

LEARN CODING

ale66

FILES

A RECAP ON SEQUENCES

- lists and dictionaries are
- *mutable* data structures
- of variable length
- any elem. can be accessed via the `[]` notation:

`mylist[i]` or `mydict['mykey']`

- we iterate on them with `for` or `while`

RECAP L/D VS. D/L



```
1 my_data = {'name':['Andrea', 'Tom'],  
2           'age':[32,35],  
3           'location':['London', 'Brighton']  
4 }
```

```
1 my_data = [  
2     {'name':'Andrea', 'Age':32, 'location':'London'},  
3     {'name':'Tom', 'Age':35, 'location':'Brighton'}  
4 ]
```

FURTHER OBSERVATIONS

- data structures exist only at run-time: they disappear as soon as the execution ends
- data is made permanent via the *file system*
- files have a *format* which plays a similar role as types for variables guides interpretation, memorization and exchange

Yet, Python can open *any* file, even AV ones

File: a mere succession of characters, terminated by the special char. EOF

FILES

The OS maintains

- the hierarchical file system structure: folders
- unique names in each folder
- changeable content but permanent memorisation

There could be two or more copies of `pg100.txt`

but only one `C:/Users/aless/git/learn-code/60-files/pg100.txt`

HANDLING FILES WITH PYTHON

THE **with** INSTRUCTION

- copies the whole content from the permanent memory to the (volatile) work memory, and
- associates it with a *handler*

```
1  MYFILE = './data/incipt.txt'
2
3  with open(MYWORKFOLDER+MYFILE) as myhandler:
4
5      mytext = myhandler.read()
```

By default the file content is read as text, **.txt** is not considered

TEXT FILES

A natural organization in *rows*: sequences terminated by a *enter* character: `\n`

Also *space* is represented,
so is *tab* with `\t`

By default, reading a file returns a list of strings.

```
1 MYFILE = './data/incipt.txt'  
2  
3 with open(MYFILE) as f:  
4     mytext = f.read()
```

here `mytext` is a *blob* or mega-string.

```
1 with open(MYFILE) as f:  
2     mytext = f.read()  
3  
4 print(mytext)
```

The file handler `f` is accessible only from the block under `with`

THE WORKING DIRECTORY

VS Code normally starts in the top user folder/directory:

```
1 >pwd
2
3 C:/Users/ale/
```

Manually set the absolute path to the data files:

```
1 MYWORKFOLDER = 'C:/Users/ale/Downloads/git/learn-coding/60-files'
2
3 MYFILE = './data/incipt.txt'
4
5 with open(MYWORKFOLDER+MYFILE) as myhandler:
6
7     mytext = myhandler.read()
```

READING VS. WRITING

`f` represents the file itself in our code, in a way similar to iterators like `range(n)`

`f.read()` copies the whole text from the computer permanent memory to our `mytext` variable

Changes to `mytext` do not reflect on the file (see below)

READING TEXT FILES

receive a string and parse its characters, then
create Latin-style all-caps text:

```
1 with open(MYFILE) as f:
2     mytext = f.read()
3     # file f 'goes away'
4
5 for c in mytext:
6     print(c.upper())
```

LINE-ORIENTED PARSING

`\n` is used as separator to create a list of strings

```
1 with open(MYFILE) as f:
2     mytext = f.readlines()
3
4 for line in mytext:
5     print(line)
```

ITERATIVE FILE ACCESS

Handling large files is better done by having `f` as an iterator

Example: `pg100.txt`, the complete Shakespeare works

iterate over lines coming out of the handler (the default setup)

```
1 with open(MYFILE) as f:  
2     for line in f:  
3         print(line)
```

Notice how `print` adds an extra `\n` each time

WRITING ON FILES



To keep a permanent copy of our results we need to write them on a file

Writing is more complex than reading:

- create a new file, write into it
- append new text at the bottom of an existing file
- overwrite an existing file with new material (irreversible)

By default, `open()` simply reads

```
1 with open(MYFILE, 'r') as f:  
2     mytext = f.read()
```

parameter	effect
r	read
w	overwrite
x	create then write
a	append to existing f.

WRITING FILES

```
1 FILE = 'test-writing.txt'
2
3 with open(FILE, 'w') as f:
4
5     for num in range(10):
6         f.write(num)
```

0123456789

Printing is but writing into a special file which represents the output window

Use the same format rules

```
1 with open(FILE, 'w') as f:
2
3     for num in range(10):
4         f.write(f'This is value {num}\n')
```

CSV/TSV

A text file with extra assumptions on how data is organised

Each line is a data point, described by an *interpretation structure* that is normally on the first line

Let's have another look at `biostats.csv`

```
1 Name, Sex, Age, Height(in), Weight(lbs)
2 Alex, M, 41, 74, 170
3 Bert, M, 42, 68, 166
4 ...
5 Ruth, F, 28, 65, 131
```

Line 1 supplies the *keys* for a dictionary while further lines supply the values

, or `\t` separate values while `\n` separates datapoints (or *records*)

Often "s are used for text, e.g., `"Jean Jacques"`

Python supports CVS files via a extra *module* (details later)

```
1 import csv
2 FILE2 = './data/biostats.csv'
3
4 with open(FILE2) as f:
5
6     lines = csv.reader(f, delimiter=',')
7
8     for l in lines:
9         print(l)
```

```
['Name', 'Sex', 'Age', 'Height(in)', 'Weight(lbs)']
```

```
['Alex', 'M', '41', '74', '170']
```

```
['Bert', 'M', '42', '68', '166']
```

```
['Dave', 'M', '32', '70', '155']
```

```
['Dave', 'M', '39', '72', '167']
```

```
['Elly', 'F', '30', '66', '124']
```

```
['Fran', 'F', '33', '66', '115']
```

```
['Gwen', 'F', '26', '64', '121']
```

```
['Hank', 'M', '30', '71', '158']
```

```
['Luke', 'M', '53', '72', '175']
```

```
['Jake', 'M', '32', '69', '143']
```

```
['Kate', 'F', '47', '69', '139']
```

```
['Luke', 'M', '34', '72', '163']
```

github.com/ale66/learn-coding

The first line is special

```
1 # get the first line out
2 header = next(lines)
3
4 for l in lines:
5     print(f'Patient: {l}')
```

FROM CSV TO DICTIONARY

Function `DictReader` uses the first line to guide the creation of dictionaries

```
1 with open(FILE2) as f:
2
3     lines = csv.DictReader(f, delimiter=',')
4
5     for l in lines:
6         print(f'Patient: {l}')
```

```
Patient: {'Name': 'Alex', 'Sex': 'M', 'Age': '41', 'Height(in)': '74', 'Weight(lbs)': '170'}
```

```
Patient: {'Name': 'Bert', 'Sex': 'M', 'Age': '42', 'Height(in)': '68', 'Weight(lbs)': '166'}
```

```
Patient: {'Name': 'Dave', 'Sex': 'M', 'Age': '32', 'Height(in)': '70', 'Weight(lbs)': '155'}
```

```
Patient: {'Name': 'Dave', 'Sex': 'M', 'Age': '39', 'Height(in)': '72', 'Weight(lbs)': '167'}
```

```
Patient: {'Name': 'Elly', 'Sex': 'F', 'Age': '30', 'Height(in)': '66', 'Weight(lbs)': '124'}
```

```
Patient: {'Name': 'Fran', 'Sex': 'F', 'Age': '33', 'Height(in)': '66', 'Weight(lbs)':
```

```
'115']
```

```
Patient: {'Name': 'Gwen', 'Sex': 'F', 'Age': '26', 'Height(in)': '64', 'Weight(lbs)':
```

```
'115']
```

A list of key names can also be supplied, to facilitate data migration

```
1 first_line = ['Name', 'Sex', 'Age', 'Height(in)', 'Weight(lbs)']
2
3 mapping_es = ['Nombre', 'Sexo', 'Edad', 'Estatura(in)', 'Peso(lbs)']
```

```
1 with open(FILE2) as f:
2     lines = csv.DictReader(f, fieldnames = mapping_es, delimiter = ',')
3
4     for l in lines:
5         print(f'Paciente: {l['Nombre'], l['Edad']})')
```

```
Paciente: ('Name', 'Age')
```

```
Paciente: ('Alex', '41')
```

```
Paciente: ('Bert', '42')
```

```
Paciente: ('Dave', '32')
```

```
Paciente: ('Dave', '39')
```

```
Paciente: ('Elly', '30')
```

```
Paciente: ('Fran', '33')
```

```
Paciente: ('Gwen', '26')
```

```
Paciente: ('Hank', '30')
```

```
Paciente: ('Luke', '53')
```

```
Paciente: ('Jake', '32')  
Paciente: ('Kate', '47')  
Paciente: ('Luke', '34')
```

DISCUSSION

CSV/TSV make exchanging data fast and reliable

However, they assume that for each datapoint we a *fixed* description that will fill the exact number of columns

Lack of data implies filling a *placeholder* or `null` value

Bert is NOT 68 years old

```
1 Name, Sex, Age, Height(in), Weight(lbs)
2 Alex, M, 41, 74, 170
3 Bert, M, NULL, 68, 166
```

But what if we know Alex's shoe size and Bert's lung capacity (and not vice versa)

JSON

FROM CSV TO JSON BY EXAMPLE

JSON is essentially *a list of nested Python dictionaries*.

Different levels of details are easily accomodated

so do data thas is naturally non-atomic, e.g., `passed_exams`

```
1  [{
2      "Financial Institution": "Financial Institution",
3      "ABANCACorporacinBancariaS.txt": {
4          "energy": 51,
5          "environmental": 32.5378277861242,
6          "management": 15.73553116878063,
7          "party": 35.37153650105044,
8          "buildings": 1.1823215567939547,
9          "sustainability": 29.487406431175053
10     },
11     // more and more...
12  }]
```

```
1 [{  
2     "Financial Institution": "Financial Institution",  
3     "ABANCACorporacinBancariaS.txt": {},  
4     "ABN_AMRO_2015_External_review_report.txt": {},  
5 }]
```

```
1 [{
2     "Financial Institution": "Financial Institution",
3     "ABANCACorporacinBancariaS.txt": {},
4     "ABN_AMRO_2015_External_review_report.txt": {
5         "ABN_AMRO_2015_External_review_report.txt": {"energy": 89,
6             "environmental": 34.57144202275696,
7             "management": 7.867765584390315,
8             "party": 7.737523609604784,
9             "buildings": 141.87858681527456,
10            "sustainability": 39.655477614338864},
11     }
12 }]
```

JSON FORMAT RECAP

From the point of view of Python, a JSON object is

- a dictionary `{"first_name": "John", "last_name": "Smith", ...}`
- a list of dictionaries `[{"first_name": "John", ...}, {...}, {...}]`

where each value is either a Boolean, a number, a string or list or a dictionary.

These examples are from Wikipedia

```
1 {"first_name": "John",
2  "last_name": "Smith",
3  "address": {
4    "street_address": "21 2nd Street",
5    "city": "New York",
6    "state": "NY",
7    "postal_code": "10021-3100"
8  }, ...}
```

```
1 {"first_name": "John",
2  "last_name": "Smith",
3  "phone_numbers": [
4    {
5      "type": "home",
6      "number": "212 555-1234"
7    },
8    {
9      "type": "office",
10     "number": "646 555-4567"
11   }
12 ], ...}
```

NON-TEXT FILES

By default, `open()` simply reads text files

```
1 with open(MYFILE, 'tr') as f:  
2     mytext = f.read()
```

parameter	effect
t	text
w	binary
+	double read/write use

Images, sound and video are treated as binary

DATA CLEANING AND DATA WRANGLING

An informal introduction through a **real project** on *Green finance*: look at cells up to [5].

CHALLENGE: CAN YOU GET FRESH DATA FROM [DATA.SPECTATOR.CO.UK](https://data.spectator.co.uk) AND DISPLAY IT?



The Spectator data tracker. Updated daily.

- HOME
- SCHOOLS
- POLLS
- ECONOMY ▾
- HEALTH ▾
- WELFARE
- MIGRATION
- SCOTLAND ▾
- ENERGY ▾
- DEFENCE
- MISC ▾
- NET ZERO

⚡ Electricity prices

In pence per kilowatt hour, including taxes, in 2023

Industrial Domestic

Green energy price support

Monthly Contracts for Difference (CfD) payments made to producers of low-carbon energy in the UK

Monthly Cumulative total