Managing Data with Python

Session 101

June 2018

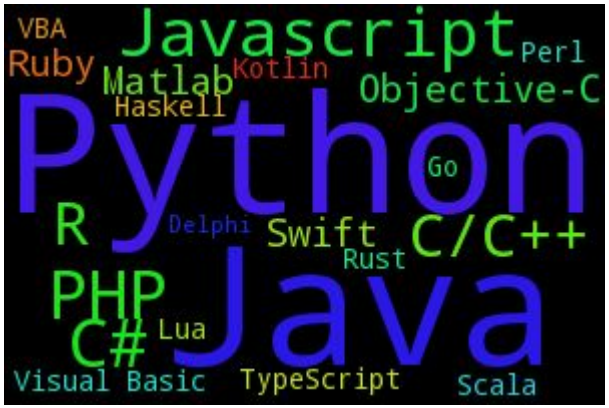
M. HOEBEKE
Ph. BORDRON
L. GUÉGUEN
G. LE CORGUILLÉ

Outline : Session 101

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

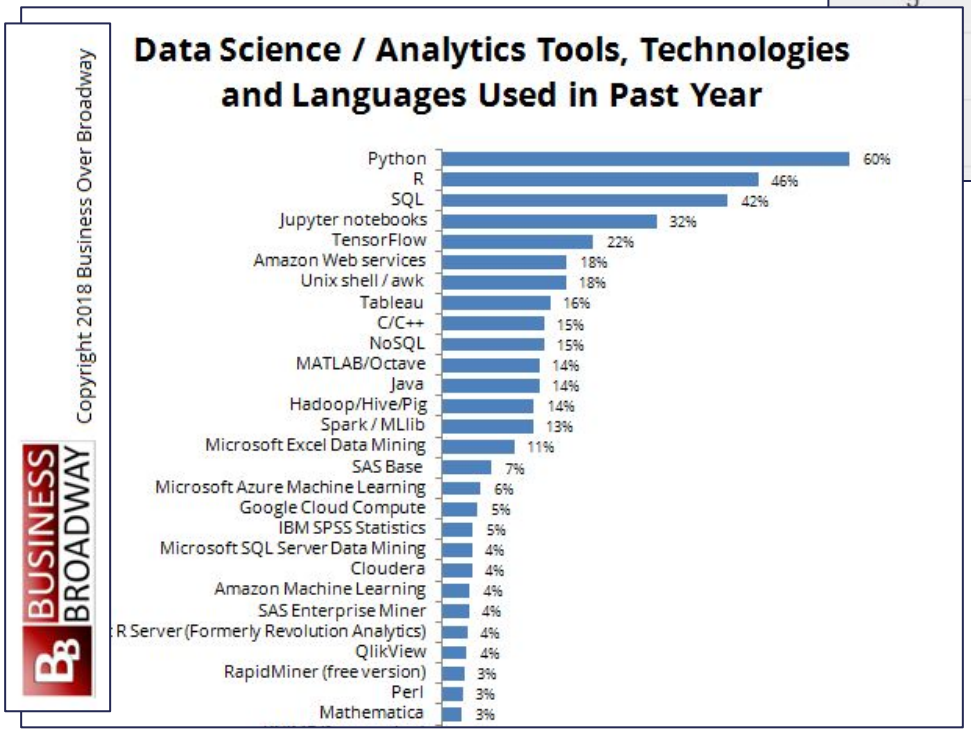
Why Python ?



PYPL

Worldwide, Jun 2018 compared to a year ago:

Rank	Change	Language	Share	Trend
1	↑	Python	23.04 %	+5.2 %
2	↓	Java	22.45 %	-0.6 %
3	↑↑	Javascript	8.6 %	+0.3 %
4	↓	PHP	8.21 %	-1.6 %
5	↓	C#	8.01 %	-0.4 %
		C/C++	6.15 %	-1.1 %
	↑	R	4.14 %	+0.1 %



© TIOBE

Jan 2018	Jan 2017	Change	Progra
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

Google
YAHOO!

© DocSity (www.docsity.com)

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)

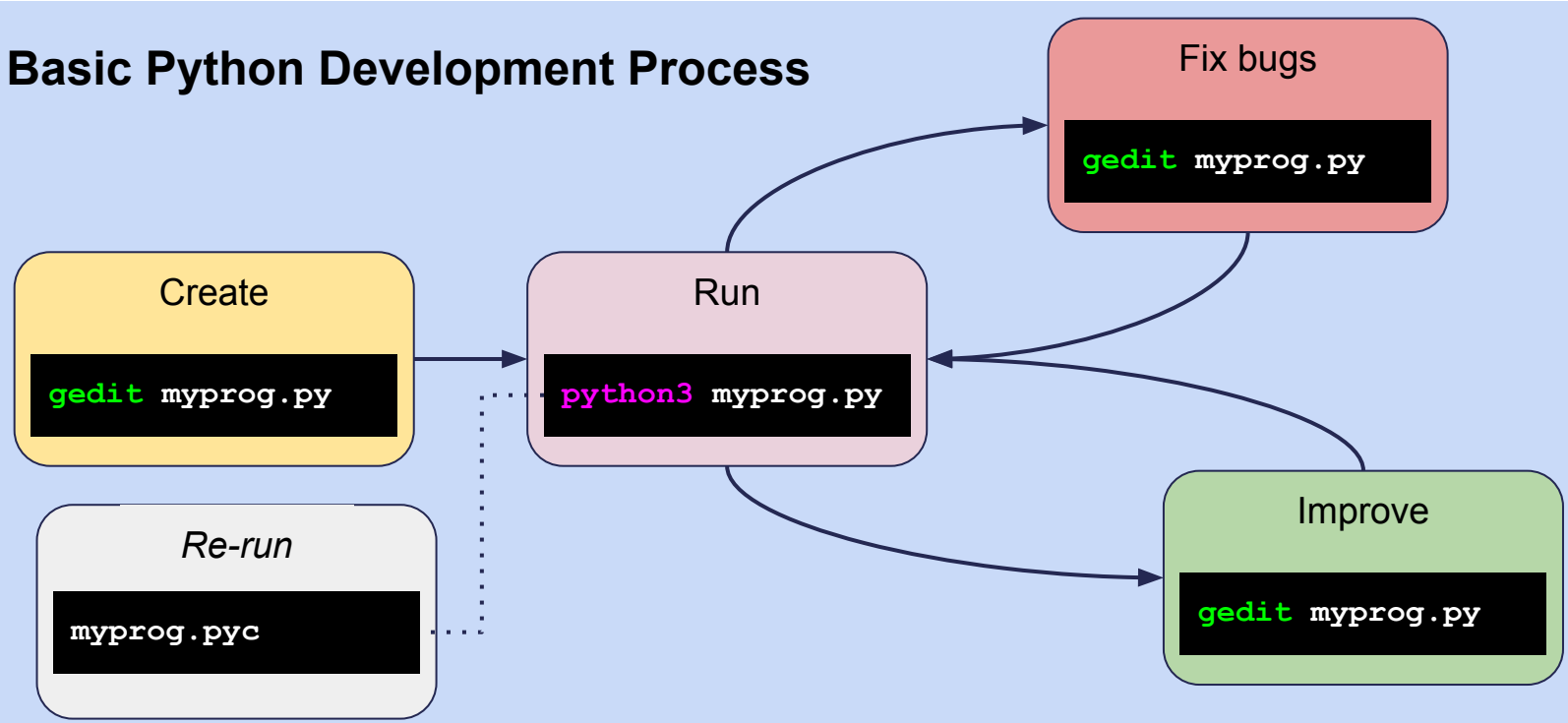
To check which version comes configured as default :

```
[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



Writing & Running Python Programs

Using Python in interactive mode :

- ★ **Done by running the interpreter without specifying à 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

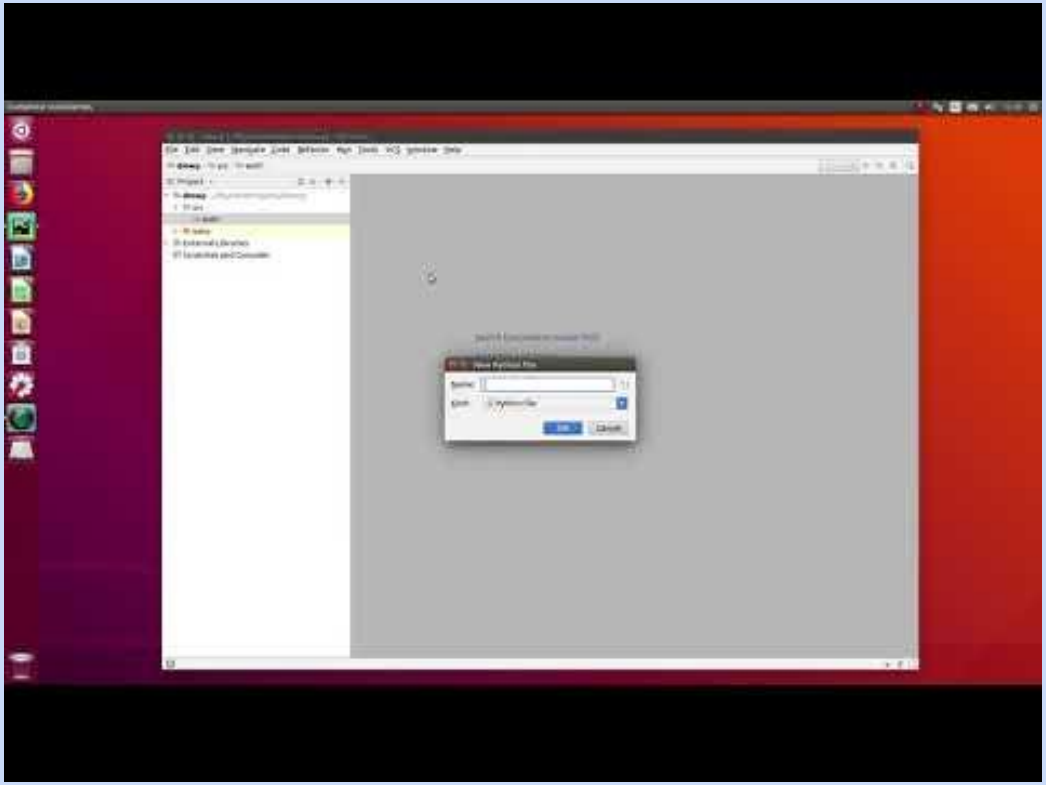
The result of the evaluation of the code I've typed

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/

Writing & Running Python Programs

Exercise 1

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

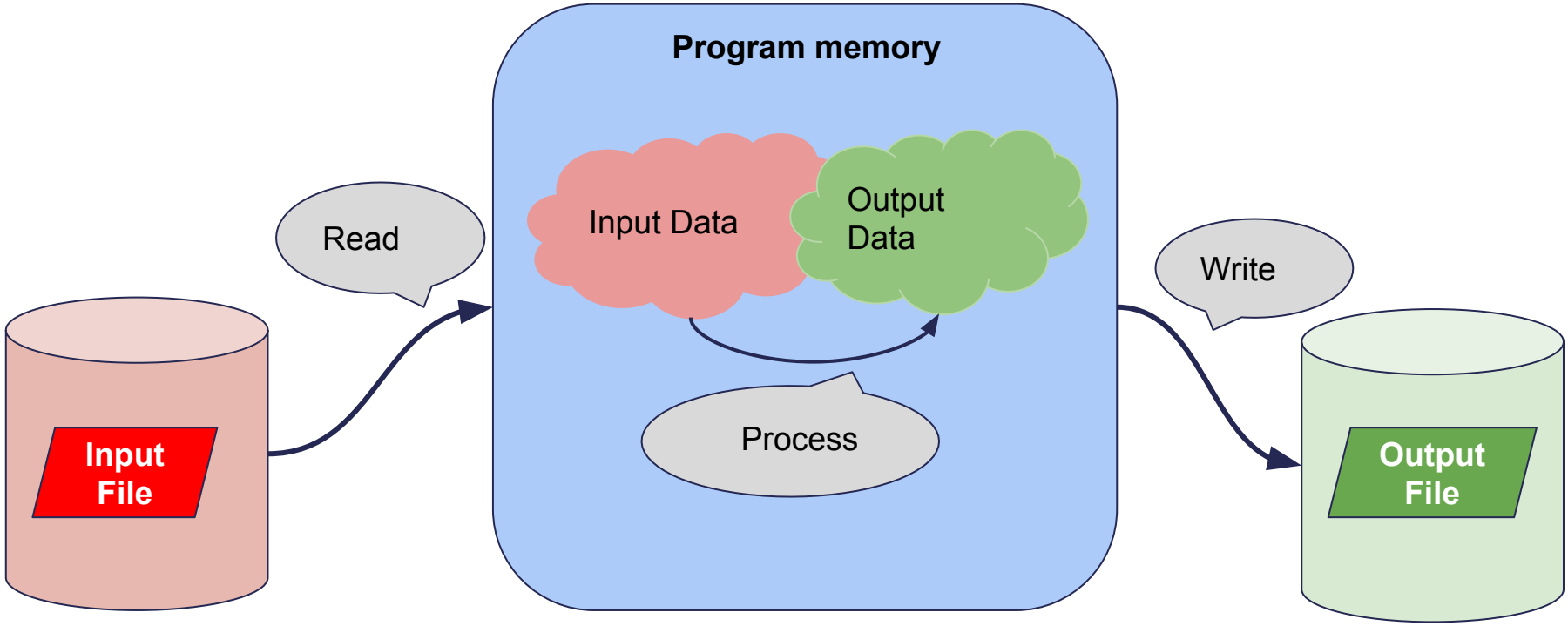
- the same project name : dmwp
- the same directory layout : src/ex01
- the same Python file (script) name : helloworld
- the script's contents :

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

Writing & Running *Useful* Python Programs

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)



Writing & Running *Useful* Python Programs

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.

line 1

line 2

```
>CK_Syn_RCC307_0001:174-1313:1|dnaN
MKLVCSQAELNGSLQLVSRAIAGRPTHPVLANVLVTADAAAGRISLTGFDLSLGIQTSFA
AVVESSGAITLPAKLFTDIVSRLPADGPLTLACPEGEEQTELSALTGSYQMRGLSAEDFP
DLPLAQNGQPLLLSGEAFAEGLRSTLFASSGDESKQILTGIHLKVEDGGLEFAATDGHRL
AVRRNGAGGQEGAESFAVTVPARSLRELERLLSARPSEESISLFCDRGQVVFLWADQVLT
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 ommitted:

- **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)
```

Working with Text Files

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

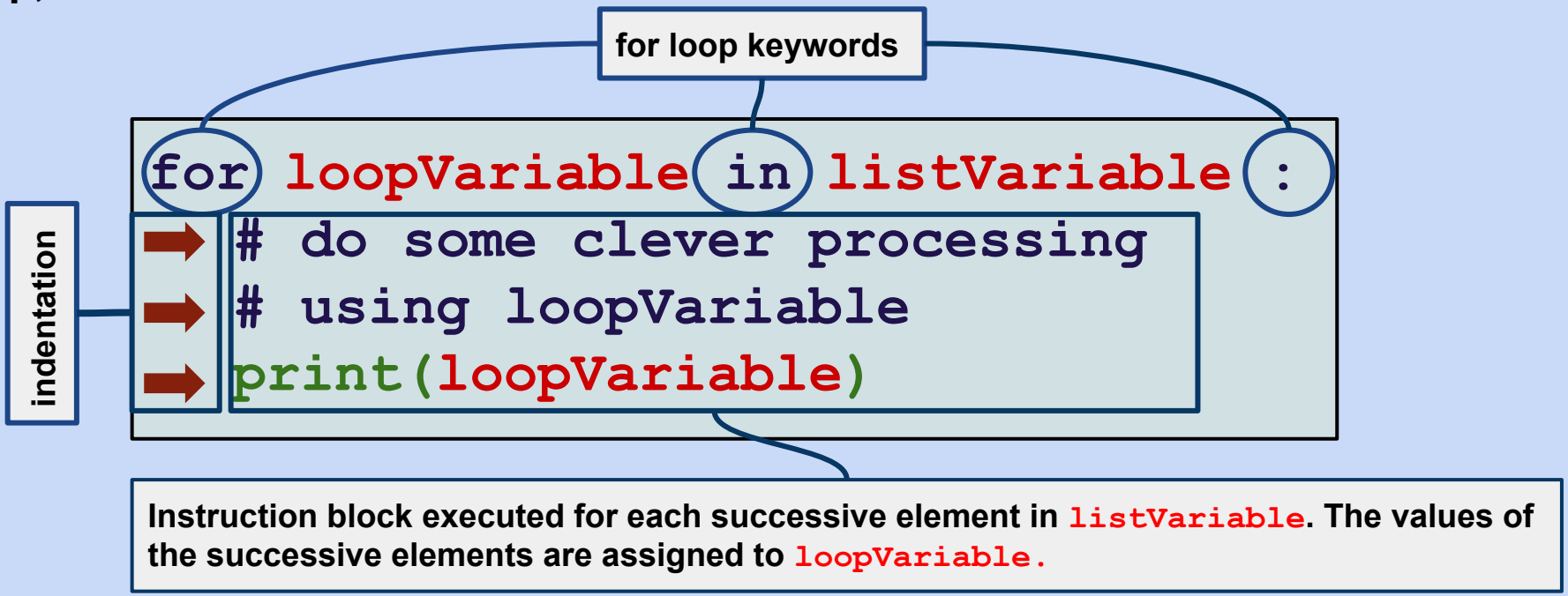
```
>>>fruitList = fruitString.split(', ')
>>>type(fruitList)
<class 'list'>
>>>fruitList
['apples','oranges','bananas']
```

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 :



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

Lists, Loops and Block Structures

A basic loop example :

```
>>>> for day in ('Saturday','Sunday') :
...       print('On '+day+' I can sleep late.')
...
On Saturday I can sleep late.
On Sunday I can sleep late.
```

Lists, Loops and Block Structures

An index based loop :

```
>>>> for index in range(0,10) :
...      print(index)
...
0
1
2
(...)
8
9
```

The `range(begin,end)` function generates integers from `begin` to `end` EXCLUDED

A First Loop

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'
```

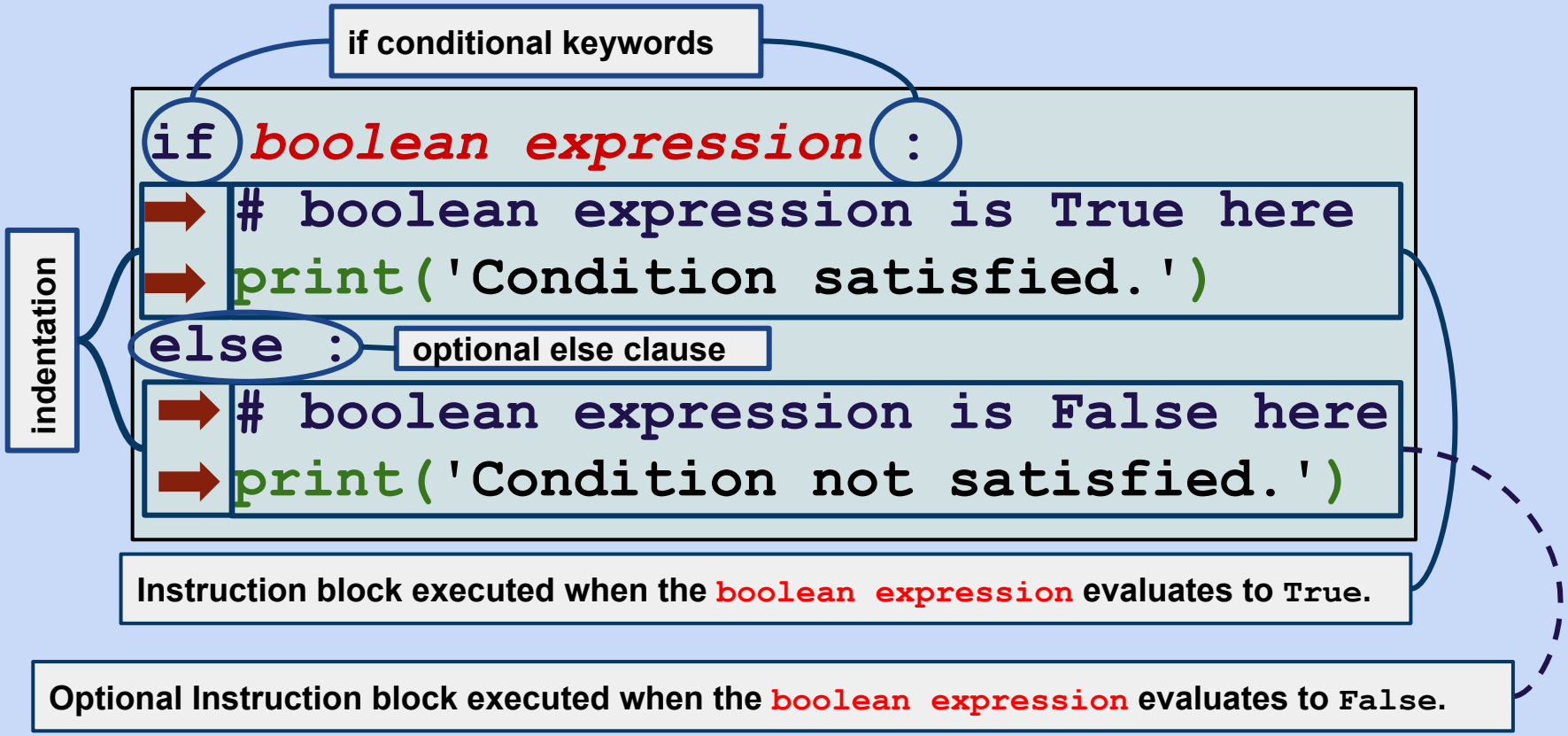
Then, using a `for` loop, concatenate the list elements to form a string where the words are separated with a space character.
After the loop has completed, print the value of the string, which should be :

```
'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.")
```

Conditionals and Boolean Expressions

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 conjuction 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

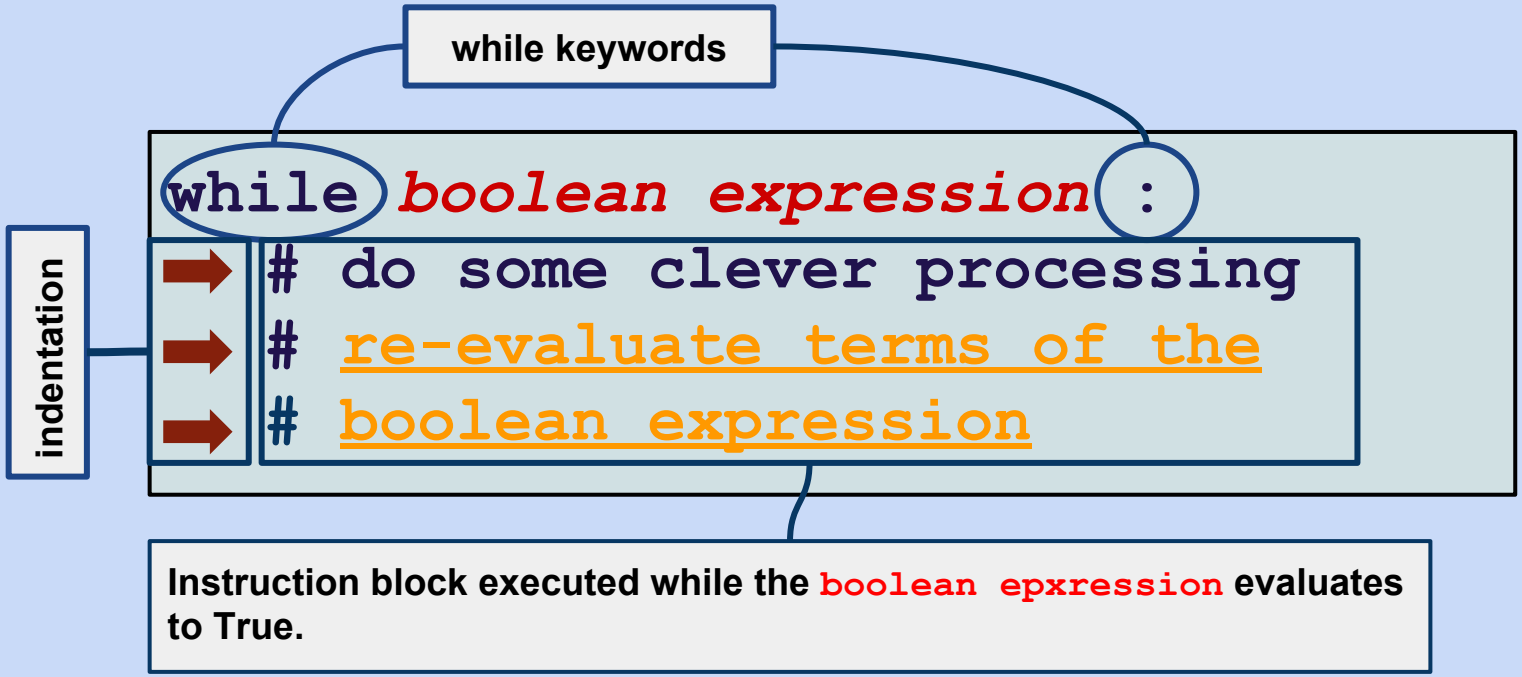
```
if ( distanceKm <= 1.0 and weather == 'Sunny' ) or weather == 'Mild' :
    print("You're better of walking.")
```

When the intended expression would be

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

Conditionals and Loop Structures

Repeatedly executing an instruction block while a condition is verified



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

Conditionals and Loop Structures

A basic while expression :

```
>>> LAPLENGTH=42.0
>>> totalDistance=0.0
>>> laps=0
>>> while totalDistance < 100.0 :
...     laps=laps+1
...     totalDistance=totalDistance+LAPLENGTH
...
>>> print(laps)
3
```

Choosing the Loop Structure

Use the for loop when the loop variable is generated from a collection (list, set, tuple, dictionary keys) :

=> the number of elements to loop over is known when starting the loop.

Use the while loop when the instruction block execution depends on the calculation of an expression :
=> the number of times the instruction block is executed is not known beforehand.

Jumping to the Next Iteration

The `continue` keyword causes execution skip the remainder of the current iteration to resume at the start of the next iteration :

```
for number in range(0,100000) :
    # Skip all even numbers
    if number % 2 == 0 :
        continue
    # Perform other compute intensive
    # checks for prime numbers
```

For even numbers, we can skip the remaining checks and start over with the next number.

Breaking out of a Loop Structure

The `break` keyword causes execution to resume immediately after the loop structure :

```
names=['Alice','Bob','Charlie',...]
found=False
for name in names :
    if name == 'Waldo' :
        found=True
        break

if found == True :
    print('I found Waldo!')
```

We don't need to carry on looking for Waldo anymore, and can exit the loop straightaway.

More loop controls : using `while`

There is even 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 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' :
'LSMAEQNSSSASLLLSALTGAAVGAAGLTWWLLSRAERRQALGDQFKRLGLNGAPTNGSSAQGSPENLEQKVNRLNL
AIEDVRRQLESMAPESSN*' }
```

Dictionaries : Basic Usage

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]
```

Dictionaries : Basic Usage

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.')
```

I'm (almost) a Bioinformatician now

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 documention matching your version of Python
https://docs.python.org/3/howto/argparse.html *is not* https://docs.python.org/2/howto/argparse.html

Using Python Modules : `argparse`

The `argparse` module provides all that's needed to make use of command-line arguments inside à 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.

Using Python Modules : argparse

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

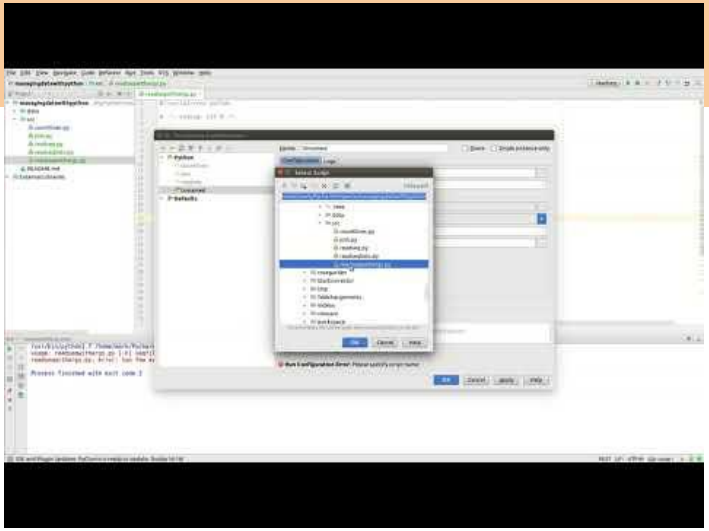
Step 3 : Use the dictionary-like structure to retrieve values of arguments passed on the command-line

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

Handling Command-Line Arguments

Exercise 6

- Copy `readseq.py` to `src/ex06/readseq.py`
- Change `readseq.py` to use the argparse module to handle a single command-line argument : the name of the file with the sequences.
- Use PyCharm to build three run-configurations for `readseq.py` :
  - A configuration with the already used sequence file as argument.
  - A configuration with no arguments (to assess that argument checking is done correctly by `argparse`)
  - A configuration with a `-h` argument (to check the help output that is automagically generated by `argparse`)



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 sequenceEntrywith 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,
                     ...
                }
```

Adding Depth to Dictionaries

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')
```

Handling Command-Line Arguments

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.

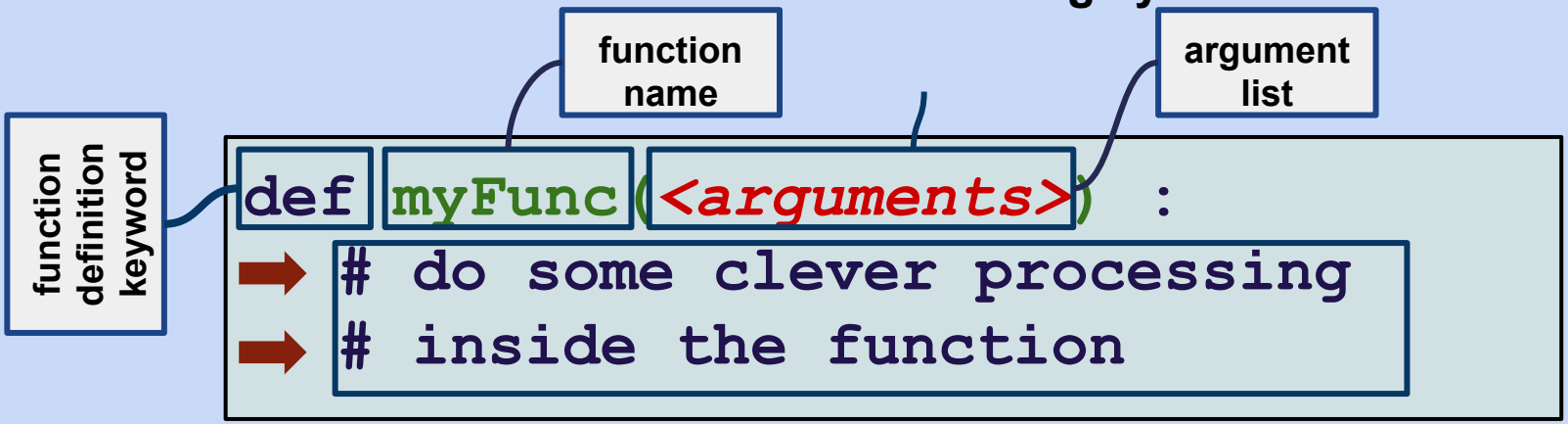
- Run the program using files :
  - `'../../data/fasta/Syn_RCC307.fna'` as nucleotide input file.
  - `'../../data/fasta/Syn_RCC307.faa'` as amino acid input file.

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

Organising Code in Functions: Argument Changes

Functions can also modify the contents of their arguments. For arguments are 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.

Organising Code : From Scripts to Modules

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.*

Organising Code : Making Scripts *importable*

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.

Organising Code : Making Scripts *importable*

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')
```

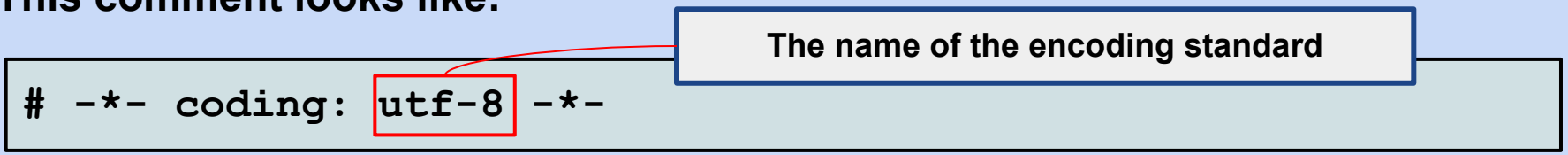
This function call gets *only* executed when the file is loaded as a script. Not when imported as a module.

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.

Organising Code : Making Scripts *portable*

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:



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.

Organising Code : Making Scripts executable

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.

Organising Code : A Script Template

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.
```

Modules: Efficiency Concerns

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.

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.