Lectures

Prof. Dr. Markus Löcher

05/28/2023

Table of contents

Preface

On this html page you will find a collection of the lectures from Digital Literacy I: Coding A SoSe-2023.

The labs can be found here

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
```

You can convert from one type to another easily:

```
int(3.5)
```

```
3
```

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	

```python
float(3)
```

```
3.0
```

More advanced numerical operations can be very useful.

- The Python Modulo, represented by the symbol %, is a mathematical operator that calculates the remainder of a division operation.
- The integer division //

```python
print("Dividing 10 by 3:")
print("minteger division:", 10 // 3)
print("remainder", 10 % 3)
```

```
Dividing 10 by 3:
remainder 3
remainder 1
```

When dealing with negative numbers, the Python Remainder Operator follows the "floored division" convention. For example, -10 % 3 would return 2, not -1. This is because -10 // 3 equals -4 with a remainder of 2.

1.1.2 Booleans

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

```python
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:

```python
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:

```python
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
```

```
#We can use lists:
for f in fam:
    print(f)
```

```
1.73
1.68
1.71
1.89
```

```
#or iterators
for i in range(len(fam)):
    print(fam[i])
```

```
1.73
1.68
1.71
1.89
```

```
#or enumerate
for i,f in enumerate(fam):
    print(fam[i], f)
```

```
1.73 1.73
1.68 1.68
1.71 1.71
1.89 1.89
```

2.0.6 Cumulative Means or Sums

As an example for a for loop, let us try to compute a "running mean" or a "cumulative sum" which is often and easily done in spreadsheets.

For example, can you compute the cumulative sum of a probability table:

```
prob = [0.05,0.1,0.2,0.25,0.15,0.1,0.1,0.05]
```

first in Excel, then in python.

```
prob = [0.05,0.1,0.2,0.25,0.15,0.1,0.1,0.05]
CumProb = prob[0]

for i,p in enumerate(prob):
    if (i==0):
        CumProb = [p]
    else:
        pCum = round(CumProb[i-1] + p,2)
        CumProb.append(pCum)

CumProb
```

```
[0.05, 0.15, 0.35, 0.6, 0.75, 0.85, 0.95, 1.0]
```

Comment: We will get to know pandas data frames later and learn about the convenient `cumsum()` method for pandas objects.

2.0.7 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 all the values.

Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods `iter()` and `next()`.

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

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

```
apple
banana
```

```
mystr = "banana"
myit = iter(mystr)

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

```
b
a
```

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

```
#funny iterators
print(range(5))
list(range(5))
```

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

```
[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: Compute

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

```
[0, 1, 2, 3, 4]
```

2.0.8 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..

```python
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*.

```
#Analogy to a language dictionary:
EnglishGerman = {"slander": "Verleumdung",
                 "salient" : ["auffallend", "hervorstechend"],
                 "secular" : "profan",
                 "rejoice" : "jubeln"}
```

```
#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],
        "gdp": [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	gdp
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6

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

```
KeyError: (2, 1)
```

```
#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["gdp"]
```

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

```
frame.gdp
```

```
<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 !!
```

	state	pop
year		
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"]
```

	year	pop
state		
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"]
```

	year	continent	lifeExp	pop	gdpPercap
country					
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 futur
  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 
  byContinent.mean()
```

	year	lifeExp	pop	gdpPercap
continent				
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])
```

	min	max	mean
continent			
Africa	241.165876	21951.21176	2193.754578
Americas	1201.637154	42951.65309	7136.110356
Asia	331.000000	113523.13290	7902.150428

continent	min	max	mean
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

```python
# 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

```python
#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 ])
```

	min	max	mean
sex			
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
```

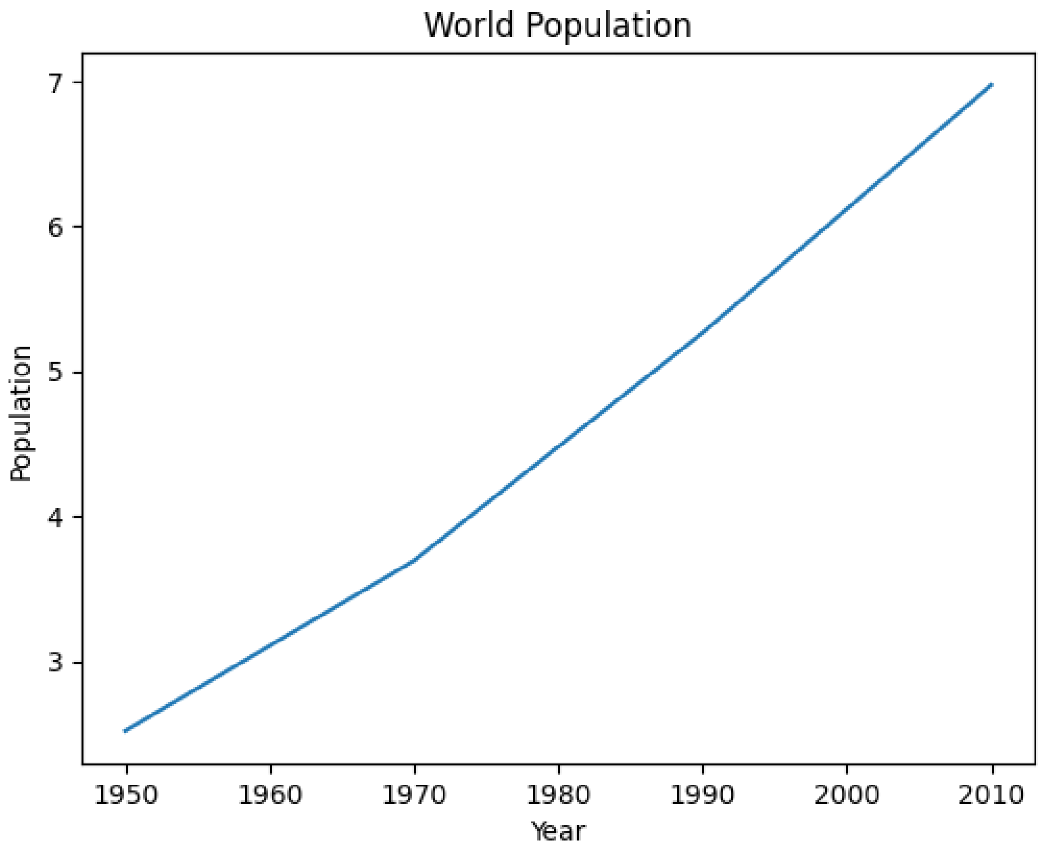
```
bySex = titanic.groupby("sex")
#here I am specifically asking for the mean
bySex["survived"].mean()
```

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.

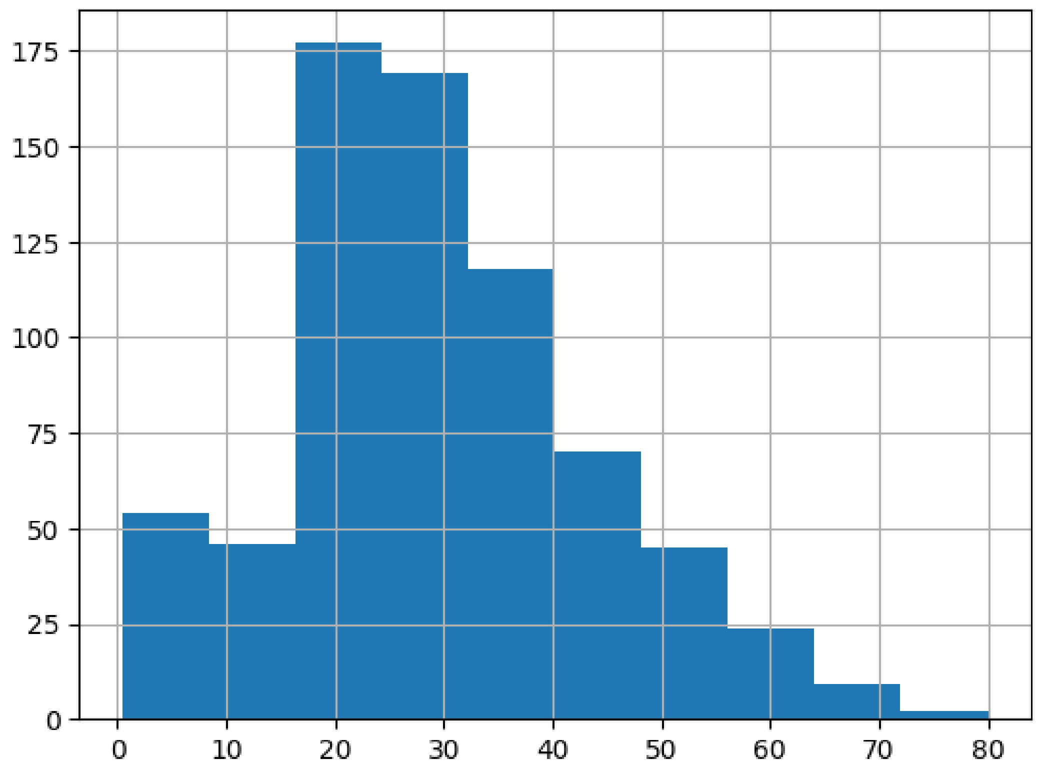
```
#%matplotlib inline
import matplotlib.pyplot as plt
```

```
#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

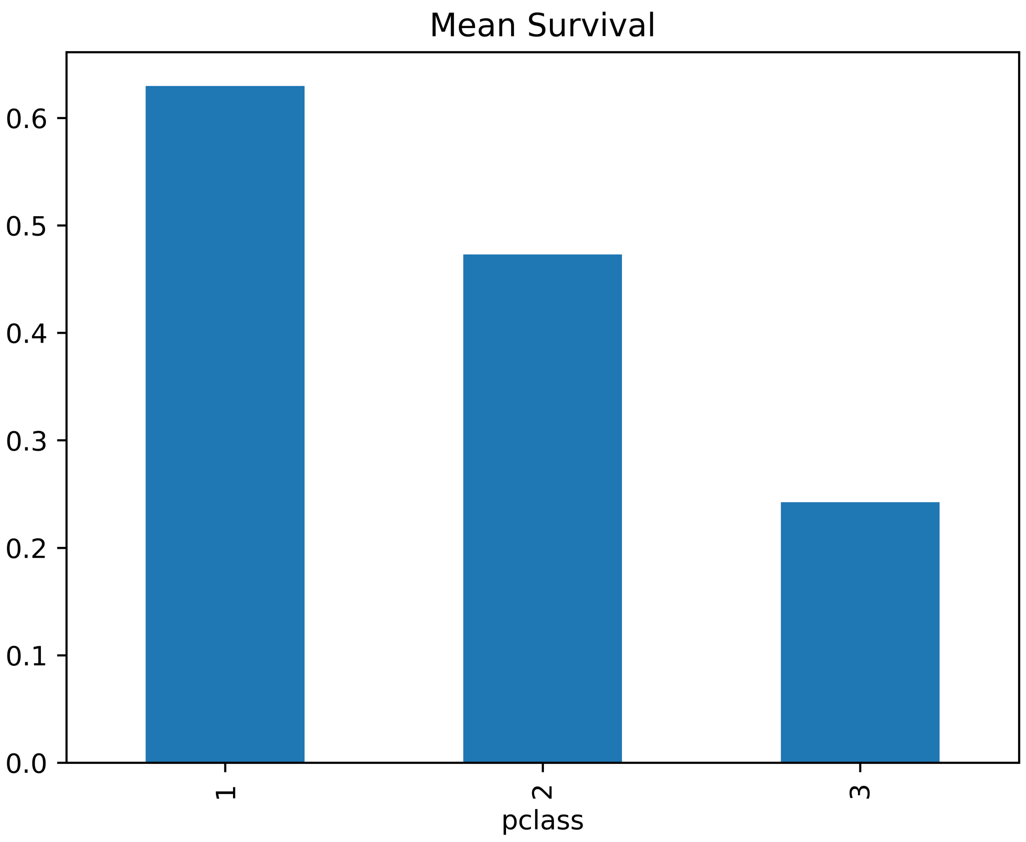
```
titanic.age.hist()
plt.show()
```



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");
```

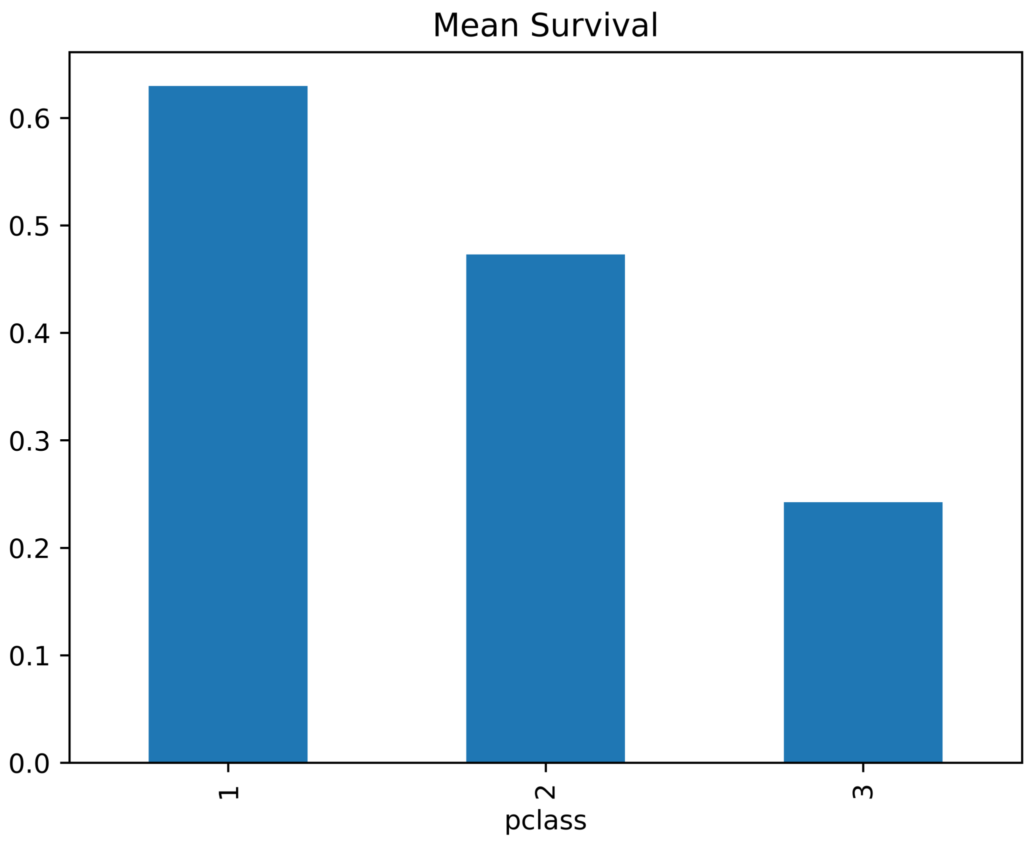
```
<Axes: title={'center': 'Mean Survival'}, xlabel='pclass'>
```



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'>
```

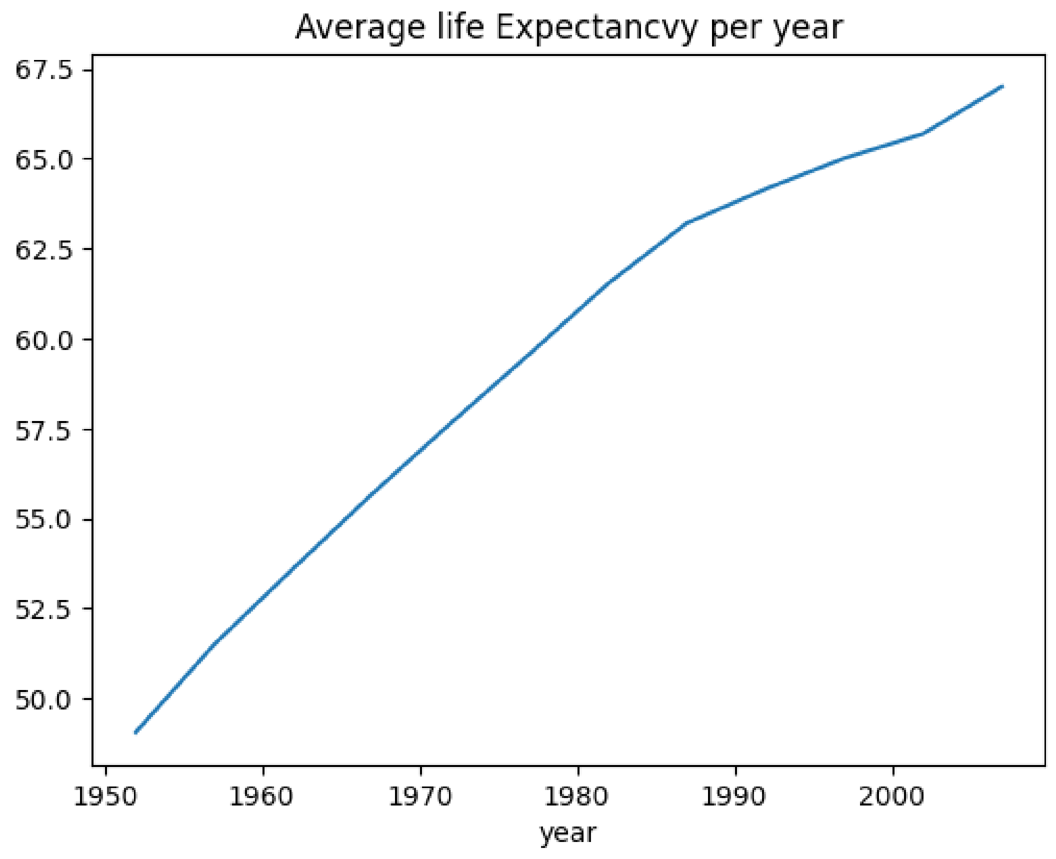


Tasks:

- 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 futur
  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.

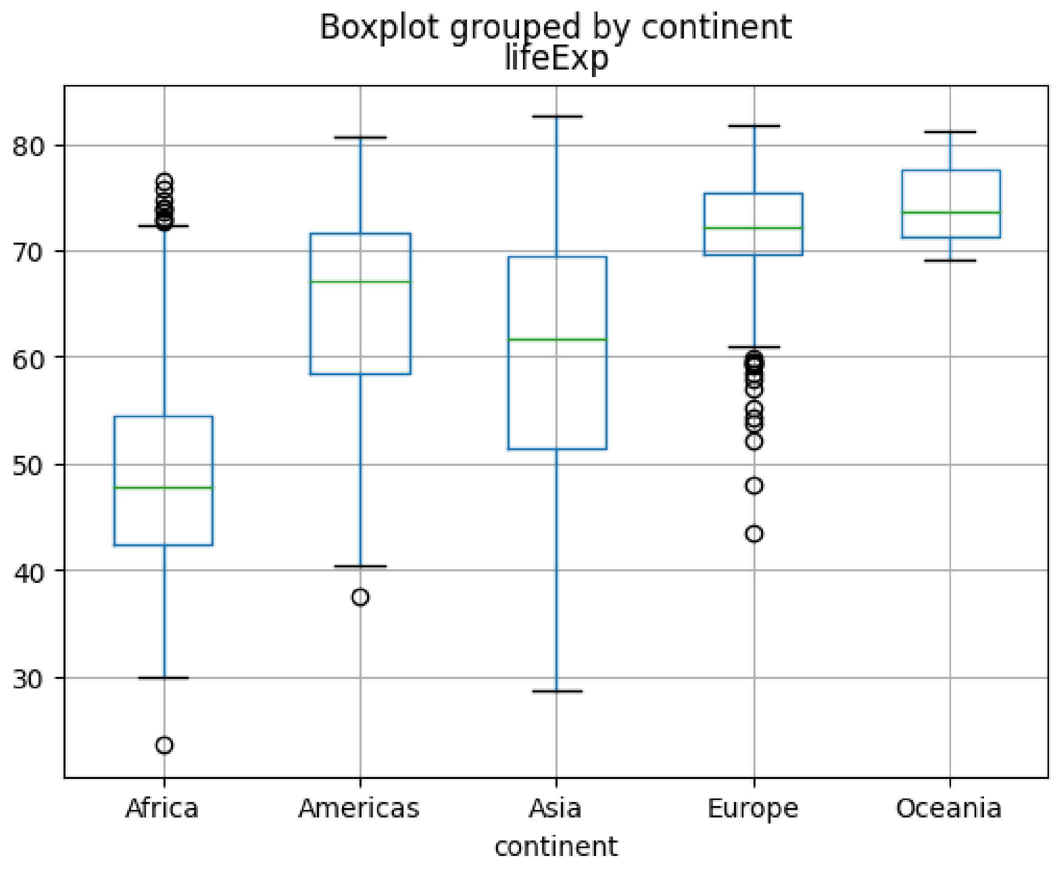
```
!pip install gapminder
from gapminder import gapminder
```

```
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");
```



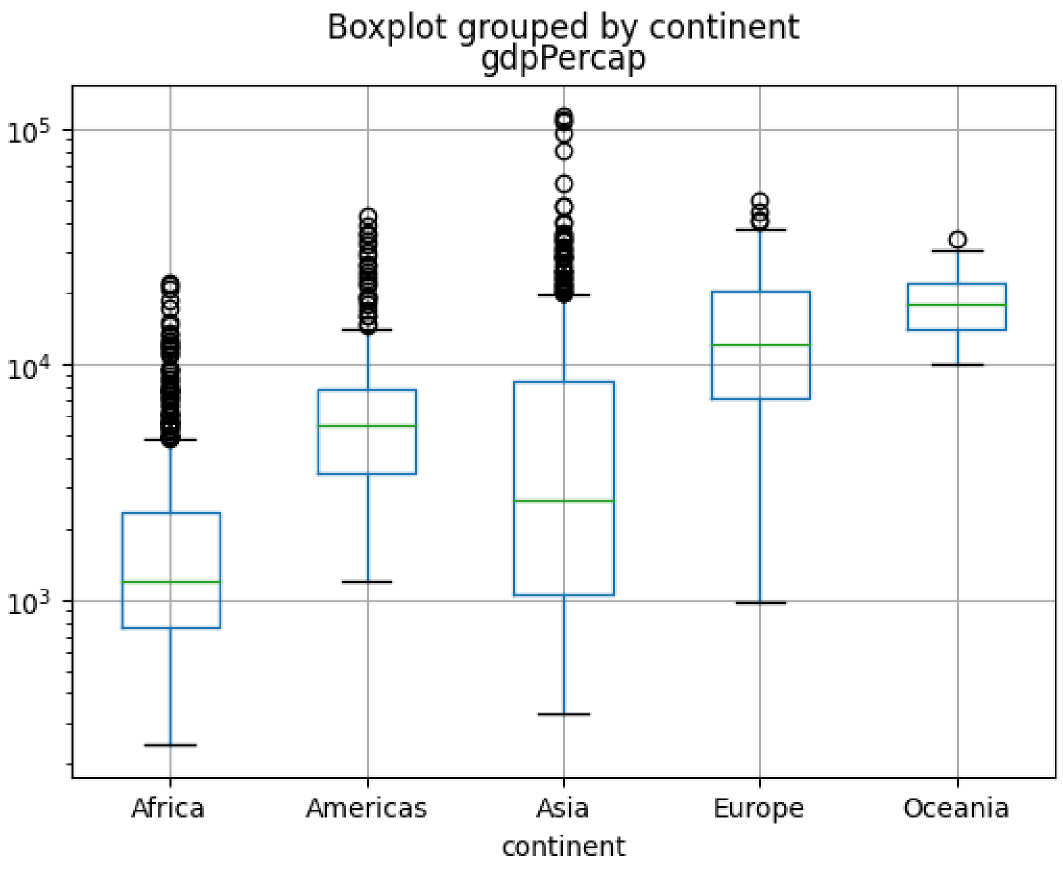
The matplotlib way

```
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 warnings
```

```
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

	country	continent	year	lifeExp	pop	gdpPercap
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
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

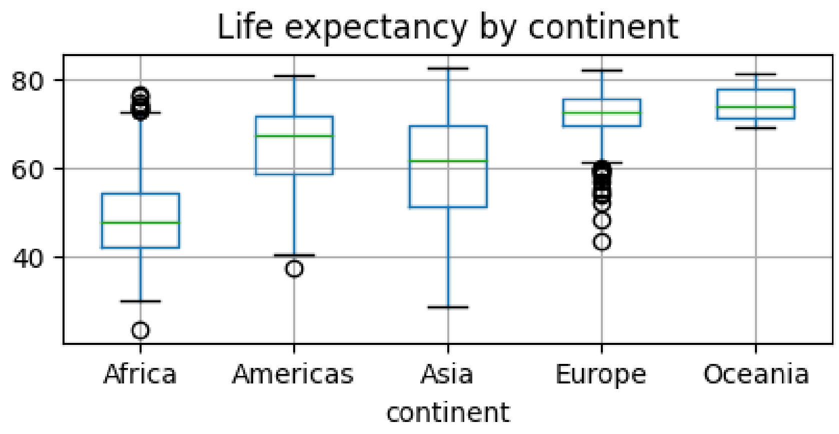
1. are not numeric (but could be represented as numbers)
2. cannot really be ordered
3. typically take on a finite set of values, or *categories*.

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.

```python
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
```

```python
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?

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

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

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

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

What are the estimated survival probabilities?

```python
#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())
```

```
[515 560 396 252 159 117 151 835 534 871 727 137 609 214 819 729 232 553
 727 432]
```

```
19
```

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

```
891
```

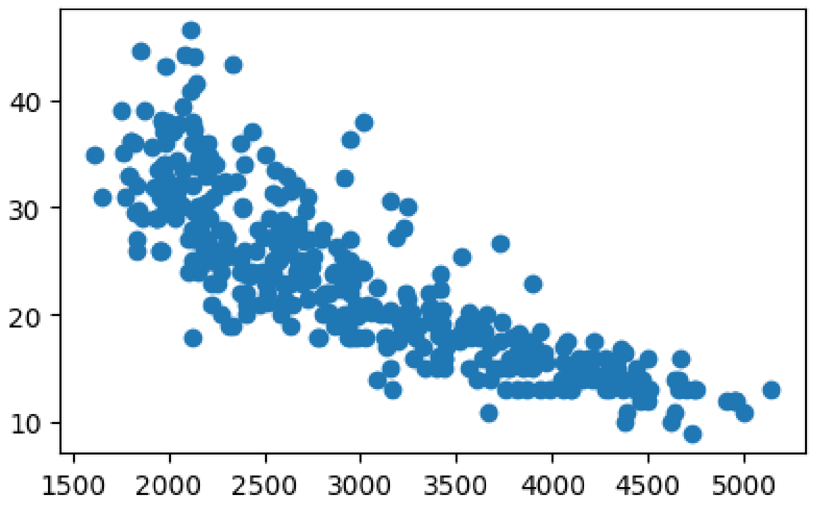
```
array([0.26956522, 0.26956522, 0.94705882, 0.13544669, 0.13544669,
       0.26956522, 0.26956522, 0.5       , 0.94705882, 0.26956522,
       0.5       , 0.13544669, 0.13544669, 0.94705882, 0.5       ,
       0.5       , 0.13544669, 0.94705882, 0.94705882])
```

8.2.2.1 From categorical to numerical relations

```
url = "https://raw.githubusercontent.com/markusloecher/DataScience2018/master/data/Auto.cs
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"]);
```

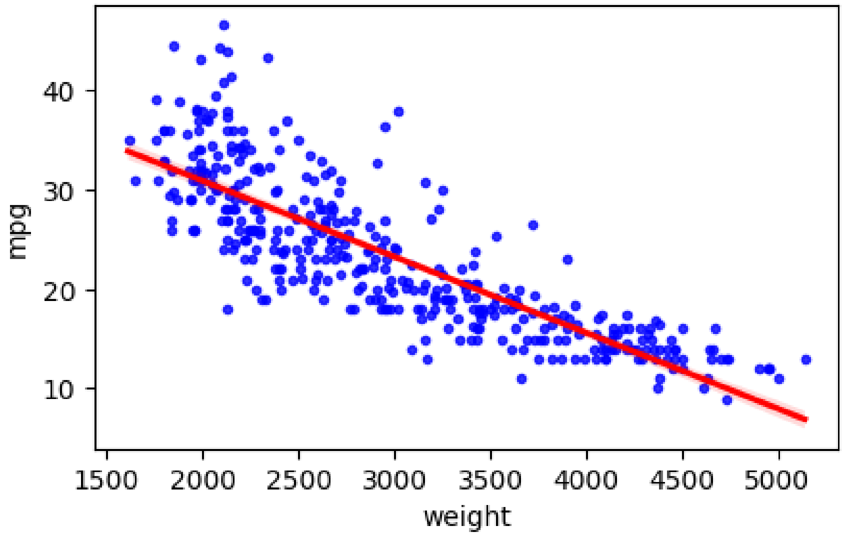


8.3 Linear Regression

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

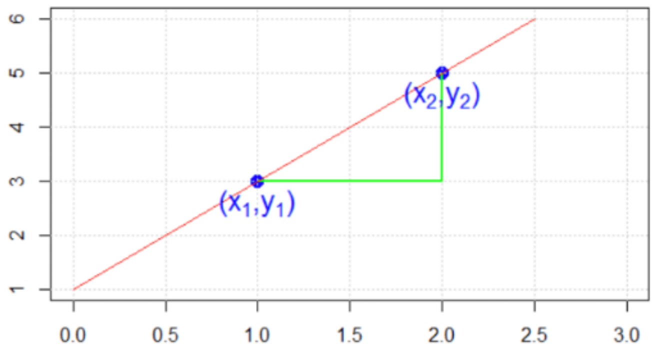


Figure 8.1: SlopeIllustration

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

```python
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

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

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

8.4 Penguins

Short description

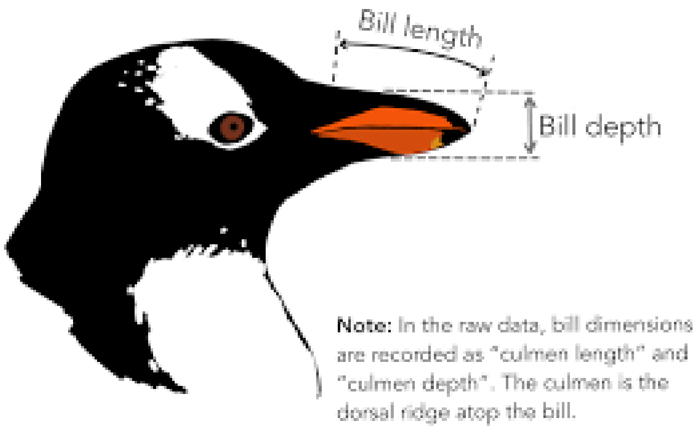

Bill length
Bill depth
Note: In the raw data, bill dimensions are recorded as "culmen length" and "culmen depth". The culmen is the dorsal ridge atop the bill.

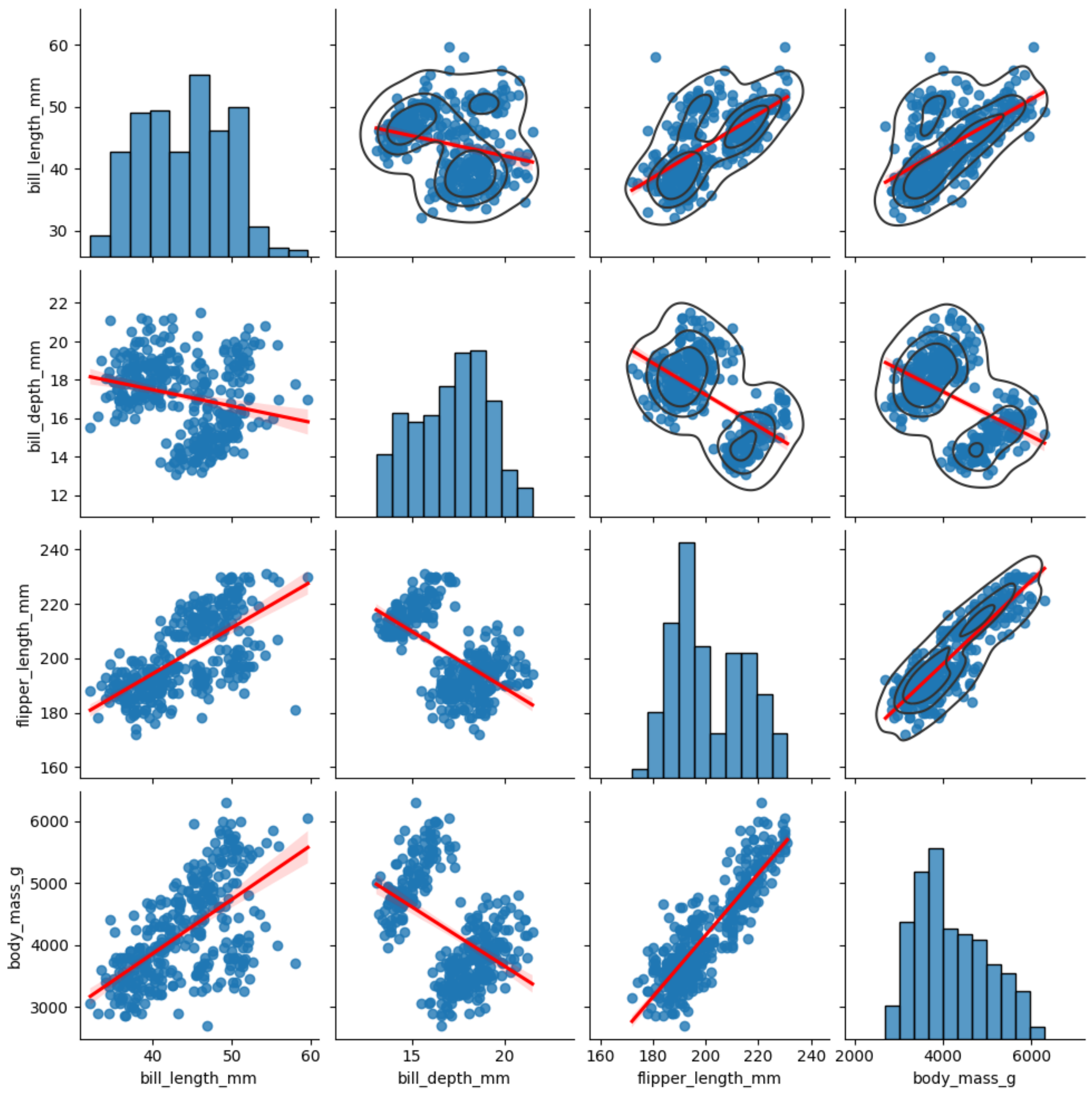
Figure 8.2: penguin image

```python
import seaborn as sns
import matplotlib.pyplot as plt
```

```
penguins = sns.load_dataset("penguins")
penguins
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Femal
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Femal
3	Adelie	Torgersen	NaN	NaN	NaN	NaN	NaN
4	Adelie	Torgersen	36.7	19.3	193.0	3450.0	Femal
...	...	...	...	...	...	...	...
339	Gentoo	Biscoe	NaN	NaN	NaN	NaN	NaN
340	Gentoo	Biscoe	46.8	14.3	215.0	4850.0	Femal
341	Gentoo	Biscoe	50.4	15.7	222.0	5750.0	Male
342	Gentoo	Biscoe	45.2	14.8	212.0	5200.0	Femal
343	Gentoo	Biscoe	49.9	16.1	213.0	5400.0	Male

```
warnings.filterwarnings("ignore")

g=sns.pairplot(penguins, kind="reg", plot_kws={'line_kws':{'color':'red'}}, corner=False);
g.map_upper(sns.kdeplot, levels=4, color=".2");
plt.show()
```



The "data tells us" that * bill depth seems to **decrease** with bill length * flipper length and body mass seem to **decrease** with bill depth. * flipper length and body mass seem to **increase** with bill length.

Can we fit simple regression lines to prove our visual hypothesis?

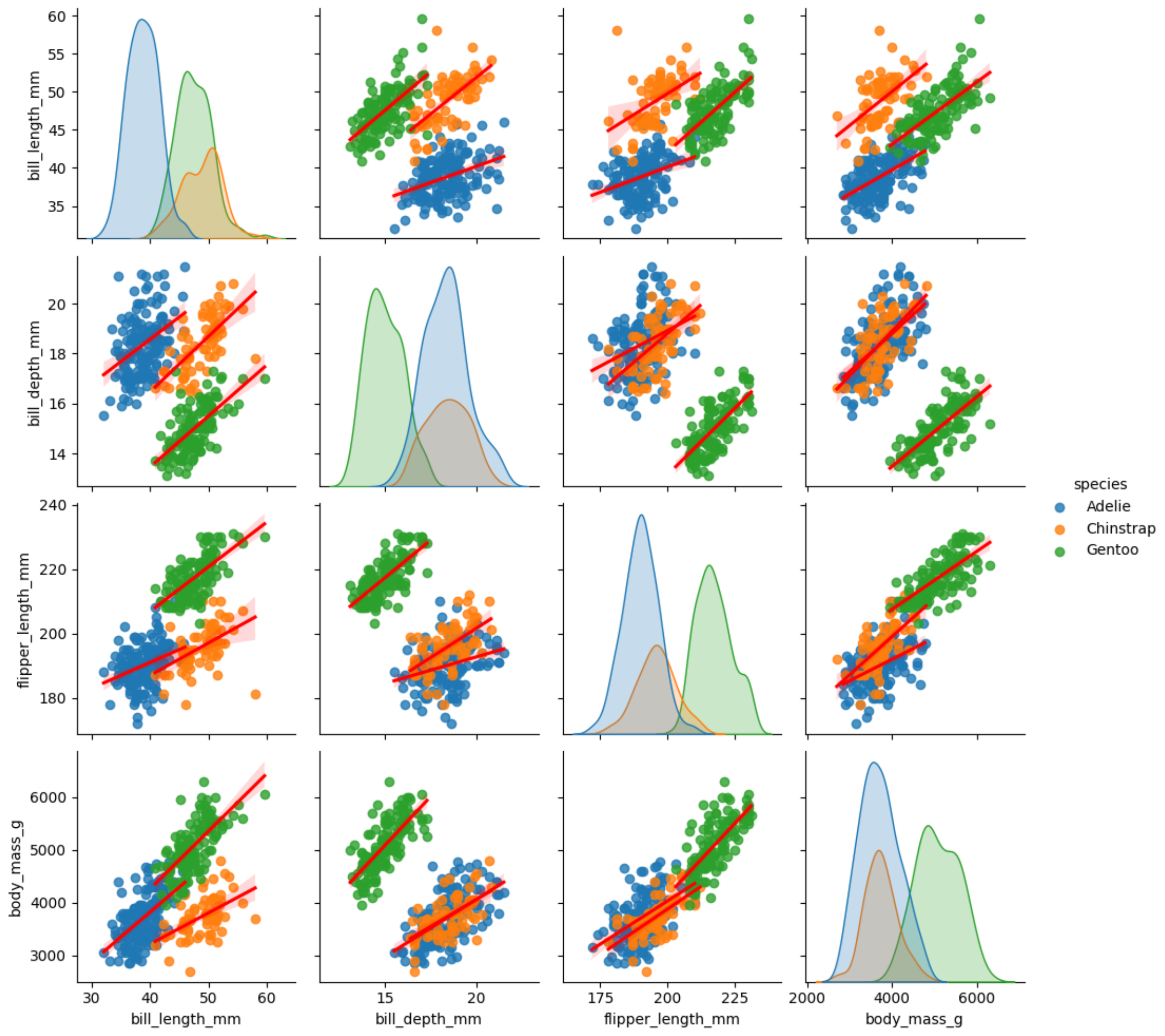
```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

```
lm1 = smf.ols('bill_depth_mm ~ bill_length_mm', penguins).fit()
lm1.summary().tables[1]

lm2 = smf.ols('flipper_length_mm ~ bill_depth_mm', penguins).fit()
lm2.summary().tables[1]
```

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	272.2190	5.413	50.294	0.000	261.573	282.865
bill_depth_mm	-4.1574	0.314	-13.261	0.000	-4.774	-3.541

BUT: the interpretation of the data is model dependent ! The data does not “tell the truth” by itself:

```
sns.pairplot(penguins,  hue="species", kind="reg", plot_kws={'line_kws':{'color':'red'}},
plt.show()
```

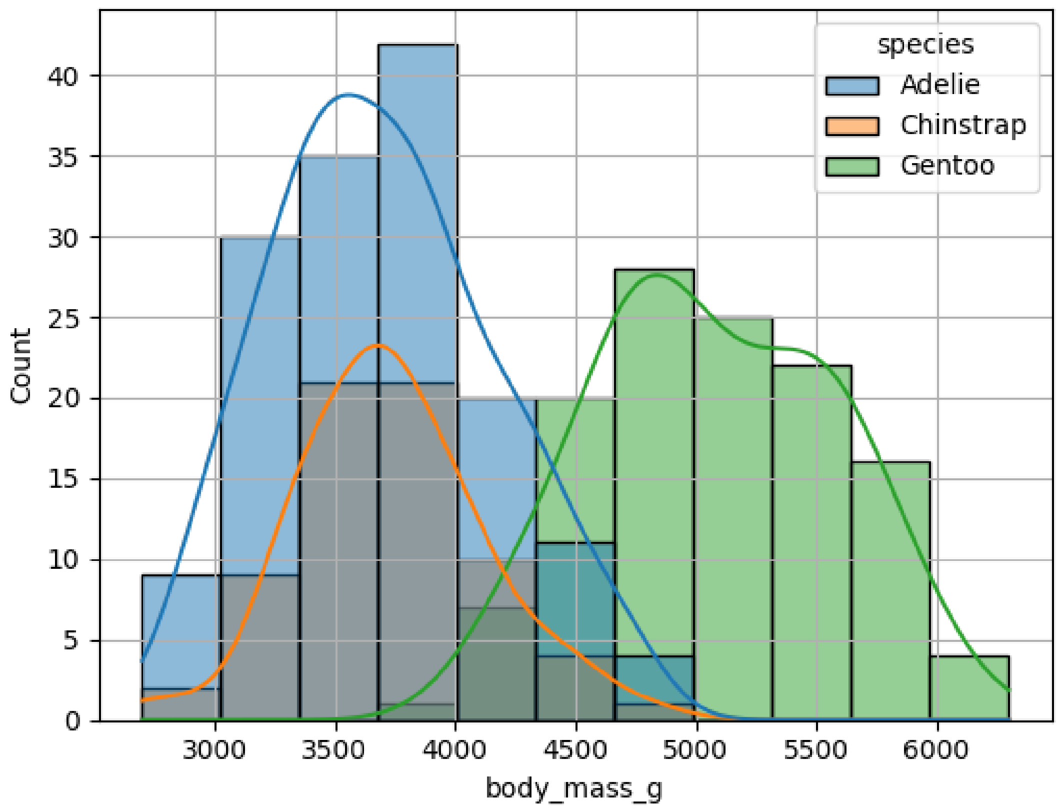


```
lm2a = smf.ols('flipper_length_mm ~ bill_depth_mm*species', penguins).fit()
lm2a.summary().tables[1]
```

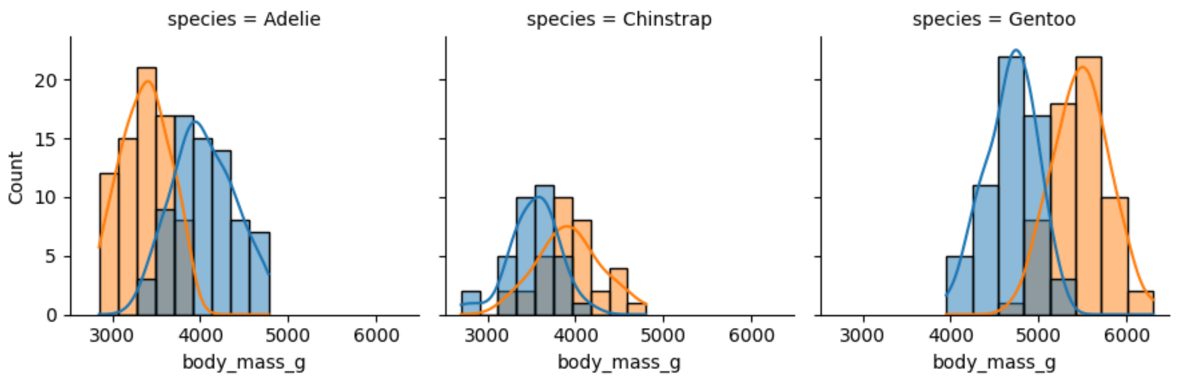
	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	159.6189	6.943	22.990	0.000	145.962	173.276
species[T.Chinstrap]	-30.9222	13.155	-2.351	0.019	-56.798	-5.046
species[T.Gentoo]	-12.3945	10.439	-1.187	0.236	-32.928	8.139
bill_depth_mm	1.6534	0.378	4.379	0.000	0.911	2.396
bill_depth_mm:species[T.Chinstrap]	1.9907	0.714	2.790	0.006	0.587	3.394
bill_depth_mm:species[T.Gentoo]	3.0163	0.642	4.698	0.000	1.754	4.279

8.5 Grouping with multiple categories

```
warnings.filterwarnings("ignore")

sns.histplot(data=penguins, x="body_mass_g", kde=True, hue = "species");plt.grid();
```



```
g = sns.FacetGrid(penguins, col="species")
g.map_dataframe(sns.histplot, x="body_mass_g",  hue = "sex", kde=True);
```



Further Reading:

-

9 String Manipulations, RegEx

(Lecture 8)

In this lecture we will continue our journey of **String Manipulation** after reviewing **list comprehensions**

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

```python
#new module
import re
```

9.1 List Comprehensions

List comprehensions are a convenient and widely used Python language feature. They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter into one concise expression. They take the basic form:

`[expr for value in collection if condition]`

This is equivalent to the following for loop:

```
result = []
for value in collection:
    if condition:
        result.append(expr)
```

The filter condition can be omitted, leaving only the expression. For example, given a list of strings, we could filter out strings with length 2 or less and convert them to uppercase like this:

```
strings = ["eye", "lived devil", "wow", "deed", "noon", "kayak"]

results = []
#convert those strings to upper case that are longer than 3 characters
for x in strings:
  if len(x) > 3:
    results.append(x.upper())

results
```

```
['LIVED DEVIL', 'DEED', 'NOON', 'KAYAK']
```

```
[x.upper() for x in strings if len(x) > 3]
```

```
['LIVED DEVIL', 'DEED', 'NOON', 'KAYAK']
```

```
[x.upper() for x in strings if len(x) > 3]
```

9.1.1 Dictionary Comprehension

A dictionary comprehension looks like this:

```
dict_comp = {key-expr: value-expr for value in collection
             if condition}
```

As a simple dictionary comprehension example, we could create a lookup map of these strings for their locations in the list:

```
results = {}
for i, x in enumerate(strings):
  results[x] = i

results
```

```
{'eye': 0, 'lived devil': 1, 'wow': 2, 'deed': 3, 'noon': 4, 'kayak': 5}
```

```
results = {x:i for i, x in enumerate(strings) if len(x) > 3}
results
```

```
{'lived devil': 1, 'deed': 3, 'noon': 4, 'kayak': 5}
```

Like list comprehensions, set and dictionary comprehensions are mostly conveniences, but they similarly can make code both easier to write and read. Consider the list of strings from before. Suppose we wanted a set containing just the lengths of the strings contained in the collection; we could easily compute this using a set comprehension:

```
unique_lengths = {len(x) for x in strings}
unique_lengths
```

9.2 String Operations

Built-in methods

```
my_string = "A man, a plan, a canal: Panama"
my_string.lower()#this is not inplace
```

```
TypeError: ignored
```

```
my_string
```

```
'A man, a plan, a canal: Panama'
```

```
my_string.count("a")#it is case sensitive
```

```
9
```

9.2.0.1 Splitting

```
IndWords = my_string.split(sep = " ")#both the colon and the commas are still present
print(IndWords)
```

```
['A', 'man,', 'a', 'plan,', 'a', 'canal:', 'Panama']
```

```
print(my_string.split(sep=","))
print(my_string.split(maxsplit=2))
#Breaking at line boundaries
print(my_string.splitlines())
```

```
['A man', ' a plan', ' a canal: Panama']
['A', 'man,', 'a plan, a canal: Panama']
['A man, a plan, a canal: Panama']
```

```
my_string = "A man, a plan, a canal:
Panama"
#Breaking at line boundaries
print(my_string.splitlines())
```

```
my_string = "A man, a plan, a canal\n Panama"
print(my_string)
print(my_string.splitlines())
```

```
A man, a plan, a canal
 Panama
['A man, a plan, a canal', ' Panama']
```

9.2.0.2 Joining

Concatenate strings from iterables

```
print(" ".join(IndWords))
```

```
A man, a plan, a canal: Panama
```

9.2.0.3 Slicing

```
print(my_string[0:5])#the same for numpy arrays or lists !
```

```
A man
```

```
print(my_string[10:20:2])
```

```
ln  a
```

9.2.0.4 Palindromes

Which words are palindromes ?

```
"eye kayak"[::-1]
"Berlin"[::-1]
```

```
'nilreB'
```

```
"edit tide"[::-1]
```

```
'edit tide'
```

```
print(my_string[::-1])
```

9.3 Find and Replace

```
my_string.find("Panama")#success
```

```
24
```

```
 my_string.find("Berlin")#not found, failure
```

```
-1
```

```
my_string.index("Panama")
```

```
24
```

```python
my_string.replace("Panama", "Suez")
```

```
'A man, a plan, a canal\n Suez'
```

```python
my_string.replace("Berlin", "Suez")#failure mode
```

9.4 Regular Expressions

The re module offers a set of functions that allows us to search a string for a match:

Function

Description

findall

Returns a list containing all matches

search

Returns a Match object if there is a match anywhere in the string

split

Returns a list where the string has been split at each match

sub

Replaces one or many matches with a string

Metacharacters are characters with a special meaning:

Character

Description

Example

Try it

[]

A set of characters

”[a-m]”

Try it »

</td>

Signals a special sequence (can also be used to escape special characters)

"."

Try it »

.

Any character (except newline character)

"he..o"

Try it »

^

Starts with

"^hello"

Try it »

*

Zero or more occurrences

"he.*o"

Try it »

+

One or more occurrences

"he.+o"

Try it »

?

Zero or one occurrences

"he.?o"

Try it »

{}

Exactly the specified number of occurrences

"he.{2}o"

Try it »

|

Either or

”falls|stays”

Try it »

()

Capture and group

```
#Find digits via \d
string="The winners are: User2-4, UserN, User1, UserMarkus"
re.findall(r"User\d", string)
```

```
['User2', 'User1']
```

```
#exclude digits via \D
re.findall(r"User\D", string)
```

```
['UserN']
```

```
#find words via \w
re.findall(r"User\w", string)
```

```
['User2', 'UserN', 'User1', 'UserM']
```

```
#white spaces \s
string="Is it New York or  New-York?"
re.findall(r"New\sYork", string)
```

```
['New York']
```

```
#not a white spaces \S
string="Is it New York or  New-York?"
re.findall(r"New\SYork", string)
```

```
['New-York']
```

9.4.1 Repetitions

```
string = "my password is password1234"
#re.search(r"\w{8}\d{4}", string)
re.findall(r"\w{8}\d{4}", string)
```

```
['password1234']
```

9.4.2 Quantifiers

How many times to match a pattern immediately to **its left**

- `+` once or more
- `*` zero times or more
- `?` zero times or once
- `{n,m}` n times at least, m times at most

```
text = "Possible exam dates: 6-25 or 7-5 or 2023-12-24"
re.findall(r"\d+-\d+", text)
```

```
['6-25', '7-5', '2023-12']
```

```
text = "Possible exam dates: 6-25 or 7-5 or 2023-12-24"
re.findall(r"\d+-\d+-*\d*", text)
```

```
['6-25', '7-5', '2023-12-24']
```

```
text = "Possible exam dates: 6-25 or 7-5 or 2023-12-24"
re.findall(r"\d+-\d+-?\d?", text)
```

```
['6-25', '7-5', '2023-12-2']
```

"Escaping" the special meaning of `+`

```
phone_number = "Mobile: +49 1578-1378941 or (49)1578-1378941"
re.findall(r"\+\d{2}\s*\d{4}-\d{7}", phone_number)
```

```
['+49 1578-1378941']
```

"Escaping" the special meaning of `()`

```python
re.findall(r"\(\d{2}\)\s*\d{4}-\d{7}", phone_number)
```

```
['(49)1578-1378941']
```

The or operator `|`

```python
re.findall(r"\+\d{2}\s*\d{4}-\d{7}|\(\d{2}\)\s*\d{4}-\d{7}", phone_number)
```

```
['+49 1578-1378941', '(49)1578-1378941']
```

```python
phone_number = "At HWR: 030-30877-1443 Mobile: 01578-1378941 or "
re.findall(r"\d{2,3}-\d{4,5}-\d{4}|\d{5}-\d{7}", phone_number)
```

```
['030-30877-1443', '01578-6798941']
```

9.4.3 Special Characters

- `.` Match any character (except newlines)
- `^` Start of the String
- `$` End of the String
- `\` Escape special characters
- `|` OR operator
- `[]` group of characters

```python
my_links = "check out my blogs: https://markusloecher.github.io codeandstats.github.io"
re.findall(r"https://.+github.io", my_links)
```

```
['https://markusloecher.github.io codeandstats.github.io']
```

```python
print(re.findall(r"^out", my_links))
re.findall(r"^check", my_links)
```

```
[]
```

```
['check']
```

```
re.findall(r"\w+\.github.io$", my_links)
```

```
['codeandstats.github.io']
```

```
my_links = "check out my blogs: https://markusloecher.github.io codeandstats.github.io"
re.findall(r"[a-z.]github.io", my_links)
```

```
NameError: ignored
```

Further Reading:

-

10 Subplots and Pivoting

(Lecture 9)

In this lecture we will continue our journey of pandas after reviewing a few more details of `matplotlib`

1. Grid of subplots
    - Annotations
    - Legends
2. More pandas
    - Assign
    - Pivoting
3. Smoking Data

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

10.1 Grid of subplots

A Figure object is the outermost container for a matplotlib graphic, which can contain multiple Axes objects. One source of confusion is the name: an Axes actually translates into what we think of as an individual plot or graph (rather than the plural of "axis," as we might expect).

You can think of the Figure object as a box-like container holding one or more Axes (actual plots). Below the Axes in the hierarchy are smaller objects such as tick marks, individual lines, legends, and text boxes. Almost every "element" of a chart is its own manipulable Python object, all the way down to the ticks and labels:

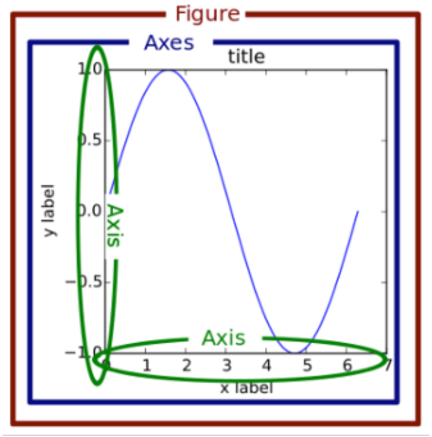
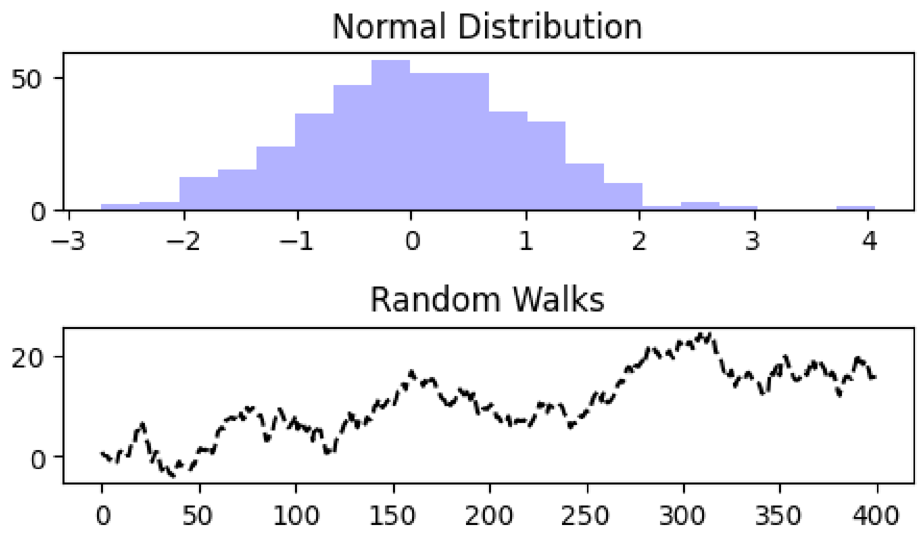


Figure 10.1: https://realpython.com/python-matplotlib-guide

10.1.0.1 Via `add_subplot()`

```
#create a grid of two rows and one column
fig = plt.figure(figsize=(5, 3))
ax1 = fig.add_subplot(2, 1, 1)
ax2 = fig.add_subplot(2, 1, 2)

RanNums = np.random.standard_normal(400)
ax1.hist(RanNums, bins=20, color="blue", alpha=0.3);
ax1.set_title("Normal Distribution");

RanWalk = RanNums.cumsum()
ax2.plot(RanWalk, color="black", linestyle="dashed");
ax2.set_title("Random Walks");
fig.tight_layout()
```



10.1.0.2 Via `subplots()`

To make creating a grid of subplots more convenient, matplotlib includes a `plt.subplots` method that creates a new figure and returns a NumPy array containing the created subplot objects:

```
x = np.random.randint(low=1, high=11, size=50)
y = x + np.random.randint(1, 5, size=x.size)
data = np.column_stack((x, y))

fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,
                               figsize=(8, 4))

ax1.scatter(x=x, y=y, marker='o', c='r', edgecolor='b')
ax1.set_title('Scatter: $x$ versus $y$')
ax1.set_xlabel('$x$')
ax1.set_ylabel('$y$')

ax2.hist(data, bins=np.arange(data.min(), data.max()),
         label=('x', 'y'))
ax2.legend(loc=(0.65, 0.8))
ax2.set_title('Frequencies of $x$ and $y$')
ax2.yaxis.tick_right()
```

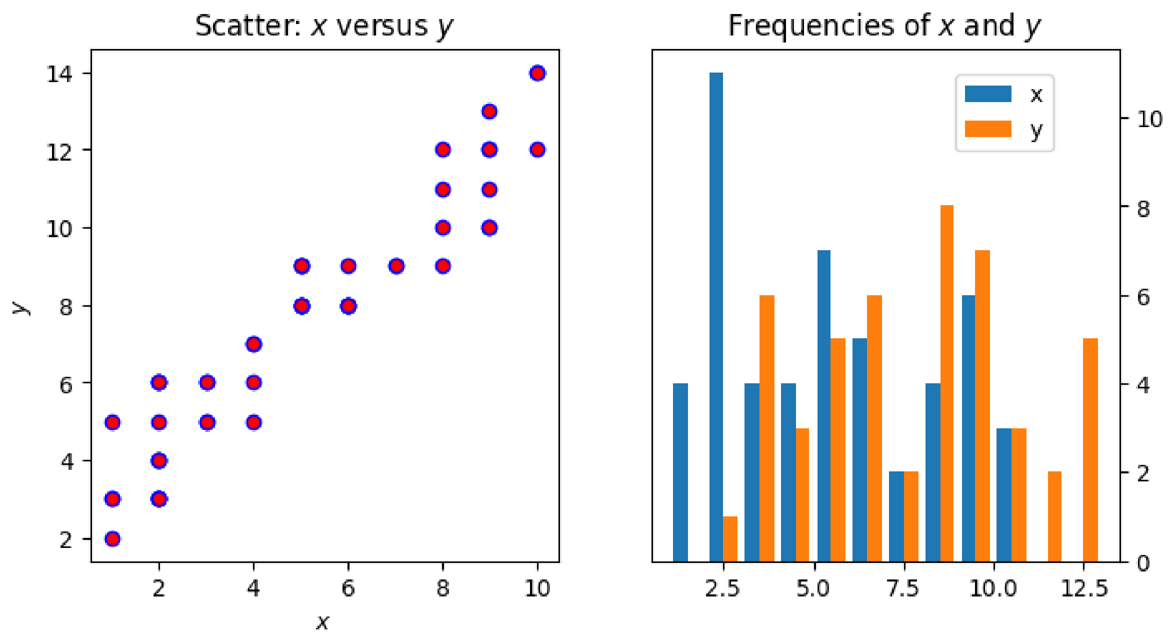


Table 9.1: matplotlib.pyplot.subplots options

Argument	Description
nrows	Number of rows of subplots
ncols	Number of columns of subplots
sharex	All subplots should use the same x-axis ticks (adjusting the xlim will affect all subplots)
sharey	All subplots should use the same y-axis ticks (adjusting the ylim will affect all subplots)
subplot_kw	Dictionary of keywords passed to add_subplot call used to create each subplot
**fig_kw	Additional keywords to subplots are used when creating the figure, such as plt.subplots(2, 2, figsize=(8, 6))

10.1.1 Annotations

```
from datetime import datetime
url = "https://raw.githubusercontent.com/wesm/pydata-book/3rd-edition/examples/spx.csv"
stock_data = pd.read_csv(url, index_col=0, parse_dates=True)
spx = stock_data["SPX"]
```

```
stock_data.head()
```

	SPX
Date	
1990-02-01	328.79
1990-02-02	330.92
1990-02-05	331.85
1990-02-06	329.66
1990-02-07	333.75

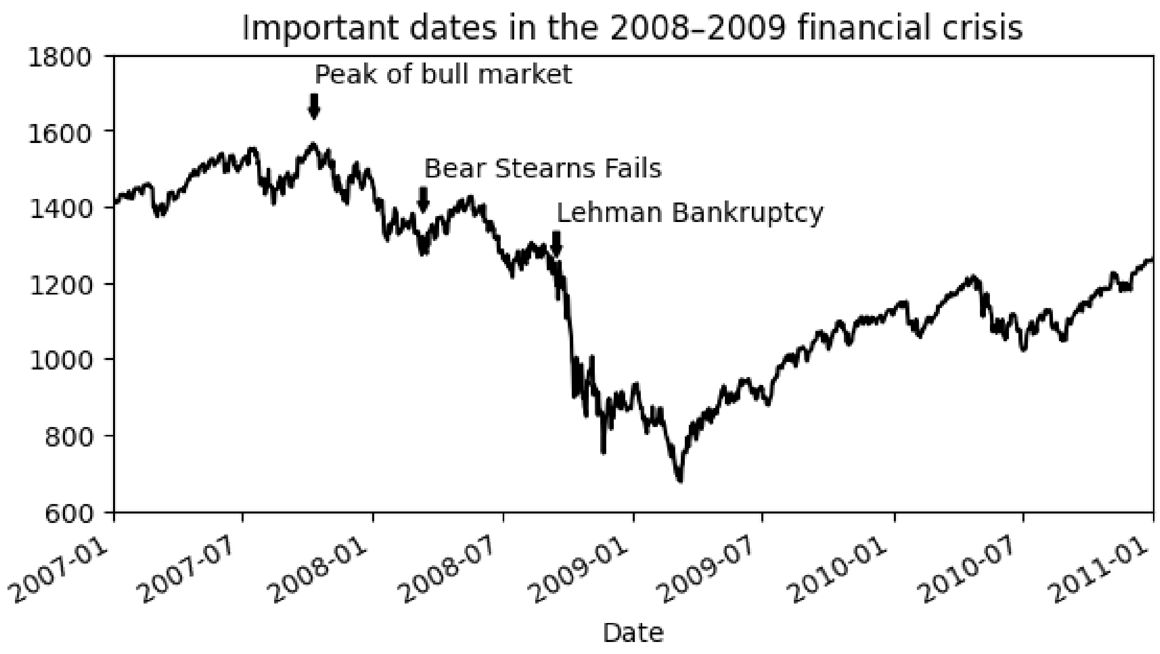
```
spx[0]

spx["2007-10-11"]

spx.asof(datetime(2007, 10, 11))
```

```
1554.41
```

```
crisis_data = [
    (datetime(2007, 10, 11), "Peak of bull market"),
    (datetime(2008, 3, 12), "Bear Stearns Fails"),
    (datetime(2008, 9, 15), "Lehman Bankruptcy")
]

#crisis_data[0][0]
spx.asof(crisis_data[0][0])
```

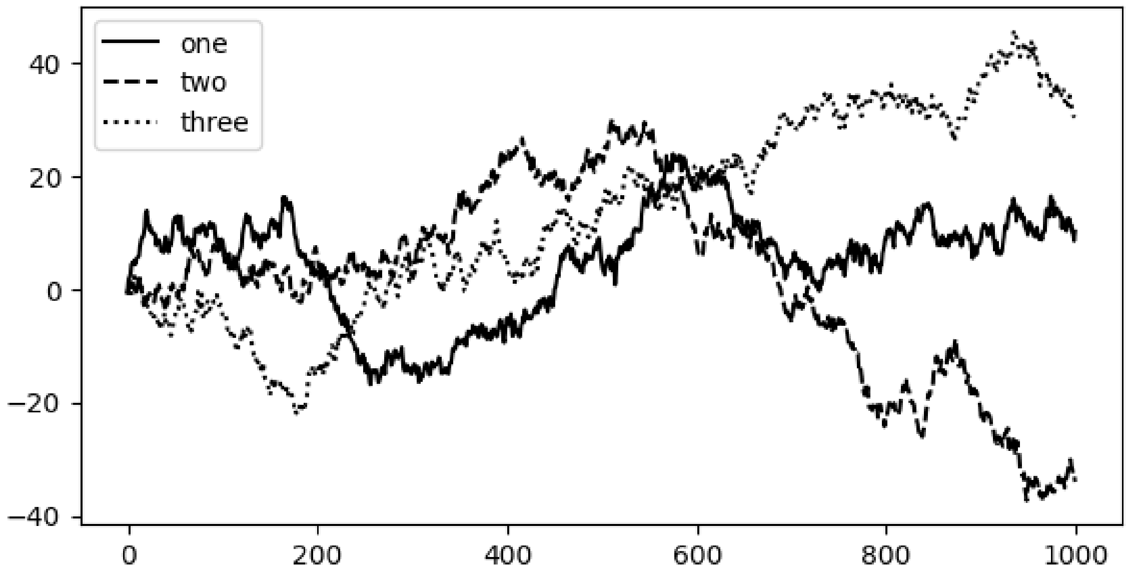
```
1554.41
```

```python
fig, ax = plt.subplots(figsize=(7, 3.5))

spx.plot(ax=ax, color="black");

for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor="black", headwidth=4, width=2,
                                headlength=4),
                horizontalalignment="left", verticalalignment="top")

# Zoom in on 2007-2010
ax.set_xlim(["1/1/2007", "1/1/2011"])
ax.set_ylim([600, 1800])

ax.set_title("Important dates in the 2008-2009 financial crisis");
```



10.1.2 Legends

```
fig, ax = plt.subplots(figsize=(7, 3.5))

ax.plot(np.random.randn(1000).cumsum(), color="black", label="one");
ax.plot(np.random.randn(1000).cumsum(), color="black", linestyle="dashed",label="two");
ax.plot(np.random.randn(1000).cumsum(), color="black", linestyle="dotted",label="three");
#I would expect a legend to be a lot of work
# I would have to specify which lines has which legend
ax.legend();
```



10.1.3 Math Notations in Notebook

If you wanted to show a sum using fancy math notations you would deploy **Latex**

$\alpha = \sum_{i=1}^{N} \sqrt{x_i}$

10.2 Pandas

10.2.1 Assign

We have already created new columns on the fly by the `[]` method. There is also a built-in method `assign()`

```
df = pd.DataFrame({'temp_c': [17.0, 25.0]},index=['Portland', 'Berkeley'])
df
```

	temp_c
Portland	17.0
Berkeley	25.0

create a new column which measures temperature in Fahrenheit $9/5 \cdot C + 32$

```
#method1
df["temp_f"] = 9/5 * df["temp_c"] + 32
df
```

	temp_c	temp_f
Portland	17.0	62.6
Berkeley	25.0	77.0

```
df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
```

	temp_c	temp_f
Portland	17.0	62.6
Berkeley	25.0	77.0

```
df.assign(temp_f = df['temp_c'] * 9 / 5 + 32,
          temp_k = df['temp_c'] + 273.15)
```

	temp_c	temp_f	temp_k
Portland	17.0	62.6	290.15

	temp_c	temp_f	temp_k
Berkeley	25.0	77.0	298.15

```
#What does lambda mean ?
def ConvertCtoF(x):
  return x['temp_c'] * 9 / 5 + 32

#ConvertCtoF(df)
df.assign(temp_f = ConvertCtoF(df))

#this is a lot of overhead to efine a function that does basically a one-liner
lambda x: x['temp_c'] * 9 / 5 + 32
```

	temp_c	temp_f
Portland	17.0	62.6
Berkeley	25.0	77.0

You can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:

```
df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
    temp_k=lambda x: (x['temp_f'] + 459.67) * 5 / 9)
```

	temp_c	temp_f	temp_k
Portland	17.0	62.6	290.15
Berkeley	25.0	77.0	298.15

10.2.2 Pivoting

In Pandas, the pivot table function takes simple data frame as input, and performs grouped operations that provides a multidimensional summary of the data.

You can also think of it as a **long to wide** translation.

```
stocks_long = pd.read_csv('https://gist.githubusercontent.com/alexdebrie/b3f40efc3dd7664df
stocks_long.head()
```

	date	symbol	open	high	low	close	volume
0	2019-03-01	AMZN	1655.13	1674.26	1651.00	1671.73	4974877
1	2019-03-04	AMZN	1685.00	1709.43	1674.36	1696.17	6167358
2	2019-03-05	AMZN	1702.95	1707.80	1689.01	1692.43	3681522
3	2019-03-06	AMZN	1695.97	1697.75	1668.28	1668.95	3996001
4	2019-03-07	AMZN	1667.37	1669.75	1620.51	1625.95	4957017

```
stocks_long["date"].unique()
#stocks_long["symbol"].unique()
stocks_long.shape
```

```
(15, 7)
```

The data has a number of columns and that the rows are organized by trading date and stock symbol.

That organization may be helpful for some analysis, but it can be hard to glean information about trading volume across dates and stock symbols. Let's reshape our data to look closer at volume.

```
stocks_wide = stocks_long.pivot(index='symbol', columns='date', values='volume')
print(stocks_wide.shape)
stocks_wide
```

```
(3, 5)
```

date	2019-03-01	2019-03-04	2019-03-05	2019-03-06	2019-03-07
symbol					
AAPL	25886167	27436203	19737419	20810384	24796374
AMZN	4974877	6167358	3681522	3996001	4957017
GOOG	1450316	1446047	1443174	1099289	1166559

How to use the Pandas pivot method

To use the pivot method in Pandas, you need to specify three parameters:

- Index: Which column should be used to identify and order your rows vertically

- Columns: Which column should be used to create the new columns in our reshaped DataFrame. Each unique value in the column stated here will create a column in our new DataFrame.
- Values: Which column(s) should be used to fill the values in the cells of our DataFrame.

In the example below, I use pivot to examine the closing trading price for each stock symbol over our trading window.

```
stocks_long.pivot(index='date', columns='symbol', values=['close', "volume"])
```

	close			volume		
symbol	AAPL	AMZN	GOOG	AAPL	AMZN	GOOG
date						
2019-03-01	174.97	1671.73	1140.99	25886167.0	4974877.0	1450316.0
2019-03-04	175.85	1696.17	1147.80	27436203.0	6167358.0	1446047.0
2019-03-05	175.53	1692.43	1162.03	19737419.0	3681522.0	1443174.0
2019-03-06	174.52	1668.95	1157.86	20810384.0	3996001.0	1099289.0
2019-03-07	172.50	1625.95	1143.30	24796374.0	4957017.0	1166559.0

10.2.3 Apply

As an example of dealing with time series in pandas, consider a financial dataset originally obtained from Yahoo! Finance containing end-of-day prices for a few stocks and the S&P 500 index (the `SPX` symbol).

Note that we activate the `parse_dates=True` option:

```
url="https://raw.githubusercontent.com/wesm/pydata-book/3rd-edition/examples/stock_px.csv"

close_px = pd.read_csv(url, parse_dates=True, index_col=0)
close_px.head()
```

	AAPL	MSFT	XOM	SPX
2003-01-02	7.40	21.11	29.22	909.03
2003-01-03	7.45	21.14	29.24	908.59
2003-01-06	7.45	21.52	29.96	929.01
2003-01-07	7.43	21.93	28.95	922.93
2003-01-08	7.28	21.31	28.83	909.93

Next, we compute percent change on `close_px` using the pandas function `pct_change`:

```
rets = close_px.pct_change().dropna()
rets
```

	AAPL	MSFT	XOM	SPX
2003-01-03	0.006757	0.001421	0.000684	-0.000484
2003-01-06	0.000000	0.017975	0.024624	0.022474
2003-01-07	-0.002685	0.019052	-0.033712	-0.006545
2003-01-08	-0.020188	-0.028272	-0.004145	-0.014086
2003-01-09	0.008242	0.029094	0.021159	0.019386
...	...	...	...	...
2011-10-10	0.051406	0.026286	0.036977	0.034125
2011-10-11	0.029526	0.002227	-0.000131	0.000544
2011-10-12	0.004747	-0.001481	0.011669	0.009795
2011-10-13	0.015515	0.008160	-0.010238	-0.002974
2011-10-14	0.033225	0.003311	0.022784	0.017380

We can apply a function along an axis of the DataFrame:

```
def spx_corr(df):
   return df.corr(rets["SPX"])

rets.apply(spx_corr)
```

```
AAPL    0.564474
MSFT    0.714763
XOM     0.764643
SPX     1.000000
dtype: float64
```

10.3 Smoking Data

3 columns: “outcome” measures whether the person is still alive after 10 years.

We want to glean the effect of smoking on survival probability.

```
df = pd.read_csv("https://calmcode.io/datasets/smoking.csv")
df.head()
```

	outcome	smoker	age
0	Alive	Yes	23
1	Alive	Yes	18
2	Dead	Yes	71
3	Alive	No	67
4	Alive	No	64

```
df.shape
```

```
(1314, 3)
```

```
df.mean()
```

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

```
age    46.920091
dtype: float64
```

```
df = df.assign(alive = (df['outcome'] == 'Alive').astype(int),
               smokes = (df['smoker'] == 'Yes').astype(int))
df.head()
```

	outcome	smoker	age	alive	smokes
0	Alive	Yes	23	1	1
1	Alive	Yes	18	1	1
2	Dead	Yes	71	0	1
3	Alive	No	67	1	0
4	Alive	No	64	1	0

```
df.mean()
#71.9% is  is the overall survival rate for all people
```

```
#How would you separately compute this for smokers and non smokers ?
#groupby is your friend !!
```

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

```
age       46.920091
alive      0.719178
smokes     0.442922
dtype: float64
```

Smoking is good for your health

```
df.groupby(['smoker']).alive.mean()
```

```
smoker
No     0.685792
Yes    0.761168
Name: alive, dtype: float64
```

```
df.groupby(['smoker']).agg(prob=('alive', np.mean))
```

	prob
smoker	
No	0.685792
Yes	0.761168

Age adjustment

```
df = df.assign(age10 =np.round(df['age'] / 10) * 10)
```

```
(df.groupby(['age10'])
  .agg(p_alive=('alive', np.mean)
   ,p_smokes=('smokes', np.mean)))
```

	p_alive	p_smokes
age10		
20.0	0.980645	0.445161
30.0	0.972332	0.438735
40.0	0.900000	0.495833
50.0	0.821622	0.632432
60.0	0.587302	0.464286
70.0	0.203947	0.236842
80.0	0.000000	0.168831

```python
from datetime import datetime

datetime(2018, 6, 24)
```

```
datetime.datetime(2018, 6, 24, 0, 0)
```

Group by age and smoking status and plot the survival rates separately for the two groups

---

Further Reading:

-

11 Dates and Times

```python
from datetime import datetime, timedelta
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```python
now = datetime.now()
now
```

```
datetime.datetime(2024, 6, 18, 15, 11, 39, 903326)
```

```python
now.year, now.month, now.day
```

```
(2024, 6, 18)
```

```python
delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)
delta
```

```
datetime.timedelta(days=926, seconds=56700)
```

```python
start = datetime(2011, 1, 7)

start + timedelta(12)
```

```
datetime.datetime(2011, 1, 19, 0, 0)
```

Formatting datetime objects:

```python
str(start)
```

```
'2011-01-07 00:00:00'
```

Table 11.1: **?(caption)**

Table 11.2: datetime format specification (ISO C89 compatible)

Type	Description
%Y	Four-digit year
%y	Two-digit year
%m	Two-digit month [01, 12]
%d	Two-digit day [01, 31]
%H	

```
now.strftime("%Y-%m-%d")
```

```
'2024-06-18'
```

`datetime` objects also have a number of locale-specific formatting options for systems in other countries or languages. For example, the abbreviated month names will be different on German or French systems compared with English systems. See Table 11.3 for a listing.

```
now.strftime("%A, %b, %p")
```

```
'Tuesday, Jun, PM'
```

11.0.0.1 Generating Date Ranges

```
dates = pd.date_range("2012-03-09 09:30", periods=6)

ts = pd.Series(np.random.standard_normal(len(dates)), index=dates)
ts
```

```
2012-03-09 09:30:00   -0.003219
2012-03-10 09:30:00    0.595169
2012-03-11 09:30:00    0.299114
2012-03-12 09:30:00   -1.711842
2012-03-13 09:30:00   -0.321497
2012-03-14 09:30:00    0.860591
Freq: D, dtype: float64
```

11.0.1 Time Zones

Working with time zones can be one of the most unpleasant parts of time series manipulation. As a result, many time series users choose to work with time series in coordinated universal time or UTC, which is the geography-independent international standard. Time zones are expressed as offsets from UTC; for example, New York is four hours behind UTC during daylight saving time (DST) and five hours behind the rest of the year.

In Python, time zone information comes from the third-party `pytz` library (installable with pip or conda), which exposes the Olson database, a compilation of world time zone information. This is especially important for historical data because the DST transition dates (and even

Table 11.2: **?(caption)**

Table 11.3: Locale-specific date formatting

Type	Description
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	

UTC offsets) have been changed numerous times depending on the regional laws. In the United States, the DST transition times have been changed many times since 1900!

For detailed information about the pytz library, you'll need to look at that library's documentation. pandas wraps pytz's functionality so you can ignore its API outside of the time zone names. Since pandas has a hard dependency on pytz, it isn't necessary to install it separately. Time zone names can be found interactively and in the docs:

11.0.1.0.1 Time Zone Localization and Conversion

By default, time series in pandas are time zone agnostic.

```
print(ts.index.tz)
```

```
None
```

Date ranges can be generated with a time zone set:

```
pd.date_range("2012-03-09 09:30", periods=10, tz="UTC")
```

Conversion from naive to localized (reinterpreted as having been observed in a particular time zone) is handled by the `tz_localize` method:

```
ts_utc = ts.tz_localize("UTC")
ts_utc
```

```
2012-03-09 09:30:00+00:00   -0.003219
2012-03-10 09:30:00+00:00    0.595169
2012-03-11 09:30:00+00:00    0.299114
2012-03-12 09:30:00+00:00   -1.711842
2012-03-13 09:30:00+00:00   -0.321497
2012-03-14 09:30:00+00:00    0.860591
Freq: D, dtype: float64
```

Once a time series has been localized to a particular time zone, it can be converted to another time zone with `tz_convert`:

```
ts_utc.tz_convert("America/New_York")
```

```
2012-03-09 04:30:00-05:00   -0.003219
2012-03-10 04:30:00-05:00    0.595169
2012-03-11 05:30:00-04:00    0.299114
2012-03-12 05:30:00-04:00   -1.711842
2012-03-13 05:30:00-04:00   -0.321497
2012-03-14 05:30:00-04:00    0.860591
Freq: D, dtype: float64
```