# Data handling in Python

*Thursday 19, September*

## Contents



### 1. Introduction

Python is a simple programming language that would be rather unexciting by itself. For researchers, its appeal comes from the vast and endlessly growing collection of packages with which you can do pretty much anything you want. Packages, libraries or modules (these terms are synonyms) are user-written lists of Python functions helping you do whatever you may want to achieve without requiring you to code your own functions.

There are 4 essential modules for empirical quantitative projects: **NumPy**, **SciPy**, **pandas** and **matplotlib**. they all serve different purposes.

- **NumPy**, for mathematical functions and scientific computing
- **SciPy**, for more advanced mathematical computations
- **pandas** (shorthand for "panel data") for data manipulation
- **matplotlib**, for graphs

Beyond these common modules, , we will use two other Python packages later in the course: **statsmodels** and **scikit-learn**.

- **statsmodels** provides functions for the estimation of most statistical models, statistical tests and descriptive statistics social scientists may be interested in. It generates outputs that are similar to Stata and R's. **statsmodels** is heavily dependent on your data being structured with **panda** dataframes.
- **scikit-learn** offers off-the-shelf functions for data mining and machine learning. It builds on **NumPy**, **SciPy** and **matplotlib**. Classification or supervised learning, clustering or unsupervised learning, dimensionality reduction and model selection can all be performed with **scikit-learn**.

**To summarise.** In the beginning of each Notebook or Python script, you will need to import the modules you will use. It is helpful to give short names to the modules in order to call them more easily later. To import and name modules, just follow use the syntax shown below

In [1]:

```
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib as mp

import statsmodels as sm
import sklearn as sl
```

You can call the function `help()` to get more information about a function. For instance, if we need explanations about NumPy's random number generator, we run

In [2]:

```
help(np.random.rand)
```

```
Help on built-in function rand:

rand(...) method of mtrand.RandomState instance
    rand(d0, d1, ..., dn)

    Random values in a given shape.

    Create an array of the given shape and populate it with
    random samples from a uniform distribution
    over ``[0, 1)``.

    Parameters
    ----------
    d0, d1, ..., dn : int, optional
        The dimensions of the returned array, should all be positive.
        If no argument is given a single Python float is returned.

    Returns
    -------
    out : ndarray, shape ``(d0, d1, ..., dn)``
        Random values.

    See Also
    --------
    random

    Notes
    -----
    This is a convenience function. If you want an interface that
    takes a shape-tuple as the first argument, refer to
    np.random.random_sample .

    Examples
    --------
    >>> np.random.rand(3,2)
    array([[ 0.14022471,  0.96360618],  #random
           [ 0.37601032,  0.25528411],  #random
           [ 0.49313049,  0.94909878]]) #random
```

To know all the functions of a module, write the module name followed by a `.` , then press `TAB` . An autocomplete menu will appear. You can do the same with a function, and the menu will suggest all the different options of that function.

In [ ]:

```
np.
np.random.
```

## 2. Numpy

See the "Introduction to Python" notebook.

## 3. Panda

We will manipulate two objects when doing research with structured data in Pandas: **Series** and **Dataframes**.

#### 3.1. Series

##### *3.1.1. Basics*

Series are lists of integers, floats, booleans or strings. By default, the values are indexed by integers starting from 0. MATLAB users beware , indexation does not start with 1 here. But you can define your own index scheme by adding `index=list('abcdef')` as an option to `Series()` . We define a few series below to illustrate their properties.

In [3]:

```
S1 = pd.Series([1, 1, 2, 3, 4, 5])
S2 = pd.Series([0, 0.2 , 0.4, 0.6, 0.8, 1])
S3 = pd.Series([1, 0.8, 0.6, 0.4, 0.2, 0] ,index=list('abcdef'))
S4 = pd.Series(['hello','world','byebye'])
S5 = pd.Series([True, False, True, False])
S6 = pd.Series([0, 0.2, 'hello', True])
```

You can see the content and the indices of a series by typing their names and running the code.

In [4]:

```
S1
```

Out[4]:

```
0    1
1    1
2    2
3    3
4    4
5    5
dtype: int64
```

In [5]:

```
S2, S3, S4, S5, S6
```

Out[5]:

```
(0    0.0
 1    0.2
 2    0.4
 3    0.6
 4    0.8
 5    1.0
 dtype: float64, a    1.0
 b    0.8
 c    0.6
 d    0.4
 e    0.2
 f    0.0
 dtype: float64, 0     hello
 1     world
 2    byebye
 dtype: object, 0     True
 1    False
 2     True
 3    False
 dtype: bool, 0        0
 1      0.2
 2    hello
 3     True
 dtype: object)
```

#### *3.1.2. Indexing*

You can get access to each value of a series separately. You can also extract a section of the series through a selection of their indices, or if they satisfy a given condition. See examples below.

In [6]:

```
S4.values       # display all the values of S4
S3.index        # display indices of S3
S5[0]           # first value of S5
S5[:]           # all values of S5
S5[1:3]         # 3 first values of  S5
S2[[5, 3, 1]]   # values of S2 indexed by 5, 3 and 1
S3[['f', 'b']]  # value of S3 indexed by g and b
S3 > 0.5        # returns a series of booleans, resulting from evaluating the condition
at each value of S3
S3[S3 > 0.5]    # only returns the values of S3 satisfying the condition
```

Out[6]:

```
a    1.0
b    0.8
c    0.6
dtype: float64
```

### *3.1.3. Functions of Series*

Some Series function can be useful. See below.

In [7]:

```
S1.size            # returns the number of values (scalar)
S1.prod()          # product of all values (scalar)
S1.sum()           # sum of all values (scalar)
S1.cumsum()        # sumulative sum (vector)
S1.max()           # maximum (scalar)
S1.idxmax()        # maximum index (scalar)
S2.round()         # series rounded to the nearest integer (float vector)
np.ceil(S2)        # series rounded up (float vector)
np.floor(S2)       # series rounded down (float vector)
S1.unique()        # series of unique values
S3.sort_values()   # values sorted in ascending order
S1.sort_index(ascending=False)  # sort in descending order of indices
S1.isin([1,3,5,7,9])            # returns series of booleans equal to "True" if the val
ue is in the list of values provided
```

Out[7]:

```
0     True
1     True
2    False
3     True
4    False
5     True
dtype: bool
```

### *3.1.4. Missing values*

In Pandas, a missing value is coded `NaN` for "Not a Number". It is also the result given by Python after a forbidden mathematical operation such as

In [8]:

```
np.sqrt(-2)
```

```
C:\Users\dyevre\AppData\Local\Continuum\anaconda3\lib\site-packages\ipyker
nel_launcher.py:1: RuntimeWarning: invalid value encountered in sqrt
  """Entry point for launching an IPython kernel.
```

Out[8]:

```
nan
```

The count function does not count `NaN` as values. Let's see how, by replacing a value in a series by a forbidden mathematical operation.

In [9]:

```
S1.count()
```

Out[9]:

```
6
```

In [10]:

```
S1[1] = np.inf - np.inf
S1.count()
```

Out[10]:

```
5
```

In [11]:

```
S1
```

Out[11]:

```
0    1.0
1    NaN
2    2.0
3    3.0
4    4.0
5    5.0
dtype: float64
```

Note that the `size` function still counts 6 elements as it returns the number of missing and non-missing values.

The function `isnull()` returns a series of boolean equals to `True` if the value of the original series is not `NaN` and `False` otherwise.

In [12]:

```
S1.isnull()
```

Out[12]:

```
0    False
1     True
2    False
3    False
4    False
5    False
dtype: bool
```

When cleaning data in Python, we can delete all the `NaN` of a Series with the `dropna` function. This delete the missing values, yet it keep the old indexation.

In [13]:

```
S1 = S1.dropna()
S1
```

Out[13]:

```
0    1.0
2    2.0
3    3.0
4    4.0
5    5.0
dtype: float64
```

### 3.2. Data frames

#### *3.2.1. Basics*

In Pandas, a data frame is simply a table of structured data, with variables as columns and observations as rows. Each column can have a **label** that we can simply assimilate to a variable name here. By default, observations are indexed by integers, starting from 0, but we can re-index observation at will.

We create below a data frame of 10 observations of 3 random numbers from the uniform distribution in $[0, 1)$

In [14]:

```
randomTable = pd.DataFrame(np.random.rand(10,3), columns = pd.Series(["Number 1", "Numb
er 2", "Number 3"]) )
randomTable
```

Out[14]:

| | Number 1 | Number 2 | Number 3 |
|---|---|---|---|
| **0** | 0.943727 | 0.979628 | 0.781167 |
| **1** | 0.234868 | 0.009590 | 0.339131 |
| **2** | 0.395244 | 0.262144 | 0.534559 |
| **3** | 0.543150 | 0.294868 | 0.768259 |
| **4** | 0.979284 | 0.159358 | 0.842265 |
| **5** | 0.761091 | 0.000773 | 0.644708 |
| **6** | 0.214362 | 0.612162 | 0.988245 |
| **7** | 0.241826 | 0.166216 | 0.318080 |
| **8** | 0.182792 | 0.298934 | 0.274575 |
| **9** | 0.203176 | 0.941243 | 0.520999 |

#### *3.2.2. Indexing*

We can index the rows differently, for instance by making them start from 1. To do so, we define a Pandas Series and define it as the index.

In [15]:

```
obs = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
obs
```

Out[15]:

```
0     1
1     2
2     3
3     4
4     5
5     6
6     7
7     8
8     9
9    10
dtype: int64
```

In [16]:

```
randomTable = randomTable.set_index(obs)
randomTable
```

Out[16]:

| | Number 1 | Number 2 | Number 3 |
|---|---|---|---|
| **1** | 0.943727 | 0.979628 | 0.781167 |
| **2** | 0.234868 | 0.009590 | 0.339131 |
| **3** | 0.395244 | 0.262144 | 0.534559 |
| **4** | 0.543150 | 0.294868 | 0.768259 |
| **5** | 0.979284 | 0.159358 | 0.842265 |
| **6** | 0.761091 | 0.000773 | 0.644708 |
| **7** | 0.214362 | 0.612162 | 0.988245 |
| **8** | 0.241826 | 0.166216 | 0.318080 |
| **9** | 0.182792 | 0.298934 | 0.274575 |
| **10** | 0.203176 | 0.941243 | 0.520999 |

### *3.2.3. Column names*

We can add variable names by creating a list.

In [17]:

```
columnNames = pd.Series(["A", "B", "C"])
randomTable.columns = columnNames
randomTable
```

Out[17]:

| | A | B | C |
|---|---|---|---|
| **1** | 0.943727 | 0.979628 | 0.781167 |
| **2** | 0.234868 | 0.009590 | 0.339131 |
| **3** | 0.395244 | 0.262144 | 0.534559 |
| **4** | 0.543150 | 0.294868 | 0.768259 |
| **5** | 0.979284 | 0.159358 | 0.842265 |
| **6** | 0.761091 | 0.000773 | 0.644708 |
| **7** | 0.214362 | 0.612162 | 0.988245 |
| **8** | 0.241826 | 0.166216 | 0.318080 |
| **9** | 0.182792 | 0.298934 | 0.274575 |
| **10** | 0.203176 | 0.941243 | 0.520999 |

Let's rename variables separately, just like with STATA's `rename` function. Here we are renaming variables using a mapping from old names to new names. This mapping is called a **dictionary**; it is the way Python associate a **key** --like a word in a dictionary-- to a **value** --the definition of the word. The basic structure of a dictionary is `{"key 1": "value 1", "key 2": "value 2", ... , "key N": "value N"}`

In [18]:

```
randomTable = randomTable.rename(columns={"A": "a"})
randomTable
```

Out[18]:

| | a | B | C |
|---|---|---|---|
| **1** | 0.943727 | 0.979628 | 0.781167 |
| **2** | 0.234868 | 0.009590 | 0.339131 |
| **3** | 0.395244 | 0.262144 | 0.534559 |
| **4** | 0.543150 | 0.294868 | 0.768259 |
| **5** | 0.979284 | 0.159358 | 0.842265 |
| **6** | 0.761091 | 0.000773 | 0.644708 |
| **7** | 0.214362 | 0.612162 | 0.988245 |
| **8** | 0.241826 | 0.166216 | 0.318080 |
| **9** | 0.182792 | 0.298934 | 0.274575 |
| **10** | 0.203176 | 0.941243 | 0.520999 |

Similarly, we can rename several columns at once, just by defining a bigger dictionary.

In [19]:

```
randomTable = randomTable.rename(columns={"B": "b", "C": "c"})
randomTable
```

Out[19]:

| | a | b | c |
|---|---|---|---|
| **1** | 0.943727 | 0.979628 | 0.781167 |
| **2** | 0.234868 | 0.009590 | 0.339131 |
| **3** | 0.395244 | 0.262144 | 0.534559 |
| **4** | 0.543150 | 0.294868 | 0.768259 |
| **5** | 0.979284 | 0.159358 | 0.842265 |
| **6** | 0.761091 | 0.000773 | 0.644708 |
| **7** | 0.214362 | 0.612162 | 0.988245 |
| **8** | 0.241826 | 0.166216 | 0.318080 |
| **9** | 0.182792 | 0.298934 | 0.274575 |
| **10** | 0.203176 | 0.941243 | 0.520999 |

We can rename indices similarly:

In [20]:

```
randomTable = randomTable.rename(index={1: "One", 2: "Two"})
randomTable
```

Out[20]:

| | a | b | c |
|---|---|---|---|
| **One** | 0.943727 | 0.979628 | 0.781167 |
| **Two** | 0.234868 | 0.009590 | 0.339131 |
| **3** | 0.395244 | 0.262144 | 0.534559 |
| **4** | 0.543150 | 0.294868 | 0.768259 |
| **5** | 0.979284 | 0.159358 | 0.842265 |
| **6** | 0.761091 | 0.000773 | 0.644708 |
| **7** | 0.214362 | 0.612162 | 0.988245 |
| **8** | 0.241826 | 0.166216 | 0.318080 |
| **9** | 0.182792 | 0.298934 | 0.274575 |
| **10** | 0.203176 | 0.941243 | 0.520999 |

We can define data frames more explicitely, by defining each variable as a label and a series.

In [23]:

```
syllabus = pd.DataFrame({ 'Date' : pd.date_range('20190916', '20190927'),   # creates a series of dates from 16/09/2019 to 27/09/2019
    'Topic' : pd.Categorical(["Version control","Cloud computing","Intro to Python","Basics of data handling","OLS, GLS, IV and NLLS","WEEKEND","WEEKEND","MLE","Time Series","-No class-","Intro to machine learning 1","Intro to machine learning 2"]), # string variable of topics
    'Alphabetical index' : list('abcdefghijkl'),                                # another index, defined as a list
    'Numerical index' : np.linspace(1, 12, 12)   })                            # another index, defined with a useful NumPy function

syllabus
```

Out[23]:

| | Date | Topic | Alphabetical index | Numerical index |
|---|---|---|---|---|
| **0** | 2019-09-16 | Version control | a | 1.0 |
| **1** | 2019-09-17 | Cloud computing | b | 2.0 |
| **2** | 2019-09-18 | Intro to Python | c | 3.0 |
| **3** | 2019-09-19 | Basics of data handling | d | 4.0 |
| **4** | 2019-09-20 | OLS, GLS, IV and NLLS | e | 5.0 |
| **5** | 2019-09-21 | WEEKEND | f | 6.0 |
| **6** | 2019-09-22 | WEEKEND | g | 7.0 |
| **7** | 2019-09-23 | MLE | h | 8.0 |
| **8** | 2019-09-24 | Time Series | i | 9.0 |
| **9** | 2019-09-25 | -No class- | j | 10.0 |
| **10** | 2019-09-26 | Intro to machine learning 1 | k | 11.0 |
| **11** | 2019-09-27 | Intro to machine learning 2 | l | 12.0 |

#### *3.2.4. Functions on data frames*

See below some useful data frame functions:

In [24]:

```
randomTable.shape              # To get the dimensions of the data frame
list(randomTable)              # To get a series containing the variable names
randomTable.columns            # To get a series containing the variable names and thei
r data type
randomTable.index              # To get a series containing the indices and their data
 types
syllabus.dtypes                # To get the types of data objects in the table
syllabus.head(4)               # To display the first 4 obs of the frame (5 by default
 if no argument entered in head() )
syllabus.tail(3)               # To display the last 3 obs of the fram (5 by default)
syllabus.values                # To get all values of the dataframe as an array of arra
ys (basically an array of rows)
randomTable.describe()         # generates summary statistics about the valiables
```

Out[24]:

| | a | b | c |
|---|---|---|---|
| **count** | 10.000000 | 10.000000 | 10.000000 |
| **mean** | 0.469952 | 0.372492 | 0.601199 |
| **std** | 0.317253 | 0.354494 | 0.244154 |
| **min** | 0.182792 | 0.000773 | 0.274575 |
| **25%** | 0.219489 | 0.161072 | 0.384598 |
| **50%** | 0.318535 | 0.278506 | 0.589634 |
| **75%** | 0.706606 | 0.533855 | 0.777940 |
| **max** | 0.979284 | 0.979628 | 0.988245 |

#### *3.2.5. Acessing elements of a data frame*

To extract columns from dataframe, we simply use:

In [25]:

```
randomTable[['a', 'b']]
```

Out[25]:

| | a | b |
|---|---|---|
| **One** | 0.943727 | 0.979628 |
| **Two** | 0.234868 | 0.009590 |
| **3** | 0.395244 | 0.262144 |
| **4** | 0.543150 | 0.294868 |
| **5** | 0.979284 | 0.159358 |
| **6** | 0.761091 | 0.000773 |
| **7** | 0.214362 | 0.612162 |
| **8** | 0.241826 | 0.166216 |
| **9** | 0.182792 | 0.298934 |
| **10** | 0.203176 | 0.941243 |

To extract observations, we can use row numbers, indices and conditions on values.

In [26]:

```
randomTable[0:2]
randomTable.index
```

Out[26]:

```
Index(['One', 'Two', 3, 4, 5, 6, 7, 8, 9, 10], dtype='object')
```

In [27]:

```
randomTable['One': 'Two']
```

Out[27]:

| | a | b | c |
|---|---|---|---|
| **One** | 0.943727 | 0.979628 | 0.781167 |
| **Two** | 0.234868 | 0.009590 | 0.339131 |

In [28]:

```
randomTable[randomTable['a'] >= 0.5]   # Extracts all observations whose a is greater t
han, or equal to 0.5
```

Out[28]:

| | a | b | c |
|---|---|---|---|
| **One** | 0.943727 | 0.979628 | 0.781167 |
| **4** | 0.543150 | 0.294868 | 0.768259 |
| **5** | 0.979284 | 0.159358 | 0.842265 |
| **6** | 0.761091 | 0.000773 | 0.644708 |

More sophisticated selections, combining conditions on rows and columns, can be extracted from a dataframe with the **.loc** and **.iloc** functions.

- **.loc** takes as arguments indices for rows and labels for columns
- **.iloc** takes as arguments integers for rows and columns

In [29]:

```
randomTable.loc['One': 'Two', ['a', 'b']]
randomTable.iloc[0: 4, 0: 2]
```

Out[29]:

| | a | b |
|---|---|---|
| **One** | 0.943727 | 0.979628 |
| **Two** | 0.234868 | 0.009590 |
| **3** | 0.395244 | 0.262144 |
| **4** | 0.543150 | 0.294868 |

These selectors also accept boolean tests.

In [30]:

```
randomTable.loc[randomTable['a'] >= 0.5, ['b', 'c']]
```

Out[30]:

| | b | c |
|---|---|---|
| **One** | 0.979628 | 0.781167 |
| **4** | 0.294868 | 0.768259 |
| **5** | 0.159358 | 0.842265 |
| **6** | 0.000773 | 0.644708 |

In [31]:

```
syllabus.loc[syllabus['Topic'] == 'WEEKEND', ['Date']]
```

Out[31]:

| | Date |
|---|---|
| **5** | 2019-09-21 |
| **6** | 2019-09-22 |

#### *3.2.6. Adding data to a data frame*

We can add a column and populate it with a given value:

In [32]:

```
randomTable['d'] = np.pi      # adds a column with the value of pi
randomTable
```

Out[32]:

| | a | b | c | d |
|---|---|---|---|---|
| **One** | 0.943727 | 0.979628 | 0.781167 | 3.141593 |
| **Two** | 0.234868 | 0.009590 | 0.339131 | 3.141593 |
| **3** | 0.395244 | 0.262144 | 0.534559 | 3.141593 |
| **4** | 0.543150 | 0.294868 | 0.768259 | 3.141593 |
| **5** | 0.979284 | 0.159358 | 0.842265 | 3.141593 |
| **6** | 0.761091 | 0.000773 | 0.644708 | 3.141593 |
| **7** | 0.214362 | 0.612162 | 0.988245 | 3.141593 |
| **8** | 0.241826 | 0.166216 | 0.318080 | 3.141593 |
| **9** | 0.182792 | 0.298934 | 0.274575 | 3.141593 |
| **10** | 0.203176 | 0.941243 | 0.520999 | 3.141593 |

In [33]:

```
randomTable['e'] = np.linspace(1, 10, 10)  # adds a 'e' column with floats randging fro
m 1 to 10, separated by increments of 1.0
randomTable
```

Out[33]:

| | a | b | c | d | e |
|---|---|---|---|---|---|
| **One** | 0.943727 | 0.979628 | 0.781167 | 3.141593 | 1.0 |
| **Two** | 0.234868 | 0.009590 | 0.339131 | 3.141593 | 2.0 |
| **3** | 0.395244 | 0.262144 | 0.534559 | 3.141593 | 3.0 |
| **4** | 0.543150 | 0.294868 | 0.768259 | 3.141593 | 4.0 |
| **5** | 0.979284 | 0.159358 | 0.842265 | 3.141593 | 5.0 |
| **6** | 0.761091 | 0.000773 | 0.644708 | 3.141593 | 6.0 |
| **7** | 0.214362 | 0.612162 | 0.988245 | 3.141593 | 7.0 |
| **8** | 0.241826 | 0.166216 | 0.318080 | 3.141593 | 8.0 |
| **9** | 0.182792 | 0.298934 | 0.274575 | 3.141593 | 9.0 |
| **10** | 0.203176 | 0.941243 | 0.520999 | 3.141593 | 10.0 |

### *3.2.7. Sorting values in a data frame*

We can sort the table by the values of a variable or of several variables.

In [34]:

```
randomTable.sort_values(by= 'a')
randomTable.sort_values(by= ['a', 'e'])
```

Out[34]:

| | a | b | c | d | e |
|---|---|---|---|---|---|
| **9** | 0.182792 | 0.298934 | 0.274575 | 3.141593 | 9.0 |
| **10** | 0.203176 | 0.941243 | 0.520999 | 3.141593 | 10.0 |
| **7** | 0.214362 | 0.612162 | 0.988245 | 3.141593 | 7.0 |
| **Two** | 0.234868 | 0.009590 | 0.339131 | 3.141593 | 2.0 |
| **8** | 0.241826 | 0.166216 | 0.318080 | 3.141593 | 8.0 |
| **3** | 0.395244 | 0.262144 | 0.534559 | 3.141593 | 3.0 |
| **4** | 0.543150 | 0.294868 | 0.768259 | 3.141593 | 4.0 |
| **6** | 0.761091 | 0.000773 | 0.644708 | 3.141593 | 6.0 |
| **One** | 0.943727 | 0.979628 | 0.781167 | 3.141593 | 1.0 |
| **5** | 0.979284 | 0.159358 | 0.842265 | 3.141593 | 5.0 |

We can also sort by index of observation ( `axis=0` ), or by index of variable ( `axis=1` ).

In [35]:

```
syllabus.sort_index(axis=0, ascending=False)
```

Out[35]:

| | Date | Topic | Alphabetical index | Numerical index |
|---|---|---|---|---|
| **11** | 2019-09-27 | Intro to machine learning 2 | l | 12.0 |
| **10** | 2019-09-26 | Intro to machine learning 1 | k | 11.0 |
| **9** | 2019-09-25 | -No class- | j | 10.0 |
| **8** | 2019-09-24 | Time Series | i | 9.0 |
| **7** | 2019-09-23 | MLE | h | 8.0 |
| **6** | 2019-09-22 | WEEKEND | g | 7.0 |
| **5** | 2019-09-21 | WEEKEND | f | 6.0 |
| **4** | 2019-09-20 | OLS, GLS, IV and NLLS | e | 5.0 |
| **3** | 2019-09-19 | Basics of data handling | d | 4.0 |
| **2** | 2019-09-18 | Intro to Python | c | 3.0 |
| **1** | 2019-09-17 | Cloud computing | b | 2.0 |
| **0** | 2019-09-16 | Version control | a | 1.0 |

#### *3.2.7. Importing a structured dataset*

Pandas can read most structured data formats: .csv, Excel and SQL being the most common. We use the function `pd.read_csv` to import a CSV file into a data frame. We can indicate which delimiter to use, if the data is compressed or not, and if Python needs to skip a header in the original file. See below two imports we will use in the next section.

In [37]:

```
top1Percent = pd.read_excel('http://gabriel-zucman.eu/files/Zucman2019Data.xlsx', sheet
_name='DataF4', header=2, squeeze=True)
top10Percent = pd.read_excel('http://gabriel-zucman.eu/files/Zucman2019Data.xlsx', shee
t_name='DataF5', header=2, squeeze=True)
```

In [38]:

```
top10Percent
```

Out[38]:

| | Year | China | France | Russian Federation | United Kingdom | USA |
|---|---|---|---|---|---|---|
| **0** | 1910 | NaN | NaN | NaN | NaN | NaN |
| **1** | 1911 | NaN | NaN | NaN | NaN | NaN |
| **2** | 1912 | NaN | NaN | NaN | NaN | NaN |
| **3** | 1913 | NaN | 0.849030 | NaN | 0.925733 | NaN |
| **4** | 1914 | NaN | 0.849074 | NaN | 0.929655 | NaN |
| **5** | 1915 | NaN | 0.843429 | NaN | NaN | NaN |
| **6** | 1916 | NaN | 0.843037 | NaN | NaN | NaN |
| **7** | 1917 | NaN | 0.842252 | NaN | NaN | 0.782449 |
| **8** | 1918 | NaN | 0.838413 | NaN | NaN | 0.785049 |
| **9** | 1919 | NaN | 0.833341 | NaN | 0.885341 | 0.800170 |
| **10** | 1920 | NaN | 0.822932 | NaN | 0.879738 | 0.780308 |
| **11** | 1921 | NaN | 0.815696 | NaN | 0.881781 | 0.779401 |
| **12** | 1922 | NaN | 0.809572 | NaN | 0.888246 | 0.791782 |
| **13** | 1923 | NaN | 0.804844 | NaN | 0.883304 | 0.796282 |
| **14** | 1924 | NaN | 0.803360 | NaN | 0.879293 | 0.810724 |
| **15** | 1925 | NaN | 0.786832 | NaN | 0.881648 | 0.821536 |
| **16** | 1926 | NaN | 0.787089 | NaN | 0.872117 | 0.830990 |
| **17** | 1927 | NaN | 0.798049 | NaN | 0.879828 | 0.841124 |
| **18** | 1928 | NaN | NaN | NaN | 0.866827 | 0.843926 |
| **19** | 1929 | NaN | 0.802657 | NaN | 0.870702 | 0.843327 |
| **20** | 1930 | NaN | 0.802256 | NaN | 0.861311 | 0.845685 |
| **21** | 1931 | NaN | 0.787573 | NaN | 0.858074 | 0.843043 |
| **22** | 1932 | NaN | 0.779655 | NaN | 0.857418 | 0.847409 |
| **23** | 1933 | NaN | 0.781155 | NaN | 0.864071 | 0.845613 |
| **24** | 1934 | NaN | NaN | NaN | 0.861166 | 0.829752 |
| **25** | 1935 | NaN | 0.772239 | NaN | 0.858730 | 0.816211 |
| **26** | 1936 | NaN | 0.766867 | NaN | 0.851632 | 0.821281 |
| **27** | 1937 | NaN | 0.763813 | NaN | 0.854700 | 0.802343 |
| **28** | 1938 | NaN | 0.747334 | NaN | 0.850125 | 0.799010 |
| **29** | 1939 | NaN | 0.755728 | NaN | 0.842894 | 0.802181 |
| **...** | ... | ... | ... | ... | ... | ... |
| **76** | 1986 | NaN | 0.505658 | NaN | 0.488240 | 0.606497 |
| **77** | 1987 | NaN | 0.504989 | NaN | 0.503588 | 0.615782 |
| **78** | 1988 | NaN | 0.504901 | NaN | 0.481854 | 0.627376 |
| **79** | 1989 | NaN | 0.507558 | NaN | 0.485264 | 0.627007 |
| **80** | 1990 | NaN | 0.502717 | NaN | 0.459857 | 0.628830 |
| **81** | 1991 | NaN | 0.506542 | NaN | 0.455891 | 0.627435 |

| | Year | China | France | Russian Federation | United Kingdom | USA |
|---|---|---|---|---|---|---|
| **82** | 1992 | NaN | 0.510053 | NaN | 0.479958 | 0.642536 |
| **83** | 1993 | NaN | 0.512132 | NaN | 0.498296 | 0.645715 |
| **84** | 1994 | NaN | 0.511994 | NaN | 0.495453 | 0.646331 |
| **85** | 1995 | 0.408106 | 0.511167 | 0.525537 | 0.469170 | 0.650016 |
| **86** | 1996 | 0.430038 | 0.540069 | 0.544091 | 0.483788 | 0.654424 |
| **87** | 1997 | 0.446414 | 0.552385 | 0.595655 | 0.515730 | 0.659853 |
| **88** | 1998 | 0.459108 | 0.563284 | 0.624042 | 0.518868 | 0.668028 |
| **89** | 1999 | 0.469236 | 0.568759 | 0.657416 | 0.500720 | 0.670313 |
| **90** | 2000 | 0.477504 | 0.570562 | 0.646475 | 0.505551 | 0.673758 |
| **91** | 2001 | 0.484382 | 0.561083 | 0.667414 | 0.502400 | 0.664474 |
| **92** | 2002 | 0.490194 | 0.546057 | 0.643013 | 0.508456 | 0.663484 |
| **93** | 2003 | 0.490297 | 0.538409 | 0.667090 | 0.502553 | 0.665587 |
| **94** | 2004 | 0.506145 | 0.529699 | 0.670242 | NaN | 0.673698 |
| **95** | 2005 | 0.522943 | 0.523728 | 0.657116 | 0.511891 | 0.674178 |
| **96** | 2006 | 0.539353 | 0.528147 | 0.638349 | 0.519773 | 0.679792 |
| **97** | 2007 | 0.558198 | 0.535888 | 0.638570 | NaN | 0.690305 |
| **98** | 2008 | 0.569170 | 0.532034 | 0.664426 | NaN | 0.719994 |
| **99** | 2009 | 0.582027 | 0.540526 | 0.628777 | 0.540135 | 0.727669 |
| **100** | 2010 | 0.627582 | 0.559136 | 0.659879 | NaN | 0.732543 |
| **101** | 2011 | 0.667127 | 0.550742 | 0.683105 | NaN | 0.732633 |
| **102** | 2012 | 0.665248 | 0.545121 | 0.679313 | 0.519160 | 0.737443 |
| **103** | 2013 | 0.665624 | 0.548516 | 0.678541 | NaN | 0.723082 |
| **104** | 2014 | 0.667396 | 0.552765 | 0.684878 | NaN | 0.721835 |
| **105** | 2015 | 0.674086 | NaN | 0.713222 | NaN | NaN |

106 rows × 6 columns

### *3.2.8. Appending and merging*

Pandas is very efficient at performing join operation, it is over an order of magnitude faster than R in some cases according to [the Pandas documentation (https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html#database-style-dataframe-or-named-series-joining-merging)](https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html#database-style-dataframe-or-named-series-joining-merging). This is both due to the clever design of pandas' merging algorithms, and inherently efficient structure of data frames.

Merging data frames or series by rows is done with Pandas' `merge` function. The two datasets we need to merge are called **left** and **right**, and a `merge` call creates either a left, right, inner or outer join:

- with a left join, only rows from the left dataset are kept
- right join = only rows from the right dataset
- inner join = only rows from both datasets (default option)
- outer join = all rows, with non-existing values filled with `NaN`

The `merge` function takes the following important arguments:

- a 'left' data frame
- a 'right' data frame
- a type of join, entered as an argument for after `how=`
- keys for the join. If the columns or indices used for the merge are similarly named in both datasets then a single key name can be passed to the `merge` function as `on=` . If they differ in both datasets, then we use `left_on=` and `right_on` .
- `sort = True` if we want the merged data frame to be sorted. Set to `False` for improved performance
- `indicator = True` to create a new column to the merged data frame called `_merge` . The column takes the values `left_only` , `right_only` or `both` depending on the availability of keys in the data frames

In what follows, we set the years as indices for the observations in our data frame, we create a multi-index for the column and merge the two data frames. The multi-indexing of the columns allows us to then merge the two data frames without renaming columns.

In [39]:

```
# Re-indexing Top 1% data set
top1Percent_reIndex = top1Percent.set_index(['Year'])
top1Percent_reIndex
```

Out[39]:

| | China | France | Russia | UK | US |
|---|---|---|---|---|---|
| **Year** | | | | | |
| **1910** | NaN | NaN | NaN | NaN | NaN |
| **1911** | NaN | NaN | NaN | NaN | NaN |
| **1912** | NaN | NaN | NaN | NaN | NaN |
| **1913** | NaN | 0.545610 | NaN | 0.665846 | NaN |
| **1914** | NaN | 0.545639 | NaN | 0.672140 | NaN |
| **1915** | NaN | 0.540021 | NaN | NaN | NaN |
| **1916** | NaN | 0.537610 | NaN | NaN | NaN |
| **1917** | NaN | 0.534866 | NaN | NaN | 0.405009 |
| **1918** | NaN | 0.528085 | NaN | NaN | 0.370107 |
| **1919** | NaN | 0.520013 | NaN | 0.625506 | 0.399661 |
| **1920** | NaN | 0.504585 | NaN | 0.573150 | 0.356388 |
| **1921** | NaN | 0.493960 | NaN | 0.605379 | 0.367703 |
| **1922** | NaN | 0.484599 | NaN | 0.617355 | 0.399458 |
| **1923** | NaN | 0.477312 | NaN | 0.602446 | 0.353625 |
| **1924** | NaN | 0.474269 | NaN | 0.594641 | 0.374402 |
| **1925** | NaN | 0.446987 | NaN | 0.602700 | 0.408918 |
| **1926** | NaN | 0.453574 | NaN | 0.568876 | 0.425562 |
| **1927** | NaN | 0.477408 | NaN | 0.591104 | 0.448943 |
| **1928** | NaN | NaN | NaN | 0.564596 | 0.477967 |
| **1929** | NaN | 0.490732 | NaN | 0.563224 | 0.479612 |
| **1930** | NaN | 0.496065 | NaN | 0.569378 | 0.433664 |
| **1931** | NaN | 0.463320 | NaN | 0.531109 | 0.386037 |
| **1932** | NaN | 0.447956 | NaN | 0.543186 | 0.380773 |
| **1933** | NaN | 0.445935 | NaN | 0.559489 | 0.403569 |
| **1934** | NaN | NaN | NaN | 0.537953 | 0.409703 |
| **1935** | NaN | 0.437453 | NaN | 0.539764 | 0.404630 |
| **1936** | NaN | 0.432667 | NaN | 0.534268 | 0.429832 |
| **1937** | NaN | 0.426368 | NaN | 0.531311 | 0.436398 |
| **1938** | NaN | 0.396942 | NaN | 0.540719 | 0.397460 |
| **1939** | NaN | 0.399935 | NaN | 0.511888 | 0.407887 |
| **...** | ... | ... | ... | ... | ... |
| **1991** | NaN | 0.180916 | NaN | 0.155803 | 0.259942 |
| **1992** | NaN | 0.174981 | NaN | 0.169917 | 0.275662 |
| **1993** | NaN | 0.187896 | NaN | 0.182895 | 0.276869 |
| **1994** | NaN | 0.193238 | NaN | 0.176451 | 0.276058 |
| **1995** | 0.157972 | 0.196422 | 0.215031 | 0.162256 | 0.279207 |

| Year | China | France | Russia | UK | US |
|---|---|---|---|---|---|
| **1996** | 0.170144 | 0.233209 | 0.234242 | 0.165481 | 0.285775 |
| **1997** | 0.179232 | 0.253082 | 0.315070 | 0.192691 | 0.294624 |
| **1998** | 0.186277 | 0.266986 | 0.357450 | 0.199612 | 0.306972 |
| **1999** | 0.191897 | 0.278355 | 0.412466 | 0.193029 | 0.314705 |
| **2000** | 0.196486 | 0.281123 | 0.391769 | 0.184968 | 0.322992 |
| **2001** | 0.200303 | 0.270501 | 0.428869 | 0.188568 | 0.313342 |
| **2002** | 0.203528 | 0.254023 | 0.384765 | 0.180453 | 0.301582 |
| **2003** | 0.205002 | 0.246183 | 0.427292 | 0.167896 | 0.303230 |
| **2004** | 0.224526 | 0.237642 | 0.430843 | NaN | 0.314760 |
| **2005** | 0.237035 | 0.225111 | 0.404504 | 0.187657 | 0.320966 |
| **2006** | 0.262048 | 0.221321 | 0.367203 | 0.198744 | 0.328335 |
| **2007** | 0.284824 | 0.223749 | 0.359593 | NaN | 0.339606 |
| **2008** | 0.292496 | 0.215929 | 0.393182 | NaN | 0.360910 |
| **2009** | 0.311558 | 0.217011 | 0.317463 | 0.205814 | 0.361491 |
| **2010** | 0.304504 | 0.235066 | 0.342774 | NaN | 0.375695 |
| **2011** | 0.279195 | 0.229755 | 0.359799 | NaN | 0.374284 |
| **2012** | 0.272453 | 0.223578 | 0.354666 | 0.198812 | 0.388486 |
| **2013** | 0.272461 | 0.229046 | 0.354627 | NaN | 0.370319 |
| **2014** | 0.278310 | 0.233789 | 0.369069 | NaN | 0.372446 |
| **2015** | 0.296290 | NaN | 0.425818 | NaN | 0.372094 |
| **2016** | NaN | NaN | NaN | NaN | 0.369031 |
| **2017** | NaN | NaN | NaN | NaN | NaN |
| **2018** | NaN | NaN | NaN | NaN | NaN |
| **2019** | NaN | NaN | NaN | NaN | NaN |
| **2020** | NaN | NaN | NaN | NaN | NaN |

111 rows × 5 columns

In [40]:

```
# Re-indexing Top 10% data set
top10Percent_reIndex = top10Percent.set_index(['Year'])
top10Percent_reIndex
```

Out[40]:

| | China | France | Russian Federation | United Kingdom | USA |
|---|---|---|---|---|---|
| **Year** | | | | | |
| **1910** | NaN | NaN | NaN | NaN | NaN |
| **1911** | NaN | NaN | NaN | NaN | NaN |
| **1912** | NaN | NaN | NaN | NaN | NaN |
| **1913** | NaN | 0.849030 | NaN | 0.925733 | NaN |
| **1914** | NaN | 0.849074 | NaN | 0.929655 | NaN |
| **1915** | NaN | 0.843429 | NaN | NaN | NaN |
| **1916** | NaN | 0.843037 | NaN | NaN | NaN |
| **1917** | NaN | 0.842252 | NaN | NaN | 0.782449 |
| **1918** | NaN | 0.838413 | NaN | NaN | 0.785049 |
| **1919** | NaN | 0.833341 | NaN | 0.885341 | 0.800170 |
| **1920** | NaN | 0.822932 | NaN | 0.879738 | 0.780308 |
| **1921** | NaN | 0.815696 | NaN | 0.881781 | 0.779401 |
| **1922** | NaN | 0.809572 | NaN | 0.888246 | 0.791782 |
| **1923** | NaN | 0.804844 | NaN | 0.883304 | 0.796282 |
| **1924** | NaN | 0.803360 | NaN | 0.879293 | 0.810724 |
| **1925** | NaN | 0.786832 | NaN | 0.881648 | 0.821536 |
| **1926** | NaN | 0.787089 | NaN | 0.872117 | 0.830990 |
| **1927** | NaN | 0.798049 | NaN | 0.879828 | 0.841124 |
| **1928** | NaN | NaN | NaN | 0.866827 | 0.843926 |
| **1929** | NaN | 0.802657 | NaN | 0.870702 | 0.843327 |
| **1930** | NaN | 0.802256 | NaN | 0.861311 | 0.845685 |
| **1931** | NaN | 0.787573 | NaN | 0.858074 | 0.843043 |
| **1932** | NaN | 0.779655 | NaN | 0.857418 | 0.847409 |
| **1933** | NaN | 0.781155 | NaN | 0.864071 | 0.845613 |
| **1934** | NaN | NaN | NaN | 0.861166 | 0.829752 |
| **1935** | NaN | 0.772239 | NaN | 0.858730 | 0.816211 |
| **1936** | NaN | 0.766867 | NaN | 0.851632 | 0.821281 |
| **1937** | NaN | 0.763813 | NaN | 0.854700 | 0.802343 |
| **1938** | NaN | 0.747334 | NaN | 0.850125 | 0.799010 |
| **1939** | NaN | 0.755728 | NaN | 0.842894 | 0.802181 |
| **...** | ... | ... | ... | ... | ... |
| **1986** | NaN | 0.505658 | NaN | 0.488240 | 0.606497 |
| **1987** | NaN | 0.504989 | NaN | 0.503588 | 0.615782 |
| **1988** | NaN | 0.504901 | NaN | 0.481854 | 0.627376 |
| **1989** | NaN | 0.507558 | NaN | 0.485264 | 0.627007 |
| **1990** | NaN | 0.502717 | NaN | 0.459857 | 0.628830 |

| Year | China | France | Russian Federation | United Kingdom | USA |
|---|---|---|---|---|---|
| **1991** | NaN | 0.506542 | NaN | 0.455891 | 0.627435 |
| **1992** | NaN | 0.510053 | NaN | 0.479958 | 0.642536 |
| **1993** | NaN | 0.512132 | NaN | 0.498296 | 0.645715 |
| **1994** | NaN | 0.511994 | NaN | 0.495453 | 0.646331 |
| **1995** | 0.408106 | 0.511167 | 0.525537 | 0.469170 | 0.650016 |
| **1996** | 0.430038 | 0.540069 | 0.544091 | 0.483788 | 0.654424 |
| **1997** | 0.446414 | 0.552385 | 0.595655 | 0.515730 | 0.659853 |
| **1998** | 0.459108 | 0.563284 | 0.624042 | 0.518868 | 0.668028 |
| **1999** | 0.469236 | 0.568759 | 0.657416 | 0.500720 | 0.670313 |
| **2000** | 0.477504 | 0.570562 | 0.646475 | 0.505551 | 0.673758 |
| **2001** | 0.484382 | 0.561083 | 0.667414 | 0.502400 | 0.664474 |
| **2002** | 0.490194 | 0.546057 | 0.643013 | 0.508456 | 0.663484 |
| **2003** | 0.490297 | 0.538409 | 0.667090 | 0.502553 | 0.665587 |
| **2004** | 0.506145 | 0.529699 | 0.670242 | NaN | 0.673698 |
| **2005** | 0.522943 | 0.523728 | 0.657116 | 0.511891 | 0.674178 |
| **2006** | 0.539353 | 0.528147 | 0.638349 | 0.519773 | 0.679792 |
| **2007** | 0.558198 | 0.535888 | 0.638570 | NaN | 0.690305 |
| **2008** | 0.569170 | 0.532034 | 0.664426 | NaN | 0.719994 |
| **2009** | 0.582027 | 0.540526 | 0.628777 | 0.540135 | 0.727669 |
| **2010** | 0.627582 | 0.559136 | 0.659879 | NaN | 0.732543 |
| **2011** | 0.667127 | 0.550742 | 0.683105 | NaN | 0.732633 |
| **2012** | 0.665248 | 0.545121 | 0.679313 | 0.519160 | 0.737443 |
| **2013** | 0.665624 | 0.548516 | 0.678541 | NaN | 0.723082 |
| **2014** | 0.667396 | 0.552765 | 0.684878 | NaN | 0.721835 |
| **2015** | 0.674086 | NaN | 0.713222 | NaN | NaN |

106 rows × 5 columns

In [41]:

```
# We need to do a bit of renaming to homogenise variable names across datasets, this pa
rt is not really important
top10Percent_reIndex = top10Percent_reIndex.rename(columns={"Russian Federation": "Russ
ia", "United Kingdom": "UK", "USA": "US"})

# Creating a MultiIndex for the top 10% dataset
top10Percent_reIndex.columns = pd.MultiIndex.from_tuples([('Top 10%', c) for c in top10
Percent_reIndex.columns])

# What the nested indices look like
top10Percent_reIndex.columns
```

Out[41]:

```
MultiIndex(levels=[['Top 10%'], ['China', 'France', 'Russia', 'UK', 'U
S']],
           codes=[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4]])
```

In [42]:

```
# What the new data frame looks like
top10Percent_reIndex
```

Out[42]:

| | Top 10% | | | | |
|---|---|---|---|---|---|
| | **China** | **France** | **Russia** | **UK** | **US** |
| **Year** | | | | | |
| **1910** | NaN | NaN | NaN | NaN | NaN |
| **1911** | NaN | NaN | NaN | NaN | NaN |
| **1912** | NaN | NaN | NaN | NaN | NaN |
| **1913** | NaN | 0.849030 | NaN | 0.925733 | NaN |
| **1914** | NaN | 0.849074 | NaN | 0.929655 | NaN |
| **1915** | NaN | 0.843429 | NaN | NaN | NaN |
| **1916** | NaN | 0.843037 | NaN | NaN | NaN |
| **1917** | NaN | 0.842252 | NaN | NaN | 0.782449 |
| **1918** | NaN | 0.838413 | NaN | NaN | 0.785049 |
| **1919** | NaN | 0.833341 | NaN | 0.885341 | 0.800170 |
| **1920** | NaN | 0.822932 | NaN | 0.879738 | 0.780308 |
| **1921** | NaN | 0.815696 | NaN | 0.881781 | 0.779401 |
| **1922** | NaN | 0.809572 | NaN | 0.888246 | 0.791782 |
| **1923** | NaN | 0.804844 | NaN | 0.883304 | 0.796282 |
| **1924** | NaN | 0.803360 | NaN | 0.879293 | 0.810724 |
| **1925** | NaN | 0.786832 | NaN | 0.881648 | 0.821536 |
| **1926** | NaN | 0.787089 | NaN | 0.872117 | 0.830990 |
| **1927** | NaN | 0.798049 | NaN | 0.879828 | 0.841124 |
| **1928** | NaN | NaN | NaN | 0.866827 | 0.843926 |
| **1929** | NaN | 0.802657 | NaN | 0.870702 | 0.843327 |
| **1930** | NaN | 0.802256 | NaN | 0.861311 | 0.845685 |
| **1931** | NaN | 0.787573 | NaN | 0.858074 | 0.843043 |
| **1932** | NaN | 0.779655 | NaN | 0.857418 | 0.847409 |
| **1933** | NaN | 0.781155 | NaN | 0.864071 | 0.845613 |
| **1934** | NaN | NaN | NaN | 0.861166 | 0.829752 |
| **1935** | NaN | 0.772239 | NaN | 0.858730 | 0.816211 |
| **1936** | NaN | 0.766867 | NaN | 0.851632 | 0.821281 |
| **1937** | NaN | 0.763813 | NaN | 0.854700 | 0.802343 |
| **1938** | NaN | 0.747334 | NaN | 0.850125 | 0.799010 |
| **1939** | NaN | 0.755728 | NaN | 0.842894 | 0.802181 |
| **...** | ... | ... | ... | ... | ... |
| **1986** | NaN | 0.505658 | NaN | 0.488240 | 0.606497 |
| **1987** | NaN | 0.504989 | NaN | 0.503588 | 0.615782 |
| **1988** | NaN | 0.504901 | NaN | 0.481854 | 0.627376 |
| **1989** | NaN | 0.507558 | NaN | 0.485264 | 0.627007 |

| | Top 10% | | | | |
|---|---|---|---|---|---|
| | China | France | Russia | UK | US |
| Year | | | | | |
| 1990 | NaN | 0.502717 | NaN | 0.459857 | 0.628830 |
| 1991 | NaN | 0.506542 | NaN | 0.455891 | 0.627435 |
| 1992 | NaN | 0.510053 | NaN | 0.479958 | 0.642536 |
| 1993 | NaN | 0.512132 | NaN | 0.498296 | 0.645715 |
| 1994 | NaN | 0.511994 | NaN | 0.495453 | 0.646331 |
| 1995 | 0.408106 | 0.511167 | 0.525537 | 0.469170 | 0.650016 |
| 1996 | 0.430038 | 0.540069 | 0.544091 | 0.483788 | 0.654424 |
| 1997 | 0.446414 | 0.552385 | 0.595655 | 0.515730 | 0.659853 |
| 1998 | 0.459108 | 0.563284 | 0.624042 | 0.518868 | 0.668028 |
| 1999 | 0.469236 | 0.568759 | 0.657416 | 0.500720 | 0.670313 |
| 2000 | 0.477504 | 0.570562 | 0.646475 | 0.505551 | 0.673758 |
| 2001 | 0.484382 | 0.561083 | 0.667414 | 0.502400 | 0.664474 |
| 2002 | 0.490194 | 0.546057 | 0.643013 | 0.508456 | 0.663484 |
| 2003 | 0.490297 | 0.538409 | 0.667090 | 0.502553 | 0.665587 |
| 2004 | 0.506145 | 0.529699 | 0.670242 | NaN | 0.673698 |
| 2005 | 0.522943 | 0.523728 | 0.657116 | 0.511891 | 0.674178 |
| 2006 | 0.539353 | 0.528147 | 0.638349 | 0.519773 | 0.679792 |
| 2007 | 0.558198 | 0.535888 | 0.638570 | NaN | 0.690305 |
| 2008 | 0.569170 | 0.532034 | 0.664426 | NaN | 0.719994 |
| 2009 | 0.582027 | 0.540526 | 0.628777 | 0.540135 | 0.727669 |
| 2010 | 0.627582 | 0.559136 | 0.659879 | NaN | 0.732543 |
| 2011 | 0.667127 | 0.550742 | 0.683105 | NaN | 0.732633 |
| 2012 | 0.665248 | 0.545121 | 0.679313 | 0.519160 | 0.737443 |
| 2013 | 0.665624 | 0.548516 | 0.678541 | NaN | 0.723082 |
| 2014 | 0.667396 | 0.552765 | 0.684878 | NaN | 0.721835 |
| 2015 | 0.674086 | NaN | 0.713222 | NaN | NaN |

106 rows × 5 columns

In [43]:

```
# We define the multiIndex for the top 1% in the same way
top1Percent_reIndex.columns = pd.MultiIndex.from_tuples([('Top 1%', c) for c in top1Per
cent_reIndex.columns])
```

In [44]:

```
# We can now merge the two datasets
topShares = pd.merge(top10Percent_reIndex, top1Percent_reIndex,
                     how='outer', on='Year',
                     sort=True,
                     copy=True, indicator=False)
```

In [45]:

```
# What the merged dataset looks like
topShares
```

Out[45]:

| | Top 10% | | | | | Top 1% | | | |
|---|---|---|---|---|---|---|---|---|---|
| | China | France | Russia | UK | US | China | France | Russia | UK |
| Year | | | | | | | | | |
| 1910 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | N |
| 1911 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | N |
| 1912 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | N |
| 1913 | NaN | 0.849030 | NaN | 0.925733 | NaN | NaN | 0.545610 | NaN | 0.665 |
| 1914 | NaN | 0.849074 | NaN | 0.929655 | NaN | NaN | 0.545639 | NaN | 0.672 |
| 1915 | NaN | 0.843429 | NaN | NaN | NaN | NaN | 0.540021 | NaN | N |
| 1916 | NaN | 0.843037 | NaN | NaN | NaN | NaN | 0.537610 | NaN | N |
| 1917 | NaN | 0.842252 | NaN | NaN | 0.782449 | NaN | 0.534866 | NaN | N |
| 1918 | NaN | 0.838413 | NaN | NaN | 0.785049 | NaN | 0.528085 | NaN | N |
| 1919 | NaN | 0.833341 | NaN | 0.885341 | 0.800170 | NaN | 0.520013 | NaN | 0.625 |
| 1920 | NaN | 0.822932 | NaN | 0.879738 | 0.780308 | NaN | 0.504585 | NaN | 0.573 |
| 1921 | NaN | 0.815696 | NaN | 0.881781 | 0.779401 | NaN | 0.493960 | NaN | 0.605 |
| 1922 | NaN | 0.809572 | NaN | 0.888246 | 0.791782 | NaN | 0.484599 | NaN | 0.617 |
| 1923 | NaN | 0.804844 | NaN | 0.883304 | 0.796282 | NaN | 0.477312 | NaN | 0.602 |
| 1924 | NaN | 0.803360 | NaN | 0.879293 | 0.810724 | NaN | 0.474269 | NaN | 0.594 |
| 1925 | NaN | 0.786832 | NaN | 0.881648 | 0.821536 | NaN | 0.446987 | NaN | 0.602 |
| 1926 | NaN | 0.787089 | NaN | 0.872117 | 0.830990 | NaN | 0.453574 | NaN | 0.568 |
| 1927 | NaN | 0.798049 | NaN | 0.879828 | 0.841124 | NaN | 0.477408 | NaN | 0.591 |
| 1928 | NaN | NaN | NaN | 0.866827 | 0.843926 | NaN | NaN | NaN | 0.564 |
| 1929 | NaN | 0.802657 | NaN | 0.870702 | 0.843327 | NaN | 0.490732 | NaN | 0.563 |
| 1930 | NaN | 0.802256 | NaN | 0.861311 | 0.845685 | NaN | 0.496065 | NaN | 0.569 |
| 1931 | NaN | 0.787573 | NaN | 0.858074 | 0.843043 | NaN | 0.463320 | NaN | 0.531 |
| 1932 | NaN | 0.779655 | NaN | 0.857418 | 0.847409 | NaN | 0.447956 | NaN | 0.543 |
| 1933 | NaN | 0.781155 | NaN | 0.864071 | 0.845613 | NaN | 0.445935 | NaN | 0.559 |
| 1934 | NaN | NaN | NaN | 0.861166 | 0.829752 | NaN | NaN | NaN | 0.537 |
| 1935 | NaN | 0.772239 | NaN | 0.858730 | 0.816211 | NaN | 0.437453 | NaN | 0.539 |
| 1936 | NaN | 0.766867 | NaN | 0.851632 | 0.821281 | NaN | 0.432667 | NaN | 0.534 |
| 1937 | NaN | 0.763813 | NaN | 0.854700 | 0.802343 | NaN | 0.426368 | NaN | 0.531 |
| 1938 | NaN | 0.747334 | NaN | 0.850125 | 0.799010 | NaN | 0.396942 | NaN | 0.540 |
| 1939 | NaN | 0.755728 | NaN | 0.842894 | 0.802181 | NaN | 0.399935 | NaN | 0.511 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| 1991 | NaN | 0.506542 | NaN | 0.455891 | 0.627435 | NaN | 0.180916 | NaN | 0.155 |
| 1992 | NaN | 0.510053 | NaN | 0.479958 | 0.642536 | NaN | 0.174981 | NaN | 0.169 |
| 1993 | NaN | 0.512132 | NaN | 0.498296 | 0.645715 | NaN | 0.187896 | NaN | 0.182 |
| 1994 | NaN | 0.511994 | NaN | 0.495453 | 0.646331 | NaN | 0.193238 | NaN | 0.176 |

| | Top 10% | | | | | Top 1% | | | |
|---|---|---|---|---|---|---|---|---|---|
| | China | France | Russia | UK | US | China | France | Russia | UK |
| Year | | | | | | | | | |
| 1995 | 0.408106 | 0.511167 | 0.525537 | 0.469170 | 0.650016 | 0.157972 | 0.196422 | 0.215031 | 0.162 |
| 1996 | 0.430038 | 0.540069 | 0.544091 | 0.483788 | 0.654424 | 0.170144 | 0.233209 | 0.234242 | 0.165 |
| 1997 | 0.446414 | 0.552385 | 0.595655 | 0.515730 | 0.659853 | 0.179232 | 0.253082 | 0.315070 | 0.192 |
| 1998 | 0.459108 | 0.563284 | 0.624042 | 0.518868 | 0.668028 | 0.186277 | 0.266986 | 0.357450 | 0.199 |
| 1999 | 0.469236 | 0.568759 | 0.657416 | 0.500720 | 0.670313 | 0.191897 | 0.278355 | 0.412466 | 0.193 |
| 2000 | 0.477504 | 0.570562 | 0.646475 | 0.505551 | 0.673758 | 0.196486 | 0.281123 | 0.391769 | 0.184 |
| 2001 | 0.484382 | 0.561083 | 0.667414 | 0.502400 | 0.664474 | 0.200303 | 0.270501 | 0.428869 | 0.188 |
| 2002 | 0.490194 | 0.546057 | 0.643013 | 0.508456 | 0.663484 | 0.203528 | 0.254023 | 0.384765 | 0.180 |
| 2003 | 0.490297 | 0.538409 | 0.667090 | 0.502553 | 0.665587 | 0.205002 | 0.246183 | 0.427292 | 0.167 |
| 2004 | 0.506145 | 0.529699 | 0.670242 | NaN | 0.673698 | 0.224526 | 0.237642 | 0.430843 | N |
| 2005 | 0.522943 | 0.523728 | 0.657116 | 0.511891 | 0.674178 | 0.237035 | 0.225111 | 0.404504 | 0.187 |
| 2006 | 0.539353 | 0.528147 | 0.638349 | 0.519773 | 0.679792 | 0.262048 | 0.221321 | 0.367203 | 0.198 |
| 2007 | 0.558198 | 0.535888 | 0.638570 | NaN | 0.690305 | 0.284824 | 0.223749 | 0.359593 | N |
| 2008 | 0.569170 | 0.532034 | 0.664426 | NaN | 0.719994 | 0.292496 | 0.215929 | 0.393182 | N |
| 2009 | 0.582027 | 0.540526 | 0.628777 | 0.540135 | 0.727669 | 0.311558 | 0.217011 | 0.317463 | 0.205 |
| 2010 | 0.627582 | 0.559136 | 0.659879 | NaN | 0.732543 | 0.304504 | 0.235066 | 0.342774 | N |
| 2011 | 0.667127 | 0.550742 | 0.683105 | NaN | 0.732633 | 0.279195 | 0.229755 | 0.359799 | N |
| 2012 | 0.665248 | 0.545121 | 0.679313 | 0.519160 | 0.737443 | 0.272453 | 0.223578 | 0.354666 | 0.198 |
| 2013 | 0.665624 | 0.548516 | 0.678541 | NaN | 0.723082 | 0.272461 | 0.229046 | 0.354627 | N |
| 2014 | 0.667396 | 0.552765 | 0.684878 | NaN | 0.721835 | 0.278310 | 0.233789 | 0.369069 | N |
| 2015 | 0.674086 | NaN | 0.713222 | NaN | NaN | 0.296290 | NaN | 0.425818 | N |
| 2016 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | N |
| 2017 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | N |
| 2018 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | N |
| 2019 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | N |
| 2020 | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | N |

111 rows × 10 columns

Appending data is relatively sraightforward if the two datasets do not have meaningful row indices. Below, we append the dataset of top 1% shares to that of top 10% shares, with years entered as columns. Note how variables with different column names across datasets are treated.

In [47]:

```
appendedData = top10Percent.append([top1Percent], sort=True)
```

In [48]:

```
appendedData
```

Out[48]:

| | China | France | Russia | Russian Federation | UK | US | USA | United Kingdom | Year |
|---|---|---|---|---|---|---|---|---|---|
| **0** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 1910 |
| **1** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 1911 |
| **2** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 1912 |
| **3** | NaN | 0.849030 | NaN | NaN | NaN | NaN | NaN | 0.925733 | 1913 |
| **4** | NaN | 0.849074 | NaN | NaN | NaN | NaN | NaN | 0.929655 | 1914 |
| **5** | NaN | 0.843429 | NaN | NaN | NaN | NaN | NaN | NaN | 1915 |
| **6** | NaN | 0.843037 | NaN | NaN | NaN | NaN | NaN | NaN | 1916 |
| **7** | NaN | 0.842252 | NaN | NaN | NaN | NaN | 0.782449 | NaN | 1917 |
| **8** | NaN | 0.838413 | NaN | NaN | NaN | NaN | 0.785049 | NaN | 1918 |
| **9** | NaN | 0.833341 | NaN | NaN | NaN | NaN | 0.800170 | 0.885341 | 1919 |
| **10** | NaN | 0.822932 | NaN | NaN | NaN | NaN | 0.780308 | 0.879738 | 1920 |
| **11** | NaN | 0.815696 | NaN | NaN | NaN | NaN | 0.779401 | 0.881781 | 1921 |
| **12** | NaN | 0.809572 | NaN | NaN | NaN | NaN | 0.791782 | 0.888246 | 1922 |
| **13** | NaN | 0.804844 | NaN | NaN | NaN | NaN | 0.796282 | 0.883304 | 1923 |
| **14** | NaN | 0.803360 | NaN | NaN | NaN | NaN | 0.810724 | 0.879293 | 1924 |
| **15** | NaN | 0.786832 | NaN | NaN | NaN | NaN | 0.821536 | 0.881648 | 1925 |
| **16** | NaN | 0.787089 | NaN | NaN | NaN | NaN | 0.830990 | 0.872117 | 1926 |
| **17** | NaN | 0.798049 | NaN | NaN | NaN | NaN | 0.841124 | 0.879828 | 1927 |
| **18** | NaN | NaN | NaN | NaN | NaN | NaN | 0.843926 | 0.866827 | 1928 |
| **19** | NaN | 0.802657 | NaN | NaN | NaN | NaN | 0.843327 | 0.870702 | 1929 |
| **20** | NaN | 0.802256 | NaN | NaN | NaN | NaN | 0.845685 | 0.861311 | 1930 |
| **21** | NaN | 0.787573 | NaN | NaN | NaN | NaN | 0.843043 | 0.858074 | 1931 |
| **22** | NaN | 0.779655 | NaN | NaN | NaN | NaN | 0.847409 | 0.857418 | 1932 |
| **23** | NaN | 0.781155 | NaN | NaN | NaN | NaN | 0.845613 | 0.864071 | 1933 |
| **24** | NaN | NaN | NaN | NaN | NaN | NaN | 0.829752 | 0.861166 | 1934 |
| **25** | NaN | 0.772239 | NaN | NaN | NaN | NaN | 0.816211 | 0.858730 | 1935 |
| **26** | NaN | 0.766867 | NaN | NaN | NaN | NaN | 0.821281 | 0.851632 | 1936 |
| **27** | NaN | 0.763813 | NaN | NaN | NaN | NaN | 0.802343 | 0.854700 | 1937 |
| **28** | NaN | 0.747334 | NaN | NaN | NaN | NaN | 0.799010 | 0.850125 | 1938 |
| **29** | NaN | 0.755728 | NaN | NaN | NaN | NaN | 0.802181 | 0.842894 | 1939 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **81** | NaN | 0.180916 | NaN | NaN | 0.155803 | 0.259942 | NaN | NaN | 1991 |
| **82** | NaN | 0.174981 | NaN | NaN | 0.169917 | 0.275662 | NaN | NaN | 1992 |
| **83** | NaN | 0.187896 | NaN | NaN | 0.182895 | 0.276869 | NaN | NaN | 1993 |
| **84** | NaN | 0.193238 | NaN | NaN | 0.176451 | 0.276058 | NaN | NaN | 1994 |
| **85** | 0.157972 | 0.196422 | 0.215031 | NaN | 0.162256 | 0.279207 | NaN | NaN | 1995 |

| | China | France | Russia | Russian Federation | UK | US | USA | United Kingdom | Year |
|---|---|---|---|---|---|---|---|---|---|
| **86** | 0.170144 | 0.233209 | 0.234242 | NaN | 0.165481 | 0.285775 | NaN | NaN | 1996 |
| **87** | 0.179232 | 0.253082 | 0.315070 | NaN | 0.192691 | 0.294624 | NaN | NaN | 1997 |
| **88** | 0.186277 | 0.266986 | 0.357450 | NaN | 0.199612 | 0.306972 | NaN | NaN | 1998 |
| **89** | 0.191897 | 0.278355 | 0.412466 | NaN | 0.193029 | 0.314705 | NaN | NaN | 1999 |
| **90** | 0.196486 | 0.281123 | 0.391769 | NaN | 0.184968 | 0.322992 | NaN | NaN | 2000 |
| **91** | 0.200303 | 0.270501 | 0.428869 | NaN | 0.188568 | 0.313342 | NaN | NaN | 2001 |
| **92** | 0.203528 | 0.254023 | 0.384765 | NaN | 0.180453 | 0.301582 | NaN | NaN | 2002 |
| **93** | 0.205002 | 0.246183 | 0.427292 | NaN | 0.167896 | 0.303230 | NaN | NaN | 2003 |
| **94** | 0.224526 | 0.237642 | 0.430843 | NaN | NaN | 0.314760 | NaN | NaN | 2004 |
| **95** | 0.237035 | 0.225111 | 0.404504 | NaN | 0.187657 | 0.320966 | NaN | NaN | 2005 |
| **96** | 0.262048 | 0.221321 | 0.367203 | NaN | 0.198744 | 0.328335 | NaN | NaN | 2006 |
| **97** | 0.284824 | 0.223749 | 0.359593 | NaN | NaN | 0.339606 | NaN | NaN | 2007 |
| **98** | 0.292496 | 0.215929 | 0.393182 | NaN | NaN | 0.360910 | NaN | NaN | 2008 |
| **99** | 0.311558 | 0.217011 | 0.317463 | NaN | 0.205814 | 0.361491 | NaN | NaN | 2009 |
| **100** | 0.304504 | 0.235066 | 0.342774 | NaN | NaN | 0.375695 | NaN | NaN | 2010 |
| **101** | 0.279195 | 0.229755 | 0.359799 | NaN | NaN | 0.374284 | NaN | NaN | 2011 |
| **102** | 0.272453 | 0.223578 | 0.354666 | NaN | 0.198812 | 0.388486 | NaN | NaN | 2012 |
| **103** | 0.272461 | 0.229046 | 0.354627 | NaN | NaN | 0.370319 | NaN | NaN | 2013 |
| **104** | 0.278310 | 0.233789 | 0.369069 | NaN | NaN | 0.372446 | NaN | NaN | 2014 |
| **105** | 0.296290 | NaN | 0.425818 | NaN | NaN | 0.372094 | NaN | NaN | 2015 |
| **106** | NaN | NaN | NaN | NaN | NaN | 0.369031 | NaN | NaN | 2016 |
| **107** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 2017 |
| **108** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 2018 |
| **109** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 2019 |
| **110** | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | 2020 |

217 rows × 9 columns

### *3.2.9. Reshaping data*

Our merged series of top income shares are in long format. We can easily reshape them in wide format, just like if we were using the `reshape` command in STATA. We will again make use of Pandas' `MultiIndex` functionality. These hierarchical indices are very useful to move seemlessly between long and wide formats when the data is indexed at many levels like in our merged dataset where columns are organised by shares of income first, and by country in a second time.

We use `pivot_table` to create a long panel, which will be more useful for panel data analysis. `pivot_table` needs to be fed:

- a data frame
- the list of variables used as values
- the list of variables kept as variables
- the list of variables used as indices
- [optional] a function processing the data at the level specified by the other option. The default is *numpy.mean*

In [53]:

```
# [To be changed]

#reshapedData = topShares.pivot_table(values = ['China', 'France', 'Russia', 'UK', 'U
S'],
#                             index = ['Year', 'Top 1%', 'Top 10%'],
#                            columns = ['China', 'France', 'Russia', 'UK', 'US'])
#topShares.columns
```

The `stack` and `unstack` commands can also be used to rotate the lowest and highest levels of the column to the row index, respectively.

In [54]:

```
reshapedData_stacked = topShares.stack()
reshapedData_stacked
```

Out[54]:

| | | Top 1% | Top 10% |
|---|---|---|---|
| **Year** | | | |
| **1913** | **France** | 0.545610 | 0.849030 |
| | **UK** | 0.665846 | 0.925733 |
| **1914** | **France** | 0.545639 | 0.849074 |
| | **UK** | 0.672140 | 0.929655 |
| **1915** | **France** | 0.540021 | 0.843429 |
| **1916** | **France** | 0.537610 | 0.843037 |
| **1917** | **France** | 0.534866 | 0.842252 |
| | **US** | 0.405009 | 0.782449 |
| **1918** | **France** | 0.528085 | 0.838413 |
| | **US** | 0.370107 | 0.785049 |
| **1919** | **France** | 0.520013 | 0.833341 |
| | **UK** | 0.625506 | 0.885341 |
| | **US** | 0.399661 | 0.800170 |
| **1920** | **France** | 0.504585 | 0.822932 |
| | **UK** | 0.573150 | 0.879738 |
| | **US** | 0.356388 | 0.780308 |
| **1921** | **France** | 0.493960 | 0.815696 |
| | **UK** | 0.605379 | 0.881781 |
| | **US** | 0.367703 | 0.779401 |
| **1922** | **France** | 0.484599 | 0.809572 |
| | **UK** | 0.617355 | 0.888246 |
| | **US** | 0.399458 | 0.791782 |
| **1923** | **France** | 0.477312 | 0.804844 |
| | **UK** | 0.602446 | 0.883304 |
| | **US** | 0.353625 | 0.796282 |
| **1924** | **France** | 0.474269 | 0.803360 |
| | **UK** | 0.594641 | 0.879293 |
| | **US** | 0.374402 | 0.810724 |
| **1925** | **France** | 0.446987 | 0.786832 |
| | **UK** | 0.602700 | 0.881648 |
| **...** | **...** | ... | ... |
| **2009** | **China** | 0.311558 | 0.582027 |
| | **France** | 0.217011 | 0.540526 |
| | **Russia** | 0.317463 | 0.628777 |
| | **UK** | 0.205814 | 0.540135 |
| | **US** | 0.361491 | 0.727669 |

| | | Top 1% | Top 10% |
|---|---|---|---|
| **Year** | | | |
| **2010** | **China** | 0.304504 | 0.627582 |
| | **France** | 0.235066 | 0.559136 |
| | **Russia** | 0.342774 | 0.659879 |
| | **US** | 0.375695 | 0.732543 |
| **2011** | **China** | 0.279195 | 0.667127 |
| | **France** | 0.229755 | 0.550742 |
| | **Russia** | 0.359799 | 0.683105 |
| | **US** | 0.374284 | 0.732633 |
| **2012** | **China** | 0.272453 | 0.665248 |
| | **France** | 0.223578 | 0.545121 |
| | **Russia** | 0.354666 | 0.679313 |
| | **UK** | 0.198812 | 0.519160 |
| | **US** | 0.388486 | 0.737443 |
| **2013** | **China** | 0.272461 | 0.665624 |
| | **France** | 0.229046 | 0.548516 |
| | **Russia** | 0.354627 | 0.678541 |
| | **US** | 0.370319 | 0.723082 |
| **2014** | **China** | 0.278310 | 0.667396 |
| | **France** | 0.233789 | 0.552765 |
| | **Russia** | 0.369069 | 0.684878 |
| | **US** | 0.372446 | 0.721835 |
| **2015** | **China** | 0.296290 | 0.674086 |
| | **Russia** | 0.425818 | 0.713222 |
| | **US** | 0.372094 | NaN |
| **2016** | **US** | 0.369031 | NaN |

327 rows × 2 columns

## 4. Matplotlib

Matplotlib is a powerful graphing module for Python. Getting it to generate the graph you want might be a bit less intuitive than with STATA's graphs functions, but matplotlib can do many things STATA cannot:

- 3D plots
- Varying opacities
- sophisticated LaTeX integration
- Animation of graphs

When using matplotlib in a Jupyter Notebook, we will use the `%matplotlib nbagg` option to display the graphs within the Notebook. If we do not use this option, the graphs will appear in a separate window.

In [55]:

```
%matplotlib nbagg

import matplotlib.pyplot as plt
```

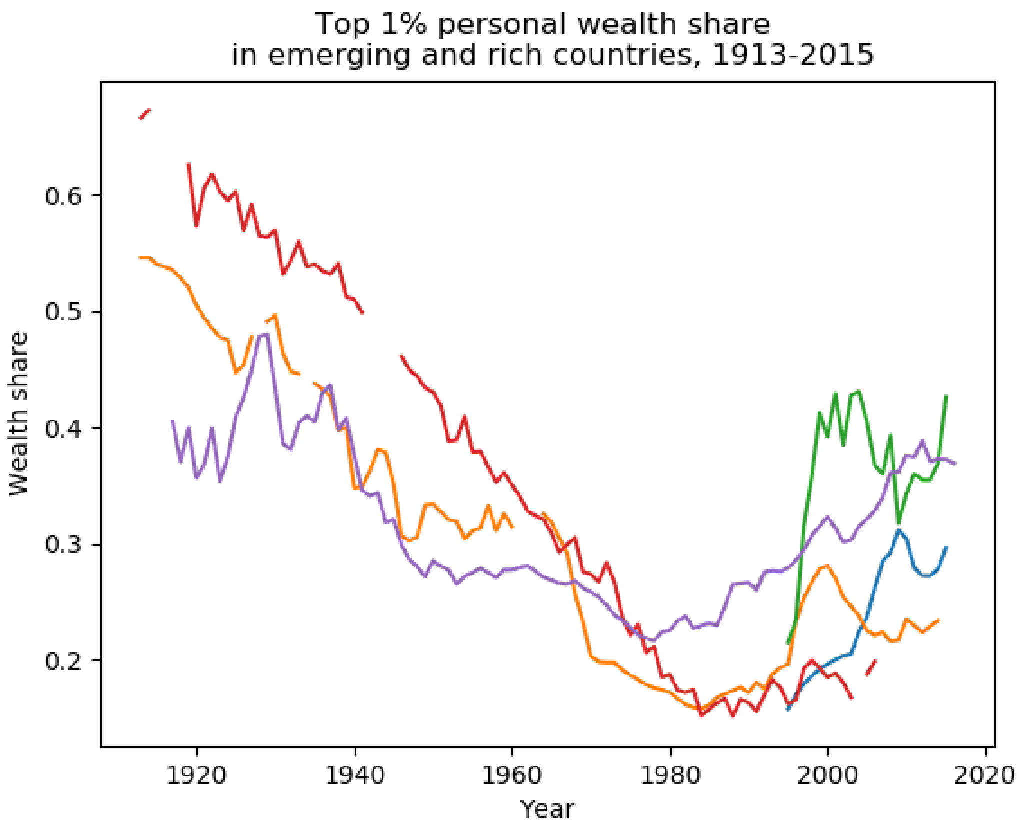
### 4.1. Creating a simple figure

To create a 2D graph, we need to:

- Define a new figure with `plt.figure(` *figure number* `)`
- Define the data by creating arrays or lists containing the coordinates of the data points
- Add options for the legend, title, axis labels, appearance, etc.
- Generate the plot with `plt.plot( x1, y1, x2, y2, ..., xn, yn)`

In [56]:

```
# Open new figure
plt.figure(1)

# Define data, from previously imported data frame
x = top1Percent['Year']
yCh = top1Percent['China']
yFr = top1Percent['France']
yRu = top1Percent['Russia']
yUK = top1Percent['UK']
yUS = top1Percent['US']

# Graph options
plt.title("Top 1% personal wealth share \n in emerging and rich countries, 1913-2015")
plt.ylabel("Wealth share")
plt.xlabel("Year")

# Generate graph
plt.plot(x, yCh, x, yFr, x, yRu, x, yUK, x, yUS)
```




Out[56]:

```
[<matplotlib.lines.Line2D at 0x1e0c716fdd8>,
 <matplotlib.lines.Line2D at 0x1e0c716ff28>,
 <matplotlib.lines.Line2D at 0x1e0c71800b8>,
 <matplotlib.lines.Line2D at 0x1e0c7180208>,
 <matplotlib.lines.Line2D at 0x1e0c7180358>]
```

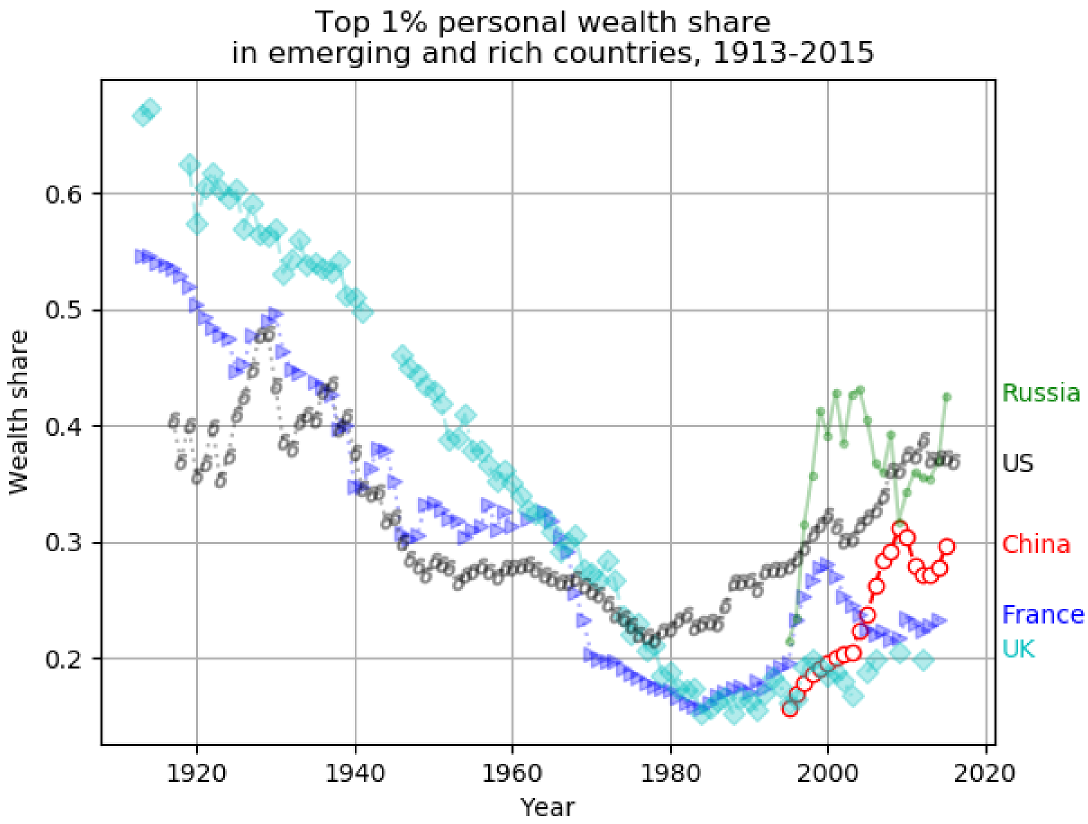
Note that if you use a string of characters containing apostrophes and accents, you will need to add `u` before the string to tell Python which encoding to use ( `u` is for Unicode).

We can change the appearance of the graph after having generated it. The command below shows you how to play with opacity ( `alpha` parameter), colours and line patterns ( `r--` for instance), marker shapes, and fillings.

To get more control on the appearance of the series, we will need to add them to the graph separately. Each series will be generated by its own `plt.plot(...)` command.

In [57]:

```
plt.figure(2)

# Graph options
plt.title("Top 1% personal wealth share \n in emerging and rich countries, 1913-2015")
plt.ylabel("Wealth share")
plt.xlabel("Year")
plt.grid()

# Generate graph
plt.plot(x, yCh, 'r--', marker='o', fillstyle='full', markerfacecolor='white')
plt.plot(x, yFr, 'b:', marker='>', alpha=0.3)
plt.plot(x, yRu, 'g-', marker='.', alpha=0.3)
plt.plot(x, yUK, 'c-.', marker='D', alpha=0.3)
plt.plot(x, yUS, 'k:', marker='$ \delta $', alpha=0.3)      # We use TeX to define the
 marker. This is a bit of a silly example here,
                                                            #   but we just want to sho
w how to use TeX in a figure

# Label curves
plt.text(2022, 0.29, 'China', {'color': 'r', 'fontsize': 10}, ha="left")
plt.text(2022, 0.23, 'France', {'color': 'b', 'fontsize': 10}, ha="left")
plt.text(2022, 0.42, 'Russia', {'color': 'g', 'fontsize': 10}, ha="left")
plt.text(2022, 0.2, 'UK', {'color': 'c', 'fontsize': 10}, ha="left")
plt.text(2022, 0.36, 'US', {'color': 'k', 'fontsize': 10}, ha="left")
```

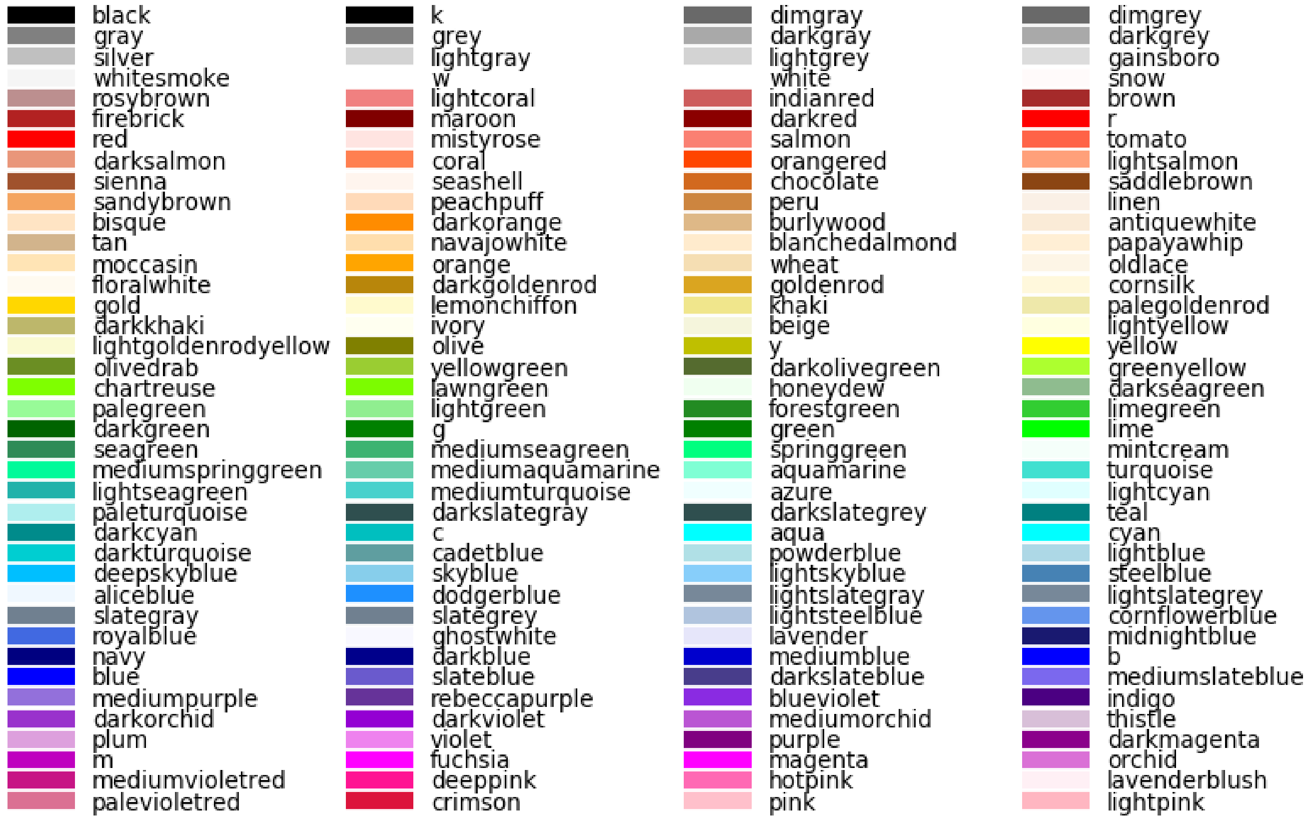



Out[57]:

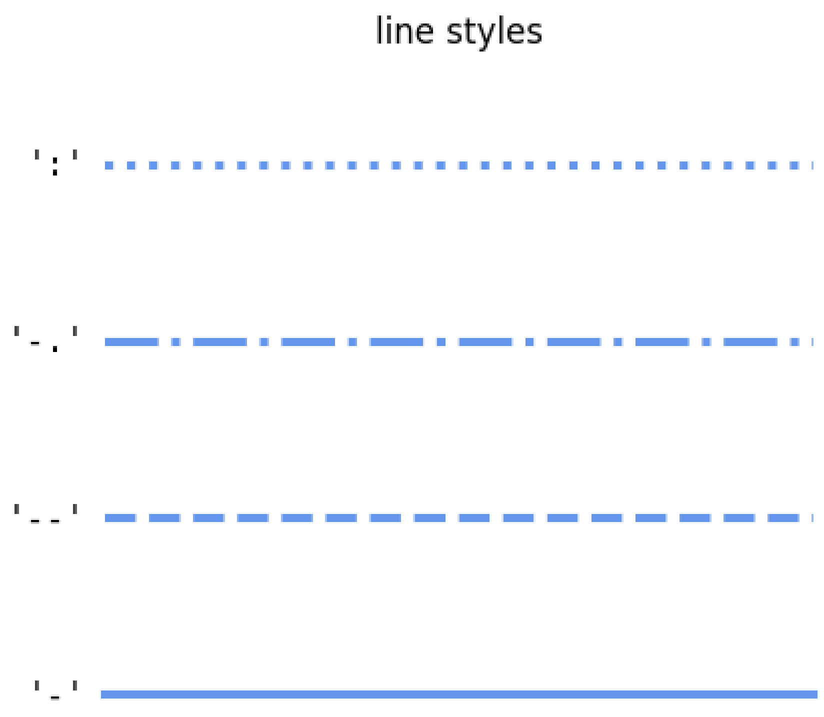
```
Text(2022, 0.36, 'US')
```

The colours, marker shapes and line styles provided by Matplotlib are easily accessible through the [matplotlib website (https://matplotlib.org/2.0.0/index.html)](https://matplotlib.org/2.0.0/index.html). We include them here for future reference.

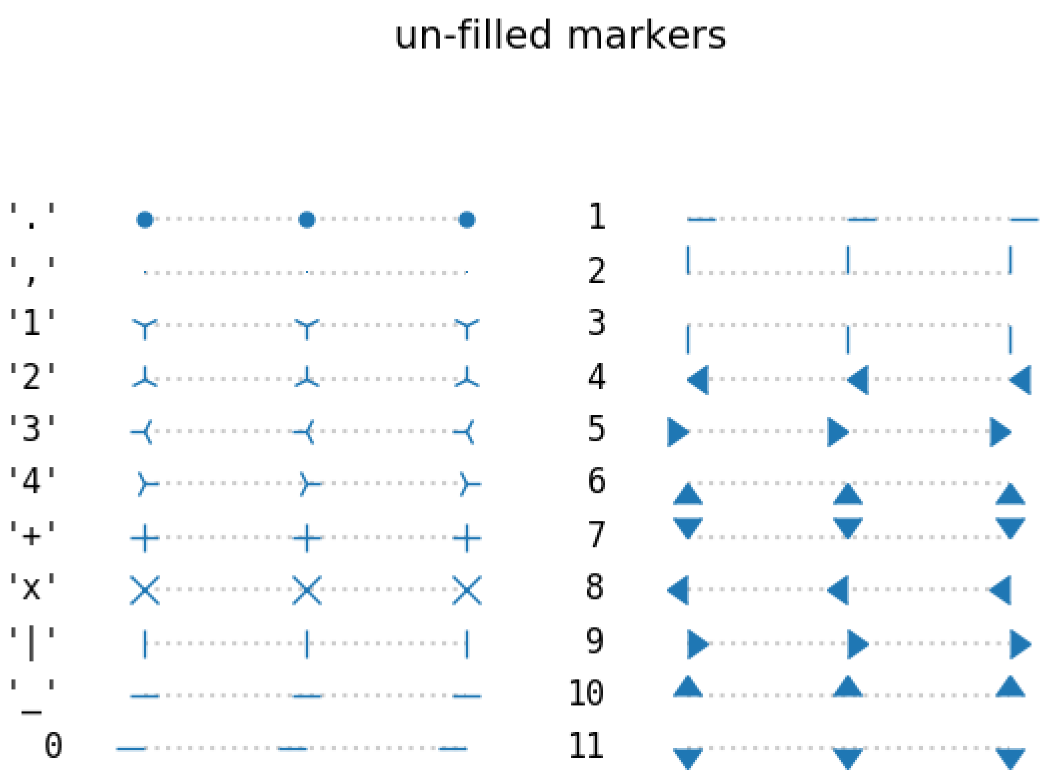
### Colours

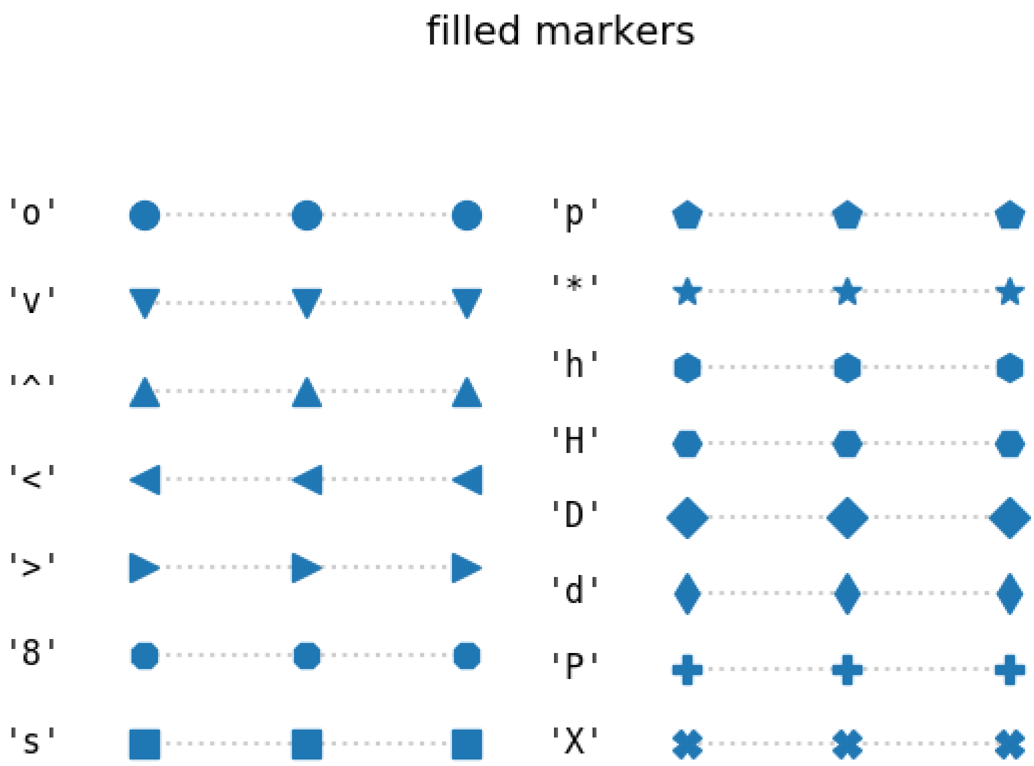


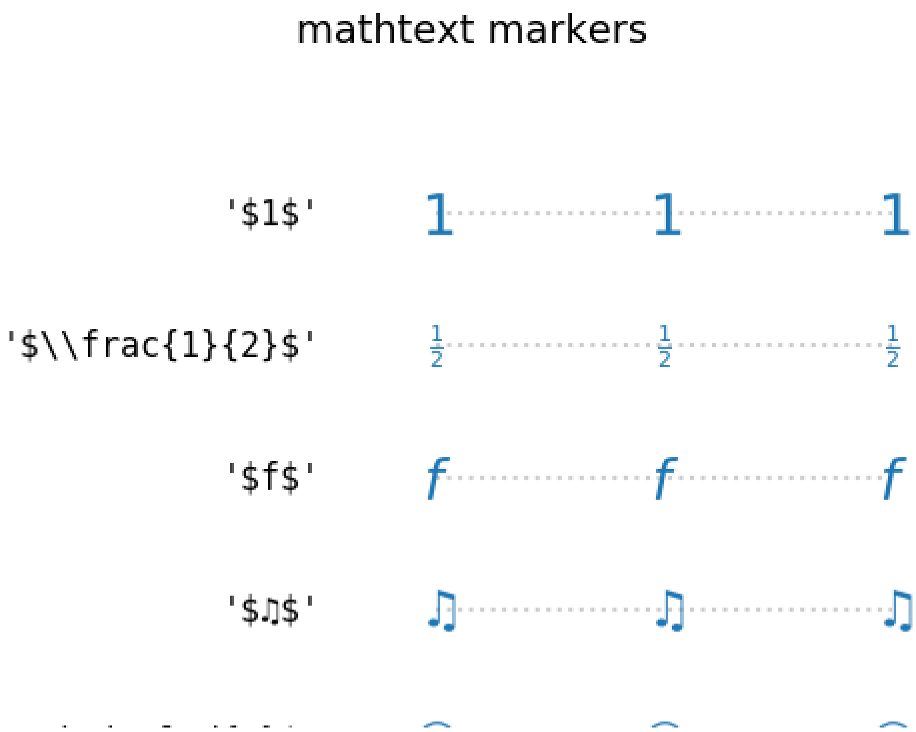

### Line styles




**Markers**

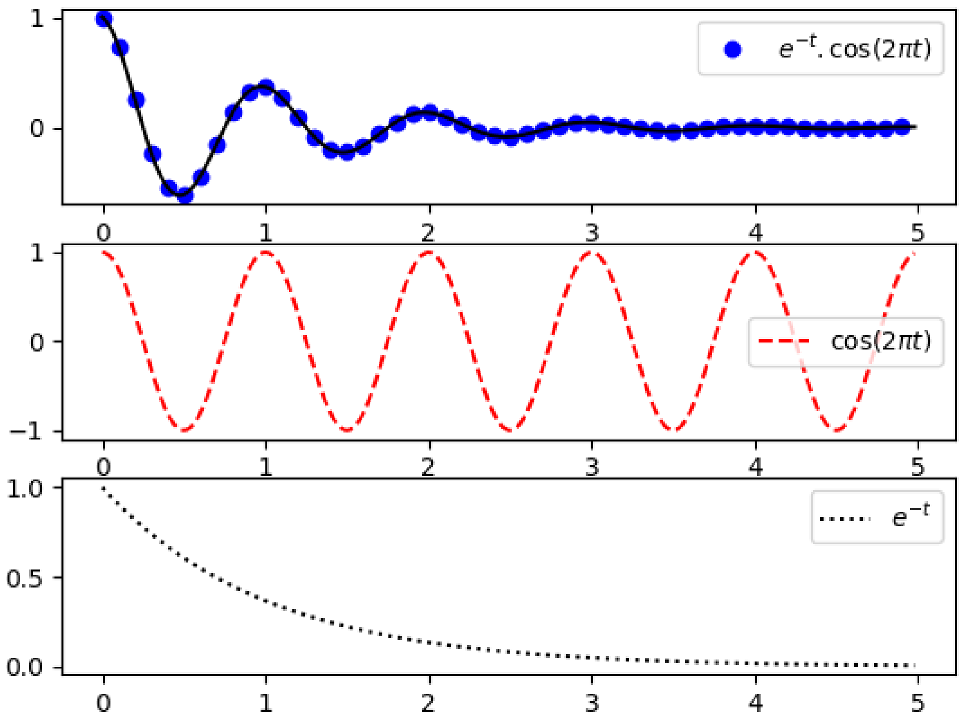



filled markers
'o'
'v'
'^'
'<'
'>'
'8'
's'
'p'
'*'
'h'
'H'
'D'
'd'
'P'
'X'


mathtext markers
'$1$'
'$\\frac{1}{2}$'
'$f$'
'$♫$'

In [58]:

```
# We first define a function we are interested in plotting
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

# These are the two series we will use as x-coordinates in the plot
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

# We open a new figure
plt.figure(3)
plt.subplot(311)
plt.plot(t1, f(t1), 'bo', label='$ e^{-t}.\cos(2 \pi t)$')
plt.plot(t2, f(t2), 'k')
plt.legend()

plt.subplot(312)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--', label='$ \cos(2 \pi t)$')
plt.legend()

plt.subplot(313)
plt.plot(t2, np.exp(-t2), 'k:', label='$ e^{-t}$')
plt.legend()
```




Out[58]:

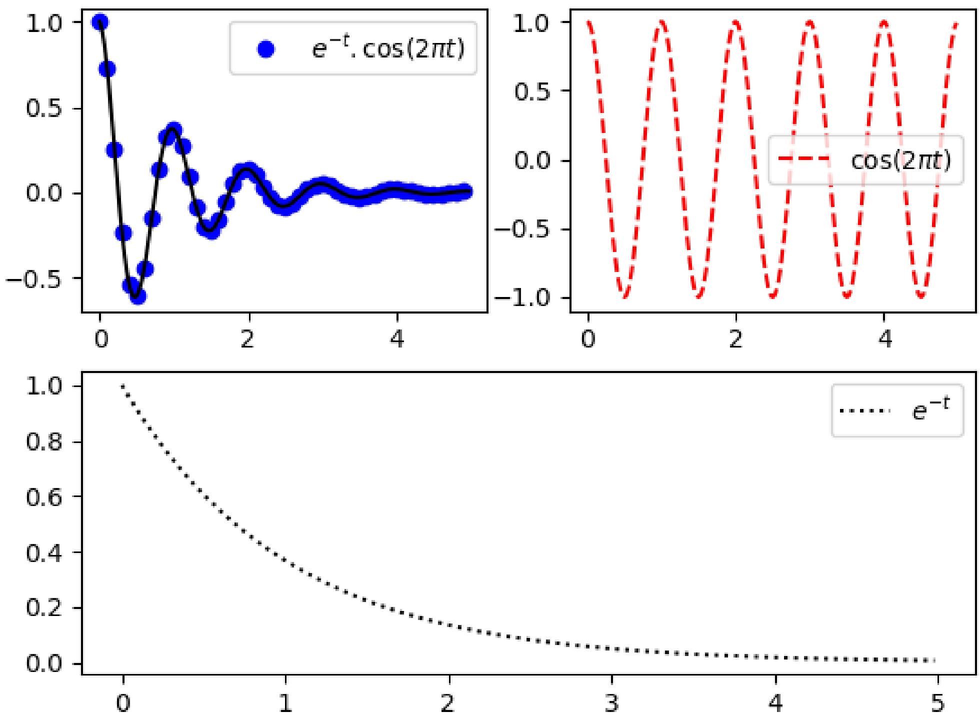
```
<matplotlib.legend.Legend at 0x1e0c8827d68>
```

We can also display the figures with one of them having different dimensions than the others. In the example below, we define a 2 by 2 grid and then use the image indices (1 to 3) to position them on the grid. Indices start at 1 in the upper-left corner and increase to the right.

So to define 3 sub-figures with two at the top and a large one at the bottom, we need to define a grid with 2 rows (in all sub-figures `subplot()` commands) and either 1 or two columns depending on the width of the sub-figure. Indices will allow us to position them as we want.

In [59]:

```
# We open a new figure

plt.figure(4)
plt.subplot(221) # in a 2*2 grid, this figure has index 1
plt.plot(t1, f(t1), 'bo', label='$ e^{-t}.\cos(2 \pi t)$')
plt.plot(t2, f(t2), 'k')
plt.legend()

plt.subplot(222) # in a 2*2 grid, this figure has index 2
plt.plot(t2, np.cos(2*np.pi*t2), 'r--', label='$ \cos(2 \pi t)$')
plt.legend()

plt.subplot(212) # in a 2*1 grid, this figure has index 2
plt.plot(t2, np.exp(-t2), 'k:', label='$ e^{-t}$')
plt.legend()
```




Out[59]:

```
<matplotlib.legend.Legend at 0x1e0c5d2e128>
```

### 4.3. 3D plots

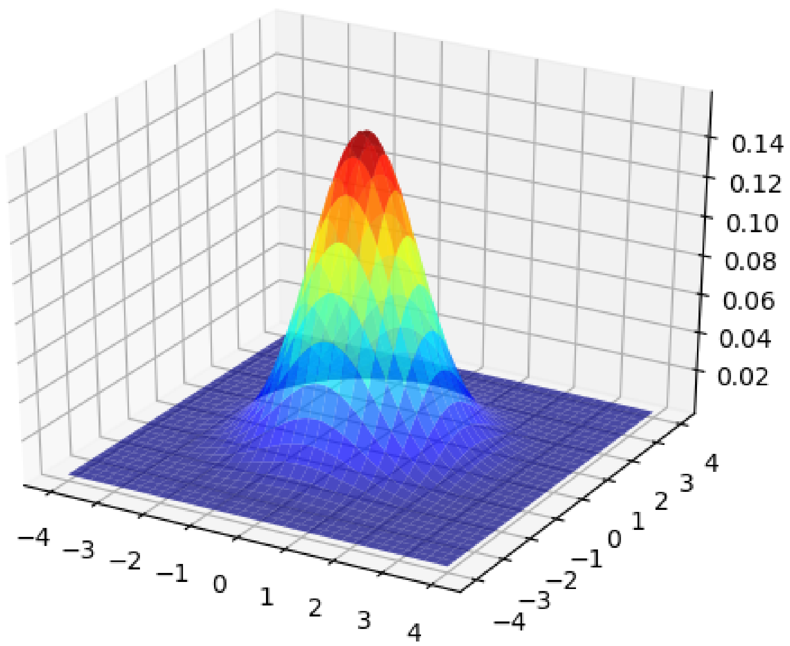
Matplotlib is excellent at producing beautiful 3D plots.

To generate a 3D plot, we need to import the `Axes3D` submodule and define the figure in slightly more abstract terms. For this, we use Python's object-oriented API (Application Programming Interface). So far, we have been creating graphs using a simpler, MATLAD-style API. This is convenient and it allowed us to do most things we were interested in, but this way of coding turns a blind eye on many other objects that are being created. Now, we create a figure canvas, and a set of axes explicitely. This requires somne additional typing, but it allows us to add a third axis for instance.

See below how to plot a bivariate normal distribution.

In [60]:

```
from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import multivariate_normal
from matplotlib import cm

x = np.linspace(-4, 4, 100)                    # Defining the [-2, 2] by [-2, 2] measure of
 our pdf
y = np.linspace(-4, 4, 100)
X, Y = np.meshgrid(x, y)                       # Creating a grid with the x, y coordinates c
reated above

mu = np.array([ 0., 0.])                      # Defining means of the bivariate standard nor
mal
sigma = np.array([[1., 0.], [0., 1.]])   # SD of the bivariate standard normal

# Putting X and Y into a 3 dimensional object (note: X and Y are 200*200 matrices and t
he third dimension is simply the number of such matrices, here two)
pos = np.zeros(X.shape + (2,))
pos[:, :, 0] = X
pos[:, :, 1] = Y

# Defining the density distribution we will use
F = multivariate_normal(mu, sigma)

# Storing the densities in a Z vector
Z = F.pdf(pos)

# Plotting the results in 3D
fig = plt.figure(5)
ax  = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z,
                rstride=3, cstride=3,
                cmap=cm.jet,
                alpha=0.7)
```




Out[60]:

```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at 0x1e0c9052be0>
```

We can now export our figure and close the figure windows we opened in order to de-clutter our memory. All vectorial and roster formats are supported.

In [61]:

```
plt.savefig('H:\\exampleDataHandling.pdf')

plt.close(1)
plt.close(2)
plt.close(3)
plt.close(4)
plt.close(5)
```

## 5. SciPy

SciPy is a powerful package for scientific computing. It builds on top of NumPy and allows us to perform the mathematical manipulations that would be required in a social science project: optimisation, integration, solving differential equations, finding solutions of polynomials, performing linear algebra operations, signal processing and some useful statistical functions.

We will rarely use SciPy's extensive capabilities in this course, so we will not discuss it further here. But you will certainly use it in your own work.

## References

Datasets on top income shares come from **Gabriel Zucman's website (http://gabriel-zucman.eu/)**.

**Saez, E and G. Zucman** (2019) "Progressive Wealth Taxation", *Brookings Papers on Economic Activity*, Fall 2019, forthcoming. Link (http://gabriel-zucman.eu/files/SaezZucman2019BPEA.pdf).