Table of Contents

---

1 IO: Files

- **File**: a *sequence* of data held in a *storage medium,* that can be either *volatile or not*
  - User data are written on the file according to a *custom organization* that reflects user's needs
  - **Data structure:** a collection of data elements organized in some way (e.g., list, dictionary, set)

A file is a custom, *permanent* data structure to hold data

- A file data structure can be used to hold and represent virtually anything ….
  - Image, in different formats, such as jpeg, png, svg, gif …
  - Data of interest: Genome maps, financial data, climate data, traffic logs, medical data….
  - Log about a running program, such as a Web server, or your own python program …
  - Data collected by sensors during experiments
  - Text of a novel, a poem, a song, …
  - Specification of a web page, in HTML, CSS, JavaScript, …
  - Notes about … anything!
  - ….

In [48]:
```
1 %ls
```

```
110-S21-19-Jupyter_notebooks.ipynb  TurtleCode/
110-S21-19-Jupyter_notebooks.pdf    Untitled.ipynb
IO_Files.ipynb                      accounts.txt
L0.py*                              cmu_logo.png
L1.py*                              data.txt
L10.py*                             data.txt~
L11.py*                             data2.txt
L12.py*                             employee_addresses.csv
L14.py*                             frog.txt
L15.py*                             hw05.py*
L17.py*                             price_list.csv
L1_1.py*                            scratch.py*
L8.py*                              tales_two_cities.txt
L9.py*                              untitled3.py*
Lab-1.py                            untitled4.py*
Mall_Customers.csv
```

In [49]:
```
1 %cat tales_two_cities.txt
```

```
CHAPTER I.
The Period


It was the best of times, it was the worst of times, it was the age of
wisdom, it was the age of foolishness, it was the epoch of belief, it
was the epoch of incredulity, it was the season of Light, it was the s
eason of Darkness, it was the spring of hope, it was the winter of des
pair, we had everything before us, we had nothing before us, we were
all going direct to Heaven, we were all going direct the other way--in
short, the period was so far like the present period, that some of its
noisiest authorities insisted on its being received, for good or for e
vil, in the superlative degree of comparison only.
```

CHAPTER I. The Period

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

In [ ]:
```
1 %cat IO_Files.ipynb
```

...

Text and Binary Files

Files can be broadly classified as:

- **Binary**
  - *Images*: jpg, png, gif, bmp, tiff, psd, ...
  - *Videos*: mp4, mkv, avi, mov, mpg, vob, ...
  - *Audio:* mp3, aac, wav, flac, mka, wma, ...
  - *Documents*: pdf, doc, xls, ppt, docx, odt, ...
  - *Archive*: zip, rar, 7z, tar, iso, ...
  - *Database*: mdb, accde, frm, sqlite, ...
  - *Executable*: exe, dll, so, class, ...
- **Text**
  - *Web tools*: html, xml, css, svg, json, ...
  - *Source code*: c, cpp, h, cs, js, py, java, php, sh, ...
  - *Documents*: txt, tex, markdown, asciidoc, rtf, ps, ...
  - *Configuration*: ini, cfg, rc, reg, ...
  - *Tabular data*: csv, tsv, ...

Text files: records and fields

- **Text:**
  - Human-readable: mostly composed of *printable* characters
  - Can be read / write using any text editor / viewer program
  - Organized in **multiple records** separated by **newline characters**
  - Each record is a piece of information, possibly structured in multiple **fields**

- Six records
- Four fields (at most):
  First name, Last name, ID, Sex

```
John Smith    642876 M
Adam Smith    787294 M
Ann White.    889220  F
Joan Black     627291  F
Mary Brown  78979
```

In [54]:
```
1 %cat accounts.txt
```

```
John Smith    642876 M
Adam Smith  787294 M
Ann White.    889220  F
Joan Black      627291  F
Mary Brown  78979 F
```

- Three records
- Variable number of fields per record

Old pond
Frog jumps in
Sound of water

In [9]:
```
1 %cat frog.txt
```

```
Old pond
Frog jumps in
Sound of water
```

- Four records
- Nine fields (at most), specified in the first record

STATION,STATION_NAME,ELEVATION,LATITUDE,LONGITUDE,DATE,HPCP,Measurement Flag,Quality Flag
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100101 00:00,99999,],
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100101 01:00,0,g,
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100102 06:00,1, ,

---

What type of **operations** can we do on / using a file (i.e., data storage)?

Data storage: storing data in a named location (a 'known' place, a ***file***) that can be accessed (open) later for read / write / update operations, and can put aside (closed) when done, and it can also be removed if stored data are not anymore needed

Create a named file

✓ **Open** a *new* file

✓ **Open** an *existing* file

- ✓ **Read** *existing* data
- ✓ **Write** *new* data
- ✓ **Modify / write** *existing* data

Temporary shutdown an existing file

✓ **Close** *existing* file

Delete an existing file from the system

✓ **Remove** an *existing* file

Open a file (and let's focus on text files): open( ) function

✓ While each OS has its own way to deal with file (**file system** of an OS) , Python achieves OS-independence by accessing a file through a **file handle** that holds the reference to the file in the system

➢ **Open** a file to make I/O on it, using the function:

```
file_handle = open(file_name, <modes>)
```

`TextIOWrapper` object type (**stream**) — `str` object type — `str` object type

```
f = open('data.txt', 'rt')
f = open('data.txt', 'r')
f = open('data.txt', 'w')
f = open('data.txt', 'r+')
f = open('data.txt', 'w+')
```

modes:
1. Read/Write
2. Text/Binary

`'r'`	open an existing file for reading (default)
`'w'`	open for writing, *truncating* file to 0 bytes if it exists, or creating a new file otherwise
`'x'`	create a new file and open it for writing
`'a'`	open for writing, *appending* to end of file if it exists, or creating a new file otherwise
`'+'`	open a file for updating (both reading and writing)
`'b'`	binary mode
`'t'`	text mode (default)

Default: `'rt'`

In [55]:
```
1 %ls *.txt *.csv
```

```
Mall_Customers.csv      data2.txt                 new.txt
accounts.txt            employee_addresses.csv    price_list.csv
data.txt                frog.txt                  tales_two_cities.txt
```

In [59]:
```
1 f = open('data.txt', 'tr')
```

In [60]:
```
1 f
```

Out[60]:
```
<_io.TextIOWrapper name='data.txt' mode='tr' encoding='UTF-8'>
```

The `'tr'` flag doesn't need to be there!

In [61]:
```
1 f = open('data.txt')
```

In [62]:
```
1 f
```

Out[62]:
```
<_io.TextIOWrapper name='data.txt' mode='r' encoding='UTF-8'>
```

In [66]:
```
1 %pwd
```

Out[66]:
```
'/Users/giannidicaro/Dropbox/CMU-Lectures/PrinciplesOfComputing-15110/
Code'
```

A **full path** (in the local file tree) can be specified to find a file

In [67]:
```
1 f = open('/Users/giannidicaro/workplan.tex')
```

Read from a file: `read( )` method

- **Read** data from a file open with the `r` or `r+` mode flag (or, also, `w+`, `a+`)

```
string_with_data = file_handle.read(number_of_bytes)
```

One byte == One character

The method read() reads at most number_of_bytes bytes from the file, from **current position** in file

In [20]:
```
1 %cat data.txt
```

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
New line: 0 3 5.5
```

In [68]:
```
f = open('data.txt', 'tr')
s = f.read(10)
```

In [69]:
```
s
```

Out[69]:
```
'This line '
```

In [17]:
```
print(s)
```

```
This line
```

In [21]:
```
s = f.read(8)
```

In [22]:
```
s
```

Out[22]:
```
'is 26 ch'
```

How to go *back* to a certain position?

`.seek(pos)`

Go back to the **beginning of the file?**

In [24]:
```
f.seek(0)
```

Out[24]:
```
0
```

In [25]:
```
f.read(20)
```

Out[25]:
```
'This line is 26 char'
```

In [27]:
```
f.read(1000)
```

Out[27]:
```
'acters\nLine 1: 0.1 5.4 2 4 20 .03\nLine 2: 1 3. 2.0 43 12 \nLine 3:
1 2 \nThis is the last record\nNew line: 0 3 5.5\n'
```

To read the **whole file** in a shot

In [30]:
```
f.seek(0)
s = f.read()
```

In [31]:
```
print(s)
```

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
New line: 0 3 5.5
```

Big MORAL: `strings` data types are important because we read everything into a string!

Read a file **line by line, record by record?**

In [33]:
```
f.seek(0)

for record in f:
    print(record, end = '')
```

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
New line: 0 3 5.5
```

In [34]:
```
f.seek(0)

for record in f:
    if '1' in record:
        print(record, end = '')
```

```
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
```

Read **one line** only: current line only

`readline()`

In [35]:
```
f.seek(0)
record = f.readline()
```

In [36]:
```
print(record)
```

```
This line is 26 characters
```

In [37]:
```
s = f.readline()
```

In [38]:
```
print(s)
```

```
Line 1: 0.1 5.4 2 4 20 .03
```

Read ALL lines:

`readlines()`

In [39]:
```
f.seek(0)
s = f.readlines()
s
```

Out[39]:
```
['This line is 26 characters\n',
 'Line 1: 0.1 5.4 2 4 20 .03\n',
 'Line 2: 1 3. 2.0 43 12 \n',
 'Line 3: 1 2 \n',
 'This is the last record\n']
```

A **list of strings**!!!

Newline character is included!

In [40]:
```
for line in s:
    print(line, end = '')
```

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```

In [ ]:
```

```

2 Write data into a file

`.write(string)`

- To write something into a file, the open function must be invoked with one of the mode flags: `w`, `a`, `x`, `r+`, `w+`, `a+`
- Opening with `w` *erases* file's content if file exists, writing starts at the (new) beginning
- Opening with `a` lets writing start at the end of the file (appending)
- Opening with `r+` lets writing start at the beginning of the file (overwriting)
- If a file doesn't exist, `w`, `a`, `x` will create it
- The `+` versions allow both writing and reading

In [44]:
```
f = open('data2.txt', 'w')
nbytes = f.write('New line: 0 3 5.5')
nbytes = f.write('Another new line: 1 2 3')

```

In [45]:
```
%cat data2.txt
```

```
New line: 0 3 5.5Another new line: 1 2 3
```

3 Close the file after use

Closing a file after use: `close()` method

➢ **Close** a file when no further operations are needed / allowed:

```
file_handle.close()
```

- Closing a file frees up used file resources (and let the file accessible for deleting/renaming by the OS)
- If a close() isn't explicitly called, python's garbage collector does eventually the job of closing the file
- Explicitly closing the file prevents the program to perform any (unwanted) further operations on file
- close() returns `None`

In [74]:
```
1 f = open('numbers.txt', 'w')
2 f.write('This file contains important data\n')
3 f.write('0 1 2 3 4\n')
4 f.write('4 3 2 1\n')
5 f.close()
```

In [75]:
```
1 s = f.read()
```

```
---------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
 last)
<ipython-input-75-5c4f2c28a2d3> in <module>
----> 1 s = f.read()

ValueError: I/O operation on closed file.
```

4 Dealing with errors

Dealing with errors: `try-except-else-finally` construct

- When an **error** occurs during the program, Python generates an **exception:** it generates an error type that identifies the exception and then **stops** the execution
- Exceptions can be handled using the **try statement** to avoid that the program does actually stop when an error occurs during the execution

- `try-except-else-finally` blocks:
  - ✓ The **try** block let executing a block of code that can potentially generate an exception
  - ✓ The **except** block let handling the error, if generated by the try block (i.e., what to do when an error occurs)
  - Optional
    - ✓ The **else** block let specifying a block of code that is executed if the try block *didn't generate any exception*
    - ✓ The **finally** block let executing the code, regardless of the result of the try- and except blocks.

```
y = 1
try:
    x /= 10
    y += x
except:
    print("x doesn't exist")
else:
    print('x:', x)
    del x
finally:
    print('y:', y)
```

Try / Except with files, simple version:

In [78]:
```
f = None
try:
    filename = 'xyz.txt'
    f = open(filename)
except:
    print('File ' + filename + ' not found or not readabl
print(f)
```

```
File xyz.txt not found or not readable!
None
```

In [79]:
```
try:
    filename = 'xyz.txt'
    f = open(filename)
except:
    print('File ' + filename + ' not found or not readabl
    print('Opening numbers.txt ...')
    f = open('numbers.txt')
print(f)
```

```
File xyz.txt not found or not readable!
Opening numbers.txt
<_io.TextIOWrapper name='numbers.txt' mode='r' encoding='UTF-8'>
```

5 Reading CSV files

CSV stands for **Comma Separated Values.** It's a quite flexible and compact for storing data.

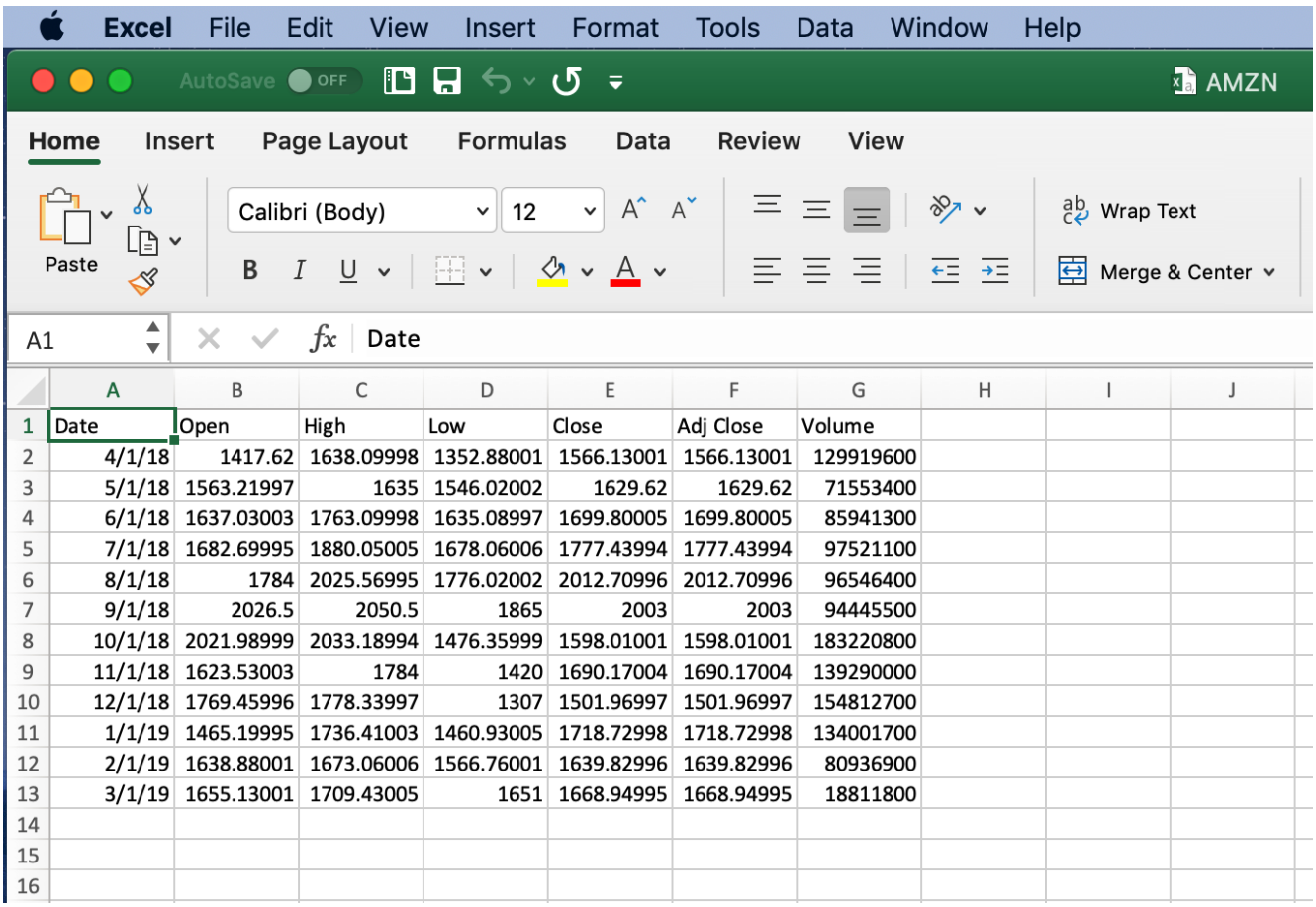
It's around since long time, and it's the main format used by popular spreadsheet programs such as Excel.

Many data repositories make use of CSV as one their standard formats for data.

E.g., a CSV file with monthly evolution of the Amazon's stock market prices at Nasdaq from Yahoo! Finance:

https://finance.yahoo.com/quote/AMZN/history?period1=1521362028&period2=1552898028&interval=1mo&filter=history&freque
(https://finance.yahoo.com/quote/AMZN/history?period1=1521362028&period2=1552898028&interval=1mo&filter=history&freque

Opening the file in **Excel**



	A	B	C	D	E	F	G	H	I	J
1	Date	Open	High	Low	Close	Adj Close	Volume			
2	4/1/18	1417.62	1638.09998	1352.88001	1566.13001	1566.13001	129919600			
3	5/1/18	1563.21997	1635	1546.02002	1629.62	1629.62	71553400			
4	6/1/18	1637.03003	1763.09998	1635.08997	1699.80005	1699.80005	85941300			
5	7/1/18	1682.69995	1880.05005	1678.06006	1777.43994	1777.43994	97521100			
6	8/1/18	1784	2025.56995	1776.02002	2012.70996	2012.70996	96546400			
7	9/1/18	2026.5	2050.5	1865	2003	2003	94445500			
8	10/1/18	2021.98999	2033.18994	1476.35999	1598.01001	1598.01001	183220800			
9	11/1/18	1623.53003	1784	1420	1690.17004	1690.17004	139290000			
10	12/1/18	1769.45996	1778.33997	1307	1501.96997	1501.96997	154812700			
11	1/1/19	1465.19995	1736.41003	1460.93005	1718.72998	1718.72998	134001700			
12	2/1/19	1638.88001	1673.06006	1566.76001	1639.82996	1639.82996	80936900			
13	3/1/19	1655.13001	1709.43005	1651	1668.94995	1668.94995	18811800			
14										
15										
16										

Opening the file with a **regular text editor/viewer**

jupyter AMZN.csv ✔ a few seconds ago

File Edit View Language

```
Date,Open,High,Low,Close,Adj Close,Volume
2018-04-01,1417.619995,1638.099976,1352.880005,1566.130005,1566.130005,129919600
2018-05-01,1563.219971,1635.000000,1546.020020,1629.619995,1629.619995,71553400
2018-06-01,1637.030029,1763.099976,1635.089966,1699.800049,1699.800049,85941300
2018-07-01,1682.699951,1880.050049,1678.060059,1777.439941,1777.439941,97521100
2018-08-01,1784.000000,2025.569946,1776.020020,2012.709961,2012.709961,96546400
2018-09-01,2026.500000,2050.500000,1865.000000,2003.000000,2003.000000,94445500
2018-10-01,2021.989990,2033.189941,1476.359985,1598.010010,1598.010010,183220800
2018-11-01,1623.530029,1784.000000,1420.000000,1690.170044,1690.170044,139290000
2018-12-01,1769.459961,1778.339966,1307.000000,1501.969971,1501.969971,154812700
2019-01-01,1465.199951,1736.410034,1460.930054,1718.729980,1718.729980,134001700
2019-02-01,1638.880005,1673.060059,1566.760010,1639.829956,1639.829956,80936900
2019-03-01,1655.130005,1709.430054,1651.000000,1668.949951,1668.949951,18811800
```

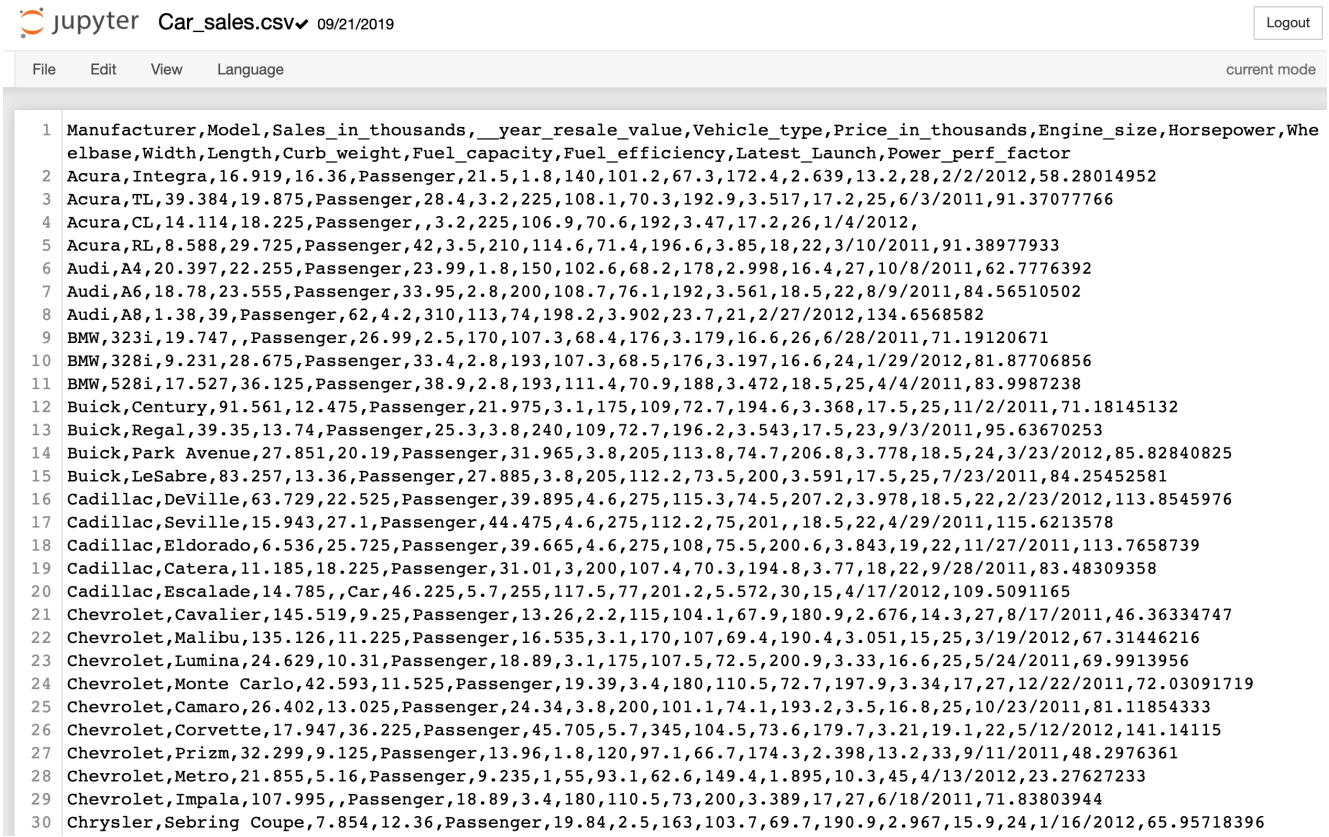
Example, a Kaggle dataset on car sales

Car_sales.csv (15.64 KB)

Detail Compact Column — 16 of 16 columns

Sales announcements with multiple info

Manufacturer	Model	Sales_in_thousands	__year_resale_value	Vehicle_type	Price_in_thousands	Engine_size	Horsepower	Wheelbase	Width	Length	Curb_weight
Dodge 7% / Ford 7% / Other (135) 86%	Neon 1% / Integra 1% / Other (154) 98%	0.11 – 541	5.16 – 67.5	Passenger 74% / Car 26%	9.23 – 85.5	1 – 8	55 – 450	92.6 – 139	62.6 – 79.9	149 – 225	1.9 – 5.57
Acura	Integra	16.919	16.36	Passenger	21.5	1.8	140	101.2	67.3	172.4	2.639
Acura	TL	39.384	19.875	Passenger	28.4	3.2	225	108.1	70.3	192.9	3.517
Acura	CL	14.114	18.225	Passenger		3.2	225	106.9	70.6	192	3.47
Acura	RL	8.588	29.725	Passenger	42	3.5	210	114.6	71.4	196.6	3.85
Audi	A4	20.397	22.255	Passenger	23.99	1.8	150	102.6	68.2	178	2.998
Audi	A6	18.78	23.555	Passenger	33.95	2.8	200	108.7	76.1	192	3.561
Audi	A8	1.38	39	Passenger	62	4.2	310	113	74	198.2	3.902
BMW	323i	19.747		Passenger	26.99	2.5	170	107.3	68.4	176	3.179
BMW	328i	9.231	28.675	Passenger	33.4	2.8	193	107.3	68.5	176	3.197
BMW	528i	17.527	36.125	Passenger	38.9	2.8	193	111.4	70.9	188	3.472
Buick	Century	91.561	12.475	Passenger	21.975	3.1	175	109	72.7	194.6	3.368
Buick	Regal	39.35	13.74	Passenger	25.3	3.8	240	109	72.7	196.2	3.543
Buick	Park Avenue	27.851	20.19	Passenger	31.965	3.8	205	113.8	74.7	206.8	3.778
Buick	LeSabre	83.257	13.36	Passenger	27.885	3.8	205	112.2	73.5	200	3.591
Cadillac	DeVille	63.729	22.525	Passenger	39.895	4.6	275	115.3	74.5	207.2	3.978
Cadillac	Seville	15.943	27.1	Passenger	44.475	4.6	275	112.2	75	201	
Cadillac	Eldorado	6.536	25.725	Passenger	39.665	4.6	275	108	75.5	200.6	3.843
Cadillac	Catera	11.185	18.225	Passenger	31.01	3	200	107.4	70.3	194.8	3.77
Cadillac	Escalade	14.785		Car	46.225	5.7	255	117.5	77	201.2	5.572
Chevrolet	Cavalier	145.519	9.25	Passenger	13.26	2.2	115	104.1	67.9	180.9	2.676
Chevrolet	Malibu	135.126	11.225	Passenger	16.535	3.1	170	107	69.4	190.4	3.051
Chevrolet	Lumina	24.629	10.31	Passenger	18.89	3.1	175	107.5	72.5	200.9	3.33

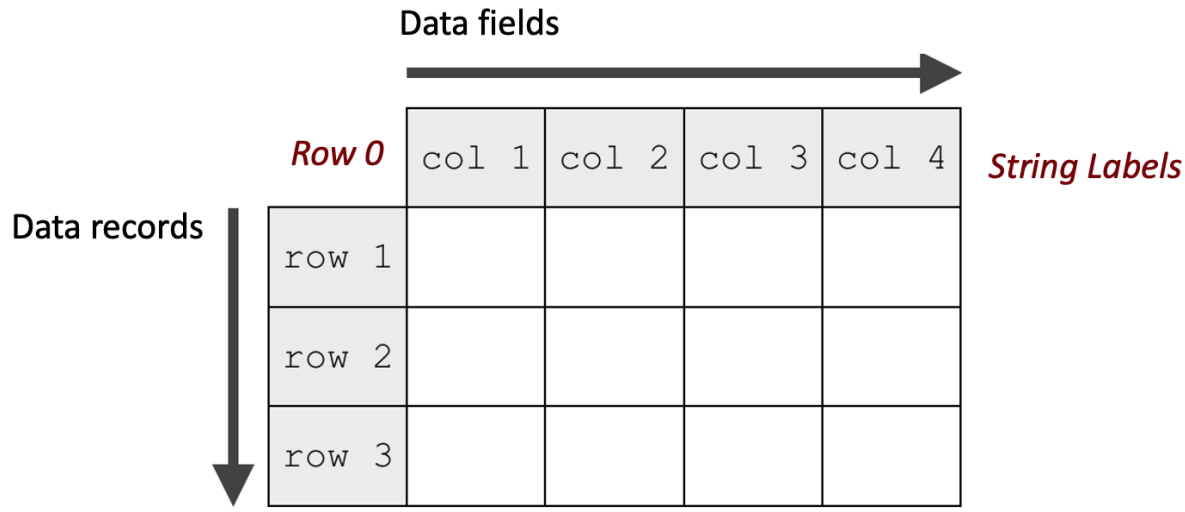


Datasets are commonly stored in **tabular form** (a *matrix* of data, a two-dimensional *data frame*): M data records/rows, each consisting of (at most) N ordered fields/columns

→ Each column/field describes the value of a property of interest.

→ Each row / record is a set of data making one **sample/example.**

→ The **first row,** optionally, can/should include **string labels**, explaining the meaning of the columns (i.e., giving a specific name to a column/attribute of data).



Both columns and rows (optionally) usually have explanatory **labels,** (e.g., 'Model', 'Manufacturer', 'Switzerland') or numeric indices (e.g., record #15) that are useful to refer to data, especially when manipulating them interactively.

Column data are separated by a given <u>delimiter.</u>

The default delimiter is a *comma,* but other characters can be used as a delimiter

→ The module `csv` makes available useful methods for dealing with CSV files

```
import csv
```

In [112]:

```
import csv

file_path = 'employee_addresses.csv'

f_csv = open(file_path)

csv_data = csv.reader(f_csv, delimiter=',')

```

In [101]:
```
1 csv_data
```

Out[101]:
```
<_csv.reader at 0x7ff20a70fb38>
```

`csv_data` is an **iterator:** at each call will return the **next line** in the file

Data records are read into **lists of strings**, where each list element is a string with a value identified based on the given delimiter

In [109]:
```
1 line_count = 0
2
3 for row in csv_data:
4     print(row)
5     line_count += 1
```

```
['name', 'address', 'date joined']
['john smith', '1132 Anywhere Lane Hoboken NJ', ' 07030', 'Jan 4']
['erica meyers', '1234 Smith Lane Hoboken NJ', ' 07030', '  March 2']
['ann mcdonald', ' 9223 Yoda Lane Pythonopolis CA', ' 90001', 'April
1']
```

We can use `sep.join(row)` to get / print out the whole row as a string with the desired separator, `sep`

In [110]:
```
1 f_csv.seek(0)
2
3 line_count = 0
4
5 for row in csv_data:
6     print(' '.join(row))
7     line_count += 1
```

```
name address date joined
john smith 1132 Anywhere Lane Hoboken NJ  07030 Jan 4
erica meyers 1234 Smith Lane Hoboken NJ  07030   March 2
ann mcdonald  9223 Yoda Lane Pythonopolis CA  90001 April 1
```

Remember that the **first line,** usually represents a **header,** that specifies the names of the fields in the records.

A `csv` file can be seen as a **dictionary:** each column has a label, hence, we can read the csv data file (or, more generically, tabular data) into an `ordered dictionary` , an dictionary that preserves/remembers the order for entering the keys. The keys are sorted by the order associated to their entrance in the dictionary.

Each row is an ordered dictionary with respect to the keys/columns.

→ Need to get the iterator using the call to `csv.DictReader()`

In [113]:
```
1 file_path = 'employee_addresses.csv'
2
3 f_csv = open(file_path)
4
5 csv_data = csv.DictReader(f_csv)
```

With `csv_data.filednames` we get the content of the first row, that is, the field names into a list of strings.

In [114]:
```
1 print(csv_data.fieldnames)
```

```
['name', 'address', 'date joined']
```

How many fields / attributes / columns do we have in the dataset?

In [119]:
```
1 num_of_keys = len(csv_data.fieldnames)
2 print('The file has ' + str(num_of_keys) + ' fields')
```

```
The file has 3 fields
```

The really useful thing is that we can now retrieve the individual fields as in a dictionary, using the keys!

In [130]:

```
f_csv.seek(0)

r_num = 1
next(csv_data)  # this reades / advances one record, the
for r in csv_data:
    print('Record n. ' + str(r_num) +  ' - ' + 'Address i
    r_num += 1

```

```
Record n. 1 - Address is: 1132 Anywhere Lane Hoboken NJ
Record n. 2 - Address is: 1234 Smith Lane Hoboken NJ
Record n. 3 - Address is:  9223 Yoda Lane Pythonopolis CA
```

Get the value of the name field if the address is in California

In [132]:

```
f_csv.seek(0)

r_num = 1
for r in csv_data:
    if 'CA' in r['address']:
        print('Record n. ' + str(r_num) +  ' - ' + 'Name
    r_num += 1


```

```
Record n. 4 - Name is: ann mcdonald
```

6 Read input from the keyboard

What about reading data directly from user input?

```
input()
```

In [135]:

```
n = input('Enter a number: ')

print('You have entered:', n)
```

```
Enter a number: 123
You have entered:  123
```

In [136]:

```
while True:
    n = input('Enter a number: ')

    if n.isnumeric():
        print('You have entered:', n)
        break
    else:
        print('Please enter a valid number!')

```

```
Enter a number: jadgf
Please enter a valid number!
Enter a number: 23
You have entered: 23
```

In [140]:

```
print('I\'ll make the sum of the number that you will ent
print('Enter 0 to end')

numbers = []
while True:
    n = input('Enter a number: ')

    if n == '0':
        print('Sum is: ' + str(sum(numbers)))
        break

    if n.isnumeric():
        numbers.append(int(n))
    else:
        print('Please enter a valid number!')

```

```
I'll make the sum of the number that you will enter!
Enter 0 to end
Enter a number: 2
Enter a number: 3
Enter a number: 5
Enter a number: 0
Sum is: 10
```