is because most operations, such as `set_index`, `sort_index`, `drop`, etc. do not operate **inplace** unless specified!

7 Missing Values/Duplicates

In this lecture we will continue our journey of Data Manipulation with pandas after reviewing some fundamental aspects of the syntax

1. Review
    - Review of "brackets"
2. Pandas
    - Dealing with Missing Values
    - Dealing with Duplicates
3. Plot of the Day
    - boxplot

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

7.0.1 Review of "brackets"

In Python, there are several types of brackets used for different purposes. Here's a brief review of the most commonly used brackets:

1. Parentheses `( )`: Parentheses are used for grouping expressions, defining function parameters, and invoking functions. They are also used in mathematical expressions to indicate order of operations.
2. Square brackets `[ ]`: Square brackets are primarily used for indexing and slicing operations on lists, tuples, and strings. They allow you to access individual elements or extract subsequences from these data types.
3. Curly brackets or braces `{ }`: Curly brackets are used to define dictionaries, which are key-value pairs. Dictionaries store data in an unordered manner, and you can access or manipulate values by referencing their corresponding keys within the curly brackets.

It's important to note that the usage of these brackets may vary depending on the specific context or programming paradigm you're working with. Nonetheless, understanding their general purpose will help you navigate Python code effectively.

7.0.1.0.1 Examples

Here are a few examples of how each type of bracket is used in Python:

1. Parentheses `( )`:
   - Grouping expressions:

```
result = (2 + 3) * 4
# Output: 20
```

   - Defining function parameters:

```
def greet(name):
    print("Hello, " + name + "!")

greet("Alice")
# Output: Hello, Alice!
```

   - Invoking functions:

```
result = max(5, 10)
# Output: 10
```

2. Square brackets `[ ]`:
   - Indexing and slicing:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[0])
# Output: 1
print(my_list[1:3])
# Output: [2, 3]
```

   - Modifying list elements:

```
my_list = [1, 2, 3]
my_list[1] = 10
print(my_list)
```

```
# Output: [1, 10, 3]
```

3. Curly brackets or braces `{ }`:
   - Defining dictionaries:

```
my_dict = {"name": "Alice", "age": 25, "city": "London"}
print(my_dict["name"])
# Output: Alice
```

   - Modifying dictionary values:

```
my_dict = {"name": "Alice", "age": 25}
my_dict["age"] = 30
print(my_dict)
# Output: {'name': 'Alice', 'age': 30}
```

Remember that the usage of brackets can vary depending on the specific programming context, but these examples provide a general understanding of their usage in Python.

7.0.2 Tasks

7.0.3 Data Manipulation with pandas

While we have seen panda's ability to (i) mix data types (strings, numbers, categories, Boolean, …) and (ii) refer to columns and rows by names, this library offers a lot more powerful tools for efficiently gaining insights from data, e.g.

- deal with missing values

7.0.4 Missing Values

Missing data occurs commonly in many data analysis applications. Most often they are a consequence of

- data entry errors, or
- unknown numbers, or
- `groupby` operations, or
- wrong mathematical operations (`1/0`, $\sqrt{-1}$,$\log(0)$, …)
- "not applicable" questions, or
- ….

For data with float type, pandas uses the floating-point value `NaN` (Not a Number) to represent missing data.

Pandas refers to missing data as `NA`, which stands for *not available*.

The built-in Python `None` value is also treated as NA:

Recall constructing a DataFrame from a dictionary of equal-length lists or NumPy arrays (Lecture 4). What if there were data entry errors or just unknown numbers

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, None, 2001, 2002, 2003],
        "gdp": [1.5, 1.7, 3.6, 2.4, np.nan, 3.2]}
frame = pd.DataFrame(data)# creates a dataframe out of the data given!
df = frame
frame
```

	state	year	gdp
0	Ohio	2000.0	1.5
1	Ohio	2001.0	1.7
2	Ohio	NaN	3.6
3	Nevada	2001.0	2.4
4	Nevada	2002.0	NaN
5	Nevada	2003.0	3.2

```
x = np.array([1,2,3,4])
x==2
#remove those values of x that are equal to 2
#x[[0,2,3]]
x[x!=2]
```

```
array([1, 3, 4])
```

```
frame.columns[frame.isna().any()]
```

```
Index(['year', 'gdp'], dtype='object')
```

(Note the annoying conversion of integers to float, a solution discussed here)

7.0.4.1 Filtering Out Missing Data

There are a few ways to filter out missing data. While you always have the option to do it by hand using `pandas.isna` and Boolean indexing, `dropna` can be helpful.

With DataFrame objects, there are different ways to remove missing data. You may want to drop rows or columns that are all `NA`, or only those rows or columns containing any `NA`s at all. `dropna` by default drops any row containing a missing value:

```
frame.dropna()
```

	state	year	gdp
0	Ohio	2000.0	1.5
1	Ohio	2001.0	1.7
3	Nevada	2001.0	2.4
5	Nevada	2003.0	3.2

Passing `how="all"` will drop only rows that are all NA:

```
frame.dropna(how="all")
```

	state	year	gdp
0	Ohio	2000.0	1.5
1	Ohio	2001.0	1.7
2	Ohio	NaN	3.6
3	Nevada	2001.0	2.4
4	Nevada	2002.0	NaN
5	Nevada	2003.0	3.2

To drop columns in the same way, pass `axis="columns"`:

```
frame.dropna(axis="columns")
```

	state
0	Ohio
1	Ohio
2	Ohio
3	Nevada

	state
4	Nevada
5	Nevada

Suppose you want to keep only rows containing at most a certain number of missing observations. You can indicate this with the `thresh` argument:

```
frame.iloc[2,2] = np.nan
frame
```

	state	year	gdp
0	Ohio	2000.0	1.5
1	Ohio	2001.0	1.7
2	Ohio	NaN	NaN
3	Nevada	2001.0	2.4
4	Nevada	2002.0	NaN
5	Nevada	2003.0	3.2

```
frame.dropna(thresh=2)
```

	state	year	gdp
0	Ohio	2000.0	1.5
1	Ohio	2001.0	1.7
3	Nevada	2001.0	2.4
4	Nevada	2002.0	NaN
5	Nevada	2003.0	3.2

7.0.4.2 Filling In Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the "holes" in any number of ways. For most purposes, the fillna method is the workhorse function to use. Calling `fillna` with a constant replaces missing values with that value:

```
frame2 = frame
frame2.iloc[0,0] = np.nan#None
frame2.iloc[0,2] = np.nan
```

```
frame2
#frame2.fillna(0)
#type(frame2["state"])
```

	state	year	gdp
0	NaN	2000.0	NaN
1	Ohio	2001.0	1.70
2	Ohio	NaN	3.60
3	Nevada	2001.0	2.40
4	Nevada	2002.0	2.48
5	Nevada	2003.0	3.20

Calling `fillna` with a dictionary, you can use a different fill value for each column:

```
frame2.fillna({"state" : "Neverland", "year": -999, "gdp": 0})
```

	state	year	gdp
0	Neverland	2000.0	0.00
1	Ohio	2001.0	1.70
2	Ohio	-999.0	3.60
3	Nevada	2001.0	2.40
4	Nevada	2002.0	2.48
5	Nevada	2003.0	3.20

With `fillna` you can do lots of other things such as simple data imputation using the median or mean statistics, at least for purely numeric data types:

```
frame['gdp'] = frame['gdp'].fillna(frame['gdp'].mean())
frame
```

	state	year	gdp
0	NaN	2000.0	1.50
1	Ohio	2001.0	1.70
2	Ohio	NaN	3.60
3	Nevada	2001.0	2.40
4	Nevada	2002.0	2.48
5	Nevada	2003.0	3.20

Different values for each column

```
df.fillna(df.mean())
```

```
FutureWarning: The default value of numeric_only in DataFrame.mean is deprecated. In a future
  df.fillna(df.mean())
```

	state	year	gdp
0	Ohio	2000.0	1.50
1	Ohio	2001.0	1.70
2	Ohio	2001.4	3.60
3	Nevada	2001.0	2.40
4	Nevada	2002.0	2.48
5	Nevada	2003.0	3.20

```
fillna(frame['gdp'].mean())
```

7.0.5 Duplicate Values

Duplicate rows may be found in a DataFrame for any number of reasons.

Removing duplicates in pandas can be useful in various scenarios, particularly when working with large datasets or performing data analysis. Here's a convincing example to illustrate its usefulness:

Let's say you have a dataset containing sales transactions from an online store. Each transaction record consists of multiple columns, including customer ID, product ID, purchase date, and purchase amount. Due to various reasons such as system glitches or human errors, duplicate entries might exist in the dataset, meaning that multiple identical transaction records are present.

In such a scenario, removing duplicates becomes beneficial for several reasons:

1. Accurate Analysis: Duplicate entries can skew your analysis and lead to incorrect conclusions. By removing duplicates, you ensure that each transaction is represented only once, providing more accurate insights and preventing inflated or biased results.

2. Data Integrity: Duplicate entries consume unnecessary storage space and can make data management more challenging. By eliminating duplicates, you maintain data integrity and ensure a clean and organized dataset.
3. Efficiency: When dealing with large datasets, duplicate records can significantly impact computational efficiency. Removing duplicates allows you to streamline your data processing operations, leading to faster analysis and improved performance.
4. Unique Identifiers: Removing duplicates becomes crucial when working with columns that should contain unique values, such as customer IDs or product IDs. By eliminating duplicates, you ensure the integrity of these unique identifiers and prevent issues when performing joins or merging dataframes.

To remove duplicates in pandas, you can use the `drop_duplicates()` function. It identifies and removes duplicate rows based on specified columns or all columns in the dataframe, depending on your requirements.

Overall, removing duplicates in pandas is essential for maintaining data accuracy, integrity, and efficiency, allowing you to derive meaningful insights and make informed decisions based on reliable data.

```
#super simple example:
frame.iloc[5] = frame.iloc[4]#this line makes lines 5 and 6 equal
frame
```

	state	year	gdp
0	NaN	2000.0	NaN
1	Ohio	2001.0	1.70
2	Ohio	NaN	3.60
3	Nevada	2001.0	2.40
4	Nevada	2002.0	2.48
5	Nevada	2002.0	2.48

The DataFrame method `duplicated` returns a Boolean Series indicating whether each row is a duplicate

```
frame.duplicated()
```

```
0    False
1    False
2    False
3    False
```

```
4    False
5     True
dtype: bool
```

Relatedly, `drop_duplicates` returns a DataFrame with rows where the `duplicated` array is `False` filtered out:

```
frame.drop_duplicates()
```

	state	year	gdp
0	NaN	2000.0	NaN
1	Ohio	2001.0	1.70
2	Ohio	NaN	3.60
3	Nevada	2001.0	2.40
4	Nevada	2002.0	2.48

Both methods by default consider all of the columns; alternatively, you can specify any `subset` of them to detect duplicates.

```
frame.drop_duplicates(subset=["year"])
```

	state	year	gdp
0	NaN	2000.0	NaN
1	Ohio	2001.0	1.70
2	Ohio	NaN	3.60
4	Nevada	2002.0	2.48

7.0.5.0.1 Titanic data

```
# Since pandas does not have any built in data, I am going to "cheat" and
# make use of the `seaborn` library
import seaborn as sns

titanic = sns. load_dataset('titanic')
titanic["3rdClass"] = titanic["pclass"]==3
titanic["male"] = titanic["sex"]=="male"
```

```
titanic
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True
...	...	...	...	...	...	...	...	...	...	...	...
886	0	2	male	27.0	0	0	13.0000	S	Second	man	True
887	1	1	female	19.0	0	0	30.0000	S	First	woman	False
888	0	3	female	NaN	1	2	23.4500	S	Third	woman	False
889	1	1	male	26.0	0	0	30.0000	C	First	man	True
890	0	3	male	32.0	0	0	7.7500	Q	Third	man	True

```
#how many missing values in age ?
np.sum(titanic["age"].isna())
```

```
177
```

```
titanic.describe()
```

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

```
np.mean(titanic["age"])

np.sum(titanic["age"])/714
```

```
titanic['age'] = titanic['age'].fillna(titanic['age'].mean())
```

```
29.69911764705882
```

Notice that the age columns contains missing values, which is a big topic by itself in data science.

How should an aggregating react to and handle missing values? The default often is to ignore and exclude them from the computation, e.g.

```
#the following shoudl be equal but is not due to missing values
meanAge =np.mean(titanic.age)
print(meanAge)
print(np.sum(titanic.age)/len(titanic.age))
```

```
29.69911764705882
23.79929292929293
```

In general, it is a good idea to diagnose how many missing values there are in each column. We can use some handy built-in support for this task:

```
titanic.isna().sum()
```

```
survived         0
pclass           0
sex              0
age            177
sibsp            0
parch            0
fare             0
embarked         2
class            0
who              0
adult_male       0
deck           688
embark_town      2
alive            0
alone            0
3rdClass         0
male             0
dtype: int64
```

7.0.6 Tasks

Dropping or replacing NAs:

7.1 Plotting

The "plot type of the day" is one of the most popular ones used to display data distributions, the **boxplot**.

Boxplots, also known as **box-and-whisker plots**, are a statistical visualization tool that provides a concise summary of a dataset's distribution. They display key descriptive statistics and provide insights into the central tendency, variability, and skewness of the data. Here's a brief introduction and motivation for using boxplots:

1. Structure of Boxplots: Boxplots consist of a box and whiskers that represent different statistical measures of the data:
    - The box represents the interquartile range (IQR), which spans from the lower quartile (25th percentile) to the upper quartile (75th percentile). The width of the box indicates the spread of the middle 50% of the data.
    - A line (whisker) extends from each end of the box to show the minimum and maximum values within a certain range (often defined as 1.5 times the IQR).
    - Points beyond the whiskers are considered outliers and plotted individually.
2. Motivation for Using Boxplots: Boxplots offer several benefits and are commonly used for the following reasons:
    - Visualizing Data Distribution: Boxplots provide a concise overview of the distribution of a dataset. They show the skewness, symmetry, and presence of outliers, allowing for quick identification of key features.
    - Comparing Groups: Boxplots enable easy visual comparison of multiple groups or categories. By placing side-by-side boxplots, you can assess differences in central tendency and variability between groups.
    - Outlier Detection: Boxplots explicitly mark outliers, aiding in the identification of extreme values or data points that deviate significantly from the overall pattern.
    - Data Summary: Boxplots summarize key statistics, including the median, quartiles, and range, providing a quick understanding of the dataset without the need for detailed calculations.
    - Robustness: Boxplots are relatively robust to skewed or asymmetric data and can effectively handle datasets with outliers.

Boxplots are widely used in various fields, including data analysis, exploratory data visualization, and statistical reporting. They offer a clear and concise representation of data distribution, making them a valuable tool for understanding and communicating the characteristics of a dataset.

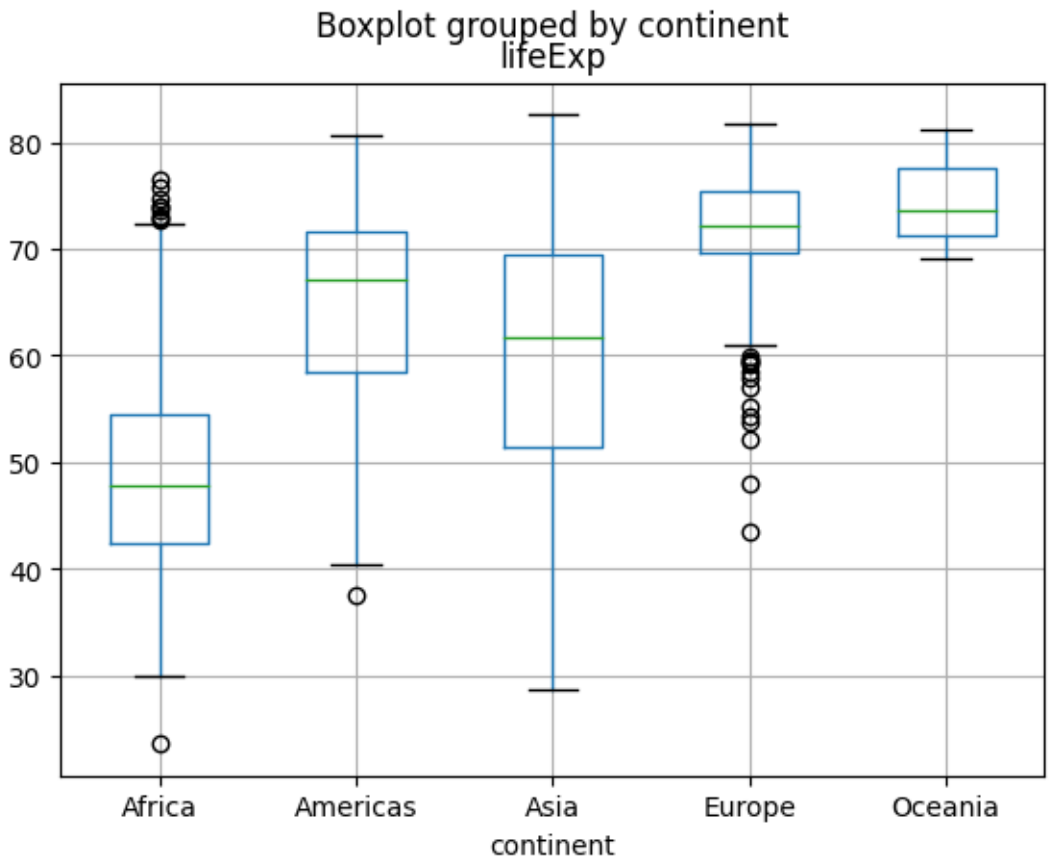
```
!pip install gapminder
from gapminder import gapminder
```

```
gapminder
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106
...	...	...	...	...	...	...
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298

The pandas way

```
gapminder.boxplot(column = "lifeExp", by="continent");
```



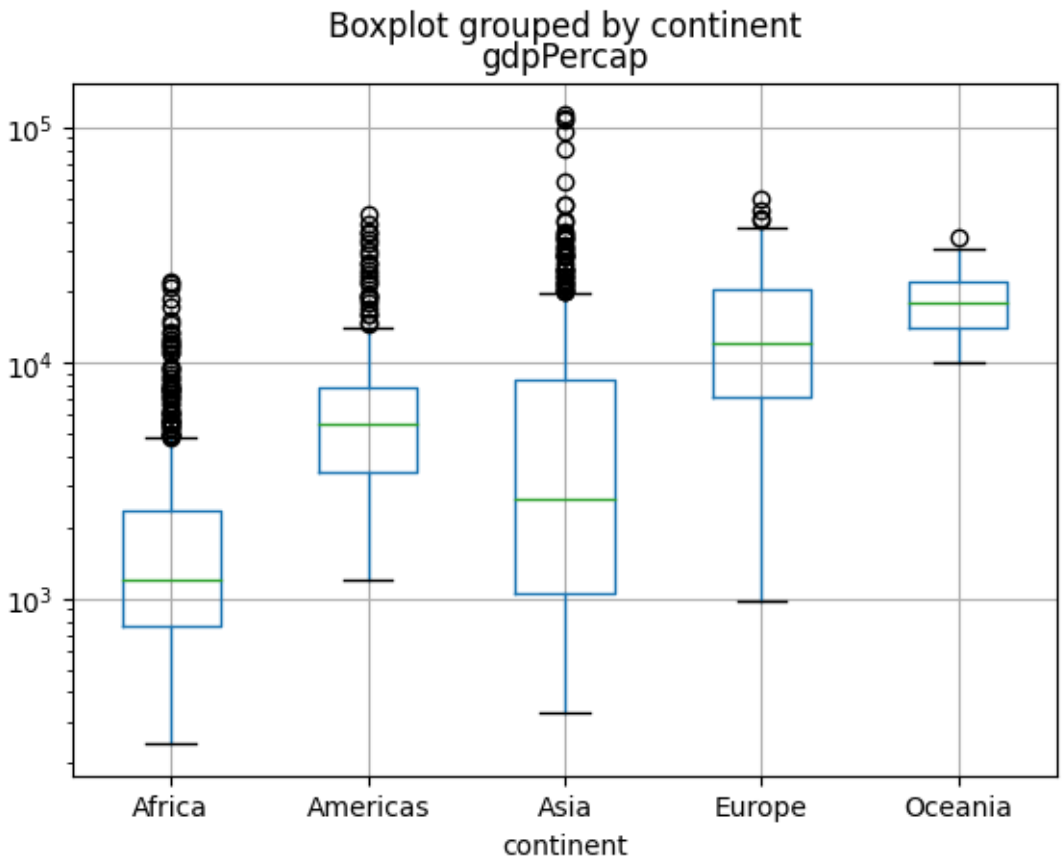
The matplotlib way

```
plt.boxplot(gapminder["continent"], gapminder["lifeExp"]);
```

7.1.1 Task

- Create a boxplot for `gdpPercap` instead. What do you notice ? Are you happy with how the plot looks? Any "trick" you can think to make this more readable?
- Advanced: can you create boxplots for `gdpPerCap` and `lifeExp` in one command?

```
gapminder.boxplot(column = "gdpPercap", by="continent");
plt.yscale("log")
```



Further Reading:

- Python Plotting With Matplotlib Tutorial.)

```
import numpy as np
from scipy.stats import entropy

p = np.array([1/100, 99/100])
n=2
#p = np.array(np.ones)/n
H = entropy(p, base=2)
H
```

```
0.08079313589591118
```

8 Intro to Models

In this lecture we will learn about **modeling** data for the first time. After this lesson, you should know what we generally mean by a "model", what linear regression is and how to interpret the output. But first we need to introduce a new data type: *categorical variables*.

1. Categorical variables
2. Models
    - Tables as models
    - Modeling Missing Values
    - Linear Regression

Online Resources:

Chapter 7.5 of our textbook introduces categorical variables.

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

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

```
#!pip install gapminder
from gapminder import gapminder
```

```
gapminder.head()
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138

	country	continent	year	lifeExp	pop	gdpPercap
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

8.1 Categorical variables

As a motivation, take another look at the gapminder data which contains variables of a **mixed type**: numeric columns along with string type columns which contain repeated instances of a smaller set of distinct or **discrete** values which

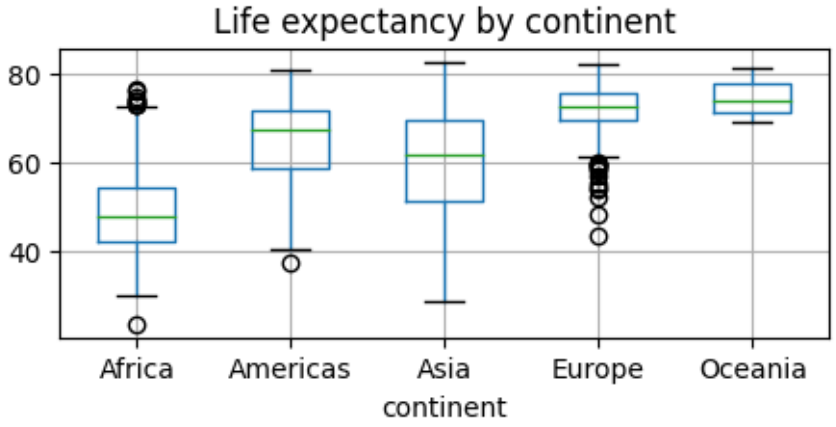
1. are not numeric (but could be represented as numbers)
2. cannot really be ordered
3. typically take on a finite set of values, or *categories*.

We refer to these data types as **categorical**.

We have already seen functions like `unique` and `value_counts`, which enable us to extract the distinct values from an array and compute their frequencies.

Boxplots and grouping operations typically use a categorical variable to compute summaries of a numerical variables for each category separately, e.g.

```
gapminder.boxplot(column = "lifeExp", by="continent",figsize=(5, 2));
plt.title('Life expectancy by continent')
# Remove the default suptitle
plt.suptitle("");
```



pandas has a special `Categorical` extension type for holding data that uses the integer-based categorical representation or encoding. This is a popular data compression technique for data with many occurrences of similar values and can provide significantly faster performance with lower memory use, especially for string data.

```
gapminder.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   country    1704 non-null   object
 1   continent  1704 non-null   object
 2   year       1704 non-null   int64
 3   lifeExp    1704 non-null   float64
 4   pop        1704 non-null   int64
 5   gdpPercap  1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
```

```
gapminder['country'] = gapminder['country'].astype('category')
gapminder.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   country    1704 non-null   category
 1   continent  1704 non-null   object
 2   year       1704 non-null   int64
 3   lifeExp    1704 non-null   float64
 4   pop        1704 non-null   int64
 5   gdpPercap  1704 non-null   float64
dtypes: category(1), float64(2), int64(2), object(1)
memory usage: 75.2+ KB
```

We will come back to the usefulness of this later.

8.2 Tables as models

For now let us look at our first "model":

```
titanic = sns.load_dataset('titanic')
titanic["class3"] = (titanic["pclass"]==3)
titanic["male"] = (titanic["sex"]=="male")
titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	dec
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	Na
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	Na
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	Na

```
vals1, cts1 = np.unique(titanic["class3"], return_counts=True)
print(cts1)
print(vals1)
```

```
[400 491]
[False  True]
```

```
print("The mean survival on the Titanic was", np.mean(titanic.survived))
```

```
The mean survival on the Titanic was 0.3838383838383838
```

```
ConTbl = pd.crosstab(titanic["sex"], titanic["survived"])
ConTbl
```

survived	0	1
sex		
female	81	233
male	468	109

What are the estimated survival probabilities?

```
#the good old groupby way:
bySex = titanic.groupby("sex").survived
bySex.mean()
```

```
sex
female    0.742038
male      0.188908
Name: survived, dtype: float64
```

```
p3D = pd.crosstab([titanic["sex"], titanic["class3"]], titanic["survived"])
p3D
```

sex	survived class3	0	1
female	False	9	161
	True	72	72
male	False	168	62
	True	300	47

What are the estimated survival probabilities?

```
#the good old groupby way:
bySex = titanic.groupby(["sex", "class3"]).survived
bySex.mean()
```

```
sex     class3
female  False     0.947059
        True      0.500000
male    False     0.269565
        True      0.135447
Name: survived, dtype: float64
```

The above table can be looked at as a **model**, which is defined as a function which takes *inputs* $\mathbf{x}$ and "spits out" a *prediction*:

$y = f(\mathbf{x})$

In our case, the inputs are $x_1 =$ sex, $x_2 =$ class3, and the output is the estimated survival probability!

It is evident that we could keep adding more *input* variables and make finer and finer grained predictions.

8.2.1 Linear Models

```
lsFit = smf.ols('survived ~ sex:class3-1', titanic).fit()
lsFit.summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
sex[female]:class3[False]	0.9471	0.029	32.200	0.000	0.889	1.005
sex[male]:class3[False]	0.2696	0.025	10.660	0.000	0.220	0.319
sex[female]:class3[True]	0.5000	0.032	15.646	0.000	0.437	0.563
sex[male]:class3[True]	0.1354	0.021	6.579	0.000	0.095	0.176

8.2.2 Modeling Missing Values

We have already seen how to detect and how to replace missing values. But the latter -until now- was rather crude: we often replaced all values with a "global" average.

Clearly, we can do better than replacing all missing entries in the *survived* column with the average 0.38.

```
rng = default_rng()

missingRows = rng.integers(0,890,20)
print(missingRows)
#introduce missing values
titanic.iloc[missingRows,0] = np.nan
np.sum(titanic.survived.isna())
```

```
[864 299 857 182 808 817 802 295 255 644   1 685 452 463 303 551 517 502
 495 412]
```

```
20
```

```
predSurv = lsFit.predict()
print( len(predSurv))
predSurv[titanic.survived.isna()]
```

```
891
```

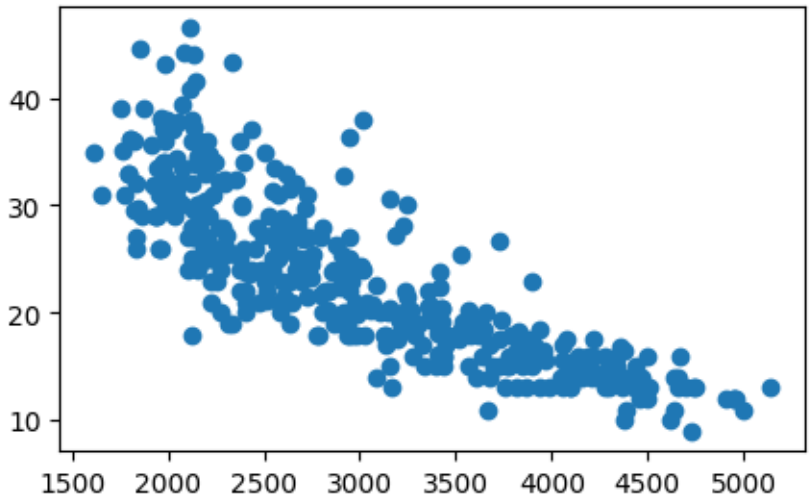
```
array([0.94705882, 0.13544669, 0.5       , 0.26956522, 0.94705882,
       0.94705882, 0.94705882, 0.26956522, 0.26956522, 0.13544669,
       0.5       , 0.13544669, 0.26956522, 0.5       , 0.26956522,
       0.26956522, 0.26956522, 0.26956522, 0.26956522, 0.26956522])
```

8.2.2.1 From categorical to numerical relations

```
url = "https://drive.google.com/file/d/1UbZy5Ecknpl1GXZBkbhJ_K6GJcIA2Plq/view?usp=share_li
url='https://drive.google.com/uc?id=' + url.split('/')[-2]
auto = pd.read_csv(url)
auto.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle mal
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino

```
plt.figure(figsize=(5,3))
plt.scatter(x=auto["weight"], y=auto["mpg"]);
```

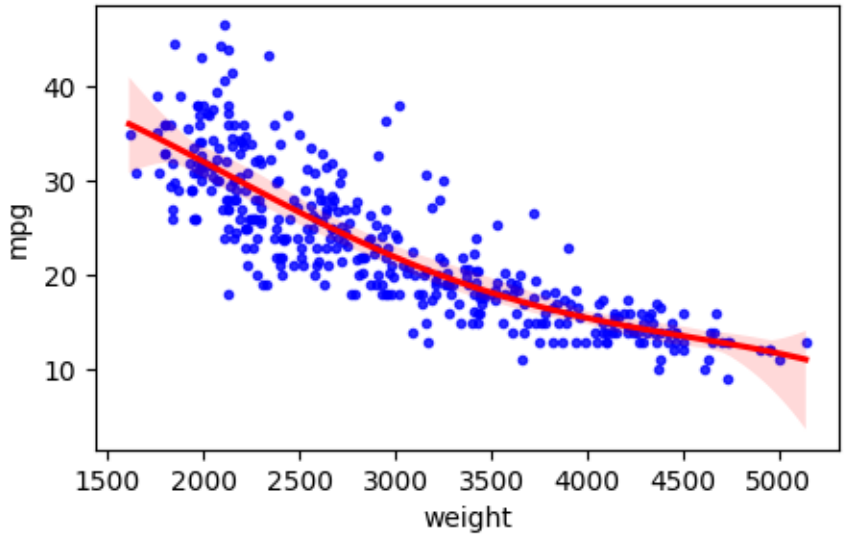


8.3 Linear Regression

We can roughly estimate, i.e. "model" this relationship with a straight line:

$$y = \beta_0 + \beta_1 x$$

```
plt.figure(figsize=(5,3))
tmp=sns.regplot(x=auto["weight"], y=auto["mpg"], order=1, ci=95,
                scatter_kws={'color':'b', 's':9}, line_kws={'color':'r'})
```



Remind yourself of the definition of the slope of a straight line

$$\beta_1 = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

```
est = smf.ols('mpg ~ weight', auto).fit()
est.summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	46.2165	0.799	57.867	0.000	44.646	47.787
weight	-0.0076	0.000	-29.645	0.000	-0.008	-0.007

```
np.corrcoef(auto["weight"], auto["mpg"])
```

```
array([[ 1.        , -0.83224421],
       [-0.83224421,  1.        ]])
```

Further Reading:

-

9 Sampling Distributions

Overview

1. Taking Samples
   - Variation of samples
   - The $1/\sqrt{n}$ law
2. Distributions
   - densities
   - ecdfs
3. Parametric ("analytic") versus nonparametric ("hacker statistics")
   - Confidence Intervals
   - Testing
4. Resampling
   - Permutations
   - Bootstrap

Importing libraries

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from numpy import random
random.seed(42) #What is this for ???
```

Importing the Birthweights Dataframe

```
df = pd.read_csv("https://raw.githubusercontent.com/markusloecher/DataScience-HWR/main/dat
#df = pd.read_csv('../data/BirthWeights.csv')[["sex", "dbirwt"]]
df = df[["sex", "dbirwt"]]
df.head()
```

	sex	dbirwt
0	male	2551
1	male	2778
2	female	2976
3	female	3345
4	female	3175

```
df.groupby("sex").mean()
```

	dbirwt
sex	
female	3419.186742
male	3507.089865

9.1 Operator Overloading

The [] operator is overloaded. This means, that depending on the inputs, pandas will do something completely different. Here are the rules for the different objects you pass to just the indexing operator.

- string — return a column as a Series
- list of strings — return all those columns as a DataFrame
- a slice — select rows (can do both label and integer location — confusing!)
- a sequence of booleans — select all rows where True

In summary, primarily just the indexing operator selects columns, but if you pass it a sequence of booleans it will select all rows that are `True`.

```
df = df[(df[ "dbirwt"] < 6000) & (df[ "dbirwt"] > 2000)] # 2000 < birthweight < 6000
df#.head()
```

	sex	dbirwt
0	male	2551
1	male	2778
2	female	2976

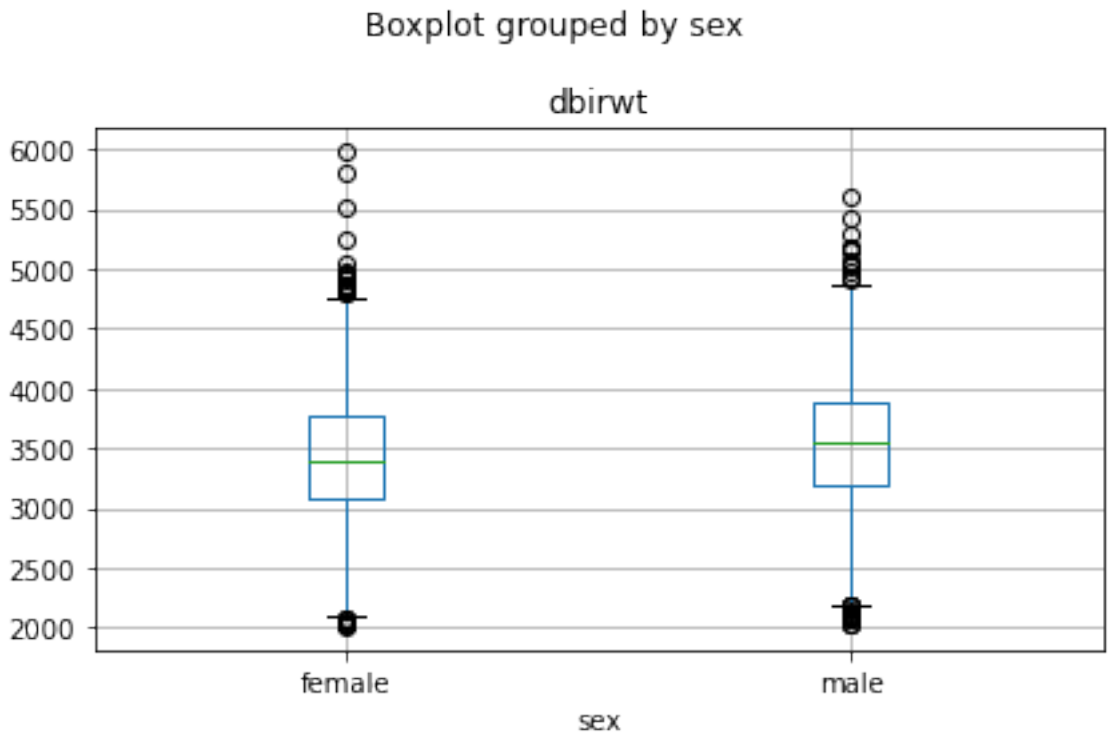
	sex	dbirwt
3	female	3345
4	female	3175
...	...	...
4995	male	4405
4996	male	2764
4997	female	2776
4998	female	3615
4999	male	3379

Boxplot of weight vs. sex

```
df.describe()
```

	dbirwt
count	4909.000000
mean	3480.557344
std	529.280103
min	2012.000000
25%	3146.000000
50%	3486.000000
75%	3827.000000
max	5981.000000

```
tmp=df.boxplot( "dbirwt","sex")
plt.tight_layout()
```

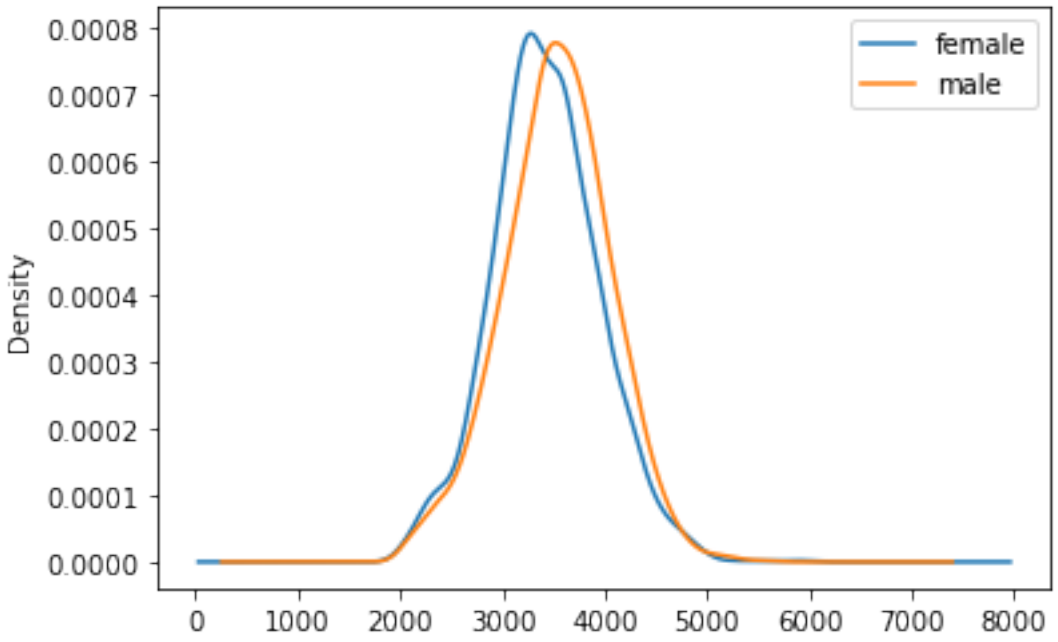


We notice a small difference in the average weight, which is more clearly visible when we plot overlaying densities for male/female

```
bwghtBySex = np.round(df[["dbirwt","sex"]].groupby("sex")[["dbirwt"]].mean())

print(bwghtBySex, '\n')
print('mean: ',bwghtBySex.mean())
```

```
        dbirwt
sex
female  3427.0
male    3533.0

mean:  dbirwt    3480.0
dtype: float64
```

```
tmp=df[["dbirwt","sex"]].groupby("sex")["dbirwt"].plot(kind='density', legend=True)
```



9.2 A/B Testing

Let us hypothesize that one wanted to classify babies into male/female solely based on their weight. What would its accuracy be if we applied the following simple rule:

if dbirwt $>$ 3480 y = male else y = female

This would be the equivalent of testing for global warming by measuring the temperature on **one** day. We all know that it took a long time (= many samples) to reliably detect a small difference like 0.5 degrees buried in the noise. Let us apply the same idea here. Maybe we can build a high-accuracy classifier if we weighed enough babies separately for each sex.

Confusion Matrix for simple classifier

```
df["predMale"] = (df["dbirwt"] > 3480)
ConfMat = pd.crosstab(df["predMale"], df["sex"])
ConfMat
```

sex predMale	female	male
False	1331	1105
True	1100	1373

```
N = np.sum(ConfMat.values)
acc1 = np.round( (ConfMat.values[0,0]+ConfMat.values[1,1]) / N, 3)

#Acc0 = (1331+1373)/5000
print("Accuracy of lame classifier:", acc1)
#Think about the baseline accuracy
```

```
Accuracy of lame classifier: 0.551
```

9.3 Distributions

9.3.1 Mean Density Comparison Function

Write a function which:

1. draws repeated (e.g. M=500) random samples of size n (e.g. 40, 640) from each sex from the data
2. Computes the stdevs for the sample means of each sex separately
3. Repeats the above density plot for the sample mean distributions
4. computes the confusion matrix/accuracy of a classifier that applies the rule $\bar{x} > 3480$.

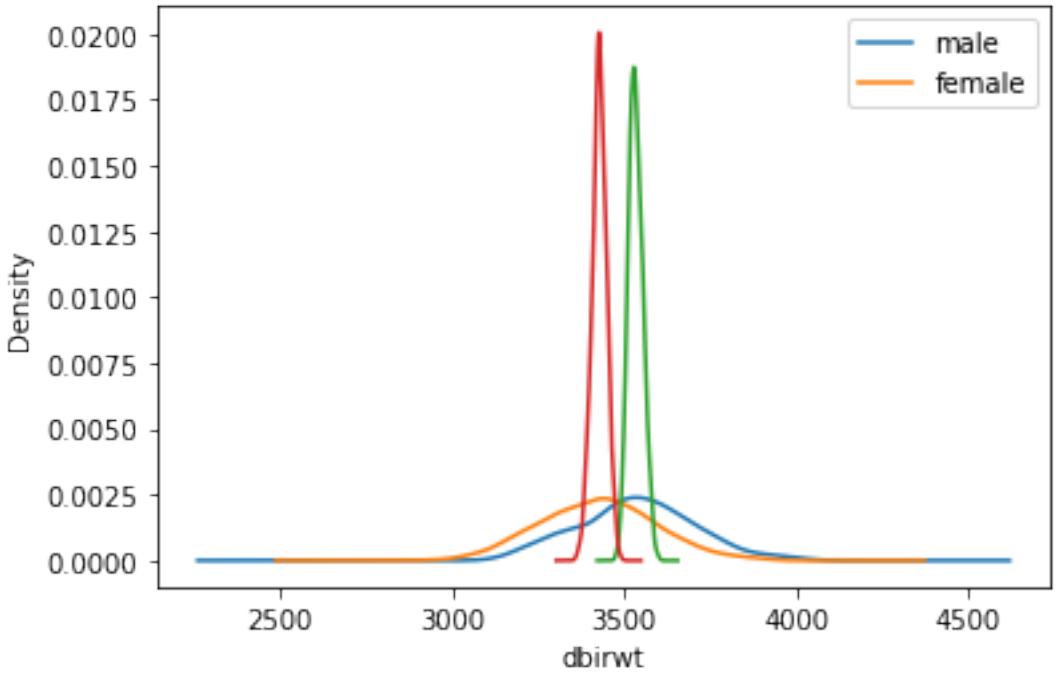
```
Hint: np.random.choice(df["dbirwt"],2)
```

```
def mean_density_comparison(df_cleaned, M=500, n=10):

    #Generate a sex iteration array
    sex_iter = ['male', 'female']
```

```
    #Create an empty DataFrame with 'sex' and 'dbirwt' column
    columns = ['sex', 'dbirwt']
    df_new = pd.DataFrame(columns=columns)

    #Create an empty array to store the standard deviation of the differnt sex 'male' = st
    std_dev = np.empty(2)

    #Iterate over sex and create a specific data subset
    for ind,v in enumerate(sex_iter):
        subset = df_cleaned[df_cleaned.sex == v]

        #create M random sample means of n samples and add it to df_new
        for i in range(M):
            rand_samples = np.random.choice(subset.dbirwt, n)
            x = np.mean(rand_samples)#sample mean per sex
            df_new.loc[len(df_new)+1] = [v, x]

        #plot male and female data and calculate the standard deviation of the data
        plot_data = df_new[df_new.sex == v]
        std_dev[ind] = np.std(plot_data['dbirwt'])

        plot_data.dbirwt.plot.density()
        plt.xlabel('dbirwt')
        plt.legend(sex_iter)
        #plt.grid()
        #plt.title("n=" + str(n))

    #return the sample mean data
    return df_new
```

Testing the Function

```
SM10 = mean_density_comparison(df, M=500, n=10)
SM640 = mean_density_comparison(df, M=500, n=640)
```




```
SM10["predMale"] = (SM10["dbirwt"] > 3480)
ConfMat10 = pd.crosstab(SM10["predMale"], SM10["sex"])
ConfMat10
```

sex predMale	female	male
False	320	182
True	180	318

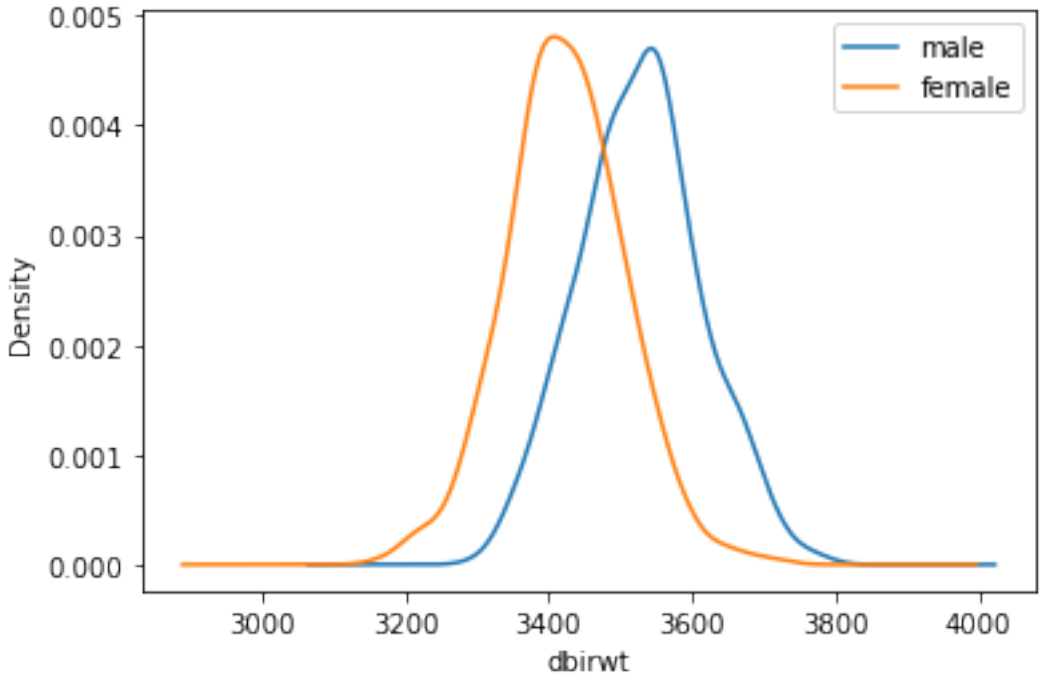
```
SM640["predMale"] = (SM640["dbirwt"] > 3480)
ConfMat640 = pd.crosstab(SM640["predMale"], SM640["sex"])
ConfMat640
```

sex predMale	female	male
False	498	0
True	2	500

```
grouped640 = SM640["dbirwt"].groupby(SM640["sex"])

print("n=640, means:", grouped640.mean())
print()
print("n=640, SESMs:", grouped640.std())
```

```
n=640, means: sex
female    3427.276397
male      3533.081803
Name: dbirwt, dtype: float64

n=640, SESMs: sex
female    20.937106
male      19.819436
Name: dbirwt, dtype: float64
```

```
SM40 = mean_density_comparison(df, M=500, n=40)
```



```
grouped40 = SM40["dbirwt"].groupby(SM40["sex"])

print("n=40, means:", grouped40.mean())
print()
print("n=40, SESMs:", grouped40.std())
```

```
n=40, means: sex
female    3423.30740
male      3527.34015
Name: dbirwt, dtype: float64

n=40, SESMs: sex
female    82.787145
male      84.921927
Name: dbirwt, dtype: float64
```

How much smaller is $\sigma_{\bar{x},640}$ than $\sigma_{\bar{x},40}$? Compare that factor to the ratio of the sample sizes $640/40 = 16$

9.3.2 Empirical Cumulative Distribution Function

The density -like a histogram- has a few complications that include the arbitrary choice of bin width (kernel width for density) and the loss of information. Welcome to the *empirical cumulative distribution function* **ecdf**

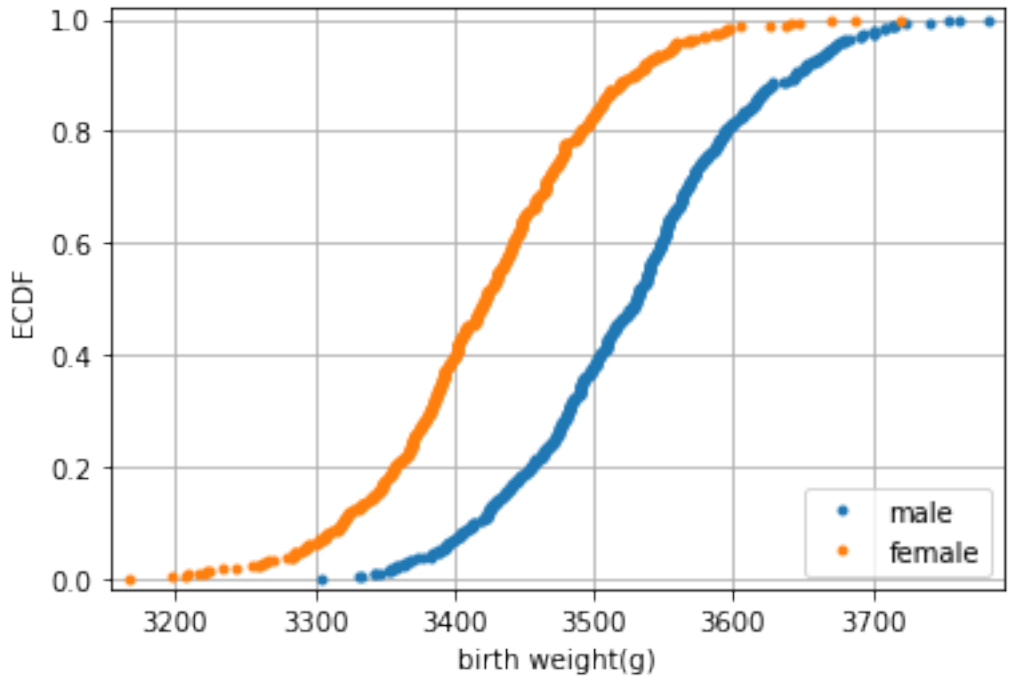
ECDF Function

```
def ecdf(data):
    """Compute ECDF for a one-dimensional array of measurements."""

    # Number of data points: n
    n = len(data)

    # x-data for the ECDF: x
    x = np.sort(data)

    # y-data for the ECDF: y
    y = np.arange(1, n+1) / n
```

```
    return x, y
```

ECDF Plot

```
# Compute ECDF for sample size 40: m_40, f_40
male40 = SM40[SM40.sex == "male"]["dbirwt"]
female40 = SM40[SM40.sex == "female"]["dbirwt"]

mx_40, my_40 = ecdf(male40)
fx_40, fy_40 = ecdf(female40)

# Plot all ECDFs on the same plot
fig, ax = plt.subplots()
_ = ax.plot(mx_40, my_40, marker = '.', linestyle = 'none')
_ = ax.plot(fx_40, fy_40, marker = '.', linestyle = 'none')

# Make nice margins
plt.margins(0.02)

# Annotate the plot
plt.legend(('male', 'female'), loc='lower right')
_ = plt.xlabel('birth weight(g)')
_ = plt.ylabel('ECDF')

# Display the plot
plt.grid()
plt.show()
```

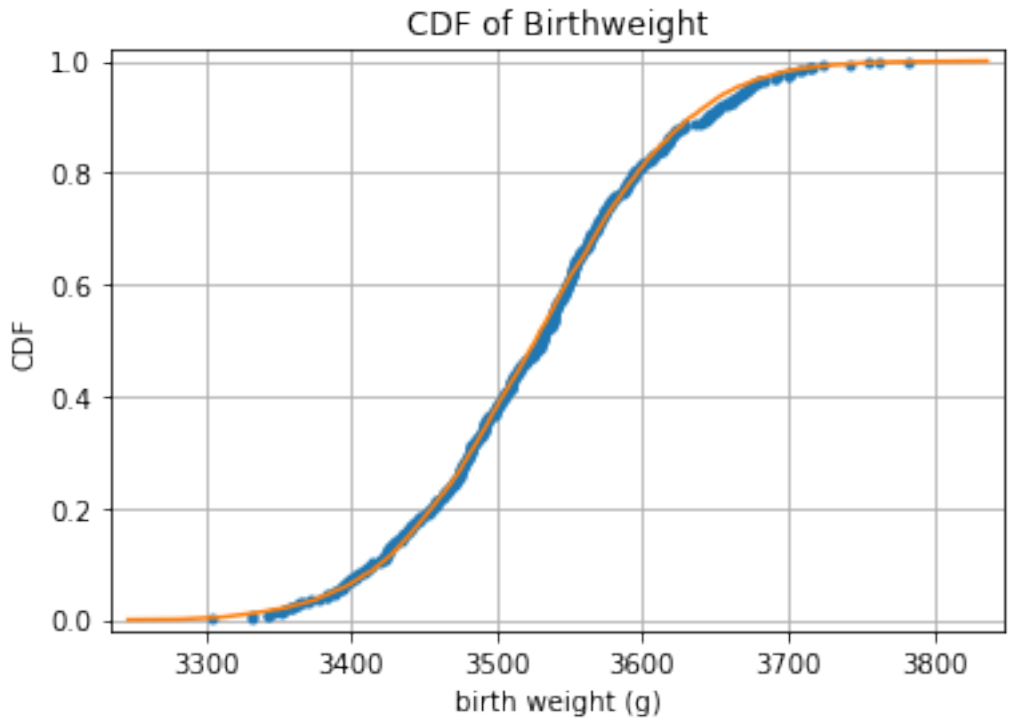


- What is the relationship to quantiles/percentiles ?
- Find the IQR !
- Sketch the densities just from the ecdf.

9.3.3 Checking Normality of sample mean distribution

```
# Compute mean and standard deviation: mu, sigma
mu = np.mean(male40)
sigma = np.std(male40)

# Sample out of a normal distribution with this mu and sigma: samples
samples = np.random.normal(mu, sigma, 10000)

# Get the CDF of the samples and of the data
x_theor, y_theor = ecdf(samples)
```

```
# Plot the CDFs and show the plot
_ = plt.plot(mx_40, my_40, marker='.', linestyle='none')
_ = plt.plot(x_theor, y_theor)

plt.margins(0.02)
_ = plt.xlabel('birth weight (g)')
_ = plt.ylabel('CDF')
_ = plt.title('CDF of Birthweight')

plt.grid()
plt.show()
```



Tasks

1. Find the "5% tails" which are just the (0.05, 0.95) quantiles
2. Read up on theoretical quantiles: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.h
3. stone age: get the "5% tails" from a normal table.
4. How many stdevs do you need to cover the 90% sample interval ?
5. Can you replace the "empirical theoretical cdf" from above with the exact line without sampling 10000 random numbers from a normal distribution ?

Let us recap what we observed when sampling from a "population": The *sample mean distribution* gets narrower with increasing sample size n, SESM $=\sigma_{\bar{x}} = \sigma/\sqrt{n}$. Take a look at this interactive applet for further understanding.

How is this useful ? And how is it relevant because in reality we would only have **one sample**, not hundreds !

Small Tasks

1. Choose one random sample of size n=40 from the male babies and compute $\bar{x}$, $\hat{\sigma}$. Assume all that is known to you, are these two *summary statistics*. In particular, we do **not know** the true mean μ!
2. Argue intuitively with the ecdf plot about plausible values of μ.
3. More precisely: what interval around $\bar{x}$ would contain μ with 90% probability ?

9.4 Hacker Statistic

The ability to draw new samples from a population with a known mean is a luxury that we usually do not have. Is there any way to "fake" new samples using just the one "lousy" sample we have at hand ? This might sound like an impossible feat analogously to "pulling yourself up by your own **bootstraps**"!

Bootstrap Ilustration

But that is exactly what we will try now:

Tasks

1. Look up the help for np.random.choice()
2. Draw repeated samples of size n=40 from the sample above.
3. Compute the mean of each sample and store it an array.
4. Plot the histogram
5. Compute the stdev of this distribution and compare to the SEM.
6. Write a function that computes *bootstrap replicates* of the mean from a sample.
7. Generalize this function to accept any summary statistic, not just the mean.

10 Hypothesis Tests

Overview

1. Hypothesis Tests
    - 1-sample
    - 2-sample
2. Permutation Tests

10.1 WarmUp/Review Exercises

Load the *Auto.csv* data (into a dataframe *cars*) for the following tasks and create a scatter plot `mpg ~ weight`. Take a look at row with index 25 (i.e. row 26); we will refer to that data point as x_{25} from now on.

- Compute the standard score for
    - x_{25}["mpg"]
    - x_{25}["weight"]
- Compute the product of these standard scores (call it p_{25}).
- If you repeated this process for all rows of the *cars* dataframe and averaged all products p_i, what would the resulting number tell you ? (What is it called?)
- Take a bootstrap sample from cars.mpg and compute the mean

Importing Standard Libraries

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
#pd.options.mode.chained_assignment = None # disable chained assignment warning
import seaborn as sns
```

Loading our Functions

```
%run ./ourFunctions.py
%precision 3
```

```
'%.3f'
```

Read in the Cars DF

```
#cars = pd.read_csv('../data/Auto.csv')
cars = pd.read_csv("https://raw.githubusercontent.com/markusloecher/DataScience-HWR/main/d
#d
cars.shape
```

```
(392, 10)
```

```
cars.head()
```

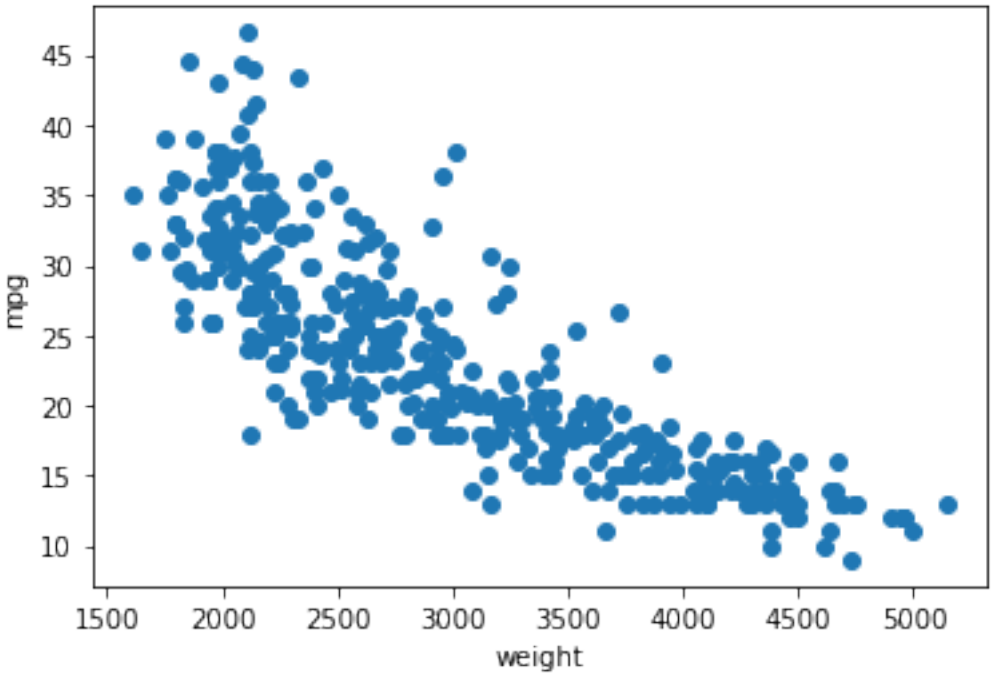
	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle mal
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino

```
print(cars.iloc[25,])
```

```
mpg                 10.0
cylinders              8
displacement       360.0
horsepower           215
weight              4615
acceleration        14.0
```

```
year                    70
origin                   1
name            ford f250
Manufacturer          ford
Name: 25, dtype: object
```

```
plt.scatter("weight", "mpg",data=cars)
plt.xlabel("weight")
plt.ylabel("mpg")
plt.show()
```



10.2 Hypothesis Tests

We have learned about the **bootstrap** as a slick way of resampling your data to obtain sampling distributions of various measures of interest, Without having to learn highly specific distributions (such as the χ^2, Poisson, Binomial or F-distribution) the bootstrap enables us to

1. get **confidence intervals**
2. perform **one-sample tests**

3. perform **two-sample tests**

Imagine the EPA requiring the average mpg for 4-cylinder cars to be at least $\mu_0 = 30$ and needs to decide -based on this sample only- whether the manufacturers need to implement some improvements. In statistical jargon: can the EPA **reject the claim** that the **true mean** is at least 30 ?

```
cars4=cars[cars["cylinders"]==4]
empirical_mean = np.mean(cars4.mpg)
empirical_mean
```

```
29.28391959798995
```

```
empirical_sd = np.std(cars4.mpg)
empirical_sd
```

```
5.656280603443601
```

Compute a confidence interval of the true mpg of 4-cyl cars via the bootstrap ! (Is there just "THE CI" ??)

```
mpg_bs = draw_bs_reps(cars4.mpg, func = np.mean, size=1000)
```

```
np.percentile(mpg_bs, [2.5, 97.5])
```

```
array([28.498, 30.093])
```

```
from scipy.stats import norm
norm.ppf(0.975)
```

```
1.959963984540054
```

There, we have it, the empirical mean is in fact below 30. Is this sufficient evidence for the EPA to demand improvements ? Does $\bar{x} < \mu_0$ "prove" that $\mu < \mu_0$??

Think about an experiment which tries to establish whether a pair of dice in a casino is biased ("gezinkt" in German). You toss the dice 50 times and observe an average number of pips of $\bar{x} = 3.8$ which is clearly "way above" the required value of $\mu_0 = 3.5$ for a clean pair of dice. Should you go to the authorities and file a complaint ? Since no one really knows the true value of μ for these dice, how would a court decide whether they are unbiased ?

1. **Innocent until proven guilty**: Always assume the opposite of what you want to illustrate ! That way you can gather evidence against this **Null hypothesis**, e.g. $H_0 : \mu = 3.5 (\equiv \mu_0)$.
2. Define a **test statistic** which is pertinent to the question you are seeking to answer (computed entirely from your sample). In this case, your test statistics could be e.g. the (scaled?) difference between the observed sample mean and the claim: $\bar{x} - \mu_0$.
3. We define the **p-value** as the probability of observing a test statistic equally or more extreme than the one observed, given that H_0 is true.
4. If the observed value of your test statistic seems very unlikely under H_0 we would favor the **alternative hypothesis** which is usually complementary, e.g. $H_A : \mu \neq 3.5$

Notice that we used vague expressions such as *unlikely* in the procedure above. In a legal context, we would want to make statements such as "beyond a reasonable doubt". Well, in statistics we ry to quantifiy that notion by limiting the type-I error probability α to be at most e.g. 0.01 or 0.05. So none our decisions are ever "proofs" or free of error.

It also means that you, the person conducting the hypothesis test, need to specify a value of α, which clearly introduces a certain amount of subjectivity. Later on, we will discuss the inherent tradeoffs between type-I and type-II errors: only then will one fully understand the non-arbitrary choice of α.

10.2.1 Parametric Tests

1. From the sample, compute $\bar{x}$ and $\hat{\sigma}$.
2. Compute the test statistic

$$t = \frac{\bar{x} - \mu_0}{\hat{\sigma}/\sqrt{n}}$$

3. Reject H_0
   - Two-sided test: if $|t| > t_{n-1,1-\alpha/2}$
   - Left-sided test: if $t < t_{n-1,\alpha}$
   - Right-sided test: if $t > t_{n-1,1-\alpha}$
   - If $n > 50$ one may replace the t distribution with the normal

4. Define the p-value as twice the tail area of a t distribution
5. Alternatively one rejects the Null if p-val $< \alpha$.

Back to the casino example, assume the following empirical data:

$\bar{x} = 3.8, n = 100, \hat{\sigma} = 1.7$

$H_0 : \mu = 3.5$, $H_A : \mu \neq 3.5$, $\alpha = 0.01$

```
from scipy.stats import norm
# that is the critical value that we compare our test statistic with
print('two-sided cv: ',norm.ppf(1-0.01/2)) # returns the norm score
print('one-sided cv: ',norm.ppf(1-0.01))
```

```
two-sided cv:  2.5758293035489004
one-sided cv:  2.3263478740408408
```

10.2.1.1 Two sided test

```
# We care about both sides since we do not know how the casino is using the biased dice.
a=0.01
n=200

critValue = norm.ppf(1-a/2)
z = np.abs((3.8-3.5)/(1.7/np.sqrt(n)))

if (z < critValue):
    s = "not"
    s2 = "fail to"
else:
    s = ""
    s2 = ""

print("The test statistic t =", np.round(z,3), "is", s ,"larger than the critical value",n
norm.cdf(1-a/2)
```

```
The test statistic t = 2.496 is not larger than the critical value 2.576 so we fail to reject
```

```
0.840131867824506
```

10.2.1.2 One sided test!

```
# We care only about one side since we do know that the casino makes
# money off upward biased dice.

critValue = norm.ppf(1-a)
z = (3.8-3.5)/(1.7/np.sqrt(n))
if (z < critValue):
    s = "not"
    s2 = "fail to"
else:
    s = ""
    s2 = ""

print("The test statistic t =", np.round(z,3), "is", s ,"larger than the critical value",n
#norm.cdf(1-a)
```

```
The test statistic t = 2.496 is  larger than the critical value 2.326 so we  reject the Null
```

10.2.1.3 p-value

```
#p-value is the right tail probability of values equal or larger than your test statistic
pVal = 1-norm.cdf(z)
print("The p value ", np.round(pVal,3), "is less than alpha", a, "so we reject the Null")
```

```
The p value  0.006 is less than alpha 0.01 so we reject the Null
```

10.2.2 Non parametric Tests

Parametric Tests require many assumptions, that may not always be true, and are also a bit abstract. It often helps (i) robustness and (ii) the understanding of the testing process to use simulations instead. We now learn about two different ways of such simulations, each following the basic "pipeline":

1. Clearly define the Null hypothesis.
2. Define the test statistic.

3. Generate many sets of simulated data (bootstrap replicates) **assuming the null hypothesis is true**
4. Compute the test statistic for each simulated data set
5. The p-value is the fraction of your simulated data sets for which the test statistic is at least as extreme as for the real data

10.2.2.1 Bootstrap Hypothesis Tests

Besides computing confidence intervals we can also use the bootstrap to perform hypothesis test. We need to "fake" the process of drawing new samples again and again under the Null hypothesis, which might appear impossible since our one sample likely will have a sample mean not equal to μ_0.

The trick is to "shift" our data such that $\bar{x} = \mu_0$! As a test statistic we use the difference of the mean of the bootstrap value minus μ_0. Notice that there is no need to scale this difference!

We illustrate this with our casino example.

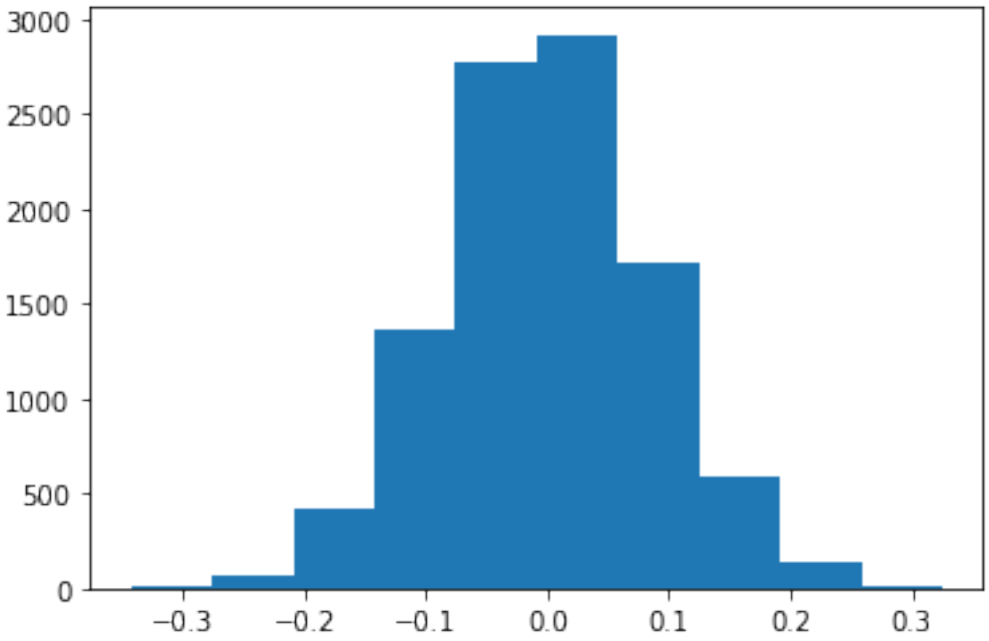
```
np.random.choice(6,20)+1
```

```
array([5, 6, 4, 5, 2, 2, 3, 2, 5, 6, 4, 1, 3, 4, 5, 5, 6, 5, 5, 4])
```

```
np.random.seed(123)
n=400
#x = np.random.choice(6,n,p=[0.14,0.14,0.15,0.18,0.19,0.20])+1 #add a bias
x = np.random.choice(6,n)+1
#cheating:
x = x + 0.2 #add a bias
a=0.01

mu0=3.5
xBar = np.round(np.mean(x),2)
obsDiff = np.round(mu0-xBar,2)
print("The sample mean is", xBar, ", so we shift our data by", obsDiff)
```

```
The sample mean is 3.65 , so we shift our data by -0.15
```

```
#Alternatively, if we had not shifted our data
#TS = bs_mean_dice-xBar

#The test statistic
bs_mean_dice = draw_bs_reps(x+obsDiff,np.mean,10000)
TS = bs_mean_dice-mu0
plt.hist(TS);
```




```
#two sided p value
pVal = np.mean( abs(TS) > abs(obsDiff))
print("The two sided p value of", np.round(pVal,3), "is not smaller than alpha=",a,"so we
```

```
The two sided p value of 0.012 is not smaller than alpha= 0.01 so we fail to reject the Null
```

10.2.2.2 Tasks

- Test $H_0 : \mu \geq 30, H_A : \mu < 30$ for the mean mpg of 4-cylinder cars
  - using bootstrap replicates

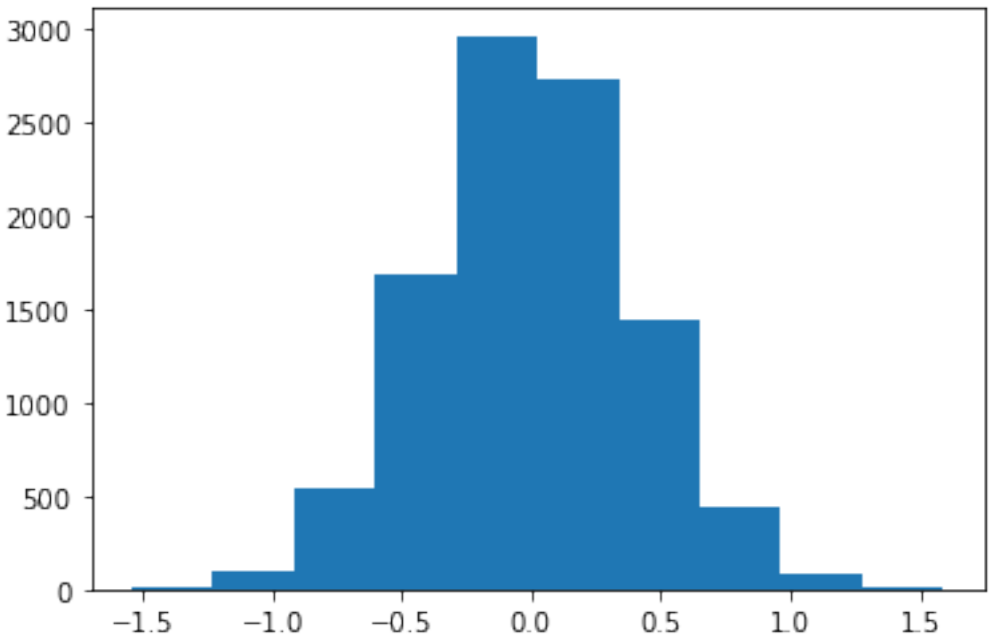
  - via standard testing theory.

- Compute the corresponding **p-values**.

```
mu0=30
empirical_mean = np.mean(cars4.mpg)
```

```
#assume the claim is true !!
shift = mu0-empirical_mean
cars4Shifted= cars4.mpg+shift
bs_mean_mpg = draw_bs_reps(cars4Shifted,np.mean,10000)

#The test statistic
TS = bs_mean_mpg-mu0
pVal = np.mean( TS < -shift)
```

```
#assume the claim is true !!
shift = mu0-empirical_mean
cars4Shifted= cars4.mpg+shift
bs_mean_mpg = draw_bs_reps(cars4Shifted,np.mean,10000)

#The test statistic
TS = bs_mean_mpg-mu0
pVal = np.mean( TS < -shift)

plt.hist(TS);

#np.percentile(bs_mean_mpg,5)
#p value is simply the left tail beyond xBar
#np.mean(bs_mean_mpg < empirical_mean)
```




```
print('shift =',shift)
print("pValue =", pVal)
```

```
shift = 0.7160804020100429
pValue = 0.0375
```

10.2.3 From one sample to 2 samples

From Auto to birth weights

```
preg = pd.read_csv('data/pregNSFG.csv.gz', compression='gzip')

#only look at live births
firsts = preg[(preg.outcome == 1) & (preg.birthord == 1)]
#live[live.babysex == 1].babysex = "male"

#we reduce the sample size further by conditioning on
#the mother's age at the end of pregnancy
firsts = firsts[(firsts.agepreg < 30) & (firsts.prglngth >= 30)]
```

```
bwt = firsts[["babysex","totalwgt_lb"]]

bwt.babysex.replace([1.0],"male",inplace=True)
bwt.babysex.replace([2.0],"female",inplace=True)
bwt = bwt.dropna()
```

```
print('shape:',bwt.shape)
bwt.head()
```

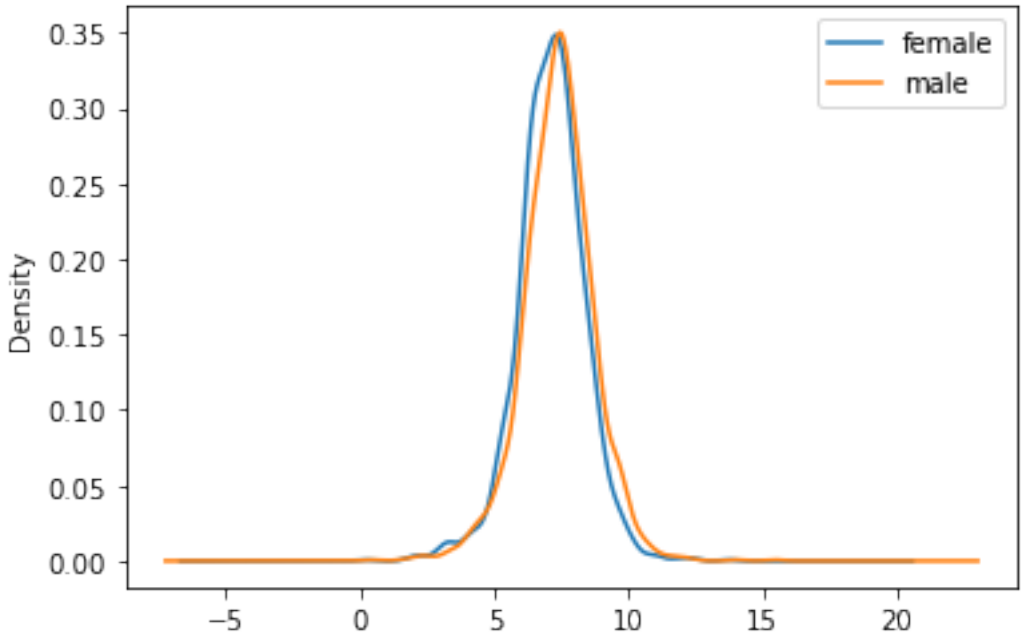
```
shape: (3758, 2)
```

	babysex	totalwgt_lb
2	male	9.1250
5	male	8.5625
8	male	7.5625
10	male	7.8125
11	female	7.0000

```
grouped = bwt["totalwgt_lb"].groupby(bwt["babysex"])
grouped.mean()
```

```
babysex
female    7.103830
male      7.378682
Name: totalwgt_lb, dtype: float64
```

```
tmp=grouped.plot(kind='density', legend=True);
```



10.2.3.1 A two-sample bootstrap hypothesis test for difference of means

A one sample test compares a data set to one fixed number !

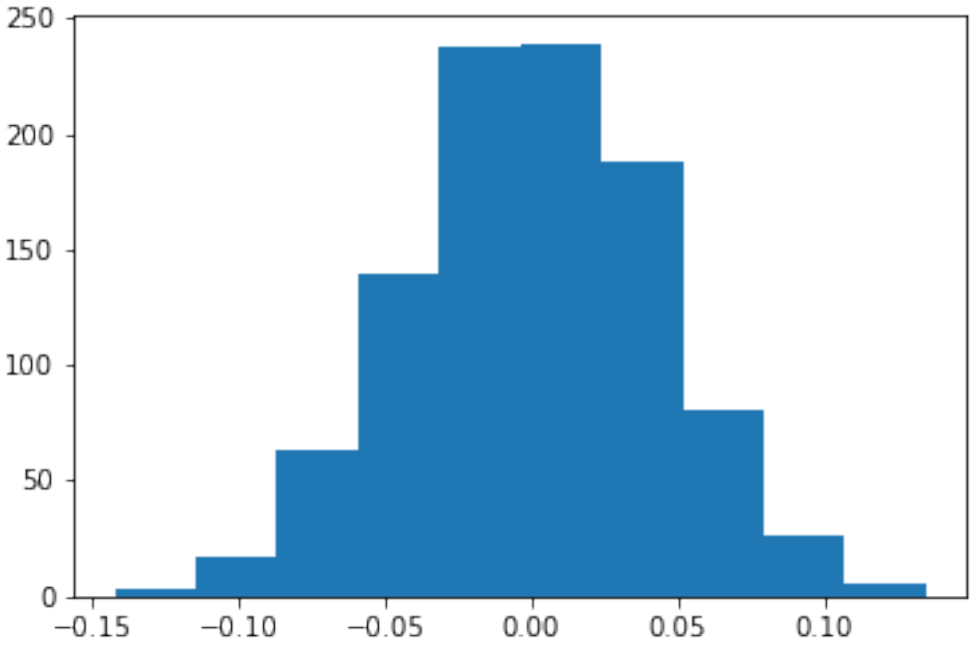
We now want to compare two sets of data, both of which are samples! In particular test the hypothesis that male and female babies have the same biological weight (but not necessarily the same distribution).

$H_0 : \mu_m = \mu_f, H_A : \mu_m \neq \mu_f$

To do the two-sample bootstrap test, we shift both arrays to have the same mean, since we are simulating the hypothesis that their means are, in fact, equal (**equal to what value ??**). We then draw bootstrap samples out of the shifted arrays and compute the difference in means. This constitutes a bootstrap replicate, and we generate many of them. The p-value is the fraction of replicates with a difference in means greater than or equal to what was observed.

```
meanNull = np.mean(bwt.totalwgt_lb)# pooled mean
w_m = bwt[bwt["babysex"]=="male"].totalwgt_lb
w_f = bwt[bwt["babysex"]=="female"].totalwgt_lb
empirical_diff_means = np.mean(w_m)-np.mean(w_f)
#shift:
```

```
w_m_shifted = w_m - np.mean(w_m) + meanNull
w_f_shifted = w_f - np.mean(w_f) + meanNull


# Compute 10,000 bootstrap replicates from shifted arrays
M=1000
bs_replicates_m = draw_bs_reps(w_m_shifted, np.mean, M)
bs_replicates_f = draw_bs_reps(w_f_shifted, np.mean, M)

# Get replicates of difference of means: bs_replicates
bs_replicates = bs_replicates_m - bs_replicates_f

tmp=plt.hist(bs_replicates)
# Compute and print p-value: p
```




```
#debugging:
if 0:
    meanNull
    #plt.hist(w_f_shifted)
    #plt.hist(bs_replicates_m)
```

```
    #bs_replicates_m
    w_m_bs = np.random.choice(w_m, size=len(w_m))
    np.nanmean(w_m_bs)
    np.argwhere(np.isnan(w_m_bs))
    np.argwhere(np.isnan(w_m))

#p-value (one-sided):
pVal = np.mean(bs_replicates> empirical_diff_means)
#cutoff right tail
#np.percentile(bs_replicates, 95)
print("The one sided p value of", np.round(pVal,3), "is much smaller than alpha=",a,", so

print("-> The observed difference of ", np.round(empirical_diff_means,5), "is exremely unl
```

```
The one sided p value of 0.0 is much smaller than alpha= 0.01 , so we fail to reject the Null
-> The observed difference of  0.275 is exremely unlikely to have occurred by chance alone
```

11 AB Testing

1. Permutation Tests
2. Binary Processes
    - AB Tests
    - Binomial Distribution

Importing Our Functions

```
%run ./ourFunctions.py
%precision 3
```

```
'%.3f'
```

```
# %load ../ourFunctions.py
import numpy as np
import matplotlib as matplt
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from scipy.stats import norm
from scipy import stats

from numpy import random
#random.seed(42)
```

11.1 Permutation 2-sample test

We have used the bootstrap to compare two sets of data, both of which are samples. In particular, we can test two-sample hypotheses such as

$H_0 : \mu_m = \mu_f, H_A : \mu_m \neq \mu_f$

or the one-sided versions:

$H_0 : \mu_m = \mu_f, H_A : \mu_m > \mu_f$

$H_0 : \mu_m = \mu_f, H_A : \mu_m < \mu_f$

Another way to compare 2 distributions (in some ways much more straightforward than the bootstrap) is **permutation sampling**. It directly simulates the hypothesis that two variables have identical probability distributions.

A permutation sample of two arrays having respectively n_1 and n_2 entries is constructed by concatenating the arrays together, scrambling the contents of the concatenated array, and then taking the first n_1 entries as the permutation sample of the first array and the last n_2 entries as the permutation sample of the second array.

At DataCamp the first example offers a nice visualization of this process:

Take a look at the code in *ourFunctions.py* to run a permutation test

---

Let us apply our first permutation sampling on the Titanic data. (First, we explore the data a bit)

```
titanic = sns.load_dataset('titanic')
titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	dec
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	Na
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	Na
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	Na

```
PclassSurv = titanic.groupby(['pclass', 'survived'])
PclassSurv.size()
```

```
pclass  survived
1       0            80
        1           136
2       0            97
        1            87
3       0           372
        1           119
dtype: int64
```

```
pd.crosstab(titanic.pclass, titanic.survived,margins=True)
```

survived pclass	0	1	All
1	80	136	216
2	97	87	184
3	372	119	491
All	549	342	891

```
WomenOnly = titanic[titanic["sex"]=="female"]
pd.crosstab(WomenOnly.pclass, WomenOnly.survived,margins=True)
```

survived pclass	0	1	All
1	3	91	94
2	6	70	76
3	72	72	144
All	81	233	314

Test the claim that the survival chances of women in 1st and 2nd class were pretty much the same.

1. Write down the Null hypothesis and test statistic
2. Write code that generates permutation samples from two data sets
3. Generate many **permutation replicates** for the relevant Titanic subset
4. Compute a p-value

We could choose alpha = 0.05, but keep in mind the following - would you step into a plane that has a 5% crash probability ? - Would you buy a drug that has a 5% chance of severe side effects ?

What is the difference between these two methods (bootstrap, permutation) ?

Testing the hypothesis that two samples have the same distribution may be done with a bootstrap test, but a permutation test is preferred because it is more accurate (exact, in fact). But a permutation test is not as versatile as the bootstrap.

We often want to test the hypothesis that population A and population B have the same mean, but not necessarily the same distribution. This is difficult with a permutation test as it assumes **exchangeability**.

We will get back to this topic!

More info..

11.2 2-sample t test

Of course there is an equivalent fully parametric 2-sample test, the t-test.

We first read in the The National Survey of Family Growth data from the think stats book.

Look at section 1.7 for a description of the variables.

```
#preg = pd.read_hdf('data/pregNSFG.h5', 'df')
preg = pd.read_csv('data/pregNSFG.csv.gz', compression='gzip')
#only look at live births
live = preg[preg.outcome == 1]

#define first babies
firsts = live[live.birthord == 1]

#and all others
others = live[live.birthord != 1]
```

```
tRes = stats.ttest_ind(firsts.prglngth.values, others.prglngth.values)
p = pd.Series([tRes.pvalue,tRes.statistic], index = ['p-value:', 'test statistic:'])
p
```

```
p-value:           0.167554
test statistic:    1.380215
dtype: float64
```

```
#ttest_ind often underestimates p for unequal variances:
tRes = stats.ttest_ind(firsts.prglngth.values, others.prglngth.values, equal_var = False)
p = pd.Series([tRes.pvalue,tRes.statistic], index = ['p-value:', 'test statistic:'])
p
```

```
p-value:           0.168528
test statistic:    1.377059
dtype: float64
```

Can you reproduce the first p-value from the test statistic ?

Random Walks

Simulating Many Random Walks at Once

Random Walks

If your goal was to simulate many random walks, say 5,000 of them, you can generate all of the random walks with minor modifications to the above code. The numpy.random functions if passed a 2-tuple will generate a 2D array of draws, and we can compute the cumulative sum across the rows to compute all 5,000 random walks in one shot

---
##
Random
Walks

::: {.cell
execu-
tion_count=15}
::: {.cell
execu-
tion_count=16}
::: {.cell-
output
.cell-
output-
display
execu-
tion_count=16}
::: {.cell
execu-
tion_count=17}
‘‘‘
{.python
.cell-code}
nwalks =
5000
nsteps =
10000
steps =
random.choice([-
1,1],
size=(nwalks,
nsteps),
p=(0.9,0.1))
# 0 or 1
#steps =
np.where(draws
> 0, 1, 0)
#steps =
np.where(np.random.rand()
<=
0.1,1,-1)

Random Walks

walks = steps.cumsum(1)

```
:::
::: {.cell execution_count=18}
::: {.cell-output .cell-output-display execution_count=18}
::: {.cell execution_count=16}
::: {.cell-output .cell-output-stdout}
::: {.cell execution_count=26}
```

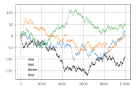
{.python .cell-code}

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
```

```
##
Random
Walks
ax.plot(walks[1,:],
'k', la-
bel='one',
linewidth=0.25)
ax.plot(walks[2,:],
la-
bel='two',
linestyle
='-',
linewidth=0.25)
ax.plot(walks[3,:],
la-
bel='three',
linestyle
='-',
linewidth=0.25)
ax.plot(walks[4,:],
la-
bel='four',
linestyle
='-',
linewidth=0.25)
ax.grid()
#a very
useless
legend
just
because
we can
ax.legend(loc='best');
'''
```

Random Walks

::: {.cell-output .cell-output-display}



::: :::

11.2.1 The $\sqrt{n}$ law again !

Compute the variance and/or standard deviation at times $1000, 4000, 9000$.

```
print('std_1000:', np.std(walks[:,1000]))
print('std_4000:', np.std(walks[:,4000]))
print('std_9000:', np.std(walks[:,9000]))
```

```
std_1000: 31.985538009544253
std_4000: 63.65415401998522
std_9000: 95.417731496824
```

Extra Credit

Out of these walks, let's compute the minimum crossing time to 30 or -30. This is slightly tricky because not all 5,000 of them reach 30. We can check this using the *any* method.

We can then use this boolean array to select out the rows of walks that actually cross the absolute 30 level and call argmax across axis 1 to get the crossing times:

```
hits30 = (np.abs(walks) >= 30).any(1)
hits30
hits30.sum() # Number that hit 30 or -30

crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)
crossing_times.mean()
```

```
892.4928
```

Tasks

Binomial Probability Distribution

1. Explore the *binom* function from scipy.stats
2. Size matters: insurance company A insures 100 cars, company B 400 cars. The probability of a car being stolen is 10%. Compute the probabilities that more than 15% of the respective fleets are stolen.

Tasks

4. Faced with a mutliple choice test containing 20 question with 4 choices each you decide in desparation to just guess all answers. What is the probability that you will pass, i.e. get at least 10 correct answers?
5. Think about non-parametric versions of the above answers
::: {.cell execu-tion_count=29}

11.2.2 A/B Testing

1. Perform a permutation test on the DataCamp example:

What does A/B testing have to do with random walks?

Table 3.11 compares the coefficient estimates obtained from two separate multiple regression models. The first is a regression of balance on age and limit, and the second is a regression of balance on rating and limit. In the first regression, both age and limit are highly significant with very small pvalues. In the second, the collinearity between limit and rating has caused the standard error for the limit coefficient estimate to increase by a factor of 12 and the p-value to increase to 0.701. In other words, the importance of the limit variable has been masked due to the presence of collinearity. To avoid such a situation, it is desirable to identify and address potential collinearity problems while fitting the model.

11.2.3 Qualitative/Categorical Variables

- Dummy Coding
- Interpretation of coefficients

Table 3.7

```
est = smf.ols('Balance ~ Gender ', credit).fit()
est.summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	509.8031	33.128	15.389	0.000	444.675	574.931
Gender[T.Female]	19.7331	46.051	0.429	0.669	-70.801	110.267

Table 3.8

```
np.unique(credit["Ethnicity"])
credit.groupby("Ethnicity")["Balance"].mean()
```

```
Ethnicity
African American    531.000000
Asian               512.313725
Caucasian           518.497487
Name: Balance, dtype: float64
```

```
est = smf.ols('Balance ~ Ethnicity', credit).fit()
est.summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	531.0000	46.319	11.464	0.000	439.939	622.061
Ethnicity[T.Asian]	-18.6863	65.021	-0.287	0.774	-146.515	109.142
Ethnicity[T.Caucasian]	-12.5025	56.681	-0.221	0.826	-123.935	98.930

11.2.4 Interactions

Table 3.9

```
est = smf.ols('Sales ~ TV + Radio + TV*Radio', advertising).fit()
est.summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	6.7502	0.248	27.233	0.000	6.261	7.239

TV	0.0191	0.002	12.699	0.000	0.016	0.022
Radio	0.0289	0.009	3.241	0.001	0.011	0.046
TV:Radio	0.0011	5.24e-05	20.727	0.000	0.001	0.001

11.2.4.1 Interaction between qualitative and quantitative variables

```
credit["Income2"] = credit["Income"] + scipy.stats.norm.rvs(400)
```

```
est1 = smf.ols('Balance ~ Income + Income2 + C(Student)', credit).fit()
regr1 = est1.params
est2 = smf.ols('Balance ~ Income + Income*C(Student)', credit).fit()
regr2 = est2.params

print('Regression 1 - without interaction term')
est1.summary().tables[1]
```

```
Regression 1 - without interaction term
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-0.0123	0.002	-7.143	0.000	-0.016	-0.009
C(Student)[T.Yes]	382.6705	65.311	5.859	0.000	254.272	511.069
Income	5.4557	0.621	8.779	0.000	4.234	6.677
Income2	0.5287	0.081	6.506	0.000	0.369	0.688

```
np.mean(credit["Income"])
```

```
45.218885000000036
```

```
est2 = smf.ols('Balance ~ Income + Income*C(Ethnicity)*C(Student)', credit).fit()
est2.summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	145.1346	64.922	2.236	0.026	17.492	272.77
C(Ethnicity)[T.Asian]	-15.5992	94.976	-0.164	0.870	-202.332	171.13
C(Ethnicity)[T.Caucasian]	122.8200	80.930	1.518	0.130	-36.297	281.93

C(Student)[T.Yes]	427.5858	246.321	1.736	0.083	-56.706	911.87
C(Ethnicity)[T.Asian]:C(Student)[T.Yes]	281.3665	297.683	0.945	0.345	-303.906	866.63
C(Ethnicity)[T.Caucasian]:C(Student)[T.Yes]	-109.7842	309.373	-0.355	0.723	-718.041	498.47
Income	7.1950	1.069	6.728	0.000	5.093	9.298
Income:C(Ethnicity)[T.Asian]	0.0963	1.667	0.058	0.954	-3.181	3.374
Income:C(Ethnicity)[T.Caucasian]	-2.0995	1.371	-1.531	0.127	-4.796	0.597
Income:C(Student)[T.Yes]	-0.0693	3.716	-0.019	0.985	-7.375	7.236
Income:C(Ethnicity)[T.Asian]:C(Student)[T.Yes]	-4.6311	4.475	-1.035	0.301	-13.429	4.167
Income:C(Ethnicity)[T.Caucasian]:C(Student)[T.Yes]	-0.9514	5.431	-0.175	0.861	-11.630	9.727

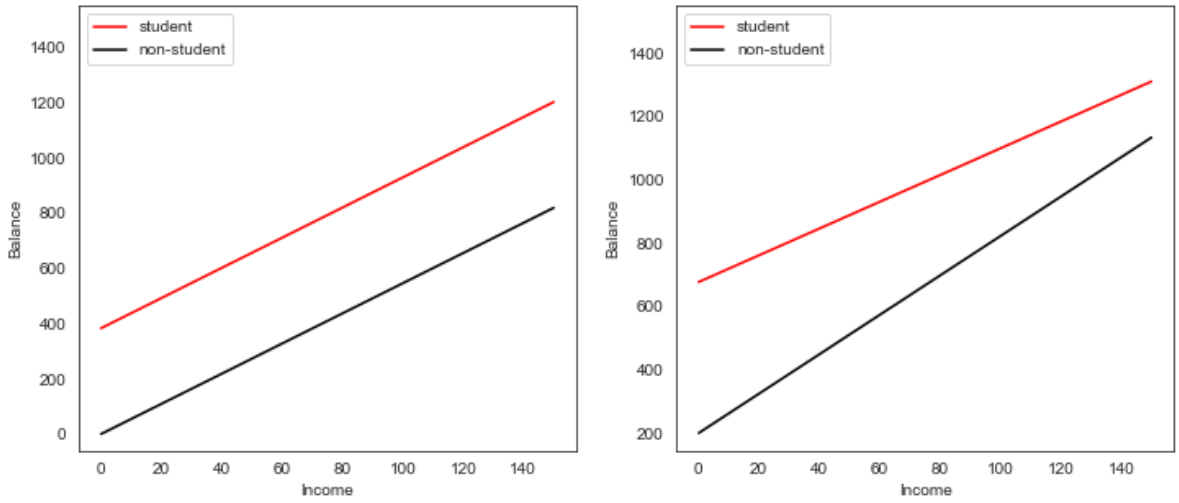
```
print('\nRegression 2 - with interaction term')
est2.summary().tables[1]
```

```
Regression 2 - with interaction term
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	145.1346	64.922	2.236	0.026	17.492	272.77
C(Ethnicity)[T.Asian]	-15.5992	94.976	-0.164	0.870	-202.332	171.13
C(Ethnicity)[T.Caucasian]	122.8200	80.930	1.518	0.130	-36.297	281.93
C(Student)[T.Yes]	427.5858	246.321	1.736	0.083	-56.706	911.87
C(Ethnicity)[T.Asian]:C(Student)[T.Yes]	281.3665	297.683	0.945	0.345	-303.906	866.63
C(Ethnicity)[T.Caucasian]:C(Student)[T.Yes]	-109.7842	309.373	-0.355	0.723	-718.041	498.47
Income	7.1950	1.069	6.728	0.000	5.093	9.298
Income:C(Ethnicity)[T.Asian]	0.0963	1.667	0.058	0.954	-3.181	3.374
Income:C(Ethnicity)[T.Caucasian]	-2.0995	1.371	-1.531	0.127	-4.796	0.597
Income:C(Student)[T.Yes]	-0.0693	3.716	-0.019	0.985	-7.375	7.236
Income:C(Ethnicity)[T.Asian]:C(Student)[T.Yes]	-4.6311	4.475	-1.035	0.301	-13.429	4.167
Income:C(Ethnicity)[T.Caucasian]:C(Student)[T.Yes]	-0.9514	5.431	-0.175	0.861	-11.630	9.727

```
# Income (x-axis)
income = np.linspace(0,150)

# Balance without interaction term (y-axis)
student1 = np.linspace(regr1['Intercept']+regr1['C(Student)[T.Yes]'],
                       regr1['Intercept']+regr1['C(Student)[T.Yes]']+150*regr1['Income'])
non_student1 =  np.linspace(regr1['Intercept'], regr1['Intercept']+150*regr1['Income'])
```

```
# Balance with iteraction term (y-axis)
student2 = np.linspace(regr2['Intercept']+regr2['C(Student)[T.Yes]'],
                       regr2['Intercept']+regr2['C(Student)[T.Yes]']+
                       150*(regr2['Income']+regr2['Income:C(Student)[T.Yes]']))
non_student2 =  np.linspace(regr2['Intercept'], regr2['Intercept']+150*regr2['Income'])

# Create plot
fig, (ax1,ax2) = plt.subplots(1,2, figsize=(12,5))
ax1.plot(income, student1, 'r', income, non_student1, 'k')

ax2.plot(income, student2, 'r', income, non_student2, 'k')

for ax in fig.axes:
    ax.legend(['student', 'non-student'], loc=2)
    ax.set_xlabel('Income')
    ax.set_ylabel('Balance')
    ax.set_ylim(ymax=1550)
```



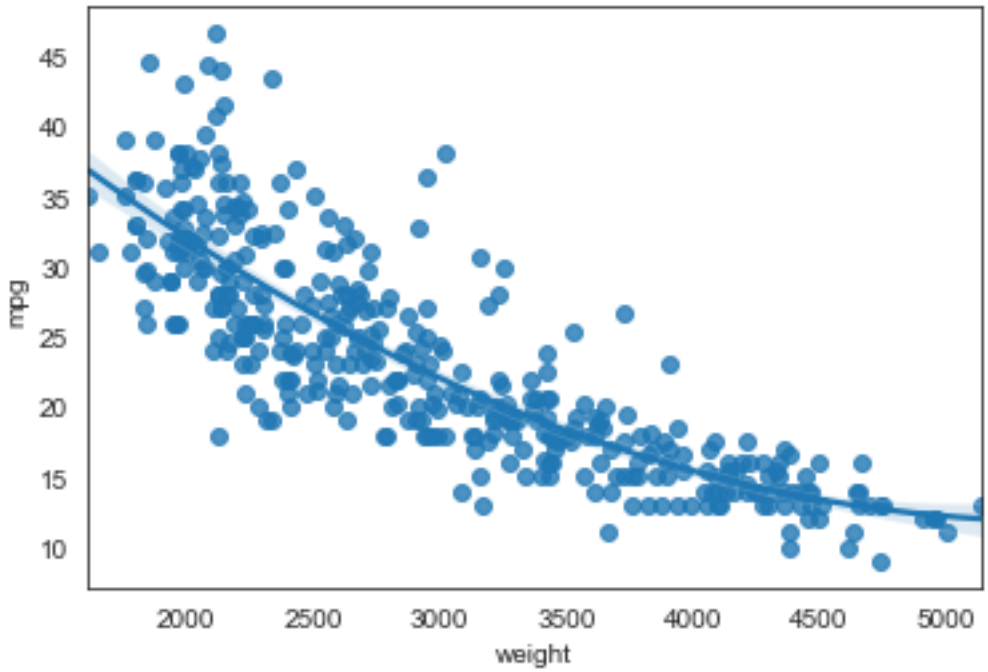
11.2.5 Non-linear relationships

Figure 3.8

```
auto = pd.read_csv('./data/Auto.csv', na_values='?').dropna()
```

```
sns.regplot(x='weight', y='mpg', data=auto,fit_reg=True, order=2);
```

```
<AxesSubplot:xlabel='weight', ylabel='mpg'>
```




```
# With Seaborn's regplot() you can easily plot higher order polynomials.
plt.scatter(x=auto.horsepower, y=auto.mpg, facecolors='None', edgecolors='k', alpha=.5)
sns.regplot(x=auto.horsepower, y=auto.mpg, ci=None, label='Linear', scatter=False, color='
sns.regplot(x=auto.horsepower, y=auto.mpg, ci=None, label='Degree 2', order=2, scatter=Fal
sns.regplot(x=auto.horsepower, y=auto.mpg, ci=None, label='Degree 5', order=5, scatter=Fal
plt.legend()
plt.ylim(5,55)
plt.xlim(40,240);
```

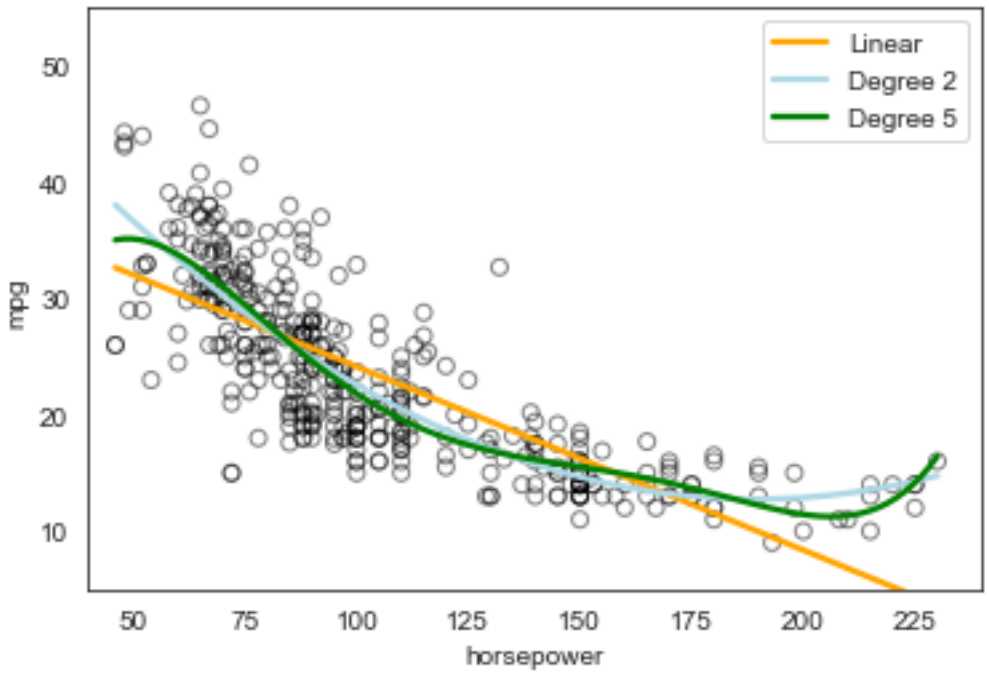


Table 3.10

```
auto['horsepower2'] = auto.horsepower**2
auto.head(3)
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle mal
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite

```
est = smf.ols('mpg ~ horsepower + horsepower2', auto).fit()
est.summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	56.9001	1.800	31.604	0.000	53.360	60.440
horsepower	-0.4662	0.031	-14.978	0.000	-0.527	-0.405
horsepower2	0.0012	0.000	10.080	0.000	0.001	0.001

12 Sample Splitting

This challenge is based on the Boston data.

- Fit "the biggest model" (multiple linear regression) that you possibly can (including interactions and categorical variables).
- Dependent Variable: `MEDV`
- Whoever achieves the highest R^2 wins :)

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

12.1 Find the best fit

Dataset Description:

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air'.

Features Description: - **CRIM**: per capita crime rate by town - **ZN**: proportion of residential land zoned for lots over 25,000 sq.ft. - **INDUS**: proportion of non-retail business acres per town - **CHAS**: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise) - **NOX**: nitric oxides concentration (parts per 10 million) - **RM**: average number of rooms per dwelling - **AGE**: proportion of owner-occupied units built prior to 1940 - **DIS**: weighted distances to five Boston employment centres - **RAD**: index of accessibility to radial highways - **TAX**: full-value property-tax rate per 10,000 dollars - **PTRATIO**: pupil-teacher ratio by town - **BLACK**: 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town - **LSTAT**: percentage lower status of the population - **MEDV**: Median value of owner-occupied homes in 1000's dollars

```
boston = pd.read_csv('./data/Boston.csv')

# independent variables
```

```
X = boston.drop('medv', axis=1)

# dependent variable
y = boston['medv']
print(boston.shape)
boston.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

```
boston.describe()
```

	crim	zn	indus	chas	nox	rm	age	dis
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.0000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.12650

```
boston.corr()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad
crim	1.000000	-0.200469	0.406583	-0.055892	0.420972	-0.219247	0.352734	-0.379670	0.6255
zn	-0.200469	1.000000	-0.533828	-0.042697	-0.516604	0.311991	-0.569537	0.664408	-0.311
indus	0.406583	-0.533828	1.000000	0.062938	0.763651	-0.391676	0.644779	-0.708027	0.5951
chas	-0.055892	-0.042697	0.062938	1.000000	0.091203	0.091251	0.086518	-0.099176	-0.007
nox	0.420972	-0.516604	0.763651	0.091203	1.000000	-0.302188	0.731470	-0.769230	0.6114
rm	-0.219247	0.311991	-0.391676	0.091251	-0.302188	1.000000	-0.240265	0.205246	-0.209
age	0.352734	-0.569537	0.644779	0.086518	0.731470	-0.240265	1.000000	-0.747881	0.4560
dis	-0.379670	0.664408	-0.708027	-0.099176	-0.769230	0.205246	-0.747881	1.000000	-0.494

	crim	zn	indus	chas	nox	rm	age	dis	rad
rad	0.625505	-0.311948	0.595129	-0.007368	0.611441	-0.209847	0.456022	-0.494588	1.0000
tax	0.582764	-0.314563	0.720760	-0.035587	0.668023	-0.292048	0.506456	-0.534432	0.9102
ptratio	0.289946	-0.391679	0.383248	-0.121515	0.188933	-0.355501	0.261515	-0.232471	0.4647
black	-0.385064	0.175520	-0.356977	0.048788	-0.380051	0.128069	-0.273534	0.291512	-0.444
lstat	0.455621	-0.412995	0.603800	-0.053929	0.590879	-0.613808	0.602339	-0.496996	0.4886
medv	-0.388305	0.360445	-0.483725	0.175260	-0.427321	0.695360	-0.376955	0.249929	-0.381

12.1.1 Additive Models first

```
#import statsmodel and sklearn Library
import sklearn.linear_model as skl_lm
import statsmodels.formula.api as smf

# Create a Baseline Multiple Linear Regression
est = smf.ols('medv ~ lstat + black + ptratio ', boston).fit()
est.summary().tables[1]

y_pred = est.predict()

sklearn.metrics.r2_score(y, y_pred)
```

```
0.6098453143448523
```

```
# Create a Multiple Linear Regression with all columns
est = smf.ols('medv ~ lstat + black + ptratio + tax + rad + dis + age + rm + nox +chas + i
est.summary().tables[1]

y_pred = est.predict()

sklearn.metrics.r2_score(y, y_pred)
```

```
0.7406426641094095
```

```
# Try out with Variables which have a higher correlation with medv (> 0.3)
est = smf.ols('medv ~ lstat + black + ptratio + tax  + rad + age + rm + nox + indus + zn +

y_pred = est.predict()
```

```
sklearn.metrics.r2_score(y, y_pred)
```

```
0.7062733492875524
```

12.1.2 Interactions

Find Variables which correlate and add their Interactions to the model

```
# add Interactions between Variables
est = smf.ols('medv ~ lstat + black + ptratio + tax + rad + dis + age + rm + nox +chas + i
y_pred = est.predict()

sklearn.metrics.r2_score(y, y_pred)
```

```
0.8276188554359157
```

Find the Categorical Variables and dummify

```
print('rad: ' , boston['rad'].nunique())
print('zn: ' ,boston['zn'].nunique())
print('indus: ' ,boston['indus'].nunique())
print('chas: ' ,boston['chas'].nunique())
print('dis: ' ,boston['dis'].nunique())
print('tax: ' ,boston['tax'].nunique())
```

```
rad:  9
zn:  26
indus:  76
chas:  2
dis:  412
tax:  66
```

```
# add Categoricals rad and chas
est = smf.ols('medv ~ lstat + black + ptratio + tax + C(rad) + dis + age + rm + nox + C(ch
y_pred = est.predict()

sklearn.metrics.r2_score(y, y_pred)
```

```
0.9247310711792476
```

most likely Overfitted Model with R^2 of 0.92

12.1.3 The perfect fit

Maximal Overfitted Model all predicting Variables as Categories + all possible Relationships

```
est = smf.ols('medv ~ C(lstat) + C(black) + C(ptratio) + C(tax) + C(rad) + C(dis) + C(age)
y_pred = est.predict()

sklearn.metrics.r2_score(y, y_pred)
```

```
1.0
```

12.1.4 Train Test Split

How do we quantify the notion of **overfitting**, i.e. the (obvious?) impression that the model is "too wiggly", or **too complex** ?

The R^2 on the data we used to fit the model is useless for this purpose because it seems to only improve the more complex the model becomes!

One useful idea seems the following: if the orange line does not really capture the "true model", i.e. has adapted too much to the noise, its performance on a test set would be worse than a simpler model.

Let us examine this idea by using a **Train Test Split**

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
#from sklearn.linear_model import LinearRegression
boston.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

- Compute the R^2 for the test portion
- Compare with the adjusted R^2.
- Think about the model complexity parameter in OLS.
- How would one choose which variables should be part of the model?

```
train, test = train_test_split(boston, test_size = 0.1, random_state=45)

y_train = train["medv"]
y_test = test["medv"]

#will be useful later
X_train = train.drop("medv", axis=1)
X_test = test.drop("medv", axis=1)
```

```
#Repeat one of the earlier amazing fits:
formula = 'medv ~ lstat + black + ptratio + tax + C(rad) + dis + age + rm + nox + C(chas)

est = smf.ols(formula, train).fit()
y_pred_train = est.predict()
y_pred_test = est.predict(X_test)

R2_train = sklearn.metrics.r2_score(y_train, y_pred_train)
R2_test = sklearn.metrics.r2_score(y_test, y_pred_test)

print("train R2:", R2_train)
print("test R2:", R2_test)

MSE_test = mean_squared_error(y_test, y_pred_test)
MSE_train = mean_squared_error(y_train, y_pred_train)

print("MSE_train:", MSE_train)
print("MSE_test:", MSE_test)
```

```
train R2: 0.9274106134480603
test R2: 0.871592468610564
MSE_train: 5.96861557537797
MSE_test: 13.241124958441453
```

12.2 Cross Validation

Drawbacks of validation set approach

- The validation estimate of the test error can be highly variable, depending on precisely which observations are included in the training/validation set.
- In the validation approach, only a subset of the observations - those that are included in the training set rather than in the validation set - are used to fit the model.
- This suggests that the validation set error may tend to **overestimate the test error** for the model fit on the entire data set.

K-fold Cross-validation

- randomly divide the data into K equal-sized parts. We leave out part k, fit the model to the other K-1 parts (combined), and then obtain predictions for the left-out kth part.
- This is done in turn for each part $k = 1, 2, \ldots, K$, and then the results are combined.

12.2.1 Design Matrix

Get to know your friendly helper library patsy

```
from patsy import dmatrices

y_0, X_wide = dmatrices(formula, boston)#, return_type='dataframe')
X_wide.shape
```

```
(506, 99)
```

12.2.2 Scoring Metrics

Model selection and evaluation using tools, such as `model_selection.GridSearchCV` and `model_selection.cross_val_score`, take a `scoring` parameter that controls what metric they apply to the estimators evaluated.

Find the options here

```
from sklearn.model_selection import cross_val_score

reg_all = LinearRegression()
#The unified scoring API always maximizes the score,
# so scores which need to be minimized are negated in order for the unified scoring API to
```

```
cv_results = cross_val_score(reg_all, X_wide, y_0, cv=10, scoring='neg_mean_squared_error'
cv_results = -np.round(cv_results,1)
print("CV results: ", cv_results)
print("CV results mean: ", np.mean(-cv_results))
```

```
CV results:  [ 24.61090588  10.30866313  62.18781304  44.2004828  107.53306211
  17.14942076  18.69173435 241.76832017 148.09771969  40.90216317]
CV results mean:  71.54502851092425
```

Comments

- For non-equal fold sizes, we need to compute the weighted mean!
- Setting $K = n$ yields n-fold or **leave-one out cross-validation** (LOOCV).
- With least-squares linear or polynomial regression, an amazing shortcut makes the cost of LOOCV the same as that of a single model fit! The following formula holds:

$$CV_n = \frac{1}{n} \sum \left(\frac{y_i - \hat{y}_i}{1 - h_i} \right)^2$$

where $\hat{y}_i$ is the ith fitted value from the original least squares fit, and h_i is the leverage (see ISLR book for details.) This is like the ordinary MSE, except the ith residual is divided by $1 - h_i$.

12.2.3 Task

- Sketch the code for your own CV function.
- Reproduce the left panel of Fig. 5.6, i.e. the right panel of Fig 2.9 from the ISLR book

13 Classification

```
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
import seaborn as sns

from sklearn.preprocessing import scale
import sklearn.linear_model as skl_lm
from sklearn.metrics import mean_squared_error, r2_score
import statsmodels.api as sm
import statsmodels.formula.api as smf

from scipy import stats
#in response to: module 'scipy.stats' has no attribute 'chisqprob'
#stats.chisqprob = lambda chisq, df: stats.chi2.sf(chisq, df)
#this one I inserted just before class, sorry !!
from sklearn.metrics import confusion_matrix, roc_curve, roc_auc_score #, log_loss

from scipy import special
%matplotlib inline
sns.set_style('white')

%precision 3
```

```
'%.3f'
```

Load data

```
#1. default data from ISLR

# In R, we exported the dataset from package 'ISLR' to an Excel file
df = pd.read_csv('./data/Default.csv',index_col=0)
#2. Titanic
```

```
titanic = sns.load_dataset('titanic')
df.head(3)
```

	default	student	balance	income
1	No	No	729.526495	44361.62507
2	No	Yes	817.180407	12106.13470
3	No	No	1073.549164	31767.13895

```
# Note: factorize() returns two objects: a label array and an array with the unique values
df.default.factorize()
```

```
(array([0, 0, 0, ..., 0, 0, 0]), Index(['No', 'Yes'], dtype='object'))
```

```
# We are only interested in the first object.
df['default2'] = df.default.factorize()[0]
df['student2'] = df.student.factorize()[0]
```

```
#are the data balanced
#very unbalanced data make modeling VERY tough !!
df['default2'].value_counts()
#np.mean(df['default2'])
```

```
0    9667
1     333
Name: default2, dtype: int64
```

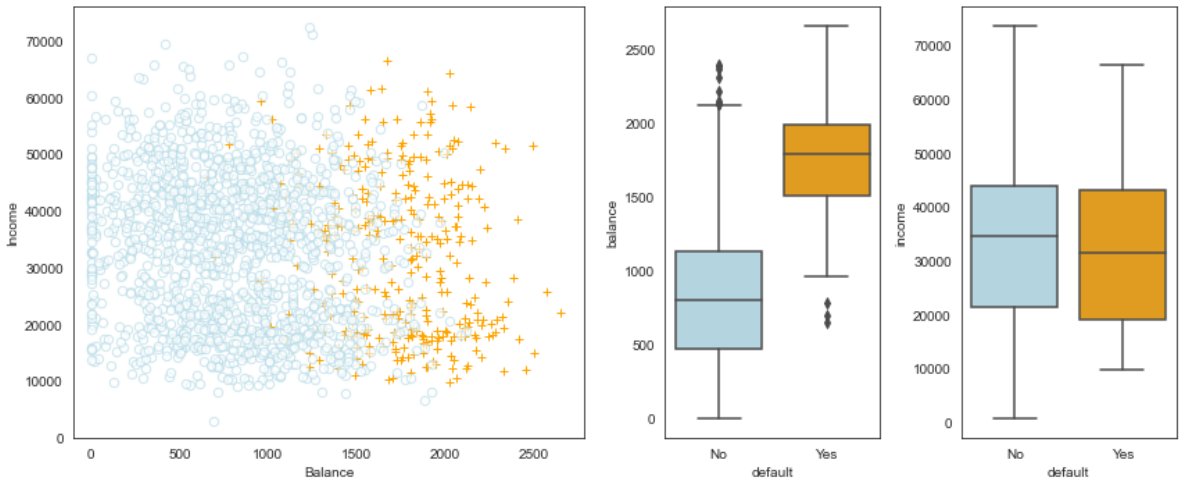
13.0.1 Data Exploration

Figure 4.1 (ISLR) - Default data set

```
fig = plt.figure(figsize=(12,5))
gs = mpl.gridspec.GridSpec(1, 4)
ax1 = plt.subplot(gs[0,:-2])
ax2 = plt.subplot(gs[0,-2])
ax3 = plt.subplot(gs[0,-1])

# Take a fraction of the samples where target value (default) is 'no'
```

```
df_no = df[df.default2 == 0].sample(frac=0.15)
# Take all samples  where target value is 'yes'
df_yes = df[df.default2 == 1]
df_ = df_no.append(df_yes)

ax1.scatter(x=df_[df_.default == 'Yes'].balance, y=df_[df_.default == 'Yes'].income, s=40,
            linewidths=1)
ax1.scatter(x=df_[df_.default == 'No'].balance, y=df_[df_.default == 'No'].income, s=40, m
            edgecolors='lightblue', facecolors='white', alpha=.6)

ax1.set_ylim(ymin=0)
ax1.set_ylabel('Income')
ax1.set_xlim(xmin=-100)
ax1.set_xlabel('Balance')

c_palette = {'No':'lightblue', 'Yes':'orange'}
sns.boxplot(x='default', y='balance', data=df, orient='v', ax=ax2, palette=c_palette)
sns.boxplot(x='default', y='income', data=df, orient='v', ax=ax3, palette=c_palette)
gs.tight_layout(plt.gcf())
```

```
/var/folders/h4/k73g68ds6xj791sf8cpmlxlc0000gn/T/ipykernel_61849/1350372965.py:11: FutureWarn
  df_ = df_no.append(df_yes)
```



13.1 Logistic Regression

Recall our fit to the Titanic data from last week and the dilemma that some predictions and interpretations (such as the intercept) often led to survival probabilities outside the range $[0, 1]$.

```
est = smf.ols('survived ~ age + C(pclass) + C(sex)', titanic).fit()
print(est.summary().tables[1])
```

```
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.1250      0.051     22.202      0.000       1.026       1.225
C(pclass)[T.2] -0.2077      0.042     -4.983      0.000      -0.290      -0.126
C(pclass)[T.3] -0.4066      0.038    -10.620      0.000      -0.482      -0.331
C(sex)[T.male] -0.4795      0.031    -15.608      0.000      -0.540      -0.419
age            -0.0055      0.001     -5.039      0.000      -0.008      -0.003
==============================================================================
```

```
est = smf.logit('survived ~ age + C(pclass) + C(sex)', data=titanic)
print(est.fit().summary())
```

```
Optimization terminated successfully.
         Current function value: 0.453279
         Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:               survived   No. Observations:                  714
Model:                          Logit   Df Residuals:                      709
Method:                           MLE   Df Model:                            4
Date:                Sun, 30 May 2021   Pseudo R-squ.:                  0.3289
Time:                        20:26:47   Log-Likelihood:                -323.64
converged:                       True   LL-Null:                       -482.26
Covariance Type:            nonrobust   LLR p-value:                 2.074e-67
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      3.7770      0.401      9.416      0.000       2.991       4.563
C(pclass)[T.2] -1.3098      0.278     -4.710      0.000      -1.855      -0.765
C(pclass)[T.3] -2.5806      0.281     -9.169      0.000      -3.132      -2.029
```

```
C(sex)[T.male]    -2.5228      0.207    -12.164      0.000      -2.929      -2.116
age               -0.0370      0.008     -4.831      0.000      -0.052      -0.022
==============================================================================
```

This is not the only shortcoming of **linear** regression (LR) for binary outcomes! Other problems include heteroskedasticity and incorrect scaling of probabilities even inside the range $[0, 1]$.

One solution is to transform the linear output of the (LR) to an S-shape via the **sigmoidal** function $s(z) = 1/(1+exp(-z))$, which is the strategy taken by **logistic regression** (example: Figure 4.2 from the ISLR book):

```
#passenger Joe, aged 25, Pclass3:
oddsJoe = np.exp(3.777 -2.5806  -2.5228  -0.0370*25)
print("The odds of survival for Joe are", str(oddsJoe))
survProbJoe = oddsJoe/(1+oddsJoe)
survProbJoe
print("The probability of survival for Joe are", str(survProbJoe))
```

```
The odds of survival for Joe are 0.1052517688905321
The probability of survival for Joe are 0.09522877217033127
```

```
from sklearn.metrics import classification_report
y_true = y
#The 50% cutoff/threshold is chosen by the user !!
y_pred = prob_train[:,0] > 0.99
#y_pred

print(classification_report(y_true, y_pred))#, target_names=['class 0', 'class 1']))
```

```
              precision    recall  f1-score   support

           0       0.16      0.00      0.00      9667
           1       0.03      0.92      0.06       333

    accuracy                           0.03     10000
   macro avg       0.10      0.46      0.03     10000
weighted avg       0.16      0.03      0.00     10000
```

```
pd.crosstab(y_true, y_pred)
```

col_0 default2	False	True
0	42	9625
1	100	233

```
#Based on the confusion matrix:
#for "1=positive class" we get
#precision TP/(TP+FP):
print("precision TP/(TP+FP)", str(np.round(233/(9625+233),3)))
#precision TP/(TP+FN):
print("recall TP/(TP+FN)", str(np.round(233/(100+233),3)))

#for "0=positive class" we get
#precision TP/(TP+FP):
print("precision TP/(TP+FP)", str(np.round(42/(100+42),3)))
#precision TP/(TP+FN):
print("recall TP/(TP+FN)", str(np.round(42/(9625+42),3)))
```

```
precision TP/(TP+FP) 0.024
recall TP/(TP+FN) 0.7
precision TP/(TP+FP) 0.296
recall TP/(TP+FN) 0.004
```

```
est = smf.logit('default2 ~ balance', data=df)
print(est.fit().summary())
```

```
Optimization terminated successfully.
         Current function value: 0.079823
         Iterations 10
                           Logit Regression Results
==============================================================================
Dep. Variable:                default2   No. Observations:                10000
Model:                           Logit   Df Residuals:                     9998
Method:                            MLE   Df Model:                            1
Date:                 Sun, 30 May 2021   Pseudo R-squ.:                  0.4534
Time:                         20:26:48   Log-Likelihood:                -798.23
```

```
converged:                       True   LL-Null:                       -1460.3
Covariance Type:            nonrobust   LLR p-value:                6.233e-290
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept    -10.6513      0.361    -29.491      0.000     -11.359      -9.943
balance        0.0055      0.000     24.952      0.000       0.005       0.006
==============================================================================

Possibly complete quasi-separation: A fraction 0.13 of observations can be
perfectly predicted. This might indicate that there is complete
quasi-separation. In this case some parameters will not be identified.
```

```
#Mini Tasks: fit a logistic regression to the Titanic data
#Try to make sense of the coefficients!
est = smf.logit('survived ~  C(pclass) + C(sex)', data=titanic)
print(est.fit().summary().tables[1])
```

```
Optimization terminated successfully.
         Current function value: 0.464023
         Iterations 6
================================================================================
                   coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept          2.2971      0.219     10.490      0.000       1.868       2.726
C(pclass)[T.2]    -0.8380      0.245     -3.424      0.001      -1.318      -0.358
C(pclass)[T.3]    -1.9055      0.214     -8.898      0.000      -2.325      -1.486
C(sex)[T.male]    -2.6419      0.184    -14.350      0.000      -3.003      -2.281
================================================================================
```

```
pHat = est.fit().predict()
pred = pHat > 0.5
pd.crosstab(pred, titanic.survived)
```

```
Optimization terminated successfully.
         Current function value: 0.464023
         Iterations 6
```

survived	0	1
row_0		
False	468	109
True	81	233

13.1.1 Coefficients as Odds

For "normal regression" we know that the value of β_j simply gives us Δy if x_j is increased by one unit.

In order to fully understand the exact meaning of the coefficients for a LR model we need to first warm up to the definition of a **link function** and the concept of **probability odds**.

Using linear regression as a starting point

$$y_i = \beta_0 + \beta_1 x_{1,i} + \beta_2 x_{2,i} + ... + \beta_k x_{k,i} + \epsilon_i$$

we modify the right hand side such that (i) the model is still basically a linear combination of the x_js but (ii) the output is -like a probability- bounded between 0 and 1. This is achieved by "wrapping" a sigmoid function $s(z) = 1/(1 + exp(-z))$ around the weighted sum of the x_js:

$$y_i = s(\beta_0 + \beta_1 x_{1,i} + \beta_2 x_{2,i} + ... + \beta_k x_{k,i} + \epsilon_i)$$

The sigmoid function, depicted below to the left, transforms the real axis to the interval $(0; 1)$ and can be interpreted as a probability.

The inverse of the sigmoid is the *logit* (depicted above to the right), which is defined as $log(p/(1-p))$. For the case where p is a probability we call the ratio $p/(1-p)$ the **probability odds**. Thus, the logit is the log of the odds and logistic regression models these *log-odds* as a linear combination of the values of x.

Finally, we can interpret the coefficients directly: the odds of a positive outcome are multiplied by a factor of $exp(\beta_j)$ for every unit change in x_j. (In that light, logistic regression is reminiscient of linear regression with logarithmically transformed dependent variable which also leads to multiplicative rather than additive effects.)

Summary

$$p(x) = \frac{e^{\beta_0 + \beta_1 x_1 + ... + \beta_k x_k}}{1 + e^{\beta_0 + \beta_1 x_1 + ... + \beta_k x_k}}$$

Odds

$$\frac{p(x)}{1-p(x)} = e^{\beta_0+\beta_1 x_1+\ldots+\beta_k x_k}$$

This post has a more detailed view on the interpretations of the coefficients:

https://blog.hwr-berlin.de/codeandstats/interpretation-of-the-coefficients-in-logistic-regression/

13.1.1.1 Comments

1. When your data are **linearly separable** there is (ironically) a fitting problem ! See iris example below
2. *Logistic regression preserves the marginal probabilities.* The sum of the predicted probability scores for any subgroup of the training data (which includes all of it) will be equal to the number of positives.
3. *What is deviance ?* Deviance (also referred to as *log loss*) is a measure of how well the model fits the data. It is 2 times the negative log likelihood of the dataset, given the model.

$$dev = -\sum_i y_i \cdot \log p_i + (1-y_i) \cdot \log(1-p_i)$$

In Python, you can use the log_loss function from scikit-learn, with documentation found here. If you think of deviance as analogous to variance, then the null deviance is similar to the variance of the data around the average rate of positive examples. The residual deviance is similar to the variance of the data around the model. As an exercise we will calculate the deviances in a homework.

4. **Pseudo** R^2 McFadden's R^2 is defined as $1 - LL_{mod}/LL_0$, where LL_{mod} is the log likelihood value for the fitted model and LL_0 is the log likelihood for the null model which includes only an intercept as predictor (so that every individual is predicted the same probability of 'success').
    - For a logistic regression model the log likelihood value is always negative (because the likelihood contribution from each observation is a probability between 0 and 1). If your model doesn't really predict the outcome better than the null model, LL_{mod} will not be much larger than LL_0 , and so $LL_{mod}/LL_0 \sim 1$, and McFadden's pseudo-R2 is close to 0 (your model has no predictive value).

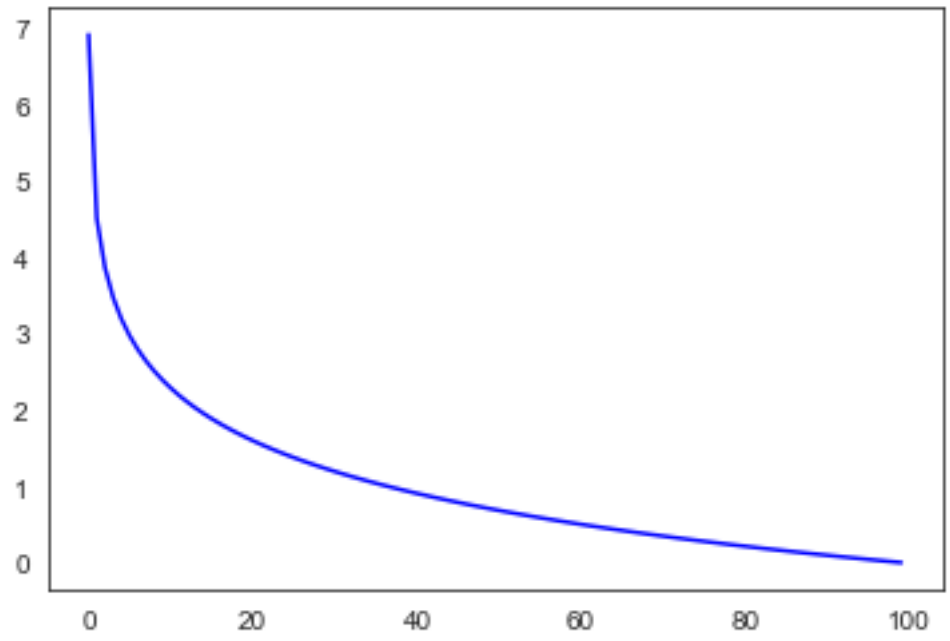
- Conversely if your model was really good, those individuals with a success (1) outcome would have a fitted probability close to 1, and vice versa for those with a failure (0) outcome. In this case if you go through the likelihood calculation the likelihood contribution from each individual for your model will be close to zero, such that LL_{mod} is close to zero, and McFadden's pseudo-R2 squared is close to 1, indicating very good predictive ability.

13.2 ROC curves

```
fpr, tpr, thresholds = metrics.roc_curve(y, scores)
roc_display = RocCurveDisplay(fpr=fpr, tpr=tpr).plot()
#or

from sklearn.metrics import RocCurveDisplay

RocCurveDisplay.from_predictions(y_test, y_pred)
plt.show()
```

```
def logloss(true_label, predicted, eps=1e-15):
    p = np.clip(predicted, eps, 1 - eps)
    if true_label == 1:
        return -np.log(p)
    else:
        return -np.log(1 - p)
```

```
p= np.linspace(0.001,1,100)#.reshape(-1,1)
plt.plot(logloss(1,p), "b-");
```



13.2.1 Think Stats Data

The NSFG dataset includes 244 variables about each pregnancy and another 3087 variables about each respondent. Maybe some of those variables have predictive power. To find out which ones are most useful, why not try them all? Testing the variables in the pregnancy table is easy, but in order to use the variables in the respondent table, we have to match up each pregnancy with a respondent. In theory we could iterate through the rows of the pregnancy table, use the caseid to find the corresponding respondent, and copy the values from the correspondent table into the pregnancy table. But that would be slow.

A better option is to recognize this process as a join operation as defined in SQL and other relational database languages (see). Join is implemented as a DataFrame method, so we can perform the operation like this:

do not run this cell, for completeness, I show how the joined dataframe was created:

```
from __future__ import print_function, division
import numpy as np
import nsfg

preg = pd.read_hdf('../data/pregNSFG.h5', 'df')
#only look at live births
live = preg[preg.outcome == 1]
```

```
live = live[live.prglngth>30]
resp = nsfg.ReadFemResp()
resp.index = resp.caseid
join = live.join(resp, on='caseid', rsuffix='_r')
#save to native python format:
#http://pandas.pydata.org/pandas-docs/version/0.23/generated/pandas.DataFrame.to_hdf.html
join.to_hdf('JoinedpregNSFG.h5', key='df', format='table',complevel =9)
```

```
live = pd.read_hdf('../data/JoinedpregNSFG.h5','df')
live.head()
```

```
#define first babies
firsts = live[live.birthord == 1]
#and all others:
others = live[live.birthord != 1]
```

```
# from this discussion, it seems that statsmodels still uses the defunct
# chisqprob, so we have to define it ourselves:
# https://github.com/statsmodels/statsmodels/issues/3931
from scipy import stats
stats.chisqprob = lambda chisq, df: stats.chi2.sf(chisq, df)
stats.chisqprob(10,3)
```

The mother's age seems to have a small, non significant effect.

```
live['boy'] = (live.babysex==1).astype(int)
SexvsAge = smf.logit('boy ~ agepreg', data=live)
results = SexvsAge.fit()
print(results.summary())
```

```
live["fmarout5"].value_counts()
```

13.2.2 The Trivers-Willard hypothesis

Exercise 11.2 The Trivers-Willard hypothesis suggests that for many mammals the sex ratio depends on "maternal condition"; that is, factors like the mother's age, size, health, and social status. See. Some studies have shown this effect among humans, but results are mixed. As an exercise, use a data mining approach to test the other variables in the pregnancy and respondent files.

In the solution for exercise 11.2 the author uses a data mining approach to find the "best" model:

(Task: can we find out the meaning of the 2 new variables??)

```
formula='boy ~ agepreg + fmarout5==5 + infever==1'
model = smf.logit(formula, data=live)
results = model.fit()
print(results.summary())
```

13.2.3 Tasks

1. Compute the ROC curve and AUC for the NSFG data
2. Use cross validation to estimate some accuracy measure of classification for the
   - Titanic survival
   - sex prediction for the NSFG data
3. Translate the coefficient for Pclass 3 into both odds and probability of survival (compared to the reference level Pclass 1).
4. Compute the survival probability of the first passenger in the data set.

```
import patsy

y, X = patsy.dmatrices('survived ~ age + C(pclass) + C(sex)', titanic)
#y = titanic["survived"]
```

```
from sklearn.model_selection import cross_val_score
clf = skl_lm.LogisticRegression() # solver='newton-cg')

cv_results = cross_val_score(clf, X, np.ravel(y), cv=10)
print(np.round(cv_results,2))
```

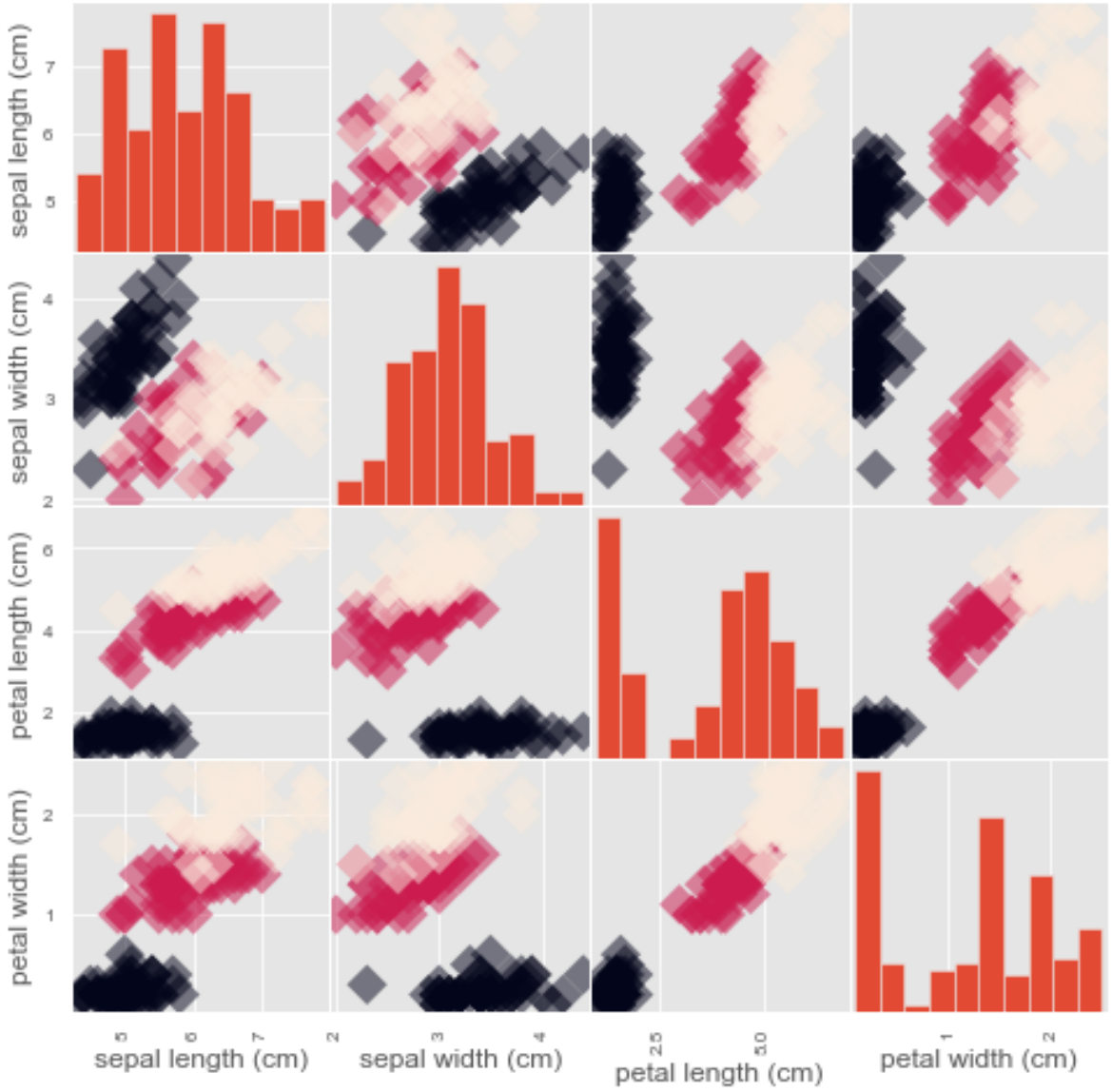
```
[0.78 0.74 0.81 0.88 0.75 0.79 0.79 0.73 0.82 0.82]
```

13.2.4 The Iris dataset

```
from sklearn import datasets

plt.style.use('ggplot')
iris = datasets.load_iris()
```

```
print(iris.data.shape)

X = iris.data
y = iris.target
df = pd.DataFrame(X, columns=iris.feature_names)
_ = pd.plotting.scatter_matrix(df, c = y, figsize = [8, 8],s=150, marker = 'D');
```

```
(150, 4)
```



Looks like we could build a perfect classifier with just *petal width* ?

```
  iris.feature_names
```

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

```
  df["setosa"]= (y==0)
  df.head()
  df.columns
```

```
Index(['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
       'petal width (cm)', 'setosa'],
      dtype='object')
```

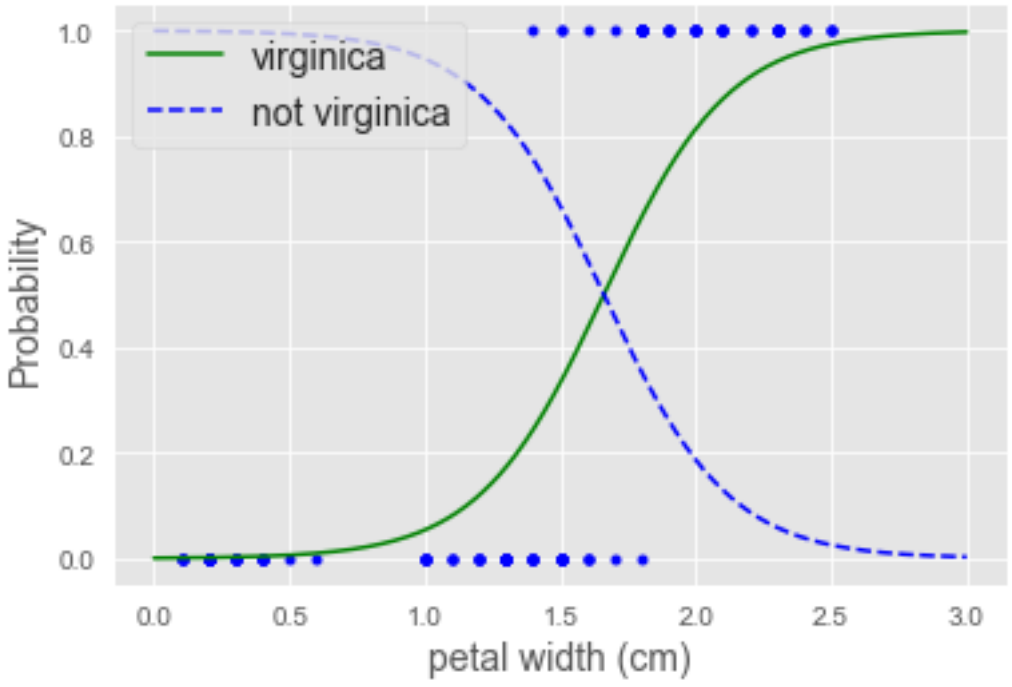
```
  X = iris["data"][:,3:]  # petal width

  logit = sm.Logit((iris["target"]==0).astype(np.int), X)
  logit.fit().params
```

```
Optimization terminated successfully.
         Current function value: 0.354801
         Iterations 7
```

```
array([-1.847])
```

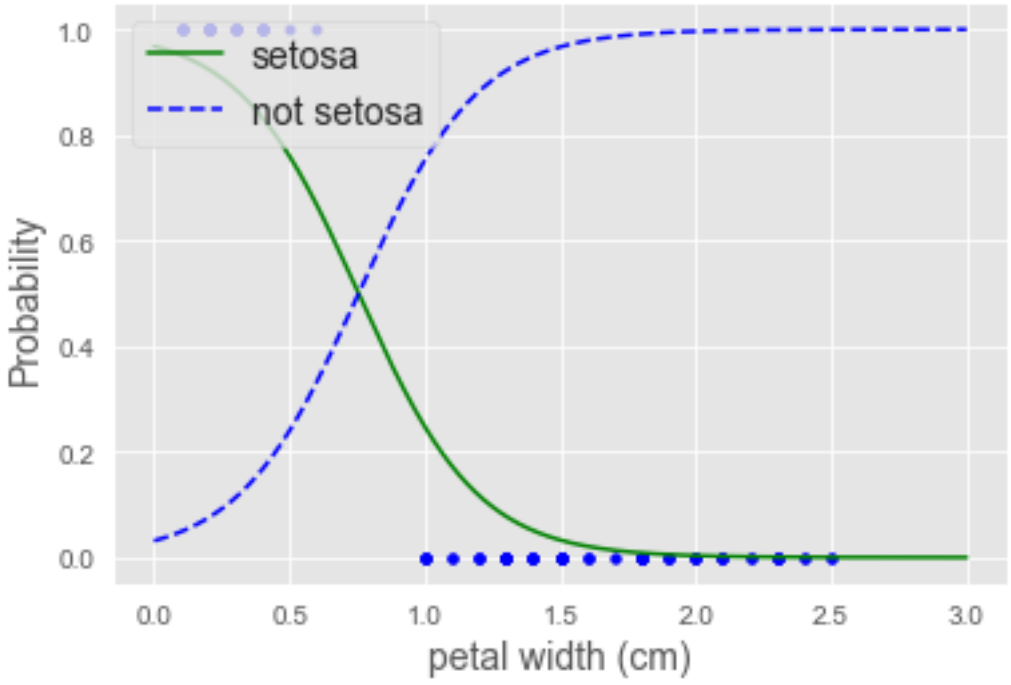
```
  #sklearn
  from sklearn.linear_model import LogisticRegression

  def LogReg(xCol=3, target=2,penalty="l2"):
      X = iris["data"][:,xCol:]  # petal width
      y = (iris["target"]==target).astype(np.int)

      log_reg = LogisticRegression(penalty=penalty)
      log_reg.fit(X,y)
```

```
    X_new = np.linspace(0,3,1000).reshape(-1,1)
    y_proba = log_reg.predict_proba(X_new)

    flowerType=["setosa", "versicolor", "virginica"]

    plt.plot(X,y,"b.")
    plt.plot(X_new,y_proba[:,1],"g-",label=flowerType[target])
    plt.plot(X_new,y_proba[:,0],"b--",label="not " + flowerType[target])
    plt.xlabel(iris.feature_names[xCol], fontsize=14)
    plt.ylabel("Probability", fontsize=14)
    plt.legend(loc="upper left", fontsize=14)
    plt.show()

    return log_reg

log_reg = LogReg()

log_reg.predict([[1.7],[1.5]])
```




```
array([1, 0])
```

```
log_reg = LogReg(target=0)
```



13.3 Other Classifiers

13.3.1 K Nearest Neighbors

```
from sklearn.neighbors import KNeighborsClassifier
iris = datasets.load_iris()
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(iris['data'], iris['target'])
```

```
KNeighborsClassifier(n_neighbors=6)
```

Task: Split the iris data into training and test. Predict on test

```
#prediction = knn.predict(X_test)
#print('Prediction {}'.format(prediction))
```

13.3.2 Multinomial Logistic Regression

```
import statsmodels.api as st
#different way of importing data
iris = st.datasets.get_rdataset('iris', 'datasets')

y = iris.data.Species

y.head(3)

x = iris.data.iloc[:, 0]

x = st.add_constant(x, prepend = False)

x.head()
```

	Sepal.Length	const
0	5.1	1.0
1	4.9	1.0
2	4.7	1.0
3	4.6	1.0
4	5.0	1.0

```
mdl = st.MNLogit(y, x)

mdl_fit = mdl.fit()

print(mdl_fit.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.606893
         Iterations 8
                          MNLogit Regression Results
==============================================================================
Dep. Variable:                Species   No. Observations:                  150
```

```
Model:                          MNLogit   Df Residuals:                      146
Method:                             MLE   Df Model:                            2
Date:                  Sun, 30 May 2021   Pseudo R-squ.:                  0.4476
Time:                          20:27:58   Log-Likelihood:                -91.034
converged:                         True   LL-Null:                       -164.79
Covariance Type:             nonrobust   LLR p-value:                 9.276e-33
==============================================================================
Species=versicolor       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Sepal.Length           4.8157      0.907      5.310      0.000       3.038       6.593
const                -26.0819      4.889     -5.335      0.000     -35.665     -16.499
------------------------------------------------------------------------------
Species=virginica       coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Sepal.Length           6.8464      1.022      6.698      0.000       4.843       8.850
const                -38.7590      5.691     -6.811      0.000     -49.913     -27.605
==============================================================================
```

```
### marginal effects ###

mdl_margeff = mdl_fit.get_margeff()

print(mdl_margeff.summary())
```

```
       MNLogit Marginal Effects
=====================================
Dep. Variable:                Species
Method:                          dydx
At:                           overall
==============================================================================
    Species=setosa      dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Sepal.Length          -0.3785      0.003   -116.793      0.000      -0.385      -0.372
------------------------------------------------------------------------------
Species=versicolor      dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Sepal.Length           0.0611      0.022      2.778      0.005       0.018       0.104
------------------------------------------------------------------------------
Species=virginica       dy/dx    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Sepal.Length           0.3173      0.022     14.444      0.000       0.274       0.360
==============================================================================
```

14 Advanced Topics

```
%precision 3

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

from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn import datasets
```

```
'%.3f'
```

14.0.1 K-Nearest Neighbors (KNN)

```
iris = datasets.load_iris()
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(iris['data'], iris['target'])
```

```
KNeighborsClassifier(n_neighbors=6)
```

Task: Split the iris data into training and test. Predict on test

```
# Split into training and test set
X = iris['data']
y = iris['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=42

# Create a k-NN classifier with 7 neighbors: knn
knn = KNeighborsClassifier(n_neighbors=6)
```

```
# Fit the classifier to the training data
knn.fit(X_train, y_train)

# Print the accuracy
print(knn.score(X_test, y_test))
```

```
1.0
```

Simple Idea, no modeling assumptions at all !! Think about the following:

- What is "the model", i.e. what needs to be stored ? (coefficients, functions, ...)
- What is the model complexity ?
- Does this only work for classification ? What would be the regression analogy ?
- What improvements could we make to the simple idea ?
- In the modeling world:
  - linear ?
  - local vs. global
  - memory/CPU requirements
  - wide versus tall data ?

14.0.1.1 Handwritten Digits

```
digits = datasets.load_digits()
import matplotlib.pyplot as plt

#how can I improve the plots ? (i..e no margins, box around plot...)
plt.figure(1)

for i in np.arange(10)+1:
    plt.subplot(2, 5, i)
    plt.axis('off')
    #plt.gray()
    #plt.matshow(digits.images[i-1])
    plt.imshow(digits.images[i-1], cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```

```
# Create feature and target arrays
X = digits.data
y = digits.target

# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42

# Create a k-NN classifier with 7 neighbors: knn
knn = KNeighborsClassifier(n_neighbors=7)

# Fit the classifier to the training data
knn.fit(X_train, y_train)

# Print the accuracy
print(knn.score(X_test, y_test))
```

```
0.9833333333333333
```

```
# Big Confusion Matrix
preds = knn.predict(X_test)

pd.crosstab(preds,y_test)
```

col_0 row_0	0	1	2	3	4	5	6	7	8	9
0	36	0	0	0	0	0	0	0	0	0
1	0	36	0	0	0	0	0	0	2	0
2	0	0	35	0	0	0	0	0	0	0
3	0	0	0	37	0	0	0	0	0	0
4	0	0	0	0	36	0	0	0	0	1
5	0	0	0	0	0	37	0	0	0	0
6	0	0	0	0	0	0	35	0	0	0
7	0	0	0	0	0	0	0	36	1	0
8	0	0	0	0	0	0	1	0	32	1
9	0	0	0	0	0	0	0	0	0	34

Task

Construct a model complexity curve for the digits dataset! In this exercise, you will compute and plot the training and testing accuracy scores for a variety of different neighbor values. By observing how the accuracy scores differ for the training and testing sets with different values of k, you will develop your intuition for overfitting and underfitting.

```
# # Setup arrays to store train and test accuracies
# neighbors = np.arange(1, 20)
# train_accuracy = np.empty(len(neighbors))
# test_accuracy = np.empty(len(neighbors))
#
# # Loop over different values of k
# for i, k in enumerate(neighbors):
#     # Setup a k-NN Classifier with k neighbors: knn
#     knn = KNeighborsClassifier(___)
#
#     # Fit the classifier to the training data
#     ___
#
#     #Compute accuracy on the training set
#     train_accuracy[i] = ___
#
#     #Compute accuracy on the testing set
#     test_accuracy[i] = ___
#
# # Generate plot
# plt.title('k-NN: Varying Number of Neighbors')
```

```
# plt.plot(___, label = 'Testing Accuracy')
# plt.plot(___, label = 'Training Accuracy')
# plt.legend()
# plt.xlabel('Number of Neighbors')
# plt.ylabel('Accuracy')
# plt.show()
```

14.0.2 Multinomial Logistic Regression

```
from sklearn.linear_model import LogisticRegression
iris = datasets.load_iris()

log_reg = LogisticRegression(multi_class='multinomial',solver='sag', max_iter=100, random_
log_reg.fit(iris["data"][:,3:],iris["target"])

preds = log_reg.predict_proba(iris["data"][:,3:])

preds[1:5,:]
```

```
array([[9.259e-01, 7.396e-02, 1.788e-04],
       [9.259e-01, 7.396e-02, 1.788e-04],
       [9.259e-01, 7.396e-02, 1.788e-04],
       [9.259e-01, 7.396e-02, 1.788e-04]])
```

Task: Compute a confusion matrix

Further Reading and Explorations: * Read about "one versus all" pitted against multinomial. Check out this notebook. * Read about marginal probabilities.

14.0.3 ROC Curves

Simplest to go back to 2 labels from lesson 7:

```
iris = datasets.load_iris()
X = iris["data"][:,3:]
y = (iris["target"]==2).astype(int)
# Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=42
```

```
log_reg = LogisticRegression()
log_reg.fit(X_train,y_train)

y_pred = log_reg.predict_proba(X_test)
y_pred_prob = y_pred[:,1]

from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

print(confusion_matrix(y_test, y_pred_prob > 0.25 ))

print(confusion_matrix(y_test, y_pred_prob > 0.5 ))

print(confusion_matrix(y_test, y_pred_prob > 0.75 ))
```

```
[[24  8]
 [ 0 13]]
[[32  0]
 [ 0 13]]
[[32  0]
 [ 4  9]]
```

The need for more sophisticated metrics than accuracy and single thresholding

This is particularly relevant for imbalanced classes, example: Emails

- Spam classification
  - 99% of emails are real; 1% of emails are spam
- Could build a classifier that predicts ALL emails as real
  - 99% accurate!
  - But horrible at actually classifying spam
  - Fails at its original purpose

Metrics from CM

- Precision: $\frac{TP}{TP+FP}$
- Recall: $\frac{TP}{TP+FN}$
- F1 score: $2 \cdot \frac{precision \cdot recall}{precision+recall}$

The F1 score is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. () - High precision: Not many real emails predicted as spam - High recall: Predicted most spam emails correctly

```
#Example:
```

```
print(classification_report(y_test, y_pred_prob > 0.5 ))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        32
           1       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

But we still need to fix a threshold for any of the metrics above to work !

Wouldn't it be best to communicate the prediction quality over a wide range (all!) of thresholds and enable the user to choose the most suitable one for his/her application ! That is the idea of the **Receiver Operating Characteristic (ROC)** curve!

```
#The pima-indians- diabetes data:
#  https://www.kaggle.com/uciml/pima-indians-diabetes-database#diabetes.csv

col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age', 'la
# load dataset
pima = pd.read_csv("../data/diabetes.csv", header=None, names=col_names,skiprows=[0])
pima.head()
```

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

```
pima.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   pregnant  768 non-null    int64
 1   glucose   768 non-null    int64
 2   bp        768 non-null    int64
 3   skin      768 non-null    int64
 4   insulin   768 non-null    int64
 5   bmi       768 non-null    float64
 6   pedigree  768 non-null    float64
 7   age       768 non-null    int64
 8   label     768 non-null    int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

```
a = [1,2,3]
b = [*a]
b
```

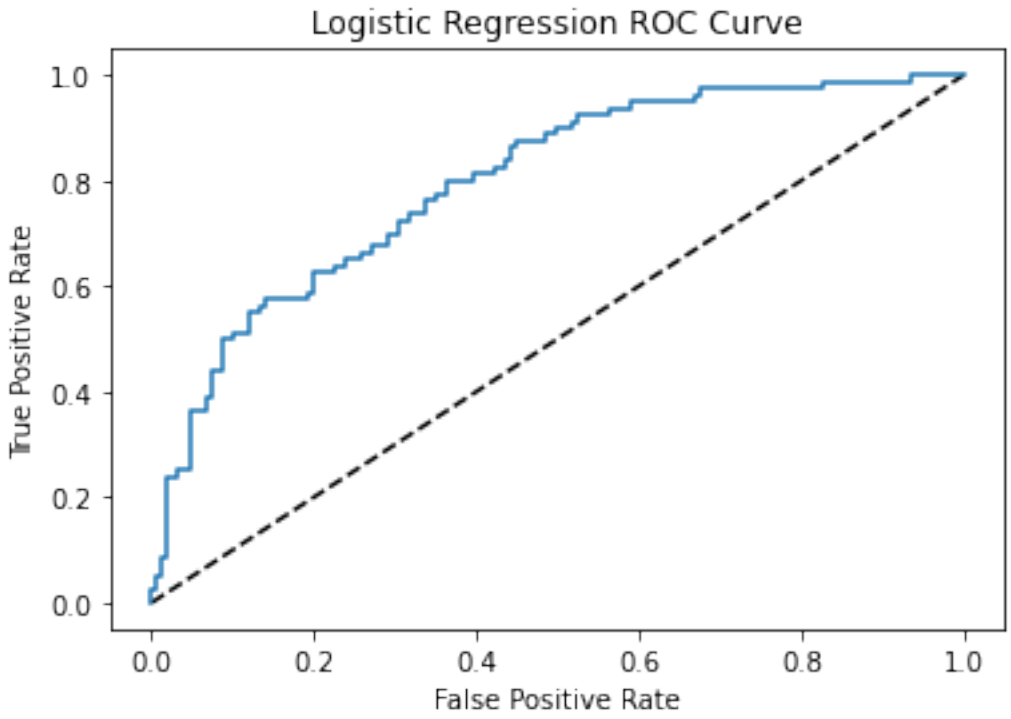
```
[1, 2, 3]
```

```
#split dataset in features and target variable
feature_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']
X = pima[feature_cols] # Features
y = pima.label # Target variable

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=42
```

```
log_reg = LogisticRegression(max_iter=200)
log_reg.fit(X_train,y_train)

y_pred = log_reg.predict_proba(X_test)
y_pred_prob = y_pred[:,1]
```

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
plt.plot([0, 1], [0, 1], 'k--')
```

```
plt.plot(fpr, tpr, label='Logistic Regression')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Logistic Regression ROC Curve')
plt.show();
```



Area under the ROC curve (AUC)

```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_test, y_pred_prob)
```

```
0.7969370860927152
```

AUC using cross-validation

```
from sklearn.model_selection import cross_val_score

cv_scores = cross_val_score(log_reg, X, y, cv=10, scoring='roc_auc')
```

```
print(cv_scores)
```

```
[0.789 0.83  0.838 0.773 0.812 0.848 0.819 0.921 0.857 0.833]
```

14.0.4 Regularized Regression

We will illustrate the concepts on the Boston housing data set

```
boston = pd.read_csv('../data/boston.csv')
#print(boston.head())
X = boston.drop('medv', axis=1).values
y = boston['medv'].values
```

Variable Selection

ISLR slides on model selection

14.0.4.1 L2 Regression

Recall: Linear regression minimizes a loss function - It chooses a coefficient for each feature variable - Large coefficients can lead to overfitting - Penalizing large coefficients: Regularization

Detour: L_p norms

http://mathworld.wolfram.com/VectorNorm.html

Our new penalty term in finding the coefficients β_j is the minimization

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}\beta_j^2=RSS+\lambda\sum_{j=1}^{p}\beta_j^2$$

Instead of obtaining **one** set of coefficients we have a dependency of β_j on λ:

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size = 0.3, random_state=42)

ridge = Ridge(alpha=0.1, normalize=True)
```

```
ridge.fit(X_train, y_train)
ridge_pred = ridge.predict(X_test)#?does this automatically take care of nromalization ??
#Returns the coefficient of determination R^2 of the prediction.
ridge.score(X_test, y_test)
ridge_pred[0:5]
```

```
array([27.97 , 35.383, 16.826, 25.145, 18.749])
```

```
ridge2 = Ridge(alpha=0.1, normalize=False)
ridge2.fit(X_train, y_train)
ridge2_pred = ridge2.predict(X_test)
ridge2_pred[0:5]
ridge2.coef_
```

```
array([-1.325e-01,  3.599e-02,  4.385e-02,  3.096e+00, -1.406e+01,
        4.061e+00, -1.200e-02, -1.365e+00,  2.395e-01, -8.817e-03,
       -8.955e-01,  1.183e-02, -5.498e-01])
```

14.0.4.2 L1 Regression

Our penalty termy looks slightly different (with big consequences for **sparsity**)

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2+\lambda\sum_{j=1}^{p}|\beta_j|=RSS+\lambda\sum_{j=1}^{p}|\beta_j|$$

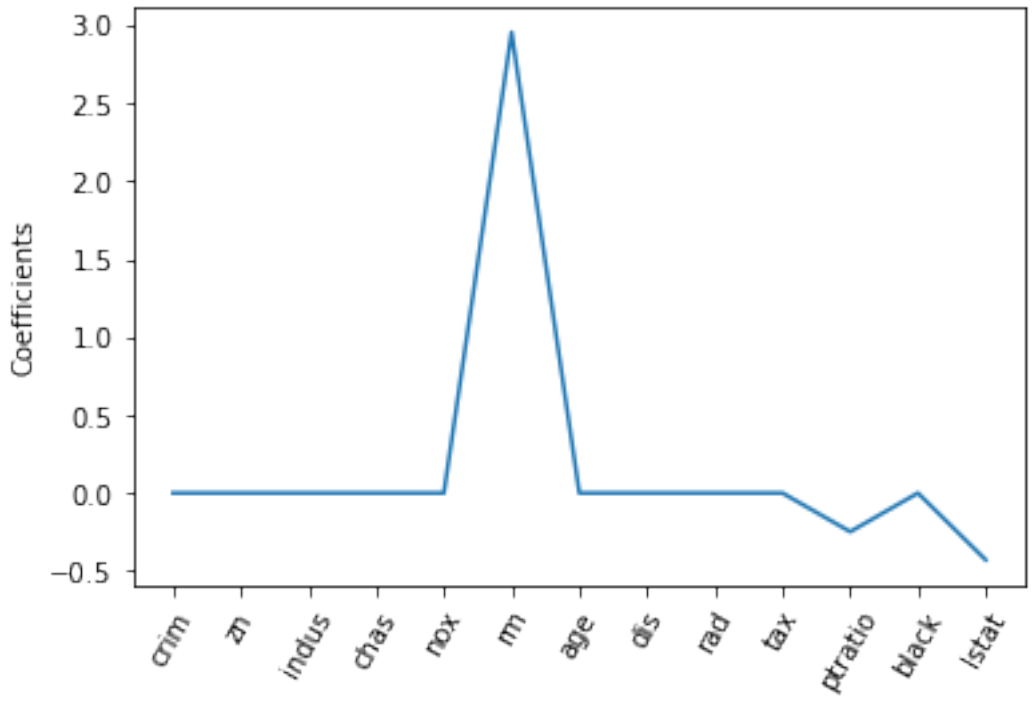
(Comment: $LASSO$ = "least absolute shrinkage and selection operator")

```
from sklearn.linear_model import Lasso

lasso = Lasso(alpha=0.1, normalize=True)
lasso.fit(X_train, y_train)
lasso_pred = lasso.predict(X_test)
#Returns the coefficient of determination R^2 of the prediction.
lasso.score(X_test, y_test)
lasso.coef_
```

```
array([-0.   ,  0.   , -0.   ,  0.   , -0.   ,  3.189, -0.   , -0.   ,
       -0.   , -0.   , -0.307,  0.   , -0.487])
```

Feature Selection Property of the LASSO

```
names = boston.drop('medv', axis=1).columns
lasso_coef = lasso.fit(X, y).coef_
_ = plt.plot(range(len(names)), lasso_coef)
_ = plt.xticks(range(len(names)), names, rotation=60)
_ = plt.ylabel('Coefficients')
plt.show()
```



Tuning the shrinkage parameter with CV

(It seems that λ is often referred to as α)

From sklearn.linear_model:

- LassoCV
- RidgeCV
- GridSearchCV

```
from sklearn.linear_model import RidgeCV
```

```
reg = RidgeCV(alphas=[1e-3, 1e-2, 1e-1, 1],store_cv_values = True)
reg.fit(X, y)
reg.score(X, y)

reg.cv_values_
#Open Questions:
#1. how to extract all scores, possibly even the individual folds?
#2. What is the optimal alpha ??
#reg.cv_values_
```

```
array([[ 37.303,  37.347,  37.766,  40.434],
       [ 11.997,  11.974,  11.757,  10.501],
       [ 17.488,  17.492,  17.532,  17.786],
       ...,
       [ 14.642,  14.676,  15.003,  17.073],
       [ 17.862,  17.902,  18.281,  20.683],
       [113.709, 113.813, 114.793, 120.997]])
```

```
from sklearn.linear_model import LassoCV

reg = LassoCV(cv=5, random_state=0).fit(X, y)
reg.score(X, y)
```

```
0.7024437179872696
```

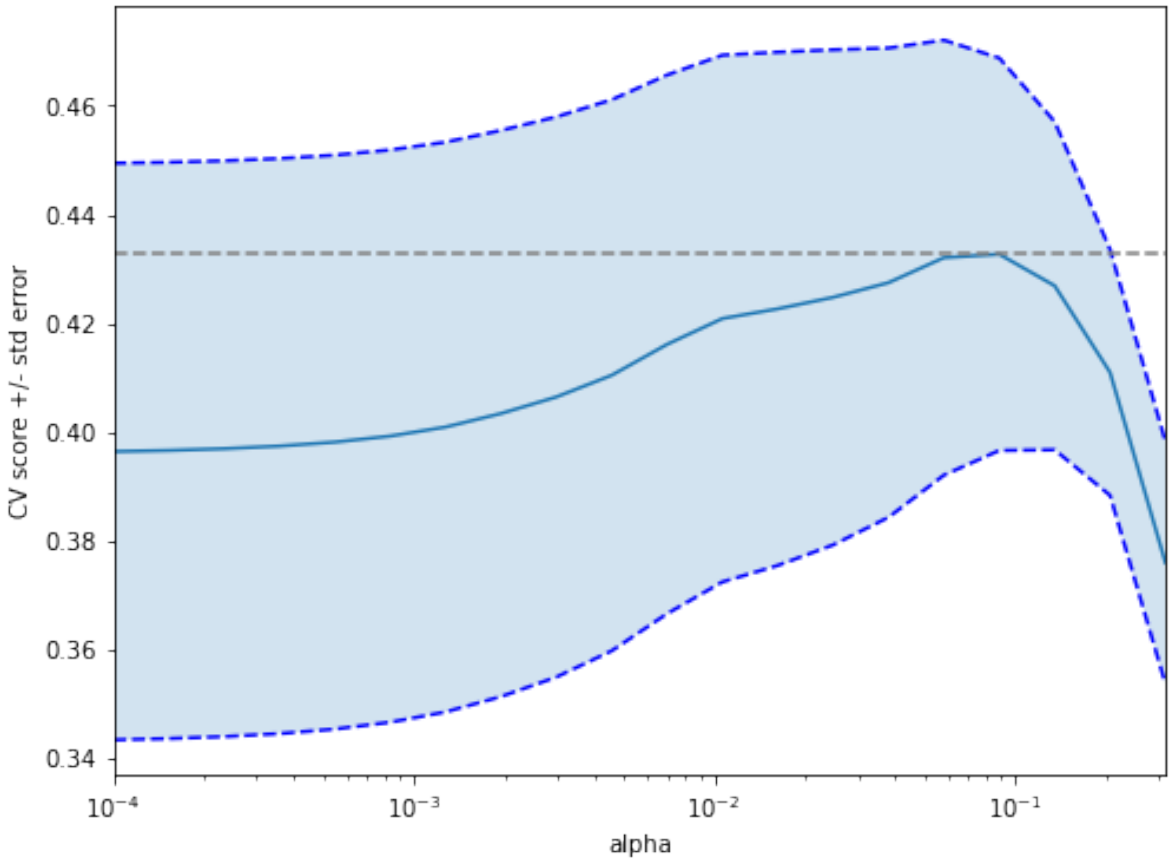
```
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV

diabetes = datasets.load_diabetes()
X = diabetes.data[:150]
y = diabetes.target[:150]


lasso = Lasso(random_state=0,max_iter=2000)
#logarithmically spaced sequence
alphas = np.logspace(-4, -0.5, 20)

tuned_parameters = [{'alpha': alphas}]
n_folds = 5
```

```
reg = GridSearchCV(lasso, tuned_parameters, cv=n_folds, refit=False)
reg.fit(X, y)
```

```
GridSearchCV(cv=5, estimator=Lasso(max_iter=2000, random_state=0),
             param_grid=[{'alpha': array([1.000e-04, 1.528e-04, 2.336e-04, 3.570e-04, 5.456e-
       1.274e-03, 1.947e-03, 2.976e-03, 4.549e-03, 6.952e-03, 1.062e-02,
       1.624e-02, 2.482e-02, 3.793e-02, 5.796e-02, 8.859e-02, 1.354e-01,
       2.069e-01, 3.162e-01])}],
             refit=False)
```

```
scores = reg.cv_results_['mean_test_score']
scores_std = reg.cv_results_['std_test_score']
plt.figure().set_size_inches(8, 6)
plt.semilogx(alphas, scores)

# plot error lines showing +/- std. errors of the scores
std_error = scores_std / np.sqrt(n_folds)

plt.semilogx(alphas, scores + std_error, 'b--')
plt.semilogx(alphas, scores - std_error, 'b--')

# alpha=0.2 controls the translucency of the fill color
plt.fill_between(alphas, scores + std_error, scores - std_error, alpha=0.2)

plt.ylabel('CV score +/- std error')
plt.xlabel('alpha')
plt.axhline(np.max(scores), linestyle='--', color='.5')
plt.xlim([alphas[0], alphas[-1]])
```

```
(0.0001, 0.31622776601683794)
```



How much can you trust the selection of alpha?

Task: Find the opimal Alpha parameters (maximising the generalization score) on different subsets of the data

14.0.4.3 ElasticNet

The mystery of the additional α paramater

- The elastic net for correlated variables, which uses a penalty that is part L1, part L2.
- Compromise between the ridge regression penalty ($\alpha = 0$) and the lasso penalty ($\alpha = 1$).
- This penalty is particularly useful in the $p >> N$ situation, or any situation where there are many correlated predictor variables.

$RSS + \lambda \sum_{j=1}^{p} \left(\frac{1}{2}(1-\alpha)\beta_j^2 + \alpha|\beta_j| \right)$

The right hand side can be written as

$\sum_{j=1}^{p} \frac{1}{2}\lambda(1-\alpha)\beta_j^2 + \alpha\lambda|\beta_j| = \sum_{j=1}^{p} \lambda_R\beta_j^2 + \lambda_L|\beta_j|$

with the Ridge penalty parameter $\lambda_R \equiv \frac{1}{2}\lambda(1-\alpha)$ and the lasso penalty parameter $\lambda_L \equiv \alpha\lambda$.

So we see that with

$\alpha = \frac{\lambda_L}{\lambda_L+2\lambda_R}$, and $\lambda = \lambda_L + 2\lambda_R$

Further Reading: - this notebook shows how to plot the entire "path" of coefficients.

14.0.5 Kaggle

14.0.5.1 Housing Data

This notebook (despite its annoying "humor") is a good start. (Get it directly from kaggle)

14.0.5.2 Your first submission

```
df_train = pd.read_csv('../data/kaggle/HousePrices/train.csv')
df_test = pd.read_csv('../data/kaggle/HousePrices/test.csv')
```

```
df_train
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl
...	...	...	...	...	...	...	...	...	...
1455	1456	60	RL	62.0	7917	Pave	NaN	Reg	Lvl
1456	1457	20	RL	85.0	13175	Pave	NaN	Reg	Lvl
1457	1458	70	RL	66.0	9042	Pave	NaN	Reg	Lvl
1458	1459	20	RL	68.0	9717	Pave	NaN	Reg	Lvl
1459	1460	20	RL	75.0	9937	Pave	NaN	Reg	Lvl

```
# Submission:
# pred_df = pd.DataFrame(y_pred, index=df_test.index, columns=["SalePrice"])
# pred_df.to_csv('../data/kaggle/HousePrices/submissions/en1.csv', header=True, index_labe
```