Biogenouest

ifb
INSTITUT FRANÇAIS DE BIOINFORMATIQUE

EMBRC FRANCE
CENTRE NATIONAL DE RESSOURCES BIOLOGIQUES MARINES

OCEANOMICS

IDEALG
seaweed for the future

ABiMS

Managing Data with Python

Session 201

June 2018

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

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

cnrs

Outline : Session 201

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

Outline : Session 201

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

Python's main data types (1)

1. **Scalars :**

```
myStringVar='Hello, World'      # A string variable
myIntVar=42                     # An integer variable
myFloatVar=3.14152              # A floating point variable
myBoolVar=True                  # A boolean variable
```

2. **Containers : lists**

```
myListVar=['Hello', 'World']
myListVar[1]='Universe'
myListVar.append('and Beyond')
myListVar.extend(['but','not','too','far' ])
myListVar=myListVar[0:1]+['Cruel']+myListVar[3:]+
          myListVar[1:2]
```

Python's main data types (2)

3. Containers : tuples (a.k.a : read-only lists)

```
myTupleVar=('Hello', 'World')
myTupleVar[1]='Universe'
myTupleVar.append('and Beyond')
myTupleVar.extend(['but','not','t      ar' ])
myTupleVar=myTupleVar[0:1]+['Cruel']+myTupleVar[3:]+
           myTupleVar[1:2]
```

A tuple can't be modified !

4. Containers : dictionaries

```
myDictVar={'lastname'   : 'Van ROSSUM',
           'firstname'  : 'Guido'}
myDictVar['employer']='Google Inc.'
myDictVar['colors']=['red','green','blue']
myDictVar['address']={'country' : 'USA',
                      'city'  : 'Mountain View'}
myDictVar['address']['street']='Snake Drive'
```

Looping constructions

1. Looping over a list or tuple :

```
for fruit in ('apple', 'banana','orange') :
    fruitJuice=fruitJuice+press(fruit)
```

2. Looping using an index :

```
for year in range(2001,2019) :
    computeVacationDaysInYear(year)
```

3. Looping over dictionaries :

```
for seqId in sequenceDict.keys() : # enumerate keys
    sequence=sequenceDict[seqId]
```

```
for sequence in sequenceDict.values() : # enumerate values
    totalLength=totalLength+len(sequence)
```

```
for (id,seq) in sequenceDict.items() : # enumerate both
    print('The sequence of: '+id+' is '+seq)
```

Using Functions

Defining a function :

```
def computeSum(a,b) :
    sum=a+b
    return sum
```

Calling a function :

```
mySum=computeSum(12,34)
```

Modifications to scalar function arguments inside a function have no effect outside the function.

Modifications to non-scalar (lists, dictionaries…) function arguments inside a function persist even when de function has completed execution.

Using Modules

Importing a module with all its components:

```
import moduleName
…
res=moduleName.computeSomethingSmart(data)
```

Selectively importing components from a module:

```
from moduleName import computeSomethingSmart
…
res=computeSomethingSmart(data)
```

Outline : Session 201

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

Object Oriented Programming Principles

Until now, we have been writing Python scripts using *procedural programming* :

- **We used more or less elaborate data structures which we stored in variables.**

- **And we used functions to which these variables were passed as arguments, or returned at the end of the functions to read or modify the variable contents, or to build new variables.**

```
seqInfo=sequencetools.readFastaFromFile(filename)
...
seqInfo=sequencetools.computeSequenceLengths(seqInfo)
...
sortedIds=sequencetools.sortSequencesByLength(seqInfo)
...
```

Object Oriented Programming Principles

This is more or less OK while :

- **We are the unique author of the code using our data structures.**
- **We remain consistent in the design and usage of our data structures (we use *constant variables* for dictionary keys, we don't change the type of data that we store with a given key in the dictionary...)**

In short :

- **People (ourselves included) wanting to (re)use our code need to understand how we built the data structures, both for reading the data they contain, and for adding new data (to ensure that no existing data is accidentally overwritten).**

For example :

- **James wants to compute the GC% of nucleotidic sequences.**
- **He's less than eager to write functions for reading sequences from files.**
- **Francis told him about that handy `sequencetools` package developed by Rosalind.**
- **To use it, James will have to read through the code of the module to learn :**
  - **What function to call to read sequences from a file**
  - **How the data structure storing the sequence data is organised to access the actual letters of the sequence.**
- **And Rosalind can't change her data structure anymore without risking to break programs relying on it written by Francis or James.**

There must be a better way to design reusable and extensible programs!

Object Oriented Programming Principles

Object Oriented Programming :
Trying to mimic the way the "real world" works.

The "real world" is made of objects :

- **The apple I'm carrying in my bag.**
- **The car I just parked outside.**
- **My son I just dropped off at school.**

And I'm constantly interacting with or operating on these objects :

- **To CHECK IN WHAT STATE they are :**
  - **Was the apple I took the shiniest ?**
  - **Was there enough fuel in my car for this morning's trip ?**
  - **Did my son brush his teeth after breakfast ?**
- **To MODIFY THEIR STATE if needed :**
  - **I really need to wash this apple before eating it.**
  - **After the trip, my car will have a little less fuel and a little more mileage.**
  - **Give my son a compliment or a blame.**

The keywords here are : OBJECTS, STATE, OPERATIONS

Object Oriented Programming Principles

A tentative definition of an object

An object is characterized by its state : A SET OF PROPERTIES

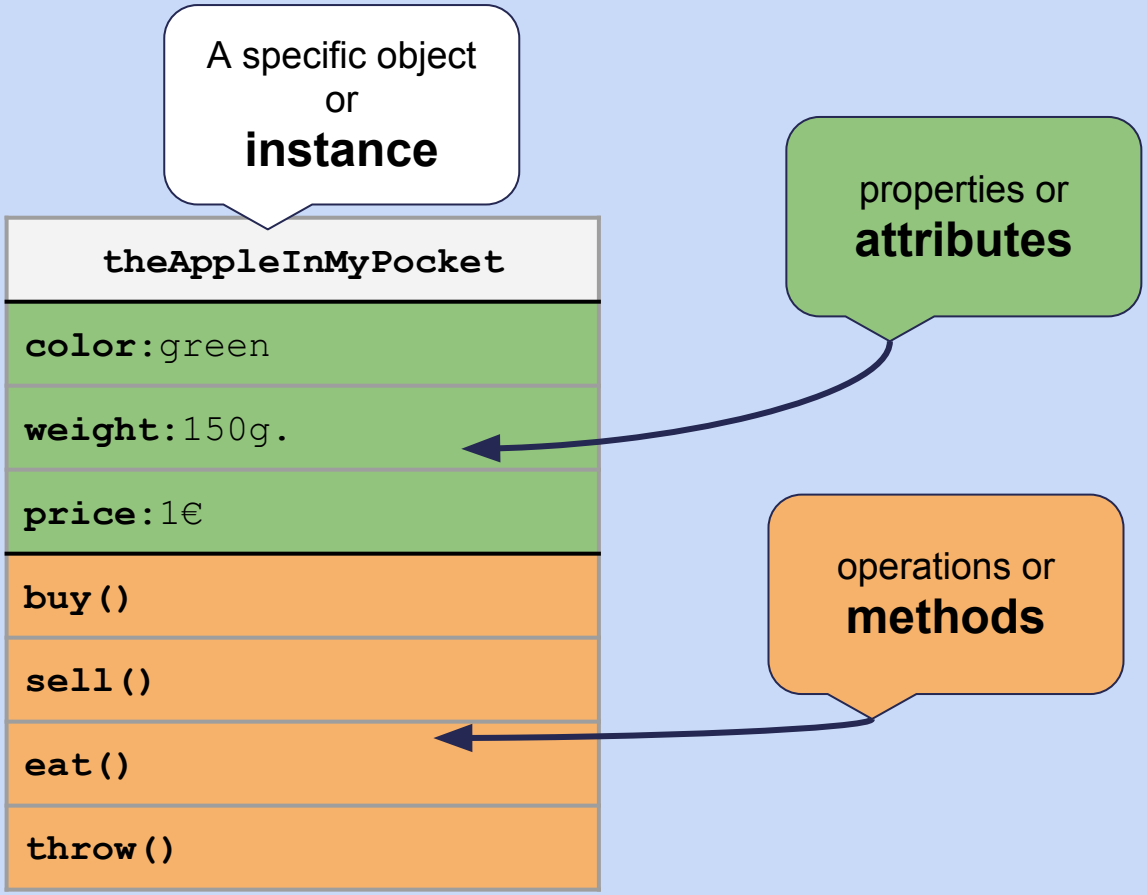
- An apple has a weight, a color, a price, belongs to a variety…
- A car has a brand, a color, a price, an owner, an engine…
- My son has a specific eye color, hair color, age, preferred videogame…

And by the SET OF OPERATIONS that it is capable of supporting :

- An apple can be bought, sold, eaten..
- A car can be driven from one place to another, be sold, be repaired…
- My son can take the bus, go for a swim, tidy his room (really ?)...

Object Oriented Programming Principles

A tentative representation of an object



Object Oriented Programming Principles

From Object to Class

Objects of the same kind share common attributes and support common methods : they belong to the same CLASS

Objects of the same class can all have a different state.

An object is said to be an instance of a class.

Outline : Session 201

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

Objects in Python

In Python, objects are "just another" type of variable. We already manipulated some objects in the Python for Beginners sessions :

```
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)
```

Creates an instance of class ArgumentParser

Variable referring to an instance of class ArgumentParser

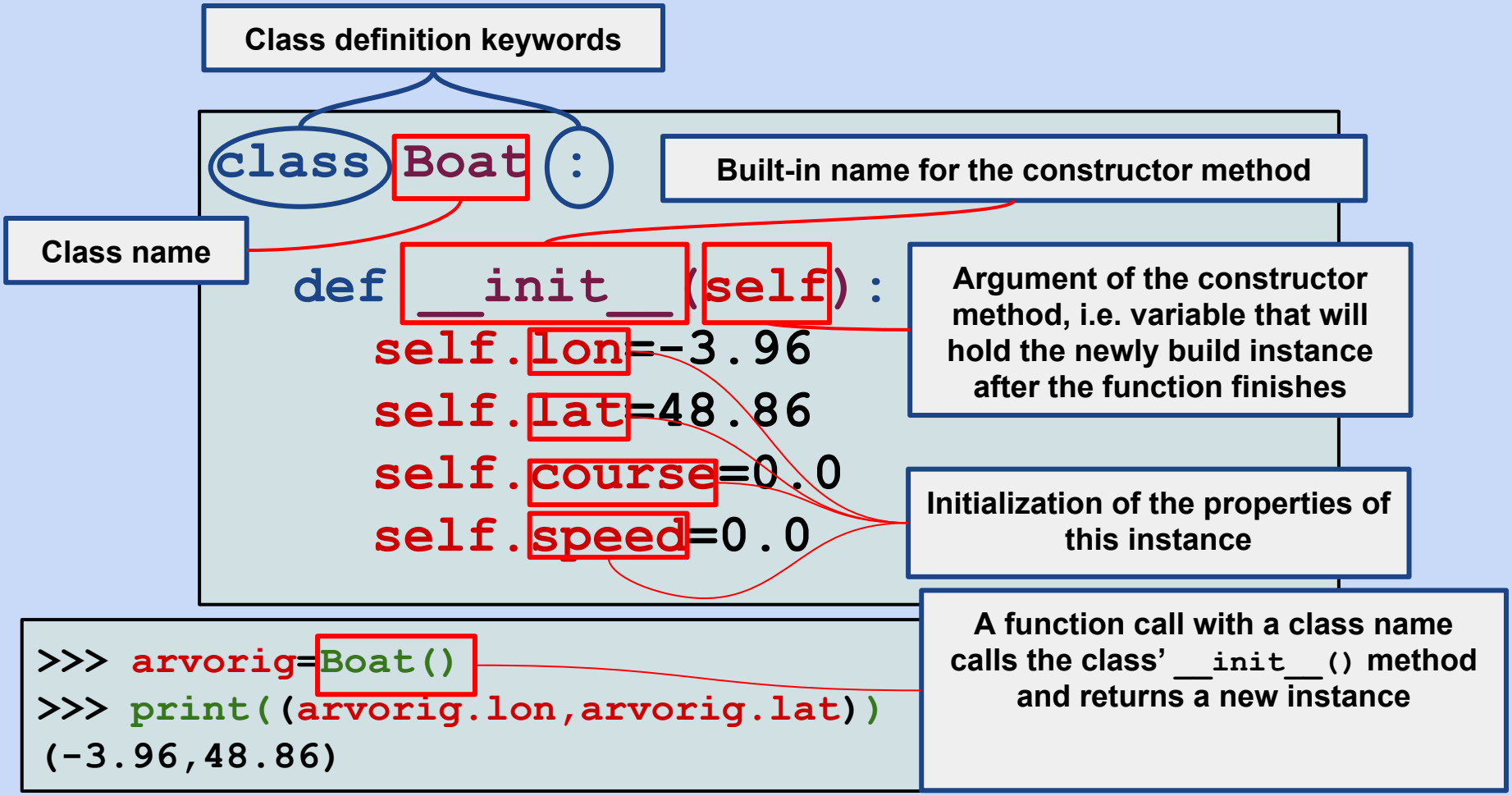
Method called on an instance of class ArgumentParser

```
import csv
…
reader=csv.DictReader(csvfile,fieldnames=['lastname','firstname','age
'],delimiter='\t'))
```

Creates an instance of class DictReader

Variable referring to an instance of class DictReader

Writing Classes in Python (I)
Defining the class and how to build instances



Writing Classes in Python (I)
Defining methods

Methods operating on an instance are defined in a class and always receive an instance as their first argument.

```
class Boat :
(...)
   def setPosition(self,newLon,newLat):
      self.lon=newLon
      self.lat=newLat

   def getPosition(self):
      return   (self.lon,self.lat)
```

Variable representing the Instance used to call the function.

```
>>> arvorig.setPosition(-5.34,48.86)
>>> print(arvorig.getPosition())
>>(-5.34,48.86)
```

Writing Classes in Python (I)

Like any other methods, methods defined in a class can have arguments (and default values).

```
class Boat :
    def __init__(self,defLon=-3.96,defLat=48.86):
        self.lon=defLon
        self.lat=defLat
        self.course=0.0
        self.speed=0.0
```

```
>>> arvorig=Boat()
>>> print(arvorig.getPosition())
(-3.96,48.86)
>>> pontaven=Boat(-5.32,31.65)
>>> print(arvorig.getPosition())
(-5.32,31.65)
```

Writing Classes in Python (I)

Methods getting or setting the value of an attribute are called accessors or getters resp. Setters :

```
class Boat :
    def setCourse(self,newCourse):
        self.course=newCourse
    def getCourse(self):
        return self.course
```

Accessors allow to hide the internal state representation of properties. This is called ENCAPSULATION.

ENCAPSULATION provides a means to allow the internal state representation to evolve without impacting programs using objects of a class.

```
class Boat :
    def __init__(self,defLon=-3.96,defLat=48.86):
        self.latlon=(defLat,defLon)

    def getPosition(self):
        return  (self.latlon)
```

Exercise 0

Using PyCharm :

- Create a new project called dmwp
- After downloading and extracting the archive contents with the training data, copy the data folder in the project folder
- After downloading and extracting the archive contents with the solutions of the Python for Beginners exercices, copy the src folder in the project folder.

Exercise 1

- Create a new folder in src called ex201
- Create a new Python file in folder ex201 called sequence.py
- In sequence.py, define a class called Sequence with the following properties : seqId, letters
- Implement the class constructor method, and the accessors ("getters" and "setters") for the each property.
- Create a new Python file in folder ex201 called main.py
- In main.py use the sequence module to create two example sequences (seqOne and seqTwo), and print the values of their properties using the "getter" methods.

Exercise 2

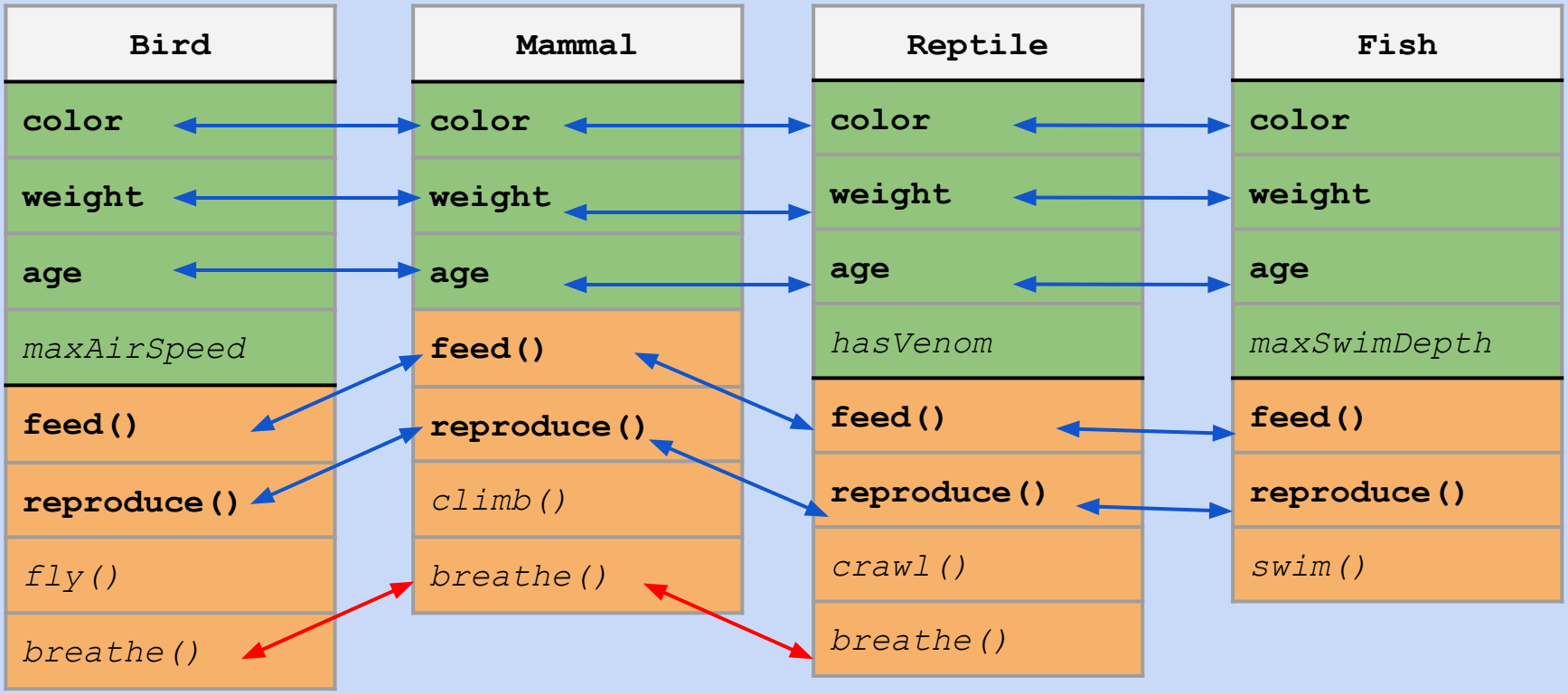
- Create a new directory `src/ex202` and copy the two files from `src/ex201`
- Enhance ce the `sequence` module by :
  - Creating a SequenceCollection class for managing sequence collections, and providing the following methods :
    - addSequence(self,sequence) : for adding a Sequence instance to the collection
    - removeSequence(self,seqId) : for removing a Sequence instance from the collection
    - getSequence(self,seqId) : returning the Sequence instance having the identifier seqId
    - getAllSequences(self) : returning all the Sequence instances of the collection (you will need to explicitly use list() for the return value).
    - len(self) : returning the number of Sequence instances in the collection.
- Modify the main.py file to :
  - Create an instance of SequenceCollection
  - Add the two sequences to the SequenceCollection instance
  - Check that the number of sequences in the collection is correct
  - Remove one of the two sequences from the collection
  - Check the length of the collection again.

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

Reusing behaviour across classes using inheritance

In the "real world" there are families of object that expose shared properties and behaviour :



Reusing behaviour across classes using inheritance

Transposing this to Object Oriented programming this could lead to code duplication :

```
class Bird:
…
    def feed(self,foodWeight):
        self.weight=self.weight+foodWeight
```

```
class Mammal:
…
    def feed(self,foodWeight):
        self.weight=self.weight+foodWeight
```

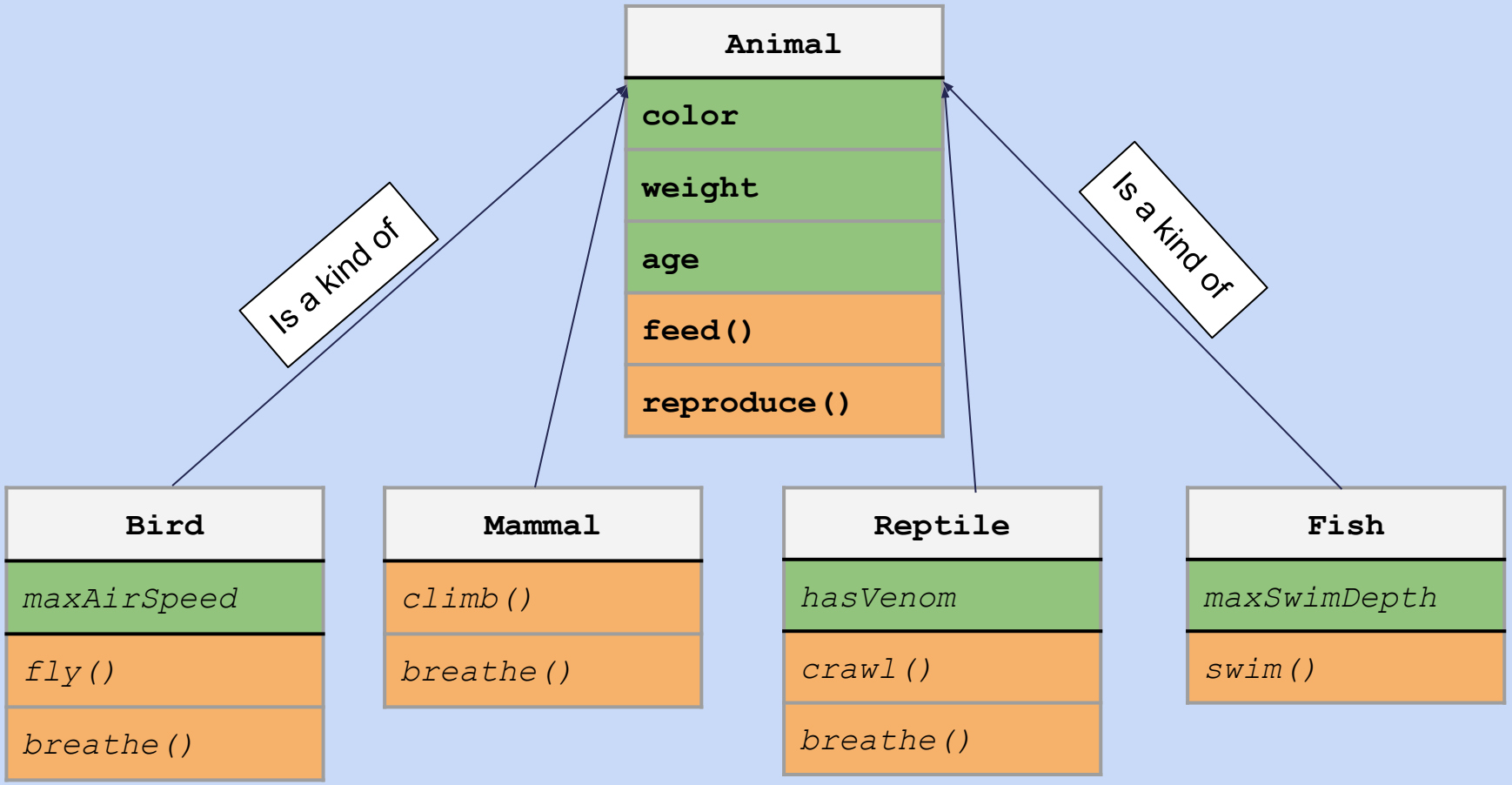
```
class Reptile:
…
    def feed(self,foodWeight):
        self.weight=self.weight+foodWeight
```

Duplicating code is evil !

Object Oriented Programming Principles

Reusing behaviour across classes using inheritance

Inheritance provides a way to group properties shared between a set of classes in a "parent class", and to derive or to "specialize" child classes. Child classes will benefit of all the properties of their parent class :



Reusing behaviour across classes using inheritance

Python allows classes to inherit from other classes :

```
class Animal:
…
    def feed(self,foodWeight):
        self.weight=self.weight+foodWeight
```

Base class

Derived classes

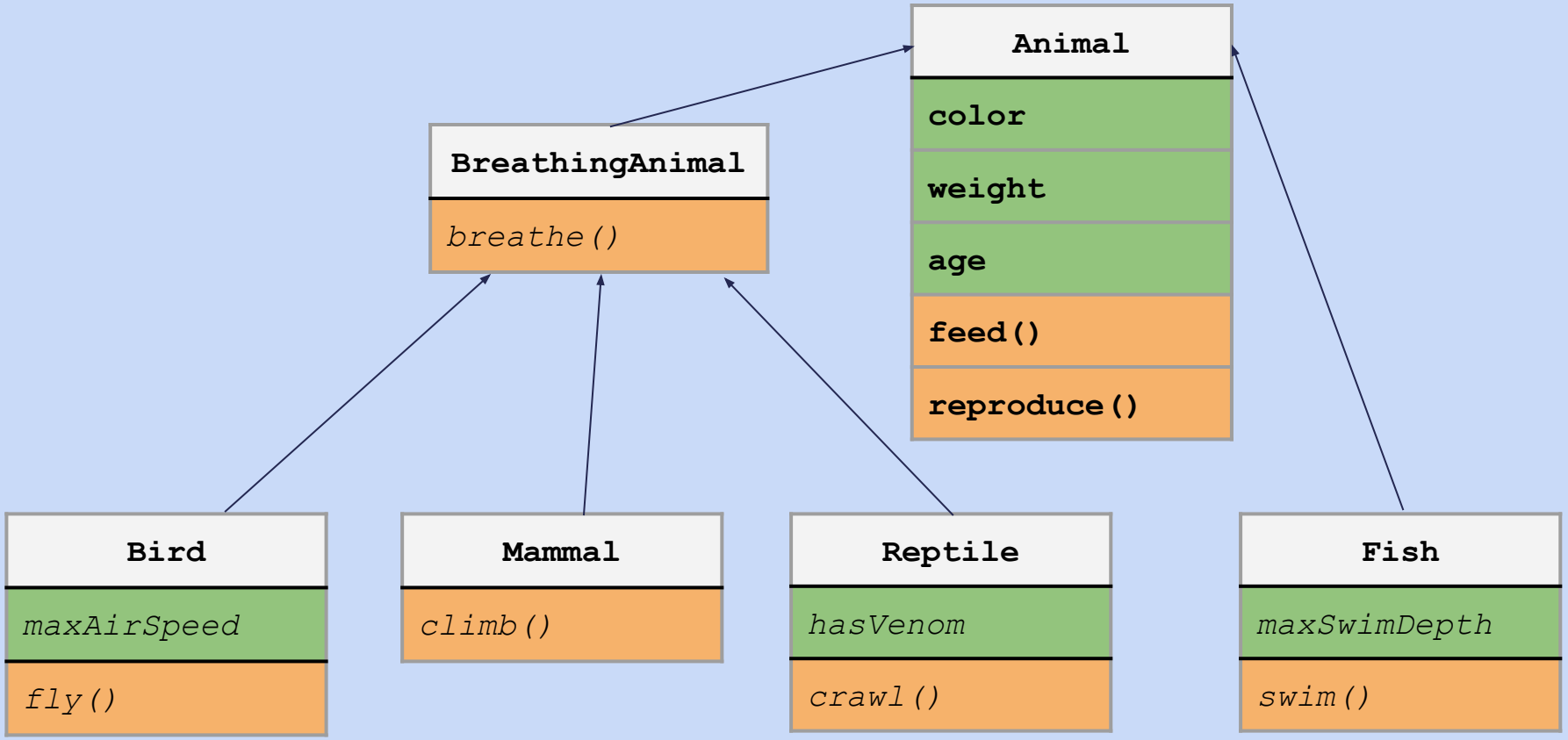
```
class Bird(Animal):
…
```

```
class Mammal(Animal):
…
```

```
class Reptile(Animal):
…
```

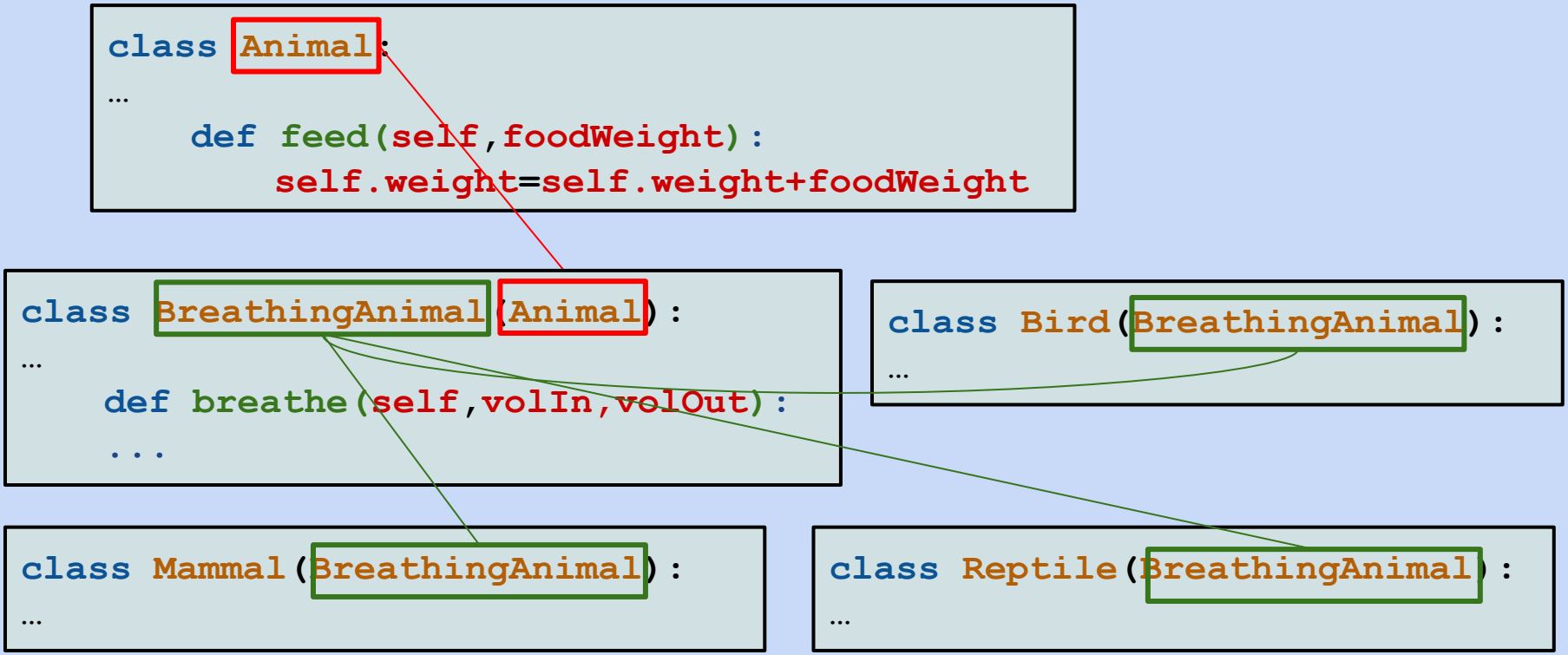
Reusing behaviour across classes using inheritance

Multiple levels of inheritance can be defined



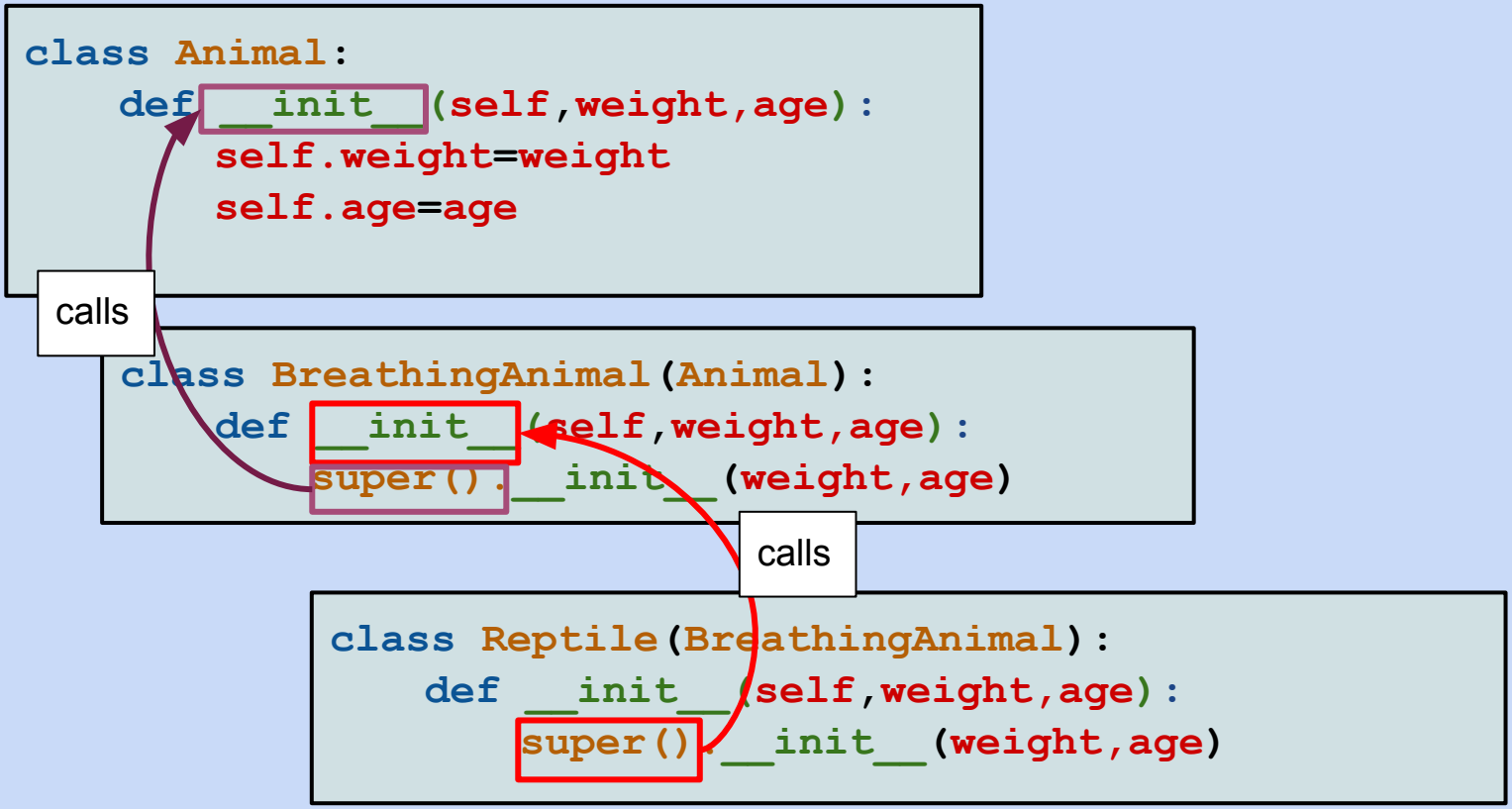
Reusing behaviour across classes using inheritance

An example of a three level inheritance tree :



Reusing behaviour across classes using inheritance

Inheritance and object construction : constructor methods in derived classes may need to call constructor methods in base classes to initialize common properties. This is done through the `super()` method.



Reusing behaviour across classes using inheritance

A derived class can OVERRIDE or REDEFINE its base class behaviour by redefining one or more of its methods :

```
class Animal:
…
    def feed(self,foodWeight):
        self.weight=self.weight+foodWeight
```

`Animal.feed()` will not be called anymore for `Mammal` instances.

```
class BreathingAnimal(Animal):
...
```

```
class Mammal(BreathingAnimal):
...
    def feed(self,foodWeight):
        # Mammals are famous for their sloppy eating
        self.weight=self.weight+foodWeight*0.6
```

`feed()` is redefined

Reusing behaviour across classes using inheritance

When overriding an inherited method, the super() method can also be used.

```
class Animal:
…
    def feed(self,foodWeight):
        self.weight=self.weight+foodWeight
```

```
class BreathingAnimal(Animal):
...
# no feed method is defined in this class
```

```
class Mammal(BreathingAnimal):
...
    def feed(self,foodWeight):
        # Mammals are famous for their sloppy eating
        foodWeight=foodWeight*0.6
        super().feed(foodWeight)
```

calls

Object Oriented Programming Principles

And last but not least :

Co-location of data (attributes) and algorithms that modify them (methods)

Until now, we had functions on the one hand, and data structures on the other hand. When calling a function, we were forced to add one or more arguments that the functions operated on.

With object oriented programming, methods are always applied to an object, and thus have access to the complete internal representation of the object. No need for arguments (except for data not part of the object whose method is called).

Exercise 3

- Create a new directory `src/203` and copy the files from `src/202`
- Write a `cyanosequence` module using the `sequence` module and defining :
  - A CyanoSequence class deriving from the Sequence class and adding a strain attribute with its accessors (getStrain() and setStrain()).
  - A CyanoSequenceCollection class deriving from the SequenceCollectionClass with a new method :
    - getSequencesInStrain(self,strain) : returning the list of Sequence instances belonging to the given strain.
    - getAllStrainNames(self) : returning the list of strains of all the sequences in the collection
- Modify the `main.py` program to :
  - Create three instances of CyanoSequence
  - Use the setStrain() accessor with arbitrary strain names (use the same strain name for two of the three instances).
  - Create an instance of CyanoSequenceCollection
  - Add all the instances to the collection.
  - Check the result of getAllStrainNames()
  - Check the result of getSequencesInStrain()

Exercise 4 (Advanced)

- Create a new directory `src/204` and copy the files from `src/203`
- Enhance the CyanoSequenceCollection class by adding readFromFastaFile(self,filename) method which :
  - Uses the readFastaSequencesFromFile() function from the sequencetools module (in the lib directory)
  - Builds an instance of CyanoSequence for each sequence read in the file, extracting the strain from the sequence ID with a regular expression :

    ```
    >CK_Pro_MED4_1241:1193591-1194439:1|PMM1241
    ```

  - Adds each of these instances to the collection instance.

- Modify main.py to :
  - Read sequences from the `cyanorak_complete.fna` file in the `data/fasta` directory.
  - Display the number of sequences for each strain in descending order.

Outline : Session 201

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

Unit Testing your Code

Acting before things go wrong!

A main concern, when developing scripts and modules is to ascertain that the code actually actually what is expected to do, without side effects.

```
class SequenceCollection:

    def addSequence(self,sequence):
        seqId=sequence.getSeqId()
        self.sequences[seqId]=sequence
```

Uh, oh ! What happens if there already was a sequence with the same seqId ?

We need a means to check that every single function runs as intended by :

1. **Before calling the function, "setting up" the environment it needs (initialize variables, create objects, open files…)**
2. **Calling the function in isolation from the others.**
3. **After the function has finished, checking that it has done its job properly.**
4. **Cleaning up the environment that was set up for the test.**

Decent programming languages provide adequate tools to build these UNIT TESTS and automate their execution.

Unit Testing your Code

How could we Unit Test method:

```
CyanoSequenceCollection.readFastaSequenceFromFile(...)
```

1. Setting-up the environment:

- Preparing a file with test sequence data whose contents are known.
- Creating in instance of CyanoSequenceCollection to be used for calling the function.

2. Calling the function in isolation from others:

- Use the previously created instance only for calling the function to test.
- Specify as filename the sequence data file we prepared for the test.

3. Checking that the function has done its job properly:

- Checking the number of sequences in the sequence collection: it has to match the number of sequences in the test file.
- Checking the names of the strains: all the strains present in the test file must be present

  …

4. Cleaning-up the environment means:

- Nothing specific in this case: no files were left unclosed, no network or database connections have been used.

Unit Testing your Code

Unit Testing in Python is heavily object oriented : it relies on a `TestCase` class in the `unittest` module.

To build or own unit tests we shall :

1. **Create a test class *derived from* `unittest.TestCase` :**

```
import unittest

class SequenceCollectionTest(unittest.TestCase):
...
```

Our very own test class

The parent class of our test class

Through inheritance, our test class will benefit from all the features of the `unittest.TestCase` class:

- **Methods to set-up and tear down (clean) the environment before and after each test function call.**
- **Methods to check that the function's outcome matches our expectations.**
- **Tight integration with our development environment to run tests and display their results.**

Unit Testing your Code

To build or own unit tests we shall :

2. Override a method to set up the test environment.

```
import unittest
import cyanosequence

class SequenceCollectionTest(unittest.TestCase):
…
    def setUp(self):
        self.seqCollection=cyanosequence.CyanoSequenceCollection()
```

`setUp()` is defined in `unittest.TestCase`. We specialize it to fit our needs.

We initialize a new attribute of our class with a new instance of the class having the functions/methods we want to test.

This method will be called just before each single test method is called. If our test class contains three test methods, there will be three calls to the `setUp()` method.

In this case, a new instance of `CyanoSequenceCollection` will be created and used in a single test function.

Unit Testing your Code

To build or own unit tests we shall :

3.Write a test method that calls the method to test

```
import unittest
import cyanosequence

class SequenceCollectionTest(unittest.TestCase):
…
    def testReadFastaSequencesFromFile(self):
        self.seqCollection.readFastaSequencesFromFile('test.faa')
```

Notice that the method name starts with `test`.

We use the instance created during setup to call the method to test.

The method to test will use a well-known test data file.

By default, all methods starting with `test` will be considered test methods and run by the testing framework (PyCharm in our case).

Unit Testing your Code

To build or own unit tests we shall :

4. Add *assertions* to our test method to verify the results of the method under test

```
import unittest
import cyanosequence

class SequenceCollectionTest(unittest.TestCase):
…
    def testReadFastaSequencesFromFile(self):
        self.seqCollection.readFastaSequencesFromFile('test.faa')
        totalSequences=self.seqCollection.len()
        self.assertEqual(totalSequences,42)
```

One of the many assertion methods provided by `unittest.TestCase`

The expected value : we know exactly how many sequences there are in our test data. So we can check if all have indeed been read.

Assertions are used to compare a value computed in the test function to an expected value and yield a boolean result. The test framework considers that a given test (method) fails if one of the assertions it contains returns `False`.

Unit Testing your Code

To build or own unit tests we shall :

5. Clean up the test environment

```
import unittest
import cyanosequence

class SequenceCollectionTest(unittest.TestCase):
…
    def tearDown(self):
        # Nothing to do here, no resources are left
        # in a pending state.
        pass
```

`tearDown()` can be overridden to clean up the test environment.

The pass keyword allows to define "do nothing" code blocks.

The `tearDown()` method is provided to clean up all resources that might have been allocated for the test function and not yet released. It might not be necessary to override `tearDown()`.

Unit Testing your Code

A warning regarding the outcomes of unit tests :
A FAILED test is different from a test with an ERROR

- A **FAILED** test is a test where the actual outcome is different from the expected outcome :

```
    def testReadFastaSequencesFromFile(self):
        self.seqCollection.readFastaSequencesFromFile('test.faa')
        totalSequences=self.seqCollection.len()
        self.assertEqual(totalSequences,42)
```

This test will FAIL if `totalSequences` is not 42

- A test with an **ERROR** is a test whose execution has gone wrong and did not proceed until the end :

```
    def testReadFastaSequencesFromFile(self):
        self.seqCollection.readFastaSequencesFromFile('bogus.faa')
        totalSequences=self.seqCollection.len()
        self.assertEqual(totalSequences,42)
```

This test will produce an ERROR because of a non existing data file.

Unit Testing your Code

The Unit Test mantra goes as follows: there MUST be at least one test method for each implemented method.

Hence, as the codebase of useful functions grows, so does the code base of unit tests. Luckily, software development environments provide tools to automate the execution of the complete set of unit tests of a module or even a project.

Writing comprehensive unit tests has two more benefits :

1. **Each test shows how a function can be used and is a way of documenting the code.**

2. **Regularly running *unit test suites* ensures that a modification in one area of the code doesn't break other areas (or else tests will pinpoint these locations).**

As a matter of fact, a number of open source projects require that contributors write unit tests when they want their code to be integrated in the project. These projects provide procedures automatically running unit tests and checking their outcome before code gets integrated.

Unit Testing your Code

The actual tests can be run by (cleanly) adding a call to the `unittest.main()` method at the end of the file with the test case:

```
import cyanosequence
import unittest

class SequenceCollectionTest(unittest.TestCase):

...

if __name__=='__main__':
    unittest.main()
```

When running this script from the command-line, the output is:

```
(myvenv)[foobar] python cyanosequencetest.py
Ran 1 test in 0.001s
OK
(myvenv)[foobar]
```

PyCharm has all the built-in functionalities to recognize and run unit tests.

Exercise 205

- Copy the contents of directory `src/ex204` in directory `src/ex205`

- Write a unit Test Case, `testcyanosequence.py`, testing the following methods available on the `CyanoStrainCollection` class :
  - `getAllStrainNames()` : check that the number of strains is as expected (14)
  - `getSequencesInStrain()` : check that the number of sequences for strain RCC307 is as expected.

- Run the tests using `../../data/fasta/cyanorak_complete.faa` to build the initial data structure.

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

Handling Exceptions

When writing a program, we try to predict how things may go wrong, and how to cope with such situations :

- **Before using a dictionary key, we check if it exists.**
- **Before accessing an element in a list, we check if the index is valid.**
- **Before trying to extract a pattern from a string, we check if the pattern is present.**

All these precautions are based on the assumption that we can anticipate faulty situations and prevent them from happening.

But what about unforeseen events ?

- **A data file given as argument may not be readable by our program**
- **A data column supposed to contain numbers may contain strings**
- **When retrieving data remotely, the network connection may fail**

Handling Exceptions

In Python, unforeseen events raise Exceptions

For ex. : When trying to read a non-existent file

1. Somewhere in `main.py`

```
...
seqCollection.readFastaSequencesFromFile('../../data/fasta/cyanorak
_complete.fab')
...
```

I made a typo in the filename. This file doesn't exist.

2. When trying to run `main.py`

```
Traceback (most recent call last):
  File "C:/Users/Mark/PycharmProjects/dmwp/src/ex205/main.py", line 4, in <module>
    seqCollection.readFastaSequencesFromFile('../../data/fasta/cyanorak_complete.fab')
  File "C:\Users\Mark\PycharmProjects\dmwp\src\ex205\cyanosequence.py", line 37, in
readFastaSequencesFromFile

allSequences=sequencetools.readFastaSequencesFromFile(filename,sequenceType=sequencetools
RESIDUES_TYPE)
  File "C:\Users\Mark\PycharmProjects\dmwp\lib\sequencetools.py", line 43, in
readFastaSequencesFromFile
    with open(filename) as infile :
FileNotFoundError: [Errno 2] No such file or directory:
'../../data/fasta/cyanorak_complete.fab'
```

The actual cause of the exception being raised

The exception stack

Program execution is brutally interrupted

Handling Exceptions

Python provides a mechanism to intercept and process Exceptions
The `try` / `except` instruction blocks

```
try :
    seqCollection.readFastaSequencesFromFile('wrongfile.data')
except :
    print('There was an error accessing the file')
```

- The `try` keyword is followed by an instruction block
- If an exception occurs in this instruction block, execution of the program "jumps" to the start of the `except` block instead of interrupting the program.
- The `except` block can process the exception, and execution then continues after the `except` block.

Handling Exceptions

Selecting which exceptions to process

Oftentimes, we can only handle certain types of exceptions: missing dictionary keys, list indexes exceeding the list size, file opening/reading errors.

Other exceptions are mostly beyond our control : insufficient memory to carry out computations, no more disk space to write results

The `except` keyword can be followed by the type of exception to intercept :

```
try :
    res=res+myList[tooLargeIndex]
except IndexError :
    print('Your index exceeds the size of the list!')
```

We shall only handle oversized index exceptions.

Exceptions of a different type occurring in the `try` block will not be processed in the `except` block.

Handling Exceptions

Some useful exception types

`IndexError`	A wrong value was used as list/tuple index
`KeyError`	A wrong (non-existent) value was used as key in a dictionary
`ZeroDivisionError`	Zero was found as denominator in a division operation
`TypeError`	The argument of a function or an operator is of a wrong type
`FileNotFoundError`	The file to which access was tempted doesn't exist
`PermissionError`	The user/program has insufficient permissions to access the file

The complete list of built-in exceptions can be found at:

https://docs.python.org/fr/3.5/library/exceptions.html#TypeError

Handling Exceptions

Defining and raising your own exceptions

Basically, exceptions are ordinary Python classes. Nothing prevents us from deriving new `Exception` classes using `Exception` (or one of its subclasses) as parent class.

```
class FastaFormatException(Exception):
    def __init__(self,message=''):
        self.message=message
```

An explicit message can be provided.

We can then raise these exceptions when needed using the `raise` keyword :

```
class CyanoSequenceCollection(SequenceCollection):
    def __checkFastaFormat(self,filename):
        line=''
        with open(filename) as fastafile :
            line=fastafile.readLine()
        if line[0] != '>' :
            raise FastaFormatException('line[:-1] is not a Fasta ID')
```

The `raise` keyword

A new instance is created

Handling Exceptions

Defining and raising your own exceptions

Custom exceptions can then be processed as any other exception :

```
class CyanoSequenceCollection(SequenceCollection):

    def readFastaSequencesFromFile(self,filename):
        try :
            __checkFastaFormat(filename)
            ...
            # The rest of the code of this method doesn't change.
            ...
        except FastaFormatException as ffe:
            print(ffe.message)
```

The instance is stored in the `ffe` variable.

Handling Exceptions

Defining and raising your own exceptions

Custom exceptions can then be processed as any other exception :

```
class CyanoSequenceCollection(SequenceCollection):

    def readFastaSequencesFromFile(self,filename):
        try :
            __checkFastaFormat(filename)
            ...
            # The rest of the code of this method doesn't change.
            ...
        except FastaFormatException as ffe:
            print(ffe.message)
```

The instance is stored in the `ffe` variable.

Handling Exceptions

Differential exception handling and the `finally` block

- A single `try` block may raise different exceptions which can be handled separately.
- Using the `finally` keyword, it is possible to define an instruction block that will be executed whether or not (an) exception(s) occurred in the `try` block.

```
class CyanoSequenceCollection(SequenceCollection):

    def readFastaSequencesFromFile(self,filename):
        try :
        ...
        except FastaFormatException as ffe:
            print(ffe.message)
        except FileNotFoundException:
            print('There was an error accessing the
        finally :
            # make sure that our sequence collection is consistent
```

Executed whatever happened in the try block

Unit Tests

Exercise 206

- Copy the contents of directory `src/ex205` in directory `src/ex206`

- Handle the case where a sequence ID does not contain a strain identifier :
  - Create a new Exception class : `MissingStrainException`
  - Modify the `CyanoSequenceCollection.readFastaSequencesFromFile` method to raise the `MissingStrainException` when the pattern looking for the strain doesn't match.

- If time allows, enhance your Unit Tests :
  - Add a test verifying that the exception is indeed raised, using the `assertRaises()` assertion method : https://docs.python.org/3/library/unittest.html

- Check the results using:

```
../../data/fasta/cyanorak_complete_bogus.faa
```

Outline : Session 201

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

Tracing execution with loggers

The limits of output generated through `print()`

We abundantly used the `print()` function to generate output from our modules and scripts. This method has several drawbacks :

- When output is really verbose, the screen scrolls and output can even be lost if the scroll memory is too small.
- All kinds of output are mixed: results destined to the user as well as progress or debug messages destined to the developer.

We need a mechanism:

- Ensuring we can easily collect all the output of our programs and scripts.
- Allowing us to select an appropriate channel depending on the nature of the information we want to communicate.

Tracing execution with loggers

Basic principles of the logging module

- Output is managed by instances of the `Logger` class : forget about `print()`
- We can create as many different `Loggers` as needed to generate output, each dedicated to a single module / class / method (our choice).
- Each `Logger` can be configured to use different output mechanisms: the terminal, a log file, emails, database records
- `Loggers` handle messages with different priority levels and can be configured with a threshold to filter out low-priority messages (remember the -v (--verbose) or -d (--debug) options) :

`DEBUG` < `INFO` < `WARNING` < `ERROR` < `CRITICAL`

Tracing execution with loggers

Basic usage of `Loggers`

The most basic way to benefit of logging is to :

- Import the module
- Set the desired logging level
- Call the methods that match the different priority levels to generate messages

```
import logging

logging.basicConfig(level=logging.DEBUG)

if __name__ == '__main__' :
    logging.debug("This is a debug message.")
```

The default logging level is WARNING. Set the level to DEBUG

Generates a message of level DEBUG

Will display on the terminal (or the console) :

```
(myvenv) [foobar] python loggingdemo.py
 DEBUG:root:This is a debug message.
(myvenv) [foobar]
```

The name of the logger.

The log message

The level of the message.

Tracing execution with loggers

Working with differentiated `Loggers`

It is however recommended to create module specific loggers. This is done with the `logging.getLogger()` function. This function takes as argument an arbitrary string which will be the name of the logger. If no logger with the same name exists, a new instance is created and returned. Otherwise, the already existing one is returned.

```
import logging
if __name__ == '__main__' :
    loggerOne=logging.getLogger("LoggerOne")

    loggerTwo=logging.getLogger("LoggerTwo")

    loggerOneAlias=logging.getLogger("LoggerOne")
```

Creation of a logger with name “LoggerOne”

Creation of a logger with name “LoggerTwo”

The previously created “LoggerOne” is returned

Tracing execution with loggers

Logger configuration

Loggers are highly configurable objects, the main areas of configuration are:

1. **The *channel* to which log messages are sent : standard output (terminal), log file, e-mail, database record...**

2. **The format of the log messages : date/time, module / class / method information, level of the message, message details**

3. **The level at which messages are handled by each channel : all messages below the threshold level are simply ignored.**

Tracing execution with loggers

Configuring log *handlers*

Log message output channels are called *handlers*. Once created, they can be added to any given logger using the `addHandler()` method. Thus, a single logger can have multiple *handlers*, and each handler can have its own level.

A `StreamHandler` can be used to channel log messages to the terminal.
A `FileHandler` can be used to channel log messages to… a file(!).

```
import logging
(…)
myModuleLogger=logging.getLogger(__name__)
myModuleVerboseHandler=logging.StreamHandler()
myModuleVerboseHandler.setLevel(logging.DEBUG)
myModuleLogger.addHandler(myModuleVerboseHandler)
myModuleTerseHandler=logging.FileHandler('/tmp/python.log')
myModuleTerseHandler.setLevel(logging.WARNING)
myModuleLogger.addHandler(myModuleTerseHandler)
(…)
myModuleLogger.debug('Some unimportant detail')
myModuleLogger.warning('Something went bad.')
```

Create two handlers and attach them to the same logger.

Generated on screen only

Generated on screen and in the logfile

Tracing execution with loggers

Formatting log messages

The actual format of the log messages is managed by `Formatters`. Formatter constructors take a single argument: a string describing how to format the log messages, which may contain "variable-like" fields which will be replaced dynamically.

%(msg)s	The message transmitted by the call to the logger
%(asctime)s	A human-readable representation of the date/time of the log message generation
%(name)s	The name of the logger used to send the message
%(lineno)d	The line number on which the call to to the logger was made
%(levelname)s	The string representation of the level of the message
%(funcName)s	The function in which the call to the logger was made
%(filename)s	The file name in which the call to the logger was made

```
myModuleLogger=logging.getLogger(__name__)
myModuleStreamHandler=logging.StreamHandler()
myModuleLogger.addHandler(myModuleStreamHandler)
myModuleStreamFormatter=logging.Formatter("%(asctime)s-%(name)s-%(msg)s")
myModuleStreamHandler.setFormatter(myModuleStreamFormatter)
```

Defines a specific format of log messages for this handler

Tracing execution with loggers

Setting the log level

Log message filtering according to the log level happens in two locations:

1. Inside the logger instance itself
2. Inside every handler

In order for a message to be generated, its level must be at least the one defined for the logger. Only then will the message be passed to each handler which, in turn, proceed to the actual generation or decide to ignore it (depending on their level configuration).

As a rule of thumb :

Set the log level both in the logger(s) and in the handler(s)

Tracing execution with loggers

Logger configuration “inheritance”

Logger configurations are “inherited” between loggers based on their names: a logger called `sampleModule` wil be considered a parent logger for a logger called `sampleModule.SampleClass`.

If no extra configuration is performed on the `sampleModule.SampleClass` logger, it will behave exactly as the `sampleModule` logger.

This allows for very simple and powerful logging strategies: one logger at the module scope, another logger at the class scope, and another at the method scope. All sharing the same characteristics but customizable at will.

There is always a logger at the root of the logger hierarchy named “root”

Tracing execution with loggers

Exercise 7

- Copy the contents of directory `src/ex206` into directory `src/ex207`
- Add loggers to the `cyanosequence.py` method:
  - generating a `debug()` message each time `getAllStrainNames()` is called.
  - generating a `warning()` message each time the strain is missing from the sequence identifier.
- Add loggers to the unit test methods:
  - generating an `info()` message at the beginning of each test method call.
- Configure the loggers to :
  - Send all `cyanosequence.py` log messages to the terminal.
  - Send all unit test log messages to a log file with the date/time information.

- Run the tests using `../../data/fasta/cyanorak_complete.fna`

Object Oriented Python

0. A Quick Refresher
1. What is Object Oriented Programming ?
2. Designing Classes & Implementing Methods
3. Using Inheritance
4. Unit Testing Your Code
5. Handling Exceptions
6. Tracing Execution With Loggers
7. Debugging under PyCharm

A final word on debugging

We already learned about three techniques enhancing confidence we may have in our code :

- Logging : we can follow (to a certain extent) what's going on in our code.
- Exceptions : we can try to intercept and handle events that otherwise cause our program to crash
- Unit testing : we can assess that our code does what it says it does

However, our code can still expose some unexplained behaviour : program crashes, missing or inconsistent results.

We need tools enabling us to debug our code, a.k.a, interactively inspect our programs while they are running

A final word on debugging

Main features of a decent debugger

1. Provide a way to stop a program on a specific line of code, before executing the latter.

2. Enable dynamic inspection of data structures (variables, objects…)

3. Allow to proceed to a step-by-step execution :
   a. By running one line of code at a time

   b. By entering into functions/methods at the location where they are called.

   c. By skipping over functions/methods we don't want to inspect

The PyCharm development environment provides all this features