# Data Science

Prof. Silvio Peroni

Jan 21, 2024

# CONTENTS

This space contains all the material related to the Data Science course of the Digital Humanities and Digital Knowledge degree at the University of Bologna.

# CHAPTER ONE

# ACADEMIC YEAR 2023/2024

## 1.1 Material

**Keys:**

- *the* = theoretical lecture
- *hon* = hands-on session

1. [29/01/24, *the*] Introduction to the course and final project specifications
   - slides: Google Slides
2. [02/02/24, *the*] What is a datum and how it can be represented computationally
   - slides: Google Slides
3. [05/02/24, *hon*] Data formats and methods for storing data in Python
   - material: GitHub
4. [07/02/24, *the*] Introduction to data modelling
   - slides: Google Slides
5. [09/02/24, *hon*] Implementation of data models via Python classes
   - material: GitHub
6. [12/02/24, *the*] Processing and querying the data
   - slides: Google Slides
7. [14/02/24, *hon*] Introduction to Pandas
   - material: GitHub
8. [16/02/24, *the*] Database Management Systems
   - slides: Google Slides
9. [19/02/24, *hon*] Configuring and populating a relational database
   - material: GitHub
10. [21/03/24, *the*] SQL, a query language for relational databases
    - slides: Google Slides
11. [23/03/24, *hon*] Configuring and populating a graph database
    - material: GitHub
12. [26/02/24, *the*] SPARQL, a query language for RDF databases

- slides: Google Slides

13. [28/02/24, *hon*] Interacting with databases using Pandas
    - material: GitHub
14. [01/03/24, *the*] Describing and visualising data
    - slides: Google Slides
15. [04/03/24, *hon*] Descriptive statistics and graphs about data using Pandas
    - material: GitHub

## 1.2 Schedule

# CHAPTER TWO

# LINKS

- Some complementary books on the topic:
  - A critical field guide for working with machine learning datasets

## 2.1 Data formats and methods for storing data in Python

In this tutorial, we see two basic formats to deal with data in Python: CSV and JSON.

### 2.1.1 Comma-separated values (CSV)

The first and simplest format you can use to store and load data in Python is the Comma-Separated Values (CSV). In practice, each CSV file defines a table of a fixed number of columns where each row represents a (subject) entity and each cell defines the (object) value associated to that entity via the predicate defined by the column label, if specified. While it is not mandatory to specify column labels, it makes a CSV file more understandable to humans and machines. These labels can be specified using the first row of a CSV, defining an header of the table represented. An example of a table represented with a CSV is shown as follows.

| | column1 | column2 | … | columnn |
|---|---|---|---|---|
| *entity1* | value1 | value1 | … | valuen |
| *entity2* | value1 | value1 | … | valuen |
| … | … | … | … | … |
| *entitym* | value1 | value1 | … | valuen |

Each cell in a row is defined by splitting the cell values using a comma (`,`). In case the comma is part of the value of a cell, it is possible to escape such a comma by putting the cell value between quotes (`"`). Finally, in case a cell value is defined using quotes and one or more quote is included in cell value, these must be escaped by using double quotes (`""`). The following table and the related CSV source show how to define in CSV cell values when these situations happen.

| column name | another name, with a comma |
|---|---|
| a value | a value, with a comma |
| a quoted “value” | a quoted “value”, with a comma |

CSV source:

```
column name,"another name, with a comma"
a value,"a value, with a comma"
a quoted "value","a quoted ""value"", with a comma"
```

Python has a dedicated library to handle this format called *csv*. In order to understand how to use it, we can start analysing two very simple CSV files, one containing publications and some of their basic metadata and another with information about the venues where such publications have been published. To open it as a source file, you can right-click on it in the left panel and select *Open With -> Editor*.

## Opening a CSV file

In order to understand how these files are represented in Python, let us try to load one of it into a Python object using the funcion *reader* included in the module *csv* mentioned above. For doing that, it is necessary to obtain a file object of a particular file using the built-in function `open` used with a `with` statement as shown in the following excerpt:

```
from csv import reader

with open("assets/data/01-publications.csv", "r", encoding="utf-8") as f:
    publications = reader(f)
```

The function `open` takes in input several parameters and returns a file object, i.e. a Python object used to interact with a file stored in the file system. However, it is highly suggested to use at least the three specified above, that are:

1. the first positional parameter must contain the path of the file one wants to open;
2. the second positional parameter is the mode used to open the file (`"r"` stands for read, `"w"` for write, etc.);
3. the named parameter `encoding` specifying the encoding to open the file (`"utf-8"` must be used, if you do not want to have issues).

In addition, all file objects one wants to create in Python to process files stored in the file system must be also closed once all the operations on that file are concluded. The keyword `with` used with the function `open` allows one to handle the opening and closing of the file object automatically. In practice, once all the operation within the `with` block are executed, the related file object will be closed. Finally, the file object openned using the function `open` will be assigned to the variable that follows the keyword `as`, i.e. `f` in the example above. It is worth mentioning that the example just shown introduces how to open, in reading mode, any file in the file system, not only CSV files.

## CSV reader

Once our file object has been defined, we can read its content interpreting it as a CSV document using the constructor `reader` included in the package `csv`, which is imported by means of the usual command:

```
from csv import reader
```

The constructor `reader` takes in input a file object and returns an object of type (i.e. class) `_csv.reader`, that enables one to iterate over the CSV document row by row. To check the actual type, you can use the built-in function `type` passing the object as input, and then printing it on screen using either the function `print` or, when we want to print a complex structure, the function `pprint` from the package `pprint` (a.k.a. *pretty print*), as shown running the following code:

```
from pprint import pprint

pprint(type(publications))
```

```
<class '_csv.reader'>
```

An object of the class `_csv.reader` behaves like a list, and can be iterated using a for earch loop, as shown as follows:

```
with open("assets/data/01-publications.csv", "r", encoding="utf-8") as f:
    publications = reader(f)

    for row in publications:
        pprint(row)
```

```
['doi',
 'title',
 'publication year',
 'publication venue',
 'type',
 'issue',
 'volume']
['10.1002/cfg.304',
 'Development of Computational Tools for the Inference of Protein Interaction '
 'Specificity Rules and Functional Annotation Using Structural Information',
 '2003',
 '1531-6912',
 'journal article',
 '4',
 '4']
['10.1016/s1367-5931(02)00332-0',
 'In vitro selection as a powerful tool for the applied evolution of proteins '
 'and peptides',
 '2002',
 '1367-5931',
 'journal article',
 '3',
 '6']
['10.1002/9780470291092.ch20',
 'Mechanisms of Toughening in Ceramic Matrix Composites',
 '1981',
 '9780470291092',
 'book chapter',
 '',
 '']
```

In case you want to skip the header of the table if present, starting to look at the values directly, you need to use the built-in function `next` that takes in input any iterator-based object, such as our CSV reader, and skips to the next line:

```
with open("assets/data/01-publications.csv", "r", encoding="utf-8") as f:
    publications = reader(f)
    next(publications)  # it skip the first row of the CSV table

    for row in publications:
        pprint(row)  # it prints all the rows except the header
```

```
['10.1002/cfg.304',
 'Development of Computational Tools for the Inference of Protein Interaction '
```

(continues on next page)

(continued from previous page)

```
 'Specificity Rules and Functional Annotation Using Structural Information',
 '2003',
 '1531-6912',
 'journal article',
 '4',
 '4']
['10.1016/s1367-5931(02)00332-0',
 'In vitro selection as a powerful tool for the applied evolution of proteins '
 'and peptides',
 '2002',
 '1367-5931',
 'journal article',
 '3',
 '6']
['10.1002/9780470291092.ch20',
 'Mechanisms of Toughening in Ceramic Matrix Composites',
 '1981',
 '9780470291092',
 'book chapter',
 '',
 '']
```

It is worth mentioning that, once you have iterated it once, the CSV reader is *consumed* and does not allow you to iterate over the same object twice. For instance, see the following execution where the same reader is iterated twice:

```python
with open("assets/data/01-publications.csv", "r", encoding="utf-8") as f:
    publications = reader(f)

    print("-- First iteration")
    for row in publications:
        pprint(row)  # all the rows will be printed, one by one

    print("\n-- Second iteration")
    for row in publications:
        pprint(row)  # no row will be printed
```

```
-- First iteration
['doi',
 'title',
 'publication year',
 'publication venue',
 'type',
 'issue',
 'volume']
['10.1002/cfg.304',
 'Development of Computational Tools for the Inference of Protein Interaction '
 'Specificity Rules and Functional Annotation Using Structural Information',
 '2003',
 '1531-6912',
 'journal article',
 '4',
 '4']
```

(continues on next page)

(continued from previous page)

```
['10.1016/s1367-5931(02)00332-0',
 'In vitro selection as a powerful tool for the applied evolution of proteins '
 'and peptides',
 '2002',
 '1367-5931',
 'journal article',
 '3',
 '6']
['10.1002/9780470291092.ch20',
 'Mechanisms of Toughening in Ceramic Matrix Composites',
 '1981',
 '9780470291092',
 'book chapter',
 '',
 '']

-- Second iteration
```

### Casting CSV reader into a list

If you want to iterate over the same rows more than one time, one possibility would be to convert your reader into a list object, by using the `list` constructor:

```python
with open("assets/data/01-publications.csv", "r", encoding="utf-8") as f:
    publications = reader(f)
    publications_list = list(publications)
```

From now on, even if the file object is closed after executing all the instructions within the `with` block, you can always access (and iterate) the rows defined in the original CSV document since you have stored them within a Python list, as shown in the following excerpt:

```python
print("-- First execution")
for row in publications_list:
    pprint(row)

print("\n-- Second execution")
for row in publications_list:
    pprint(row)
```

```
-- First execution
['doi',
 'title',
 'publication year',
 'publication venue',
 'type',
 'issue',
 'volume']
['10.1002/cfg.304',
 'Development of Computational Tools for the Inference of Protein Interaction '
 'Specificity Rules and Functional Annotation Using Structural Information',
 '2003',
```

(continues on next page)

(continued from previous page)

```
 '1531-6912',
 'journal article',
 '4',
 '4']
['10.1016/s1367-5931(02)00332-0',
 'In vitro selection as a powerful tool for the applied evolution of proteins '
 'and peptides',
 '2002',
 '1367-5931',
 'journal article',
 '3',
 '6']
['10.1002/9780470291092.ch20',
 'Mechanisms of Toughening in Ceramic Matrix Composites',
 '1981',
 '9780470291092',
 'book chapter',
 '',
 '']

-- Second execution
['doi',
 'title',
 'publication year',
 'publication venue',
 'type',
 'issue',
 'volume']
['10.1002/cfg.304',
 'Development of Computational Tools for the Inference of Protein Interaction '
 'Specificity Rules and Functional Annotation Using Structural Information',
 '2003',
 '1531-6912',
 'journal article',
 '4',
 '4']
['10.1016/s1367-5931(02)00332-0',
 'In vitro selection as a powerful tool for the applied evolution of proteins '
 'and peptides',
 '2002',
 '1367-5931',
 'journal article',
 '3',
 '6']
['10.1002/9780470291092.ch20',
 'Mechanisms of Toughening in Ceramic Matrix Composites',
 '1981',
 '9780470291092',
 'book chapter',
 '',
 '']
```

## CSV table as list of lists

As you can see from the executing the `print` function in the examples above, each row of the CSV table is represented, in Python, as a list of strings. As such, the overall table, after converted it as a list using the related constructor, is defined as a list of list of strings, following the pattern below (using as example the table introduced at the beginning of this tutorial):

```
my_list = [
    [ "column name", "another name, with a comma" ],           # row 1
    [ "a value", "a value, with a comma" ],                    # row 2
    [ "a quoted \"value\"", "a quoted \"value\", with a comma" ]  # row 3
]
pprint(my_list)
```

```
[['column name', 'another name, with a comma'],
 ['a value', 'a value, with a comma'],
 ['a quoted "value"', 'a quoted "value", with a comma']]
```

As you can see, since strings in Python can be created enclosing their characters between double quotes (i.e. "), the only character to escape in the string is the double quote character itself with a slash (i.e. \"). Alternatively, you could use the the single quote character (i.e. ') for creating strings, avoiding to escape double quote characters, if any:

```
my_list = [
    [ 'column name', 'another name, with a comma' ],          # row 1
    [ 'a value', 'a value, with a comma' ],                   # row 2
    [ 'a quoted "value"', 'a quoted "value", with a comma' ]  # row 3
]
pprint(my_list)
```

```
[['column name', 'another name, with a comma'],
 ['a value', 'a value, with a comma'],
 ['a quoted "value"', 'a quoted "value", with a comma']]
```

Since a table is a list of list, it can be accessed and modified using the common methods available for the class list, as shown in the following excerpt:

```
# retrieving the second row in the table
second_row = publications_list[1]  # remember that item indexes starts from 0
print("-- Second row")
pprint(second_row)

# retrieving the third item in the second row
third_item_second_row = second_row[2]
print("\n-- Third item in second row")
pprint(third_item_second_row)

# appending a new row at the end of the list
publications_list.append([
    "10.1080/10273660500441324",
    "Development of a Species-Specific Model of Cerebral Hemodynamics",
    "2005",
    "1027-3662",
    "journal article",
```

(continues on next page)

(continued from previous page)

```
    "3",
    "6"
])
print("\n-- Updated list")
pprint(publications_list)
```

```
-- Second row
['10.1002/cfg.304',
 'Development of Computational Tools for the Inference of Protein Interaction '
 'Specificity Rules and Functional Annotation Using Structural Information',
 '2003',
 '1531-6912',
 'journal article',
 '4',
 '4']

-- Third item in second row
'2003'

-- Updated list
[['doi',
  'title',
  'publication year',
  'publication venue',
  'type',
  'issue',
  'volume'],
 ['10.1002/cfg.304',
  'Development of Computational Tools for the Inference of Protein Interaction '
  'Specificity Rules and Functional Annotation Using Structural Information',
  '2003',
  '1531-6912',
  'journal article',
  '4',
  '4'],
 ['10.1016/s1367-5931(02)00332-0',
  'In vitro selection as a powerful tool for the applied evolution of proteins '
  'and peptides',
  '2002',
  '1367-5931',
  'journal article',
  '3',
  '6'],
 ['10.1002/9780470291092.ch20',
  'Mechanisms of Toughening in Ceramic Matrix Composites',
  '1981',
  '9780470291092',
  'book chapter',
  '',
  ''],
 ['10.1080/10273660500441324',
  'Development of a Species-Specific Model of Cerebral Hemodynamics',
```

(continues on next page)

(continued from previous page)

```
  '2005',
  '1027-3662',
  'journal article',
  '3',
  '6']]
```

### CSV writer

Once created or modified a table defined through a list of lists in Python, it can be necessary to store it into a CSV file. To do so, we can use the constructor `writer` included in the package `csv`, that must be imported. As we did for loading the content of a CSV file in Python, we use again the `open` function within a `with` statement, but the file path of the first parameter indicates the file where to store the table and we specify `"w"` as the mode to interact with the file to create a new object file, as shown as follows:

```
from csv import writer

with open("assets/data/01-publications-modified.csv", "w", encoding="utf-8") as f:
    publications_modified = writer(f)
    publications_modified.writerows(publications_list)  # it writes all the rows in the␣
↪list of lists
```

As shown in the code above, the constructor `writer` takes in input again a file object and returns an object having class `_csv.writer`. This class includes some methods to write new rows in the file pointed by the file object. In particular, the method `writerows` can be used to write the table defined as a list of lists (of strings) into the file.

### DictReader and DictWriter

In the previous section, we have seen how to load and store in Python a CSV table defined as a list of lists. There is, though, another approach that can be used to load and store CSV files using Python that represents the CSV tables as list of *dictionaries*. In this case, each key in the dictionary is a label of one of the columns of the table, that must be specified. The class `DictReader` (that must be imported as usual) is used to load a CSV table in this form, as shown in the following excerpt:

```
from csv import DictReader

with open("assets/data/01-publications-modified.csv", "r", encoding="utf-8") as f:
    publications_modified = DictReader(f)  # it is a reader operating as a list of␣
↪dictionaries
    publications_modified_dict = list(publications_modified)  # casting the reader as a␣
↪list

pprint(publications_modified_dict)
```

```
[{'doi': '10.1002/cfg.304',
  'issue': '4',
  'publication venue': '1531-6912',
  'publication year': '2003',
  'title': 'Development of Computational Tools for the Inference of Protein '
           'Interaction Specificity Rules and Functional Annotation Using '
           'Structural Information',
```

(continues on next page)

(continued from previous page)

```
  'type': 'journal article',
  'volume': '4'},
 {'doi': '10.1016/s1367-5931(02)00332-0',
  'issue': '3',
  'publication venue': '1367-5931',
  'publication year': '2002',
  'title': 'In vitro selection as a powerful tool for the applied evolution of '
           'proteins and peptides',
  'type': 'journal article',
  'volume': '6'},
 {'doi': '10.1002/9780470291092.ch20',
  'issue': '',
  'publication venue': '9780470291092',
  'publication year': '1981',
  'title': 'Mechanisms of Toughening in Ceramic Matrix Composites',
  'type': 'book chapter',
  'volume': ''},
 {'doi': '10.1080/10273660500441324',
  'issue': '3',
  'publication venue': '1027-3662',
  'publication year': '2005',
  'title': 'Development of a Species-Specific Model of Cerebral Hemodynamics',
  'type': 'journal article',
  'volume': '6'}]
```

As you can see from the output of the execution of the code above, the list defined by casting the `DictReader` object, created by passing as input the file object as before, contains dictionaries, where each dictionary represent a row. The values of the cells of each row can be accessed by using the related key which is, as anticipated, one of the label of the columns. It is worth mentioning that, in this case, the first row in the CSV table is always interpreted as the header of the table, and the content of the list of ditionaries will start considering only the values specified from the second row. The following code shows some example about how to interact with such a structure for accessing and modifying the table:

```
# retrieving the second row in the table
second_row = publications_modified_dict[1]  # remember that item indexes starts from 0
print("-- Second row")
pprint(second_row)

# retrieving the value associated with the column 'title' in the second row
title_value_second_row = second_row["title"]
print("\n-- Value assigned to 'title' in second row")
print(title_value_second_row)

# appending a new row at the end of the list
publications_modified_dict.append({
    "doi": "10.1080/10273660412331292260",
    "title": "Amplified Molecular Binding of Prion Protein Homologues in Self-
↪Progressive Injury of Neuronal Membranes and Trafficking Systems",
    "publication year": "2003",
    "publication venue": "1027-3662",
    "type": "journal article",
    "issue": "3-4",
```

(continues on next page)

(continued from previous page)

```
    "volume": "5"
})
print("\n-- Updated list of dictionaries")
pprint(publications_modified_dict)
```

```
-- Second row
{'doi': '10.1016/s1367-5931(02)00332-0',
 'issue': '3',
 'publication venue': '1367-5931',
 'publication year': '2002',
 'title': 'In vitro selection as a powerful tool for the applied evolution of '
          'proteins and peptides',
 'type': 'journal article',
 'volume': '6'}

-- Value assigned to 'title' in second row
In vitro selection as a powerful tool for the applied evolution of proteins and peptides

-- Updated list of dictionaries
[{'doi': '10.1002/cfg.304',
  'issue': '4',
  'publication venue': '1531-6912',
  'publication year': '2003',
  'title': 'Development of Computational Tools for the Inference of Protein '
           'Interaction Specificity Rules and Functional Annotation Using '
           'Structural Information',
  'type': 'journal article',
  'volume': '4'},
 {'doi': '10.1016/s1367-5931(02)00332-0',
  'issue': '3',
  'publication venue': '1367-5931',
  'publication year': '2002',
  'title': 'In vitro selection as a powerful tool for the applied evolution of '
           'proteins and peptides',
  'type': 'journal article',
  'volume': '6'},
 {'doi': '10.1002/9780470291092.ch20',
  'issue': '',
  'publication venue': '9780470291092',
  'publication year': '1981',
  'title': 'Mechanisms of Toughening in Ceramic Matrix Composites',
  'type': 'book chapter',
  'volume': ''},
 {'doi': '10.1080/10273660500441324',
  'issue': '3',
  'publication venue': '1027-3662',
  'publication year': '2005',
  'title': 'Development of a Species-Specific Model of Cerebral Hemodynamics',
  'type': 'journal article',
  'volume': '6'},
 {'doi': '10.1080/10273660412331292260',
  'issue': '3-4',
```

(continues on next page)

(continued from previous page)

```
 'publication venue': '1027-3662',
 'publication year': '2003',
 'title': 'Amplified Molecular Binding of Prion Protein Homologues in '
          'Self-Progressive Injury of Neuronal Membranes and Trafficking '
          'Systems',
 'type': 'journal article',
 'volume': '5'}]
```

As before, once created or modified a table defined through a list of ditionaries, you can store it into a CSV file using the class `DictWriter` included in the package `csv` (to be imported, as usual). As we did before, we use again the `open` function within a `with` statement, but the file path of the first parameter indicates the file where to store the table and we specify `"w"` as the mode to interact with the file to create a new object file, as shown as follows:

```
from csv import DictWriter

with open("assets/data/01-publications-modified-dict.csv", "w", encoding="utf-8") as f:
    header = [  # the fields defining the columns must be explicitly specified in the␣
↪desired order
        "doi", "title", "publication year", "publication venue", "type", "issue", "volume
↪" ]

    publications_modified = DictWriter(f, header)
    publications_modified.writeheader()  # the header must be explicitly created in the␣
↪output file
    publications_modified.writerows(publications_modified_dict)  # it writes all the␣
↪rows, as usual
```

However, the class `DictWriter` works in a slightly different way of `_csv.writer`. The main differences are:

1. the dictionaries representing the rows do not specify a precise order of the columns to be stored in the CSV file and, as such, it must be explicitly defined;
2. there is no header explicitly specified as a row of the table and, as such, it must be provided to the constructor of the class `DictWriter` and then written as first thing in the file.

For addressing 1), we simply create a new list (the variable `header` of the code above) with all the column labels in the order they must appear in the final file. Instead, for addressing 2), it is sufficient to specify such a new list as the second parameter of the `DictWriter` constructor, after the file object where to store the table; then, it is necessary to write the header of the table calling the method `writeheader()` before writing the rows with data into the file using the method `writerows`.

## CSV dialects

In the previous sections we showed how to use the classes and functions in the package `csv` in Python to handle CSV documents. It is worth mentioning, though, that CSV has several dialects that introduce small changes in the structure of a CSV document. For instance, a well-known dialect is named Tab-separated Values (TSV). Here the idea is that one has to use the tab character to separate the cells of a row instead of the comma. As such, the comma does not have any specific meaning and can be safely used in cell values withou escaping it with quote characters.

For instance, the very first example of CSV table introduced at the beginning of this tutorial can be represented in TSV as follows:

```
column name	another name, with a comma
a value	a value, with a comma
a quoted "value"	a quoted "value", with a comma
```

Of course, the `csv` package allows one to parse also these additional kinds of formats. Indeed, all the constructors of readers and writers objects (i.e. `reader`, `writer`, `DictReader` and `DictWriter`) can have in input the optional named parameter `dialect` which permits the specification of the dialect to consider for either loading or storing the CSV-like table. For instance, the following code stores the table considered in the previous excerpt of code as a TSV file:

```
with open("assets/data/01-publications-modified-dict.tsv", "w", encoding="utf-8") as f:
    header = [  # the fields defining the columns must be explicitly specified in the␣
↪desired order
        "doi", "title", "publication year", "publication venue", "type", "issue", "volume
↪" ]

    publications_modified = DictWriter(f, header, dialect="excel-tab")  # adding the␣
↪specific dialect
    publications_modified.writeheader()  # the header must be explicitly created in the␣
↪output file
    publications_modified.writerows(publications_modified_dict)  # it writes all the␣
↪rows, as usual
```

In the code above, we have specified a different output file in the `with` statement (i.e. the extension now is `.tsv`), and we have explicited asked our `DictWriter` to use the tab-separated dialect introduced by Excel (i.e. `dialect="excel-tab"`) to handle the table as a TSV file.

### 2.1.2 JavaScript Object Notation (JSON)

Another format well-known in the Web, since it is used in several different scenarios that concern data interchange, is the Javascript Object Notation (JSON). It is a simple textual format to describe objects which follow the key-value approach to specify data, where the key is always a term written within quotes, while the value can assume any of the following types:

- numbers (integers and floats), specified straight without an markup (e.g. `3` or `3.14`);
- strings, specified between double quotes (e.g. `"a string"`);
- booleans, the values `true` and `false`;
- object, a collection of key-value pairs specified within curly brackets, where each pair is separated with a comma (e.g. `{ "given name": "Silvio", "family name": "Peroni" }`);
- the null value, i.e. `null`, which mimic the `None` value in Python;
- arrays, i.e. lists of values (numbers, strings, booleans, objects, other arrays, etc.) listed between square brackets where each item is separated with a comma (e.g. `[ "a string", "another string", 4, 4.5, true ]`).

Thus, instead of CSV documents in which all the values are actually interpreted as strings, in a JSON document all the values can have different types, as shown above. In addition, each JSON document does not contain necessarily one single object (using curly brackets), but can be defined as an array of objects, and each object can contain (as some of its values) other objects, organising a tree-like structure. An example of such structure is shown in the exemplar JSON file provided in this tutorial, where all the publications and venues specified in the CSV files introduced at the very beginning of the tutorial have been reorganised according to the JSON syntax. As before, to open it as a source file, you can right-click on it in the left panel and select *Open With -> Editor*.

### Loading a JSON document in Python

We need to use specific functions of the Python package `json` to load a JSON document in Python. In particular, we use the function `load` to import in Python a JSON object, that must be imported from the `json` package as usual.

```
from json import load

with open("assets/data/01-publications-venues.json", "r", encoding="utf-8") as f:
    json_doc = load(f)
```

Differntly from the handling of CSV documents, the `load` function (that still takes in input the file object of the file to load) does not return you a reader, but rather it provides directly the representation of the JSON document into the appropriate Python data structures.

### It is a list of dictionaries!

Considering the exemplar JSON file we have used in the code above, we can print out on screen the type of the object referred by the `json_doc` variable to see what kind of class it is used to represent such a document, as shown in the following excerpt:

```
print(type(json_doc))
```

```
<class 'list'>
```

As you can see, a list is used to map the JSON array, which is indeed the most natural choice. In particular, the kind of JSON values mentioned above are converted in Python as follows:

- numbers (e.g. `3` or `3.14`) and strings (e.g. `"a string"`) in JSON are represented with the kinds of values in Python (i.e. `3`, `3.14` and `"a string"`);
- the `true` and `false` boolean values in JSON are represented in Python using `True` and `False` respectively;
- each JSON object is represented by a Python dictionary, having strings specified as keys and the appropriate kind of value assigned to their values;
- JSON arrays, as already mentioned, are represented with Python lists.

Thus you can act upon the JSON object loaded in Python as you do with the classes used to represent the various JSON values. For instance, in the following code, we show some specific item of the JSON array and add another object to the list, which includes a new publication:

```
# retrieving the second item in the JSON array
second_item = json_doc[1]  # remember that item indexes starts from 0
print("-- Second item")
pprint(second_item)

# retrieving the value associated with the key 'title' in the second item
title_value_second_item = second_item["title"]
print("\n-- Value assigned to 'title' in second item")
print(title_value_second_item)

# appending a new JSON object at the end of the list
json_doc.append({
    "doi": "10.1080/10273660412331292260",
    "title": "Amplified Molecular Binding of Prion Protein Homologues in Self-
↪Progressive Injury of Neuronal Membranes and Trafficking Systems",
```

(continues on next page)

(continued from previous page)

```
    "publication year": 2003,
    "publication venue": {
        "id": [ "1027-3662" ],
        "name": "Journal of Theoretical Medicine",
        "type": "journal"
    },
    "type": "journal article",
    "issue": "3-4",
    "volume": "5"
})
print("\n-- Updated JSON array (a.k.a. list of dictionaries)")
pprint(json_doc)
```

```
-- Second item
{'doi': '10.1016/s1367-5931(02)00332-0',
 'issue': '3',
 'publication venue': {'id': ['1367-5931'],
                       'name': 'Current Opinion in Chemical Biology',
                       'type': 'journal'},
 'publication year': 2002,
 'title': 'In vitro selection as a powerful tool for the applied evolution of '
          'proteins and peptides',
 'type': 'journal article',
 'volume': '6'}

-- Value assigned to 'title' in second item
In vitro selection as a powerful tool for the applied evolution of proteins and peptides

-- Updated JSON array (a.k.a. list of dictionaries)
[{'doi': '10.1002/cfg.304',
  'issue': '4',
  'publication venue': {'id': ['1531-6912'],
                        'name': 'Comparative and Functional Genomics',
                        'type': 'journal'},
  'publication year': 2003,
  'title': 'Development of Computational Tools for the Inference of Protein '
           'Interaction Specificity Rules and Functional Annotation Using '
           'Structural Information',
  'type': 'journal article',
  'volume': '4'},
 {'doi': '10.1016/s1367-5931(02)00332-0',
  'issue': '3',
  'publication venue': {'id': ['1367-5931'],
                        'name': 'Current Opinion in Chemical Biology',
                        'type': 'journal'},
  'publication year': 2002,
  'title': 'In vitro selection as a powerful tool for the applied evolution of '
           'proteins and peptides',
  'type': 'journal article',
  'volume': '6'},
 {'doi': '10.1002/9780470291092.ch20',
  'publication venue': {'id': ['9780470291092'],
```

(continues on next page)

(continued from previous page)

```
                        'name': 'Proceedings of the 5th Annual Conference on '
                                'Composites and Advanced Ceramic Materials: '
                                'Ceramic Engineering and Science Proceedings',
                        'type': 'book'},
  'publication year': 1981,
  'title': 'Mechanisms of Toughening in Ceramic Matrix Composites',
  'type': 'book chapter'},
 {'doi': '10.1080/10273660412331292260',
  'issue': '3-4',
  'publication venue': {'id': ['1027-3662'],
                        'name': 'Journal of Theoretical Medicine',
                        'type': 'journal'},
  'publication year': 2003,
  'title': 'Amplified Molecular Binding of Prion Protein Homologues in '
           'Self-Progressive Injury of Neuronal Membranes and Trafficking '
           'Systems',
  'type': 'journal article',
  'volume': '5'}]
```

## Storing a JSON document into a file

We use the function dump of the `json` package (to import as usual) to store a dictionary or an array of values into a JSON file, as shown in the following excerpt:

```
from json import dump

with open("assets/data/01-publications-venues-modified.json", "w", encoding="utf-8") as␣
↪f:
    dump(json_doc, f, ensure_ascii=False, indent=4)
```

The `dump` function takes in input two mandatory positional parameters - i.e. the Python representation of a JSON document as the first parameter and the file object referring to the file where to store the JSON document. In addition, it can takes several other optional named parameters, two of which are strongly suggested and have been used in the code above.

The parameter `ensure_ascii` (assigned to `True` by defalut) is responsible to keep every string value compliant with the ASCII character encoding, which will result in escaping all non-ASCII characters (that are only 128 characters). This is very undesirable when we have natural language text in the JSON object we want to store since, for instance, all the characters with accents (e.g. `"è"`) will be encoded in a different way (in the example, `"\u00e8"`, that is the UTF-8 code of the character `"è"`). That is why the code above sets the `ensure_ascii` input parameter to `False`: to avoid such an escaping, preserving the original characters as they are (i.e. encoded in UTF-8).

Instead, the parameter `indent` is used to specify how many white spaces to add for indenting the various key-value pairs in the JSON document. The choice to specify the indent is only for human consumption, since a machine does not care about how these pairs are visually organised in the document. Indeed, the JSON document

```
{ "given name": "Silvio", "family name": "Peroni }
```

and the JSON document

```
{
    "given name": "Silvio",
```

(continues on next page)

(continued from previous page)

```
    "family name": "Peroni"
}
```

are actually storing the same data, but the second is usually easier to read for humans. That is why the code above sets the input parameter `indent` to `4`, meaning that four spaces must be used to indent the various JSON pairs.

## 2.2 Implementation of data models via Python classes

In this tutorial, we see how to create Python classes to implement a model for the representation of data.

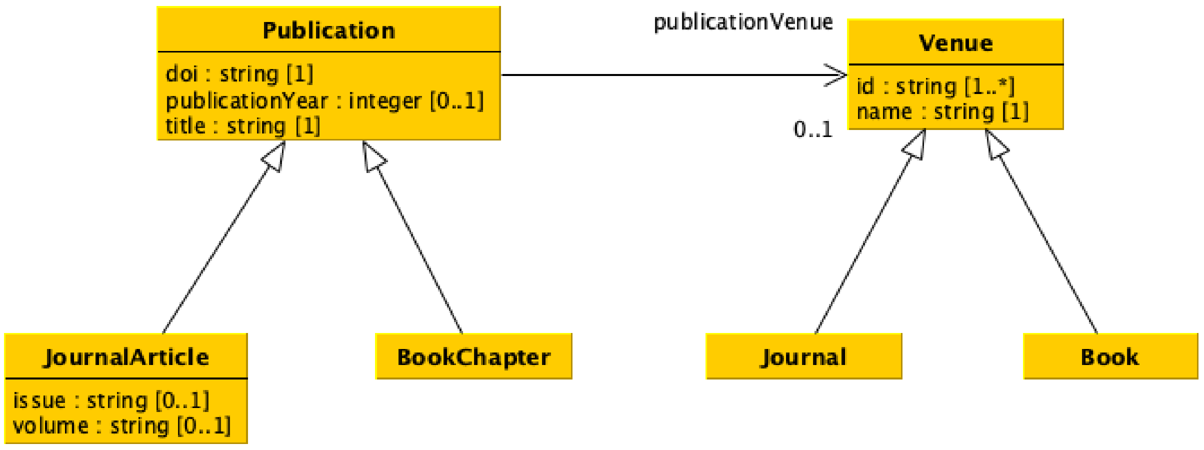
### 2.2.1 What is a class in Python

In Python, as in other object-oriented programming languages, a class is an extensible template for creating objects having a specific type. All the basic types of values (e.g. strings, integers, booleans) and the other data structures (e.g. lists, sets, dictionaries) are defined by means of particular classes.

In addition, each class makes available a set of methods that allow one to interact with the objects (i.e. the instances) of such a class. A method is a particular function that can be run only if directly called via an object. For instance, the instruction `"this is a string".split(" ")` executes the method `split` passing `" "` as the input parameter on the particular string object on which the method is called, i.e. the string `"this is a string"` (defined by the class `str` in Python).

### 2.2.2 Defining a data model using Python classes

Python classes, as the name may recall, can be used to implement a particular data model such as that introduced in the following diagram using the Unified Modelling Language (UML). We will use this example to understand how to implement classes in Python, and to show how they works.



As you can see from the diagram above, we defined six distinct classes which are, somehow, related to each other. Let us see how to define this structure in Python.

### Defining our first class

For defining classes in Python, one has to use the following signature:

```
class <class name>(<superclass 1>, <superclass 2>, ...):
    def __init__(self, <param 1>, <param 2>, ...):
        ...
```

In the excerpt above, `<class name>` is the name one wants to assign to a class, while `<superclass 1>`, `<superclass 2>`, etc., indicate the superclasses from which this class is derived from. In Python, all new classes must be subclass of the generic class `object`. Instead, the indented `def __init__` is a special methods defining the constructor of an object of that class, and it will called every time one wants to create a new object (instance) of this type. For instance, when we create a new set in Python using `set()`, we are calling the constructur of the class `set`, defined as shown above.

It is worth mentioning that all the methods of a class, including its constructor, must specify `self` as the first parameter. This special parameter represents the instance of the class in consideration. In practice, every time we instantiate a new object of that class, `self` will be assigned to that object and provides access to its attributes (i.e. variables assigned with particular values for that object) and methods as defined in the related class. In particular, it is used to access to all object related information within the class itself.

For instance, by using such a `self` parameter, it is possible to create variables and associated values that are local to a particular object of that class. In the following excerpt, we use it to define the constructor of the the class `Venue` in the data model shown above as a UML diagram:

```
class Venue(object):
    def __init__(self, identifiers, name):
        self.id = set()
        for identifier in identifiers:
            self.id.add(identifier)

        self.name = name
```

As shown in the code above, the class `Venue` is defined as subclass of the top class `object`, and its constructor takes in input three parameters: `self` (as explained above), `identifiers` and `name`.

The parameter `identifiers` is used to take in input a collection of strings that contains all the identifiers of such an object. In the above code, I decided to handle all the items included in the collection using a set to comply with the declaration in the data model class which wants to have a collection of at least one or more string identifiers (`id : string [1..*]`). Indeed, I have created a new variable `id` related to the particular object of the class `self` (i.e. `self.id`) and I have assigned a new set to it. Then, I added all the identifiers in the input collection to the set using the set method add (i.e. via the instruction `self.id.add(identifier)`.

Instead, the parameter `name` is used to specify the name of a particular venue. Thus, I have just assigned it to the variable `name` of the object `self` (i.e. `self.name`) to mimic the data model attribute `name : str [1]`. Of course, I could also use a different structure to store this information - for instance, I could use again a set which contained only one value in it. The important thing here, while trying to map the data model into a Python class, is to be compliant with the data model declaration. I chose to assigned it straight with a variable supposing that the input will be a simple string.

In practice, thanks to the `self` keyword, I can create new independent variables for each new object created using this class.

## Representing relations in Python

The Python class defined above represents (by means of its constructor) all the attributes associated to the related data model class. However, in data models, there are also relations that may exist between different kinds of objects, as the relation `publicationVenue` between the data model classes `Publication` and `Venue`. In Python, such relations can be represented as the other attributes, i.e. by assigning some specific values to `self`-declared variables, as shown in the following excerpt:

```
class Publication(object):
    def __init__(self, doi, publicationYear, title, publicationVenue):
        self.doi = doi
        self.publicationYear = publicationYear
        self.title = title
        self.publicationVenue = publicationVenue
```

As shown in the except above, the constructor of the class `Publication` takes in input not only the attributes of the related data model class but also its relations (i.e. the relation from which the class is the starting point), and considers it as additional parameters of the constructor. Then, they will be handled as the others. Of course, the type of object that should be specified in the parameter `publicationVenue` should have class `Venue`, defined above.

## Instantiating a class

Once classes are defined, we can use them to instantiate objects of that kind. For doing it, we should call their constructor (using the name of the class) passing the parameters it requires **except** `self`, that will be implicitly considered. In practice, for creating a new object of class `Venue`, we need to specify only two parameters, i.e. those for `identifiers` (i.e. a collection of strings) and `name` (i.e. a string). As an example, let us consider again the first two items of the venues CSV file we have introduced in the previous tutorial, i.e.:

| id | name | type |
|---|---|---|
| 1531-6912 | Comparative and Functional Genomics | journal |
| 1367-5931 | Current Opinion in Chemical Biology | journal |

These two entities (i.e. venues) can be defined using the Python class `Venue` as follows:

```
venue_1 = Venue(["1531-6912"], "Comparative and Functional Genomics")
venue_2 = Venue(["1367-5931"], "Current Opinion in Chemica Biology")
```

As shown in the above excerpt, I have created two new objects, assigned to two distinct variables, one for each venue. All the values specified as input of the constructur have been assigned to the `self` variables of each object, that are distinct, while share the same structure. Indeed, using the Python built-in function `id` (that takes in input an object and returns the unique integer identifying it) and function `type` (that takes in input an object and returns its related type), it is possible to see that `venue_1` and `venue_2` are different objects of the same class:

```
print("The objects in 'value_1' and 'value_2' share the same class -->", type(venue_1)␣
↪== type(venue_2))
print("Indeed, the types of the two objects are both", type(venue_1))

print("\nThe objects in 'value_1' and 'value_2' are the same object -->", id(venue_1) ==␣
↪id(venue_2))
print("Indeed, the integers identifying the two objects are", id(venue_1), "and",␣
↪id(venue_2), "respectively")
```

```
The objects in 'value_1' and 'value_2' share the same class --> True
Indeed, the types of the two objects are both <class '__main__.Venue'>

The objects in 'value_1' and 'value_2' are the same object --> False
Indeed, the integers identifying the two objects are 140166117047552 and 140166117047744␣
↪respectively
```

Similarly, we can create new objects also of other classes, such as `Publication`. In this case, the last parameter of the constructor of `Publication` (i.e. `publicationVenue`) should take in input an object having class `Venue` as defined above. As another example, let us consider again the first two items of the publications CSV file we have introduced in the previous tutorial, i.e.:

| doi | title | publication year | publication venue | type | issue | volume |
|---|---|---|---|---|---|---|
| 10.1002/cfg.304 | Development of Computational Tools for the Inference of Protein Interaction Specificity Rules and Functional Annotation Using Structural Information | 2003 | 1531-6912 | journal article | 4 | 4 |
| 10.1016/s1367-5931(02)00332-0 | In vitro selection as a powerful tool for the applied evolution of proteins and peptides | 2002 | 1367-5931 | journal article | 3 | 6 |

These two publications can be defined using the Python class `Publication` as follows:

```
publication_1 = Publication("10.1002/cfg.304",
                            2003,
                            "Development of Computational Tools for the Inference of␣
↪Protein Interaction Specificity Rules and Functional Annotation Using Structural␣
↪Information",
                            venue_1)

publication_2 = Publication("10.1016/s1367-5931(02)00332-0",
                            2002,
                            "In vitro selection as a powerful tool for the applied␣
↪evolution of proteins and peptides",
                            venue_2)
```

It is worth mentioning that, as shown in the excerpt above, we have not specified the identifier of a particular venue as input, bur rather we have provided the `Venue` object representing such a venue, as also defined by the relation `publicationVenue` specified in the data model.

### Creating subclasses of a given class

As you may have noticed, we did not map all the columns of the CSV documents introduced above in the classes we have defined. Indeed, the data model above actually specifies some of this information (for instance the concept of publication type and the fields `issue` and `volume`) into subclasses of `Publication` and `Venue`. Python makes available a mechanism to create new classes as subclasses of existing ones, thus inheriting all the attributes and methods that the superclasses already implement, similar to what a data model enables.

We can use the same signature adopted for classes for creating subclasses by specifying the classes to extend in the definition of the class, as we already did specifying the class `object` as top class of `Publication` and `Venue`, as shown as follows:

```
class BookChapter(Publication):
    pass

class Journal(Venue):
    pass

class Book(Venue):
    pass
```

In the code above, the body of each class extending the classes `Publication` and `Venue` is left unspecified. This means that the new subclasses inherit (and can access via `self`) all the attributes and methods (including the constructor) from the superclass. Thus, the only thing they really add in this case is the specification of a new characterising type, which mimic the `type` field of the CSV file presented above.

However, adding such new information is enough for classifying them as distinct classes, even if one (e.g. `Journal`) is subclass of another (e.g. `Venue`). Indeed, in the following code, I create a new instance of the class `Journal` using the same input values of `value_1`, specified above. As you can see, the classes returned by these two objects are indeed different:

```
# An object of class 'Journal' is instantiated using the same parameters
# of the constructor of its parent class 'Venue' since 'Journal' does not
# define any explicit constructor
journal_1 = Journal(["1531-6912"], "Comparative and Functional Genomics")

print("The objects in 'journal_1' and 'venue_1' share the same class -->", type(journal_
↪1) == type(venue_1))
print("Indeed, the types of the two objects are", type(journal_1), "and", type(venue_1),
↪"respectively")
```

```
The objects in 'journal_1' and 'venue_1' share the same class --> False
Indeed, the types of the two objects are <class '__main__.Journal'> and <class '__main__.
↪Venue'> respectively
```

Of course, in some cases, the new subclass may take in input additional information compared to its superclass. In these cases, e.g. for mapping in Python the data model class `JournalArticle` that introduces also the attributes `issue` and `volume`, it would be necessary to define an appropriate constructor extending that of the parent superclass. An implementation of the Python class `JournalArticle` is shown as follows:

```
class JournalArticle(Publication):
    def __init__(self, doi, publicationYear, title, publicationVenue, issue, volume):
        self.issue = issue
        self.volume = volume
```

(continues on next page)

(continued from previous page)

```
        # Here is where the constructor of the superclass is explicitly recalled, so as
        # to handle the input parameters as done in the superclass
        super().__init__(doi, publicationYear, title, publicationVenue)
```

In the code above, the additional parameters `issue` and `venue` are handled as before, while all the other are tranferred to the constructor of the superclass accessed by using the function `super` (which returns a proxy object that delegates method calls to the parent class) and then calling the `__init__` constructor with all the expected parameters **except** `self`. In this case, to instantiate an object of class `JournalArticle`, all the input parameters must be specified:

```
journal_article_1 = JournalArticle("10.1002/cfg.304",
                                   2003,
                                   "Development of Computational Tools for the Inference␣
↪of Protein Interaction Specificity Rules and Functional Annotation Using Structural␣
↪Information",
                                   journal_1,
                                   "4",
                                   "4")
```

## 2.2.3 Extending classes with methods

Once an object of a certain class is created, one can access all its attributes (i.e. those assigned to `self` variables) directly by their name using the following syntax: `<object>.<attribute name>`. For instance, if we want to print on screen the title of the journal article we have just created, we can run the following code:

```
print(journal_article_1.title)
```

```
Development of Computational Tools for the Inference of Protein Interaction Specificity␣
↪Rules and Functional Annotation Using Structural Information
```

In principle, such a way to referring to specific attributes of an object allows one to also modify the value of their attributes directly, by assigning them to a new value as we do for any variable, for instance:

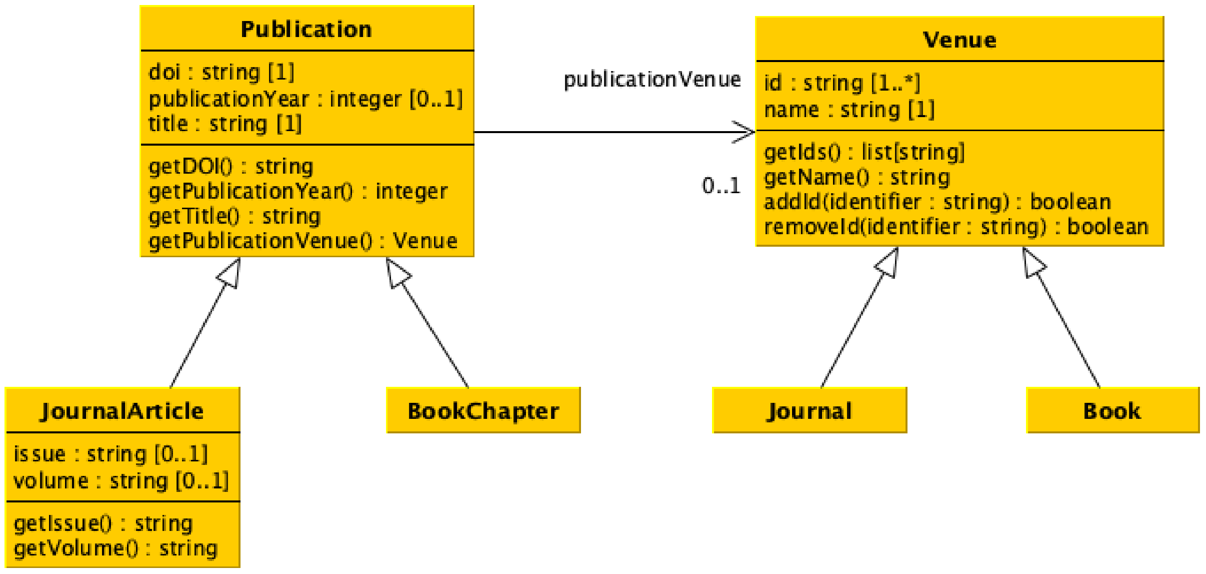
```
journal_article_1.title = "My new title!"
print(journal_article_1.title)
```

```
My new title!
```

However, this way of modifying object attributes is not safe and may have undesired outcomes if someone does not know how to deal with it properly. Thus, in order to facilitate the interaction with object's attribute and to provide operation to work with and manipolate them, Python (and all the other object-oriented programming languages) allows one to create methods.

A method of a class encapsulate an operation that can be run on an object of that class and that can, in principle, be responsible to act upon the attributes related to that object. In practice, methods are just functions tied to specific classes, and can provide also a mechanism to read (safely) values assigned to object attributes without accessing directly to them.

We can define method visually by using UML, the same language we have initially adopted for defining our exemplar data model. Indeed, UML has been originally developed as a general-purpose modeling language in the field of software engineering, and provides widgets that permit the description of a software system including classes and their methods – even if it can be useful also in the task of defining a generic data model. The following diagram shows an extension of the data model presented above with the inclusion of new methods for accessing and, in some cases, modifying the status of particular object attributes.



In UML, the methods are listed just after the attributes of a given class, following the signature:

```
<method name>(<param 1> : <class of param 1>, <param 2> : <class of param 2>, ...) :
↪<type of value returned>
```

For instance, the method `getDOI()` (no input needed here) of the class `Publication` returns a string, i.e. the DOI assigned to the particular publication; instead, the method `addId(identifier : string)` returns a boolean value that states if the operation of adding the string `identifer` to the set of identifiers of the class `Venue` went well (i.e. returned `True`) or not (i.e. returned `False`). Of course, this precise specification of the meaning of the return value of each method is not defined in the diagram itself, but it accompanies somehow the descriptive diagram as a natural language description of what the method should do. However, the diagram already provides the means of the kinds of input and the related output each method must to take and provide, respectively.

### Defining a method in Python

Python uses the same structure seen for the constructor for defining all the other methods:

```
def <name of the method>(self, <param 1>, <param 2>, ...):
    ...
```

The only thing that changes here is that one can specify the name of the method. For instance, let us define all the method of the class `Publication` as defined in the diagram - the rationale behind each method should be self-explanatory:

```
class Publication(object):
    def __init__(self, doi, publicationYear, title, publicationVenue):
        self.doi = doi
        self.publicationYear = publicationYear
        self.title = title
        self.publicationVenue = publicationVenue

```

(continues on next page)

(continued from previous page)

```
    def getDOI(self):
        return self.doi

    def getPublicationYear(self):
        return self.publicationYear

    def getTitle(self):
        return self.title

    def getPublicationVenue(self):
        return self.publicationVenue
```

As shown in the code above, the methods defined add a few hooks to access the value of all the attributes of the class. Then, one can use call methods as done for the other built-in classes, i.e. using the signature `<object>.<method to call>(<value 1>, <value 2>, ...)` (as for the constructor, the `self` parameter must not be specified when calling a method), as shown as follows:

```
# It uses the most recent definition of the class 'Publication', i.e. that with
# the new methods implemented
publication_2 = Publication("10.1016/s1367-5931(02)00332-0",
                            2002,
                            "In vitro selection as a powerful tool for the applied␣
↪evolution of proteins and peptides",
                            venue_2)

print("-- The title of this publication is:")
print(publication_2.getTitle())
```

```
-- The title of this publication is:
In vitro selection as a powerful tool for the applied evolution of proteins and peptides
```

Using methods permits one to detach the ratrionale used to store information about the attributes from the particular contract-like committment defined by the UML diagram, that is what the user expects from running a method. For instance, let us see the methods of the class `venue`:

```
class Venue(object):
    def __init__(self, identifiers, name):
        self.id = set()
        for identifier in identifiers:
            self.id.add(identifier)

        self.name = name

    def getIds(self):
        result = []
        for identifier in self.id:
            result.append(identifier)
        result.sort()
        return result

    def getName(self):
        return self.name
```

(continues on next page)

(continued from previous page)

```
    def addId(self, identifier):
        result = True
        if identifier not in self.id:
            self.id.add(identifier)
        else:
            result = False
        return result

    def removeId(self, identifier):
        result = True
        if identifier in self.id:
            self.id.remove(identifier)
        else:
            result = False
        return result
```

As you can see from the new UML diagram with methods, the method `getIds` must return a list of strings, even we have originally defined the attribute `self.id` as a set. Thus, it is up to the method to implement the request as defined in the diagram. In particular, in the implementation above, a new list has been created which contains the same identifiers in the attrubute set `self.id`, but ordered alphabetically. The list returned by the method and the set in `self.id` are two different objects (containing the same items), as shown in the following excerpt:

```
venue_1 = Venue(["1531-6912"], "Comparative and Functional Genomics")

print("The value in 'self.id' and that returned by the method 'getIds' are two different␣
↪objects -->")
print(id(venue_1.id) != id(venue_1.getIds()))

print("\nHowever, they both contains the same collection of element -->")
print(len(venue_1.id.difference(venue_1.getIds())) == 0)
```

```
The value in 'self.id' and that returned by the method 'getIds' are two different␣
↪objects -->
True

However, they both contains the same collection of element -->
True
```

This way of handling the interation with class attributes may prevent, also, some undesired effect on mutable values – as a reminder, please see the section "Clarification: immutable and mutable values" in the chapter "Divide and conquer" of the *Computational Thinking and Programming Book*. For instance:

1. What does it happen if the method `getIds` would return directly the set in `self.id`?
2. What does it happen if such a set, retriveved by using the method mentioned in the previous question, is then directly modified by a user using the `add` method of the `set` class?
3. How can the structure of the implementation of `getIds` in the code above prevent these issues?

### What about methods and inheritance

Superclass inheritance applies also to the methods, not only to attributes. For instance, let us introduce the extended implementation of the class `JournalArticle` shown above, where we add also the implementation of the additional two methods `getIssue` and `getVolume` as defined in the last UML diagram:

```
class JournalArticle(Publication):
    def __init__(self, doi, publicationYear, title, publicationVenue, issue, volume):
        self.issue = issue
        self.volume = volume

        # Here is where the constructor of the superclass is explicitly recalled, so as
        # to handle the input parameters as done in the superclass
        super().__init__(doi, publicationYear, title, publicationVenue)

    def getIssue(self):
        return self.issue

    def getVolume(self):
        return self.volume
```

In practice, when we create an new `JournalArticle` object, it will have available the methods the class `JournalArticle` defines plus all those defined by all the ancestor superclasses, at any level of the hierarchy (since I can create a non-circular tree of superclass-subclass relations among a chain of different classes). The following code shows how all both the methods of the two subclass and superclass work as expected in objects having class `JournalArticle`:

```
# It uses the most recent definition of the class 'JournalArticle', i.e. that with
# the new methods implemented
journal_article_1 = JournalArticle("10.1002/cfg.304",
                                   2003,
                                   "Development of Computational Tools for the Inference␣
↪of Protein Interaction Specificity Rules and Functional Annotation Using Structural␣
↪Information",
                                   journal_1,
                                   "4",
                                   "4")

print("-- The title of the journal article (method defined in the superclass 'Publication
↪')")
print(journal_article_1.getTitle())

print("\n-- The title of the journal article (method defined in the class 'JournalArticle
↪')")
print(journal_article_1.getIssue())
```

```
-- The title of the journal article (method defined in the superclass 'Publication')
Development of Computational Tools for the Inference of Protein Interaction Specificity␣
↪Rules and Functional Annotation Using Structural Information

-- The title of the journal article (method defined in the class 'JournalArticle')
4
```

More information about the dynamics of the class inheritance are introduced and detailed in the chapter “Understanding

Inheritance" of *How To Code in Python*.

#### Full UML diagram implementation

I have implemented in a single Python file all the classes introduced in the last UML diagram. They can be imported and reused in other files as shown in the classuse.py file using the following import notation:

```
from <Python file name> import <class 1>, <class 2>, ...
```

You can simply run all the instructions in the latter file running the following command:

```
python classuse.py
```

## 2.3 Introduction to Pandas

In this tutorial, we will see the basics of one of the most useful and used tool in Data Science projects, i.e. Python Pandas.

### 2.3.1 What is Pandas

Pandas is a powerful tool developed for data analysis and data manipulation which is used in several Data Science projects due to its flexibility. It is accompanied by a great user guide, tutorials, a cookbook, an API documentation, other free books (e.g. Python Data Science Handbook), and several articles on the topic (e.g. those available in Programming Historian).

The official website makes available a "Getting started" guide to show how to install and use it. Anyway, you can install Pandas in your machine using the pip command as follows:

```
pip install pandas
```

Among the various things, Pandas introduces two new classes of objects that are used to handle any kind of data in tabular form. They are the class `Series` and the class `DataFrame`, described in the following subsections.

#### What is a Series

A series is a one-dimensional array (i.e. it acts as a list) of objects of any data type (integers, strings, floating point numbers, Python objects, etc.). Each item in the series is indexed by a specific label (it can be an integer, a string, etc.), that can be used to access such an item. If no index is specified, the class `Series` will create such an index automatically using non-negative numbers (i.e. starting counting elements from 0).

| *index (the label)* | *element (the value)* |
|---|---|
| *0* | Ron |
| *1* | Hermione |
| *2* | Harry |
| *3* | Tom |
| *4* | James |
| *5* | Lily |
| *6* | Severus |
| *7* | Sirius |

A new series in Pandas can be created as follows:

```
from pandas import Series

my_series = Series(["Ron", "Hermione", "Harry", "Tom", "James", "Lily", "Severus",
↪"Sirius"])
print(my_series)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from pandas import Series
      3 my_series = Series(["Ron", "Hermione", "Harry", "Tom", "James", "Lily", "Severus
↪", "Sirius"])
      4 print(my_series)

ModuleNotFoundError: No module named 'pandas'
```

As you can see, by printing the series on screen you get several information. The first column defines the indexes used to label each element of the series, which are listed in the second column. Finally, there is an indication of which kind of object are included in the series. In particular, in Pandas, the `object` data type (i.e. `dtype`) is used to define series that are made of string or mixed type objects.

There are, of course, other data types that can be used in series (and, sometimes, automatically inferred by Pandas). If you already know that all the values of a certain series belong to the same data type, e.g. string as in the example above, you can force Pandas to interpret them in such a way by specifying the data type with the input named parameter `dtype` in the constructor. For instance, to list all the values of the previous series as strings, the parameter `dtype` set to the value `string` must be specified. In addition, it is also possible to assign a name (i.e. a string) to the series, using the input named parameter `name`. The use of both `dtype` and `name` are shown as follows:

```
my_series = Series(["Ron", "Hermione", "Harry", "Tom", "James", "Lily", "Severus",
↪"Sirius"],
                   dtype="string", name="given name")
print(my_series)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 my_series = Series(["Ron", "Hermione", "Harry", "Tom", "James", "Lily", "Severus
↪", "Sirius"],
      2                    dtype="string", name="given name")
      3 print(my_series)

NameError: name 'Series' is not defined
```

It is possible to use the slicing mechanism similar to what implemented in Python list (i.e. `<series>[<start>:<end>]` to create subseries on the fly – it works even when the index labels are not integers! In addition, one can use either the method `get` or, alternatively, the instruction `<series>[<index>]` (it is similar to that available in Python dictionaries), taking in input an index label, to retrieve the value at the input index, as show in the following excerpt:

```
sub_series = my_series[1:6]
print("The new subseries is:")
print(sub_series)
```

(continues on next page)

(continued from previous page)

```
print("\nThe element at index 5 of the new subseries is:")
print(sub_series[5])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 sub_series = my_series[1:6]
      2 print("The new subseries is:")
      3 print(sub_series)

NameError: name 'my_series' is not defined
```

While it may seem as a list, a series in Pandas has several additional properties and methods that enable one to access and modify the item in the series in very different ways. Thus, it is more powerful than a simple Python list, and it is the basic structure adopted in the class described in the following section.

### What is a DataFrame

In Pandas, a DataFrame is a table. You can imagine it as a set of named series containing the same amount of elements, where each series defines a column of the table, and all the series share the same index labels (referring to the rows of the table).

| *index (label for rows)* | column name 1 (a series) | column name 2 (another series) |
|---|---|---|
| *0* | Ron | Wisley |
| *1* | Hermione | Granger |
| *2* | Harry | Potter |
| *3* | Tom | Riddle |
| *4* | James | Potter |
| *5* | Lily | Potter |
| *6* | Severus | Snape |
| *7* | Sirius | Black |

A new data frame in Pandas can be created as follows:

```
from pandas import DataFrame

my_dataframe = DataFrame({
    "given name" : my_series,
    "family name" : Series(
        ["Wisley", "Granger", "Potter", "Riddle", "Potter", "Potter", "Snape", "Black"],␣
↪dtype="string")
})

print(my_dataframe)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[4], line 1
----> 1 from pandas import DataFrame
```

(continues on next page)

(continued from previous page)

```
my_dataframe = DataFrame({
    "given name" : my_series,
    "family name" : Series(
        ["Wisley", "Granger", "Potter", "Riddle", "Potter", "Potter", "Snape",
↪"Black"], dtype="string")
})
print(my_dataframe)

ModuleNotFoundError: No module named 'pandas'
```

In this case (but, please, remeber that it is not the only way to create a new data frame), we use as input a dictionary where each key defines a column name and the series associated to such a key contains the values of each cell in that column. Then we can access the various columns in the data frame using the same approach seen for series, i.e. `<dataframe>[<column name>]`, as shown in the following excerpt:

```
family_name_column = my_dataframe["family name"]
print(family_name_column)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 family_name_column = my_dataframe["family name"]
      2 print(family_name_column)

NameError: name 'my_dataframe' is not defined
```

Selecting a column returns a series defining that column that share the column name and the indexes as specified in the data frame. Similarly, using the instruction `<dataframe>.loc[<index label>]`, that takes in input an index label, returns the series defining the row at that input index, as shown in the following example:

```
third_row = my_dataframe.loc[2]
print(third_row)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 third_row = my_dataframe.loc[2]
      2 print(third_row)

NameError: name 'my_dataframe' is not defined
```

As shown in the code above, it returns a series which has the column names of the original data frame as index labels of the series, the name as the index label of the data frame row selected, and the data type of the row derived from the various data types of the data frame columns.

Finally, as seen with the series, also data frame can be sliced (by rows) using the indentical approach introduced in the series. For instance, the following code shows how to create a new data frame taking a selection of the rows:

```
print("The new subdataframe is:")
sub_dataframe = my_dataframe[1:6]
print(sub_dataframe)
```

(continues on next page)

(continued from previous page)

```
print("\nThe row at index 2 of the new subdataframe is:")
print(sub_dataframe.loc[2])
```

```
The new subdataframe is:
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[7], line 2
      1 print("The new subdataframe is:")
----> 2 sub_dataframe = my_dataframe[1:6]
      3 print(sub_dataframe)
      5 print("\nThe row at index 2 of the new subdataframe is:")

NameError: name 'my_dataframe' is not defined
```

Also a data frame in Pandas has several additional properties and methods that enable one to access and modify the cells in the data frame.

## 2.3.2 How to load data into Pandas

Pandas makes available several functions to load data stored in different formats. Indeed, there is a particular function that we can use to load data from CSV representations of tabular data (as those introduced in the *first tutorial*), i.e. the function `read_csv`.

The method `read_csv` takes in input a file path and returns a `DataFrame` representing such tabular data, as shown as follows:

```
from pandas import read_csv

df_publications = read_csv("../01/01-publications.csv")
print(df_publications)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[8], line 1
----> 1 from pandas import read_csv
      3 df_publications = read_csv("../01/01-publications.csv")
      4 print(df_publications)

ModuleNotFoundError: No module named 'pandas'
```

As you can see in the code above, the `print` function does not provide an appropriate visualisation of the data frame in Jupyter Lab, mainly because the content of its columns is more extensive than the example before. In Jupyter, it is possible to have a good preview of a data frame by simply name the variable in a runnable code, as follows:

```
df_publications
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 df_publications
```

(continues on next page)

(continued from previous page)

```

NameError: name 'df_publications' is not defined
```

By looking at the data frame, it appears clear that there is something odd in the data types shown, in particular in the columns `issue` and `volume`. In these columns there are two main issues. The first one concerns the data types associated to the column. Indeed, it seems that Pandas has interpreted automatically these columns as floating numbers, while they should be made of strings! This can be conformed by printing the data type (attribute `dtype`) associated to one of these columns, for instance:

```
print(df_publications["issue"].dtype)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 print(df_publications["issue"].dtype)

NameError: name 'df_publications' is not defined
```

The other issue concerns that strange value in the last row, i.e. `NaN`. This special object is used when there is a missing data in a cell. However, once may expect that, since in the example we are dealing with string, we should simply use an empty string to represent such a missing data instead that such special object.

In order to avoid this behaviour, it may be neccessary to use the method `read_csv` specifying an additional input named parameter, i.e. `dtype`, which enables the specification of a dictionary where the keys are column names, while the values are the strings representing the data type for each column. Instead, to force Pandas to use an empty string as default for string-based column in case of missing values, it is enough to tell the function `read_csv` not to use the default `NaN` for missing value by setting the input named parameter `keep_default_na` to `False`.

The following code shows how to specify such parameters, showing on screen how the new data frame has been modified:

```
df_publications = read_csv("../01/01-publications.csv",
                           keep_default_na=False,
                           dtype={
                               "doi": "string",
                               "title": "string",
                               "publication year": "int",
                               "publication venue": "string",
                               "type": "string",
                               "issue": "string",
                               "volume": "string"
                           })
df_publications
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[11], line 1
----> 1 df_publications = read_csv("../01/01-publications.csv",
      2                            keep_default_na=False,
      3                            dtype={
      4                                "doi": "string",
      5                                "title": "string",
      6                                "publication year": "int",
```

(continues on next page)

(continued from previous page)

```
      7                                "publication venue": "string",
      8                                "type": "string",
      9                                "issue": "string",
     10                                "volume": "string"
     11                            })
     12 df_publications

NameError: name 'read_csv' is not defined
```

### 2.3.3 Iterating over a DataFrame and a Series

Being a table, there are two possible strategies for iterating over a `DataFrame` object: row iteration and column iteration. These two can be achieved by means of two distinct methods of the class `DataFrame`: the method `iterrows` and the method `items`.

The method `iterrows` is used to retrieve a list-like structure where each item has two elements: the index label and a series representing the row related to that index.

```
for idx, row in df_publications.iterrows():
    print("\nThe index of the current row is", idx)
    print("The content of the row is as follows:")
    print(row)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 for idx, row in df_publications.iterrows():
      2     print("\nThe index of the current row is", idx)
      3     print("The content of the row is as follows:")

NameError: name 'df_publications' is not defined
```

If one wants to iterate also over a series representing a row, keeping track of the index labels of the series representing the row (i.e. the column names), one can use the method `items` of the class `Series`:

```
for row_idx, row in df_publications.iterrows():
    print("\nRow index", row_idx)
    for item_idx, item in row.items():
        print(item_idx, "-->", item)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 for row_idx, row in df_publications.iterrows():
      2     print("\nRow index", row_idx)
      3     for item_idx, item in row.items():

NameError: name 'df_publications' is not defined
```

The method `items` of the class `DataFrame` is used to retrieve a list-like structure where each item has two elements: the column name and a series representing the related column, as shown in the following excerpt:

```
for column_name, column in df_publications.items():
    print("\nThe name of the current column is", column_name)
    print("The content of the column is as follows:")
    print(column)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[14], line 1
----> 1 for column_name, column in df_publications.items():
      2     print("\nThe name of the current column is", column_name)
      3     print("The content of the column is as follows:")

NameError: name 'df_publications' is not defined
```

## How to store data with Pandas

Pandas makes available a specific method (i.e. `to_csv`) to its main classes, i.e. `Series` and `DataFrame`, to enable one to store them in the filesystem. In the `DataFrame` class, the method `to_csv` takes in input the file path where to store the CSV file representing the data frame as shown as follows:

```
df_publications.to_csv("03-publications.csv")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 df_publications.to_csv("03-publications.csv")

NameError: name 'df_publications' is not defined
```

However, the method `to_csv` called as shown above will store also an additional column at the beginning, i.e. that related with the index labels for each row. In order to avoid to preserve the index, it is possible to set the input named parameter `index` to `False`, as shown in the following excerpt:

```
df_publications.to_csv("03-publications_no_index.csv", index=False)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[16], line 1
----> 1 df_publications.to_csv("03-publications_no_index.csv", index=False)

NameError: name 'df_publications' is not defined
```

### 2.3.4 Main operations with DataFrame

Pandas makes available several operations for indexing, selecting, merging, joining, concatenating, and comparing data in a data frame. In the following section, we introduce two of them, but several additional operations are available in the documentation linked above.

## Querying

Pandas has several ways enabling querying a data frame and returning a selections of its rows. Among the various methods, one extremely useful is the method query, that takes in input a string representing an expression for querying the data frame and returns a new data frame compliant with the query.

The expression can be a combination of boolean expressions and comparisons, that enable to filter rows according to the values of its cells. For instance, to get all the rows that are journal articles, one can run the following query:

```
df_publications.query("type == 'journal article'")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[17], line 1
----> 1 df_publications.query("type == 'journal article'")

NameError: name 'df_publications' is not defined
```

In case we want to refer to columns with spaces, we must use the tick character (i.e. `) to enclose the name of the column. For instance, to get all the rows that have a publication date lesser than 2003, we can run the following query:

```
df_publications.query("`publication year` < 2003")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[18], line 1
----> 1 df_publications.query("`publication year` < 2003")

NameError: name 'df_publications' is not defined
```

It is also possible to combine queries by using the boolean operators `and` and `or`. For instance, to get all the journal articles published before 2003, we can run the following query:

```
df_publications.query("type == 'journal article' and `publication year` < 2003")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[19], line 1
----> 1 df_publications.query("type == 'journal article' and `publication year` < 2003")

NameError: name 'df_publications' is not defined
```

## Joining

Joining two data frames into a new one according to some common value is a crucial operation to enable to answer more complex query, such as getting all the articles published in the journal named *Current Opinion in Chemical Biology*. Indeed, the data frame about publications we have considered so fare does not have any information about the name of the venues, nor their types. However, considering the data we used in the first tutorial, we know that such information is actually included in another CSV file entirely dedicated to venues, that we can load in pandas as follows:

```
df_venues = read_csv("../01/01-venues.csv",
                     keep_default_na=False,
```

(continues on next page)

(continued from previous page)

```
                        dtype={
                            "id": "string",
                            "name": "string",
                            "type": "string"
                        })
df_venues  # draw the table in the notebook
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[20], line 1
----> 1 df_venues = read_csv("../01/01-venues.csv",
      2                      keep_default_na=False,
      3                      dtype={
      4                          "id": "string",
      5                          "name": "string",
      6                          "type": "string"
      7                      })
      8 df_venues  # draw the table in the notebook

NameError: name 'read_csv' is not defined
```

Thus, in order to run such a query, first we should ask to this new data frame which is the `id` associated to the journal named *Current Openion in Chemical Biology*, and then to ask the other data frame with publications to retrieve all the rows that have such an identifier as `publication venue`.

However, in Pandas this can be done in just one query if we join before the two data frames in a new one containing a combination of the two tables. Indeed, as you can obsever, the data frame of publications and that of venues share some values in common. Indeed, as mentioned above, the values specified in the `publication venue` column in the publications data frame recall those specified in the `id` column of the venue data frame. Thus, in principle, it is possible to join these two tables by considering that common values.

Pandas provides the function `merge` to perform such an operation. Among the various input parameters such a function can take in input, those we use in this example to join these two data frames are the data frames them self and the name of the columns in the first (named *left*) data frame and the second (named *right*) data frame to use for joining, specified by using the input named parameters `left_on` and `right_on` respectively, as shown in the following excerpt:

```
from pandas import merge

df_joined = merge(df_publications, df_venues, left_on="publication venue", right_on="id")
df_joined  # draw the table in the notebook
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[21], line 1
----> 1 from pandas import merge
      3 df_joined = merge(df_publications, df_venues, left_on="publication venue", right_
↪on="id")
      4 df_joined  # draw the table in the notebook

ModuleNotFoundError: No module named 'pandas'
```

As you can see from the data frame above, all the rows of the publications data frame (the *left* data frame of the join) have been extended using the values specified in the venues data frame (the *right* data frame of the join) mapping the values in the columns `publication year` (in *left*) and `id` (in *right*). In addition, Pandas modifies the name of the columns that have the same name in both data frames of the join – indeed the columns `type` became `type_x` (refferring to *left*) and `type_y` (referring to *right*).

Having this new data frame, the original query we wanted to run becomes pretty easy to define:

```
df_joined.query("type_y == 'journal' and name == 'Current Opinion in Chemical Biology'")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[22], line 1
----> 1 df_joined.query("type_y == 'journal' and name == 'Current Opinion in Chemical␣
↪Biology'")

NameError: name 'df_joined' is not defined
```

## 2.4 Configuring and populating a relational database

In this tutorial we introduce how to use SQLite in Python.

### 2.4.1 SQLite

SQLite is a relational database management system (RDBMS) which can be embedded into the end program and does not follow a classic client–server architecture, where the server database is independent and is actually accessed by client programs.

Python includes SQLite within its standard library - it means that you can create and modify SQLite database directly from Python code. This simplifies a lot the first approach to DBMS, as we will see in the following sections. In addition, there are already several documents on how to use SQLite in Python that are worth of reading to get more details about the features it provides. In this tutorial, we will see the main constructs used to create a database and populate it with tables.

### 2.4.2 Create a new database

First of all, we have to import functions defined in the `sqlite3` package in order to use its classes to create and execute operations on a database. The first thing we need, in particular, is to use the class `Connection`. This class is responsible to connect to a particular database defined as a file. In SQLite, any database is actually stored in one single `.db` file. A new object having class `Connection` is created by calling the function `connect`, as shown as follows:

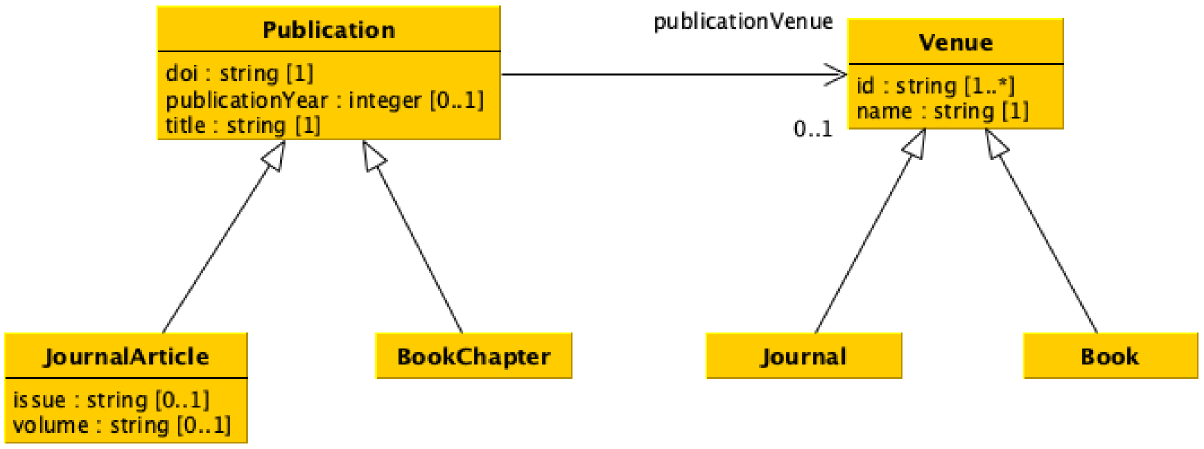
```
from sqlite3 import connect

with connect("publications.db") as con:
    # do some operation with the new connection

    con.commit()  # commit the current transaction to the database

    # when you finish, the collection will be closed automatically via 'with'
```

As you can see, the way we use a new connection to a database is similar to what we have seen with files in previous tutorials, using the instruction `with` that allows us to close the connection to the database when all the operations with it have been executed. Before closing the connection, though, it is crucial to run the method `commit` in order to commit the current transaction (i.e. the set of operations that may have changed the status of a database) to the database itself. Of course, it is possible to call the method `commit` more than once during the lifespan of a connection. In practice, you can call it every time you execute an operation that modifies the status of the database, in order to record such modification into the file system.

It is worth mentioning that the `.db` file specifed as input of the function `connect` is created if no database is available with that name in the path specified. However, if such a file already exists, it will be loaded by the connection with the information it already stores.

## 2.4.3 From a diagram to tables

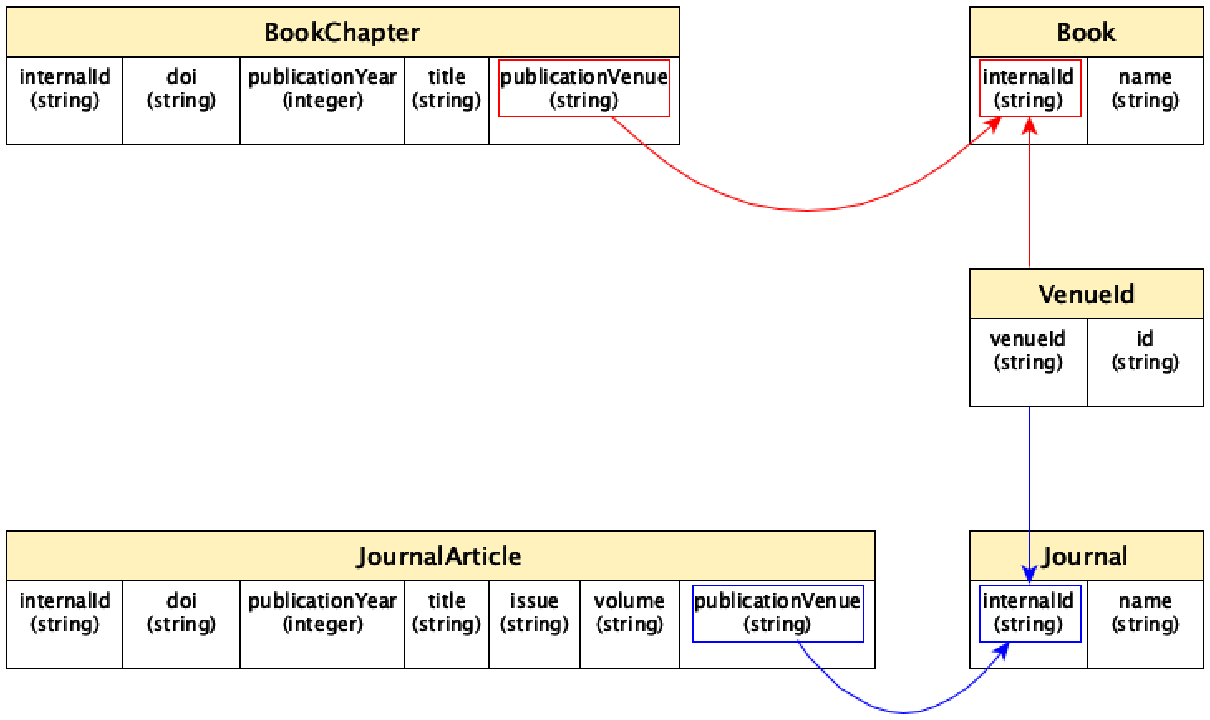
Before starting populating the database, it is necessary to create the appropriate tables that enable the description of all the entities (and related data) we need. In order to understand what to do, it is important to look at the UML diagram describing the data model introduced in a previous tutorial.



There are different strategies that we can follow to crete the tables describing a data model such as that deiscribed above. For instance, we can approach such a translation as follows:

- Create a table for each class which does not have any subclass (i.e. the most concrete classes), by using as columns all single-valued attributes and relations defined in such a class and all its superclasses. Concretely, we create four tables, i.e. for `JournalArticle` and `BookChapter` (which inherit all the attributes and relations of `Publication`), and `Journal` and `Book` (which inherit all the attributes and relations of `Venue`).
- For each of the tables above, add also a new column that enables us to specify an internal identifier we use to clearly identify all the entities of the various types. In this case, it is enough to add an additional column in each table, e.g. `internalId`. Suggestion: having an internal identifier which is globally unique in the database is the way to go.
- Keep in mind that the value of all the columns related to relations must point to an internal identifier defined in some of the tables. For instance, the column `publicationVenue` in the table `JournalArticle` will contain an internal identifier of a journal as defined in the column `internalId` of the table `Journal`.
- Create a two column table, where the first column enables the identification of an internal identifier of an entity specified in the other tables, for each multivalued attribute or relation in the diagram. In the example, only the attribute `id` of the class `Venue` is multivalued, and thus an additional table `VenueId` is created to link a `Venue` entity with one or more identifiers characterising it.

A possible translation of the UML diagram above following the rules just mentioned is shown as follows:



It is worth mentioning that this is not the only possible way to translate the original UML data mode, and other paths can be followed to this respect.

### 2.4.4 How to create and populate a table with Pandas

Pandas makes available specific methods that simplify the creation and population of database tables via `DataFrame`, and that take care also of running some database-related operations such as the `commit` shown above. The main tool to use to push a table into a SQLite database is the method `to_sql` of the class `DataFrame`. However, before seen how to use it to populate a database, let us reorganise the original data provided in CSV about publications and venues in a series of Pandas data frames recalling the tabular structures introduced in the previous section.

#### Venue-Id table

Let us start by creating the `VenueId` table. This can be done using one of the columns of the CSV document describing venues, i.e. the column `id`. Thus, we create a new sub-data frame containing only that column and we add an additional column defining the strings referring to the internal identifiers of each venue. The following code shows how to perform all these operations:

```
from pandas import read_csv, Series

venues = read_csv("../01/01-venues.csv",
                  keep_default_na=False,
                  dtype={
                      "id": "string",
                      "name": "string",
                      "type": "string"
                  })
```

(continues on next page)

(continued from previous page)

```
# This will create a new data frame starting from 'venues' one,
# and it will include only the column "id"
venues_ids = venues[["id"]]

# Generate a list of internal identifiers for the venues
venue_internal_id = []
for idx, row in venues_ids.iterrows():
    venue_internal_id.append("venue-" + str(idx))

# Add the list of venues internal identifiers as a new column
# of the data frame via the class 'Series'
venues_ids.insert(0, "venueId", Series(venue_internal_id, dtype="string"))

# Show the new data frame on screen
venues_ids
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 1
----> 1 from pandas import read_csv, Series
      3 venues = read_csv("../01/01-venues.csv",
      4                    keep_default_na=False,
      5                    dtype={
   (...)
      8                        "type": "string"
      9                    })
     11 # This will create a new data frame starting from 'venues' one,
     12 # and it will include only the column "id"

ModuleNotFoundError: No module named 'pandas'
```

In practice, after reading the CSV, we create a new data frame with only the column `id` by using the command `<data frame>[[<column name 1>, <column name 2>, ...]]`. In practice, this command will create a new sub-data frame using only the values in the columns specified.

Then, we have to define the internal identifiers for all the venues. To this end, we iterate over the rows of the new data frame and we compose a list of internal identifiers by concatenating the string `"venue-"` with the string of the values specified in the index of each row. Thus, we add that list (mediated via a `Series` of string values) into the data frame using the method `insert`, which takes in input the position where to put the new column (`0` is the first position, `1` is the second position, etc.), the column name, and the values to associate to that column.

### Tables for journals and books

With this new table, we can start to create the two additional tables for journals and books. First of all, we create two new data frames containing only entities (i.e. rows) of the same type (e.g. journals) by using the method `query`:

```
# Data frame of journals
journals = venues.query("type == 'journal'")
journals  # Showing the data frame
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[3], line 2
      1 # Data frame of journals
----> 2 journals = venues.query("type == 'journal'")
      3 journals  # Showing the data frame

NameError: name 'venues' is not defined
```

Then, for each row in the new data frame, we retrieve the associated `internalId` of each journal by looking at the table `venueId` created above. In this case, we can use the method `merge` to accomplish the task:

```
from pandas import merge

df_joined = merge(journals, venues_ids, left_on="id", right_on="id")
df_joined
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[4], line 1
----> 1 from pandas import merge
      3 df_joined = merge(journals, venues_ids, left_on="id", right_on="id")
      4 df_joined

ModuleNotFoundError: No module named 'pandas'
```

Finally, the final journal table can be defined by selecting only two columns of the last merged data frame and by modifying the column name `venueId` in `internalId`, as shown as follows:

```
journals = df_joined[["venueId", "name"]]
journals = journals.rename(columns={"venueId": "internalId"})
journals
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 journals = df_joined[["venueId", "name"]]
      2 journals = journals.rename(columns={"venueId": "internalId"})
      3 journals

NameError: name 'df_joined' is not defined
```

The rename of a column is performed with the method `rename`, that takes in input a dictonary with the named paramenter `columns`, where each key represent the old column name while the value is the new column name. The method returns a new data frame where the columns are renamed as specified.

A similar organisation can be provided also for books, by using the code specified above, customising it for handling books:

```
# Data frame of books

books = venues.query("type == 'book'")
df_joined = merge(books, venues_ids, left_on="id", right_on="id")
```

(continues on next page)

(continued from previous page)

```
books = df_joined[["venueId", "name"]]
books = books.rename(columns={"venueId": "internalId"})
books
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[6], line 3
      1 # Data frame of books
----> 3 books = venues.query("type == 'book'")
      4 df_joined = merge(books, venues_ids, left_on="id", right_on="id")
      5 books = df_joined[["venueId", "name"]]

NameError: name 'venues' is not defined
```

## Tables for publications

Similarly, all the other tables (i.e. `JournalArticles` and `BookChapters`) can be created using the same set of operations, but starting from the CSV document containing publications. First, we create a new column with all the internal identifiers for all publications, as shown as follows:

```
publications = read_csv("../01/01-publications.csv",
                        keep_default_na=False,
                        dtype={
                            "doi": "string",
                            "title": "string",
                            "publication year": "int",
                            "publication venue": "string",
                            "type": "string",
                            "issue": "string",
                            "volume": "string"
                        })

# Create a new column with internal identifiers for each publication
publication_internal_id = []
for idx, row in publications.iterrows():
    publication_internal_id.append("publication-" + str(idx))
publications.insert(0, "internalId", Series(publication_internal_id, dtype="string"))
publications
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 publications = read_csv("../01/01-publications.csv",
      2                         keep_default_na=False,
      3                         dtype={
      4                             "doi": "string",
      5                             "title": "string",
      6                             "publication year": "int",
      7                             "publication venue": "string",
      8                             "type": "string",
```

(continues on next page)

(continued from previous page)

```
      9                             "issue": "string",
     10                             "volume": "string"
     11                         })
     13 # Create a new column with internal identifiers for each publication
     14 publication_internal_id = []

NameError: name 'read_csv' is not defined
```

Then, we create the table for journal articles similarly to what we have done before:

```
# Data frame of journal articles
journal_articles = publications.query("type == 'journal article'")
df_joined = merge(journal_articles, venues_ids, left_on="publication venue", right_on="id
↪")
journal_articles = df_joined[
    ["internalId", "doi", "publication year", "title", "issue", "volume", "venueId"]]
journal_articles = journal_articles.rename(columns={
    "publication year": "publicationYear",
    "venueId": "publicationVenue"})
journal_articles
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[8], line 2
      1 # Data frame of journal articles
----> 2 journal_articles = publications.query("type == 'journal article'")
      3 df_joined = merge(journal_articles, venues_ids, left_on="publication venue",␣
↪right_on="id")
      4 journal_articles = df_joined[
      5     ["internalId", "doi", "publication year", "title", "issue", "volume",
↪"venueId"]]

NameError: name 'publications' is not defined
```

Similarly, we create the table for book chapters:

```
# Data frame of book chapters
book_chapters = publications.query("type == 'book chapter'")
df_joined = merge(book_chapters, venues_ids, left_on="publication venue", right_on="id")
book_chapters = df_joined[
    ["internalId", "doi", "publication year", "title", "venueId"]]
book_chapters = book_chapters.rename(columns={
    "publication year": "publicationYear",
    "venueId": "publicationVenue"})
book_chapters
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[9], line 2
      1 # Data frame of book chapters
----> 2 book_chapters = publications.query("type == 'book chapter'")
      3 df_joined = merge(book_chapters, venues_ids, left_on="publication venue", right_
↪on="id")
```

(continues on next page)

(continued from previous page)

```
      4 book_chapters = df_joined[
      5     ["internalId", "doi", "publication year", "title", "venueId"]]

NameError: name 'publications' is not defined
```

### Adding the tables to the database

As anticipated before, adding a table to a database is done, in Pandas, using the `DataFrame` method `to_sql`. This method takes in input two mandatory parameters (identifying the name of the table in the database and the database connection) plus a series of optional named parameters, among which the parameter `if_exists` that, when set to `"replace"`, replaces the values in an existing database table having the same name with the new data, and the parameter `index` that, when set to `False`, does not add the data frame index in the database. Thus, adding the five tables to the SQLite database created at the very beginning can be done running the following commands:

```
with connect("publications.db") as con:
    venues_ids.to_sql("VenueId", con, if_exists="replace", index=False)
    journals.to_sql("Journal", con, if_exists="replace", index=False)
    books.to_sql("Book", con, if_exists="replace", index=False)
    journal_articles.to_sql("JournalArticle", con, if_exists="replace", index=False)
    book_chapters.to_sql("BookChapter", con, if_exists="replace", index=False)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[10], line 2
      1 with connect("publications.db") as con:
----> 2     venues_ids.to_sql("VenueId", con, if_exists="replace", index=False)
      3     journals.to_sql("Journal", con, if_exists="replace", index=False)
      4     books.to_sql("Book", con, if_exists="replace", index=False)

NameError: name 'venues_ids' is not defined
```

## 2.5 Configuring and populating a graph database

In this tutorial, we show how to use RDF and Blazegraph to create a graph database using Python.

### 2.5.1 What is RDF

The Resource Description Framework (RDF) is a high-level data model (some times it is improperly called "language") based on triples *subject-predicate-object* called statements. For instance, a simple natural language sentence such as *Umberto Eco is author of The name of the rose* can be expressed through an RDF statement assigning to:

- *Umberto Eco* the role of subject;
- *is author of* the role of predicate;
- *The name of the rose* the role of object.

The main entities comprising RDF are listed as follows.

### Resources

A *resource* is an object we want to talk about, and it is identified by an IRI. IRIs are the most generic class of Internet identifiers for resources, but often HTTP URLs are used instead, which may be considered a subclass of IRIs (e.g. the URL `http://www.wikidata.org/entity/Q12807` identifies Umberto Eco in Wikidata).

### Properties

A *property* is a special type of resource since it is used to describe relation between resources, and it is identified by an IRI (e.g. the URL `http://www.wikidata.org/prop/direct/P800` identifies the property *has notable work* - which mimic the *is author of* predicate of the statement above).

### Statements

*Statements* enable one to assert properties between resources. Each statement is a triple subject-predicate-object, where the subject is a resource, the predicate is a property, and the object is either a resource or a literal (i.e. a string).

There are different notations that can be used to represent statements in RDF in plain text files. The simplest (and most verbose) one is called N-Triples. It allows to define statements according to the following syntax:

```
# 1) statement with a resource as an object
<IRI subject> <IRI predicate> <IRI object> .

# 2) statement with a literal as an object
<IRI subject> <IRI predicate> "literal value"^^<IRI type of value> .
```

Type (1) statements must be used to state relationships between resources, while type (2) statements are generally used to associate attributes to a specific resource (the IRI defining the type of value is not specified for generic literals, i.e. strings). For instance, in Wikidata, the exemplar sentence above (*Umberto Eco is author of The name of the rose*) is defined by three distict RDF statements:

```
<http://www.wikidata.org/entity/Q12807> <http://www.w3.org/2000/01/rdf-schema#label>
↪"Umberto Eco" .

<http://www.wikidata.org/entity/Q172850> <http://www.w3.org/2000/01/rdf-schema#label>
↪"The Name of the Rose" .

<http://www.wikidata.org/entity/Q12807> <http://www.wikidata.org/prop/direct/P800>
↪<http://www.wikidata.org/entity/Q172850> .
```

Actually, the relation described by the natural language sentence is defined by the third RDF statement above. However, two additional statements have been added to associate the strings representing the name of the resources referring to *Umberto Eco* and *The name of the rose*. Be aware: literals (i.e. simple values) cannot be subjects in any statement.

### A special property

While all the properties you can use in your statements as predicates can be defined in several distinct vocabularies (the Wikidata data model, schema.org data model, etc.), RDF defines a special property that is used to associate a resource to its intended type (e.g. another resource representing a class of resources). The IRI of this property is `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`. For instance, we can use this property to assign the appropriate type of object to the two entities defined in the excerpt above, i.e. that referencing to *Umberto Eco* and *The name of the rose*, as shown as follows:

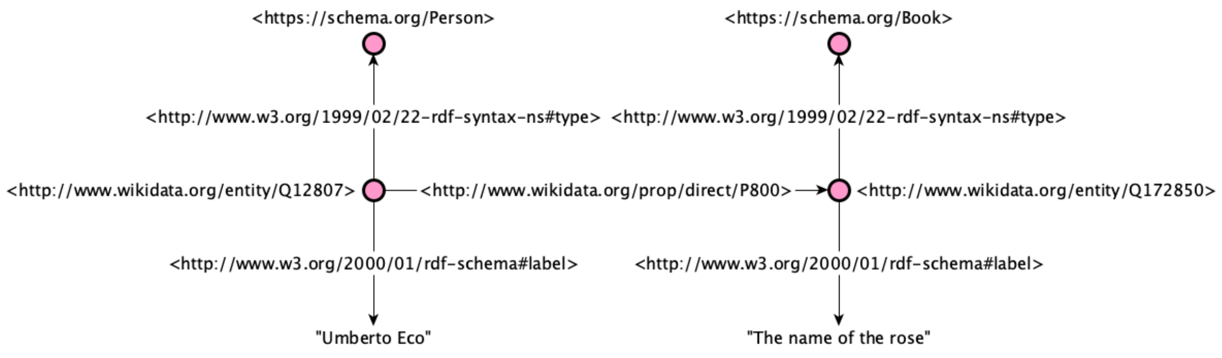
```
<http://www.wikidata.org/entity/Q12807> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
↪ <https://schema.org/Person> .

<http://www.wikidata.org/entity/Q172850> <http://www.w3.org/1999/02/22-rdf-syntax-ns
↪#type> <https://schema.org/Book> .
```

In the example above, we reuse two existing classes of resources included in schema.org for people and books. It is worth mentioning that an existing resource can be associated via `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` to one or more types, if they apply.

### RDF Graphs

An *RDF Graph* is a set of RDF statements. For instance, a file that contains RDF statements represents an RDF graph, and IRIs contained in different graph actually refer to the same resource.

We talk about graphs in this context because all the RDF statements, and the resources they include, actually defined a direct graph structure, where the direct edges are labelled with the predicates of the statements and the subjects and objects are nodes linked through such edges. For instance, the diagram below represents all the RDF statements introduced above using a visual graph.


<https://schema.org/Person>
<https://schema.org/Book>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.wikidata.org/entity/Q12807>
<http://www.wikidata.org/prop/direct/P800>
<http://www.wikidata.org/entity/Q172850>
<http://www.w3.org/2000/01/rdf-schema#label>
<http://www.w3.org/2000/01/rdf-schema#label>
"Umberto Eco"
"The name of the rose"

### Triplestores

A *triplestore* is a database built for storing and retrieving RDF statements, and can contain one or more RDF graphs.

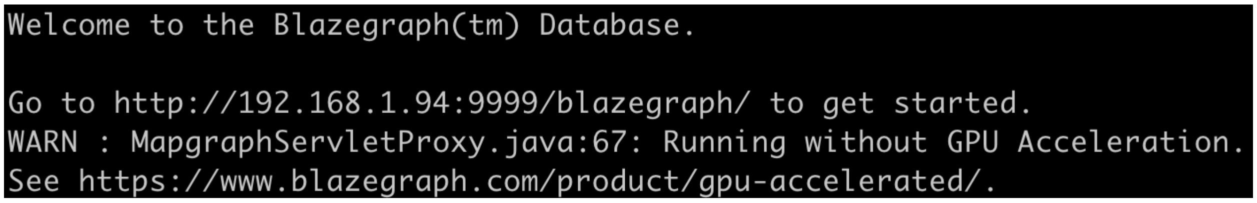
### 2.5.2 Blazegraph, a database for RDF data

Blazegraph DB is a ultra high-performance graph database supporting RDF/SPARQL APIs (thus, it is a triplestore). It supports up to 50 Billion edges on a single machine. Its code is entirely open source and available on GitHub.

Running this database as a server application is very simple. One has just to download the .jar application, put it in a directory, and run it from a shell as follows:

```
java -server -Xmx1g -jar blazegraph.jar
```

You need at least Java 9 installed in your system. If you do not have it, you can easily dowload and install it from the Java webpage. As you can see from the output of the command above, the database will be exposed via HTTP at a specific IP address:



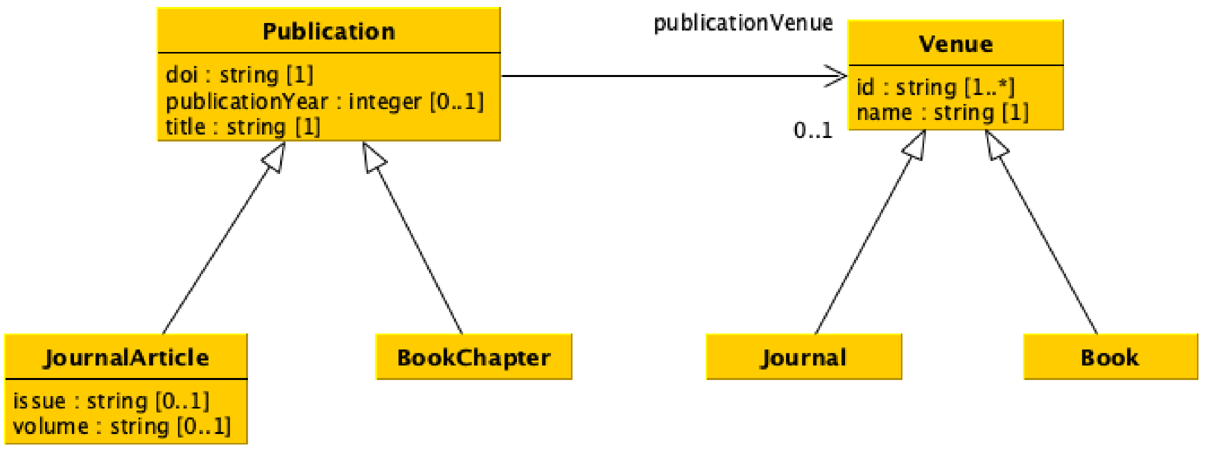
```
Welcome to the Blazegraph(tm) Database.

Go to http://192.168.1.94:9999/blazegraph/ to get started.
WARN : MapgraphServletProxy.java:67: Running without GPU Acceleration.
See https://www.blazegraph.com/product/gpu-accelerated/.
```

However, from your local machine, you can always contact it at the following URL:

```
http://127.0.0.1:9999/blazegraph/
```

### 2.5.3 From a diagram to a graph

As you can see, the UML diagram introduced in the previous lecture, which I recall below, is already organised as a (directed) graph. Thus, translating such a data model into an RDF graph database is kind of straightforward.



The important thing to decide, in this context, is to clarify what are the names (i.e. the URLs) of the classes and properties to use to represent the data compliant with the data model. In particular:

- supposing that each resource will be assigned to at least one of the types defined in the data model, we need to identify the names of all the most concrete classes (e.g. `JournalArticle`, `BookChapter`, `Journal`, `Book`);
- each attribute of each UML class will be represented by a distinct RDF property which will be involved in statements where the subjects are always resources of the class in consideration and the objects are simple literals (i.e. values). Of course, we have to identify the names of these properties (i.e. the URLs);

- each relation starting from an UML class and ending in another UML class will be represented by a distinct RDF property which will be involved in statements where the subjects are always resources of the source class while the objects are resources of the target class. Of course, we have to identify the names of these properties (i.e. the URLs);
- please, bear in mind that all attributes and relations defined in a class are inherited (i.e. can be used by) all its subclasses.

You can choose to reuse existing classes and properties (e.g. as defined in schema.org) or create your own. In the latter case, you have to remind to use an URL you are in control of (e.g. your website or GitHub repository). For instance, a possible pattern for defining your own name for the class `Book` could be `https://<your website>/Book` (e.g. `https://essepuntato.it/Book`). Of course, there are strategies and guidelines that should be used to implement appropriately data model in RDF-compliant languages. However these are out of the scope of the present course (and will be clarified in other courses).

The name of all the classes and properties I will use in the examples in this tutorial are as follows:

- UML class `JournalArticle`: `https://schema.org/ScholarlyArticle`;
- UML class `BookChapter`: `https://schema.org/Chapter`;
- UML class `Journal`: `https://schema.org/Periodical`;
- UML class `Book`: `https://schema.org/Book`;
- UML attribute `doi` of class `Publication`: `https://schema.org/identifier`;
- UML attribute `publicationYear` of class `Publication`: `https://schema.org/datePublished`;
- UML attribute `title` of class `Publication`: `https://schema.org/name`;
- UML attribute `issue` of class `JournalArticle`: `https://schema.org/issueNumber`;
- UML attribute `volume` of class `JournalArticle`: `https://schema.org/volumeNumber`;
- UML attribute `id` of class `Venue`: `https://schema.org/identifier`;
- UML attribute `name` of class `Venue`: `https://schema.org/name`;
- UML relation `publicationVenue` of class `Publication`: `https://schema.org/isPartOf`.

### 2.5.4 Using RDF in Python

The library `rdflib` provides classes and methods that allow one to create RDF graphs and populating them with RDF statements. It can be installed using the `pip` command as follows:

```
pip install rdflib
```

The class `Graph` is used to create an (initially empty) RDF graph, as shown as follows:

```
from rdflib import Graph

my_graph = Graph()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from rdflib import Graph
      3 my_graph = Graph()
```

(continues on next page)

(continued from previous page)

```

ModuleNotFoundError: No module named 'rdflib'
```

All the resources (including the properties) are defined using the class URIRef. The constructor of this class takes in input a string representing the IRI (or URL) of the resource in consideration. For instance, the code below shows all the resources mentioned above, i.e. those referring to classes, attributes and relations:

```
from rdflib import URIRef

# classes of resources
JournalArticle = URIRef("https://schema.org/ScholarlyArticle")
BookChapter = URIRef("https://schema.org/Chapter")
Journal = URIRef("https://schema.org/Periodical")
Book = URIRef("https://schema.org/Book")

# attributes related to classes
doi = URIRef("https://schema.org/identifier")
publicationYear = URIRef("https://schema.org/datePublished")
title = URIRef("https://schema.org/name")
issue = URIRef("https://schema.org/issueNumber")
volume = URIRef("https://schema.org/volumeNumber")
identifier = URIRef("https://schema.org/identifier")
name = URIRef("https://schema.org/name")

# relations among classes
publicationVenue = URIRef("https://schema.org/isPartOf")
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[2], line 1
----> 1 from rdflib import URIRef
      3 # classes of resources
      4 JournalArticle = URIRef("https://schema.org/ScholarlyArticle")

ModuleNotFoundError: No module named 'rdflib'
```

Instead, literals (i.e. value to specify as objects of RDF statements) can be created using the class Literal. The constructor of this class takes in input a value (of any basic type: it can be a string, an integer, etc.) and create the related literal object in RDF, as shown in the next excerpt:

```
from rdflib import Literal

a_string = Literal("a string with this value")
a_number = Literal(42)
a_boolean = Literal(True)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[3], line 1
----> 1 from rdflib import Literal
      3 a_string = Literal("a string with this value")
      4 a_number = Literal(42)
```

(continues on next page)

(continued from previous page)

```
ModuleNotFoundError: No module named 'rdflib'
```

Using these classes it is possible to create all the Python objects necessary to create statements describing all the data to be pushed into an RDF graph. We need to use the method add to add a new RDF statement to a graph. Such method takes in input a tuple of three elements defining the subject (an `URIRef`), the predicate (another `URIRef`) and the object (either an `URIRef` or a `Literal`) of the statements.

The following code show how to populate the RDF graph defining using the data obtained by processing the two CSV documents presented in previous tutorials. i.e. that of the publications and that of the venues. For instance, all the venues are created using the following code:

```
from pandas import read_csv, Series
from rdflib import RDF

# This is the string defining the base URL used to defined
# the URLs of all the resources created from the data
base_url = "https://comp-data.github.io/res/"

venues = read_csv("../01/01-venues.csv",
                  keep_default_na=False,
                  dtype={
                      "id": "string",
                      "name": "string",
                      "type": "string"
                  })

venue_internal_id = {}
for idx, row in venues.iterrows():
    local_id = "venue-" + str(idx)

    # The shape of the new resources that are venues is
    # 'https://comp-data.github.io/res/venue-<integer>'
    subj = URIRef(base_url + local_id)

    # We put the new venue resources created here, to use them
    # when creating publications
    venue_internal_id[row["id"]] = subj

    if row["type"] == "journal":
        # RDF.type is the URIRef already provided by rdflib of the property
        # 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
        my_graph.add((subj, RDF.type, Journal))
    else:
        my_graph.add((subj, RDF.type, Book))

    my_graph.add((subj, name, Literal(row["name"])))
    my_graph.add((subj, identifier, Literal(row["id"])))
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[4], line 1
```

(continues on next page)

(continued from previous page)

```
----> 1 from pandas import read_csv, Series
      2 from rdflib import RDF
      4 # This is the string defining the base URL used to defined
      5 # the URLs of all the resources created from the data

ModuleNotFoundError: No module named 'pandas'
```

As you can see, all the RDF triples have been added to the graph, that currently contain the following number of distinct triples (which is coincident with the number of cells in the original table):

```
print("-- Number of triples added to the graph after processing the venues")
print(len(my_graph))
```

```
-- Number of triples added to the graph after processing the venues
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 2
      1 print("-- Number of triples added to the graph after processing the venues")
----> 2 print(len(my_graph))

NameError: name 'my_graph' is not defined
```

The same approach can be used to add information about the publications, as shown as follows:

```
publications = read_csv("../01/01-publications.csv",
                        keep_default_na=False,
                        dtype={
                            "doi": "string",
                            "title": "string",
                            "publication year": "int",
                            "publication venue": "string",
                            "type": "string",
                            "issue": "string",
                            "volume": "string"
                        })

for idx, row in publications.iterrows():
    local_id = "publication-" + str(idx)

    # The shape of the new resources that are publications is
    # 'https://comp-data.github.io/res/publication-<integer>'
    subj = URIRef(base_url + local_id)

    if row["type"] == "journal article":
        my_graph.add((subj, RDF.type, JournalArticle))

        # These two statements applies only to journal articles
        my_graph.add((subj, issue, Literal(row["issue"])))
        my_graph.add((subj, volume, Literal(row["volume"])))
    else:
        my_graph.add((subj, RDF.type, BookChapter))
```

(continues on next page)

(continued from previous page)

```
    my_graph.add((subj, name, Literal(row["title"])))
    my_graph.add((subj, identifier, Literal(row["doi"])))

    # The original value here has been casted to string since the Date type
    # in schema.org ('https://schema.org/Date') is actually a string-like value
    my_graph.add((subj, publicationYear, Literal(str(row["publication year"]))))

    # The URL of the related publication venue is taken from the previous
    # dictionary defined when processing the venues
    my_graph.add((subj, publicationVenue, venue_internal_id[row["publication venue"]]))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 publications = read_csv("../01/01-publications.csv",
      2                         keep_default_na=False,
      3                         dtype={
      4                             "doi": "string",
      5                             "title": "string",
      6                             "publication year": "int",
      7                             "publication venue": "string",
      8                             "type": "string",
      9                             "issue": "string",
     10                             "volume": "string"
     11                         })
     13 for idx, row in publications.iterrows():
     14     local_id = "publication-" + str(idx)

NameError: name 'read_csv' is not defined
```

After the addition of this new statements, the number of total RDF triples added to the graph is equal to all the cells in the venue CSV plus all the non-empty cells in the publication CSV:

```
print("-- Number of triples added to the graph after processing venues and publications")
print(len(my_graph))
```

```
-- Number of triples added to the graph after processing venues and publications
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[7], line 2
      1 print("-- Number of triples added to the graph after processing venues and␣
↪publications")
----> 2 print(len(my_graph))

NameError: name 'my_graph' is not defined
```

It is worth mentioning that we should not map in RDF cells in the original table that do not contain any value. Thus, if for instance there is an `issue` cell in the publication CSV which is empty (i.e. no information about the issue have been specified), you should not create any RDF statement mapping such a non-information.

### 2.5.5 How to create and populate a graph database with Python

Once we have created our graph with all the triples we need, we can upload persistently the graph on our triplestore. In order to do that, we have to create an instance of the class `SPARQLUpdateStore`, which acts as a proxy to interact with the triplestore. The important thing is to open the connection with the store passing, as input, a tuple of two strings with the same URLs, defining the SPARQL endpoint of the triplestore where to upload the data.

Then, we can upload triple by triple using a for-each iteration over the list of RDF statements obtained by using the method `triples` of the class `Graph`, passing as input a tuple with three `None` values, as shown as follows:

```python
from rdflib.plugins.stores.sparqlstore import SPARQLUpdateStore

store = SPARQLUpdateStore()

# The URL of the SPARQL endpoint is the same URL of the Blazegraph
# instance + '/sparql'
endpoint = 'http://127.0.0.1:9999/blazegraph/sparql'

# It opens the connection with the SPARQL endpoint instance
store.open((endpoint, endpoint))

for triple in my_graph.triples((None, None, None)):
   store.add(triple)

# Once finished, remeber to close the connection
store.close()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[8], line 1
----> 1 from rdflib.plugins.stores.sparqlstore import SPARQLUpdateStore
      3 store = SPARQLUpdateStore()
      5 # The URL of the SPARQL endpoint is the same URL of the Blazegraph
      6 # instance + '/sparql'

ModuleNotFoundError: No module named 'rdflib'
```

## 2.6 Interacting with databases using Pandas

In this tutorial, we show how to use Pandas data frames to interact with SQL-based and graph-based databases.

### 2.6.1 Data available in different sources

Often, when you have to deal with and reuse existing data, the answer to a query can be possible only by combining data available in different databases. In addition, such databases can expose their data using different technologies (e.g. an SQLite database and an RDF triplestore). Thus, it is important to have a smooth method that allows one to take data from different sources, to expose these data according to a similar interface, and finally to make some additional operation on these data that, in principle, can be seen as coming from a unique abstract source.

Pandas, thanks to its standard library and additional plugins developed for it, enables us to use it as a proxy model for getting and comparing data coming from different sources (and even different formats). A few tutorials ago, indeed, we have seen how to read data stored as CSV documents using Pandas. We can use similar functions to read a result of a query sent to a database as it is a source of information. In this tutorial, we see how to do it with SQLite and Blazegraph, i.e. the two databases used in the previous tutorials.

### 2.6.2 Reading data from SQLite

Pandas makes available the method `read_sql` which enables us, among the other things, to query an SQL-based database using an SQL query and to expose the answer returned as a classic Pandas data frame. This function takes in input two mandatory parameters that are the SQL query to execute on the database and the connection to it, and returns a data frame built on the data and the parameter specified in the SQL query. For instance, the following code takes the title of all the journal articles included in the table `JournalArticle`:

```
from sqlite3 import connect
from pandas import read_sql

with connect("../04/publications.db") as con:
    query = "SELECT title FROM JournalArticle"
    df_sql = read_sql(query, con)

df_sql  # show the content of the result of the query
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 2
      1 from sqlite3 import connect
----> 2 from pandas import read_sql
      4 with connect("../04/publications.db") as con:
      5     query = "SELECT title FROM JournalArticle"

ModuleNotFoundError: No module named 'pandas'
```

It is worth mentioning that, to enable the correct definition of the results of the query into a data frame, it is always better first to create all the necessary data frames within the `with` clause, and then start to work on them "offline", once the connection to the database has been closed. Otherwise, you could observe some unexpected behaviours.

Finally, it is worth mentioning that the data type used in the database are converted into the appropriate data type in Pandas. Thus, if a column has been defined as containing integers in the database, we get back the same data type for the column in the data frame. This is clear when we try to retrieve, for instance, an entire table from the SQLite database:

```
with connect("../04/publications.db") as con:
    query = "SELECT * FROM JournalArticle"
    df_journal_article_sql = read_sql(query, con)

# Show the series of the column 'publicationYear', which as 'dtype'
# specifies 'int64', as expected
df_journal_article_sql["publicationYear"]
```

```
---------------------------------------------------------------------------
OperationalError                          Traceback (most recent call last)
Cell In[2], line 1
----> 1 with connect("../04/publications.db") as con:
      2     query = "SELECT * FROM JournalArticle"
      3     df_journal_article_sql = read_sql(query, con)
```

(continues on next page)

(continued from previous page)

```
OperationalError: unable to open database file
```

### 2.6.3 Reading data from Blazegraph

Even if Pandas does not make available any reading method to interact with RDF triplestores, some developers has implemented a facility that permits us to interact directly with a SPARQL endpoint provided by an RDF triplestore such as Blazegraph, i.e. the library `sparql_dataframe`. This library is a wrapper for a SPARQL query and shows the answer to such a query as a Pandas data frame. We can install the library using the usual command:

```
pip install sparql_dataframe
```

The function `get` is called to perform such an operation, and it takes in input three parameters: the URL of the SPARQL endpoint to contact, the query to execute, and a boolean specifying if to contact the SPARQL endpoint using the POST HTTP method (strongly suggested, otherwise it could not work correctly). An example of execution of such a function is shown in the following excerpt:

```
from sparql_dataframe import get

endpoint = "http://127.0.0.1:9999/blazegraph/sparql"
query = """
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <https://schema.org/>

SELECT ?journal_article ?title
WHERE {
    ?journal_article rdf:type schema:ScholarlyArticle .
    ?journal_article schema:name ?title .
}
"""
df_sparql = get(endpoint, query, True)
df_sparql
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[3], line 1
----> 1 from sparql_dataframe import get
      3 endpoint = "http://127.0.0.1:9999/blazegraph/sparql"
      4 query = """
      5 PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
      6 PREFIX schema: <https://schema.org/>
   (...)
     12 }
     13 """

ModuleNotFoundError: No module named 'sparql_dataframe'
```

Due to the implementation of the `get` function in the `sparql_dataframe` package, though, the values returned by running the SPARQL query will be inferred automatically by looking at all the values of a certain column. Thus, if one wants to change the data type of the values associated to a particular column, one has to cast the column on purpose and the reassigning the column to the data frame. For instance, let us build a query that takes information of all the publications available in the triplestore:

```
publication_query = """
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <https://schema.org/>

SELECT ?internalId ?doi ?publicationYear ?title ?issue ?volume ?publicationVenue
WHERE {
    VALUES ?type {
        schema:ScholarlyArticle
        schema:Chapter
    }

    ?internalId rdf:type ?type .
    ?internalId schema:identifier ?doi .
    ?internalId schema:datePublished ?publicationYear .
    ?internalId schema:name ?title .
    ?internalId schema:isPartOf ?publicationVenue .

    OPTIONAL {
        ?internalId schema:issueNumber ?issue .
        ?internalId schema:volumeNumber ?volume .
    }
}
"""

df_publications_sparql = get(endpoint, publication_query, True)
df_publications_sparql
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 25
      1 publication_query = """
      2 PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
      3 PREFIX schema: <https://schema.org/>
   (...)
     22 }
     23 """
---> 25 df_publications_sparql = get(endpoint, publication_query, True)
     26 df_publications_sparql

NameError: name 'get' is not defined
```

It is worth mentioning that the optional group in the SPARQL query (`OPTIONAL { ... }`) is used to allow information to be added to the solution if it is available, otherwise the related variables will be left empty.

### 2.6.4 Fixing some issues

As you can observed from the result of the previous query, the data frame created contains some basic information depicted by the variable names chosen, that are specified in the query itself for being equal to those returned in the last SQL query done above.

However, one unexpected behaviour is the way the columns `issue` and `volume` is handled. To see this, we use the attribute `dtypes` of our data frame to see how things are handled:

```
df_publications_sparql.dtypes
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 df_publications_sparql.dtypes

NameError: name 'df_publications_sparql' is not defined
```

As you can see, the two columns mentioned above have been assigned with a float data type, which has been inferred by Pandas by looking at the values of these two columns. In order to change it into an appropriate kind of value, e.g. a string, we have to overwrite the data type of the entire data frame (using the method `astype` that takes in input the new data type) and/or the data type of specific columns. For doing the last operation, we have to reassign the columns with the new types to the data frame using the following syntax:

```
<data frame>[<column name>] = <data frame>[<column name>].astype(<new data type>)
```

For instance, to reassign the columns `issue` and `volume` to the type `"string"`, we can run the following commands:

```
df_publications_sparql["issue"] = df_publications_sparql["issue"].astype("string")
df_publications_sparql["volume"] = df_publications_sparql["volume"].astype("string")

df_publications_sparql.dtypes
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 df_publications_sparql["issue"] = df_publications_sparql["issue"].astype("string
↪")
      2 df_publications_sparql["volume"] = df_publications_sparql["volume"].astype(
↪"string")
      4 df_publications_sparql.dtypes

NameError: name 'df_publications_sparql' is not defined
```

Similarly, if you want to replace the `NaN` values associated to the same two columns when no value is available, you can use the data frame method `fillna`, which enables one to replace all `NaN` in the data frame with a value of your choice passed as input:

```
df_publications_sparql = df_publications_sparql.fillna("")

df_publications_sparql
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
Cell In[7], line 1
----> 1 df_publications_sparql = df_publications_sparql.fillna("")
      3 df_publications_sparql

NameError: name 'df_publications_sparql' is not defined
```

Of course, this allowed us to remove all `NaN` values. However, if you look at the table and in particular to the columns `issue` and `volume`, you can see something that is still a bit in these two columns.

Indeed, the two strings defining issues and volumes associated with an article are, actually, the mere cast of the floating value into a string and, as such, they contain the `.0` part of the float that we need to remove. Since the same pattern is repeated in all the values of these two columns, we could apply a similar operation to all their values to clean them up. For doing that, we use the method `apply` of the class `Series`, which allows us to apply an input function to all the values of a column and to store, in each value, what such a function returns.

A function that would allow us perform such an operation is the following one:

```
def remove_dotzero(s):
    return s.replace(".0", "")
```

The function above takes in input a string (i.e. the value of a cell) and remove the string `".0"` from there, if present. Thus, passing this function to the method `apply` of each column and then to assign the modified column back to the data frame will fix the issue, as shown as follows:

```
df_publications_sparql["issue"] = df_publications_sparql["issue"].apply(remove_dotzero)
df_publications_sparql["volume"] = df_publications_sparql["volume"].apply(remove_dotzero)

df_publications_sparql
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 df_publications_sparql["issue"] = df_publications_sparql["issue"].apply(remove_
↪dotzero)
      2 df_publications_sparql["volume"] = df_publications_sparql["volume"].apply(remove_
↪dotzero)
      4 df_publications_sparql

NameError: name 'df_publications_sparql' is not defined
```

### 2.6.5 Combining data

In the previous section, we have introced how to obtain data from existing databases and how to manipulate them using Pandas. However, in real case scenarios, an answer to a certain query can arrive only from mixing partial data from two distinct databases. Thus, it is important to implement some mechanisms to mash data up together, clean them if needed (e.g. removing duplicates), and to return them in a certain order (e.g. alphabetically). Of course, Pandas can be used to perform all these operations.

Suppose that we want to find, by querying all the databases, all the titles and year of publication of all publications they contain (independently from their type), ordered from the oldest one to the newest one. To simplify the job for this tutorial, we could consider the two data frames computed before, i.e. `df_journal_article_sql` and `df_publications_sparql`, as the two coming from two different databases.

First of all, we need something that allows us to concat two or more data frames together. However, in order to do that, it is important that, first of all, all the data frames to contact share the same columns. Thus, if necessary, it is important to rename the columns as we have seen in a previous tutorial. In this case, instead, we have already created the data frames with the same column names and, as such, we can proceed with the concat operation, i.e. obtaining a new data frame by concatenating the rows contained in both the data frames.

This operation is implemented by the function `concat`, that takes in input a list of data frames and return a new data frame with all the rows concatenated. In addition, it can also take in input the named parameter `ignore_index` that, if set to `True`, will reindex all the rows from the beginning in the new data frame, as shown in the following code:

```
from pandas import concat

df_union = concat([df_journal_article_sql, df_publications_sparql], ignore_index=True)
df_union
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[10], line 1
----> 1 from pandas import concat
      3 df_union = concat([df_journal_article_sql, df_publications_sparql], ignore_
↪index=True)
      4 df_union

ModuleNotFoundError: No module named 'pandas'
```

After having obtained a new data frame concatenating the other two, we need to filter out duplicates. Once can follow different approaches for doing so. In this context, we will use the DOIs of the publications to perform the filtering.

A DOI (Digital Object Identifier) is a persistent identifier used to identify publications uniquely worldwide. Thus, if a publication is included in two distinct databases, it should have the same DOI despite the local identifiers the databases may use.

Once this aspect is clear, we can perform a removal of rows using the method `drop_duplicates` of the class `DataFrame`. This method allows one to specify the optional named parameter `subset` with the list of columns names to use to identify similar rows. If such a named parameter is not specified, only identical rows (those having all the values in full match) are removed from data frame. Thus, we can perform the removal of duplicates as follows:

```
df_union_no_duplicates = df_union.drop_duplicates(subset=["doi"])
df_union_no_duplicates
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[11], line 1
----> 1 df_union_no_duplicates = df_union.drop_duplicates(subset=["doi"])
      2 df_union_no_duplicates

NameError: name 'df_union' is not defined
```

Then, we have finally to sort rows in ascending order considering the publication year, and then to return just the columns publication year and title and year of publication of each row. In Pandas, the sorting can be performed using the method `sort_values` of the class `DataFrame`, that takes in input the name of the column to use to perform the sorting, as shown as follows:

```
df_union_no_duplicates_sorted = df_union_no_duplicates.sort_values("publicationYear")
df_union_no_duplicates_sorted
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 df_union_no_duplicates_sorted = df_union_no_duplicates.sort_values(
↪"publicationYear")
      2 df_union_no_duplicates_sorted

NameError: name 'df_union_no_duplicates' is not defined
```

Finally, to select a sub-data frame, we use the approach adopted in past tutorial, by creating a new data frame selecting only some of the columns of another one:

```
df_final = df_union_no_duplicates_sorted[["title", "publicationYear"]]
df_final
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 df_final = df_union_no_duplicates_sorted[["title", "publicationYear"]]
      2 df_final

NameError: name 'df_union_no_duplicates_sorted' is not defined
```

## 2.7 Descriptive statistics and graphs about data using Pandas

In this tutorial, we show how to use Pandas to calculate basic statistics of a dataset and show figures using automatically-generated graphs.

### 2.7.1 What data are about

When you receive a new dataset (such as the one included in this tutorial), the first you have to do is to analyse it to understand what its data are about, how they have been organised, what is the type of each column, and whether there are any *null* object included in it (e.g. empty cells). In a previous tutorial, we have used an input parameter specified on the function `read_csv` (i.e. `keep_default_na` set to `False`) to rewrite empty cell values as empty strings (i.e. `""`). However, by doing so, we may miss some relevant information about the dataset that we should know from the beginning. Let us see it practically with an example, using the `DataFrame` method `info` that enables us to have a summary of the data frame:

```
from pandas import read_csv

publications = read_csv("assets/data/07-publications.csv", keep_default_na=False)
publications.info()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 1
----> 1 from pandas import read_csv
      3 publications = read_csv("assets/data/07-publications.csv", keep_default_na=False)
      4 publications.info()
```

(continues on next page)

(continued from previous page)

```

ModuleNotFoundError: No module named 'pandas'
```

As you can see from the text printed on screen, this seems a perfect dataset: 995 rows, entries organised in seven columns and each cell contains only non-null values. However, is it really the case? The problem here is that, by avoiding to use the default mechanism to assign empty cells, the systems does not recognize them as empty, but rather containing something (e.g. the empty string `""`) which indeed does not contain any charater but, still, is a value associated with a cell.

Thus, as a suggestion, when you approach for the very first time a dataset using Pandas, leave the system use its own favourite ways to handle situations (such as empty cells) and observe what this may mean. The following code show the same description reported above leaving the system to handle empty cells as it prefers:

```
publications = read_csv("assets/data/07-publications.csv")
publications.info()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 publications = read_csv("assets/data/07-publications.csv")
      2 publications.info()

NameError: name 'read_csv' is not defined
```

As you can see, now the description is slightly different. Indeed, the only columns having always a non-null value specified are `doi`, `publication year` and `type`, while the other columns have, somewhere, some cell left unspecified - which is reasonable, if you think about it. For instance, a book chapter does not have any issue or volume and, thus, the related cells must be empty in such a record.

After you got an idea of what kind of columns are included in the data frame, we can move on asking for more information about the values of each column. For retrieving this information, we use the method `describe`. If you want to obtain an overall view contained in a new data frame describing the one on which the method is called, the suggestion is to call such a method using the optional input named parameter `include` set to `"all"`, as shown in the following excerpt:

```
publications.describe(include="all")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 publications.describe(include="all")

NameError: name 'publications' is not defined
```

The data frame above provide a pletora of different statistics about each single column of the original data, some of them apply to a certain columns while other do not. For instance, all the statistics about number manipulations (`mean`, `std`, `min`, etc.) do not apply to strings, and thus a `NaN` is returned in these cases.

Some of these statistics are very useful, and allow you to understand something about the data without looking at all of them. For instance:

- there are two publications sharing the same title, that is “Transformation toughening”;
- some of the publications (6) do not have a title associated;

- while all the publications have a type specified, overall these types come from 5 distinct values only (they seem to highlight descriptive categories);
- the oldest publication was published in 1886 while the newest in 2012.

Looking at these information, one could be curious about how can exist two publications with the same title. To get more information about it, we can run a query catching all the publications having in common the title "Transformation toughening":

```
publications.query('title == "Transformation toughening"')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 publications.query('title == "Transformation toughening"')

NameError: name 'publications' is not defined
```

Indeed, these two publications may seem the same one, since they share all the other data, except the DOI. Thus, one possibility would be to use the DOI resolver (https://doi.org) to see to which kind of entities these DOIs (i.e. https://doi.org/10.1007/bf00809059 and https://doi.org/10.1007/bf00809057) actually refer to. Once seen them, are they the same entity? Hint: look at the subtitle and the number of the pages…

In addition to the statistics provided in the previous data frame, other statistics can be calculated using the data in the columns of the data frame. For instance, another important statistics for numeric values is the median:

```
print("-- Median value of the publication years in the data")
print(publications["publication year"].median())
```

```
-- Median value of the publication years in the data
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 2
      1 print("-- Median value of the publication years in the data")
----> 2 print(publications["publication year"].median())

NameError: name 'publications' is not defined
```

The `Series` method `median` calculates such an important statistics that enables one to identify the average value of a series of numbers partially-limitating the effect of the outliers in the series. Indeed, the average calculated in the data frame above is lower than the median, since it is affected by some very low publication dates (e.g. 1886), which do not affect at all the median instead.

Another useful operation one can run on series is provided by the method `unique`. This method is useful to identify what are all the unique values in a series that, in principle, can contain several items. For instance, that can be used to identify the categories in the column `type`:

```
print("-- Categories describing types of publications")
publications["type"].unique()
```

```
-- Categories describing types of publications
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[6], line 2
      1 print("-- Categories describing types of publications")
----> 2 publications["type"].unique()

NameError: name 'publications' is not defined
```

### 2.7.2 Drawing data

In addition to generate data frames with descriptive statistics, Pandas makes available also methods to draw data in simple graphs, such as line charts, bar charts, etc. For instance, taking as example the types of publications described in the previous code, we could be interested in understanding how many publications of each type are included in the data. For doing so, we first have to retrieve such number for each type using the `Series` method `value_counts`, shown as follows:

```
type_count = publications["type"].value_counts()
type_count
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 type_count = publications["type"].value_counts()
      2 type_count

NameError: name 'publications' is not defined
```

The method `value_counts` applied to a series of strings returns the number of times each string appears in the series, where the unique strings of the series become the index of the new series. These kinds of series can be plotted as a bar chart easily, where the index labels are the categories shown in the x-axis, while the number of times each string is represented in the original series is the value highlighte in the y-axis. This diagram is plotted using the method `plot`, specifying the optional input named parameter `kind` set to `"bar"`, as shown as follows:

```
type_count.plot(kind="bar")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 type_count.plot(kind="bar")

NameError: name 'type_count' is not defined
```

We can use a similar approach to understand, for instance, what are the top ten venues considering all the publication in the dataset. In this case, we use again the method `value_counts` and then we select the first 10 rows, as shown as follows:

```
best_venues = publications["publication venue"].value_counts()[:10]
best_venues
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
Cell In[9], line 1
----> 1 best_venues = publications["publication venue"].value_counts()[:10]
      2 best_venues

NameError: name 'publications' is not defined
```

Again, as before, we plot it as a *horizontal* bar chart using the same command shown in the previous example, but setting the parameter `kind` to `"barh"`:

```
best_venues.plot(kind="barh")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 best_venues.plot(kind="barh")

NameError: name 'best_venues' is not defined
```

We can use a different plot when we deal with *time* series, i.e. series of data that are organised temporally (for instance, year by year). For instance, we could be interested in understading how many publications have been published in each year. To retrieve this information, we can use again the method `value_counts`, but this time applied to the column `publication year`, as shown as follows:

```
publications_per_year = publications["publication year"].value_counts()
publications_per_year
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[11], line 1
----> 1 publications_per_year = publications["publication year"].value_counts()
      2 publications_per_year

NameError: name 'publications' is not defined
```

As you can see, the series contains the number of publications year by year, sorted in descending order, from the year with most publications to that with less publications. In order to draw all these data in the correct temporal order, we need first to sort them in ascending order using the index labels (i.e. the years of publication). For doing so, we can use the `Series` method `sort_index` to generate a new series ordered as mentioned:

```
publications_per_year_sorted = publications_per_year.sort_index()
publications_per_year_sorted
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 publications_per_year_sorted = publications_per_year.sort_index()
      2 publications_per_year_sorted

NameError: name 'publications_per_year' is not defined
```

Then, finally, we can plot this new series using a simple line diagram (the default for the `plot` method) as shown as follows:

```
publications_per_year_sorted.plot()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[13], line 1
----> 1 publications_per_year_sorted.plot()

NameError: name 'publications_per_year_sorted' is not defined
```