LEARN CODING

ale66

github.com/ale66/learn-coding

FILES

A RECAP ON SEQUENCES

- lists and dictionaries are
- *mutable* data structures
- of variable lenght
- 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

```python
my_data = {'name':['Andrea', 'Tom'],
           'age':[32,35],
           'location':['London', 'Brighton']
}
```

```python
my_data = [
           {'name':'Andrea', 'Age':32, 'location':'London'},
           {'name':'Tom', 'Age':35, 'location':'Brighton'}
]
```

FURTHER OBSERVATIONS

- Python run-time data structures are *impermanent*: they disappear as soon as the execution ends
- data is made permanent con computer devices via the *file system*
- files have *format:*

  f. plays a similar rolet to variable types:

  g. guides interpretation, memorization and exchange

However, in principle Python can open and read any file,

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 ot `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 permanet memory to the (volatile) work memory, and
- associates it with a *handler*

```python
MYFILE = './data/incipt.txt'

with open(MYFILE) as myhandler:
  mytext = f.read()
```

- the known file format determines what to do next

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.

```python
MYFILE = './data/incipt.txt'

with open(MYFILE) as f:
  mytext = f.read()
```

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

```python
with open(MYFILE) as f:
    mytext = f.read()

print(mytext)
```

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

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:

```python
with open(MYFILE) as f:
    mytext = f.read()
# file f 'goes away'

for c in mytext:
    print(c.upper())
```

LINE-ORIENTED PARSING

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

```python
with open(MYFILE) as f:
    mytext = f.readlines()

for line in mytext:
  print(line)
```

ITERATIVE FILE ACCESS

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

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

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

```python
with open(MYFILE) as f:
  for line in f:
    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

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

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

WRITING FILES

```python
FILE = 'test-writing.txt'

with open(FILE, 'w') as f:

    for num in range(10):
        f.write(num)
```

0123456789

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

Use the same format rules

```python
with open(FILE, 'w') as f:

    for num in range(10):
        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`

```
Name, Sex, Age, Height(in), Weight(lbs)
Alex, M, 41, 74, 170
Bert, M, 42, 68, 166
...
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)

```python
import csv
FILE2 = './data/biostats.csv'

with open(FILE2) as f:

    lines = csv.reader(f, delimiter=',')

    for l in lines:
        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']
```

The first line is special

```python
# get the first line out
header = next(lines)

for l in lines:
  print(f'Patient: {l}')
```

FROM CSV TO DICTIONARY

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

```python
with open(FILE2) as f:

    lines = csv.DictReader(f,  delimiter=',')

    for l in lines:
      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)':
```

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

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

mapping_es = ['Nombre', 'Sexo', 'Edad', 'Estatura(in)', 'Peso(lbs)']
```

```python
with open(FILE2) as f:
  lines = csv.DictReader(f,  fieldnames = mapping_es, delimiter = ',')

  for l in lines:
    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 colums

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

Bert is NOT 68 years old

```
Name, Sex, Age, Height(in), Weight(lbs)
Alex, M, 41, 74, 170
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`

```
[{
    "Financial Institution": "Financial Institution",
    "ABANCACorporacinBancariaS.txt": {
        "energy": 51,
        "environmental": 32.5378277861242,
        "management": 15.73553116878063,
        "party": 35.37153650105044,
        "buildings": 1.1823215567939547,
        "sustainability": 29.487406431175053
    },
    // more and more...
}]
```

```json
[{
    "Financial Institution": "Financial Institution",
    "ABANCACorporacinBancariaS.txt": {},
    "ABN_AMRO_2015_External_review_report.txt": {},
}]
```

```
[{
    "Financial Institution": "Financial Institution",
    "ABANCACorporacinBancariaS.txt": {},
    "ABN_AMRO_2015_External_review_report.txt": {
      "ABN_AMRO_2015_External_review_report.txt": {"energy": 89,
      "environmental": 34.57144202275696,
      "management": 7.867765584390315,
      "party": 7.737523609604784,
      "buildings": 141.87858681527456,
      "sustainability": 39.655477614338864},
    }
}]
```

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

```json
{"first_name": "John",
 "last_name": "Smith",
 "address": {
    "street_address": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postal_code": "10021-3100"
  }, ...}
```

```json
{"first_name": "John",
 "last_name": "Smith",
 "phone_numbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ], ...}
```

NON-TEXT FILES

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

```python
with open(MYFILE, 'tr') as f:
  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/category/energy and diplay it?