



Politecnico
di Torino



Data Science Lab

Python programming

Andrea Pasini
Flavio Giobergia
Elena Baralis

DataBase and Data Mining Group



- **Python language**
 - Python data types
 - Controlling program flow
 - Functions
 - Lambda functions
 - List comprehensions
 - Classes
- **Structuring Python programs**



- Python is an **object oriented** language
- Every piece of data in the program is an **Object**
 - Objects have **properties** and **functionalities**
 - Even a simple **integer** number is a Python **object**

Example of an integer object

```
type: int  
id: 140735957856544  
value: 3
```



- **Reference** = **symbol** in a program that refers to a particular **object**
- A single Python object can have **multiple references (alias)**

references

x

y

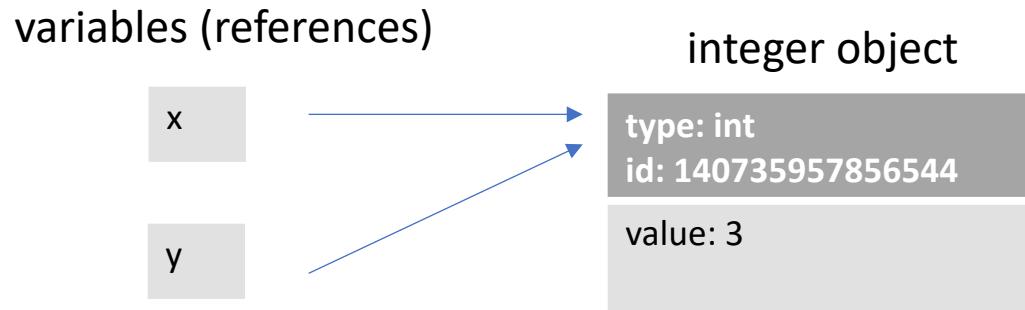
integer object

type: int
id: 140735957856544

value: 3



- In Python
 - **Variable** = **reference** to an object
- When you **assign** an object to a variable it becomes a **reference** to that object





- **Defining a variable**
 - **No need** to specify its data type
 - **Just assign** a value to a new variable name

```
a = 3
```

a



type: int

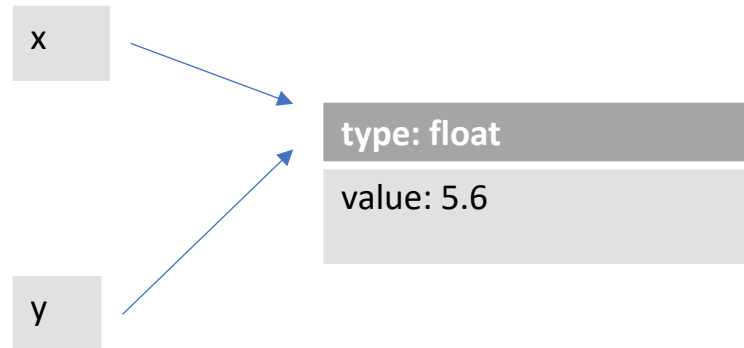
id: 140735957856544

value: 3



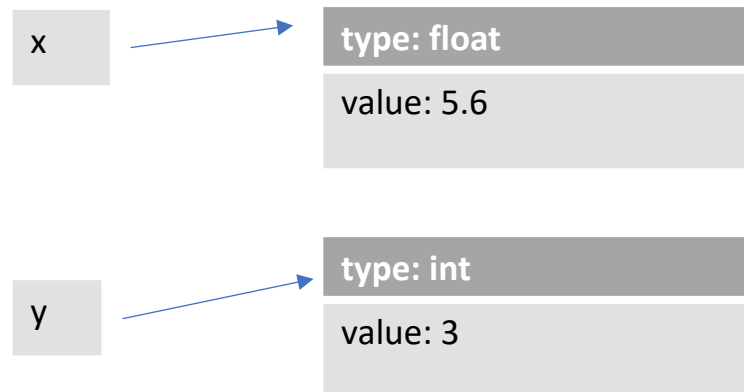
■ Example

```
x = 5.6  
y = x
```



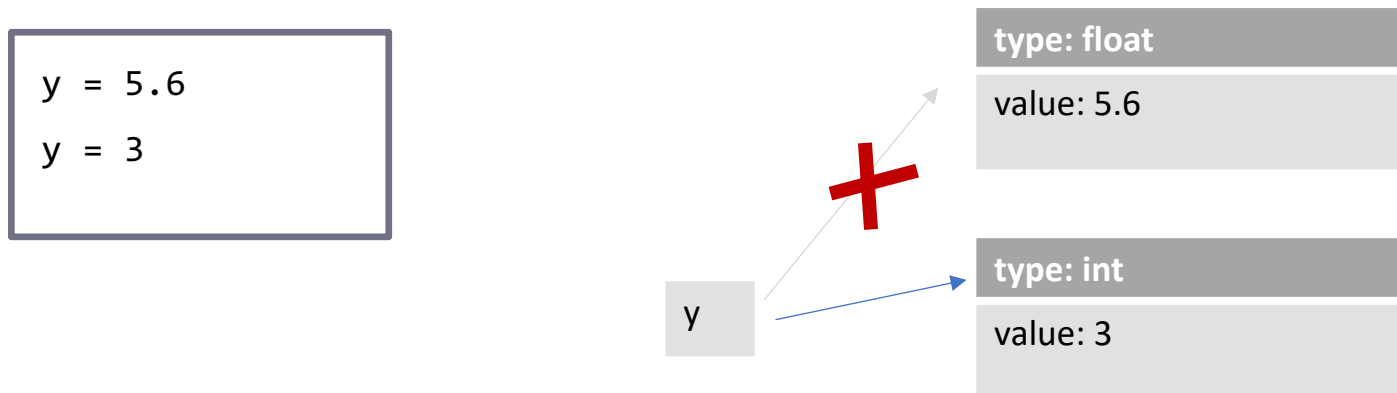
■ If you assign `y` to a new value...

```
y = 3
```





- From the previous example we learn that:
 - Basic data types, such as integer and float variables are **immutable**:
 - Assigning a new number will not change the value inside the object by rather create a new one





- Verify this reasoning with `id()`
 - **`id(my_variable)`** returns the **identifier** of the object that the variable is referencing

my_variable



type: int

id: 140735957856544

value: 3



■ Jupyter example

- Type in your code



```
In [1]: x = 1  
        y = x  
        print(id(x))  
        print(id(y))
```

- Press CTRL+ENTER to run and obtain a result

```
Out[1]: 140735957856544  
        140735957856544
```



- **Basic data types**
 - *int, float, bool, str*
 - *None*
 - All of these objects are **immutable**
- **Composite data types**
 - *tuple* (**immutable** collections of objects)
 - *list, set, dict* (**mutable** collections of objects)



■ int, float

- No theoretical size limit
 - Effectively limited by memory available
- Available operations
 - `+`, `-`, `*`, `/`, `//` (integer division), `%` remainder, `**` (exponentiation)
 - Example

In [1]:

```
x = 9
y = 5
r1 = x // y      # r1 = 1
r2 = x % y       # r2 = 4
r3 = x / y       # r3 = 1.8
r4 = x ** 2      # r4 = 81
```

- Note that dividing 2 **integers** yields a **float**



■ **bool**

- Can assume the values True, False
- Boolean operators: **and**, **or**, **not**
 - Example

```
In [1]: is_sunny = True
        is_hot = False
        is_rainy = not is_sunny                # is_rainy = False
        bad_weather = not (is_sunny or is_hot) # bad_weather = False

        temperature1 = 30
        temperature2 = 35
        raising = temperature2 > temperature1  # raising = True
```



■ String



```
In [1]: string1 = "Python's nice"           # with double quotes
        string2 = 'He said "yes"'         # with single quotes
        print(string1)
        print(string2)
```

```
Out[1]: Python's nice
        He said "yes"
```

- Definition with single or double quotes is equivalent



■ Conversion between types

■ Example



```
In [1]: x = 9.8
        y = 4
        r1 = int(x)           # r1 = 9
        r2 = float(y)        # r2 = 4.0
        r3 = str(x)          # r3 = '9.8'
        r4 = float("6.7")    # r4 = 6.7
        r5 = bool("True")    # r5 = True
        r6 = bool("False")   # r6 = True :(
        r7 = bool(0)         # r7 = False
```

- Only `0`, `""`, `[]`, `{}`, `set()`, `()` convert to `False` through `bool()`



■ Working with strings



- **string[i]**: get i-th character of string (0-indexed)
- **len**: get string length
- **strip**: remove leading and trailing spaces (tabs or newlines)
- **upper/lower**: convert uppercase/lowercase
- Full list → <https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

In [1]:

```
s1 = ' My string '  
length = len(s1)           # length = 11  
s2 = s1.strip()            # s2 = 'My string'  
s3 = s1.upper()            # s3 = ' MY STRING '  
s4 = s1.lower()            # s4 = ' my string '
```

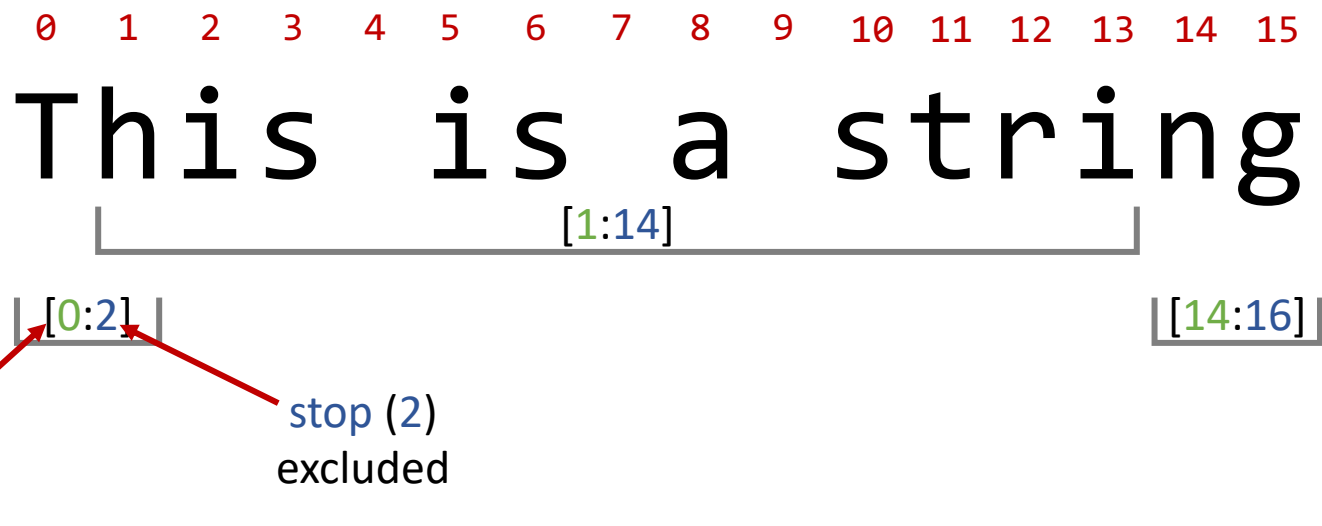



■ Sub-strings



■ `string[start:stop]`

- The `start` index is **included**, while `stop` index is **excluded**
- Index of characters starts **from 0**
- We can optionally specify a step `string[start:stop:step]` (*)





■ Shortcuts

- **Omit start** if you want to start from the beginning
- **Omit stop** if you want to go until the end of the string

In [1]:

```
s1 = "Hello"
charact = s1[0]           # charact = 'H'
s2 = s1[0:3]              # s2 = 'Hel'
s3 = s1[1:]               # s3 = 'ello'
s4 = s1[:3]               # s4 = 'Hell'
s5 = s1[:]                # s5 = 'Hello'
```



■ Sub-strings

■ Negative indices:

- count characters **from the end**
- **-1 = last character**



-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

This is a string

`[-15:-2]`

`[0:-14]`

`[-2:16]`

`[:-1]`



■ Sub-strings

■ Negative indices:

- count characters **from the end**
- **-1 = last character**



In [1]:

```
s1 = "MyFile.txt"

s2 = s1[:-1]           # s2 = 'MyFile.tx'
s3 = s1[:-2]           # s3 = 'MyFile.t'
s4 = s1[-3:]           # s4 = 'txt'
```



■ Strings: concatenation

- Use the + operator



In [1]:

```
string1 = 'Value of '  
sensor_id = 'sensor 1.'  
print(string1 + sensor_id)           # concatenation  
val = 0.75  
print('Value: ' + str(val))         # float to str
```

Out[1]:

```
Value of sensor 1.  
Value: 0.75
```



- **Strings are immutable**



In [1]:

```
str1 = "example"  
str1[0] = "E" # will cause an error
```

- **Use instead:**

In [1]:

```
str1 = "example"  
str1 = 'E' + str1[1:]
```



■ Formatted string literals (or f-strings)

- Introduced in Python 3.6
- Useful pattern to build a string from one or more variables
- E.g. suppose you want to build the string:

My float is 17.5 and my int is 5

var1 var2

- Syntax:
 - `f"My float is {var1} and my int is {var2}"`



■ Formatting strings (older versions)

■ Syntax:

■ "My float is %f and my int is %d" % (17.5, 5)

float placeholder

int placeholder

values to be replaced

My float is

17.5

and my int is

5

■ "My float is {0} and my int is {1}".format(17.5, 5)

index of variable that
will replace the braces



■ Example (\geq Python 3.6)

In [1]:

```
city = 'London'
temp = 19.23456
str1 = f"Temperature in {city} is {temp} degrees."
str2 = f"Temperature with 2 decimals: {temp:.2f}"
str3 = f"Temperature + 10: {temp+10}"
print(str1)
print(str2)
print(str3)
```

Out[1]:

```
Temperature in London is 19.23456 degrees.
Temperature with 2 decimals: 19.23
Temperature + 10: 29.23456
```



■ None type

- Specifies that a reference does not contain data

In [1]:

```
my_var = None

if my_var is None:
    my_var = 10
```

- Useful to:
 - Represent "missing data" in a list or a table
 - Initialize an empty variable that will be assigned later on
 - (e.g. when computing min/max)



■ Tuple

- **Immutable** sequence of *heterogeneous* variables
- Definition:



```
In [1]: t1 = ('Turin', 'Italy')      # City and State
        t2 = 'Paris', 'France'     # optional parentheses

        t3 = ('Rome', 2, 25.6)     # can contain different types
        t4 = ('London',)           # tuple with single element
```



■ Tuple unpacking

- Assigning a tuple to a set of variables



In [1]:

```
city_data = ('Turin', 'Italy', 12)
city, state, temperature = city_data

print(city)          # Turin
print(state)         # Italy
print(temperature)   # 12
```



- **Swapping** elements with tuples
 - This is an interesting case of unpacking



In [1]:

```
a = 1
b = 2
a, b = b, a
print(a)
print(b)
```

Out[1]:

```
2
1
```



■ Tuple

- Tuples can be **concatenated**
- A new tuple is generated upon concatenation

In [1]:

```
city = 'Turin', 'Italy'  
temperatures = 6, 15  
city_data = city + temperatures  
print(city_data)
```

Out[1]:

```
('Turin', 'Italy', 6, 15)
```



■ Tuple

- Accessing elements of a tuple
 - `t[start:stop]`
 - We can optionally specify a step `str[start:stop:step]` (*)

In [1]:

```
t1 = ('a', 'b', 'c', 'd')

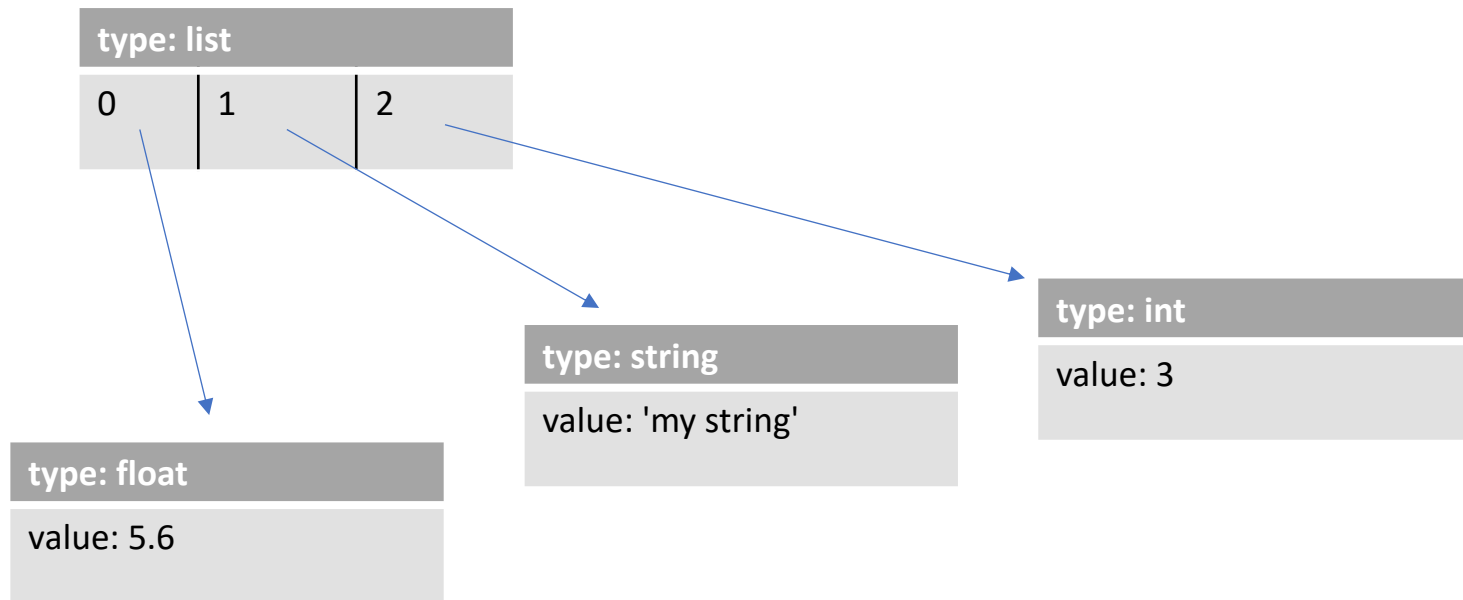
val1 = t1[0]           # val1 = 'a'
t2 = t1[1:]            # t2 = ('b', 'c', 'd')
t3 = t1[: -1]          # t3 = ('a', 'b', 'c')

t1[0] = 2              # will cause an error
                       # (a tuple is immutable)
```



List

- **Mutable** sequence of heterogeneous elements
- Each element is a **reference** to a Python object





■ List

■ Definition



```
In [1]:  l1 = []                                # empty list
         l2 = [1, 'str', 5.6, None]            # can contain different types

         a, b, c, d = l2                        # can be assigned to variables
                                                # a=1, b='str', c=5.6, d=None
```



List

■ Adding elements and **concatenating** lists



```
In [1]:  l1 = [2, 4, 6]
         l2 = [10, 12]
         l1.append(8)           # append an element to l1
         l3 = l1 + l2          # concatenate 2 lists
         print(l1)
         print(l3)
```

```
Out[1]:  [2, 4, 6, 8]
         [2, 4, 6, 8, 10, 12]
```



■ List

■ Other methods:

- `list.count(element)`
 - Number of occurrences of element
- `list1.extend(list2):`
 - Extend list1 with another list list2
- `list.insert(index, element):`
 - Insert element at position
- `list.pop(index):`
 - Remove element by position
- `list.index(element):`
 - Returns position of *first* occurrence of element



■ List

■ Accessing elements:

- Same syntax as tuples, but this time assignment is allowed

```
In [1]: l1 = [0, 2, 4, 6]
        val1 = l1[0]           # val1 = 0
        a, b = l1[1:-1]       # a=2, b=4
        l1[0] = 'a'
        print(l1)
```

```
Out[1]: ['a', 2, 4, 6]
```



List

Accessing elements

- Can also specify a **step**: [start:stop:step]
 - step = 2** skips 1 element
 - step = -1** reads the list in reverse order
 - step = -2** reverse order, skip 1 element

```
In [1]: l1 = [0, 1, 2, 3, 4]
        l2 = l1[::2]           # l2 = [0, 2, 4]
        l3 = l1[::-1]         # l3 = [4, 3, 2, 1, 0]
        l4 = l1[::-2]         # l4 = [4, 2, 0]
```



■ List

■ Assigning multiple elements

```
In [1]: l1 = [0, 1, 2, 3, 4]
        l1[1:4] = ['a', 'b', 'c']    # l1 = [0, 'a', 'b', 'c', 4]
```

■ Removing multiple elements

```
In [1]: l1 = [0, 1, 2, 3, 4]
        del l1[1:-1]                # l1 = [0, 4]
```



■ “in” operator

- **Check** if element belongs to a list



```
In [1]: l1 = [0, 1, 2, 3, 4]
        myval = 2
        myval in l1 # True, since 2 is in l1
```

- **Iterate** over list elements

```
In [1]: l1 = [0, 1, 2, 3, 4]
        for el in l1:
            print(el)
```



List

Sum, min, max of elements

```
In [1]: l1 = [0, 1, 2, 3, 4]
        min_val = min(l1)           # min_val = 0
        max_val = max(l1)           # max_val = 4
        sum_val = sum(l1)           # sum_val = 10
```

Sort list elements

- reverse=True for descending order

```
In [1]: l1 = [3, 2, 5, 7]
        l2 = sorted(l1)             # l2 = [2, 3, 5, 7]
        l3 = sorted(l1, reverse=True) # l3 = [7, 5, 3, 2]
```




■ Set

- **Unordered** collection of **unique** elements
- Definition:

```
In [1]: s0 = set()                # empty set
        s1 = {1, 2, 3}
        s2 = {3, 3, 'b', 'b'}    # s2 = {3, 'b'}
        s3 = set([3, 3, 1, 2])   # from list: s3 = {1,2,3}
```



■ Set

■ Operators between two sets



- | union (\cup)
- & intersection (\cap)
- - difference (\setminus)
- \leq subset (\subseteq)
- $<$ proper subset (\subset)
- \geq superset (\supseteq)
- $>$ proper superset (\supset)

```
s1 = {1, 2, 3}
s2 = {3, 'b'}
union = s1 | s2          # {1, 2, 3, 'b'}
intersection = s1 & s2   # {3}
difference = s1 - s2     # {1, 2}

{1,2} <= s1              # True
{1,2,3} < s1              # False (not a proper subset)
{1,2,3} <= s1            # True (same set)
```



■ Set

■ Add/remove elements



In [1]:

```
s1 = {1,2,3}
```

```
s1.add('4')
```

```
s1.remove(3)
```

```
# s1 = {1, 2, 3, '4'}
```

```
# s1 = {1, 2, '4'}
```

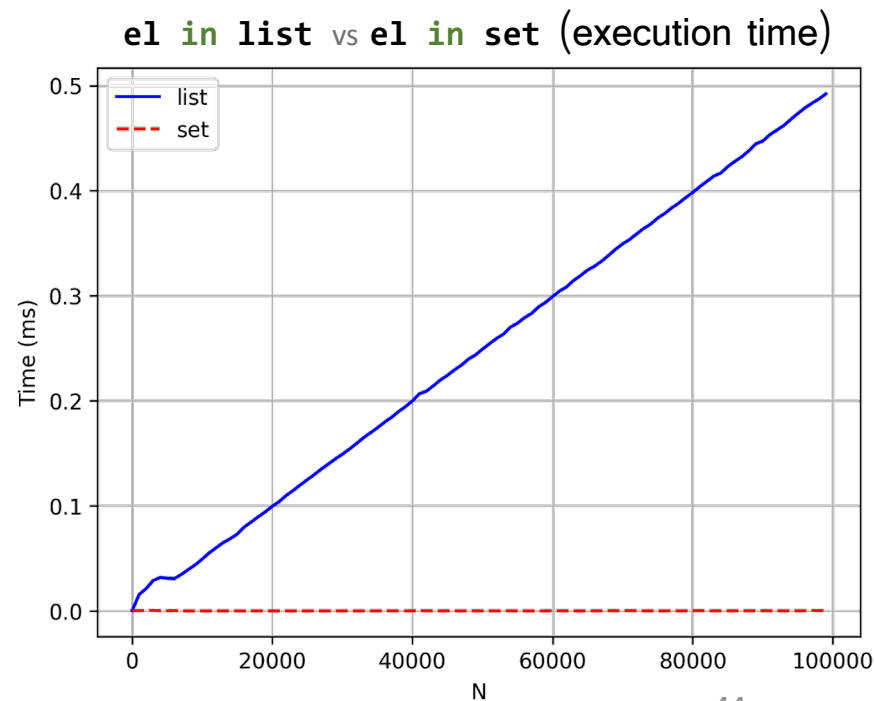


■ “in” operator

- **Check** whether element belongs to a set
- $O(1)$ operation
 - **!** Note that lists are $O(n)$

In [1]:

```
s1 = set([0, 1, 2, 3, 4])  
myval = 2  
myval in s1 # True, since 2 is in s1
```





■ “in” operator

■ Iterate over set elements

■ Note: sets are unordered

- The order during iterations is not well-defined

```
In [1]: s1 = set([0, 1, 2, 3, 4])
        for el in s1:
            print(el)
```

```
[In [1]: {1,2,3} == {3,2,1}
Out[1]: True

In [2]: for i in {1,2,3}:
        ...:     print(i)
        ...:
1
2
3

In [3]: for i in {3,2,1}:
        ...:     print(i)
        ...:
1
2
3
```



■ Set example: removing list duplicates

```
In [1]: input_list = [1, 5, 5, 4, 2, 8, 3, 3]
        out_list = list(set(input_list))

        print(out_list)
```

- **Note:** order of original elements is not preserved

```
Out [1]: [1, 2, 3, 4, 5, 8]
```



Notebook Examples

- **1-Python Examples.ipynb**
 - 1) Removing list duplicates





■ Dictionary

- Collection of key-value pairs
- Allows fast **access** of elements **by key**
 - Keys are **unique**

■ Definition:

```
In [1]: d1 = {'Name' : 'John', 'Age' : 25}
        d0 = {}                                # empty dictionary
```




■ Dictionary keys

- Must be **hashable** types
 - E.g. int, float, string, bool, **tuple**
 - Note: lists and dictionaries are not hashable
 - Hashable types are hashed with the `hash()` function
- Example: itemsets and their support

```
In [1]: d1 = {('a', 'b') : 120, ('c', 'd', 'e') : 1000}
```

- Note: the same applies for elements of sets!

■ Dictionary values

- Any Python object is allowed



■ Dictionary

■ Access by key:

```
In [1]: images = {10 : 'plane.png', 25 : 'flower.png'}  
img10 = images[10]           # img10 = 'plane.png'  
img8 = images[8]              # Get an error if key does not exist  
img8 = images.get(8)          # .get() returns None if the key does not exist  
img8 = images.get(8, 'notfound.png') # we can optionally specify a default value
```

■ Reading **keys** and **values**:

- Note: `keys()` and `values()` return **views on original data**

```
In [2]: occurrences = {'Car' : 33, 'Truck' : 55}  
keys = list(occurrences.keys())    # keys = ['Car', 'Truck']  
values = list(occurrences.values()) # values = [33, 55]
```



■ Dictionary

■ Adding/updating values:

```
In [1]: occur = {'Car' : 33, 'Truck' : 55}
        occur ['Car'] = 56           # Update existing value
        occur ['Road'] = 3          # Add a new key
```

■ Deleting a key:

```
In [2]: occur = {'Car' : 33, 'Truck' : 55}
        del d2['Truck']              # occur = {'Car':33}
```



■ Dictionary

- **Check** whether a key exists:



```
In [1]: occur = {'Car' : 33, 'Truck' : 55}
        'Truck' in occur # True since "Truck" is in occur
```



■ Dictionary

■ Iterating keys and values

- Note: Previous Python versions had no order guarantee
- However, Python 3.7+ officially preserves insertion order (*)

■ E.g. get the cumulative price of items in a market basket

In [1]:

```
basket = {'Cola' : 0.99, 'Apples' : 1.5, 'Salt' : 0.4}
price = 0
for k, v in basket.items():
    price += v
    print(f"{k}: {price}")
```

Out [1]:

```
Cola: 0.99
Apples: 2.49
Salt: 2.89
```



■ Default dictionary

- Access by key with **default value**:

```
In [1]: from collections import defaultdict

        experience = defaultdict(lambda: 1)
        experience['Mario']=3
        experience['Elena']+=1           # Even if key 'Elena' not defined
```

- Instead of writing:

```
In [2]: if 'Elena' in experience:
        experience['Elena']+=1
        else:
            experience['Elena']=2
```



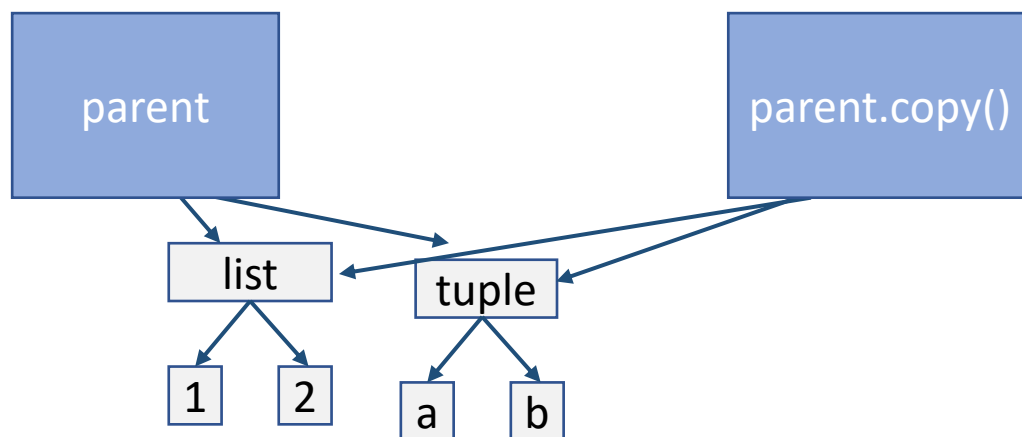
tuple vs list vs set vs dict

	tuple	list	set	dict
<i>Mutable</i>	No	Yes	Yes	Yes
<i>Ordered</i>	Yes	Yes	No*	No*
<i>Unique values</i>	No	No	Yes	Yes (keys)
<i>Constraints on values</i>	No	No	Must be hashable	Keys must be hashable
<i>Search cost</i>	$O(n)$	$O(n)$	$O(1)$	$O(1)$

* Implementation dependent – Since Python 3.7 dicts are ordered based on insertion order

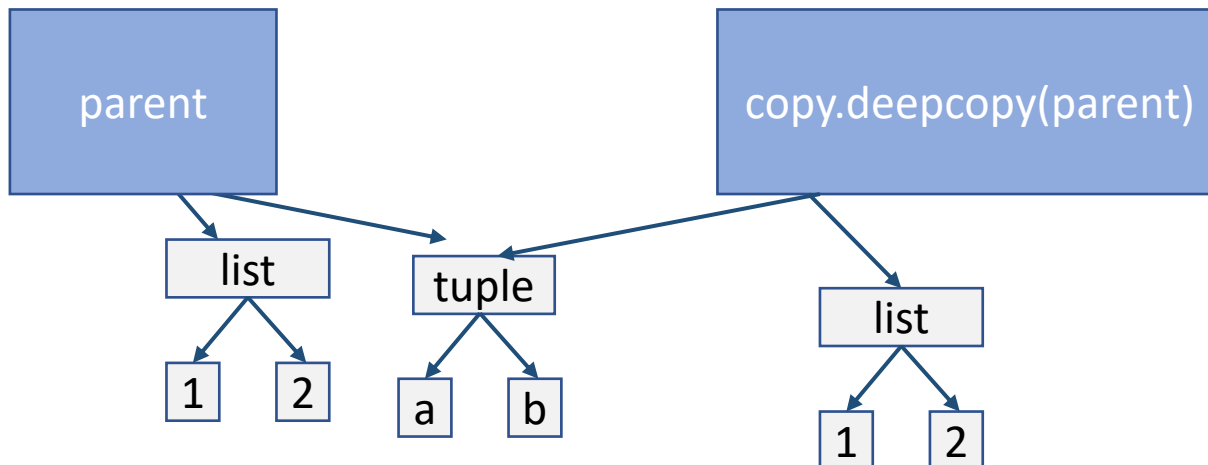


- Objects *can contain objects* within them
 - E.g., lists *of objects*
 - `parent = [[1, 2], ('a', 'b')]`
- We can create *shallow* or *deep* copies of objects
 - *Shallow*: copy the *parent* object, keep references to children





- We can create *shallow* or *deep* copies of objects
 - *Deep*: recursively copies all children nodes of parent object



Immutable objects
are not copied!



■ Shallow copies of Python objects

```
In [1]: temperatures = {'Turin':[10,12,10], 'Milan':[15,16,16]}
temp2 = temperatures.copy()
temp2['Turin'].append(13)           # Edit child node
temp2['Rome'] = [10, 11, 10]       # Edit parent node
print(temperatures)
print(temp2)
```

```
In [2]: {'Turin': [10, 12, 10, 13], 'Milan': [16, 15]}
{'Turin': [10, 12, 10, 13], 'Milan': [16, 15], 'Rome': [10, 11, 10]}
```



■ Deep copy of Python objects

```
In [1]: import copy
temperatures = {'Turin':[10,12,10], 'Milan':[15,16,16]}
temp2 = copy.deepcopy(temperatures)
temp2['Turin'].append(13)           # Edit child node
temp2['Rome'] = [10, 11, 10]       # Edit parent node
print(temperatures)
print(temp2)
```

```
In [2]: {'Turin': [10, 12, 10], 'Milan': [15,16,16]}
{'Turin': [10, 12, 10, 13], 'Milan': [15,16,16], 'Rome': [10, 11, 10]}
```



Python data types

```
import copy
a = [ [ 1, 2, 3 ], ('a', 'b', 'c') ]
ref = a
shallow_copy = a.copy()
deep_copy = copy.deepcopy(a)
id(a) == id(ref)           # True (references to the same object)
id(a) == id(shallow_copy) # False (shallow copy)
id(a[0]) == id(shallow_copy[0]) # True (shallow_copy points to a's children)
id(a[0]) == id(deep_copy[0])  # False (deep_copy copies a's children)
id(a[1]) == id(deep_copy[1])  # True (immutable objects are not copied)
```



■ if/elif/else

- Conditions expressed with `>`, `<`, `>=`, `<=`, `==`, `!=`
 - Can include boolean operators (and, not, or)

In [1]:

```
if sensor_on and temperature == 10:
    print("Temperature is 10")
elif sensor_on and 10 < temperature < 20:
    in_range = True
    print("Temperature is between 10 and 20")
else:
    print("Temperature is out of range or sensor is off.")
```

indentation is
mandatory



■ While loop

- Iterate while the specified condition is True

In [1]:

```
counter = 0
while counter < 5:
    print (f"The value of counter is {counter}")
    counter += 2    # increment counter of 2
```

Out [1]:

```
The value of counter is 0
The value of counter is 2
The value of counter is 4
```



- **Iterating** for a fixed number of times
 - Use: `range(start, stop)`

In [1]:

```
for i in range(5, 8):  
    txt = f"The value of i is {i}"  
    print(txt)
```

Out [1]:

```
The value of i is 5  
The value of i is 6  
The value of i is 7
```



■ Enumerating list objects

- Use: `enumerate(my_list)`

In [1]:

```
my_list = ['a', 'b', 'c']  
for i, element in enumerate(my_list):  
    print(f"The value of my_list[{i}] is {element}")
```

Out [1]:

```
The value of my_list[0] is a  
The value of my_list[1] is b  
The value of my_list[2] is c
```




■ Iterating on multiple lists

- Use: `zip(list1, list2, ...)`

In [1]:

```
my_list1 = ['a', 'b', 'c']  
my_list2 = ['A', 'B', 'C']  
for el1, el2 in zip(my_list1, my_list2):  
    print(f"E11: {el1}, el2: {el2}")
```

Out [1]:

```
E11: a, el2: A  
E11: b, el2: B  
E11: c, el2: C
```



■ Break/continue

- Alter the flow of a **for** or a **while** loop
- Example

my_file.txt

```
car  
skip  
truck  
end  
van
```

```
with open("./data/my_file.txt") as f:  
    for line in f:                # read file line by line  
        if line=='skip':  
            continue            # go to next iteration  
        elif line=='end':  
            break               # interrupt loop  
        print(line)
```

Out [1]:

```
car  
truck
```



- **Essential** to organize code and avoid repetitions

In [1]:

function name

return value

invocation

```
def euclidean_distance(x, y):  
    dist = 0  
    for x_el, y_el in zip(x, y):  
        dist += (x_el-y_el)**2  
    return dist ** 0.5  
print(f"{euclidean_distance([1,2,3], [2,4,5]):.2f}")  
print(f"{euclidean_distance([0,2,4], [0,1,6]):.2f}")
```

parameters

Out [1]:

```
3.00  
2.24
```



■ Variable scope

- Rules to specify the **visibility** of variables
- **Local scope**
 - Variables defined inside the function

In [1]:

```
v my_func(x, y):  
    z = 5  ← not accessible from outside  
    return x + y + z  
  
print(my_func(2, 4))  
print(z)  ← error: z undefined
```



■ Variable scope

■ Global scope

- Variables defined outside the function

In [1]:

```
def my_func(x, y):  
    return x + y + z  
  
z = 5  
my_func(2, 4)
```

← z can be read inside the function

Out [1]:

11



- **Variable scope**
 - **Global scope vs local scope**

In [1]:

```
def my_func(x, y):  
    z = 2      ← define z in local scope  
    return x + y + z ← use z from local scope  
  
z = 5      ← define z in global scope  
print (my_func(2, 4))  
print (z)    ← z in global scope is not modified
```

Out [1]:

```
8  
5
```



■ Variable scope

- Force the usage of variables in the global scope

In [1]:

```
def my_func(x, y):  
    global z          ← now z refers to global scope  
    z = 2             ← this assignment is performed to z  
    return x + y + z   in the global scope  
  
z = 5  
print (my_func(2, 4))  
print (z)
```

Out [1]:

```
8  
2
```



■ Variable scope

- Force the usage of variables in the global scope

In [1]:

```
def my_func(x, y):  
    global z          ← now z ref  
    z = 2             ← this assign  
    return x + y + z  in the glo  
  
z = 5  
print (my_func(2, 4))  
print (z)
```

Note

Avoid mixing global-local variables if possible. Pass all variables needed as arguments!

Out [1]:

```
8  
2
```




- Functions can **return tuples**

In [1]:

```
def add_sub(x, y):  
    return x+y, x-y  
  
summ, diff = add_sub(5, 3)  
print(f"Sum is {summ}, difference is {diff}.")
```

Out [1]:

```
Sum is 8, difference is 2.
```



■ Parameters with **default value**



In [1]:

```
def func(a, b, c='defC', d='defD'):
    print(f"{a}, {b}, {c}, {d}")

func(1, 2)                # use default for c, d
func(1, 2, 'a')           # use default for d, not for c
func(1, 2, d='b')         # passing keyword argument
func(b=2, a=1, d='b')     # keyword order does not matter
func(1, c='a')            # Error: b not specified
```

Out [1]:

```
1, 2, defC, defD
1, 2, a, defD
1, 2, defC, b
1, 2, defC, b
```



Map & Filter patterns

- Some patterns are commonly adopted
 - Filter pattern
 - Given a sequence of values, *keep some* and *discard the rest*
 - A function looks at each element and decides what to do
 - Function: `filter(filter_function, sequence)`
 - Map pattern
 - Given a sequence of values, map each element to a new one
 - A function applies the mapping
 - Function: `map(map_function, sequence)`



Filter pattern

- Task: Remove negative elements from a list of values
- Filter pattern
 - Given a **sequence** of values, *keep some* and *discard the rest*
 - A **function** looks at each element and decides what to do
 - return **True** if an element should be *kept*, **False** if it should be *discarded*
 - Function: **filter**(**filter_function**, **sequence**)

In [1]:

```
def is_positive(number):  
    return number >= 0  
  
numbers = [1, -8, 5, -2, 5]  
positive = list(filter(is_positive, numbers))  
# positive == [ 1, 5, 5 ]
```



Map pattern

- Task: Get squared values of the elements of a sequence
- Map pattern
 - Given a **sequence** of values, map each element to a new one
 - A **function** applies the mapping, element-wise
 - Function: `map(map_function, sequence)`

In [1]:

```
def square(number):  
    return number ** 2  
  
numbers = [1, -8, 5, -2, 5]  
squares = list(map(square, numbers))  
# squares == [ 1, 64, 25, 4, 25 ]
```



- The previous examples require creating a new function used only once
 - `is_positive()`, `square()`
- We can define lambda functions *inline* and *without a name*

In [1]:

```
numbers = [1, -8, 5, -2, 5]
positive = list(filter(lambda x: x >= 0, numbers))
squares = list(map(lambda x: x ** 2, numbers))
```

input parameter(s)

return value



- Lambda functions and conditions

- Possible with the *ternary operator*

- `[value_true] if [condition] else [value_false]`

- Examples of *conditional mappings*

In [1]:

```
numbers = [1, -1, 2, -2, 1]
sign = list(map(lambda x: '-' if x <= 0 else '+', numbers))
abs_values = list(map(lambda x: x if x > 0 else -x, numbers))
print(sign)
print(abs_values)
```

Out [1]:

```
['+', '-', '+', '-', '+']
[1, 1, 2, 2, 1]
```



■ Sort/min/max by key

```
In [1]: records = [{'name': 'v1', 'val': 5}, {'name': 'v2', 'val': 1},  
                  {'name': 'v3', 'val': 6}]  
min_val = min(records, key=lambda r: r['val'])  
sorted_records = sorted(records, key=lambda r: r['val'])  
  
print(f"Min: {min_val}")  
print(f"Sorted: {sorted_records}")
```

```
Out [1]: Min: {'name': 'v2', 'val': 1}  
Sorted: [{'name': 'v2', 'val': 1}, {'name': 'v1', 'val': 5},  
         {'name': 'v3', 'val': 6}]
```




List comprehensions

- Allow creating **lists** from other **iterables**
 - Useful for implementing the **map pattern**
 - Syntax:

In [1]:

```
res_list = [f(e1) for e1 in iterable]
```

iterate all the
elements

e.g. list or tuple

transform **e1** to
another value



List comprehensions

- Example: convert to uppercase dictionary keys
 - (**map** pattern)

In [1]:

```
dct = {'a':10, 'b':20, 'c':30}

my_list = [s.upper() for s in dct.keys()]
print(my_list)
```

Out [1]:

```
['A', 'B', 'C']
```



List comprehensions

- Allow specifying *conditions* on elements
 - Example: **square** the **positive** numbers in a list, discard the negative ones
 - **Filter** + **map** patterns

In [1]:

```
my_list1 = [-1, 4, -2, 6, 3]

my_list2 = [el**2 for el in my_list1 if el > 0]
print(my_list2)
```

Out [1]:

```
[16, 36, 9]
```



- Example: Euclidean distance

```
def euclidean_distance(x, y):  
    dist = 0  
    for x_el, y_el in zip(x, y):  
        dist += (x_el-y_el)**2  
    return dist ** 0.5
```



```
def euclidean_distance(x, y):  
    dist = sum([(x_el-y_el)**2 for x_el, y_el in zip(x, y)])  
    return dist ** 0.5
```



■ Dictionary comprehensions

- Similarly to lists, allow building dictionaries

```
In [1]: keys = ['a', 'b', 'c']  
        values = [-1, 4, -2]  
  
        my_dict = {k:v for k, v in zip(keys, values)}  
        print(my_dict)
```

```
Out [1]: {'a': -1, 'b': 4, 'c': -2}
```

■ Set comprehensions

```
In [2]: { v ** 2 for v in [ 4, 3, 2, -2, 1 ] }
```

```
Out [2]: {1, 4, 9, 16}
```



List comprehensions

- List comprehensions and lambda functions can shorten your code, but ...
 - Pay attention to **readability!!**
 - **Comments** are welcome!!



Notebook Examples

- **1-Python Examples.ipynb**
 - **2) Euclidean distance between lists**





- A class is a model that specifies a collection of
 - attributes (= variables)
 - methods (that interact with attributes)
 - a constructor (a special method called to initialize an object)
- An object is an **instance** of a specific class
- Example:
 - class: Triangle (all the triangles have 3 edges)
 - object: a specific instance of Triangle



- Simple class example:

In [1]:

```
class Triangle:  ← class name
    num_edges = 3 ← attribute definition

triangle1 = Triangle() ← class instantiation (object) creation
print(triangle1.num_edges) ← access to attribute
```

Out [1]:

3

- In this example all the object instances of Triangle have the same attribute value for num_edges: 3



■ Constructor and initialization:

In [1]:

self is a
reference to
the current
object

```
class Triangle:
```

```
    num_edges = 3
```

```
    def __init__(self, a, b, c):
```

```
        self.a = a
```

```
        self.b = b
```

```
        self.c = c
```

```
triangle1 = Triangle(2, 4, 3)
```

```
triangle2 = Triangle(2, 5, 2)
```

self is always the
first parameter

constructor
parameters

initialize attributes

invoke constructor
and instantiate a
new Triangle



- Methods
 - Equivalent to Python functions, but defined inside a class
 - The first argument is always **self** (reference to current object)
 - **self** allows accessing the object attributes
 - Example:

```
class MyClass:  
    def my_method(self, param1, param2):  
        ...  
        self.attr1 = param1  
        ...
```



■ Example with methods

In [1]:

```
class Triangle:
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c
    def get_perimeter(self): ← method
        return self.a + self.b + self.c

triangle1 = Triangle(2,4,3)
triangle1.get_perimeter() ← method invocation
                           (self is passed to the
                           method automatically)
```

use **self** for
referring to
attributes

Out [1]:

9



- **Private** attributes
 - Methods or attributes that are **available only inside the object**
 - They are **not accessible** from outside
 - Necessary when you need to define elements that are useful for the class object but must not be seen/modified from outside



■ Private attributes

In [1]:

```
class Triangle:
    def __init__(self, a, b, c):
        self.a, self.b, self.c = a, b, c
    def get_perimeter(self):
        return self.__perimeter

triangle1 = Triangle(2,4,3)
print(triangle1.get_perimeter())
print(triangle1.__perimeter)
```

← Error! Cannot access private attributes

2 leading
underscores
make variables
private

Out [1]:

9



Notebook Examples

- **1-Python Examples.ipynb**
 - **3) Classes and lambda functions: rule-based classifier**





Exception handling

- To track errors during program execution

In [1]:

```
try:
    res = my_dict['key1']
    res += 1
except:
    print("Exception during execution")
```

In [2]:

```
try:
    res = a/b
except ZeroDivisionError:
    print("Denominator cannot be 0.")
```

can specify
exception type →



- The **finally** block is executed in any case after try and except
 - It typically contains cleanup operations
 - Example: reading a file

In [1]:

```
try:
    f = open('./my_txt','r')      # open a file
    ...                          # work with file
except:
    print("Exception while reading file")
finally:
    f.close()
```



Exception handling

- The try/except/finally program in the previous slide can also be written as follows:

In [1]:

```
try:
    with open('./my_txt', 'r') as f:
        for line in f:
            ... # do something with line
except:
    print("Exception while reading file")
```

- If there is an **exception** while reading the file, the with statement ends
- In any case, when the with statement ends the file is automatically closed (similarly to the finally statement)



Notebook Examples

- **1-Python Examples.ipynb**
 - **4) Classes and exception handling: reading csv files**

