Machine Learning for Networking

Introduction to Python

Andrea Pasini
Flavio Giobergia
Elena Baralis
Gabriele Ciravegna

Politecnico di Torino

DataBase and Data Mining Group

1. **Python engine and language**

   - Setup, data types, object oriented programming

2. **Numpy library**

   - Computation with multi-dimensional arrays

3. **Pandas library**

   - Tabular data and data preprocessing

4. **Matplotlib library**

   - Data visualization and graphics

Python language

- Clean and concise syntax
  - No semi-colons to end instructions
  - No braces to define if clauses and for loops
  - No need to specify variable types
  - …

Java

```
List<String> l = new LinkedList<>();
for (int i=0; i<10; i++) {
    l.add(i);
}
```
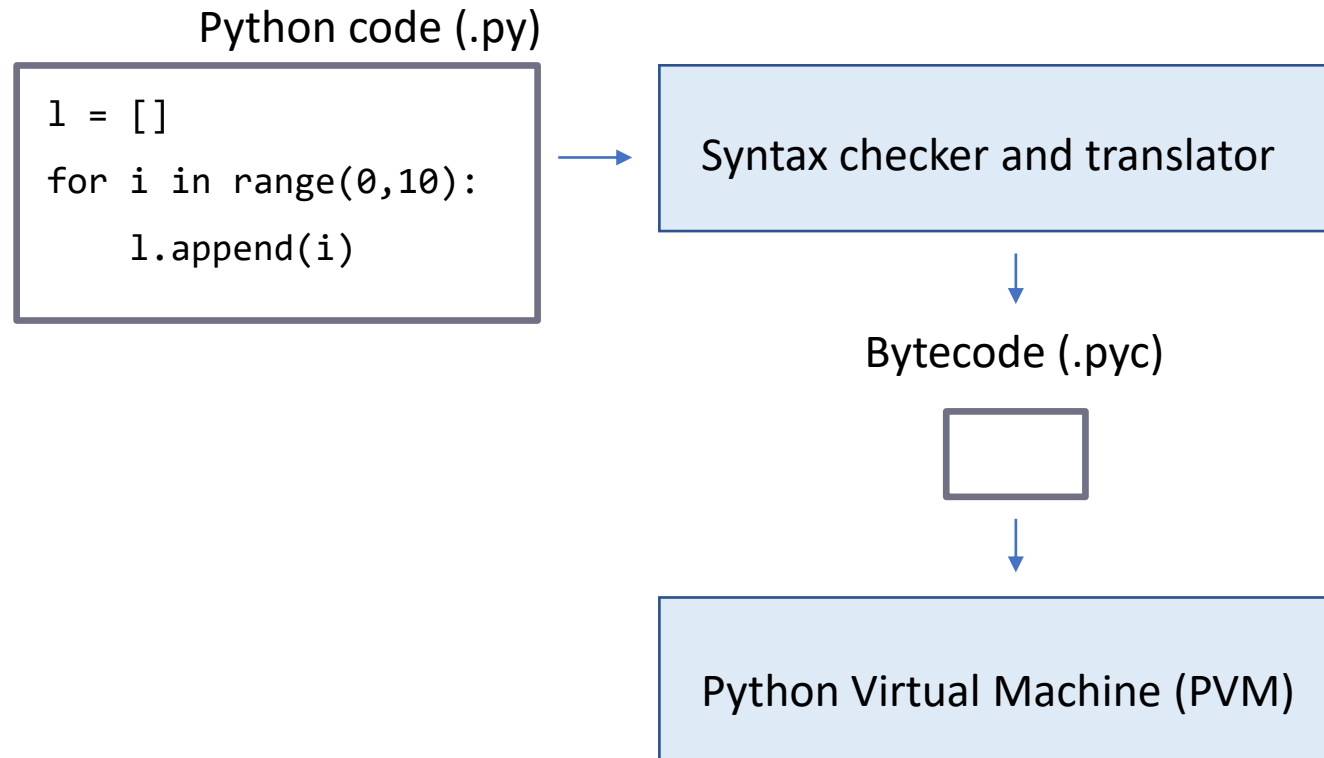
Python

```
l = []
for i in range(0,10):
    l.append(i)
```

- Python is an **interpreted** language

    - Code is not compiled to machine language

    - However the source code is compiled to an intermediate level, called **bytecode**

    - For this reason, to run Python programs, you need an **interpreter** that is able to execute the bytecode

- Sequence of operations executed by the interpreter

Python code (.py)

```
l = []
for i in range(0,10):
    l.append(i)
```

→ Syntax checker and translator

Bytecode (.pyc)

Python Virtual Machine (PVM)

- A common Python 3 setup on a **Linux** System

- Typically in the /usr/bin folder:
  - "**python3**" executable: run Python programs
  - "**pip3**" executable: install Python packages
  - "**ipython3**" executable: run programs line by line
  - "**jupyter**" executable: run a jupyter notebook

- To find where your python commands live:
  - $ which <command>

```
[fgiobergia@localhost $ which python3
/usr/local/bin/python3
fgiobergia@localhost $ ▋
```

- **Executing a Python program**

my_script.py

```
print("Hello")
```

~/Documents/MyScript
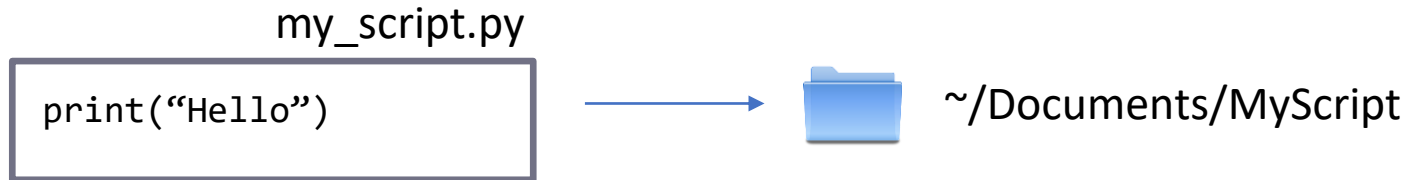
- **Type in your terminal:**
  - cd ~/Documents/MyScript
  - python3 my_script.py

- Running Python line by line with IPython

- Type in your terminal:
  - ipython3 (or ipython, depending on your installation)

Introduction to Python

- Write your program line by line to see the results step by step…

```
IPython: home/andrea

File  Modifica  Visualizza  Cerca  Terminale  Aiuto

andrea@andrea:~$ ipython3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: mystring = "hello"

In [2]: print(mystring)
hello

In [3]:
```

Introduction to Python

- **Python** and **IPython** programs are the core for executing scripts, but…

- There are two typical scenarios:

  1. Develop your Python **project** with an **IDE**

     - Example: Visual Studio Code, PyCharm
     - **Debug** and **run** your code inside the IDE

  2. Develop and test a Python **script** with **Jupyter notebook**

     - Inspect **step by step** the results
     - Keep the history of the output of the script

Introduction to Python

- **Scenario 1: PyCharm (IDE)**
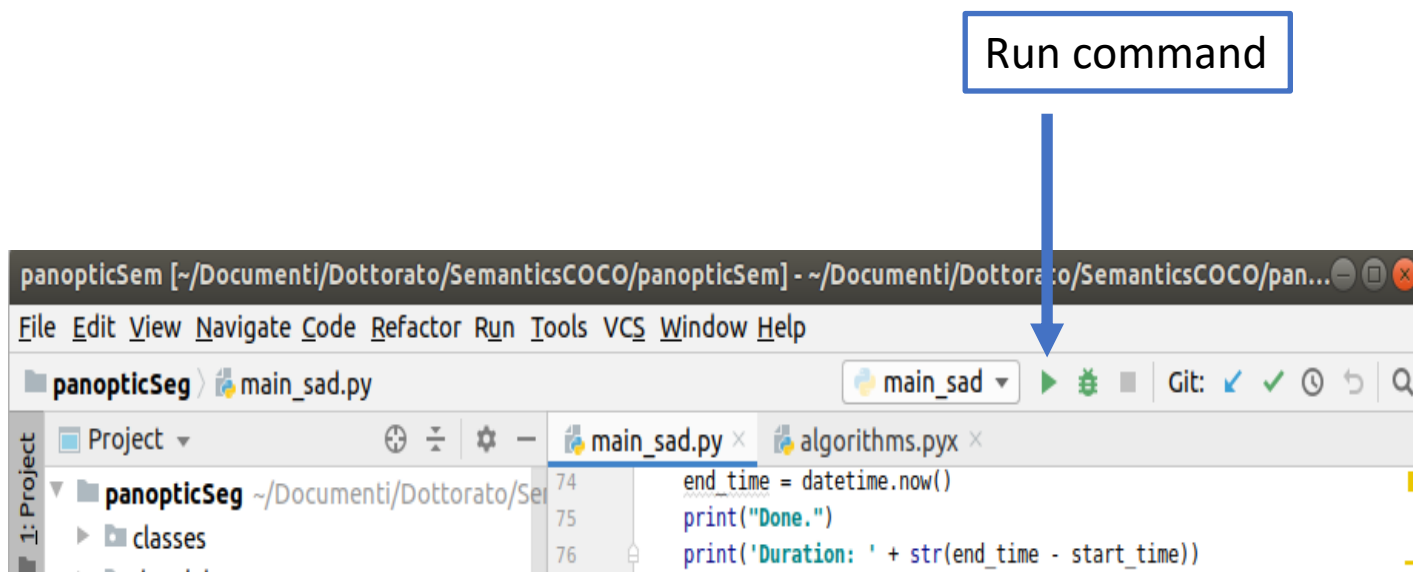

Run/Debug commands
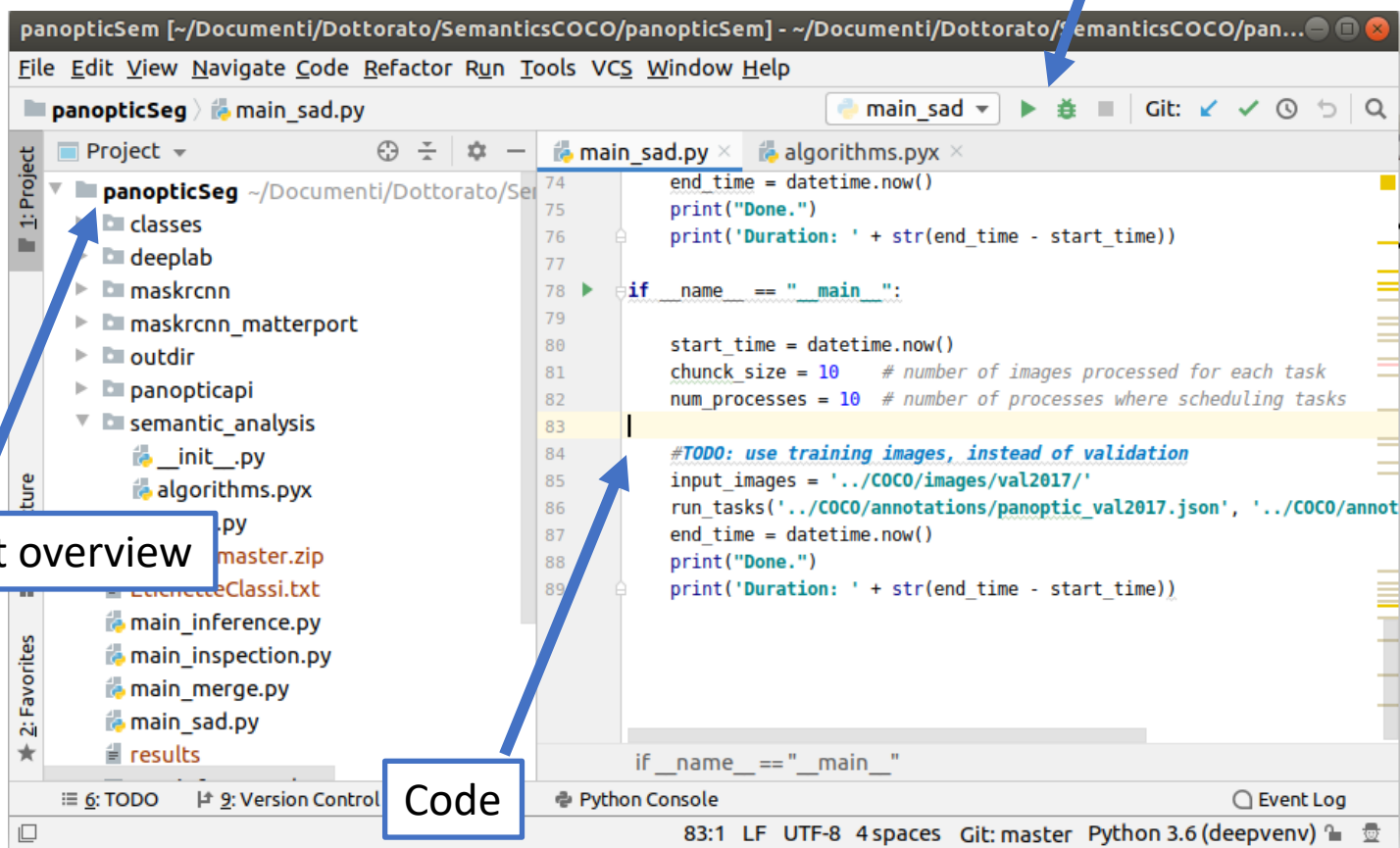
Project overview

Code

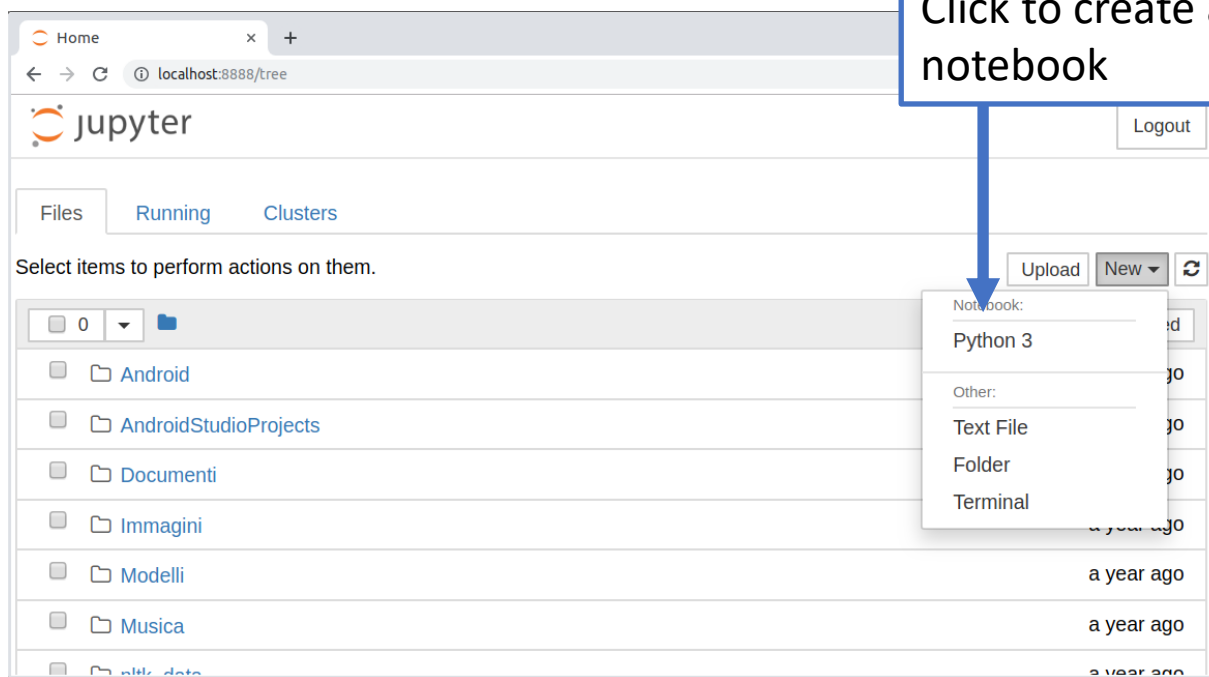- ## **Scenario 1: PyCharm (IDE)**

  - When you click on the run button, the IDE will automatically call the "**python**" command to execute your script

Run command

Scenario 2: Jupyter notebook

- Type in your terminal
  - jupyter notebook
- Jupyter will open on your browser

Click to create a new notebook

13

Scenario 2: Jupyter notebook

Markdown cell

Code cell

Result cell

Scenario 2: Jupyter notebook

- Based on **IPython** command

- Each code **cell** can be executed **separately** by pressing CTRL + ENTER

1. Simple linear regression

Generating a dataset

```
In [26]:  # Make dataset
          err = np.random.normal(0,1, 100)    # gaussian data, mean=0, std=1
          x = 10*np.random.rand(100)          # 100 data points in [0, 10]
          y = (2*x + 2) + err                 # target is a linear function of the i
```

Code cell 1

Code cell 2

```
In [27]:  # Plots
          plt.scatter(x, y, s=10, c='grey')
          plt.show()
```

IDE vs Jupyter notebook

- **IDE**

  - For more **complex** projects (many files)

  - More powerful debug commands

  - More powerful code editing tools

- **Jupyter notebook**

  - For simple scripts and prototypes

  - Great **visualization** tool

    - Example: **report** with Python code and text for explanations

Installing libraries

- Python language is provided with many useful libraries:
  - Numpy, Pandas, Matplotlib, Scikit-learn, SciPy, …
- To use any of them you first have to install it with the **pip** command: pip3 install <package>
  - pip3 install numpy
  - pip3 install pandas

Virtual environments

- The pip command will associate the libraries to your **default Python installation**

- A more powerful way of managing libraries is to use a Python **environment (virtualenv)**

  - Useful when you have **many projects** that use different libraries and **configurations (e.g. versions)**

    - Each projects is associated to a virtual environment

Virtual environments

- To **create** a new environment:

    - cd ~/Documents/My_project

    - virtualenv myenv

- It will create a new environment in your project folder

Virtual environments

- ## To **activate** the created environment:

  - cd ~/Documents/My_project

  - source myenv/bin/activate

  ```
  File  Modifica  Visualizza  Cerca  Terminale  Aiuto
  andrea@andrea:~/Documenti/MyProject$
  (myenv) andrea@andrea:~/Documenti/MyProject$
  ```

- ## To **deactivate** simply write:

  - deactivate

- ## To **remove** an environment:

  - Simply remove the folder (e.g. rm -r myenv)

Virtual environments

- After activation you can use the terminal to work within the environment

- Install libraries in the *current* environment

  - pip3 install my_library

- Execute a script/notebook within the environment

  - python3 my_script.py

  - jupyter notebook

In class Exercises

- **0) Let's setup together an environment**

Python programming

Andrea Pasini
Flavio Giobergia
Elena Baralis
Gabriele Ciravegna

DataBase and Data Mining Group

Summary

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

Python data types

- 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

Python data types

- **Reference** = **symbol** in a program that refers to a particular **object**

- A single Python object can have **multiple references (alias)**

references

integer object

x	→	**type: int**
		id: 140735957856544

| y | ↗ | value: 3 |
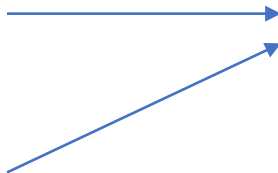
Python data types

- In Python

  - **Variable** = **reference** to an object

- When you **assign** an object to a variable it becomes a **reference** to that object

variables (references)

integer object

x	

type: int
id: 140735957856544

value: 3

y	

Python data types
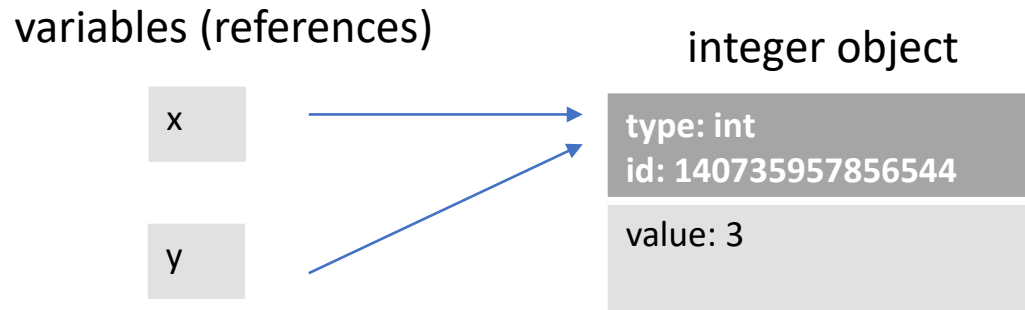
- ## **Defining a variable**

  - **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

Python data types

- Example

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

x →

type: float

value: 5.6

y →
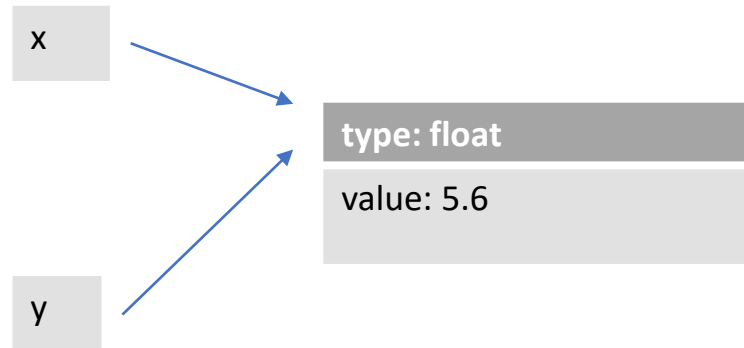
- If you assign y to a new value…

```
y = 3
```

x →

type: float

value: 5.6

y →

type: int

value: 3

Python data types
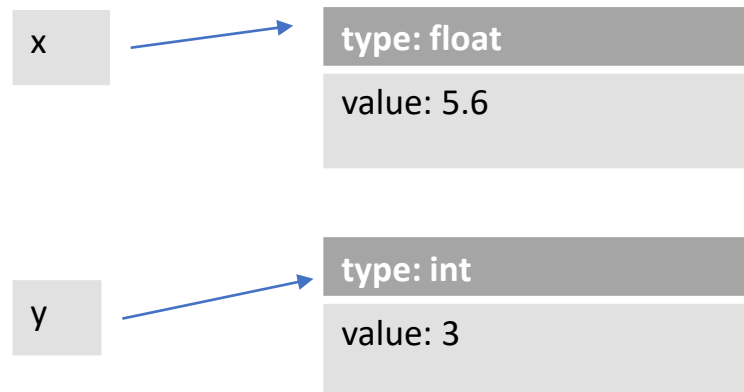
- From the previous example we learn that:

  - 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

```
y = 5.6
y = 3
```

type: float
value: 5.6

type: int
value: 3

y

Python data types
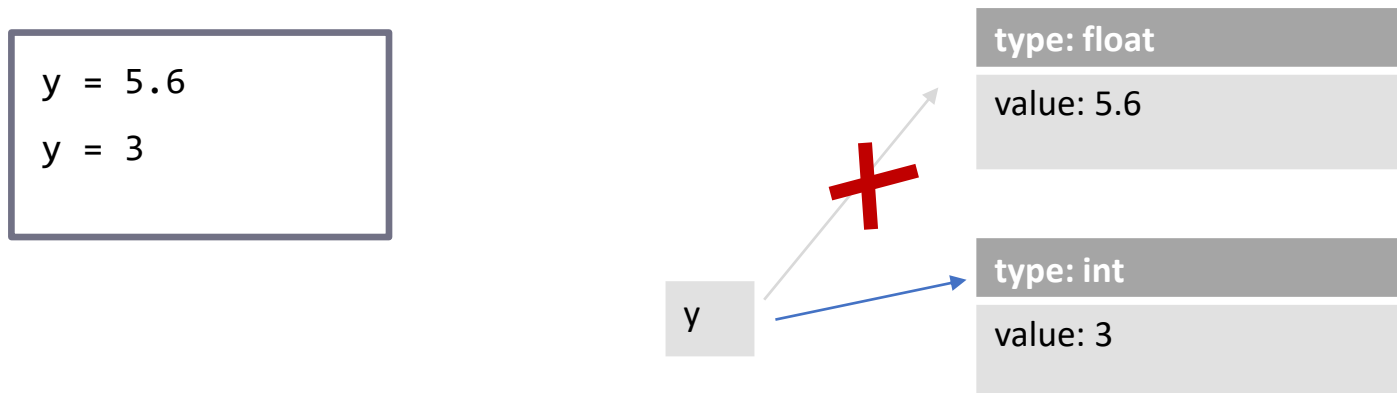
- Verify this reasoning with id()

    - **id(my_variable)** returns the **identifier** of the object that the variable is referencing

my_variable	→	**type: int** **id: 140735957856544**
		value: 3

Python data types

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** list 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

```
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(0)             # r6 = False
```

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

Python data types

- **Working with strings**

  - **len:** get string length

  - **strip:** remove leading and trailing spaces (tabs or newlines)

  - **upper/lower**: convert uppercase/lowercase

```
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

str[start:stop]

- The start index is **included**, while stop index is **excluded**
- Index of characters starts **from 0**
- We can optionally specify a step str[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[:]                   # s4 = 'Hello'
```

(∗) see details in the upcoming "list" explanation

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]:
```python
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
```

Python data types

- **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:

    var1   var2
    | My float is | 17.5 | and my int is | 5 |

- 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

44

Python data types

- **Example (>= 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
```

- **Strings**
  - **Other methods:**
    - str1.capitalize()
      - Converts the first character to upper case(element)
    - str1.find("a") or str1.index("a")
      - Return the index of the given string
    - str1.count("a"):
      - Count the number of times a given string is found
    - str1.split("a"):
      - Returns a list of sub-strings separated by the given string
    - str1.replace("Mike", "Bob")
      - Returns a new string where all "Mike" substrings are replaced by "Bob"

In class Exercises

- **1.1 Python Examples.ipynb**

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 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
```

Python data types

- **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]:
```python
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)
```

53

(∗) see details in the upcoming "list" explanation

Python data types

- **List**

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

type: list		
0	1	2

type: float
value: 5.6

type: string
value: 'my string'

type: int
value: 3

Python data types
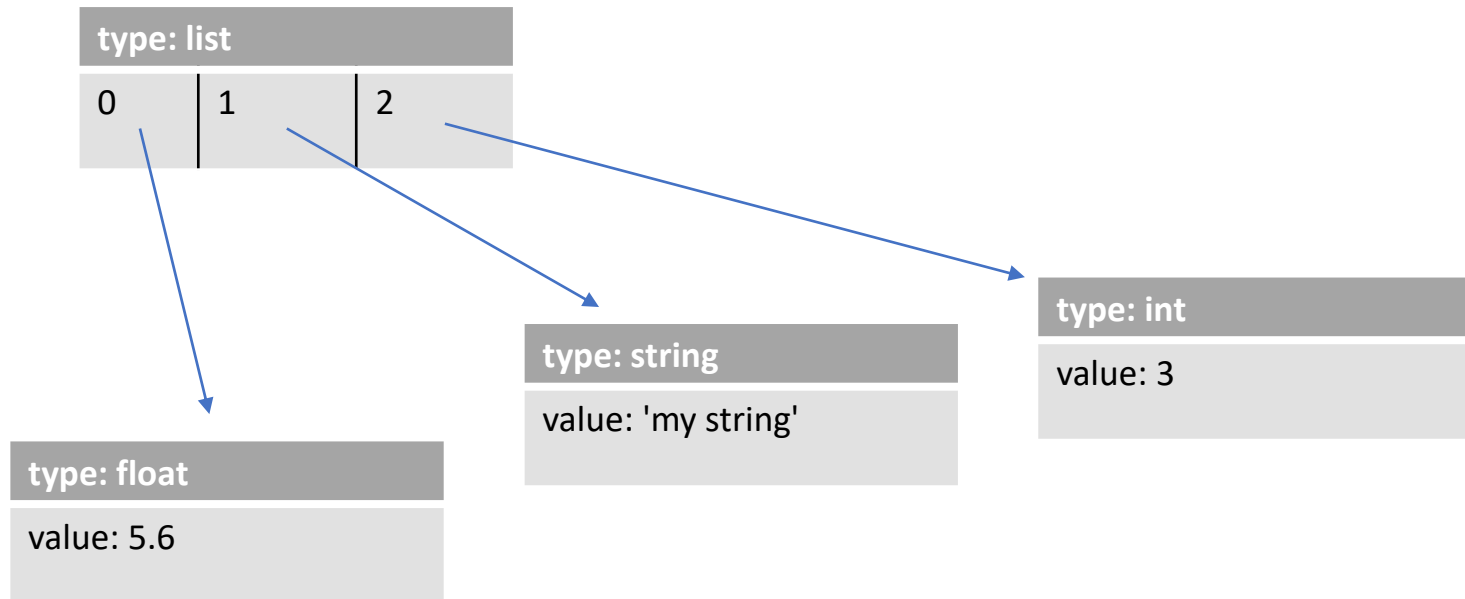
- ## **List**
  - ### Definition

```
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
```

Python data types

- ## **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]
```

56

Python data types

- **List**
  - **Other methods:**
    - list.count(element):
      - Number of occurrences of element
    - list1.extend(list2):
      - Extend list1 with another list list2
    - list1.insert(index, element):
      - Insert element at position
    - list1.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 assigment 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]
```

Python data types

- **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]                   # l3 = [4, 2, 0]
```

Python data types

- **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)
```

Python data types

- ## **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

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

Python data types

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), & (intersection), – (difference)
- <, <= ((proper) subset), >, >= ((proper) superset)

```
In [1]:   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 = {1, 2, 3, '4'}
          s1.remove(3)               # s1 = {1, 2, '4'}
```

"in" operator

Check whether element belongs to a set

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

Iterate over set elements

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

Python data types

- **"in" operator**

  - **Check** whether element belongs to a set

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

  - **Iterate** over set elements

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

Note
Sets are <u>unordered</u> – the order during iterations is not well-defined

```
[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]:
```python
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]
```

In Class Exercises

- **1.2 Python Tuples and Lists.ipynb**

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
```

Python data types

- **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.ong') # 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]
```

Python data types

- **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
```

Python data types

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
```

(∗) https://docs.python.org/3/whatsnew/3.7.html

75

- **Default dictionary**

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

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

tuple vs list vs set vs dict

	tuple	list	set	dict
Mutable	No	Yes	Yes	Yes
Ordered	Yes	Yes	No*	No*
Unique values	No	No	Yes	Yes (keys)
Limitations on values	No	No	Must be hashable	Keys must be hashable
Search cost	O(n)	O(n)	O(1)	O(1)

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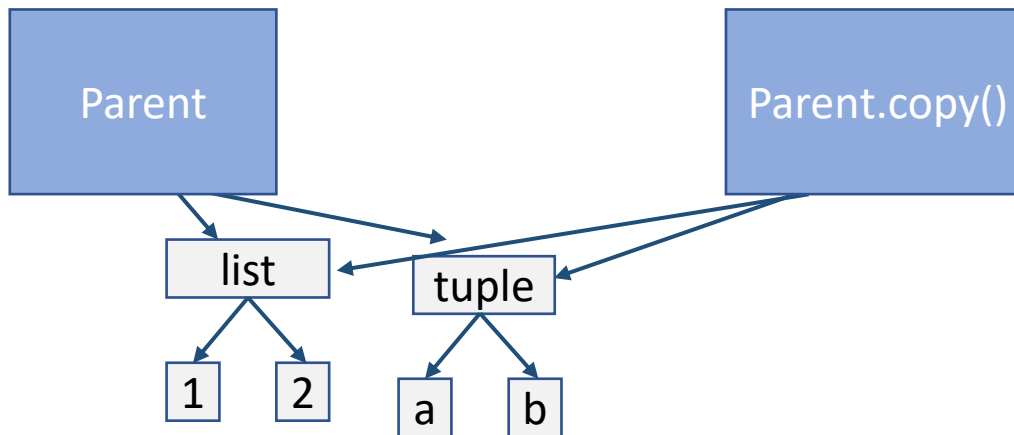
In Class Exercises

- **1.3 Python Dictionaries and Sets.ipynb**

Shallow vs deep copy

- Shallow: copies the parent object, shares references to children

Shallow vs deep copy

- Deep: recursively copies all children nodes of parent object

Parent

copy.deepcopy(Parent)

list

tuple

list

1 2

a b

1 2

Immutable objects are not copied!
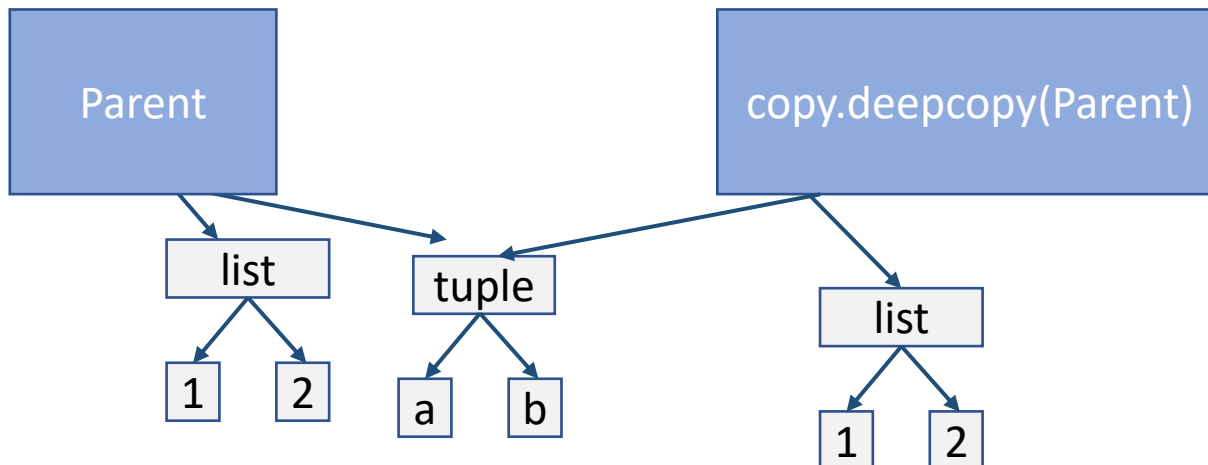
- **Shallow copies of Python objects**

In [1]:
```python
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]:
```python
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]}
```

- **if/elif/else**

  - Conditions expressed with >, <, >=, <=, ==, !=

    - Can include boolean operators (and, not, or)

In [1]:

```python
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

83

▪ **While loop**

- ▪ Iterate while the specified condition is True

In [1]:
```python
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]:
```python
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]:
```python
my_list1 = ['a', 'b', 'c']
my_list2 = ['A', 'B', 'C']
for el1, el2 in zip(my_list1, my_list2):
    print(f"El1: {el1}, el2: {el2}")
```

Out [1]:
```
El1: a, el2: A
El1: b, el2: B
El1: c, el2: C
```

Break/continue

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

my_file.txt

```
car
skip
truck
end
van
```

```python
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
```

88

Functions

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

parameters

In [1]:

function name

```python
def euclidean_distance(x, y):
    dist = 0
    for x_el, y_el in zip(x, y):
        dist += (x_el-y_el)**2
    return math.sqrt(dist) # alternatively, 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}")
```

return value

invocation

Out [1]:

```
3.00
2.24
```

Functions

- ## **Variable scope**

  - ### Rules to specify the **visibility** of variables

  - ### **Local scope**

    - Variables defined inside the function

In [1]:
```python
def 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 can be read inside the
                                          function

          z = 5

          my_func(2, 4)


Out [1]:  11
```

■ **Variable scope**

■ **Global scope vs local scope**

In [1]:
```python
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]:
```python
def my_func(x, y):
    global z          ←——————  now z refers to global scope
    z = 2             ←——————  this assigment is performed to z
                               in the global scope
    return x + y + z
z = 5
print (my_func(2, 4))
print (z)
```

Out [1]:
```
8
2
```

Functions

Variable scope

- Force the usage of variables in the global scope

In [1]:
```python
def my_func(x, y):
    global z          ←——————  now z ref
    z = 2             ←——————  this assig
    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]:
```python
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.
```

Functions

- **Parameters with default value**

In [1]:
```python
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
```

96

Lambda functions

- Functions that can be defined **inline** and **without a name**

- Example of lambda function definition:

input parameter(s)   return value

```
In [1]:    squared = lambda x: x**2
           print(squared(5))
```

```
Out [1]:   25
```

These patterns are useful shortcuts...

- Example: **filter** negative numbers from a list:

In [1]:
```
numbers = [1, -8, 5, -2, 5]
negative = []
for x in numbers:
    if x < 0:
        negative.append(x)
```

- This code can be completely rewritten with lambda functions…

Filter and map patterns

- Both apply a function element-wise to the elements of a list (iterable)
- **Filter** the elements of a list based on a condition
- **Map** each element of a list with a new value

```
In [1]:    numbers = [1, -8, 5, -2, 5]
           negative = list(filter(lambda x: x<0, numbers))
           squared = list(map(lambda x: x**2, negative))
           print(negative)
           print(squared)
```

```
Out [1]:   [-8, -2]
           [64, 4]
```

Lambda functions and conditions

- Example **conditional mapping**:

In [1]:

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

Out [1]:

```
['+', '+', '+', '-', '+']
```

■ **Sort/min/max by key**

In [1]:
```python
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:

iterate all the elements   e.g. list or tuple

In [1]:

```python
res_list = [f(el)    for el in iterable]
```

transform **el** to another value

List comprehensions

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

In [1]:
```python
dct = {'a':10, 'b':20, 'c':30}

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

Out [1]:
```python
['A', 'B', 'C']
```

List comprehensions

- **Allow specifying conditions on elements**
  - Example: **square positive** numbers in a list
    - **Filter** + **map** patterns

In [1]:
```python
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

```python
def euclidean_distance(x, y):

    dist = 0

    for x_el, y_el in zip(x, y):

        dist += (x_el-y_el)**2

    return math.sqrt(dist)
```

↓

```python
def euclidean_distance(x, y):

    dist = sum([(x_el-y_el)**2 for x_el, y_el in zip(x, y)])

    return math.sqrt(dist)
```

Other comprehensions

- ## Dictionary comprehensions

  - ### Similarly to lists, allow building dictionaries

In [1]:
```python
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

```python
[In [1]: { v ** 2 for v in [ 4, 3, 2, -2, 1 ] }
Out[1]: {1, 4, 9, 16}
```

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

- 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

Classes

- Simple class example:

In [1]:

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


triangle1 = Triangle()   ← class instantiation
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

Classes

- Constructor and initialization:

In [1]:
```python
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

self is a reference to the current object

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:

```python
class MyClass:

    def my_method(self, param1, param2):

        ...

        self.attr1 = param1

        ...
```

- **Example with methods**

In [1]:

```python
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]:

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


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

2 leading underscores make variables private → `self.__perimeter = a + b + c`

Error! Cannot access private attributes ← `print(triangle1.__perimeter)`

Out [1]:

9

In Class Exercises

- **1.4 Python Lambda functions, Functions and Classes.ipynb**

■ Use `open(filename, 'w'/'r')` to **read/write** a file

```
f = open("my_file.txt", "r")

…

f.close()
```

Watch out the strategy!
"r" - reading
"w" - writing
"a" - append

■ We always need to remember `f.close()`. Unless…

■ Use `with` statement!

```
with open("my_file.txt", "r") as f:

    …
```

■ It avoids remembering to close the file

■ Even in case of errors it guarantees a correct closure

116

File Handling

- ## How to **read**?

```
with open("my_file.txt", "r") as f:
    lines = f.readlines()
    for line in f:
        print(line)
```

readlines() read all the lines in once

Same as reading one line at a time

```
line = True
while(line):
    line = f.readline()
```

- ## What if the file is too big?

  - ### Chunk it!

```
with open("my_file.txt", "r") as f:
    for chunk in f.read(5):
        print(chunk)
```

read(n) reads n characters at a time (default n=1)

- **How to write?**

```
with open("my_file.txt", "w") as f:

    f.write("Ops I deleted everything!")
```

When you open a file with "w" you cancel all the existing content

- **Example: make a copy of a file**

```
with open("my_file.txt", "r") as f1:

    with open("my_file_copy.txt", "w") as f2:

        for line in f1:

            f2.write(line)
```

Exception handling

- To track errors during program execution

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

In [2]:
```python
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 tipically 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]:
```python
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)

In class exercises

- **1.5 Python Files and Exceptions.ipynb**