Welcome to BIO-210

Applied software engineering for life sciences

November 24th 2024 – Lecture 11

Prof. Alexander MATHIS

EPFL

	Date	Topic	Software version	Software releases	Feedback
0	08/09/2025	Python introduction I			
1	15/09/2025	Python introduction II			
2	22/09/2025	Public holiday			
3	29/09/2025	Git & GitHub (+ installation)			
4	06/10/2025	Project introduction	v1		
5	13/10/2025	Functionify	v2	v1	
6	20/10/2025	EPFL fall break			
7	27/10/2025	Visualization & documentation	v3	**v2**	code review (API)
8	03/11/2025	Unit-tests, functional tests	v4	v3	
9	10/11/2025	Code refactoring	v5	v4	graded (tests)
10	17/11/2025	Profiling & code optimization	v6	v5	code review
11	24/11/2025	Object oriented programming	v7	v6	graded (speed)
12	01/12/2025	Model analysis & project report	v8	v7	code review (OO)
13	08/12/2025	Work on project (no class)			
14	15/12/2025	Wrap up		v8	graded (project)

Announcements

- profiling v6 was due today 10am

You received a second round of code reviews (for v5):

- Please check them, and answer (close issues if you addressed them)
- If you already closed them, did you provide links or references to your updates so it is clear how you addressed the feedback?
- If you did not receive a code review, then please email me!

Object oriented programming

Everything is an object in python

```
In [1]: astring="EPFL"
In [2]: astring.lower()              # string (instances) come with methods!!
Out[2]: 'epfl'
In [3]: astring.split('P')
Out[3]: ['E', 'FL']
In [4]: x=33.0
In [5]: x.is_integer()               # float instances come with methods!
Out[5]: True
In [6]: def f(x):
   ...:     ''' custom identity function '''
   ...:     return x
   ...:

In [7]: f.__name__                   # functions come with attributes!!
Out[7]: 'f'

In [8]: f.__doc__
Out[8]: ' custom identity function '
```

Object oriented programming in Python

- Classes allow you to bundle data and functionality together
- The `class` statement creates a class object and assigns it a name. Just like the function `def` statement, the Python class statement is an executable statement.
- When reached and run, it generates a new class object and assigns it to the name in the class header. Also, like `def` s, class statements typically run when the files they are coded in are first imported.
- Classes are essentially **factories for generating one or more objects**. Every time we call a class, we generate a new object (**instance**) with a distinct namespace – you cannot do this with modules
- Each object generated from a class has access to the class' attributes and methods as well as gets a namespace of its own for data that varies per object.

A first example

```
>>> class FirstClass:                # Define a class object
        def setdata(self, value):    # Define class' methods
            self.data = value        # self is the instance
        def display(self):           # Define another method of this class
            print(self.data)         # self.data: per instance

# Creating instances (by calling the class, notice `()`)
>>> x = FirstClass()                 # Make one instance
>>> y = FirstClass()                 # Make another instance, -- each has a new namespace
>>> x.setdata("EPFL")                # runs FirstClass.setdata(x,"EPFL")
>>> y.setdata(0.0001)                # Calling methods, self is y
>>> x.display()                      # self.data differs in each instance
EPFL
>>> y.display()                      # Calling FirstClass.display(y)
0.0001
>>> x.data = "MIT"                   # Can also overwrite attributes
>>> x.display()
MIT
# NOTE: Classes usually create all of the instance's attributes
# by assignment to the self argument, but they don't have to
>>> x.someotherattribute = 1972   # you can even set new attributes
>>> x.someotherattribute
1972
```

What is `self`?

- self is the name commonly given to the first (leftmost) argument in a class's method function.
- Python automatically fills it in with the instance object that is the implied subject of the method call.

```
>>> FirstClass.setdata(y,0.0001)      # lenghty version of y.setdata(0.0001)
>>> FirstClass.display(y)             # lengthy version of y.display()
0.0001
```

How can you look up available attributes and methods?

```
In [1]: x.__dir__()
Out[1]:
['data',
 'someotherattribute',
 '__module__',
 'setdata',
 'display',
 '__dict__',
 '__weakref__',
 '__doc__',
 '__repr__',
 '__hash__',
 '__str__',
 '__getattribute__',
 '__setattr__',
 '__delattr__',
 '__lt__',
 '__le__',
 '__eq__',
 '__ne__',              # and the list goes on...
 '__gt__',
 '__ge__',
```

Initializing instances from classes

As currently coded, our class `FirstClass` does not attach an attribute to an instance until the `setdata` method is called.

```
>>> a=FirstClass()
>>> a.data
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-171-9b9edf4dc236> in <module>
----> 1 a.data

AttributeError: 'FirstClass' object has no attribute 'data'

>> a.setdata('abc')
>>> a.data
'abc'
```

However, Python automatically calls a method named `__init__` each time an instance is generated from a class. This constructor can thus be used.

A second example

```
>>> class Complex:
        """ Our complex number class """ # you can (and should) put docstrings!
        def __init__(self,real,imaginary = 0):   # Set re and im attr. when constructed
            self.real = real
            self.imaginary = imaginary           # Notice default initialization!

>>> z = Complex(0,1)      # Define an instance with values (0,1)!
>>> z.real                # Accessing attributs ()
0
>>> z.imaginary
1
# In contrast this is not implemented for FirstClass
>>> FirstClass(0)

TypeError: FirstClass() takes no arguments
>>> z2=Complex(1)          # Define another instance, with default img.
>>> z2.real,z2.imaginary   # Notice default value!
(1, 0)
```

🧩🤔: Consider the following code

```
>>> 3+3
6
>>> "EPFL" + " is great!!!"
"EPFL is great!!!"
```

What is this feature of `+` called?

Operator overloading. The `+` operator behaves differently depending on the types of its operands.

More complex example with operator overloading

```
>>> class Complex:
        def __init__(self,real,imaginary=0):
            self.real = real
            self.imaginary = imaginary
        def __add__(self,z):                    # Overloading '+'
            self.real += z.real
            self.imaginary += z.imaginary
        def __mul__(self,z):                    # Overloading '*'
            r = self.real*z.real-self.imaginary*z.imaginary
            i = self.real*z.imaginary + z.real*self.imaginary
            self.real,self.imaginary = r,i      # Tuple assignment

>>> z = Complex(0,1)       # Create an instance 0+1j
>>> z+Complex(0,1)         # We can add with '+'; calls Complex.__add__(z,Complex(0,1))
>>> z.real,z.imaginary     # Look up values
(0,2)
>>> z*Complex(1,1)         # Complex muliplication (of two instances)
>>> z.real,z.imaginary     # Look up values
(-2,2)
>>> print(z)
<__main__.Complex object at 0x7f96d9782c70>
```

Overloading continued...

```
>>> class Complex:        # Adding more bells & whistles!
        def __init__(self,real,imaginary=0):
            self.real = real
            self.imaginary = imaginary
        def __add__(self,z):                  # Overloading '+'
            self.real += z.real
            self.imaginary += z.imaginary
        def __mul__(self,z):                  # Overloading '*'
            r = self.real*z.real-self.imaginary*z.imaginary
            i = self.real*z.imaginary + z.real*self.imaginary
            self.real,self.imaginary = r,i    # Tuple assignment
        def __str__(self):               # Overloading print!
            if self.imaginary==0:
              return str(self.real)
             elif self.imaginary<0:
              return str(self.real)+ "-" + str(self.imaginary)+"j"
            else:
              return str(self.real)+ "+" + str(self.imaginary)+"j"
>>> z = Complex(0,1)              # Creating an instance
>>> z +  Complex(3,0)
>>> print(z)                      # print operator is overloaded
3+ 1j
```

Adding your own methods…

```
>>> class Complex:        # Adding more bells & whistles (removed +/* for space)
        def __init__(self,real,imaginary=0):
            self.real = real
            self.imaginary = imaginary
        def __str__(self):              # Overloading print!
            if self.imaginary==1:
              return str(self.real)+ "+" + "1j"
            else:
              return str(self.real)+ "+" + str(self.imaginary)+"j"
        def norm(self):
            import math
            self.norm_value = math.sqrt(self.real**2+self.imaginary**2)

>>> z = Complex(3,1)              # Creating an instance
>>> z.norm()                      # Running
>>> z.norm_value                  # Accessing computed norm
3.1622776601683795
>>> z.__dict__                    # Namespace dictionary for class-based obj.
{'real': 3, 'imaginary': 1, 'norm_value': 3.1622776601683795}
```

🧩🤔: Is `statement` true?

```
>>> class Complex:
        def __init__(self,real,imaginary=0):
            self.real = real
            self.imaginary = imaginary
        def __str__(self):
            if self.imaginary==1:
              return str(self.real)+ "+" + "1j"
            else:
              return str(self.real)+ "+" + str(self.imaginary)+"j"
        def norm(self):
            import math
            self.norm_value = math.sqrt(self.real**2+self.imaginary**2)

>>> z = Complex(3,1)                              # creating an instance
>>> statement = 'norm_value' in z.__dict__        # is this statement true?
```

No, as z.norm() has not been run.

Questions?

🧩🤔: how do you create a class?

You create a class by running a class statement.

Like function definitions, these statements normally run when the enclosing module file is imported.

🧩🤔: How do you create an instance?

You create a class instance by calling the class name as though it were a function.

Any arguments passed into the class name show up as arguments two and beyond in the **init** constructor method. The new instance remembers the class it was created from for inheritance purposes.

🧩🤔: What is the difference between a class object and an instance object?

- Both class and instance objects are namespaces
- The main difference between them is that classes are a kind of factory for creating multiple instances.
- Classes also support operator overloading methods, which instances inherit, and treat any functions nested in the class as methods for processing instances.

Reminder: document your classes, just like all code!

“Code is more often read than written.” — Guido van Rossum (Creator of Python)

```
>>> class SimpleClass:
        """Class docstrings go here."""

        def say_hello(self, name: str):
            """Class method docstrings go here."""

            print(f'Hello {name}')

>>> help(SimpleClass)
Help on class SimpleClass in module __main__:

class SimpleClass(builtins.object)
 |  Class docstrings go here.
 |
 |  Methods defined here:
 |
 |  say_hello(self, name: str)
 |      Class method docstrings go here.
```

Class inheritance

In Python, instances inherit from classes, and classes inherit from superclasses

- Superclasses are listed in parentheses in a class header
- Classes inherit attributes from their superclasses
- This mechanism allows one to build hierachies
- By redefining attributes in subclasses that appear lower in the hierarchy, we can make specialized methods/attributes.

A simple hierarchical example

```
>>> class FirstClass:                # Define a class object
      def setdata(self, value):      # Define class's methods
          self.data = value          # self is the instance
      def display(self):
          print(self.data)           # self.data: per instance

>>> class SecondClass(FirstClass):   # Inherits setdata
      def display(self):             # We overwrite display
          print('Current value = "%s"' % self.data)

>>> x = SecondClass()
>>> x.setdata(0)      # inherited from FirstClass
>>> x.display()
Current value = "0"
```

The BIO-210 projects in a nutshell

Dynamical system:

$$X_{n+1} = F(X_n, \theta)$$

You wrote code for:

- initialization
- running dynamics
- checking convergence
- computing invariants
- storing data and plotting

Today's task: OOP refactoring and releasing as v7 next week (not graded)!

Lotka Volterra project

```
from scipy import integrate
from LotkaVolterraModel import dX_dt

# Initializing
a = 1.0   # natural growth rate of rabbits (prey)
b = 0.1   # natural dying rate of rabbits
c = 1.5   # natural dying rate of foxes
d = 0.75  # factor describing growth of foxes based on caught rabbits
T = np.linspace(0, 15, 1000)  # time
X0 = np.array([10, 5])  # initial conditions: 10 rabbits and 5 foxes

# Running the dynamical system and storing the output
X, infodict = integrate.odeint(
    lambda x, _: dX_dt(x, a, b, c, d), X0, T, full_output=True
)
```

Lotka Volterra project

Let's first write it more similarly to your project (with Forward Euler Method)

```
import numpy as np
# Initializing
a = 1.0   # natural growth rate of rabbits (prey)
b = 0.1   # natural dying rate of rabbits
c = 1.5   # natural dying rate of foxes
d = 0.75  # factor describing growth of foxes based on caught rabbits
X0 = np.array([10, 5])  # initial conditions: 10 rabbits and 5 foxes

dt = 0.01
num_iter = int(15./dt) # to run equally long at "T"

def update(state):
    return state + dt * np.array([ a * state[0] - b * state[0] * state[1],
                                  -c * state[1] + d * b * state[0]*state[1]])

X=np.empty((num_iter,2))

X[0] = X0  # initalize
for i in range(1,num_iter):
    X[i] = update(X[i-1])
```

Paired discussion: How should we refactor this?

Questions to consider (among others):

- First practical question: what functionality do you need to support?
- What are natural objects (classes and instances) for supporting those roles?
- What classes do you need? What attributes and methods?
- How should the classes relate to each other – what hiearchy of classes makes sense (to enable reuse)?

Refactoring in object oriented way

```
class LVM:
    """ Simple LotkaVolterraClass with Euler Integration """
    def __init__(self, a=1.0, b=0.1, c=1.5, d=0.75, dt=0.1):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
        self.dt = dt

    def update(self, X):
        """ Forward Euler method update """
        return X + self.dt * np.array(
            [self.a * X[0] - self.b * X[0] * X[1],
                -self.c * X[1] + self.d * self.b * X[0] * X[1]])

    def dynamics(self, X0, num_iter, saver):    # saver defined later!
        X = X0  # initalize
        saver.store_iter(X, 0)
        for i in range(num_iter):
            X = self.update(X)
            saver.store_iter(X, self.dt * i)
```

```python
class DataSaver:
    def __init__(self, instance=None):
        # self.a = LVM.a
        self.data = {"state_X": [], "state_T": []}
        if instance is not None:  # to store parameters with the result
            self.a = instance.a
            self.b = instance.b
            self.c = instance.c
            self.d = instance.d
            self.dt = instance.dt

    def reset(self):
        self.data = {"state_X": [], "state_T": []}

    def store_iter(self, X=None, T=None):
        if X is not None:
            self.data["state_X"].append(X.copy())
        if T is not None:
            self.data["state_T"].append(T)

    def get_data(self):
        return self.data

    def LyapunovFunction():
        raise NotImplementedError()
```

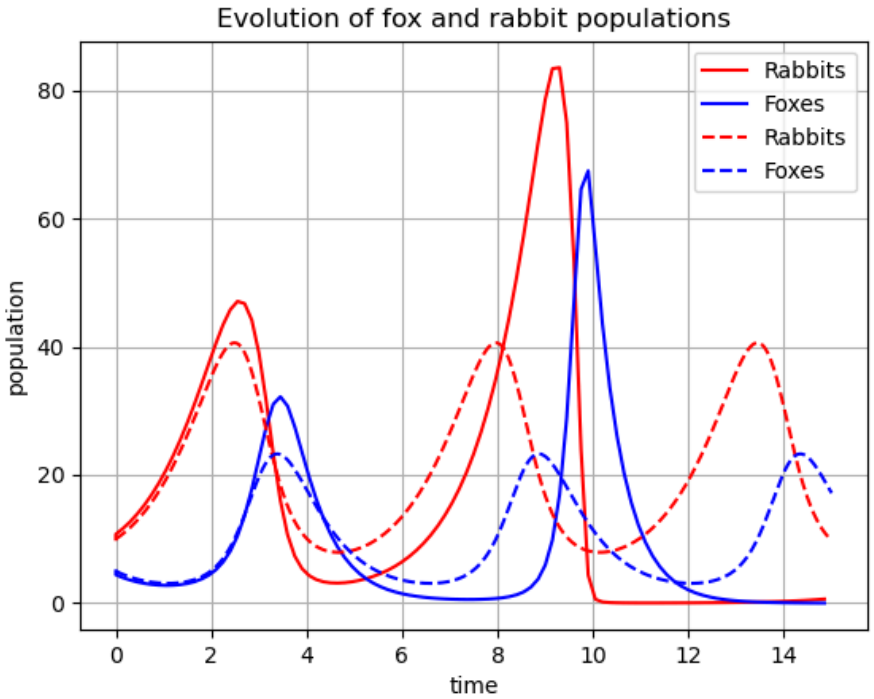
Simulating and storing the results

```
# Definition of parameters
a = 1.0
b = 0.1
c = 1.5
d = 0.75

numiter = 10000
dt = 15 * 1.0 / numiter

X0 = np.array([10, 5])        # initial conditions

lvm = LVM(a, b, c, d, dt)     # Creating model instance
saver = DataSaver(lvm)        # Creating saving instance
# Note: passing instance to store parameters!

lvm.dynamics(X0, numiter, saver)    # Running the dynamics and storing in saver instance

T, Xeuler = saver.get_data()["state_T"], np.array(saver.get_data()["state_X"])
rabbits, foxes = Xeuler.T
```

🧩🤔: What should you consider when you refactor?

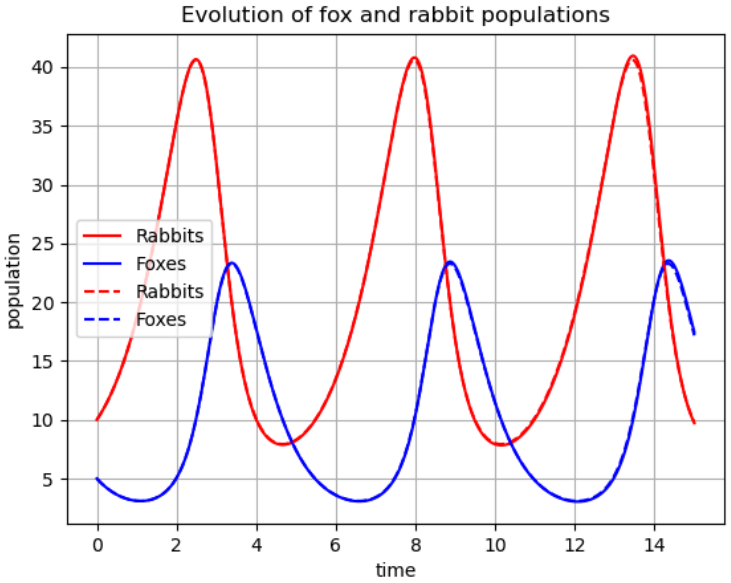
- do not introduce bugs
- so, be sure to use your tests

🧩🤔: What's wrong?



Legend: dashed: integrate.odeint; solid: our class

Smaller timestep, dt = 15 * 1.0 / 10**4



Questions?

Today’s summary

- object oriented programming in python
- writing a custom complex number class (as an example). Of course, python has built-in complex numbers! This example was purely pedagogical.
- refactoring the Lotka Volterra project

After lunch:

- Monday 13 - 15: exercises working on your project
- No office hours today (please email me)