

Python Basic Course

Part II

Stefano Alberto Russo

Outline

- Part I: introduction and basics

- What is Python
- Tools and “hello world”
- Basic syntax and data types
 - assignments, types and operators
 - conditional blocks and loops

- Part II: architecture

- Functions
- Scope
- Built-ins
- Modules

- Part IV: manipulating data

- List operations
- String operations
- Dealing bad data
- Reading and writing files

- Part VI: Pandas

- Series and Dataframes
- Common operations
- How to read documentation

List operations

→ *Appending*

- Appending an element to the end:

```
my_list = [13,14,15,16]  
my_list.append(22)  
print(my_list)
```

```
[13, 14, 15, 16, 22]
```

List operations

→ *Popping*

- Popping an element (from the end):

```
my_list = [13,14,15,16]  
print(my_list.pop())  
print(my_list)
```

```
16  
[13, 14, 15]
```

List operations

→ *Slicing*

- Slicing a list uses the “double column” notation:

```
my_list = [13,14,15,16]  
print(my_list[1:3])
```

```
[14, 15]
```

List operations

→ *Slicing*

- Slicing a list uses the “double column” notation:

```
my_list = [13,14,15,16]  
print(my_list[:3])
```

```
[13, 14, 15]
```

List operations

→ *Slicing*

- Slicing a list uses the “double column” notation:

```
my_list = [13,14,15,16]  
print(my_list[2:])
```

```
[15, 16]
```

List operations

→ *Slicing*

- Slicing a list uses the “double column” notation:

```
my_list = [13,14,15,16]  
print(my_list[:-1])
```

```
[13, 14, 15]
```


List operations

→ Concatenation

- I can concatenate two lists

```
my_list_1 = [13,14,15,16]  
my_list_2 = [21,22]  
print(my_list_1 + my_list_2)
```

```
[13, 14, 15, 16, 21, 22]
```

String operations

→ *Slicing*

- Slicing a string uses the same “double column” notation as the list:

```
my_string = 'hello'  
print(my_string[1:3])
```

```
el
```

String operations

→ *Access by negative index*

- I can easily access the last character

```
my_string = 'hello'  
print(my_string[-1])
```

o

String operations

→ Concatenation

- I can easily concatenate strings

```
my_string_1 = 'hello'  
my_string_2 = 'anyone'  
print('Hey ' + my_string_1 + ' ' + my_string_2 + '!')
```

Hey hello anyone!

String operations

→ *Formatting*

- ...or format them:

```
my_string_1 = 'hello'  
my_string_2 = 'anyone'  
print('Hey {} {}!'.format(my_string_1, my_string_2))
```

```
Hey hello anyone!
```

String operations

→ *Replacing*

- I can also replace parts of the string

```
my_string = 'Hello world!'
print(my_string.replace('world', 'anyone'))
```

```
Hello anyone!
```

String operations

→ *Uppercasing*

- ..or make it uppercase

```
my_string = 'Hello world!'
print(my_string.upper())
```

HELLO WORLD!

String operations

→ *Splitting*

- The split is very useful when parsing data:

```
my_string = 'Hello anyone'  
print(my_string.split(' '))
```

```
['Hello', 'anyone']
```


String operations

→ *Cleaning*

- I can remove leading and trailing spaces, tabs and newline characters

```
my_string = '\t Hello world! \n'  
print('My string: "{}"'.format(my_string))  
print('My string: "{}"'.format(my_string.strip()))
```

```
My string: "      Hello world!  
"  
My string: "Hello world!"
```

String operations

→ *Conversions*

- I can convert strings to the numbers they naturally represent

```
my_string = '5.76'  
my_number = float(my_string)
```

String operations

→ *Conversions*

- When I convert (cast), the strip() function is automatically applied

```
my_string = ' 5.76 \n'  
my_number = float(my_string)
```

String operations

→ Conversions

- If the conversion fails, I have a value error:

```
my_string = '3,14'  
my_number = float(my_string)
```

```
Traceback (most recent call last):  
  File "main.py", line 2, in <module>  
    my_number = float(my_string)  
ValueError: could not convert string to float: '3,14'
```

String operations

→ *Conversions*

- If the conversion fails, I have a value error:

```
my_string = None  
my_number = float(my_string)
```

```
Traceback (most recent call last):  
  File "main.py", line 2, in <module>  
    my_number = float(my_string)  
TypeError: float() argument must be a string or a  
number, not 'NoneType'
```

Dealing with bad data

→ Check for type

- I can check for type before the conversion to prevent crashes:

```
my_var = 3
if type(my_var) not in [int, float, str]:
    print('Cannot use or convert type "{}".format(type(my_var)))
else:
    my_number = float(my_var)
```

Cannot use or convert type "<class 'NoneType'>"

Dealing with bad data

→ Check for type

- I can check also check for value before the conversion to prevent crashes:

```
my_var = 'a'
if type(my_var) not in [int, float, str]:
    print('Cannot use or convert type "{}"'.format(type(my_var)))
else:
    if type(my_var) == str and not my_var.isnumeric():
        print('Cannot convert value "{}"'.format(my_var))
    else:
        my_number = float(my_var)
```

Cannot convert value "a"

Dealing with bad data

→ *Check for values*

- If the conversion fails, I have a value error:

```
my_string = '3,14'  
my_number = float(my_string)
```

```
Traceback (most recent call last):  
  File "main.py", line 2, in <module>  
    my_number = float(my_string)  
ValueError: could not convert string to float: '3,14'
```


Dealing with bad data

→ *How to ask for forgiveness...*

- However, the best way is usually to always try to do the conversion and handle the error when and if this occurs.
- This is done using the try- except logic, but we will not cover it here.
- Just FYI:

```
try:  
    float(my_var)  
except:  
    # Handle the error
```

Reading and writing files

→ *What is a file, after all?*

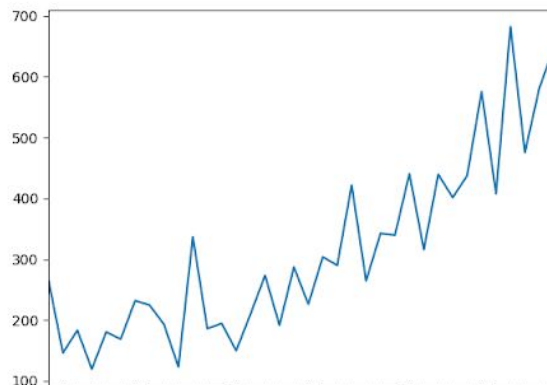
- Files are convenient structures where to store data
- A common format for this is the CSV, which stands for Comma-Separated Values.
- Usually, for CSV files each line is an “entry”, which has “fields”
- An optional header might or might not be present on top.
 - CSV files have several “dialects”, for example when values are separated with a semicolon instead of a comma.

Reading and writing files

→ *The shampoo sales*

- We will use a simple example: The shampoo sales over a few years.

```
Date,Sales
01-01-2012,266.0
01-02-2012,145.9
01-03-2012,183.1
01-04-2012,119.3
...
```



Reading and writing files

→ *How to read a file in Python*

- There are various ways, as for everything. The “open” function can open the file and return a file data type, that I can read in one go or line by line:

```
my_file = open('shampoo_sales.csv', 'r')  
print(my_file.read())  
my_file.close()
```

```
Date,Sales  
01-01-2012,266.0  
01-02-2012,145.9  
01-03-2012,183.1  
...
```

Reading and writing files

→ *How to read a file in Python*

- There are various ways, as for everything. The “open” function can open the file and return a file data type, that I can read in one go or line by line:

```
my_file = open('shampoo_sales.csv', 'r')  
print(my_file.readline())  
print(my_file.readline())  
my_file.close()
```

```
Date,Sales
```

```
01-01-2012,266.0
```

```
...
```

Reading and writing files

→ *How to read a file in Python*

- There are various ways, as for everything. The “open” function can open the file and return a file data type, that I can read in one go or line by line:

```
my_file = open('shampoo_sales.csv', 'r')  
for line in my_file:  
    print(line)  
my_file.close()
```

```
Date,Sales
```

```
01-01-2012,266.0
```

```
...
```

Reading and writing files

→ *How to read a file in Python*

- Using also the “with” statement will automatically handle closing the file for us when we are done with it:

```
with open('shampoo_sales.csv') as my_file:  
    for line in my_file:  
        print(line)
```

```
Date,Sales  
01-01-2012,266.0  
...
```

Reading and writing files

→ *How to read a file in Python*

- If I want to write to a file, I need to open it in write mode ("w"). To instead add content to an existing file, I have to open it in append mode ("a").

```
with open('new_file.csv', 'w') as my_file:  
    my_file.write('2013,22\n')  
    my_file.write('2014,27\n')  
    my_file.write('2015,25\n')
```

```
2013,22  
2014,27  
2015,25
```


End of part III

→ *Questions?*

Next: exercise 3

Exercise 3

Write a function that sums all the values of a CSV file.

- Name it “**sum_csv**” and accept a parameter for the file name. Return the sum.
- You can assume that files always have:
 - a first header line with the labels (to be skipped)
 - a date as the first item of every line (to be skipped)
 - a single data column (to be summed)
- If the file is empty, the function must return “None”
- If there are values that cannot be converted to a numerical type, ignore the line without crashing the code