

Welcome to BIO-210

Applied software engineering for life sciences

November 25th 2024 – Lecture 11

Prof. Alexander MATHIS

EPFL

	Date	Topic	Software version	Software releases	Grading / Feedback
0	09/09/2024	Python introduction I			
1	16/09/2024	Public holiday			
2	23/09/2024	Python introduction II			
3	30/09/2024	Git and GitHub (+installation VS Code)			
4	07/10/2024	Project introduction	v1		
5	14/10/2024	Functionify	v2	v1	
6	21/10/2024	EPFL fall break			
7	28/10/2024	Visualization and documentation	v3	v2	code review (API)
8	04/11/2024	Unit-tests, functional tests	v4	v3	
9	11/11/2024	Code refactoring	v5	v4	graded (tests)
10	18/11/2024	Profiling and code optimization	v6	v5	code review
11	25/11/2024	Object oriented programming	v7	v6	graded (speed)
12	02/12/2024	Model analysis and project report	v8	v7	code review (OO)
13	09/12/2024	Work on project			
14	16/12/2024	Wrap up		v8	graded (project)

Announcements

- final quiz (please fill it out by Friday 23:59)
- profiling v6 was due today 10am

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

- please check them, and answer
- if you already closed them, did you link your updates so it is clear where something was addressed?
- if you did not receive a code review, email me!

Object oriented programming

Everything is an object in python

```
1 In [1]: astring="EPFL"
2 In [2]: astring.lower()           # string (instances) come with methods!!
3 Out[2]: 'epfl'
4 In [3]: astring.split('P')
5 Out[3]: ['E', 'FL']
6 In [4]: x=33.0
7 In [5]: x.is_integer()           # float instances come with methods!
8 Out[5]: True
9 In [6]: def f(x):
10     ...:     ''' custom identity function '''
11     ...:     return x
12     ...:
13
14 In [7]: f.__name__                # functions come with attributes!!
15 Out[7]: 'f'
16
17 In [8]: f.__doc__
18 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

```
1  >>> class FirstClass:                # Define a class object
2      def setdata(self, value):         # Define class' methods
3          self.data = value            # self is the instance
4      def display(self):                # Define another method of this class
5          print(self.data)              # self.data: per instance
6
7  # Creating instances (by calling the class, notice `()`)
8  >>> x = FirstClass()                 # Make one instance
9  >>> y = FirstClass()                 # Make another instance, -- each has a new namespace
10 >>> x.setdata("EPFL")                 # runs FirstClass.setdata(x,"EPFL")
11 >>> y.setdata(0.0001)                # Calling methods, self is y
12 >>> x.display()                     # self.data differs in each instance
13 EPFL
14 >>> y.display()                     # Calling FirstClass.display(y)
15 0.0001
16 >>> x.data = "MIT"                  # Can also overwrite attributes
17 >>> x.display()
18 MIT
19 # NOTE: Classes usually create all of the instance's attributes
20 # by assignment to the self argument, but they don't have to
21 >>> x.someotherattribute = 1972      # you can even set new attributes
22 >>> x.someotherattribute
23 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.

```
1  >>> FirstClass.setdata(y,0.0001)      # lengthy version of y.setdata(0.0001)
2  >>> FirstClass.display(y)              # lengthy version of y.display()
3  0.0001
```

How can you look up available attributes and methods?

```
1 In [1]: x.__dir__()
2 Out[1]:
3 ['data',
4  'someotherattribute',
5  '__module__',
6  'setdata',
7  'display',
8  '__dict__',
9  '__weakref__',
10 '__doc__',
11 '__repr__',
12 '__hash__',
13 '__str__',
14 '__getattr__',
15 '__setattr__',
16 '__delattr__',
17 '__lt__',
18 '__le__',
19 '__eq__',
20 '__ne__',
21 '__gt__',
22 'ge',
```

and the list goes on...

Initializing instances from classes

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

```
1  >>> a=FirstClass()
2  >>> a.data
3  -----
4  AttributeError                                Traceback (most recent call last)
5  <ipython-input-171-9b9edf4dc236> in <module>
6  ----> 1 a.data
7
8  AttributeError: 'FirstClass' object has no attribute 'data'
9
10 >> a.setdata('abc')
11 >>> a.data
12 '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

```
1  >>> class Complex:
2      """ Our complex number class """ # you can (and should) put docstrings!
3      def __init__(self, real, imaginary = 0): # Set re and im attr. when constructed
4          self.real = real
5          self.imaginary = imaginary          # Notice default initialization!
6
7  >>> z = Complex(0,1)          # Define an instance with values (0,1)!
8  >>> z.real                    # Accessing attributs ()
9  0
10 >>> z.imaginary
11 1
12 # In contrast this is not implemented for FirstClass
13 >>> FirstClass(0)
14
15 TypeError: FirstClass() takes no arguments
16 >>> z2=Complex(1)            # Define another instance, with default img.
17 >>> z2.real,z2.imaginary      # Notice default value!
18 (1, 0)
```

Quiz: Consider the following code

```
1  >>> 3+3
2  6
3  >>> "EPFL" + " is great!!!"
4  "EPFL is great!!!"
```

What is this feature of ``+`` called?

More complex example with operator overloading

```
1  >>> class Complex:
2      def __init__(self, real, imaginary=0):
3          self.real = real
4          self.imaginary = imaginary
5      def __add__(self, z):                    # Overloading '+'
6          self.real += z.real
7          self.imaginary += z.imaginary
8      def __mul__(self, z):                    # Overloading '*'
9          real, imaginary = z
10         r = self.real*z.real-self.imaginary*z.imaginary
11         i = self.real*z.imaginary + z.real*self.imaginary
12         self.real, self.imaginary = r, i      # Tuple assignment
13 >>> z = Complex(0,1)                        # Create an instance 0+1j
14 >>> z+Complex(0,1)                          # We can add with '+'; calls Complex.__add__(z,Complex(0,1))
15 >>> z.real, z.imaginary                     # Look up values
16 (0,2)
17 >>> z*Complex(1,1)                          # Complex multiplication (of two instances)
18 >>> z.real, z.imaginary                     # Look up values
19 (-2,2)
20 >>> print(z)
21 <__main__.Complex object at 0x7f96d9782c70>
```

Overloading continued...

```
1  >>> class Complex:          # Adding more bells & whistles!
2      def __init__(self,real,imaginary=0):
3          self.real = real
4          self.imaginary = imaginary
5      def __add__(self,z):      # Overloading '+'
6          self.real += z.real
7          self.imaginary += z.imaginary
8      def __mul__(self,z):      # Overloading '*'
9          r = self.real*z.real-self.imaginary*z.imaginary
10         i = self.real*z.imaginary + z.real*self.imaginary
11         self.real,self.imaginary = r,i      # Tuple assignment
12     def __str__(self):        # Overloading print!
13         if self.imaginary==0:
14             return str(self.real)
15         else:
16             return str(self.real)+ "+" + str(self.imaginary)+"j"
17 >>> z = Complex(0,1)         # Creating an instance
18 >>> z + Complex(3,0)
19 >>> print(z)                 # print operator is overloaded
20 3+ 1j
```

Adding your own methods...

```
1  >>> class Complex:          # Adding more bells & whistles (removed +/* for space)
2      def __init__(self,real,imaginary=0):
3          self.real = real
4          self.imaginary = imaginary
5      def __str__(self):        # Overloading print!
6          if self.imaginary==1:
7              return str(self.real)+ "+" + "1j"
8          else:
9              return str(self.real)+ "+" + str(self.imaginary)+"j"
10     def norm(self):
11         import math
12         self.norm = math.sqrt(self.real**2+self.imaginary**2)
13
14 >>> z = Complex(3,1)          # Creating an instance
15 >>> z.norm()                  # Running
16 >>> z.norm
17 3.1622776601683795
18 >>> z.__dict__                # Namespace dictionary for class-based obj.
19 {'real': 3, 'imaginary': 1, 'norm': 3.1622776601683795}
```

Quiz: Is ``statement`` true?

```
1  >>> class Complex:          # Adding more bells & whistles (removed +/* for space)
2      def __init__(self,real,imaginary=0):
3          self.real = real
4          self.imaginary = imaginary
5      def __str__(self):        # Overloading print!
6          if self.imaginary==1:
7              return str(self.real)+ "+" + "1j"
8          else:
9              return str(self.real)+ "+" + str(self.imaginary)+"j"
10     def norm(self):
11         import math
12         self.norm = math.sqrt(self.real**2+self.imaginary**2)
13
14 >>> z = Complex(3,1)          # creating an instance
15 >>> statement = 'norm' in z.__dict__    # is this statement true?
```

Questions?

Quiz: how do you create a class?

Quiz: How do you create an instance?

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

Reminder: document your classes, just like all code!

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

```
1  >>> class SimpleClass:
2      """Class docstrings go here."""
3
4      def say_hello(self, name: str):
5          """Class method docstrings go here."""
6
7          print(f'Hello {name}')
8
9  >>> help(SimpleClass)
10 Help on class SimpleClass in module __main__:
11
12 class SimpleClass(builtins.object)
13 |   Class docstrings go here.
14 |
15 |   Methods defined here:
16 |
17 |   say_hello(self, name: str)
18 |       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 hierarchies
- By redefining attributes in subclasses that appear lower in the hierarchy, we can make specialized methods/attributes.

A simple hierarchical example

```
1  >>> class FirstClass:                # Define a class object
2      def setdata(self, value):         # Define class's methods
3          self.data = value            # self is the instance
4      def display(self):
5          print(self.data)              # self.data: per instance
6
7  >>> class SecondClass(FirstClass):    # Inherits setdata
8      def display(self):                # We overwrite display
9          print('Current value = "%s"' % self.data)
10
11 >>> x = SecondClass()
12 >>> x.setdata(0)                      # inherited from FirstClass
13 >>> x.display()
14 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

```
1  from scipy import integrate
2  from LotkaVolterraModel import dX_dt
3
4  # Initializing
5  a = 1.0    # natural growth rate of rabbits (prey)
6  b = 0.1    # natural dying rate of rabbits
7  c = 1.5    # natural dying rate of foxes
8  d = 0.75   # factor describing growth of foxes based on caught rabbits
9  T = np.linspace(0, 15, 1000) # time
10 X0 = np.array([10, 5]) # initial conditions: 10 rabbits and 5 foxes
11
12 # Running the dynamical system and storing the output
13 X, infodict = integrate.odeint(
14     lambda x, _: dX_dt(x, a, b, c, d), X0, T, full_output=True
15 )
```

Lotka Volterra project

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

```
1  import numpy as np
2  # Initializing
3  a = 1.0    # natural growth rate of rabbits (prey)
4  b = 0.1    # natural dying rate of rabbits
5  c = 1.5    # natural dying rate of foxes
6  d = 0.75   # factor describing growth of foxes based on caught rabbits
7  X0 = np.array([10, 5]) # initial conditions: 10 rabbits and 5 foxes
8
9  dt = 0.01
10 num_iter = int(15./dt) # to run equally long at "T"
11
12 def update(state):
13     return state + dt * np.array([ a * state[0] - b * state[0] * state[1],
14                                     -c * state[1] + d * b * state[0]*state[1]])
15
16 X=np.empty((num_iter,2))
17
18 X[0] = X0 # initialize
19 for i in range(1,num_iter):
20     X[i] = update(X[i-1])
21
```

Paired discussion: How should we refactor this?

Refactoring in object oriented way

```
1  class LVM:
2      """ Simple LotkaVolterraClass with Euler Integration """
3      def __init__(self, a=1.0, b=0.1, c=1.5, d=0.75, dt=0.1):
4          self.a = a
5          self.b = b
6          self.c = c
7          self.d = d
8          self.dt = dt
9
10     def update(self, X):
11         """ Forward Euler method update """
12         return X + self.dt * np.array(
13             [self.a * X[0] - self.b * X[0] * X[1],
14              -self.c * X[1] + self.d * self.b * X[0] * X[1]])
15
16     def dynamics(self, X0, num_iter, saver):      # saver defined later!
17         X = X0  # initialize
18         saver.store_iter(X, 0)
19         for i in range(num_iter):
20             X = self.update(X)
21             saver.store_iter(X, self.dt * i)
22
```

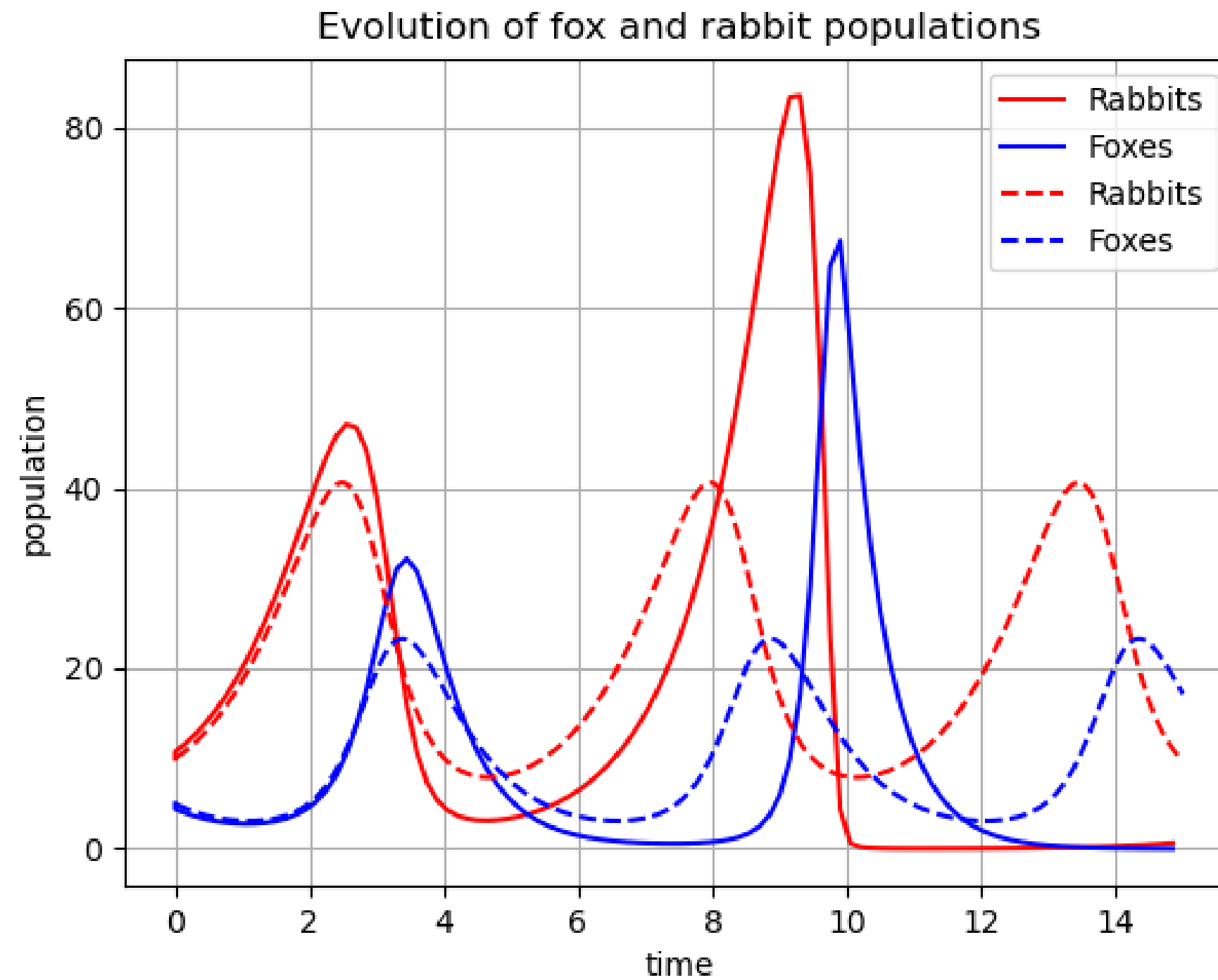
```
1  class DataSaver:
2      def __init__(self, instance=None):
3          # self.a = LVM.a
4          self.data = {"state_X": [], "state_T": []}
5          if instance is not None: # to store parameters with the result
6              self.a = instance.a
7              self.b = instance.b
8              self.c = instance.c
9              self.d = instance.d
10             self.dt = instance.dt
11
12     def reset(self):
13         self.data = {"state_X": [], "state_T": []}
14
15     def store_iter(self, X=None, T=None):
16         if X is not None:
17             self.data["state_X"].append(X.copy())
18         if T is not None:
19             self.data["state_T"].append(T)
20
21     def get_data(self):
22         return self.data
23
24     def LyapunovFunction():
25         raise NotImplementedError()
```

Simulating and storing the results

```
1  # Definition of parameters
2  a = 1.0
3  b = 0.1
4  c = 1.5
5  d = 0.75
6
7  numiter = 10000
8  dt = 15 * 1.0 / numiter
9
10 X0 = np.array([10, 5])      # initial conditions
11
12 lvm = LVM(a, b, c, d, dt)    # Creating model instance
13 saver = DataSaver(lvm)      # Creating saving instance
14 # Note: passing instance to store parameters!
15
16 lvm.dynamics(X0, numiter, saver)  # Running the dynamics and storing in saver instance
17
18 T, Xeuler = saver.get_data()["state_T"], np.array(saver.get_data()["state_X"])
19 rabbits, foxes = Xeuler.T
```

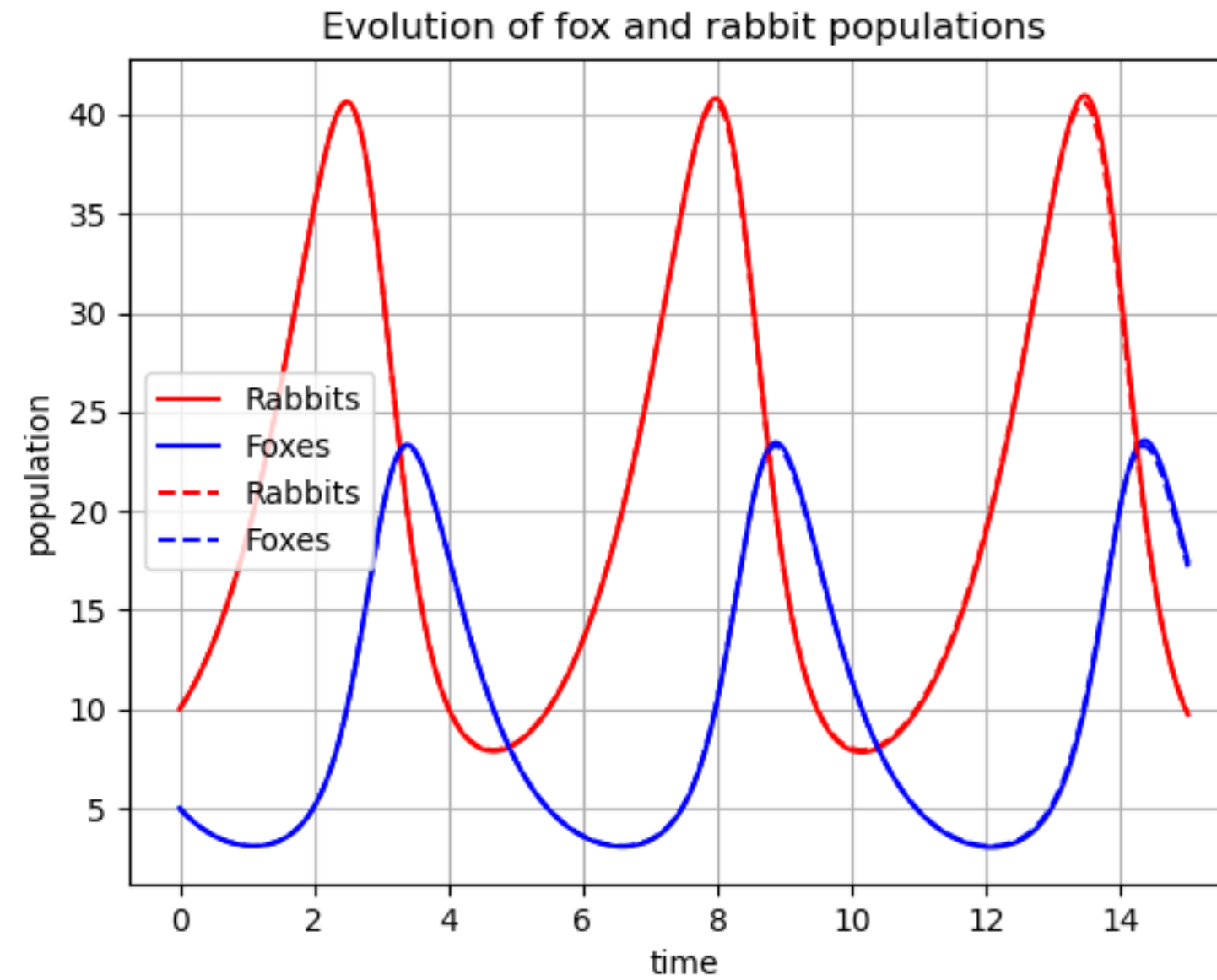
Quiz: What should you consider when you refactor?

Quiz: What's wrong?



dashed: integrate.odeint; lines: 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)
- refactoring the Lotka Volterra project

After lunch:

- Monday 13 - 15: exercises working on your project
- Monday 15:15 - 16 office hours in SV 2811
- Do the final quiz at your own pace until Friday 23:59