

ABIMS⁴

Data Management with Python

January 2018 Session

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

UPMC
SORBONNE UNIVERSITÉS



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. [\[link\]](#)



From “Hello world” to your first Python Module

1. Introduction
2. Running Python programs
3. Reading Data from Text Files
4. Essential Data Types
5. Flow Control Instructions
6. Structuring Data
7. Handling Program Arguments
8. Using Modules
9. Writing Functions
10. Turning a Python Script into a Module

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. Intermezzo : Virtual Environments
6. A Word on XLS(X) Files : `openpyxl`
7. Grabbing Data From the Web : `requests` & `json`
8. Managing Configuration Files : `configparser`
9. Interacting With the Operating System : `os`

Object Oriented Python

1. What is Object Oriented Programming ?
2. Object Oriented Python
3. Unit Testing your Python Code
4. Using the `logging` Module
5. The Basics of Exception Handling

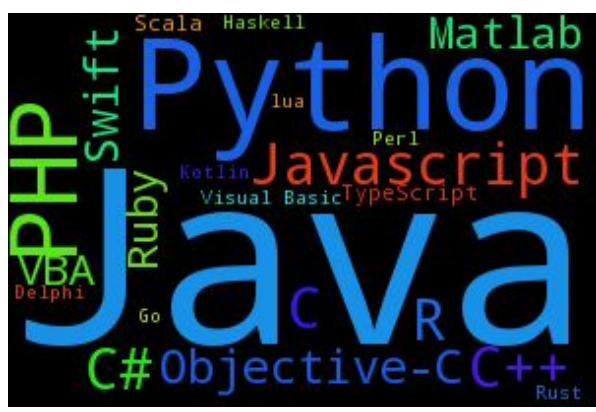
Domain Specific Python Modules

1. The BioPython toolkit
2. Graph Data and NetworkX

From “Hello world” to your first Python Module

1. Introduction
2. Running Python programs
3. Reading Data from Text Files
4. Essential Data Types
5. Flow Control Instructions
6. Structuring Data
7. Handling Program Arguments
8. Using Modules
9. Writing Functions
10. Turning a Python Script into a Module

Why Python ?



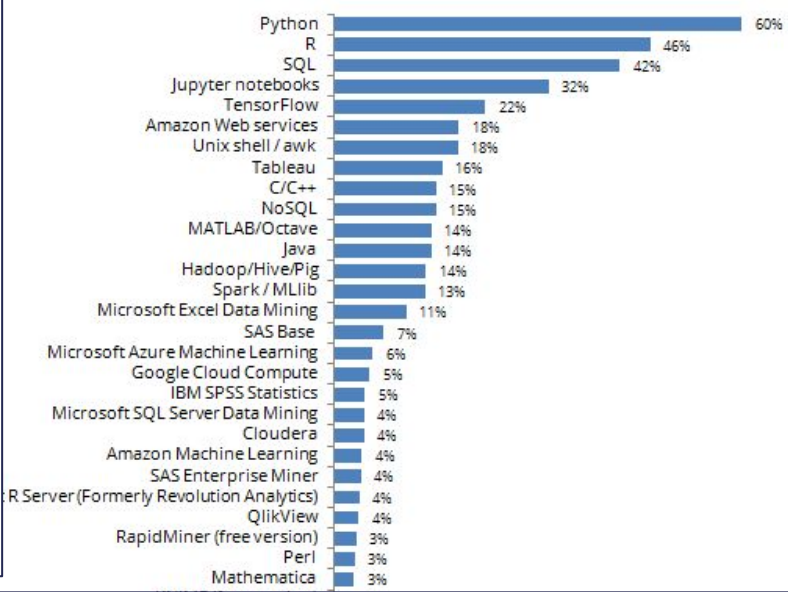
P
Y
P
L

Worldwide, Jan 2018 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Java	21.2 %	-1.4 %
2		Python	19.3 %	+4.9 %
3		PHP	8.0 %	-1.7 %
4	↑	Javascript	7.9 %	+0.1 %
5	↓	C#	7.5 %	-1.0 %
	↑	C++	6.3 %	-0.7 %
	↓	C	6.3 %	-0.8 %
	↑	R	3.9 %	+0.6 %

Copyright 2018 Business Over Broadway

Data Science / Analytics Tools, Technologies and Languages Used in Past Year



© TIOBE



Jan 2018	Jan 2017	Change	Program
1	1		Java
2	2		C
3	3		C++
4	5	↑	Python
5	4	↓	C#
6	7	↑	JavaScript
7	6	↓	Visual Basic .NET
8	16	↑↑	R

Why Python ?

1991

PYTHON (FOR BRITISH COMEDY TROUPE
MONTY PYTHON)

General-purpose, high-level.
Created to support a variety of
programming styles and be fun to
use. Tutorials, sample code, and
instructions often contain
Monty Python references.

CREATOR

GUIDO VAN ROSSUM
CWI



PRIMARY USES

Web applications,
software
development,
information
security

USED BY

Google, Yahoo,
Spotify



Which version of Python ?

Two major versions are still actively maintained :

- The 2.x branch (2.7 being the last and latest) :
 - Found on many platforms as default version (including your workstations, and the ABiMS cluster nodes)
 - Accessible unambiguously through the `python2` command
- The 3.x branch (3.6 being the latest as of this writing) :
 - Available and installed by default but not configured as default on many recent Linux distributions
 - Accessible unambiguously through the `python3` command

Whenever possible, prefer the 3.x version

(even if the differences with 2.x are not obvious for this introductory course)

To check which version comes configured as default :

```
[mark@~] python --version  
Python 2.7.14
```

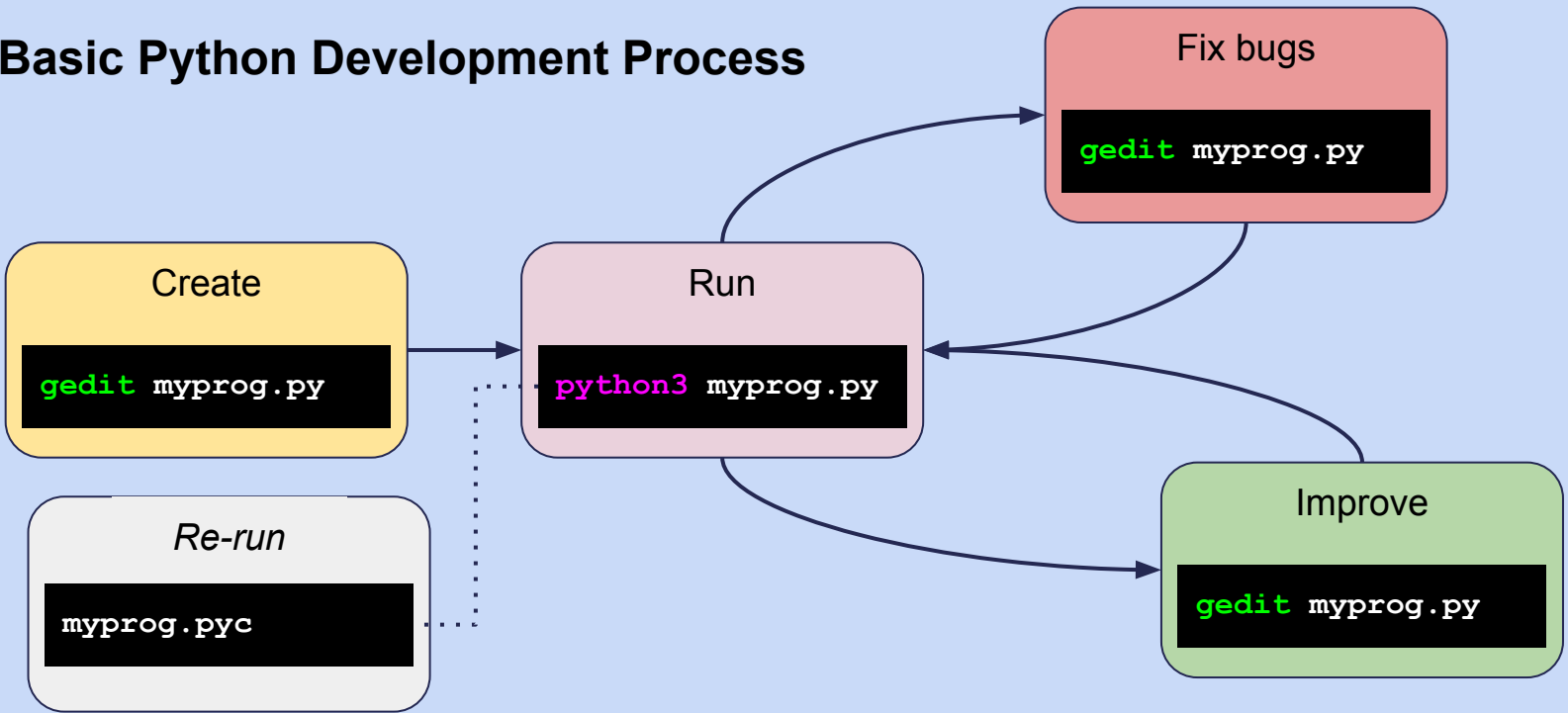
Don't worry : we'll learn how to master which version to use

(even if the differences with 2.x are not obvious for this introductory course)

Writing & Running Python Programs

- A Python program is made of one or more **sequences of properly formatted lines of text** containing instructions that can be executed by the **Python interpreter**.
- Python programs are often stored in (text) files conventionally suffixed with **.py**
- **The Python Interpreter** translates your program code (text) into a machine-readable representation

Basic Python Development Process



Writing & Running Python Programs

Using Python in interactive mode :

- ★ Done by running the interpreter without specifying a program file
- ★ Suitable for performing small checks or tests

```
[mark@~] python3
Python 3.6.3 (default, Oct 3 2017, 21:45:48)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("My First Python Line of Code")
My First Python Line of Code
```

The Python *prompt*

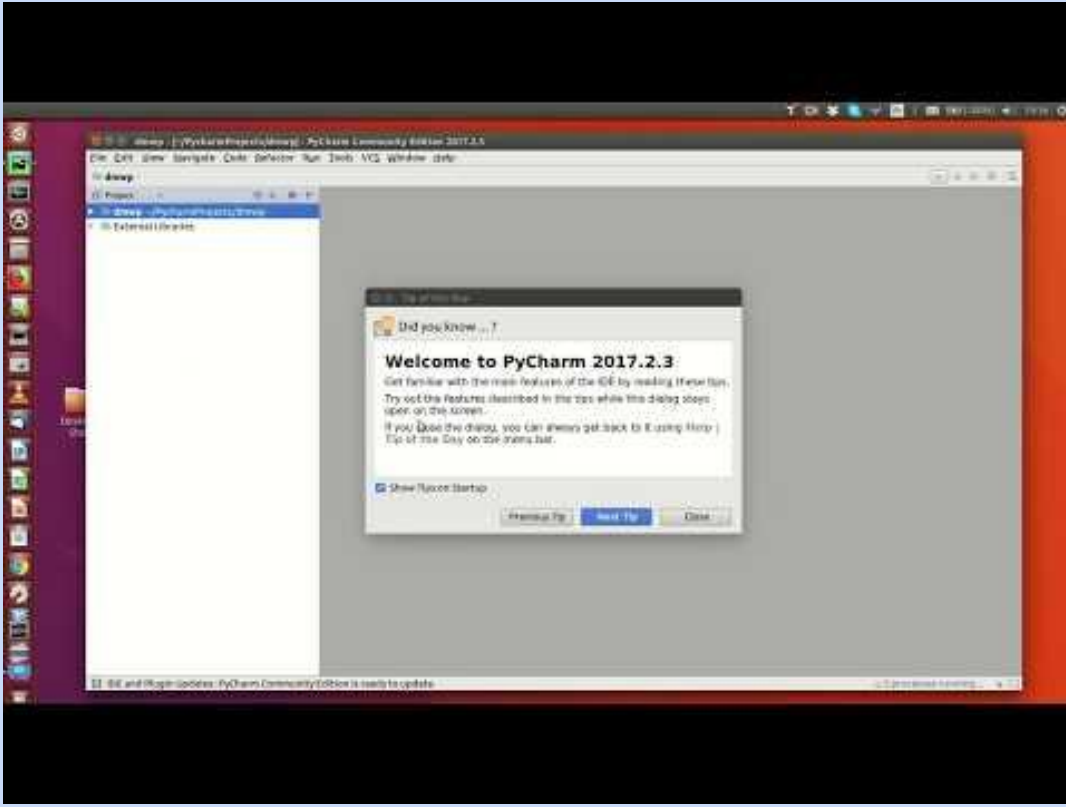
The code I've typed

The result of the evaluation of the code I've typed

Writing & Running Python Programs

Developing With an Integrated Development Environment (IDE) :

- ★ **Platform Independent** : Linux, Windows, MacOS X
- ★ **Streamlines the Development Process** : Project File Organization / Coding / Debugging / Version Control / Dependency Management



<https://www.jetbrains.com/pycharm/>

Exercise 1

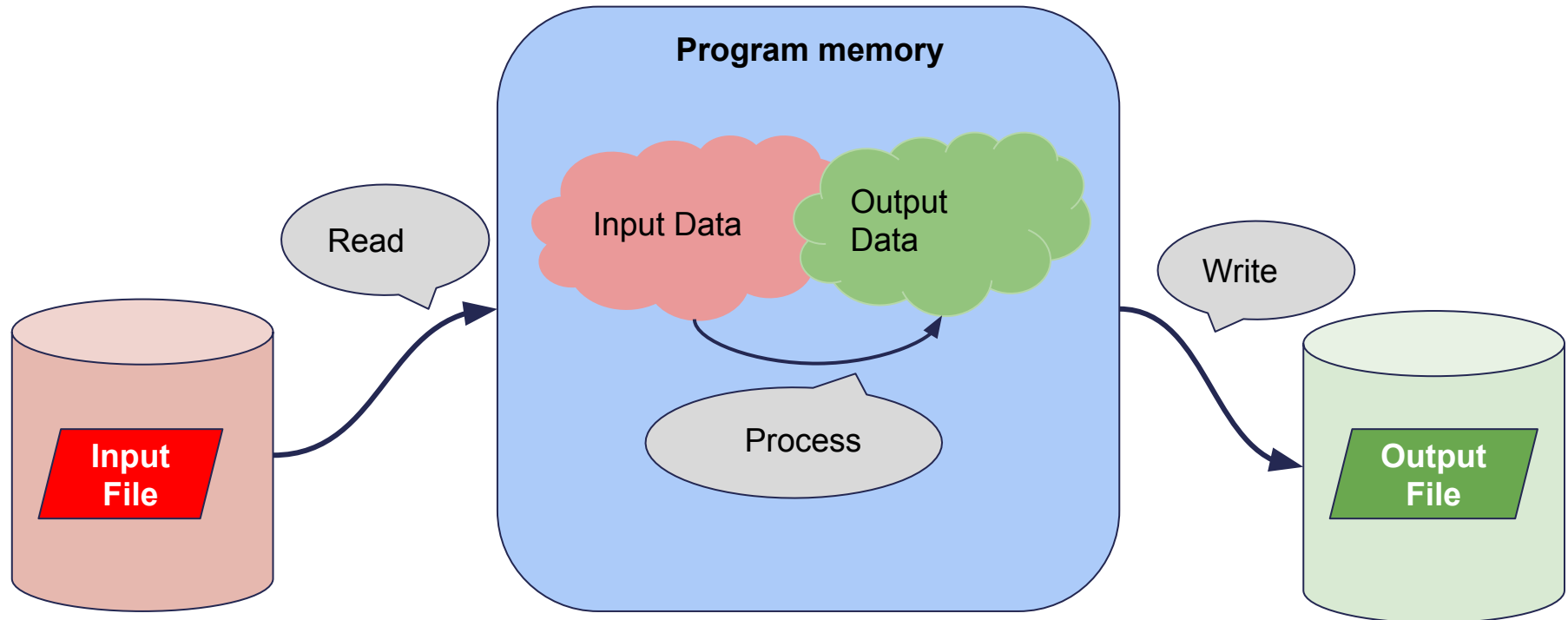
Write & run your `helloworld.py` script using PyCharm, based on the previous screencast, with:

- the **same project name**
- the **same directory layout**
- the script's contents :

```
print('Hello world')
```

A *useful* program :

1. Reads some data (from a file, from the network)
2. Processes the data (mainly in memory) to compute the wanted results
3. Generates the results as output (files)



Issues to consider when writing a *useful* program :

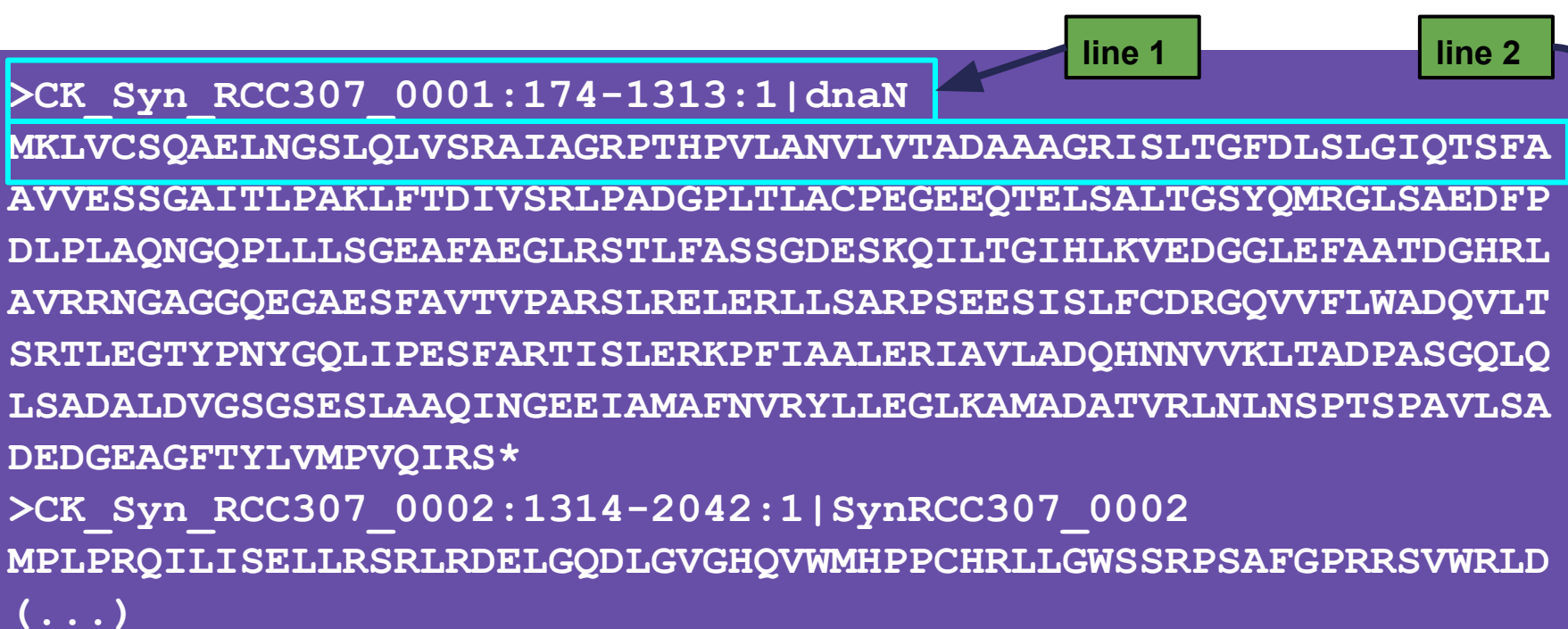
1. What are the most appropriate data structures for the processing step(s) :
 - a. easy access to the elementary data needed for processing
 - b. logical grouping of elementary data chunks used together
 - c. minimal in-memory redundancy

2. How to fit the input data in these appropriate data structures :
 - a. what are the accepted/recognized input formats ?
 - b. are there already existing tools to read these formats ?
 - c. how to balance slow read operations with the program's memory footprint ?

3. What are the requirements for the output file(s) :
 - a. what is the expected output format(s) ?
 - b. are there already existing tools to write these formats ?
 - c. when is it possible to start writing the results (at the end only, or are there intermediary results available early-on) ?

Reading Data from Text Files

A Text File contains a **series of lines**. Each **line** is made of a **series of characters** followed by a newline character.



The diagram shows two lines of text on a purple background. The first line is highlighted with a cyan box and labeled 'line 1'. The second line is also highlighted with a cyan box and labeled 'line 2'. Arrows point from the labels to the corresponding lines. The text is as follows:

```
>CK_Syn_RCC307_0001:174-1313:1|dnaN
MKLVCSQAELNGSLQLVSRAIAGRPTHPLANVLVTADAAAGRISLTGFDLSLGIQTSFA
AVVSSGAITLPAKLFTDIVSRLPADGPLTLACPEGEEQTELSALTGSYQMRGLSAEDFP
DLPLAQNGQPLLLSGEAFAGLRSTLFASSGDESKQILTGIHLKVEDGGLEFAATDGHRL
AVRRNGAGGQEGAESFAVTVPARSLRELERLLSARPSEESISLFCDRGQVFLWADQVLT
SRTLEGTYPNYGQLIPESFARTISLERKPFIAALERIAVLADQHNNVVKLTADPASGQLQ
LSADALDVGSGSESLAAQINGEEIAMAFNVRYLLEGLKAMADATVRLNLNSPTSPAVLSA
DEDGEAGFTYLVMPVQIRS*
>CK_Syn_RCC307_0002:1314-2042:1|SynRCC307_0002
MPLPRQILISELLRSRLRDELGQDLGVGHQVWMHPPCHRLLGWSSRPSAFGPRRSVWRLD
(...)
```


Reading Data from Text Files

In Python, reading a text file can be done as follows :

```
infile = open('/path/to/my/file.txt')  
data = infile.readlines()  
infile.close()
```

- **Line 1 :**
 - The `open()` function is called to open(!) the input file, called `/path/to/my/file.txt`
 - A reference to the opened file is stored in a variable called `infile` using the assignment operator (equals sign)
- **Line 2 :**
 - The `readlines()` function is applied to the `infile` reference using the dot operator. It reads the file contents, line by line and returns the result.
 - The result is stored in the `data` variable.
- **Line 3 :**
 - The `close()` function is applied to the `infile` reference to free any resources (memory) used for reading the file.

Variables : Scalar Types

Now, how can we access the contents of the file, a.k.a what is the nature of the **data** variable ?

Python variables come in various *types*, among which :
Scalar types, used to store a single value, including :

- a. String variables used for storing letters, words, texts...

```
language = 'Python.'
```

- b. Numerical variables used for storing numbers :
 - i. Integer values :

```
one = 1
```

- ii. Floating point values :

```
pi = 3.141592
```

- c. Boolean variables, used for storing True or False values

```
bioInformaticsIsEasy = True
```

Variables : Container Types

Container types (or collections), are used to store several elements of any type. They include :

- a. **Lists** - used to store *indexed* collections of elements

```
weekendDays = ['Saturday', 'Sunday']
```

- b. **Tuples** - used to store *immutable indexed* collections of elements

```
timeSpans = ('AM', 'PM')
```

- c. **Dictionaries** - used to store *{key:value} pairs*

```
languageCodes = {'FR': 'French',  
                 'EN': 'English'}
```

- d. **Sets** - used to store unordered *unique* elements

```
seasons = set(('Spring', 'Summer',  
              'Fall', 'Winter'))
```

Variables : Type Determination

Variables are created when they are first assigned a value using the assignment operator (=).

In our example, `infile` is created on completion of line 1, and `data` is created on completion of line 2.

Variables are “destroyed” when they are no longer visible (more on that later)

The actual type of a variable can be determined using the `type()` function :

```
>>>data=[1,2,3]  
>>>type(data)
```

will display :

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

Don't bother just yet

Variables : Type Conversions

Type conversions (when they make sense) are possible using the destination type name as function :

- any scalar type can be converted to a string variable, using `str()`
- integer and float types can be inter-converted using `int()` or `float()`
- when applicable, strings can be converted to integers or floats using `int()` or `float()`

```
>>>booleanTrue = True
>>>booleanString = str(booleanTrue)
>>>booleanString
'True'
>>>floatPi = 3.141592
>>>strPi = str(floatPi)
>>>strPi
'3.141592'
>>>intPi = int(floatPi)
>>>strExp = '2.71828'
>>>floatExp = float(strExp)
>>>floatExp
2.71828
```

Variables : Using Lists

List elements are accessed through their numerical index, starting with zero, as in :

```
firstLine = data[0]
```

Negative indices can be used to access elements from the end of the list :

```
lastLine = data[-1]
```

List slices can be extracted using colons.

```
firstThree = data[0:3]
```

Warning : the slice does not include the element with the upper index.

Either of the two indices (or both!) can be omitted:

- without lower index, the slice starts at the beginning of the list.
- without upper index, the slice extends to the end of the list

```
allButLast = data[:-1]
```

```
completeCopy = data[:]
```

Variables : Using Lists

Lists grow automatically in size, when adding new elements with the `append()` function :

```
weekEndDays.append('Friday')
```

Or by adding other lists with the `extend()` function:

```
weekEndDays.extend(['Monday', 'Tuesday'])
```

New lists can be built by using the addition operator (+)

```
extraLongWEs = weekEndDays + ['Thursday']
```

The length of a list can be determined with the `len()` function :

```
numberOfWEDays = len(weekEndDays)
```

Exercise 2

Write a `countlines.py` script (in the `src/ex02` folder of your project) printing the number of lines in the `Syn_RCC307.faa` file located in your data directory.

Hint - the path to the data file is :

```
' ../../data/fasta/Syn_RCC307.faa '
```


Variables : Using Strings

Strings share some basic features with lists, such as :

- Bracket operators to access elements and/or slices

```
firstLetter = stringExample[0]
```

- String concatenation with the + operator

```
helloWorld = 'Hello ' + 'World'
```

- Use of the len() function

```
>>>len(helloWorld)  
11
```

Strings can be built from lists using the join() function

```
>>>fruitString = ','.join(['apples', 'oranges', 'bananas'])  
>>>type(fruitString)  
<class 'str'>  
>>>fruitString  
'apples, oranges, bananas'
```

Variables : Using Strings

Strings can be separated into list elements using the `split()` function

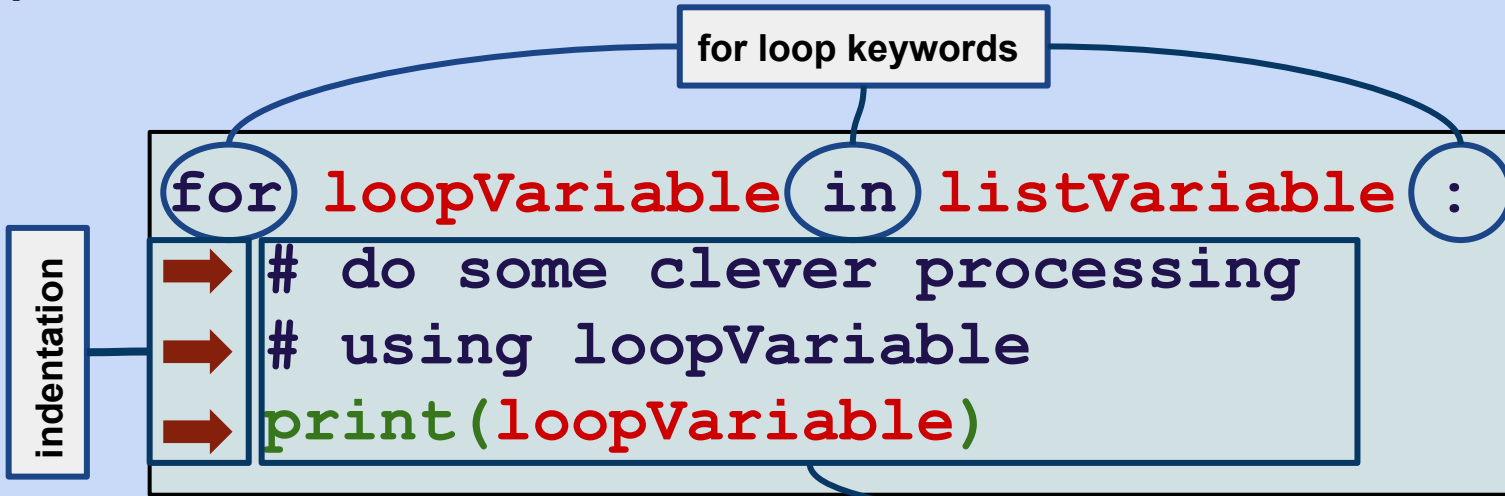
```
>>>fruitList = fruitString.split(', ')
>>>type(fruitList)
<class 'list'>
>>>fruitList
['apples', 'oranges', 'bananas']
```

Strings (as almost any variables) can be modified using the assignment operator =

```
>>>greetings = 'Hello'
>>>greetings = greetings + ', world.'
>>>greetings
'Hello, world.'
```

Lists, Loops and Block Structures

A straightforward way to access each element of a list in turn relies on the `for` loop, written as follows :



Instruction block executed for each successive element in `listVariable`. The values of the successive elements are assigned to `loopVariable`.

Each line of the *instruction block* is indented (space(s) or tab) wrt. the line with the `for ... in ... :` instruction.

Exercise 3

Write a `join.py` script (in the `src/ex03` folder of your project) building a list with the following strings :

```
'Union', 'of', 'the', 'Snake'
```

Then, using a `for` loop, concatenate the list elements to form a string where the words are separated with a space character.

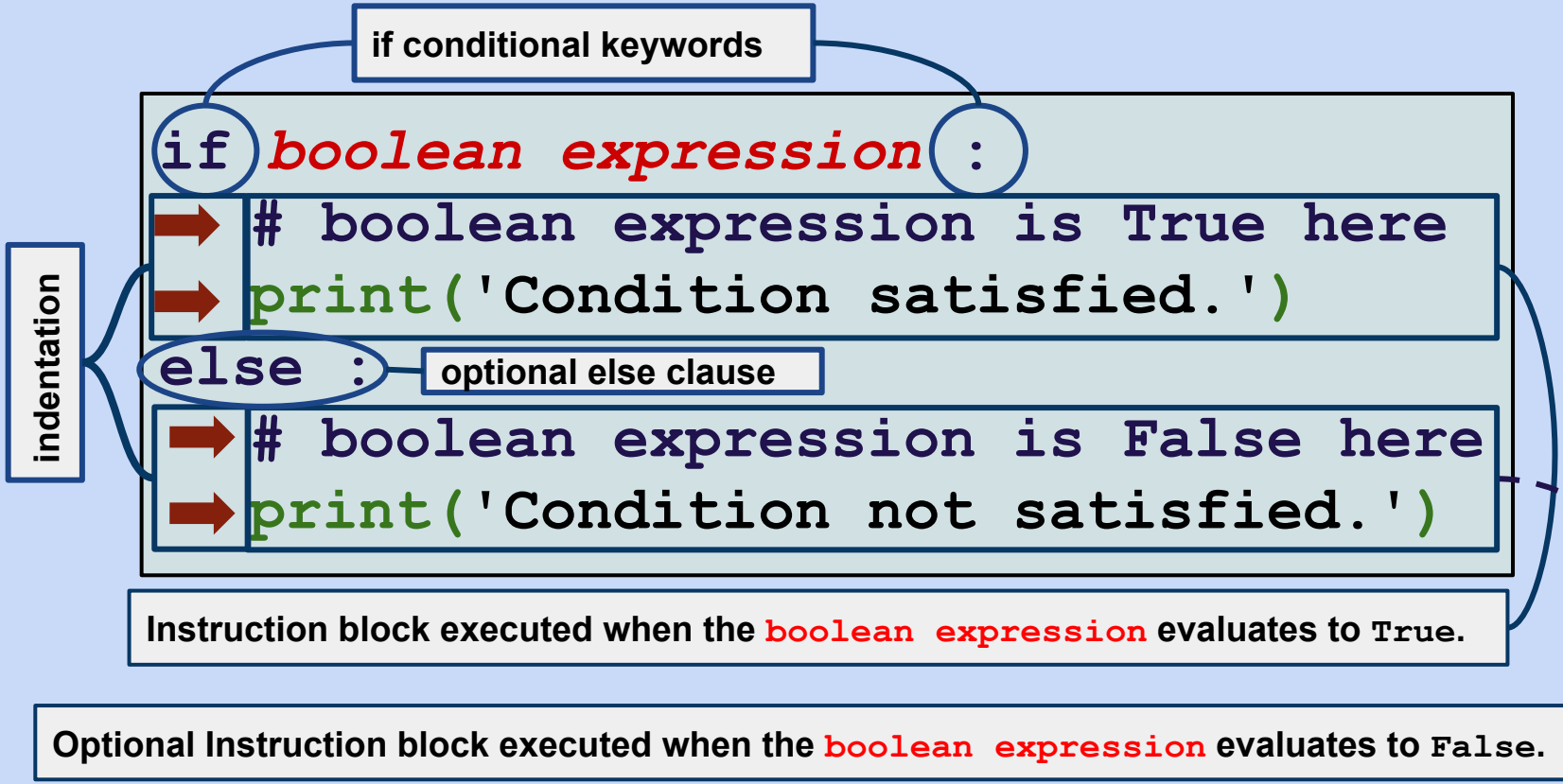
After the loop has completed, print the value of the string, which should be :

```
'Union of the Snake '
```

How would you remove the trailing space ?

Block Structures and Conditionals

The most often used conditional control flow structure is the `if (else)` construction, which is build as follows :



Conditionals and boolean expressions

Boolean expressions are expressions whose evaluation yields either `True` or `False`.

They very often rely on one of the following operators :

- the equality operator : `==`

```
if language == 'Perl' :  
    print("You're in the wrong class, mate.")
```

- the inequality operator : `!=`

```
if language != 'Python' :  
    print("You're in the wrong class, mate.")
```

- comparison operators : `>` (greater than), `>=` (greater or equal), `<` (lesser than), `<=` (lesser or equal)

```
if distanceKm <= 1.0 :  
    print("You're better of walking.")
```

Caution : don't try to use operators with incompatible types

```
>>> distanceKm='One'  
>>> if distanceKm <= 1.0 :  
...     print("You're better of walking.")  
...  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: '<=>' not supported between instances of 'str'  
and 'float'
```

Conditionals and boolean expressions

Boolean expressions can be combined with logical operators : **and** and **or**

- the conjunction operator : **and**

```
if distanceKm <= 1.0 and weather == 'Sunny' :  
    print('You're better of walking.')
```

- the disjunction operator : **or**

```
if winspeedKmH >= 100.0 or weather == 'Overcast' :  
    print('Consider taking a cab.')
```

Boolean expressions can be negated using the **not** operator :

```
if not weather == 'Sunny' :  
    print('An umbrella might be useful.')
```


Conditionals and boolean expressions

- Logical operators have priorities : `not` > `and` > `or`
- When in doubt, using parentheses may lift ambiguities

```
if distanceKm <= 1.0 and weather == 'Sunny' or weather == 'Mild' :  
    print('You're better of walking.')
```



Is interpreted as

```
if ( distanceKm <= 1.0 and weather == 'Sunny' ) or weather == 'Mild' :  
    print('You're better of walking.')
```

When the intended expression would be

```
if distanceKm <= 1.0 and ( weather == 'Sunny' or weather == 'Mild' ) :  
    print('You're better of walking.')
```



Exercise 4

Write a `readseq.py` script (in the `src/ex04` folder of your project) taking as input, the already used file:

```
'../..../data/fasta/Syn_RCC307.faa'
```

and building :

- a list (called `seqIds`) with the sequence identifiers
- a list (called `sequences`) with the sequence amino acids.

Check that at index 1234 :

- the identifier is :
`>CK_Syn_RCC307_1247:1103206-1103493:1|SynRCC307_1247`
- and the length of the amino acid sequence is : 96

Caution : when reading a line, Python also reads (and stores) the newline character ending the line.

When Structuring Data Makes Sense

In the previous exercise, information about a single sequence was stored in two separate collections :

- One collection for the sequence identifiers
- One collection for the amino acids

The relationship between the two was implicit through the use of an identical index.

It makes more sense to explicitly link an identifier with its amino acids using a *dictionary*.

One possibility, usable *when the identifiers are unique* :

- The **dictionary key** is the sequence identifier
- The **associated information** is the amino acid sequence.

```
{ '>CK_Syn_RCC307_1247:1103206-1103493:1|SynRCC307_1247' :  
'LSMAEQNSSASLLLSALTGA AVGAAGLTWVLLSRAERRQALGDQFKRLGLNGAPTNGSSAQGSPENLEQKVNRLNI  
AIEDVRRQLESMAPESSN*' }
```

Creating an empty dictionary :

```
sequenceInfo = {}
```

Adding an element to a dictionary :

```
sequenceInfo[dictKey] = dictInfo
```

- `dictKey` is often a string variable,
- `dictInfo` can be a variable of any type.

For example when both `dictKey` and `dictInfo` are strings:

```
sequenceInfo['>sample_seq_id'] = 'ADGKORML(...)'
```

Removing an *existing* element from a dictionary :

```
del(sequenceInfo[dictKey])
```

Retrieving the number of elements of a dictionary :

```
totalSequences=len(sequenceInfo)
```

Dictionaries : Basic Usage

Checking if a dictionary contains a specific key with the `in` operator :

```
if seqId in sequenceInfo :  
    print(seqId+' is already known!')
```

More frequently used associated to the `not` operator to check whether a key is not already present in a dictionary :

```
if seqId not in sequenceInfo :  
    sequenceInfo[seqId] = residues
```

Retrieving the list of keys of a dictionary :

```
dictKeys = sequenceInfo.keys()
```

Can be used to loop over dictionary entries :

```
for dictKey in sequenceInfo.keys():  
    residues = sequenceInfo[dictKey]
```

Retrieving the list of values of a dictionary :

```
allResidues = sequenceInfo.values()
```

Can be used to loop over dictionary entries :

```
totalResidues = 0  
for seqResidues in sequenceInfo.values():  
    totalResidues = totalResidues + len(seqResidues)
```

Looping over complete dictionary items :

```
for (id,residues) in sequenceInfo.items():  
    print('Seq.: '+id+' has '+len(residues)+' aa.')
```

Exercise 5

- Copy `readseq.py` to directory `src/ex05/readseq.py`
- Change `readseq.py` to use a single sequence dictionary instead of a pair of lists.
- Check that there are no duplicated sequence identifiers in the data file.
- Check that the length of the sequence with identifier

```
>CK_Syn_RCC307_1247:1103206-1103493:1|SynRCC307_1247
```

is indeed 96.

Handling Command-Line Arguments

The current version of our script has one major shortcoming: the name of the data file is *hard coded*. Meaning that *in order to parse another data file, we have to modify our code !*

To overcome this flaw, it would be nice to be able to specify the name of the data file as argument to our script as in :

```
[mark@~] python3 readseq.py mysequences.faa
```

This can be done using a *module* that comes standard with Python.

Using Python Modules : argparse

A Python *module* is a package or library providing a set of features aimed to be reused across programs (ex. `biopython` for bioinformatics, `numpy` for scientific computation, `networkx` for graph manipulation...)

These features can include:

- data structures
- functions
- *classes* (which we'll see later on)

To access the features in a Python program, the module needs to be imported in the program.

To import the complete set of features of a module in a Python program, the `import` instruction is used. The features included in the module are then accessed in the Python program by prefixing the feature name with the module name :

```
import moduleName  
...  
result=moduleName.featureName()
```

Using Python Modules : argparse

It is also possible to import specific features supplied by a module by using the `from ... import` instructions. Accessing the feature can then be done directly, without prefixing it with the module name :

```
from moduleName import featureName
...
result=featureName()
```

The former method is recommended over the latter one. It is less subject to name collisions which can occur when two modules define a feature with the same name.

Its drawback is that it imports the whole contents of the module. But that's usually not a problem.

Each standard module is duly documented on the Python reference documentation web site :

<https://docs.python.org>

Double-check however that you are reading the documentation matching your version of Python
<https://docs.python.org/3/howto/argparse.html> *is not* <https://docs.python.org/2/howto/argparse.html>

The `argparse` module provides all that's needed to make use of command-line arguments inside a Python program, and relies on a three step method :

1. **Declare the structure of the possible command-line arguments and options.**
2. **Call a function that fills the structure by analyzing how the program was run.**
3. **Use the structure to retrieve values that were provided for arguments and options on the command-line.**

Step 1 : define the ArgumentParser

```
import argparse
...
parser=argparse.ArgumentParser(description='Read sequences
from a multi-fasta file')
parser.add_argument('infile',help='multi-fasta input file')
```

Create the parser using the `ArgumentParser()` (special) function. Use a named argument - `description` - to give some human readable information on the program's purpose.

Declare that our program will take an argument (the input file with the fasta sequences) using the `add_argument()` function. This argument will be accessible in our program through (the dictionary key) `infile`. Add some help text describing the argument.

Using Python Modules : argparse

Step 2 : Tell the parser to analyze the command-line.

```
args=parser.parse_args()
```

The `parse_args()` function will:

- check whether the command-line matches the previously declared structure, and generate an error message if not.
- build a **dictionary-like structure** where the “keys” will be named after the arguments that were declared.

The resulting **dictionary-like structure** will be stored in the `args` variable.

In fact it's an object. More on that later

Step 3 : Use the **dictionary-like structure** to retrieve values of arguments passed on the command-line

```
print('The input file is: '+args.infile)
```


Adding Depth to Dictionaries

Until now, dictionary entries used only scalar types (strings) as element values. Often, for efficiency reasons, we want to access several chunks of information using a single key.

For instance, a sequence, identified by a fasta identifier, can be described by its nucleotide sequence, its amino acid sequence, their respective lengths, the GC-percent, the codon-usage frequencies and so on.

To handle such “records”, the dictionary value is itself a dictionary where the keys are the descriptors or attributes, and the values, the... values(!).

```
sequenceEntry = { 'nucleotides' : 'ATAGCGT...',  
                  'nucleotidelength' : 2562,  
                  'residues' : 'GIEDKD...',  
                  'residuelength' : 854,  
                  'gcpercent' : 61.0  
                }
```

```
sequenceInfo[sequenceId]=sequenceEntry
```

Adding Depth to Dictionaries

When using record-like structures as dictionary elements, keep in mind that:

- the descriptor names are arbitrary and subject to spelling inconsistencies between records.
- there is no guarantee that all descriptors are initialized for each record.

```
sequenceEntry = { 'nucleotydes' : 'ATAGCGT...',  
                  ...  
                  }  
sequenceInfo[sequenceId]=sequenceEntry  
for (id,info) in sequenceInfo.items() :  
    print(info['nucleotides'])
```

Will raise an error when processing the sequenceEntry with the typo.

Adding Depth to Dictionaries

Best practice 1 : Use “constant variables” for record descriptors instead of plain strings. By convention, constant variables are variables whose value *does not change* after initialization. They are written in uppercase.

Best practice 2 : Initialize all descriptors when creating a dictionary entry. For descriptors whose value cannot be determined at creation time, use the special `None` value.

```
NUCLEOTIDES_KEY='nucleotides'  
RESIDUES_KEY='residues'  
GCPERCENT_KEY='gcpercent'  
...  
sequenceEntry = { NUCLEOTIDES_KEY : 'ATAGCGT...',  
                  RESIDUES_KEY   : 'GIEDKD...',  
                  GCPERCENT_KEY  : None,  
                  ...  
                  }
```

Exercise 7

- Copy `readseq.py` to `src/ex07/readseq.py`
- Enhance `readseq.py` to use a record-like structure for storing the residues.
- Use “constant variables” to define and access record descriptors.

argparse : Arguments vs. Options

As seen before, program *arguments* are words following the command (script) name. Mapping of program arguments to variables in a Python script is done according to the position of the argument in the argument list. Arguments are mandatory.

Program *options* are composite : they include an option name (in short or long form) and an option value. Their relative positions on the command-line are not important. They can be optional(!) or required.

```
[mark@~] python3 readseq.py -n mynucsequences.fna -r \  
myaasequences.faa
```

```
[mark@~] python3 readseq.py --residues \  
mynucsequences.faa --nucleotides myaasequences.fna
```

argparse : Arguments vs. Options

In argparse, options are declared in the same way arguments are, with the following differences :

- The variable name *must* start with a dash (for the short form) or a double dash (for the long form)
- A boolean required parameter can be used to make an option mandatory (or optional)

```
import argparse
...
parser=argparse.ArgumentParser(description='Read sequences
from a multi-fasta file')
parser.add_argument('-n', '--nucleotides', help='multi-fasta
input file with nucleotide sequence', required=True)
```

argparse : Arguments vs. Options

`argparse` also handles the special case of *flags* : options without a value, whose mere presence on the command-line is enough. For example:

- the `-v` (or `--verbose`) flag to generate a lot of output on the program's progress
- the `-d` (or `--debug`) flag to run the program in debug mode.

Flags are declared as ordinary options, with the addition of a specific `action` named parameter to describe what to do when the option is present.

```
import argparse
...
parser=argparse.ArgumentParser(description='Read sequences
from a multi-fasta file')
parser.add_argument('-v', '--verbose', action='store_true')
args=parser.parse_args()
if args.verbose is True :
    print('Entering verbose mode')
```

Exercise 8

- Copy `readseq.py` to `src/ex08/readseq.py`
- Extend `readseq.py` to use two options :
 - an '-n' ('--nucleotides') option to specify the fasta input file containing the nucleotide sequences,
 - an '-r' ('--residues') option to specify the fasta input file containing the amino acid sequences.
 - read the two files, and store information about the two sequence types in a single record-structured dictionary
- Allow the use of a -v ('--verbose') option printing :
 - the total number of sequences read,
 - the number of sequences without an associated nucleotide sequence,
 - the number of sequences without an associated residue sequence.
- Run the program using files :
 - `'../..../data/fasta/Syn_RCC307.fna'` as nucleotide input file.
 - `'../..../data/fasta/Syn_RCC307.faa'` as amino acid input file.

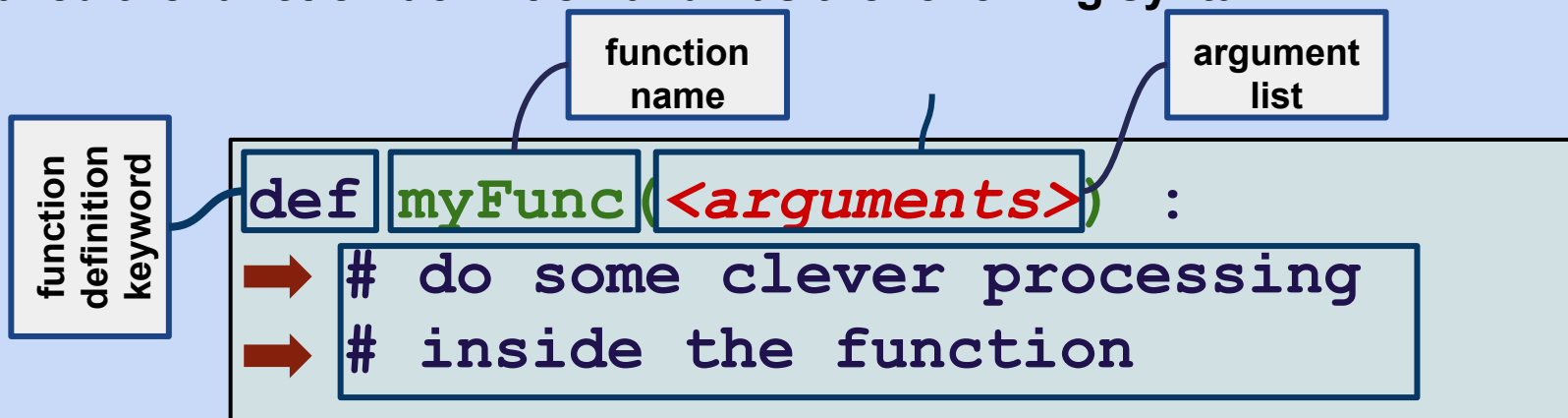
Organising Code in Functions: Definitions

The latest version of our script contains two code sections that are almost identical: they read a multi-fasta sequence file and store the result in a dictionary record structure.

Duplicating code is evil !

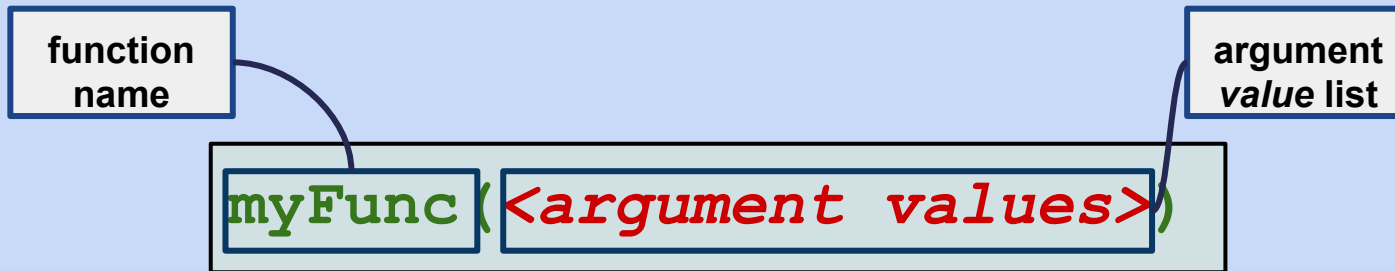
Python offers a construction that allows us to group instruction blocks that can be executed (called) at will later on. The execution can also be parameterized with arguments. Such a construction is called a *function*.

The location where the function is declared, with its arguments and its code is called the *function definition* and has the following syntax :



Organising Code in Functions: Calling a Function

The location(s) in the program where we want the function to be executed are called the *function calls*. The syntax of a function call is:



A function printing ten times “Hello” could be written as:

```
def tenTimesHello() :  
    for index in range(0,10) :  
        print('Hello')
```

And called with:

```
tenTimesHello()
```


Organising Code in Functions: Arguments

The use of arguments allows us to parameterize the function execution. Each argument in the argument will be given a (potentially different) value on each function call.

A function printing ten times the message given as argument could be written as:

```
def decaPrint(message) :  
    for index in range(0,10) :  
        print(message)
```

And called with:

```
decaPrint('Hello')
```

prints ten times 'Hello'

or with:

```
decaPrint('Goodbye')
```

prints ten times 'Goodbye'

Organising Code in Functions: Arguments

When defining a function, arguments may be given a default value. Arguments with a default value may be omitted from function calls.

A function printing a message given as first argument a number of times specified in the second argument, with a default value can be written as :

```
def spamPrinter(message, repeats = 10) :  
    for index in range(0, repeats) :  
        print(message)
```

And called with:

```
spamPrinter('Python Rulez', 100)
```

prints 100 times 'Python Rulez'

```
spamPrinter('Python is easy')
```

prints 10 times 'Python is easy'

prints 3 times 'Python is a snake'

```
spamPrinter('Python is a snake', repeats=3)
```

Organising Code in Functions: Return Value

Functions can return data to the caller on completion with the `return` statement :

```
def myFunc (<arguments>) :  
    result = None  
    # do some clever processing  
    # inside the function and  
    # store the result in the  
    # result variable  
    return result
```

When calling such a function, the result can be stored in a variable:

```
myFuncResult = myFunc (<arguments>)
```

Organising Code in Functions: Return Value

A function returning the sum of the list elements given as argument can be written as :

```
def sumList(values) :  
    total = 0  
    for element in values :  
        total = total + element  
    return total
```

And called with:

```
myListTotal = sumList([1, 2, 3, 4, 10, 20, 100, 2000])
```

stores the sum of 1,2,3,4,10,20,100,2000 in variable myListTotal

Functions can also modify the contents of their arguments. For arguments of scalar types (strings, numbers, booleans), the modification is kept local to the function block. *For arguments of container types, the modifications will persist after the function has returned.*

```
def myFunc(intArg, listArg) :  
    intArg = intArg + 10  
    listArg.extend(['A', 'Ton', 'of', 'Pie'])
```

```
intVal = 1  
listVal = ['I', 'Like', 'To', 'Eat']  
myFunc(intVal, listVal)  
print(intVal) # prints 1 : not changed outside myFunc.  
print(listVal) # prints ['I', 'Like', 'To', 'Eat', 'A',  
                  # 'Ton', 'of', 'Pie'] : change persists.
```

Exercise 9

- Copy `readseq.py` to `src/ex09/readseq.py`
- Enhance `readseq.py` to :
 - define a function capable of reading the contents of a multi-fasta file.
 - call the function for reading the nucleotide input file
 - call the function for reading the amino acid input file
- Run the program using files :
 - `'../..../data/fasta/Syn_RCC307.fna'` as nucleotide input file.
 - `'../..../data/fasta/Syn_RCC307.faa'` as amino acid input file.

A set of functionalities should typically be reusable across various scripts. Until now, our script file (`readseq.py`) contains a single useful function (`readFastaSequencesFromFile`), and the “main” code calling the function parameterized with command-line arguments.

If we want to reuse the `readFastaSequencesFromFile` function in other scripts, *without cutting & pasting its definition !!!*, we can store it in a *module*.

A module contains a collection of definitions (constants, classes) and declarations (functions). It should not contain any “main” code that will be directly executed when the module file is loaded.

In order to be able to import a module, it has to be located in one of the directories where the Python interpreter looks for modules.

With PyCharm, these directories have to be marked explicitly. This is done by right-clicking on the directory, and choosing *Mark Directory As -> Sources Root*.

A Python script may contain a mix of definitions (constants, functions, *classes*) and of instructions at the outermost scope. These instructions are executed whenever the script is loaded either to be run as a script or through an `import` instruction.

```
USEFUL_CONSTANT='useful value'
```

```
def usefulFunction(args):
```

```
    ...
```

```
usefulFunction('argval')
```

This function call gets executed whenever the script module is imported.

This may not be desired inside scripts that want to access functions defined in the script (using `import`) but don't want the code in the outermost scope to be run.

A special variable (`__name__`), maintained by the Python interpreter allows to check if a Python file is loaded as a script to be run or as a module.

In the former case, the value of `__name__` is `'__main__'` and the test can be written as:

```
USEFUL_CONSTANT='useful value'  
  
def usefulFunction(args):  
    ...  
  
if __name__ == '__main__':  
    usefulFunction('argval')
```

This function call gets *only* executed when the file is loaded as a script. Not when imported as a module.

It is a **highly** recommended practice to use the `__name__` based test in every Python file so as to promote the reuse of its contents.

Python programs and modules are written by people all over the world on various platforms. They do not always use the same character encoding standards. To lift any ambiguity regarding these standards, Python encourages the use of a specially formatted comment at the beginning of scripts and modules.

This comment looks like:

```
# -*- coding: utf-8 -*-
```

The name of the encoding standard

And will allow you to use any special character (most notable those with accents), either by directly typing them the script or module, or by printing strings read from a file and containing these characters.

On Linux/Unix systems, Python scripts can be run directly from the command-line (i. e. not as arguments given after the name of the python interpreter), provided the following two conditions are met:

1. They must be executable (does `chmod +x` ring a bell ?)
2. They must specify where to find the Python interpreter to run the contents of the script. This is done by putting a special comment as the first line of the script :

```
#!/usr/bin/env python3
```

This will run the `/usr/bin/env` command and tell it to look for a program named `python3`. That program will then be used to run the contents of the script.

The advantage is that this ensures that the script will be run by a Python 3.x interpreter but it lets the script's user configure her environment to select which version of the Python 3.x interpreter to use.

To sum up, this is what a well-behaved script or module should look like:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# lots of useful python code here:
# constants/variables
# functions classes
# classes

if __name__ == '__main__':

    # process arguments using argparse
    # use classes, functions and
    # constants/variables defined above
    # or imported from other modules.
```

The method we use to read data from sequence file has a major drawback: it loads the whole contents of the file at once in memory.

This does not scale well!

And is not suitable to figure in a decent module. Python provides an idiom to perform efficient line oriented data reading using the `with ... as ...` construction which can be used as follows :

```
with open('myfile.dat') as datafile :  
    for line in datafile :  
        # process the contents of line
```

The `with ... as ...` construction starts a new block. The (file) variable specified after the `as` keyword is usable inside this block. At the end of the block, the variable goes out of scope and the file is automatically closed.

The `for` loop reads the file *one line at a time* needing memory to store only a single line.

The success of a module significantly depends on its reusability, in which the documentation plays a major role.

The recommended way to document Python code is detailed in a Python Enhancement Proposal (PEP) :

`https://www.python.org/dev/peps/pep-0257/`

It relies on so-called *docstrings* : special blocks of formatted text delimited by three double-quote characters (""")

Docstrings should be present at different levels of Python modules :

- At the top of a module file, with a description of the module's purpose
- After each function declaration to explain how the function can be used

When docstrings span several lines, the first line is considered a summary of the docstring block.

Docstrings can be used by documentation generation tools (ex.: Sphinx) to generate(!) HTML or PDF versions of the documentation.

Modules: Documentation

Example of a module documentation:

The summary line

```
"""GreatModule: a Module to Achieve Great Things

The GreatModule contains a whole lot of useful functions
and classes that everyone should use.

"""
```

Example of a function documentation:

documentation keywords

```
def greatFunction(arg1, arg2=[], arg3=None) :
    """Perform some great function on data.

    :param arg1: the first argument
    :param arg2: the second argument (default value =
empty list)
    :param arg3: the third argument (default value =
None)
    :return: some very useful value.

    """
```

BTW: PyCharm generates docstring templates for functions.

Exercise 10

- Copy `readseq.py` to `src/ex10/readseq.py`
- Create a module named `sequencetools.py` containing the `readFastaSequencesFromFile` code, and the definitions it relies on.
- Enhance the code reading the sequence data from a file
- Add comments at the module and function level
- Modify the `readseq.py` script to make use of the `sequencetools` module.
- Configure PyCharm to define the `src/ex10/` directory as a source directory.

- Run the program using files :
 - `'../..../data/fasta/Syn_RCC307.fna'` as nucleotide input file.
 - `'../..../data/fasta/Syn_RCC307.faa'` as amino acid input file.

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. Intermezzo : Virtual Environments
6. A Word on XLS(X) Files : `openpyxl`
7. Grabbing Data From the Web : `requests` & `json`
8. Managing Configuration Files : `configparser`

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)

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 :  **number** - space - **letters** - space - **number**

 **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

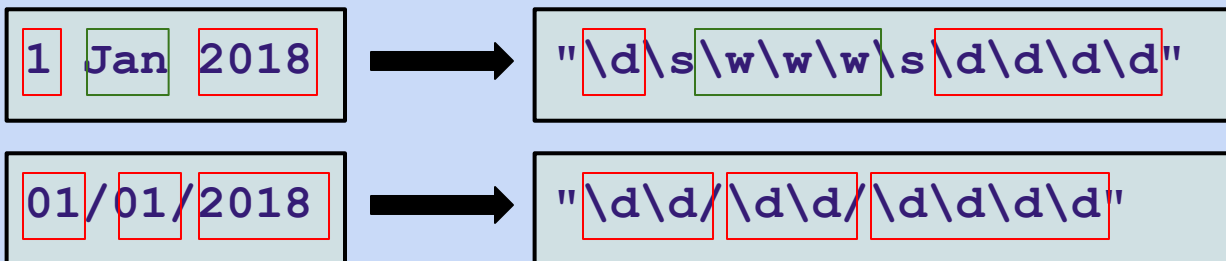
In Python, regular expressions are made available through the `re` module. This module provides, amongst other things, a set of special syntax elements to define patterns. A pattern is then an ordinary string containing one or more of these special syntax elements.

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 <i>a,e,i,o,u</i> characters.
[^aeiou]	Any character except <i>a,e,i,o,u</i>

In uppercase, they mean "anything except."

A date can then be defined with the following pattern :

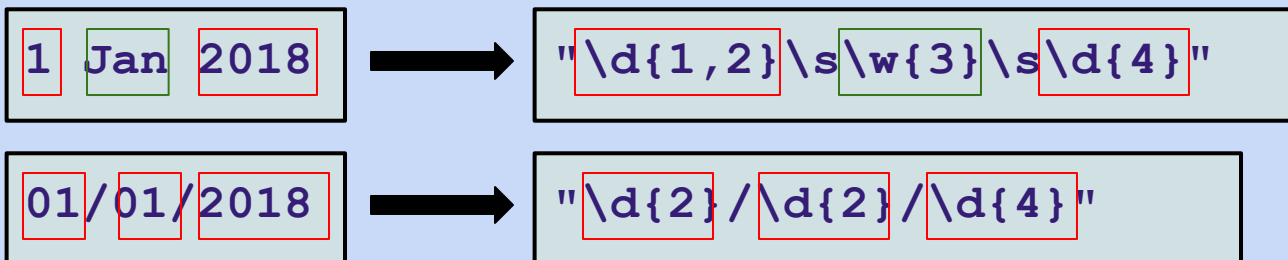


Regular Expressions : Patterns

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 <i>n</i> occurrences
{n,}	At least <i>n</i> occurrences
{,m}	At most <i>m</i> occurrences
{n,m}	Between <i>n</i> and <i>m</i> occurrences

The date patterns can be expressed as :



Regular Expressions : Patterns

The sequence patterns can be expressed as :

- DNA :

```
"[atgcATGC]+"
```

- Proteins :

```
"[mM] [ACDEFGHIKLMNOPQRSTUVWXYZ]+"
```

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}"
```

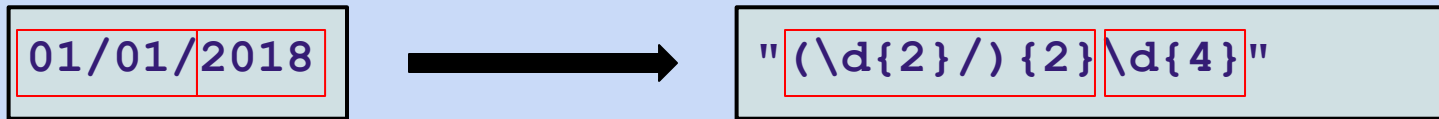
And to look for a date at the end of a line, we would use:

```
"\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")
```


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'
```

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.

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

Start position

Stop position

- 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']
```

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]
```

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]
```

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']]
```

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.

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.

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)
```

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') 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.

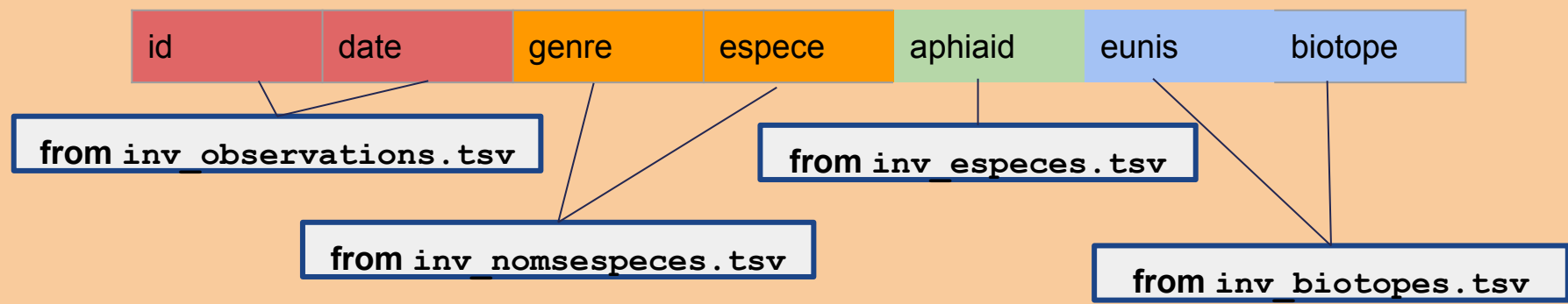
```
(...)
with open('fruit.csv') as csvfile :
    writer=csv.DictWriter(csvfile, fieldnames=fruit[0].keys())
    writer.writeheader()
    writer.writerows(fruit)
```

Exercise 15

- Create a new directory `src/ex15`
- Write a module called `inventairetools.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 `inventairetools.py`
 - displaying the contents of the first data line of the file.
- Run the program using the file :
`'../..../data/tabular/inv_biotopes.tsv'`

Exercise 16

- Look at the following data files in the `data/tabular` directory :
 - `inv_observations.tsv`, `inv_nomespecies.tsv`, `inv_especies.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_nomesspece`, `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 :



Exercise 16

- Copy both `inventairetools.py` and `loadinv.py` to a new directory `src/ex16`
- Rename `loadinv.py` to `summarizeobs.py`
- Extend the `inventairetools.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 `inventairetools.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.

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.

```

1 import argparse
2 import csv
3
4 import inventairetools
5
6 parser=argparse.ArgumentParser()
7 parser.add_argument("-b","--biotopes",required=True,help="Table with biotope records")
8 parser.add_argument("-s","--speciesnames",required=True,help="Table with species names")
9 parser.add_argument("-o","--observations",required=True,help="Table with observations")
10 parser.add_argument("-r","--resultfile",required=True,help="Table with the observations")
11 parser.add_argument("-s","--species",required=True,help="Table with species records")
12 parser.add_argument("-v","--verbose",action="store_true",help="Verbose mode: print")
13
14 args=parser.parse_args()
15
16 biotopesById=inventairetools.loadBiotopes(args.biotopes)
17 speciesNamesById=inventairetools.loadSpeciesNames(args.speciesnames)
18 observationsById=inventairetools.loadObservations(args.observations)
19 speciesById=inventairetools.loadSpecies(args.species)
20
21 observationSummaries=[]
22 for (observationId,observationRecord) in observationsById.items():
23     biotopeId=observationRecord['id_biotope']
24     eunis=eunis=observationRecord['code_eunis']
25     biotopes=""
26     if biotopeId in biotopesById:
27         biotopeRecord=biotopesById[biotopeId]
28         if 'code_eunis' in biotopeRecord:
29             eunis=biotopeRecord['code_eunis']
30         if 'description' in biotopeRecord:
31             biotopes=biotopeRecord['description']
32     else:
33         print("Biotope identifier not found: "+str(biotopeId))
34
35 genre=""
  
```

Available Packages

Package Name	Version	Description
0		
0.0.1		
00SMALINUX		
0fchanger		
02overcode		
080fneater		
Dedgobymum		
da10c-wm		
1		
1020-number		
117a-ragpat		
12factor-vault		
131228_pytest_1		
1337		
153957-theme		
157ve-django-ajax-selects		
17Manip		
18+		
199fx		
1and1		
1c-c08tes		
1d4folabs-test-script		
1ee		
1number		
1pass		

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
```

ScreenCast Time Again!

```
mhoebek@elc4-p236:~/PycharmProjects/managingdatawithpython
mhoebek@elc4-p236:~/PycharmProjects/managingdatawithpython$ pwd
/home/mhoebek/PycharmProjects/managingdatawithpython
mhoebek@elc4-p236:~/PycharmProjects/managingdatawithpython$ virtualenv -p python3 ndxprevnc11
Already using interpreter /usr/bin/python3
Using base prefix '/usr'
New python executable in /home/mhoebek/PycharmProjects/managingdatawithpython/ndxprevnc11/bin/python3
Also creating executable in /home/mhoebek/PycharmProjects/managingdatawithpython/ndxprevnc11/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
mhoebek@elc4-p236:~/PycharmProjects/managingdatawithpython$ ./ndxprevnc11/bin/activate
(ndxprevnc11) mhoebek@elc4-p236:~/PycharmProjects/managingdatawithpython$ which python
/home/mhoebek/PycharmProjects/managingdatawithpython/ndxprevnc11/bin/python
(ndxprevnc11) mhoebek@elc4-p236:~/PycharmProjects/managingdatawithpython$
```

Exercise 17

- **With PyCharm :**
 - **create a virtual environment, based on the `python3` interpreter, and with the name `abimsenv`**
 - **install the `openpyxl` 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?)**

Manipulating Excel (XLSX) files

In some cases, it is necessary to work with Excel (XLSX) files instead of text-based tabular data :

- When the Excel workbook contains multiple worksheets
- When the content of the table relies on formulas
- When it is necessary to keep the original data format (such as dates or times)

Python provides the `openpyxl` module to handle these Excel files.

Loading an Excel file is done as follows:

```
import openpyxl
...
workbook=openpyxl.load_workbook('filename.xlsx')
```

It becomes possible to access individual worksheets:

```
import openpyxl
...
activeworksheet=workbook.active           get the active worksheet
...
namedworksheet=workbook['sheetname']     get the worksheet with a given name
...
allsheetnames=workbook.sheetnames       get a list with all the sheet names
```


Manipulating Excel files

Individual cells can be accessed using an indexed notation, and their value can be read and/or changed with the `value` attribute (more on that later) :

```
a1cell=worksheet['A1']  
a1value=a1cell.value  
a1cell.value='Foobar'
```

`a1cell` is a variable describing the cell, not only its contents

Looping over rows can be done using the `iter_rows()` function. Each `row` returned by the function and then be used to loop over the cells :

```
for row in worksheet.iter_rows():  
    for cell in row :  
        cell.value=re.sub('UPMC', 'Sorb. Univ.', cell.value)
```

The limits of the area of the worksheet that contains data can be obtained with :

```
worksheet.max_row  
worksheet.max_column
```

Manipulating Excel files

A word of caution : when using the indexed notation, non-existing cells are automatically created. For ex. :

```
>>> (worksheet.max_row, worksheet.max_column)
(15, 7)
>>> worksheet['ZZ2048'].value='Hello, World'
>>> (worksheet.max_row, worksheet.max_column)
(2048, 702)
```

Saving a modified workbook is done with the `save()` function:

```
workbook.save('myworkbook.xlsx')
```

The `openpyxl` module provides a lot more features to handle formulas, styles, validation etc.. which are beyond the scope of this introduction. For more information :

<http://openpyxl.readthedocs.io/en/default/index.html>

More loop controls

With the `for` instruction, the loop block is executed once for each element of the collection.

Sometimes, it might be more efficient to directly “jump” to the next loop element without processing the remainder of the loop block. This can be done using the `continue` instruction :

```
for number in numbercollection :  
    if number % 2 == 1 :  
        continue  
# Process only even numbers in the remainder of the block.
```

jump to next element of collection

It might sometimes be useful to finish looping before all collection elements have been processed. In this case, the `break` instruction can be used.

```
for number in orderednumbercollection :  
    if number >= thresh :  
        break  
# Process number in the remainder of the block.  
# Code following the for block.
```

elements in `orderednumbercollection` are supposed to be ordered

jump right after the for block

More loop controls : using while

Sometimes, we need to carry out a series of steps (a block of code) repeatedly while a condition is satisfied. Python provides the `while` instruction to do this:

```
while expression :  
    # Process the following block while  
    # expression evaluates to True.
```

Almost always, *expression* contains terms whose value is modified inside the `while` block, potentially changing the outcome of the evaluation of *expression*.

```
basket={'total' : 0, 'contents' : []}  
while basket['total'] <= 100.0 :  
    someItem=selectItemFromStore()  
    basket['contents'].append(someItem)  
    basket['total']=basket['total']+someItem['price']  
# Code following the while block
```

whenever total exceeds 100,
execution resumes after the while
block

More loop controls : using `while`

There is however a Python idiom relying on a perpetual `while` and using the `break` instruction to end the `while` block :

```
while True :  
    outcome = perform_some_sophisticated_calculation()  
    if outcome == 'unexpected' :  
        break  
# Code following the while block
```

Exercise 18

- Create a new directory `src/ex18`
- Write a script called `extractaphiaids.py` using `openpyxl` and `csv`.
- Taking the following arguments :
 - `-i` or `--infile` followed by an XLSX filename
 - `-o` or `--outfile` followed by a CSV filename
 - `-w` or `--worksheet` followed by the name of the worksheet in the input file containing the data
 - `-a` or `--aphiaid` followed by the name (letter) of the column in the input file containing the Aphiaids
 - `-g` or `--genus` followed by the name (letter) of the column in the input file containing the genus
 - `-s` or `--species` followed by the name (letter) of the column in the input file containing the species
- Generating an output CSV file with the following columns extracted from the XLSX file : genus, species, aphiaid.
- Every aphiaid present in the XLSX file must appear only once in the CSV file.
- Run the program using `../../data/xlsx/observationsummaries.xlsx`

More and more, repositories allow *direct access* to datasets. If retrieving a small number of datasets by hand is possible, it quickly becomes important to be able to download large collections of files.

In order to do so, two key issues need to be addressed:

1. How did the repository design the dataset identifiers allowing each of them to have a unique reference ?

In technical terms: how to build the URL (a.k.a “web address”) referencing datasets of interest.

2. What is the format of the data that will be downloaded ?

It may be in tabular format, or other more structured formats (XML, JSON), or even plain HTML.

The download protocol (HTTP, FTP or other) is a minor issue because it is nicely handled by the various Python modules.

Fetching data from the Web : URLs

Each data repository has its own way of defining URLs for datasets. Methods for building these URLs are most of the time documented on the data supplier's website: search for API, or better REST API.

However, there always is:

- A constant part including the name of the server and the path to the “parent directory” where the datasets are made available :

<http://data.myrepository.org/rest/datasets/>

- A variable part, appended to the constant part, including an identifier that uniquely references a given dataset :

http://data.myrepository.org/rest/datasets/sequences/Syn_RCC307

http://data.myrepository.org/rest/datasets/taxons/Syn_RCC307

- There may also be optional parameters for specifying a data format to return:

http://data.myrepository.org/rest/datasets/sequences/Syn_RCC307?fmt=xml

Ex. 1: Retrieving taxon description records from WoRMS.

1. The base address is :

<http://www.marinespecies.org/rest/>

(BTW, this is also the page where the documentation is found)

2. It provides a *method* to retrieve taxon description record knowing an AphiaID :

[/AphiaRecordByAphiaID/{ID}](#)

3. The method contains a parameter between curly braces [{ID}](#) that has to be replaced by an actual value

(There is no mention of formatting options)

The complete URL to retrieve a taxon description record for the taxon identified by AphiaID 131173, is thus :

<http://www.marinespecies.org/rest/AphiaRecordByAphiaID/131173>

Ex. 2: Retrieving sequence information from EBI/ENA

The documentation on *programmatic access* for data retrieval is available at:

<https://www.ebi.ac.uk/ena/browse/data-retrieval-rest>

1. The base address is :

<https://www.ebi.ac.uk/ena/data/view>

2. Any ENA identifier or identifiers can be appended to the base address:

[/{ID1, ID2, ID3}](#)

3. ENA allows to specify an optional format parameter to choose how the return the data (xml,text,fasta) :

[?display=format](#)

The complete URL to retrieve the ENA record for the petB gene in WH8102, in text format (EMBL) is thus:

<https://www.ebi.ac.uk/ena/data/view/AAC05630?display=text>

Fetching data from the Web with requests

Python provides a `requests` module with the most user (programmer?) friendly functions to retrieve data from the web.

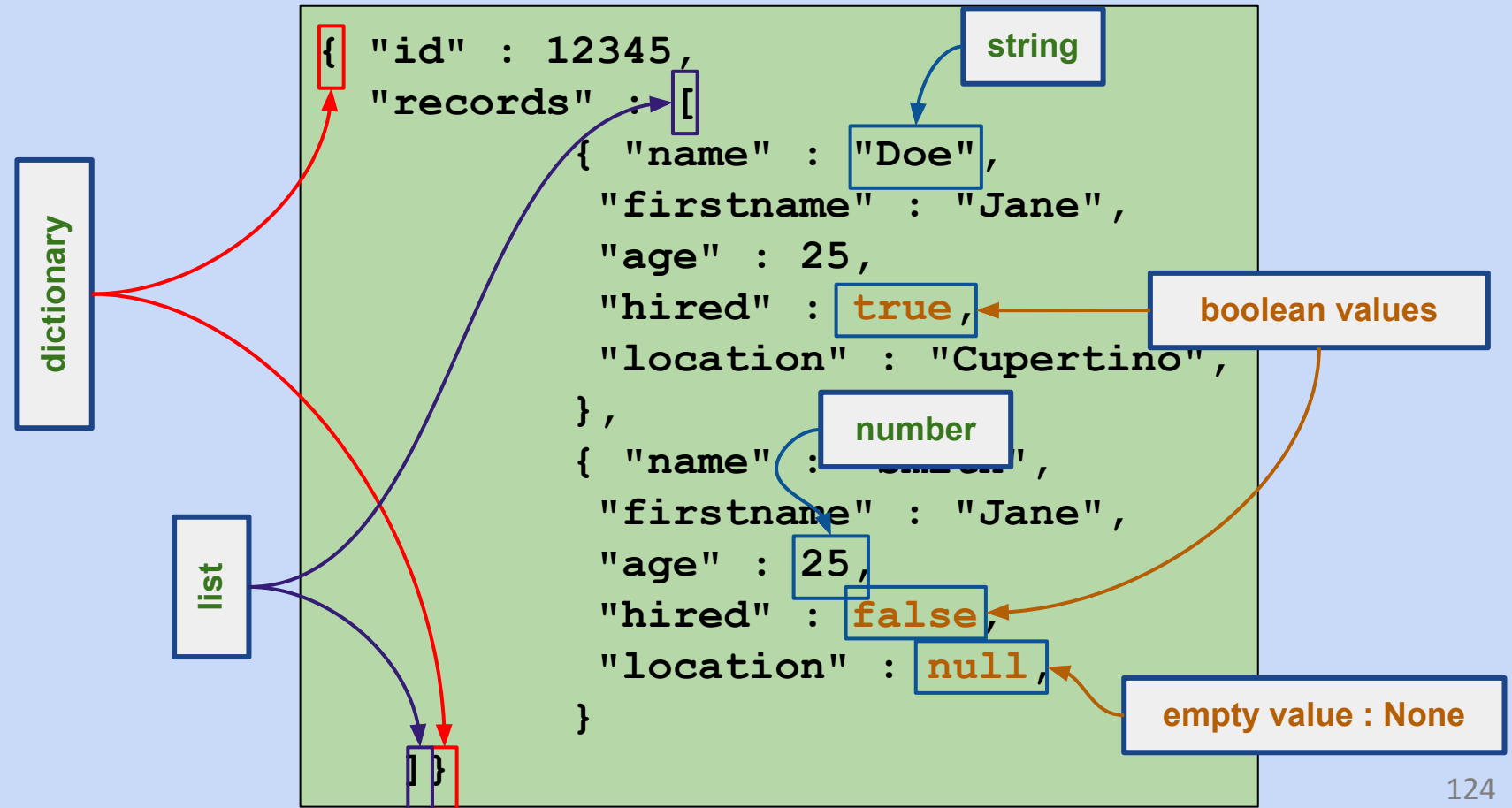
For a basic usage, `requests` includes a `get()` function where the only argument is the URL to use for data retrieval. This function returns an *object* (an enhanced data structure, more on objects later on) containing both the retrieved data itself and some metadata (the status of the request, the encoding of the returned data, the headers sent back from the server...).

The actual data can be accessed in various formats using one of the attributes or methods of the object : `raw`, `text`, `json()`

```
>>>import requests
...
>>>r=requests.get('https://www.ebi.ac.uk/ena/data/view/AAC05630&display=fasta')
>>>r.text
'>ENA|AAC05630|AAC05630.1 Synechococcus sp. WH 8103 partial cytochrome b6
\nTACGTGTTCCGGGTCTACCTCACCGGTGGTTTCAAGCGTCCCCGTGAGCTCACCTGGGTC\nACCGGCGTGACCATGGC
CGTGATCACAGTTTCTTCGGTGTACCCGGTTACTCCCTGCC\nTGGGACCAGGTTGGTTATTGGGCCGTCAAGATTGTT
TCCGGCGTCCCAGCAGCCATCCCA\nGTTGTGGGTGACTTCATGGTGGAGCTGCTCCGCGGTGGCGAAAGTGTCCGGTCAGT
CCACA\nCTCACTCGCTTCTACAGCCTCCACACCTTTGTGATGCCATGGCTGCTCGCCGTATTCATG\nCTCATGCACTTC
CTGATGATTCGGAAGCAGGGCATTCTGGTCCCTTGTGA\n'
```

Fetching data from the Web : JSON format

Among the variety of formats proposed by data suppliers, JSON is one of the most frequently used (with XML). It is a lightweight text based format suited for the representing structured data that can be described by dictionaries, lists and scalar values. Data descriptions in JSON look very similar to their Python counterparts:



The `json` module allows conversion between text-based JSON data structures and Python data structures. It is used in pretty much the same way as the `pickle` module.

Ex. 1: Writing a Python data structure into a JSON text file

```
import json
...
fruit=[...]
with open('fruit.json','w') as jsonfile :
    json.dump(fruit,jsonfile)
```

Ex. 2: Loading a Python data structure from a JSON text file

```
import json
...
fruit=[...]
with open('fruit.json') as jsonfile :
    fruit=json.load(jsonfile)
```

Fetching data from the Web : JSON format

The `json` module also handles coding/decoding data from or into text strings. This makes it unnecessary to use temporary files when retrieving JSON data from the web destined to be stored in Python variables.

Ex. 3: Converting a Python data structure into a JSON text string.

```
import json
...
fruit=[...]
jsonFruit=json.dumps(fruit)
```

Ex. 2: Building a Python data structure from a JSON text string

```
import json
...
jsonFruit=' [{"kind": "apples", "qt": 10}, ... ] '
fruit=json.loads(jsonfile)
```

Exercise 19

- If needed, add the `requests` module to your virtual environment.
- Create a new directory `src/ex19`
- Write a script called `gettaxinfo.py` using `requests`, `json` (and `csv`).
- Taking the following arguments :
 - `-i` or `--infile` followed by a CSV filename, with three columns:
`genus,species,aphiaid`
 - `-o` or `--outfile` followed by a CSV filename
- Retrieving an AphiaIDRecord from the WoRMS data repository, and storing classification descriptors included in the record:
`kingdom,phylum,class,order,family,genus,scientificname,authority`.
- Generating an output CSV file with the AphiaID and the previous columns.
- Run the program using `../../data/tabular/species_aphiaids.csv`

Using configparser to load configurations

For configuring scripts with complex arguments and/or options, using the command-line quickly becomes unwieldy. In those cases, the configuration is better stored in a file, which is then given as argument to the script. Various configuration file formats exist, but only a few are both human-readable (and editable) and easily parsable in a program. One of these formats has been popularized by Windows INI files. These are text files divided in named sections wherein basic assignments can be defined.

```
[firstsectionname]  
parameter name=parameter value  
boolparam=yes  
[secondsectionname]  
#comments are also allowed  
other parameter name : other parameter value  
empty parameter=  
other empty parameter  
multi line parameter : a lot of text spanning  
a block of lines
```

The diagram illustrates the INI file format with several callouts:

- section name delimiters**: Points to the square brackets around the section names.
- separators**: Points to the equals sign and colon characters used as separators.
- comment**: Points to the hash symbol (#) at the start of a line.
- Indentation for multi-line values**: Points to the indentation of the multi-line parameter value.

Using configparser to load configurations

The configuration format even allows simple variable interpolations to achieve more flexible configurations : a base parameter is defined once, and other parameters depending on the base parameter can reference the latter. The base parameter is the surrounded by an opening % (and a closing) s

ordinary definition

```
[customizablebynoob]
basedirectory=/home/user/applications/superapp

[customizablebyexpert]
inputdir=% (basedirectory) s/infiles
outputdir=% (basedirectory) s/results
tempdir=% (basedirectory) s/tmp
```

References to an existing definition

Using configparser to load configurations

In Python programs, these kind of configuration files can be easily read with the `configparser` module. It stores the contents of the file as a dictionary of dictionaries. The first key being the section name and the second key the parameter name.

```
>>>import configparser
>>>configparser=configparser.ConfigParser()
>>>configparser.read('config.ini')
>>>configparser[ firstsectionname ][ parameter name ]
'parameter value'
```

Section name Parameter name

The names of the sections can also be provided with the `sections()` function

```
>>>import configparser
>>>configparser=configparser.ConfigParser()
>>>configparser.read('config.ini')
>>>configparser.sections()
['firstsectionname', 'secondsectionname']
```

Using configparser to load configurations

By default, all parameter values are strings. If needed, they have to be explicitly converted to other scalar types. However, the module comes with a function, `getboolean()`, returning a boolean value after parsing a string whose values can be: 'yes', 'no', 'on', 'off', 'true', 'false', '1' or '0'.

```
>>>import configparser
>>>configparser=configparser.ConfigParser()
>>>configparser.read('config.ini')
>>>configparser['firstsectionname'].getboolean('boolparam')
True
```

Finally, a variable of “type” `configparser` can be written to a file with the `write()` method.

```
>>>import configparser
>>>configparser=configparser.ConfigParser()
>>>configparser['section one']={'param one':'value one',
                                'param two':'value two'}
>>>configparser.write('config.ini')
```

Exercise 20

- Create a new directory `src/ex20`
- Copy the `gettaxinfo.py`
- Enhance the script to make it capable of reading a configuration file with the following section :
 - `outfile`, declaring a series of boolean parameters :
`kingdom, phylum, class, order, family, genus`
When set to one of the possible truth values, the column will be generated in the output file otherwise it will be omitted.
- Add the `-c (--configfile)` option to the script loading the configuration from a file
- Add a `-w (--writeconfig)` option to the script writing the configuration to the given filename
- Define a default configuration dictionary in the script.
- Run the program using `.././data/tabular/species_aphiaids.csv` once to write the configuration file
- Edit the configuration file to change the columns in the output file and rerun the program with the configuration file as argument.