# Managing Data with Python

## Session 102

June 2018

M. HOEBEKE
Ph. BORDRON
L. GUÉGUEN
G. LE CORGUILLÉ

# Outline : Session 2

## Working With Heterogeneous Data

1. Regular Expressions: `re`
2. Methods for Sorting Data: `sort & lambda functions`
3. Storing Intermediate Results: `pickle`
4. Using Tabular Data : `csv`
5. Virtual Environments

# Regular Expressions : an Overview

**Regular Expressions are used to analyse and process text information :**

- **By searching for a specific constructs: *patterns*, combinations of *patterns*)**
- **By extracting the portions of text that *match* the *patterns* for later use**
- **By using the portions to *replace* portions of the original text**

**Examples from our fasta example file:**

```
>CK_Syn_RCC307_2183:1894037-1895116:1|psbA
```

- **Extract strain information (species, strain identifier)**
- **Extract position information (start, stop, strand)**
- **Correct position information (replace start, stop or strand with updated values)**

# Regular Expressions : Patterns

**Patterns are the building blocks for searching textual data. They are defined using specific syntactic elements of two types :**

- **Elements to specify the nature of the pattern components (letters, digits…)**
- **Elements to specify how the components are organized wrt. one another (location in the text, number of occurrences)**

**Some common text-based structures that can be defined by patterns are :**

- **Dates :** `1 Jan 2018` — **number - space - letters - space - number**

  `01/01/2018` — **number - slash - number- slash - number**

- **DNA sequences: a series of letters taken from the set {a,t,g,c,A,T,G,C}**

- **Protein sequences: a series of letters taken from the amino acid codes, with constraints on the starting letter (usually M).**

# Regular Expressions : Patterns

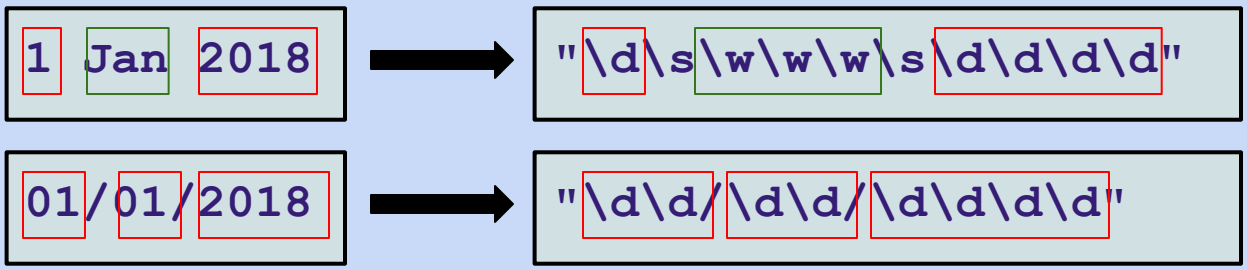
**In Python, regular expressions are made available through the `re` module. This module provides, amongst other things, a set of special syntax elements to define patterns. A pattern is then an ordinary string containing one or more of these special syntax elements.**

**These syntax elements allow to define constraints on the type of the allowed characters :**

| | |
|---|---|
| . | Any character |
| \d | A digit (a character in the range 0 to 9) |
| \w | An alphanumerical character: a to z, A to Z, a digit, an underscore |
| \s | A space character (space or tab) |
| [aeiou] | One of the *a,e,i,o,u* characters. |
| [^aeiou] | Any character except *a,e,i,o,u* |

In uppercase, they mean "anything except."

**A date can then be defined with the following pattern :**

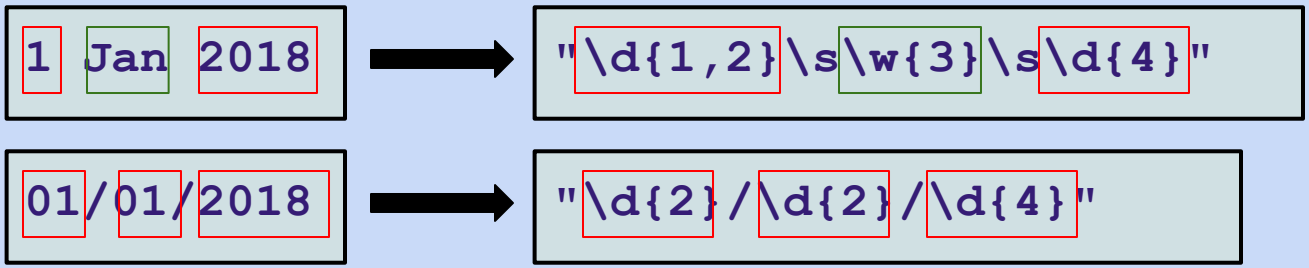


# Regular Expressions : Patterns

**These syntax elements also allow to define constraints on how many occurrences of a character (or character type) are allowed :**

| | |
|---|---|
| * | Any number of occurrences |
| ? | Zero or one occurrence |
| + | One or more occurrences |
| {n} | Exactly *n* occurrences |
| {n,} | At least *n* occurrences |
| {,m} | At most *m* occurrences |
| {n,m} | Between *n* and *m* occurrences |

**The date patterns can be expressed as :**



# Regular Expressions : Patterns

**The sequence patterns can be expressed as :**

- **DNA :** `"[atgcATGC]+"`

- **Proteins :** `"[mM][ACDEFGHIKLMNOPQRSTUVWXY]+"`

**Two special characters allow to "anchor" a pattern at either end of a text line:**

| | |
|---|---|
| ^ | Anchors the pattern from the beginning of the line |
| $ | Anchors the pattern at the end of the line |

**To look for a date of the first example type at the beginning of a line we would use:**

```
"^\d{1,2}\s\w{3}\s\d{4}"
```

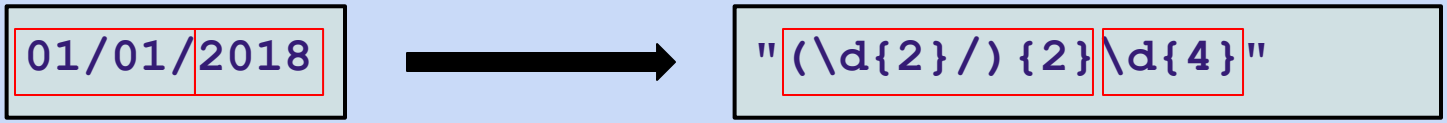
**And to look for a date at the end of a line, we would use:**

```
"\d{1,2}\s\w{3}\s\d{4}$"
```

# Regular Expressions : Patterns

**Pattern elements can also be grouped using parentheses.**

**Hence, to look for a date of the second example type at the beginning of a line we would use:**




**Pattern grouping will allow us in Python to assign the matching groups to variables. More on that later.**

# Regular Expressions : Using the `re` module

1. Define the regular expression by using `re.compile()`

```
patternVar = re.compile(r"patterndef")
```

Denotes a string containing a regular expression.

```
datePattern = re.compile(r"(\d{2})/(\d{2})/(\d{4})")
```

2. Look for the pattern in a candidate string
   a. By matching the whole string

```
matchVar = re.match(patternVar,candidateText)
```

```
dateMatch = re.match(datePattern,"01/01/2018")
```

   b. By looking “inside” the string

```
matchVar = re.search(patternVar,candidateText)
```

```
dateMatch = re.search(datePattern,"01/01/2018")
```

# Regular Expressions : Using the `re` module

## 3. Check if the pattern was found and use it if it was found

```
if matchVar is not None :
    # do something useful with matchVar
    matchString=matchVar.group(0)
```

Group 0 stands for the whole pattern match

```
if dateMatch is not None :
    day=int(dateMatch.group(1))
```

Group 1 stands for the match between the first parentheses

## Exercise 11

- Copy `sequencetools.py` and `readseq.py` to `src/ex11`
- Rename `readseq.py` to `countseqstrand.py`
- Modify the `countseqstrand.py` script so that it takes only one sequence file as argument (`-s` or `--seqfile`)
- Modify the `countseqstrand.py` script to make use of regular expressions to count the number of sequences on both strands (leading and lagging), knowing that the sequence identifiers have the following form :

```
>CK_Syn_RCC307_2183:1894037-1895116:1|psbA
```

Strand : 1, 0 or -1

- Run the program using the file :
  - `'../../data/fasta/Syn_RCC307.fna'`

# Regular Expressions : Tips & Tricks

**Grouping Tip : groups can be named, and once matched, can be referenced by their name.**

```
datePattern = re.compile(r"(?P<day>\d{2})/
                           (?P<month>\d{2})/
                           (?P<year>\d{4})")
```

```
dateMatch = re.match(datePattern,"01/01/2018")
```

```
if dateMatch is not None :
    day=int(dateMatch.group('day'))
```

# Regular Expressions : Tips & Tricks

**Case sensitivity tip: The re `compile()`, `search()` and `match()` function have an optional *flags* argument. One of its values, `re.IGNORECASE` (or `re.I`) makes these function ignore the case of letters in the text to match.**

```
datePattern = re.compile(r"(?P<day>\d{2})
                           (?P<month>\w{3})
                           (?P<year>\d{4})",
                          re.IGNORECASE)
```

**All the following text strings match the pattern :**

```
dateMatch = re.match(datePattern,"01 Jan 2018")
```

```
dateMatch = re.match(datePattern,"01 jan 2018")
```

```
dateMatch = re.match(datePattern,"01 JAN 2018")
```

# Regular Expressions : Tips & Tricks

**Text replacement tip: `re` proposes a `sub()` function allowing to substitute (replace) a pattern with another string using a single call.**

```
>>>newFruit = re.sub(r"oranges",r"bananas",
                    "I like oranges")
>>>newFruit
'I like bananas'
```

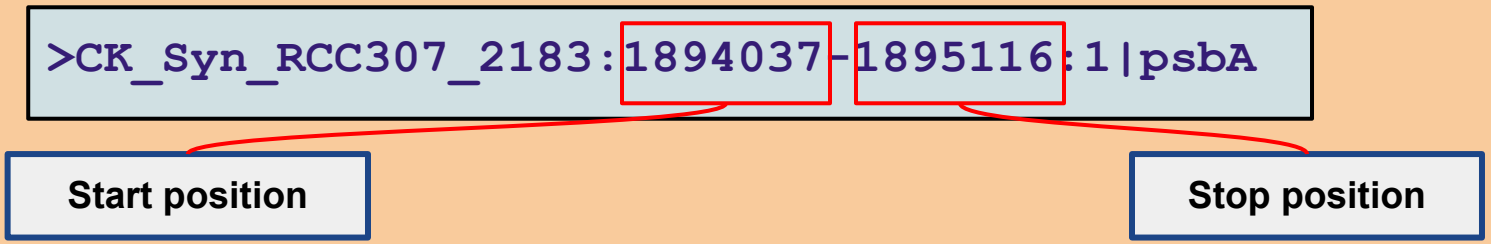
**Groups can be reused in the replacement string with a special syntax :**

```
>>>newDate = re.sub(r"(?P<day>\d{2})/
                   (?P<month>\d{2})/
                   (?P<year>\d{4})",
                   r"\g<month>/)
                   \g<day>/)
                   \g<year>",
                   "31/01/2018")
>>>newDate
'01/31/2018'
```

# Regular Expressions

## Exercise 12

- Copy `sequencetools.py` and `countseqstrand.py` to `src/ex12`
- Rename `countseqstrand.py` to `countgenelength.py`
- Modify the `sequencetools.py` module :
  - to add the following descriptors to the sequence record structure : POSITION_MIN, POSITION_MAX, STRAND
  - to add a function `computeSequencePositionInfo` using regular expressions to fill the above descriptors for a sequence record structure based on the contents of the sequence identifier.




- Modify the `countgenelength.py` script to print the size of the shortest and the longest genes.
- Run the program using the file :
  - `'../../data/fasta/Syn_RCC307.fna'`

# Sorting: using the default sort features

Python provides a `sort()` function to sort lists "in-place" : it changes the order of the list elements. By default elements are sorted in ascending order using the "natural" comparison method of elements. The list elements must be of a homogeneous comparable type.

This is the case with scalar types:

```
>>> fruit=['oranges','bananas','apples']
>>> fruit.sort()
>>> fruit
['apples', 'bananas', 'oranges']
>>> numbers=[678,341,108,834]
>>> numbers.sort()
>>> numbers
[108, 341, 678, 834]
```

# Sorting: using the default sort features

**But lists of lists (of lists) of homogeneous comparable types can also be sorted:**

```
>>> basket=[['oranges',10],['apples',20],
['bananas',2],['apples',3]]
>>> basket.sort()
>>> basket
[['apples', 3], ['apples', 20], ['bananas', 2], ['oranges',
10]]
```

**When the list elements cannot be compared, an exception is thrown :**

```
>>> basket=[ {'oranges' : 10}, {'apples' : 20},
{'bananas':2}]
>>> basket.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'dict' and
'dict'
```

# Sorting: reversing the sort order

**The sort order can be reversed by adding the `reverse=True` parameter to the `sort()` function.**

```
>>> basket=[['oranges',10],['apples',20],
['bananas',2],['apples',3]]
>>> basket.sort(reverse=True)
>>> basket
[['oranges', 10], ['bananas', 2], ['apples', 20], ['apples',
3]]
```

# Sorting: generating a sorted element collection

Python also provides a `sorted()` function. This function returns a new collection with ordered elements of the collection it is applied upon.

```
>>> fruit=['oranges','bananas','apples']
>>> sortedfruit=sorted(fruit)
>>> fruit
['oranges', 'bananas', 'apples']
>>> sortedfruit
['apples', 'bananas', 'oranges']
>>> revsortedfruit=sorted(sortedfruit,reverse=True)
>>> revsortedfruit
['oranges', 'bananas', 'apples']
```

# Sorting: generating a sorted element collection

**This function can also be applied to a dictionary. It will then generate a list with the ordered keys of the dictionary.**

```
>>> basket={'oranges' : 10,'apples' : 20,'bananas' : 2}
>>> sorted(basket)
['apples', 'bananas', 'oranges']
```

# Sorting: defining the sort key

**Both `sort()` and `sorted()` allow to define which "key" to use to perform the sorting. A key is a function that will be applied to each element of the collection to be sorted prior to its comparison.**

**Ex 1: Ordering angles (in degrees)**

```
>>> def angular_compare(degrees) :
...     return (degrees % 360)
...
>>> angles=[0,90,180,270,360,450,540,630,720,810,900]
>>> sorted(angles,key=angular_compare)
[0, 360, 720, 90, 450, 810, 180, 540, 900, 270, 630]
```

# Sorting: defining the sort key

**Ex 2: Ordering a list of dictionaries according to a dictionary key name**

```
>>> basket=[{'fruit':'apple', 'qt' :
20},{'fruit':'banana','qt':10},{'fruit':'orange','qt': 2}]
>>> def fruitname_compare(fruititem):
...     return fruititem['fruit']
...
>>> sorted(basket,key=fruitname_compare)
[{'fruit': 'apple', 'qt': 20}, {'fruit': 'banana', 'qt': 10},
{'fruit': 'orange', 'qt': 2}]
```

**Ex 3: Ordering a list of lists according to a given inner element index**

```
>>> basket=[[10,'apples'],[3,'oranges'],[5,'bananas']]
>>> def fruitposition_compare(fruitelement):
...     return fruitelement[1]
...
>>> sorted(basket,key=fruitposition_compare)
[[10, 'apples'], [5, 'bananas'], [3, 'oranges']]
```

# Sorting: using lambda functions

**When a (sort) function is only used once, it is cumbersome to define it as such. Python provides a special syntax allowing to define a function "on the fly". These functions are called lambda functions, and are built as follows :**

```
lambda args : expression_using_arg
```

**The variable where the argument(s) will be made available on each call**

**An expression using the argument(s) and that will be returned as the result of the lambda function call**

**Ex. 4 : Using a lambda function to sort angles**

```
>>> angles=[0,90,180,270,360,450,540,630,720,810,900]
>>> sorted(angles,key=lambda degrees : (degrees % 360))
[0, 360, 720, 90, 450, 810, 180, 540, 900, 270, 630]
```

# Sorting: using lambda functions

**Ex 5: Ordering a list of dictionaries according to a dictionary key name, using a lambda function**

```
>>> basket=[{'fruit':'apple', 'qt' :
20},{'fruit':'banana','qt':10},{'fruit':'orange','qt': 2}]
>>> sorted(basket,key=lambda d : d['fruit'])
[{'fruit': 'apple', 'qt': 20}, {'fruit': 'banana', 'qt': 10},
{'fruit': 'orange', 'qt': 2}]
```

**Ex 6: Ordering a list of lists according to a given inner element index using a lambda function**

```
>>> basket=[[10,'apples'],[3,'oranges'],[5,'bananas']]
>>> sorted(basket,key=lambda e : e[1])
[[10, 'apples'], [5, 'bananas'], [3, 'oranges']]
```

# Sorting Data

## Exercise 13

- Copy `sequencetools.py` and `countgenelength.py` to `src/ex13`
- Rename `countgenelength.py` to `sortgenes.py`
- Modify the `sequencetools.py` module :
  - to add a function `sortSequencesByLength` taking the sequence information dictionary as argument and returning the list of sequence identifiers ordered by ascending sequence length. Add a second optional argument allowing to define the sort order (ascending or descending).

- Modify the `sortgenes.py` script to display the first and last sequence ids and lengths after sorting the genes.

- Run the program using the file : `'../../data/fasta/Syn_RCC307.fna'`

- Check that the optional sort order argument works as expected.

# Managing intermediate results with `pickle`

When loading and parsing the original input data is expensive (time-consuming), Python offers an easy way to store data structures in a "pickle" file. Data stored in such pickle files can be rapidly loaded by other Python programs, or by subsequent runs of the same program.

To store a data structure in a pickle file, the `pickle.dump()` function is used as follows :

Open the file for writing ('w') data in binary ('b') format.

```
import pickle
…
with open('mydata.pickle','wb') as pfile :
   pickle.dump(mydatastructure,pfile)
```

The variable to store in the pickle file.

# Managing intermediate results with `pickle`

**To load a data structure previously dumped in a pickle file, the `pickle.load()` function is used as follows :**

Open the file for reading ('r') data in binary ('b') format.

```
import pickle
…
with open('mydata.pickle','rb') as pfile :
    mydatastructure=pickle.load(pfile)
```

The variable to fill with the contents of the pickle file.

# Storing intermediate results

## Exercise 14

- Copy `sequencetools.py` and `sortgenes.py` to `src/ex14`
- Modify the `sequencetools.py` module :
  - to add a function `saveSequenceIntoPickleFile` taking a filename and the sequence information dictionary as arguments and storing the sequence information dictionary in pickle format in the file
  - to add a function `loadSequenceFromPickleFile` taking a filename as argument and returning the sequence information dictionary after loading it from the pickle file

- Modify the `sortgenes.py` script to add an option (-p or --pickle) followed by a filename :
  - when both options -s and -p are present, store the sequence information dictionary in a pickle file whose name is given
  - when -p is present without -s, load the sequence information dictionary from the pickle file whose name is given
  - use the verbose mode to print which pickle function is used.

- Run the program using the file :
  `'../../data/fasta/cyanorak_complete.fna'`

# Using tabular data with csv

Data is often stored in tabular data : each line (or record) contains a fixed number of columns separated by a well-defined character (most frequently a semi-colon or a tab character). The first line of the file may be a header line containing the column labels.

The csv module provides all the functionality to read and write tabular data.

When loading data, it reads one line at a time, and returns the parsed result either as a list or as a dictionary.

Ex. 1: Reading tabular data from a tab-delimited file, one list per line

```
import csv
…
with open('mydata.tsv') as csvfile :
    reader=csv.reader(csvfile,delimiter='\t'))
    for line in reader :
        print(", ".join(line))
```

Optional, comma by default.

Each element of line contains the value of one column

# Using tabular data with csv

**Ex. 2: Reading tabular data from a tab-delimited file, one dictionary per line. The first line of the file contains the column headers.**

```
import csv
…
with open('mydata.tsv') as csvfile :
    reader=csv.DictReader(csvfile,delimiter='\t'))
    for line in reader :
        print(", ".join(line.values()))
```

A dictionary where the keys are the column values of the first line in the file.

**Ex. 3: Reading tabular data from a tab-delimited file, one dictionary per line, explicitly defining the column headers.**

```
import csv
…
with open('mydata.tsv') as csvfile :
reader=csv.DictReader(csvfile,fieldnames=['lastname','firstna
me','age'],delimiter='\t'))
    for line in reader :
        print(", ".join(line.values()))
```

A dictionary where the keys are the column values of the `fieldnames` argument.

# Using tabular data with `csv`

**For storing data, the `writerows()` method allows to write a whole list of lines at once.**
**Lines can also be written one at a time with `writerow()`**

**Ex. 4: Writing tabular data into a comma delimited file, one list per line.**

```
import csv
…
fruit=[['apples',10],['bananas',3],['oranges',5]]
with open('fruit.csv','w') as csvfile :
    writer=csv.writer(csvfile)
    writer.writerows(fruit)
```

# Using tabular data with csv

**Ex. 5: Writing tabular data into a comma delimited file, one dictionary per line, using a subset of the keys.**

```
import csv
…
fruit=[{'name':'apples','qt' : 10,'price':4.5},
       {'name':'oranges','qt' : 5,'price':3.2},
       {'name':'bananas','qt' : 3,'price':2.0}]

with open('fruit.csv','w') as csvfile :
    writer=csv.DictWriter(csvfile,fieldnames=['name','qt']))
    writer.writeheader()
    writer.writerows(fruit)
```

**Ex. 6: Writing tabular data into a comma delimited file, one dictionary per line, using all the keys.**

```
(…)
with open('fruit.csv','w') as csvfile :
    writer=csv.DictWriter(csvfile,fieldnames=fruit[0].keys()))
    writer.writeheader()
    writer.writerows(fruit)
```

# Working with tabular data

## Exercise 15

- Create a new directory `src/ex15`
- Write a module called `inventairestools.py` containing :
  - a function called `loadBiotopes` taking a filename as argument and using a `csv.DictReader` to load the contents of the file which is supposed to contain a header line, and columns separated by a tab character (`'\t'`)
  - the function returns a list of dictionaries, one for each data line of the file.

- Write a script called `loadinv.py` :
  - processing the command line arguments (`-b` or `--biotopes` followed by a filename)
  - calling the `loadBiotopes` function of `inventairestools.py`
  - displaying the contents of the first data line of the file.

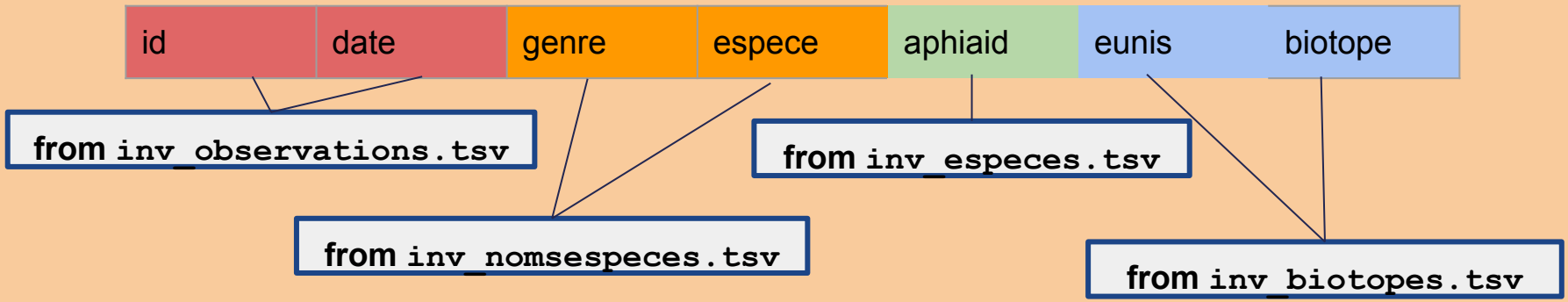
- Run the program using the file :
  `'../../data/tabular/inv_biotopes.tsv'`

# Working with tabular data

## Exercise 16

- Look at the following data files in the `data/tabular` directory :
  - `inv_observations.tsv`, `inv_nomsespeces.tsv`, `inv_especes.tsv` and `inv_biotopes.tsv`
  - Each table contains an `id` column.
  - The table in file `inv_observations.tsv` contains columns with ids referencing the entries of the other tables (`id_biotope`, `id_nomespece`, `id_espece`).

- The goal of the exercise is to generate a CSV file with a line for each observation containing a summary of the related data as follows :



# Working with tabular data

## Exercise 16

- Copy both `inventairestools.py` and `loadinv.py` to a new directory `src/ex16`
- Rename `loadinv.py` to `summarizeobs.py`
- Extend the `inventairestools.py` module to add the new functions :
  - `loadSpeciesNames, loadObservations, loadSpecies` taking a filename as argument and using a `csv.DictReader` to load the contents of a file which is supposed to contain a header line, and columns separated by a tab character (`'\t'`)
  - the function returns a list of dictionaries, one for each data line of the file.
  - try to minimize cutting/pasting code, write functions instead
- Modify the `summarizeobs.py` script to :
  - process the command line arguments (`-b` or `--biotopes, -s` or `--species, -n` or `--speciesnames,-o` or `--observations,-r` or `--resultfile`)
  - call the `loadXXX` functions of `inventairestools.py`
  - store the table with the summary in a resultfile (you can use `'/tmp/observations.csv'` for ex.)
- Run the program with the `inv_*` data files.

# Virtual Environments

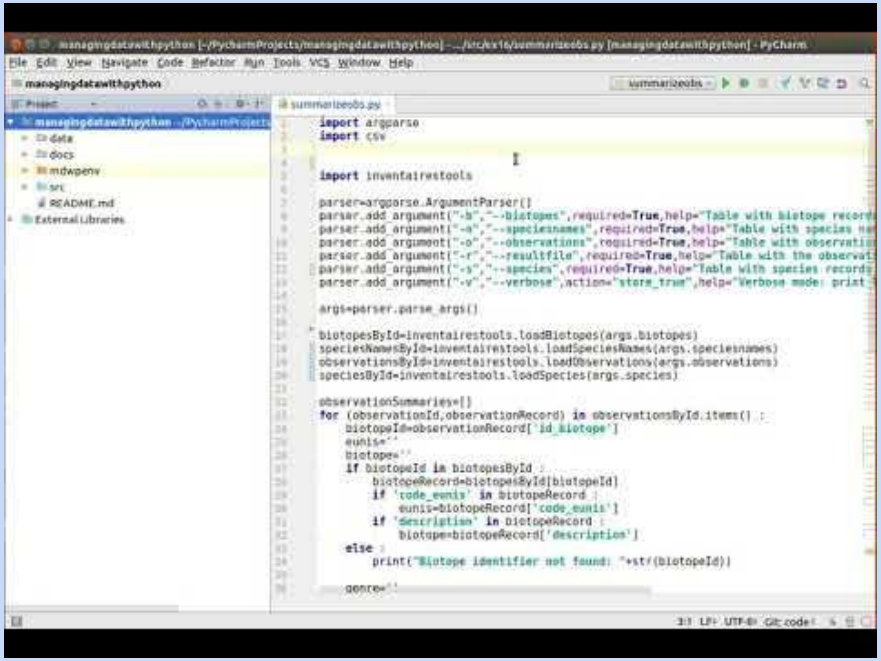
**The Python interpreter comes with a load of standard modules. However, sometimes it is necessary to install additional modules. Their installation in the system (shared) Python directories is not always possible or recommended.**

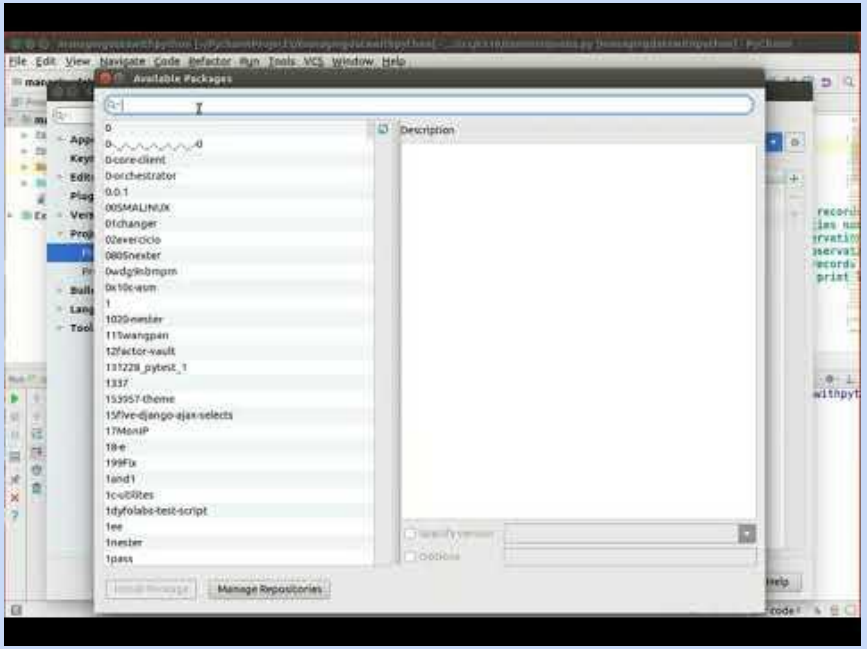
**Enter the Python virtual environments. These allow the installation in a user directory of an instance of the Python interpreter and its standard modules. This instance can then be *activated* making it the default Python installation for a work session. Once activated, module installations can be carried out in this virtual environment by the user who created the virtual environment in the first place.**

**This is a very cheap operation. It is thus frequent to create one instance of a virtual environment for every application. This allows to tailor which modules or even which module versions are available to applications.**

# Virtual Environments with PyCharm

**PyCharm provides all the functionalities to create virtual environments and to manage module installations (or removals) in these environments.**



# Virtual Environments Using a Terminal

Python installation provide a `virtualenv` command. It takes an argument with a (not already existing) directory name where the virtual environment will be created.
It supports several options, most notably `-p` followed by the Python interpreter that is to be used in the virtual environment.

```
[foobar] virtualenv -p python3 myvenv
```

Once the virtual environment is created, it has to be activated with the following command :

```
[foobar] . ./myvenv/bin/activate
```

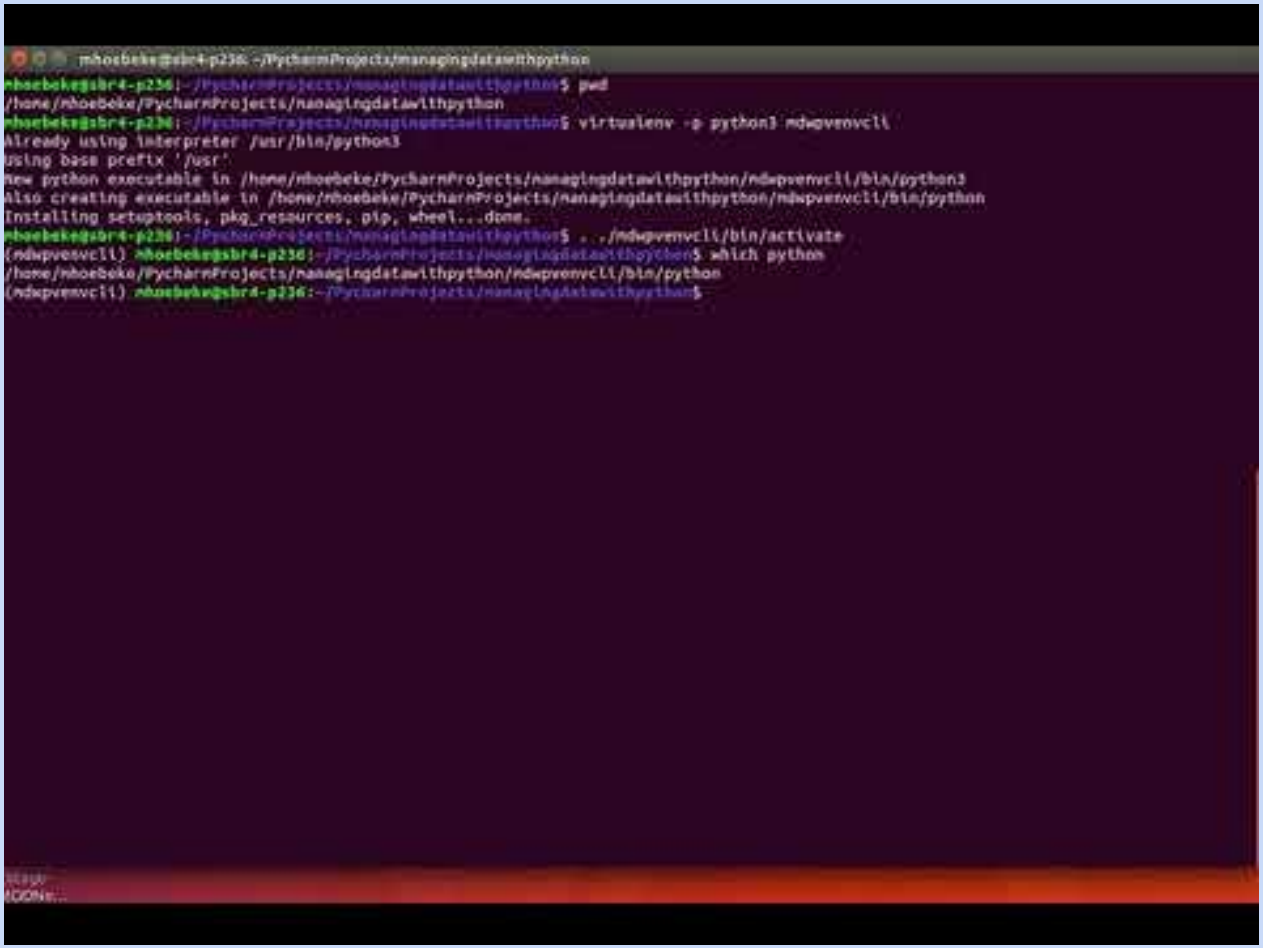
The prompt will be prefixed with the name of the virtual environment indicating that activation was successful.

It then becomes possible to install new modules in the environment by using the `pip install` command

```
(myvenv)[foobar] pip install modulename
```

# Virtual Environments Using a Terminal

## Screencast Time Again!



## Exercise 17

- With PyCharm :
  - create a virtual environment, based on the `python3` interpreter, and with the name `abimsenv`
  - install the `requests` module in this virtual environment
  - check that the module can be imported without errors

- For those having a little Linux know-how:
  - perform the same operations in your project directory (remember the `cdprojet` command?)