Note about object oriented programming and Python

Jakob Schiøtz

1 Introduction

In this note we introduce you to object oriented programming (OOP) and how to structure scientific software using Python.

2 Getting started

In this course we will be using Python and a number of scientific packages. We will extensively use *Jupyter Notebooks*, a framework for making interactive Python sessions with comments and documentation.

It is most practical if you have Python and Jupyter installed on your own computer. You can follow the instructions in Appendix A.

Note that two versions of Python are still in circulation: Python 3.X and Python 2.7. *We will exclusively work with Python 3.X,* the now decade-old Python 2 had end-of-life 31.12.2019, and may finally die off. MacOS and some Linux distributions still come with Python 2 build in, please install a more modern version!

2.1 Jupyter Notebooks

We shall often work in *Jupyter Notebooks*, as they make it easy to experiment with Python code interactively. You start it by opening a terminal, and then type

```
jupyter notebook
```

On a Mac use spotlight (Cmd-Space) to search for Terminal - or find it in the Other or Utilities folder of LaunchPad. On Windows, you can open an Anaconda Terminal from the start menu, or you can start the main Anaconda app, and from there start a notebook.

In any case, a browser opens. In the browser, you can start a new Python 3 notebook. You can now type Python code in the code boxes, and run them by pressing Shift-Enter.

3 What are ”objects”?

An “object” is a collection of data and some methods for working on such data. It *can* represent a physical object (i.e. an atom, a fruit or an employee), but may also be more abstract, for example an algorithm.

In Python, all variables are objects, and we can illustrate objects using the standard Python strings:

```
# Creating a string
a = "A string"
# The .upper() method creates an upper-case version
print(a.upper())
```

Here we first create the object and give it the name `a` (“assign it to the variable `a`”). Then we call the `upper()` *method* of the string.

A method is a function that is only intended to be used for objects of a specific type. It is called with the `object.method()` syntax. A method can take extra arguments in addition to the object belongs to.

The main reason for `upper()` being a method rather than a function is that it only makes sense for strings. There is no upper-case version of the number 42. There may be other kinds of objects than string where an "upper" operation makes sense, but it is likely to be a different operation.

Method can thus be used to define functions that only makes sense for a specific type of object — or it can be used to define behaviour that makes sense on many kinds of objects, but in a different way. An example is addition of objects: We can in a meaningful way define addition both for numbers and for strings, but it will be two different operations both represented by the + operator.

```
print(7 + 4)
print("Monty" + "Python")
```

In this case the + operator does two different things that both make sense for the object types at hand (the second example also illustrates that computers do what you tell them to do, not what you intend them to do: no space is added between the words).

So how does the + operator relate to methods? Behind the scenes, there is no global addition function. Instead, any type where addition makes sense can define a method called `__add__` which will be called when the + operator is used. This is an example of so-called *magic methods* in Python: methods whose name begin and end with two underscores will have special meaning. We shall meet them again soon.

Operator overloading. Most object-oriented programming languages allows you to "overload" operators: by defining special methods for your objects, you define what the operators of the programming language do when applied to these objects.

Polymorphism is when the same method (or the same operator) exists for different types of objects, and does something different according to the type of object. Often it will be a related operation which makes sense for multiple kinds of objects.

3.1 Exercise

Start a Jupyter Notebook, and try out the examples above. Also investigate what happens if you multiply a string with a number? Does it make sense to multiply two strings?

4 Defing your own object types

We need to start with a bit of nomenclature:

A class is a definition of a data type and its associated methods. Creating a class defines a new type in Python.

An instance is a variable of a given class.

For example, the string `"Example"` is an instance of the class `str` (the class of strings is called `str` in Python).

You can define a new class by using the `class` keyword. As an example let us create a class that represents a vector in two-dimensional space (Listing 1).

Listing 1 A very simple class representing a vector.

```
class Vector:
    def set(self, x, y):
        self.x = x
        self.y = y

v = Vector()
v.set(4, 5)
print(v.x)
print(v.y)
```

Here we create a class that has a single method, called `set`. The method takes three arguments, the implicit first argument is the instance itself, the following arguments are the ones we give explicitly when we call the method, it this case the desired coordinates.

The self argument: The first argument of a method is traditionally always called `self` in Python. When the method is called on an object, it will be set to the object (the variable before the dot operator). You can in principle give this variable any name, but calling it anything else than "self" is considered anti-social behaviour among Python programmers.

Note how data is stored in the object using *attributes*: local variables that belong to the specific instance, and are accessed with the syntax `instance.attribute`

This implementation of `Vector` has a few shortcomings. First, it is unfortunate that creating a vector is a two-step process, we first create the object and then assign it its value with the `set` method. This is both impractical an error-prone, and we can improve the class by defining an *initializer*, a function that is called immediately after creating the object.[1] If a class defines an `__init__` method, that methods is called when you create a new instance, and used to initialize it. If `__init__` takes arguments (in addition to self), then those arguments are specified upon object creation, see the example below.

Another serious flaw is that the class definition lacks documentation. Python makes it easy to define documentation strings, and you should always add documentation strings to all classes and methods. Triple quotes are used for multi-line strings.

We also add two more methods to the class definition: `print` can be used to make the object print itself, and `__add__` implements vector addition. See Listing 2.

[1] Many object-oriented programming languages instead defines *constructors*, methods that create and initialize the object. In Python the operations of object creation and object initialization are separated. While you can in principle redefine the constructor as well, doing so is usually a symptom of the programmer not really knowing what he is doing.

Listing 2 A Vector class supporting vector addition and printing.

```
class Vector:
    """A vector in two dimensions.

    Usage:
       Vector(x, y)
    """
    def __init__(self, x, y):
        self.set(x, y)
    def set(self, x, y):
        "Set the coordinates of the vector"
        self.x = x
        self.y = y
    def print(self):
        "Print a string representation of the vector"
        print("Vector({:.3f},{:.3f})".format(self.x, self.y))
    def __add__(self, other):
        "Vector addition"
        return Vector(self.x + other.x, self.y + other.y)

a = Vector(4,5)
b = Vector(3,4.4)
c = a + b
c.print()
```

It would be more natural to be able to use the vector directly in a Python print call, instead of calling a print method. This can be done with the magic method `__str__` which defines a string representation of the instance.

```
    def __str__(self):
        return "Vector({:.3f},{:.3f})".format(self.x, self.y)
```

5 Inheritance

Inheritance is one of the common relationships between classes, it represents that one class is somehow built from another class, often as a specialization of a more general class.

An example could be objects in a drawing program. A most general class could be `GeometricShape`, any geometric shape can draw itself on the screen and save itself into a file, but the top-level class `GeometricShape` does not contain any actual code for doing anything; it is an *abstract base class*.

The classes `Circle` and `Polygon` are two subclasses of `GeometricShape`. They contain actual code to draw a circle and a polygon. The class `RegularPolyhedron` is derived from `Polygon`, it is a specialized version that is easier to create than the more general class, but does not otherwise contain any special code. The class `Triangle` is also a subclass of `Polygon`, but in

this case it contains a special optimized drawing method. The classes could be defined as shown in Listing 3.

Listing 3 A hierarchy of classes in a drawing program

```
class GeometricShape:
    def __init__(self, color):
        self.color = color
        # some 'householding' code to register the object in
        # the framework of the program.

    def draw(self):
        raise NotImplementedError(
            "An abstract GeometricShape cannot draw itself.")

class Polygon(GeometricShape):
    def __init__(self, color, list_of_vertices):
        super().__init__(color)  # Call base class initializer
        self.vertices = list_of_vertices

    def draw(self):
        # Code to actually draw itself.

class RegularPolygon(Polygon):
    def __init__(self, color, n, center, size):
        # Code to calculate the vertices
        super().__init__(color, vertices)

class Triangle(Polygon):
    def __init__(self, color, list_of_vertices):
        if len(list_of_vertices) != 3:
            raise ValueError(
                "A triangle must have exactly three corners!")
        super().__init__(color, list_of_vertices)

    def draw(self):
        # More efficient drawing code using that this is
        # a triangle
```

Notes:

1. The `GeometricShape.draw()` method is not really necessary, and could be omitted. Methods like that are typically there to document how the family of classes can be used.[2]

[2]This is an example of an *abstract method*, a method without a useful implementation that is only intended as a placeholder to be overridden in a sub-class. Instead of making a method that throws an error when called, it is also possible to flag a method as being abstract in such a way that objects cannot be created unless an explicit implementation is being given. More about this later.

2. The initialize `Polygon.__init__` calls the `__init__` method of its base class. This is normally done using the helper function `super()` that gives access to the base class. If a base class contains an initializer you must always either call it, or replicate its functionality, otherwise you will get an incompletely initialized instance.

3. `RegularPolygon` extends the `__init__` method of `Polygon`, giving the user a simpler interface to create this kind of object. The `draw()` method is not reimplemented, so `RegularPolygon` has exactly the same `draw()` method as `Polygon`.

4. `Triangle` completely replaces the `draw()` method with an implementation that takes advantage of the simplicity of the triangle object to use a faster drawing algorithm.

6 Composition and aggregation

A common way to build one object from another is *composition*, where the new object contains the old object as one of its parts. An example could be in a program for molecular simulations. One part of such a program could be an object representing an atom. This Atom object contains data about the atom, such as its position and its atomic number.

One might first create a class representing a Position object.[3] The Atom object could then inherit from the Position class. While this would most likely be a completely viable solution, it seems more natural to say that the atom *has* a position rather than the atom *is* a position. That would be composition: the Atom object has an attribute (e.g. called pos) which is a Position object, and another which is an atomic number.

Sometimes it is less clear whether inheritance or composition is the right solution. Returning to the example of the drawing program in the previous section. It should be possible to save everything to a file, and one way of doing that is that each object can save itself to an open file. There may be other object besides the `GeometricShape` and its subclasses that need to be saved, for example the `Background`. Let us assume that the act of writing an object to file is so similar that it is most sensible to share the code for doing so.

One obvious solution would then be to let `GeometricShape` and `Background` inherit from a common base class, say `SaveableObject`, that would be a solution based on *inheritance.* When the user saves, the main program opens the file and then loops over all objects, calling `object.save()`.

An alternative solution would be that each object has a member object, the `ObjectSaver` that can be used to save the object. In this case the program would loop over all objects and call `object.saver.save()`. Instead of saying that the `GeometricShape` *is* an object that can be saved (inheritance), we say that is *has* the ability to be saved (composition).

It is clear from the example above that often the choice between composition and inheritance is somewhat arbitrary. It is worth bearing in mind that experience shows, that *although inheritance may be the most obvious choice for the programmer, composition often leads to code that is easier to maintain*.

Finally, it should be noted that the litterature distinguishes between *composition* and *aggregation* based on whether the sub-objects can exist on their own (aggregation) or only make sense as part of the composite object (composition). This distinction is rarely important, in particular not in a language like Python where the programmer is not responsible for memory management (i.e. explicitly destroying the sub-objects when the main object is destroyed).

[3] In reality you would probably just use a Numpy array with three elements — no need to reinvent the wheel. But for arguments sake let's assume you make your own class for a position.

7 Writing maintainable code

Any computer code tends to grow into an unmaintainable mess, unless thought is given to the structure. Writing maintainable code is not particularly difficult, but it is a habit one needs to get into. Fortunately, some of the tricks that makes code easier to maintain also makes it faster to write code. Maintainability is based on two simple principles: Make it clear what the code is supposed to do, and make it easy to understand the flow through the code — without needing to build a huge map of the code in your head.

A few sound principles:

Divide and conquer: Split complicated tasks into smaller tasks. Write the outline of your code first, postpone writing anything even slightly complicated by delegating it to a function call that you write later.

One task, one method (or function): A method should do only one thing. A high-level method does its complicated thing by calling lower-level methods.

Split up long methods: If your method becomes too long, consider splitting it up. You probably broke the previous rule!

Write documentation in the code: This is particularly easy in Python, every time you write a method, you start by writing the docstring. *Every single time!*

Split up large classes: If a class gains too many methods, consider splitting it up.

- Composition can be used to split off related methods.
- Thin wrapper methods may delegate tasks to a member class.

Split up large files: Module files should be short, one class in one file. More on modules below.

8 Modules and packages

Whenever you do something like

```
import numpy
```

you are importing a *module* or a *package*. The difference is minor, a package is like a module with submodules.

A Python package is implemented as a file, `module.py`. It can contain any valid Python code, and is executed the first time the module is imported (and *only* the first time). A module usually only contains class and function definitions, possibly with some initialization code setting up e.g. module-global variables or instantiating internal objects. Any symbol defined in the file will be part of the module.

An example code is shown in listing 4. If the module is imported with the statement

```
import silly
```

the class and the two variables will be accessible as `silly.Aplha`, `silly.alpha` and `silly.beta`, respectively.

Ideally, a module should only define a single class/function, or very few related classes/functions, and the file should not be so long that it is difficult to maintain an overview of it mentally — also when returning to look at it one year from now.

Listing 4 A somewhat silly sample module. The file name is `silly.py`.

```
class Alpha:
    def __init__(self):
        pass

alpha = Alpha()
beta = 42
```

If the code becomes too complex for a module, it can be split up into a *package*. A package is a folder containing multiple python files and possibly sub-folders with python files. A sample structure of a hypothetical package for handling audio files and audio data is shown in figure 1. The folders may contain python files defining sub-packages. The folders also all contain a file named `__init__.py`, those files are often empty, any code they contain is executed when the package or sub-package is imported.[4] A common use is to import (some of) the sub-packages. For example, when you `import numpy`, you are importing a package where the main `__init__.py` file imports the rest of the package.

In the example in figure 1, you may import a class defined in `sound/effects/echo.py` in any of the following ways

```
import sound.effects.echo
sound.effects.echo.EchoFilter(input, delay=0.7, atten=4)
```

or

```
from sound.effects import echo
echo.EchoFilter(input, delay=0.7, atten=4)
```

or

```
from sound.effects.echo import EchoFilter
EchoFilter(input, delay=0.7, atten=4)
```

9 Design patterns

Design patterns are standardized relations between objects, solving a typical problem in object-oriented programming. A number of standardized solutions to common problems have been developed and catalogued. Further information can be found on Wikipedia [2] and in the by now classical book by Gamma *et al.* [3]. Below are some examples of design patterns

Adapter Provides an alternative interface to a class, often a class provided by a third-party library that does not conform to the interface in our code.

Facade Very similar to the Adapter, it provides a new interface to one or more objects, but the purpose it to simplify the interface, often in the form of a higher-level but less flexible interface.

[4]The newest versions of Python supports packages without the initialization file, they work in slightly different ways than normal packages. Unless you know exactly what you are doing, avoid trouble and put an empty initialization file in your modules. As a benefit it will load marginally faster.

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Figure 1: A hypothetical package for handling sound files and sound data. The main package name is `sound`, it contains sub-packages `sound.formats`, `sound.effects` and `sound.filters`. Reproduced from the Python Tutorial section 6.4 [1].

Factory (often regarded as two or three related patterns) A way to create objects where the exact class of the object is determined at run-time.

Composite Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. An example could be molecules and atomic-scale systems, where the latter might be composed of multiple molecules.

Delegation Forward a call to another object which does the actual work. Common when composition is used to extend an object instead of inheritance.

Observer One or more *observer* objects are notified every time the state of a *subject* object changes.

10 Libraries for (data) science

The Python language comes with a rather extensive *standard library*, and the serious Python programmer should at least scan the index of the documentation of the Python Standard Library [4].

In addition, there is an enormous collection of packages that can be used for all kinds of tasks, inlcuding scientific processing.

10.1 Installing third-party libraries

The vast majority of third-party Python libraries are available from the Python Package Index, PyPI [5]. Packages distributed through PyPI can be installed with the `pip` command. Unfortuntately, as we are in a transition period where it is common to have both Python 2 and Python 3 installed, there is a real risk of getting the `pip` command belonging to the wrong version of Python. It is therefore a good habit to invoke the pip command through the Python interpreter, it is done with the `-m` command line option to the interpreter (the `-m` option means *run a named module*):

```
python3 -m pip install --user packagename
```

Here we are assuming that the python interpreter is called `python3`, if it is called `python` you should write that instead.

The `--user` option instructs pip to install the module for the current user only, instead of attempting to install it for all users on the computer (which will often fail).

Users of the Anaconda distribution need to be aware that Anaconda use their own parallel distribution system. You install new packages either through their graphical user interface (which is often incredibly slow), or with the command line

```
conda install packagename
```

If (and only if) the package is not provided through the Anaconda channels should you install with pip, in that case you do as above but leave out the `--user` option which is disabled on an Anaconda distribution as everything is installed for the user only.

10.2 The numpy package — numerics

The `numpy` package (for Numerical Python) is the core of all numerical work in Python. It provides highly efficient multi-dimensional arrays and operations on these. For an introduction to numpy arrays, see the Quickstart Tutorial [6].

An overview of the most important functionality of the numpy package

Element-wise operations Operations on all the elements of an array in parallel. This includes basic arithmetics as well as applying mathematical functions to all elements of an array.

Array shape manipulation Extracting sub-arrays, reshaping arrays etc.

Fancy indexing Arrays can be indexed with arrays of integers or arrays of bools to select specific elements.

Linear Algebra Basic linear algebra is provided directly in the `numpy` package, more advanced functions are in the `scipy` package.

Random numbers Arrays of random numbers drawn from all the common distributions.

10.3 The scipy package — scientific computing

The `scipy` package is `numpy`'s big sister, maintained by the same group of people. It contains all kinds of more advanced scientific algorithms, almost always operating on numpy arrays. A few highlights

Special functions (`scipy.special`) Bessel functions, Gamma function, spherical harmonics, Legendre functions and many many others.

Integration (`scipy.integrate`) Algorithms for numerical integration of functions and of ordinary differential equations.

Optimization (`scipy.optimize`) Minimization of functions in many dimensions. Global optimization. Root finding.

Interpolation (`scipy.interpolate`) Interpolation in one or more dimensions.

Fourier transforms (`scipy.fft`) Discrete Fourier Transforms. For performance critical use, install the FFTW library and the `PyFFTW` package.

Linear Algrebra (scipy.linalg) Advanced linear algebra.

Spatial data structures and algorithms (scipy.spatial) Operations on sets of points in n-dimensional space.

and many others ...

10.4 The matplotlib package — plotting data

Matplotlib has become the defacto standard for plotting data in Python. Unfortunately, matplotlib is a huge package with a complex user interface. Furthermore, the developers decided to make two different interfaces, one emulating how plotting works in MATLAB, and one with a more "pythonic" object-oriented interface, and both the examples in the documentation and the examples you will find on the internet tend to mix the two in a less-than-elegant melange.

The documentation for matplotlib is very thorough, very complete, and gives a detailed description of what every function and method does. It is less easy to find out *which* function to use to get a desired effect. Fortunately, matplotlib is so widely used that it is overwhelmingly likely that somebody else has had the same question, and has had it answered on one of the knowledge-sharing platforms (mainly StackExchange and StackOverflow). Google is the most important tool when using matplotlib!

The important functions are all in the `matplotlib.pyplot` subpackage, which is traditionally imported like this:

```
import matplotlib.pyplot as plt
```

if you are using a Jupyter notebook, you first enable a tighter integration between matplotlib and the notebook with a Jupyter-specific command:

```
%matplotlib notebook
import matplotlib.pyplot as plt
```

In the MATLAB-like interface, matplotlib operates with a "current figure" and a "current axis" (subplot) in each figure. You can switch the active figure, and plot to it. An example is shown in Listing 5 and the result in Figure 2.

In the pythonic interface, you create e.g. a figure with subplots, and get an object for each subplot. You can then use the objects to plot. This is shown in Listing 6, the result is identical to the previous example.

Listing 5 Plotting two graphs using the MATLAB-like interface. The result is seen in figure 2.

```
import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'o-')
plt.title('A tale of 2 subplots')
plt.ylabel('Damped oscillation')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, '.-')
plt.xlabel('time (s)')
plt.ylabel('Undamped')

plt.show()
```

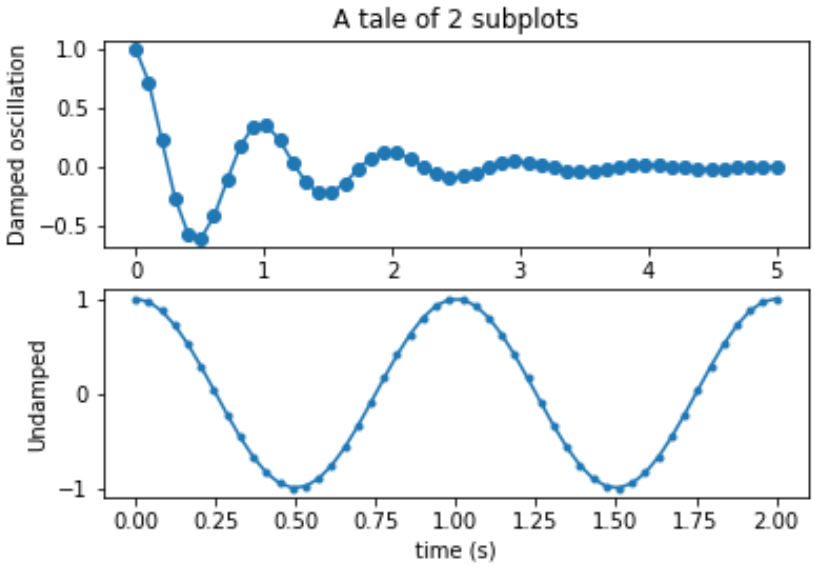

Figure 2: The plot resulting from the code in listing 5 or 6.

Listing 6 Plotting two graphs using the object-oriented interface. The result is identical to the result from Listing 5 seen in figure 2.

```
import numpy as np
import matplotlib.pyplot as plt


x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

fig, (ax1, ax2) = plt.subplots(2, 1)

ax1.plot(x1, y1, 'o-')
ax1.set_title('A tale of 2 subplots')
ax1.set_ylabel('Damped oscillation')

ax2.plot(x2, y2, '.-')
ax2.set_xlabel('time (s)')
ax2.set_ylabel('Undamped')
fig.show()
```

10.5 The pandas package — data handling for data science

The `pandas` package is intended to make it easier to represent data for data science and machine learning. It is mainly focussing on one- and two-dimensional data sets, where the two-dimensional sets may be seen as columns of different data kinds. Pandas support a number of different data types, including date-time data, and supports that some data points are missing. It also supports easy addition of new data in existing structures.

Instead of providing an overview here, we refer to the documentation, in particular the Package Overview [7] and the 10-minutes introduction [8].

11 Tutorial exercise: Building a simple molecular dynamics program

In the following you will build a collection of classes for a molecular modelling program. Later in the course you will be using an established framework for working with atomic simulations, the Atomic Simulation Environment, but to learn about object oriented programming it is more useful to write a simple such environment.

11.1 Part I: The atoms

The first task is to write the classes representing the atoms of the system. We will need two classes, an `Atom` class representing a single atom, and a class representing the system, you can call it `AtomicSystem`, `Molecule` or something similar.

The `Atom` class should have the following properties.

- A chemical symbol (e.g. a string).
- A position, this would typically be three numbers and could be a one-dimensional numpy array of length 3.
- A representation: It should be possible to print the atom and see what it is.

The atomic system should have the following properties.

- It should behave like a Python list with the elements of the list being the atoms.
- It should have convenience functions for accessing the chemical symbols and positions of all N atoms:
  - `atoms.get_symbols()` should return a list of the chemical symbols.
  - `atoms.get_positions()` should return an $N \times 3$ numpy array containing the positions of all the atoms
- It should be able to print itself to a file in a standardized format so the atoms can be visualized with standard tools.

There are two ways to create a new class that behaves like a Python list. One can define the "magic" methods needed to behave like a list, i.e. methods for accessing elements, calculating the length of the list, iterating over the list etc. There are base classes than can help with this. But the simpler alternative is to define a class that inherits from the standard `list` class in Python. Remember when doing this that your new class is also a list, and in the methods you define you will have access to all the operations of a normal list, i.e. you can use `len(self)` to find the length of the list, etc.

As for writing the atoms to a standardized format, the simplest format is the so-called XYZ format. An XYZ file is a text file with extension `.xyz` containing the following lines:

Line 1: The number of atoms.

Line 2: A description of exactly one line of text. The line may be empty (no description).

Remaining lines: One line per atom. Each line has four fields, separated by one or more spaces. The first is the chemical symbol (max two letters), the remaining three are the coordinates as three numbers (in Ångström).

Test it! Make a small molecule, e.g. water (you can google the OH bond length and the angle between the two bonds). Make the atoms write themselves to a file, and view it using the Atomic Simulation Environment (ASE) graphical user interface. In a terminal, write

```
ase gui water.xyz
```

A window showing the molecule should appear. You can rotate it with the mouse by right-clicking on the window.

Note: If the `ase` command is not found, install ASE as discussed in Appendix A.

11.2 Part II: The interactions between atoms

An important part of a molecular simulation is a model for how the atoms interact with each other. It makes sense to program this as a separate class, since one may want to be able to have different models for the same kind of systems.

Examples of models typically used in this kind of simulations, in increasing order of accuracy and required computer power.

- A pairwise interaction between atoms, e.g. the Lennard-Jones potential.
- A more complex but classical model, e.g. a *bond-order potential*.
- A machine-learning model trained on quantum calculations.
- A full quantum calculation within one of the multiple frameworks for doing such calculations.

As there is a need for replacable objects, we define an interface that they can all follow. A simple interface could be the one shown in Listing 7.

Listing 7 The interface for the Model classes.

```
class Model:
    def __init__(self, ...):
    """Create the Model.

    The parameters are model-dependent."""

    def calculate(self, system):
    """Calculate energy and forces.

    Returns two values.  The first is the energy (a number),
    the second is the force on all atoms as a (N,3) numpy array.
    """
```

In the following, we will use one of the simplest possible models: the Lennard-Jones potential. In the Lennard-Jones potential, the energy is given by

$$E = \sum_i \sum_{j>i} 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] \tag{1}$$

$$\vec{F}_k = -\frac{\partial}{\partial \vec{r}_k} E \tag{2}$$

where $r_{ij} = |\vec{r}_i - \vec{r}_j|$ is the distance between atoms i and j, and where ϵ_{ij} and σ_{ij} are the Lennard-Jones parameters, they depend on the atoms as they depend on the atomic number of the interacting atoms.

Calculating the forces is a simple application of the chain rule (the factor $\frac{1}{2}$ is because we now sum over all $j \neq i$ instead of over $j > i$):

$$\vec{F}_k = -\frac{1}{2} \sum_i \sum_{j \neq i} 4\varepsilon_{ij} \left(-12 \frac{\sigma_{ij}^{12}}{r_{ij}^{13}} + 6 \frac{\sigma_{ij}^{6}}{r_{ij}^{7}} \right) \frac{\partial r_{ij}}{\partial \vec{r}_k} \tag{3}$$

$$\frac{\partial r_{ij}}{\partial \vec{r}_k} = \frac{\partial}{\partial \vec{r}_k} \sqrt{(\vec{r}_i - \vec{r}_j)^2} = \frac{\vec{r}_i - \vec{r}_j}{\sqrt{(\vec{r}_i - \vec{r}_j)^2}} \cdot (\delta_{ik} - \delta_{jk}) \tag{4}$$

and using the Kroenecker deltas to kill one of the summations, we get

$$\vec{F}_k = \sum_{j \neq k} 4\varepsilon_{jk} \left[12 \left(\frac{\sigma_{jk}}{r_{jk}} \right)^{12} - 6 \left(\frac{\sigma_{jk}}{r_{jk}} \right)^{6} \right] \frac{\vec{r}_k - \vec{r}_j}{r_{jk}^2} \tag{5}$$

where the *two* Kroenecker deltas have resulted in the factor $\frac{1}{2}$ being eliminated (assuming that $\varepsilon_{ij} = \varepsilon_{ji}$ and similar for σ).

Your task: Write a `LennardJonesModel` class following the interface in Listing 7 and implementing the Lennard Jones potential as described in equations (1) and (5).

Simplifying assumption: You can assume that there is only one kind of atoms present. Then there is only one ε and one σ. But you should then test that all atoms are the right kind when `.calculate()` is called. It would be reasonable if the model can be created using a call like this:

```
ljmodel = LennardJonesModel(element, epsilon, sigma)
```

You can then later remove the restriction that only one element is present, if you want.

Hint 1: Both when calculating the energy and the forces, you will need a double sum over all atoms — for the energy because the expression (1) contains a double sum, for the forces because you calculate the forces on all atoms, and each force contains a sum over the other atoms. You can therefore combine the two calculations into a single double sum, adding up contributions to the energy in a scalar variable, and to the forces in an array.

Hint 2: The expressions only depend on *even* powers of the interatomic distances. You therefore never need to calculate the square root in the expression for r_{ij}.

11.3 Part III: The dynamics

The final part of a molecular dynamics program is the dynamics: a numerical integration of Newton's second law. The idea is to step forward in time with a (usually fixed) time step, using the forces to upgrade the velocities of the atoms, and the velocities to upgrade the positions. A straight-forward implemenation of this idea naturally leads to *Euler's algorightm*, which is numerically unstable. A simple yet stable algorithm is the *velodity Verlet algorithm*:

$$\vec{v}\left(t + \tfrac{1}{2}\,\Delta t\right) = \vec{v}(t) + \tfrac{1}{2}\,\vec{a}(t)\,\Delta t \tag{6}$$

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{v}\left(t + \tfrac{1}{2}\,\Delta t\right)\,\Delta t \tag{7}$$

$$\vec{v}(t + \Delta t) = \vec{v}\left(t + \tfrac{1}{2}\,\Delta t\right) + \tfrac{1}{2}\,\vec{a}(t + \Delta t)\Delta t \tag{8}$$

with

$$a(t') = \frac{1}{m}\vec{F}\left(\vec{x}(t')\right) \tag{9}$$

To use this, we need to be able to update the positions of our atoms. Once that is done, a new object implementing the velocity Verlet algorithm can be made.

- Update the atomic system class in part I (section 11.1) with an `atoms.set_positions(x)` method.
- Decide if the velocities of the atoms should be stored on the atoms (requiring adding new methods to it) or in the velocity verlet algorithm.
- Write a velocity verlet algorithm object. It should take an atomic system and an interaction model as arguments, and have a `.run(N)` method taking N time steps.
- Add a way to write a line to a log file for each time step.
  - Deside if this should be done by the dynamics object or by a separate object. If it is a separate object, is there a relevant design pattern?
  - You should log the total potential energy, the total kinetic energy and the sum of the two.
  - The total energy should be constant when you run the algorithm, otherwise either your time step is too large or there is a bug in the calculation of the forces and/or energies.
- Collect the various classes you have built into a molecular dynamics package that can be used to script a simulation.
- Optional: You already have a way to store the final configuration of the atoms as an XYZ file. If there is time, you may want to write a function that reads atoms from an XYZ file so you can restart a simulation. This can be done as a helper function or as a class method (as it does not operate on a specific instance of the atoms, but creates one).

A Installing Python and Jupyter on your own computer

A.1 Windows installation

1. Download the Anaconda Distribution from `https://www.anaconda.com/` (click on Download, then chose the Python 3.X download).
2. Install it. Do not change the default settings it suggests, in particular installing Anaconda for all users may result in future breakage.
3. Install the Atomic Simulation Environment. It is not available through the Anaconda packaging system, but can be installed by opening an Anaconda Prompt window and typing

```
python -m pip install ase
```

A.2 Mac installation

On a Mac, you can either install Python system-wide with the Homebrew package manager, or install Anaconda. We have the best experiences with the Homebrew version, so unless you already have Anaconda installed we recommend it. It will also give you the leanest installation.

A.2.1 Installing Python with Homebrew on a Mac

1. Open a Terminal window. You can find Terminal in the LaunchPad in a folder called either Other or Utilities. Or you can open Spotlight (Command-Space) and type Terminal.
2. Open a browser, and go to `https://brew.sh`
3. Follow the instructions: cut-and-paste the cryptic line into the Terminal window. Along the way it will ask for your password to the Mac so it can install what it needs.
4. Open a new Terminal window (Command-N) and then close the old one.
5. In the new window, type

```
brew install python
python3 -m ensurepip
python3 -m pip install ase scipy scikit-learn jupyter
```

Note that now `python` is the system-installed Python 2.7, and `python3` is the Homebrew installed `python3`. If you want `python` to refer to Python 3, add the folder `/usr/local/opt/python/libexec/bin` to your `$PATH` — doing so is safe.

A.2.2 Installing Python with Anaconda on a Mac

Please follow the installation for installing Anaconda for Windows, the procedure is essentially the same (section A.1).

A.3 Linux installation

Some newer Linux distributions come with Python 3 installed, other only come with Python 2. Check what you have by opening a Terminal window, and typing `python` or `python3`. If you do not have Python 3, use the package manager of your Linux distribution to install it, the package is usually called something like `python3`. You should also install the package `python3-pip`, if you can find it.

Once Python 3.X is installed (you should be OK regardless of whether you got Python version 3.6, 3.7 or 3.8), you can install Jupyter, scipy, scikit-learn and ase. If you can find these packages in the package manager (for Python 3), you can install them this way.

Otherwise, install them manually from the command line. First, make sure that the folder `$HOME/.local/bin` is on your PATH, for example by adding this line to the `.bashrc` file:

```
export PATH=$HOME/.local/bin:$PATH
```

Make sure that there are no spaces except after the `export` keyword, and open a new terminal after editing the file to ensure that it takes effect. Then install from the command line

```
python3 -m ensurepip --user
python3 -m pip install --user scipy
python3 -m pip install --user jupyter
python3 -m pip install --user scikit-learn
python3 -m pip install --user ase
```

(Omit any line if you have installed the package through the package manager).

References

[1] "The python tutorial." [Online]. Available: https://docs.python.org/3/tutorial/index.html

[2] "Wikipedia: Software design pattern." [Online]. Available: https://en.wikipedia.org/wiki/Software_design_pattern

[3] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

[4] "The python standard library." [Online]. Available: https://docs.python.org/3/library/index.html

[5] "The python package index (PyPI)." [Online]. Available: https://pypi.org

[6] "The numpy quickstart tutorial." [Online]. Available: https://docs.scipy.org/doc/numpy/user/quickstart.html

[7] "Pandas: Package overview." [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/getting_started/overview.html

[8] "10 minutes to pandas." [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html