ABiMS

Managing Data with Python

Session 202

June 2018

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

Outline : Session 2

Using Domain Specific Modules

1. The BioPython Toolkit
2. Processing Sequence Data
3. Running BLAST and Processing Results
4. Accessing Remote Resources
5. Working with Graph Structures
6. Putting it all Together

Outline : Session 2

Using Domain Specific Modules

1. The BioPython Toolkit
2. Processing Sequence Data
3. Running BLAST and Processing Results
4. Accessing Remote Resources
5. Working with Graph Structures
6. Putting it all Together

The biopython toolkit

As the Python-savvy bioinformatics community gained momentum, the will to build a BioPerl-like toolkit in Python became prevalent.The first announcement of the availability of a BioPython set of modules goes back to 2000. The latest release is biopython-1.71 dating from April 2017.

The project's entry point is:

http://www.biopython.org

The bulk of BioPython's features cover biological sequences:

- Reading and writing sequences using standard file formats
- Manipulating sequence annotations
- Parsing output from ubiquitous bioinformatics tools (Blast, clustalw)
- Interacting with remote servers (NCBI-Blast, UniProt, ProSite)

But it also provides tools to work with protein 3D structures or with phylogenetics trees. Clustering functions are provided as well, but it remains to be determined where they fit in wrt to dedicated clustering packages like scikit-learn.

Unsurprisingly, BioPython is heavily object-oriented.

Outline : Session 2

Using Domain Specific Modules

1. The BioPython Toolkit
2. Processing Sequence Data
3. Running BLAST and Processing Results
4. Accessing Remote Resources
5. Working with Graph Structures
6. Putting it all Together

The biopython toolkit : reading sequences

Reading sequences in BioPython relies on `parse()` method of the `SeqIO` module, which needs two arguments: a filename with the sequence date, and a format name.
The complete list of recognized formats and their name to use in BioPython is available on the Wiki dedicated to SeqIO:

http://biopython.org/wiki/SeqIO

Most frequently used format identifiers for sequences are: fasta, genbank (or gb), embl, swiss. FastQ sequence files are also supported in various dialects: fastq (or fastq-sanger), fastq-solexa, fastq-illumina. GFF is not yet part of the BioPython release, more on this later.

The parse() method will return an *iterator* to loop over each entry in the file. Entries are `SeqRecord` objects.

```
import Bio.SeqIO
(…)
with open('mysequencedata.faa') as seqfile :
    for record in Bio.SeqIO.parse(seqfile,'fasta'):
        print(record.id)
```

The biopython toolkit : SeqRecord objects

The `Bio.SeqRecord.SeqRecord` class is *the* class for manipulating sequence objects. Its main attributes are :

- `SeqRecord.id` : the sequence identifier (a string)
- `SeqRecord.seq` : the actual sequence (a `Bio.Seq.Seq` object)
- `SeqRecord.annotations` : the sequence annotations (a Python dictionary, whose values are mainly strings)

More on these later

- `SeqRecord.features` : the sequence features (a list of `Bio.SeqFeature.SeqFeature` objects)

```
import Bio.SeqIO
(…)
for record in Bio.SeqIO.parse('mysequencedata.faa','fasta'):
    print(record.id)
```

The biopython toolkit : Seq objects

The `Bio.Seq.Seq` holds the actual sequence, as well as information about the sequence alphabet.
Moreover, it provides methods to extract sequence portions or to compute the reverse or reverse complement of a sequence. Depending on the nature of the sequence, translation or reverse transcription functions can also be used.
Finally, ungapped version of sequences can also be built.

```
import Bio.SeqIO
(…)
for record in Bio.SeqIO.parse('mysequencedata.faa','fasta'):
    seq=record.seq
    seqslice=seq[start:end]

    revseqslice=seqslice.reverse_complement()
```

Use array notation to extract sequence slice in a new Seq object

Build another Seq object by reverse complementing the already extracted slice.

The biopython toolkit : Seq objects

Bio.Seq.Seq instances can also be built from scratch. The constructor method takes as arguments:

- **A string with the sequence's letters**
- **The name of the alphabet necessary to define which type of sequence the object belongs to.**

The most frequently used alphabets are :

- **Bio.Alphabet.ProteinAlphabet**
- **Bio.Alphabet.DNAAlphabet**
- **Bio.Alphabet.RNAAlphabet**

The module also provides for degenerate or ambiguous alphabets.

Ex.: Building a new protein sequence.

```
import Bio.Seq.Seq
import Bio.Alphabet
(…)
newseq=Bio.Seq.Seq('ZZREZRLVLVPERLPSQSD',
                   Bio.Alphabet.ProteinAlphabet())
```

Using `biopython` in a custom module

Exercise 0

- Create a new virtual environment called `biopythonvenv` (based on Python 3.6)

- Install the `biopython` package in this virtual environment.

- Change your project settings in order to use this virtualenvironment instead of the previous one

Using `biopython` in a custom module

Exercise 8

- Create a new directory `src/ex208`
- Create a new file `fastadataextractor.py` defining a class FastaDataExtractor with :
  - A parseFile method parsing a (multi-)fasta file.
  - A getAllFastaIds method returning the alphabetically ordered list of identifiers read from the file.

- Use BioPython for reading fasta files.

- Create a new file `testfastadataextractor.py` defining a test case for the FastaDataExtractor.getAllFastaIds method.

- Run the tests using `../../data/fasta/cyanorak_complete.faa` (containing 35,993 sequences).

Using `biopython` in a custom module

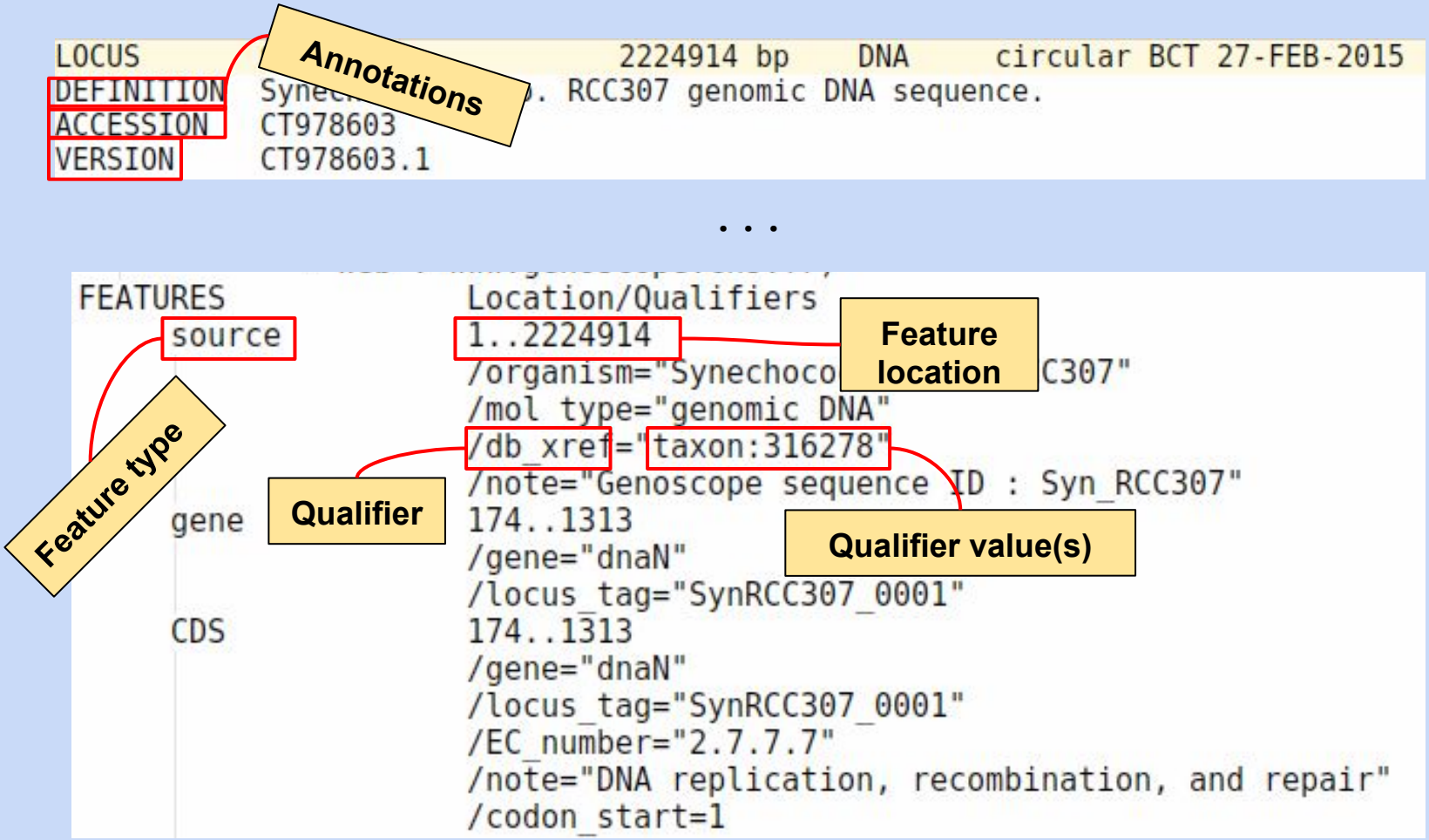
Exercise 9

- Copy the contents of directory `src/ex208` in directory `src/ex209`
- Enhance the `fastadataextractor.py` with method :
  - getSequenceWithId(self,seqId,minpos,maxpos), returning (if seqId is a valid identifier), the sequence slice between minpos and maxpos. If either minpos or maxpos are outside the sequence boundaries, an IndexError will be raised.

- Enhance the `testfastadataextractor.py` methods testing the getSequenceWithId method :
  - testGetSequenceWithIdWholeSequence for returning a whole sequence (the length of the first sequence should be 386)
  - testGetSequenceWithIdSlice for returning a portion of a sequence (with valid positions)
  - testGetSequenceWithIdOutsideBoundaries for returning a portion of a sequence (with invalid positions)
- Run the tests using `../../data/fasta/cyanorak_complete.faa` (containing 35,993 sequences).

The biopython toolkit : SeqFeature objects

Properties of sequence portions can be described through features. Not to be confused with annotations. The latter are pieces of informations related to the whole sequence (source organism, accession number, bibliographical references).



The biopython toolkit : SeqFeature objects

In BioPython, a `SeqRecord` stores features as a list of `SeqFeature` objects.

Each `SeqFeature` has a `type` and a `location` (start and end position, strand) and a dictionary of `qualifiers`. Each qualifier comes with a *list of values*. This is necessary because for a single feature, a qualifier may appear multiple times (the 'note' qualifier for instance).

```
import Bio.SeqIO
...
for record in Bio.SeqIO.parse('Syn_RCC307.gbk','genbank'):
    firstfeature=record.features[0]
    print(firstfeature.type)
    print(firstfeature.location)
    qualifiers=firstfeature.qualifiers
    print(qualifiers['organism'][0])
```

In GenBank files, the first feature is often the 'source' feature

```
source
[0:2224914](+)
Synechococcus sp. RCC307
```

The `biopython` toolkit : `FeatureLocation` objects

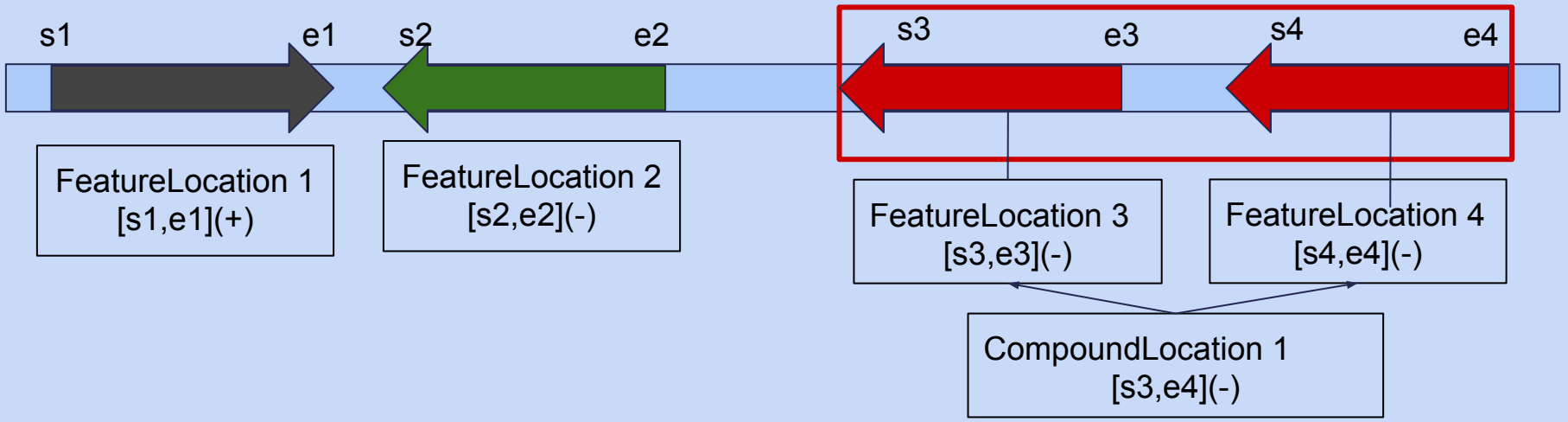
The two most important classes to describe feature locations are `FeatureLocation` and `CompoundFeatureLocation`.

The former represents a continuous stretch of a sequence between a start and an end position on one of the strands.

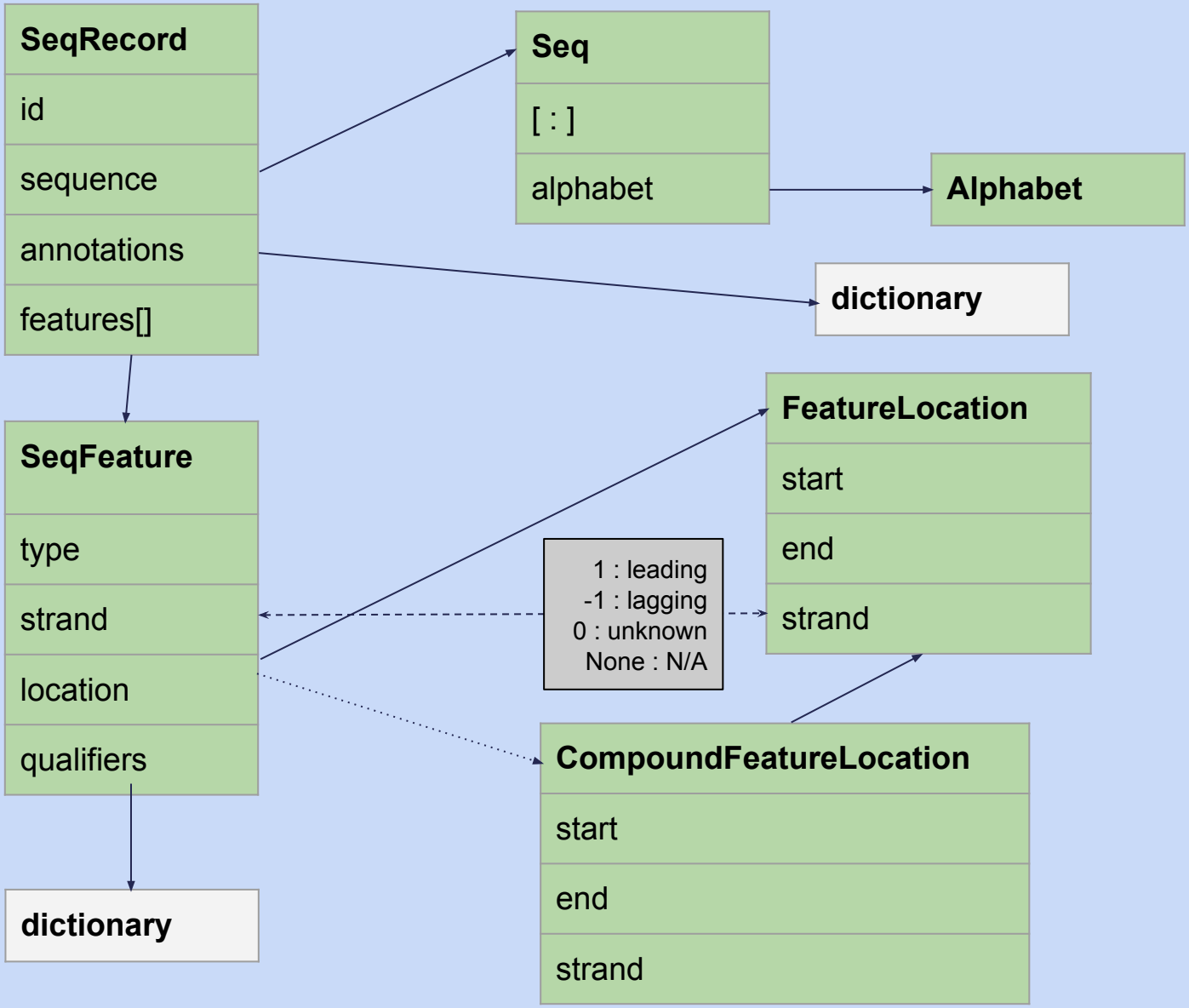
Warning : locations are indexed "à la" Python slices [0:max not included]

The latter can be used to store a discontinuous series of `FeatureLocation` objects (which is needed for instance to describe exons).

For location descriptions, regardless of the strand, the `start` attribute always stores the leftmost position and the `end` attribute always the rightmost.



The biopython toolkit: Class & data structure summary



The biopython toolkit : writing sequences

Writing sequences is a matter of calling the `write()` method of the `Bio.SeqIO` module. The arguments of `write()` are:

- A list of `SeqRecords`
- A filename or an already opened file variable
- A string specifying the file format.

```
import Bio.SeqIO
(…)
records=...
with open('mysequencedata.faa','w') as seqoutfile :
    Bio.SeqIO.write(records,seqoutfile,'fasta')
```

BioPython also provides a convenience method to convert files from one format to another :

```
import Bio.SeqIO
(…)
Bio.SeqIO.write('mysequencedata.gbk','genbank',
                'mysequencedata.faa','fasta')
```

Using `biopython` in a custom module

Exercise 10

- Create a new directory `src/ex210`
- Create a new file `genbankdataextractor.py` defining a class GenbankDataExtractor with :
  - A parseFile method parsing a (multi-)GenBank file.
  - A getFeaturesOfType(self,ftype,strand=None) method returning the ordered list of features whose type matches the ftype argument. If a strand argument is given, the set of returned features will be limited to those located on the strand.
- Use BioPython for reading genbank files.

- Create a new file `testgenbankdataextractor.py` defining a test case for the GenBankDataExtractor.getFeaturesOfType method :
  - A first test returning all the features of type 'gene'
  - A second test returning all the features of type 'CDS' on the leading strand.

- Run the tests using `../../data/genbank/Syn_RCC307.gbk` (containg 2,583 genes and 1,287 CDS on the leading strand).

Using `biopython` in a custom module

Exercise 11

- Copy the contents of directory `src/ex210` in directory `src/ex211`
- Enhance the `genbankdataextractor.py`
  - Add a qualifiers attribute and accessors to your feature class
  - Add method getFeaturesWithQualifier(self, qualifierName=None, qualifierValue=None), to the data extractor class, returning a list containing the features having qualifier as one of their qualifiers. If value is given, it must be one of the qualifier values.
- Enhance the `testgenbankdataextractor.py` methods testing the getFeaturesWithQualifier method :
  - testGetFeaturesWithQualifier with qualifier name 'locus_tag' and qualifier value 'SynRCC307_2134', which sould return two features.
  - testGetFeaturesWithQualifierHavingValue with qualifier value 'GOA:A5GVX9', which sould return a single feature of type CDS and positions 1842137 and 1842866.

- Run the tests using `../../data/genbank/Syn_RCC307.gbk`

The biopython toolkit : Using GFF files

The GFF file format is not yet included in the official BioPython release. However, one of the BioPython core developers has developed a GFF-dedicated module which is available separately.

The project's page is:

http://biopython.org/wiki/GFF_Parsing

Parsing a GFF file can be done using the same method as for other formats:

```
import BCBio.GFF
(…)
with open('myannotations.gff') as gffFile :
    for record in BCBio.GFF.parse(gffFile):
        print(record.id)
```

The biopython toolkit : Using GFF files

Parsing can be fine-tuned in several ways to avoid loading a complete GFF file at once.

1. **Limiting the set of features using a dictionary to define the identifiers/types of interest.**

The recognized dictionary keys are:

- **`'gff_id'`: the record identifier**
- **`'gff_source'`: the source description (second column)**
- **`'gff_type'`: the feature type (third column)**
- **`'gff_source_type'`: a combination (tuple) of source and type.**

Dictionary values are lists of strings (or tuples with two strings)

```
import BCBio.GFF
(…)
annotations={'gff_id': ['contig_1'],'gff_type' : ['CDS'],
                'gff_source_type' : [('Chromosome','CDS')]}
with open('myannotations.gff') as gffFile :
    for record in BCBio.GFF.parse(gffFile,limit_info=annotations):
        print(record.id)
```

The biopython toolkit : Using GFF files

2. Working with chunks of lines.

When specifying an additional parameter `target_lines` with a numerical value, the parser will process the file in blocks while preserving the GFF record structure : a record will not be loaded partially.

```
import BCBio.GFF
(…)
with open('myannotations.gff') as gffFile :
    for record in BCBio.GFF.parse(gffFile,target_lines=2000):
        print(record.id)
```

The biopython toolkit : Using GFF files

As GFF files contain only sequence annotations and no sequence, the GFF modules provides a way to associate a sequence to a GFF entry in order to have a complete `SeqRecord`.
To achieve this, it is necessary to build a dictionary from one or more `SeqRecord` objects with the `Bio.SeqIO.to_dict()` function. This dictionary can then be used as value for an additional argument for the `parse()` function.

```
import Bio.SeqIO
import BCBio.GFF
(…)

fastaseqdict=Bio.SeqIO.to_dict(Bio.SeqIO.parse('seq.faa','fasta')):

with open('myannotations.gff') as gffFile :
    for record in BCBio.GFF.parse(gffFile,base_dict=fastaseqdict):
        print(record.id)
```

The biopython toolkit : Using GFF files

It is also possible to save `SeqRecord` objects in GFF files using the `write()` function.

```
import BCBio.GFF
(…)

with open('myannotations.gff','w') as gffFile :
    BCBio.GFF.write(mySeqRecord,gffFile)
```

Outline : Session 2

Using Domain Specific Modules

1. The BioPython Toolkit
2. Processing Sequence Data
3. Running BLAST and Processing Results
4. Accessing Remote Resources
5. Working with Graph Structures
6. Putting it all Together

The biopython toolkit : running BLAST

BioPython is capable of running BLAST both locally and remotely at NCBI.

To run BLAST locally, you need :

- **The blast programs (blastn, blastp, blastall…)**
- **One or more databases formatted with makeblastdb**
- **Your candidate sequence(s)**
-

The BioPython local blast wrapper is available in the Bio.Blast.Applications module. There is a wrapper for the main BLAST types :

- **NcbiblastpCommandLine,**
- **NcbiblastnCommandLine,**
- **NcbiblastxCommandLine,**
- **NcbipsiplastCommandLine**

The specific type of BLAST program we want to run

```
from Bio.Blast.Applications import NcbiblastpCommandline

commandLine=NcbiblastpCommandline(query='file.fasta',
                                  db='mydb',
                                  outfmt=5,
                                  out='result.xml')

stdout,stderr=commandLine()
```

query sequence file

sequence database

result format

Result file

Weird syntax, granted !

The biopython toolkit : running BLAST

BioPython is capable of running BLAST both locally and remotely at NCBI.

To run BLAST remotely, you need :

- **The name of the blast program you want to use**
- **The name of the database you want to blast against**
- **Your candidate sequence(s)**

The BioPython remote NCBI wrapper is available in the Bio.Blast module. There is a single wrapper for all the blast types NCBIWWW. The input of the wrapper can be a SeqRecord instance. The output can be handled like a file variable opened for reading.

```
from Bio.Blast import NCBIWWW
import Bio.SeqIO
with open('candidate.fasta') as fastafile :
    records=Bio.SeqIO.parse(fastafile,'fasta')
    record=next(records)
    blastresult=NCBIWWW.qblast('blastp','nr',record.seq)
    with open('blastresult.xml','w') as resultfile :
        resultfile.write(blastresult.read())
```

Shortcut to get the next SeqRecord

Run remote blast

Write report to file

The `biopython` toolkit : parsing BLAST results

BioPython has tried to keep up with the various file formats in which blast reports can be generated : text, HTML, XML.

However, as text and HTML formats evolve with the blast or wwwblast versions without warning nor sufficient documentation, BioPython has decided to maintain only XML Blast parsing tools.

When planning to use BioPython for parsing XML output, the following option must be present on the (local) blast command-line: `-outfmt 5`

Blast report parsing uses the same principles as (annotated) sequence file parsing. The contents of the file is stored in a set of objects belonging to classes dedicated to specific parts of the alignment description. All these classes are described in the `Bio.Blast` module and submodules.

At the top level, parsing a blast report yields one or more `BlastRecord` objects

```
import Bio.Blast.NCBIXML
with open('blastreport.xml') as blastfile:
    for record in Bio.Blast.NCBIXML.parse(blastfile)
```

The biopython toolkit : parsing BLAST results

Blast report parsing uses the same principles as (annotated) sequence file parsing : the contents of the file is stored in a set of objects belonging to classes dedicated to specific parts of the alignment description.

All these classes are described in the `Bio.Blast` module and submodules.

At the top level, parsing a blast report yields one `Record` object per query sequence.

```
import Bio.Blast.NCBIXML

with open('blastreport.xml') as blastfile:
    for record in Bio.Blast.NCBIXML.parse(blastfile)
```

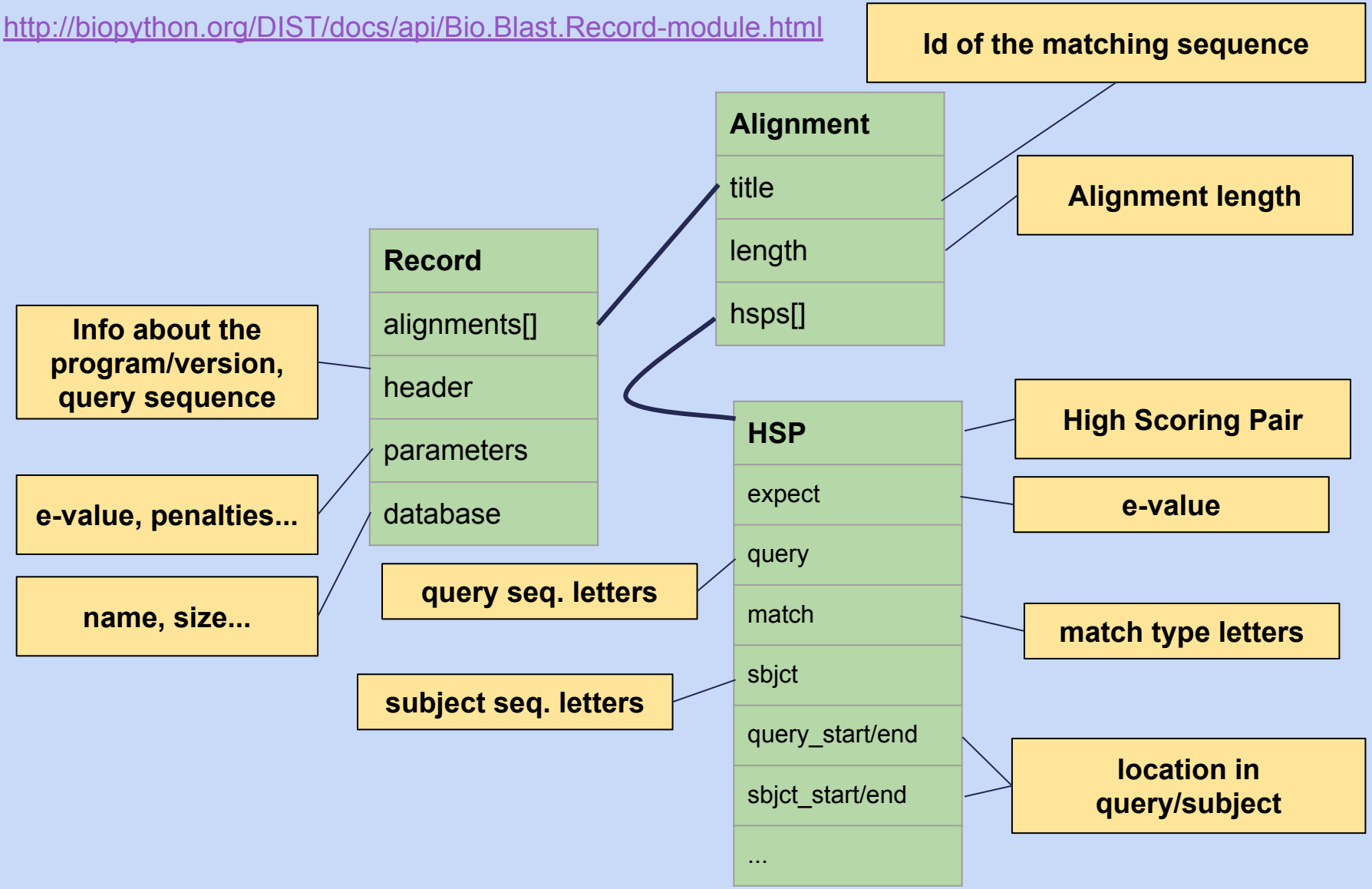
Module providing Blast XML parsing functions

return one Record at a time

The biopython toolkit : parsing BLAST results

A Diagram of BLAST result related classes with their main properties

http://biopython.org/DIST/docs/api/Bio.Blast.Record-module.html



The biopython toolkit : parsing BLAST results

Printing information about alignments and HSPs

```
import Bio.Blast.NCBIXML
with open('localblastreport.xml') as blastfile:
    for record in Bio.Blast.NCBIXML.parse(blastfile)
        for alignment in record.alignments:
            for hsp in alignment.hsps:
                if hsp.expect < MAX_EXPECT:
                    print(alignment.title)
                    print(alignment.length)
                    print(hsp.expect)
                    print(hsp.query[0:50])
                    print(hsp.match[0:50])
                    print(hsp.sbjct[0:50])
```

Exercise 12 : Converting blast results to CSV

- Create a new directory `src/ex212`
- Create a new file `blast2csv.py` defining a class Blast2Csv with :
  - A constructor taking as additional arguments : a blastfile and a csvfile with the names of the input and output files.
  - A generateCsvFromBlast(self) method extracting information from the blast report an generating a CSV row for each HSP with a set of default fields : queryname, subjectname, qstart (start of the HSP in the query), qend (end of the HSP in the query).

- Create a new file `testblast2csv.py` defining a test case for the Csv.generateCsvFromBlast method :
  - A test counting the lines of the CSV file after conversion (you can generate the CSV file as `../../tmp/localblastresult.csv`)

- Run the tests using `../../data/blast/localblastresult.xml` (the resulting file should contain 20 lines).

Using `biopython` in a custom module

Exercise 13 : Converting blast results to CSV with filters

- Copy the files from directory `src/ex212` to directory `src/ex213`
- Enhance the Blast2Csv class to allow output filtering according to a minimum alignment length percent with respect to the query sequence :
  - Define a new attribute (minalignpercent) and its accessor
  - Enhance the generateCsvFromBlast(self) method to write only HSPs exceeding the minalignpercent threshold
  - Add the alignment length and percent to the output columns.

- Add a new method to `testblast2csv.py` to test the new functionality:
  - Generate only rows for alignments where the size of the alignment exceeds the size of the subject (minalignpercent >= 1.0)

- A test counting the lines of the CSV file after conversion (you can generate the CSV file as `../../tmp/localblastresultwiththreshold.csv`)

- Run the tests using `../../data/blast/localblastresult.xml` (the resulting file should contain 7 lines).

BioPython wrap up

BioPython could be the subject of several training sessions considering the wide area of topics it covers.

For (beginning) Pythonistas eager to make use of this set of libraries, two valuable entry points are:

- **The top level page of the documentation Wiki with links to the cookbook and tutorials covering the main BioPython classes :**

  http://biopython.org/wiki/Documentation

- **The top level page of the BioPython API documentation. This set of pages is automatically generated from the BioPython docstrings embedded in the code :**

  http://biopython.org/DIST/docs/api/

Outline : Session 2

Using Domain Specific Modules

1. The BioPython Toolkit
2. Processing Sequence Data
3. Running BLAST and Processing Results
4. Accessing Remote Resources
5. Working with Graph Structures
6. Putting it all Together

Fetching data from the Web

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

Fetching data from the Web : URLs

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`

Fetching data from the Web : URLs

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.

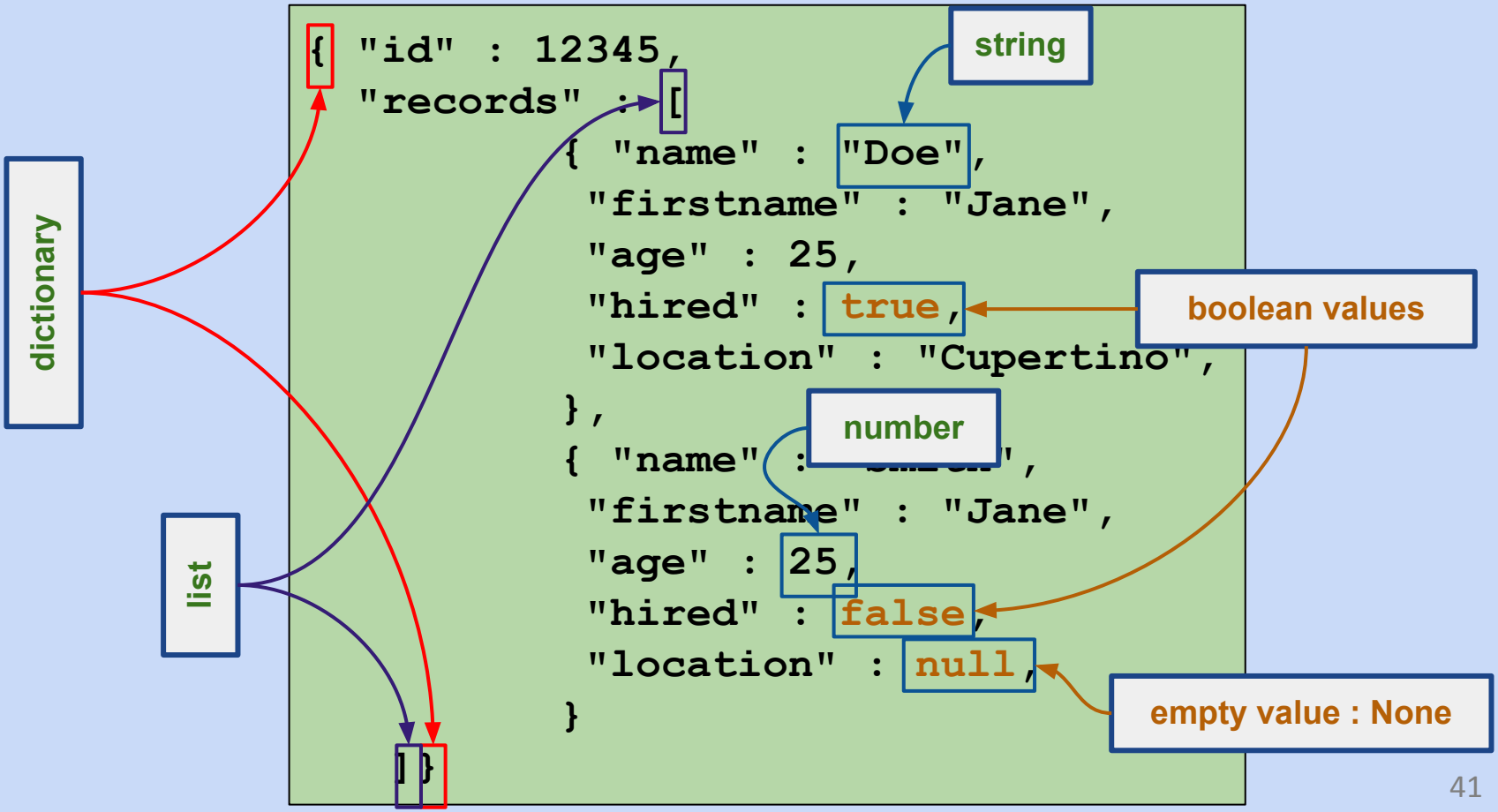
For a basic usage, `requests` includes a `get()` function where the only argument is the URL to use for data retrieval. This function returns an *object* (an enhanced data structure, more on objects later on) containing both the retrieved data itself and some metadata (the status of the request, the 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
CGTGATCACAGTTTCCTTCGGTGTCACCGGTTACTCCCTGCCC\nTGGGACCAGGTTGGTTATTGGGCCGTCAAGATTGTT
TCCGGCGTCCCAGCAGCCATCCCA\nGTTGTGGGTGACTTCATGGTGGAGCTGCTCCGCGGTGGCGAAAGTGTCGGTCAGT
CCACA\nCTCACTCGCTTCTACAGCCTCCACACCTTTGTGATGCCATGGCTGCTCGCCGTATTCATG\nCTCATGCACTTC
CTGATGATTCGGAAGCAGGGCATTTCTGGTCCCTTGTGA\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:



Fetching data from the Web : JSON format

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)
```

Fetching data from the Web : JSON format

Most well behaved web accessible resources will be able return data in several formats : text, HTML, XML, JSON.

However, JSON may not be the default format. It is then needed to explicitly define that we want to retrieve JSON formatted data.

This is achieved by setting a specific "header" to that end, when making the request :

```
import requests
…
result=requests.get(url,headers={'Accept' :'application/json'})
```

Exercise 14 : Remotely retrieving Taxonomic Information

- Create a new directory `src/ex214`
- Create a new file `taxoninforetriever.py` defining a class TaxonInfoRetriever with :
  - A getTaxonInfoForAphiaId(self,aphiaId) method retrieving a taxonomic decription record from Worms and returning a dictionary with a subset of fields (AphiaID, kingdom, phylum, class, order, family, genus, scientificname, authority).

- Create a new file `testtaxoninforetriever.py` defining a test case for the getTaxonInfoRetriever method.

- Run the tests using aphia ID 130714, and check that the kingdom is 'Animalia', the phylum is 'Annelida' and the genus is 'Polygordius'

- Remember, the URL pattern looks like :

  http://www.marinespecies.org/rest/AphiaRecordByAphiaID/*{aphiaId}*

Fetching data from the Web : EBI-Search

The EBI provides a significant amount of reference data through Web Services (i.e. : web servers that deliver machine-readable data instead of human-readable web pages).

The entry point is called EBI-Search :

https://www.ebi.ac.uk/ebisearch

And summary documentation on how to use the search is available at:

https://www.ebi.ac.uk/ebisearch/swagger.ebi

All web-service URLs will start with :

https://www.ebi.ac.uk/ebisearch/ws/rest/

Fetching data from the Web : EBI-Search

EBI-Search is based on the concept of domains. Roughly, each "database" to which access is provided represents a domain, 151 in total as of June 2018.

Among these domains, are : GO (genome ontology), Interpro (Interpro domains), PFam (protein families), Enzymes…

When performing a request, the actual domain name must be added to the URL

https://www.ebi.ac.uk/ebisearch/ws/rest/{domain}

Then, to retrieve information about a specific entry of this domain, the URL is completed as follows:

https://www.ebi.ac.uk/ebisearch/ws/rest/{domain}/entry{entryId}

If the entry exists, this returns a short document assessing the validity of the entry.

Try for yourself :

https://www.ebi.ac.uk/ebisearch/ws/rest/go/entry/GO:1990103

Fetching data from the Web : EBI-Search

To have more extensive information about an entry, it is necessary to specify which pieces of data (or attributes) we want to retrieve.

Again, this is done by completing the URL as follows :

https://www.ebi.ac.uk/ebisearch/ws/rest/{domain}/entry{entryId}?fields={f1,f2,..}

The actual list of fields can be found by inspecting the domain description document available at :

https://www.ebi.ac.uk/ebisearch/ws/rest/{domain}

This document describes each field in a block looking like :

```
<fieldInfo id="..." name="fieldname">
    <options>
    …
    </options
</fieldInfo>
```

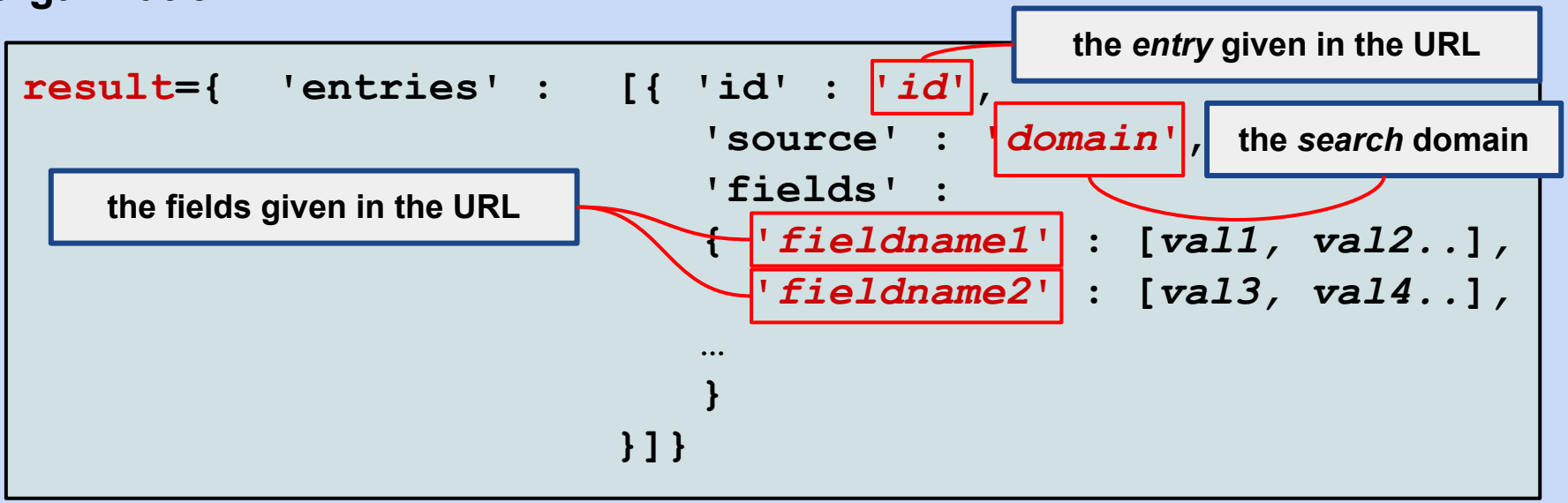
The name of the field that can be used to retrieve information

Fetching data from the Web : EBI-Search

A valid EBI-Search URL for retrieving the name, description and type of a GO-Term would be :

https://www.ebi.ac.uk/ebisearch/ws/rest/go/entry/GO:1990103?fields=name,description

The structure of the query result can be seen as a dictionary with the following organization :



Fetching data from the Web : EBI-Search

The result from the example URL :

https://www.ebi.ac.uk/ebisearch/ws/rest/go/entry/GO:1990103?fields=name,description

Would be :

```
{'entries' : [{'id': 'GO:1990103',
               'source': 'go',
               'fields': {'name': ['DnaA-HU complex'],
                          'description': ['A (...) oriC.']}}]
```

Exercise 15 : Remotely retrieving Interpro data

- Create a new directory `src/ex215`
- Create a new file `interproinforetriever.py` defining a class InterproInfoRetriever with :
  - A getInterproInfo(self,interprold,fields) method retrieving information about the given interpro entry. The information includes the values of the fields given as argument.
  - Default fields are : name, description, FO

- Create a new file `testinterproinforetriever.py` defining a test case for the getInterproInfoRetriever method.

- Run the tests using aphia ID IPR000850, and check that the result contains one entry with three elements for the GO field.

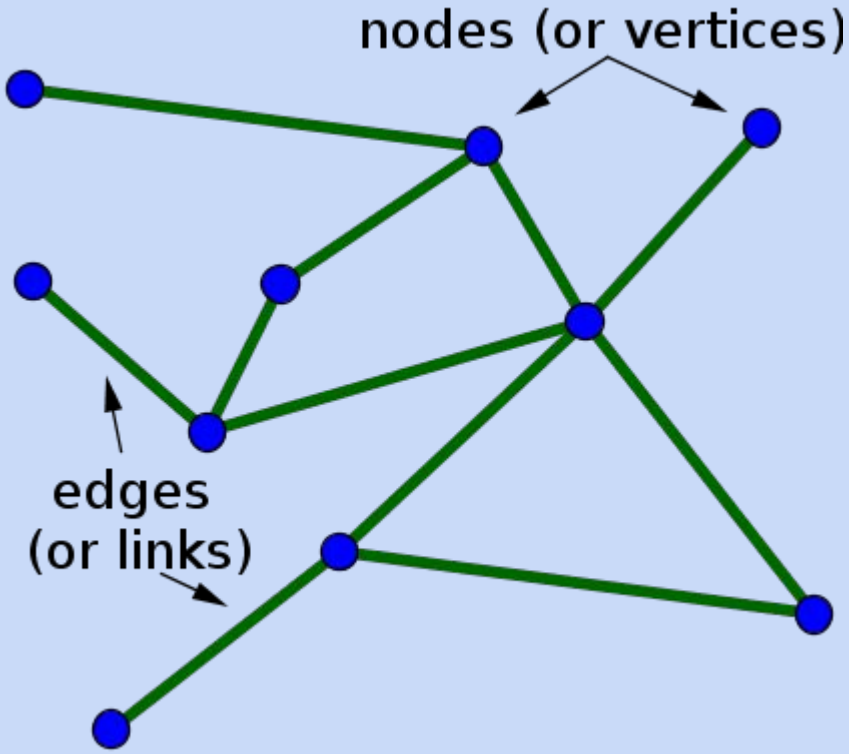
Outline : Session 2

Using Domain Specific Modules

1. The BioPython Toolkit
2. Processing Sequence Data
3. Running BLAST and Processing Results
4. Accessing Remote Resources
5. Working with Graph Structures
6. Putting it all Together

Working with graph data: `networkx`

The most widely used Python toolkit to handle graph data is undoubtedly `networkx`. It provides a relatively intuitive means to manipulate graph elements: nodes (or vertices) and edges (or links). Both of which can contain a rich set of attributes.



Working with graph data: networkx

A graph can be created using the `Graph()` constructor method :

```
import networkx as nx

simpleGraph=nx.Graph()
```

By default, graphs are undirected and support only one edge between two nodes. To build these special type of graphs, specific constructors have to be used:

```
import networkx as nx

directedGraph=nx.DiGraph()
```

Edges are directed from source to destination nodes

```
multiGraph=nx.MultiGraph()
```

There can be more than one edge between two nodes

```
multiDirectedGraph=nx.MultiDiGraph()
```

You guessed it: a combo of the above

Working with graph data: networkx

Graph nodes can be of any *hashable* type (a type having some kind of id that does not change over the lifetime of an object):

- Scalar types are hashable types, whereas mutable collections such as lists or sets are not.

- Objects of user defined classes are hashable but they all have a different id, even when all their attributes have identical values.

  While it is possible to use objects as graph nodes it is not recommended. A best practice is to use string representations of objects instead, and to keep a separate dictionary of objects indexed by the string representation or to add objects as node attributes.

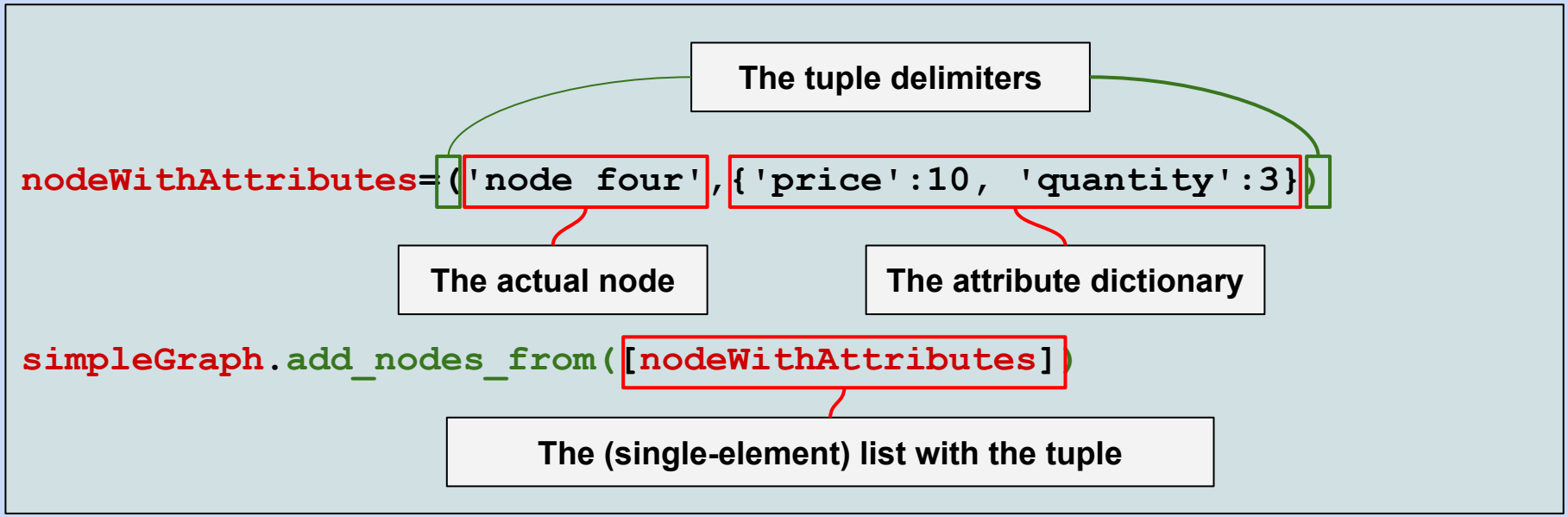
To add one or several nodes to a graph, the add_node() or add_nodes_from() methods can be used:

```
simpleGraph=nx.Graph()
simpleGraph.add_node('node one')
simpleGraph.add_nodes_from(['node two','node three'])
```

Any iterable collection can be used

Working with graph data: networkx

It is also possible to add nodes together with (a dictionary of) attributes. In order to do so, the container must contain(!) tuples with two elements: the node, and a dictionary of attributes :



Working with graph data: networkx

Similarly, edges are added with the `add_edge()` or `add_egdes_from()` methods. If the nodes of the edge or edge list do not already exist in the graph, they are automatically added. Just as with nodes, edges can also have an attribute dictionary which can be added at the same time as an edge.

```
simpleGraph.add_edge('node one','node five')

edgeWithAttributes=('node six','node seven', {'throughput' : 20})

simpleGraph.add_edges_from([edgeWithAttributes])
```

Already part of the graph

Automatically added to the graph

Edge endpoints

Edge attributes

Working with graph data: networkx

Adding attributes a posteriori is also possible. Both for nodes with the `nodes` attribute, And for edges with the `edges` attribute. These attributes can also be used to return information about a node (resp. edge):

```
>>>simpleGraph=nx.Graph()
>>>simpleGraph.add_node('node one')
```

Addition of the 'colour' attribute to an existing node

```
>>>simpleGraph.nodes['node one']['colour']='white'
>>>simpleGraph.nodes['node one']
{'colour':'white'}
```

```
>>>simpleGraph=nx.Graph()
>>>simpleGraph.add_edge('Roscoff','Brest')
```

Addition of the 'distance' attribute to an existing edge

```
>>>simpleGraph.edges['Roscoff']['Brest']['distance']=67
>>>simpleGraph.edges['Roscoff']['Brest']
{'distance':67}
```

Working with graph data: networkx

Retrieving the list of nodes and edges is also straightforward using these attributes:

```
>>>simpleGraph=nx.Graph()
>>>simpleGraph.add_node('node one')
```

Explicit list conversion is needed here

```
>>>list(simpleGraph.nodes)
['node one']
```

```
>>>simpleGraph=nx.Graph()
>>>simpleGraph.add_edge('Roscoff','Brest')

>>>list(simpleGraph.edges)
[('Roscoff','Brest')]
```

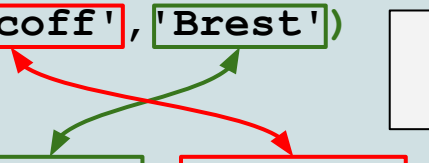
A list of tuples with two elements each: the source and destination nodes of the edge

Working with graph data: networkx

Finally, removing graph components is done through the remove_node(), remove_nodes_from() and remove_edge(), remove_edges_from() methods.

```
>>>simpleGraph=nx.Graph()
>>>simpleGraph.add_node('node one')
>>>list(simpleGraph.nodes)
['node one']
>>>simpleGraph.remove_node('node one')
>>>list(simpleGraph.nodes)
[]
```

```
>>>simpleGraph=nx.Graph()
>>>simpleGraph.add_edge('Roscoff','Brest')
>>>list(simpleGraph.edges)
[('Roscoff','Brest')]
>>>simpleGraph.remove_edge('Brest','Roscoff')
>>>list(simpleGraph.edges)
[]
```



Works because the graph is undirected

Working with graph data

Exercise 16

- Create a new directory `src/ex216`
- Create a new file `graphbuilder.py` defining a class GraphBuilder with :
  - A buildGraphFromCsv(self,filename,srccolumn,destcolumn) reading a directed graph from a CSV file, assuming source nodes are in the column named srccolumn, and destination nodes are in the column destcolumn
  - All information in other columns will be stored as a dictionary where keys are column headers and values cell values. The list of dictionaries of each edge will be stored in the edge's 'info' attribute.
  - An extractSubGraphWithSourceNode(self,srcnode) method returning a subgraph with all the edges of the complete graph having srcnode as source. Associated attributes will also be copied to the subgraph edges.

- Create a new file `testgraphbuilder.py` defining a test case with one test method each of the two above methods.

- Run the tests using `'../../data/tabular/aquasymbio-data.csv'`. The complete graph should contain 1207 nodes. Check that the edge from 'Parvilucifera infectans' to 'Alexandrium pacificum' contains two entries in its 'info' attribute. Check that the subgraph with source node 'Amyloodinium ocellatum' contains 129 nodes.

Outline : Session 2

Using Domain Specific Modules

1. The BioPython Toolkit
2. Processing Sequence Data
3. Running BLAST and Processing Results
4. Accessing Remote Resources
5. Working with Graph Structures
6. Putting it all Together

Final wrap up

What should have been imprinted in your brain after these training sessions:

- **The basics of the Python language with its data structures and flow control mechanisms**
- **Knowledge on how to use a portfolio of general purpose modules (argument parsing, logging, unit testing, using the web)**
- **Basic reflexes on where and how to look for in-depth documentation when necessary**
- **Best practices on how to write WORM code [Write Once Read Many (times)]**
- **An overview of bioinformatics (or related) toolkits: BioPython, NetworkX**