Exercise 1

Molecular Statistics, Week 1

Introduction

The exercises are split into two parts: A general introduction to new concepts and programming tricks and the main part which is to simulate physical systems. It is very important that you go trough each exercise, type each example code in exactly and run it. That means, do not copy-paste. The point of the exercises is to train in how to write, read and debug code.

There is no Python curriculum, but two online books, namely learnpythonthehardway.org/book and pymbook.readthedocs.org/en/latest, contains all the concepts with examples. However, this being a programming course we require you to use Google for most information searching and debugging.

1 Python Introduction

First thing first, we want to create our very first program. To run programs with python we save a file with the extension `.py` and run it via the terminal window. Try to write the following python code and save it as `example_program.py`

```python
# you can use comments underway to remind yourself later what the code should do.
print "I want to print this statement" # you can also comment after a line
```

run it as

```
python example_program.py
```

Is the output as expected?

1.1 Python as a calculator

Now, let's start by using Python as a calculator. Remember to print the result of your calculation using the `print` statement. We encourage you to have some sort of logical file system, saving all the exercises in different files and folders so you can find it again when needed.

Some useful commands for navigating your filesystem:

```
cd <directory> # enter the folder called <directory>
cd ..  # navigate to the parent directory.
ls # print the contents of the current directory
mkdir <directory> # creates an empty folder called <directory>
cp <file1> <file2> # copy <file1> to <file2>
cp <file> <directory> # copy <file> to <directory>
cp -r <directory1> <directory2> # copy <directory1> to <directory2>.
mv <file1> <file2> # move <file1> to <file2>
```

1. Execute the following statements. Does it behave like your regular calculator? If anything unusual happens try and explain why. *Hint*; what is the difference between **integer** and **float**?

   `5+9, 5-9, 5*9, 5/9, 5+2*9, 5.0+2.0*9.0, 5.0/9 5**2, 5%2`

Since manually entering numbers really does us no good, we might as well use a calculator. Instead, we want to utilize what programming languages can provide, namely storage of values in what is known as **variables**. Variables can store anything that you can think of, i.e. numbers, strings, lists of numbers and so forth. Variables are assigned values by specifying a variable name and a value, e.g.

```python
my_first_variable = 5.0
```

where a variable named `my_first_variable` has been assigned the value of 5.0. Some restrictions apply for variable names (must start with a letter, can't have names of in-build functions etc.), but otherwise we are not restricted in the naming. We encourage you to give them useful names, such as `no_particles` for representing number of particles. That way you can more easily remember what values are stored in the variable and it makes the code more readable for others.

Since a variable can contain anything, a very useful function is `type(arg)`, where `arg` is a variable. This function will return the type of value stored in the variable.

2. Execute the same statements from task 1, but using variables to store the values and results. Print the results.

3. Use `type` to print out the type of the variables used.

Usually when working with a program we work with a range of numbers that we need to manipulate, and for this it is fairly useful to create a Python **list** of numbers. A list is defined with the same syntax as a float variable;

```
simple_list = []
```

The above code creates an empty list named `simple_list`. Even though the list is empty we can still print the content of the variable (try it). If you want to add an element to the list you can append it using `list_name.append(arg)` where `arg` is the element to append the list.
For example if you want to append 2.0 to a list you write;

```
simple_list.append(2.0)
```

4. Create an empty list. Print the empty list. Append a number and print it again. What changed in the output?

5. Print the type of the variable.

6. Add the following numbers to the list: -1.0, 1.5, 2.0, -2.0, -3.0, 3.0 and print the content of the variable again.

Different variables have different attributes, which are called *methods*. These methods are executed using 'dot'. For instance, the variable of type list has the method `sort()`, which can be called as follows;

```
simple_list.sort()
```

For a list of floats, this results in the items of the list being sorted by value.

Another way of populating a list with numbers is to do it directly when we create the list. This is done almost the same way as when we created the empty list, except we provide the initial content right away. E.g.

```
another_list = [1.0, 2.0, 3.0, 4.0, -2.0, -4.5, -1.0]
```

7. Define `another_list` and repeat task 6.

8. Print the list. Sort the list. Print the sorted list. Does it provide the correct result?

9. Print `another_list` in reverse sorted order. Use Google to find out how.[1]

We've now seen that python lists can be created in various ways and even be sorted, but it is quite tedious to enter all data manually. Especially if there is a lot of it. Luckily, Python provides us with the means to construct lists using other approaches. The `range()` command is one of them and is probably the command that you will spend the most time with during this course.

10. Write the following commands in the python shell and explain the results before moving on.

    Hint: use the type command to get the data type of the commands, i.e. `type(range(10))`

    ```
    print range(10)
    print range(3, 10)
    print range(-3, 10, 4)
    ```

11. Understand how the range function works.

If we want to access the i'th element in a list we use square brackets. To print the second element in the list from above you write;

```
print another_list[1]
```

[1]Using Google is by far the most important tool as a programmer.

Notice that we wrote 1 and not 2. This was not a typo! This is because the list index starts at 0 and not 1.

To get the length of a list you use the function `len(arg)` where `arg` is the variable with the list. If you want the length of our example list, the syntax is

```
print len(another_list)
```

12. Initialize the following list and test your knowledge about lists with the following print-statements. Describe the result of every print-statement. Test your understanding of lists by guessing what the output will be before you run it. *Hint;* one of the lines will give an error. Why?

```
q_list=[45, 23, 56, 34, 76, 50]

print q_list[3]
print q_list[0]
print q_list[-1]
print q_list[len(q_list)]
print q_list[len(q_list)-1]
```

13. What is the index of the first item in a list? What is the index of the last item?

Creating lists from lists. Say that we have a list `x_list` and we want to create y values as a function of these x values. To do this we want to iterate over the elements of `x_list` to create the new list. This is where we want to use *for-loops*. Let's jump straight into the syntax. Say that we already have defined a variable containing the x-values, `x_list`, then `y_list` is created as,

```
y_list = []
for x in x_list:
    y = x**2
    y_list.append(y)
```

First we initialize a new list, then we fill in y values by iterating over all x values in `x_list`. As you might have guessed the above code is equivalent to the function $f(x) = x^2$.

14. Create a list of x values from -5 to 5 (both included). Use this list to calculate values for another list, with the function $f(x) = -6x^2 + 6x$. Print the result.

Another one-line way of working with lists is the following syntax;

```
  y_list = [x**2 for x in x_list]
```

which does exactly the same as the above example[2].

15. Repeat exercise 14, but using the shorthand way of creating lists.

What if we want to use math functions (or other modules)? We can import them! This is always done in the top of the python file using the syntax;

```
import math
```

The math module has a lot of useful mathematical functions like `sin`, `cos` and `exp`. When the math package has been imported it is used in the following way;

```
print math.cos(math.pi)
print math.exp(math.pi)
```

16. Create a list of x values from -5 to 5 (both included). Calculate sine values, $f(x) = \sin x$ using the math module, for the x-list. Print the result.

17. Use the matplotlib-manual (found on the website) to save plots of task 14 and 16.

[2]This method is called "list comprehension". Knowing what methods are called makes it easier to get help from Google when you get stuck.

2 Non-Interacting particles

The goal of today's simulation is to initialize particles with random coordinates and random velocities confined in a 2 dimensional box, and propagate the particle positions in time. The code below is your starting point for today's exercise.

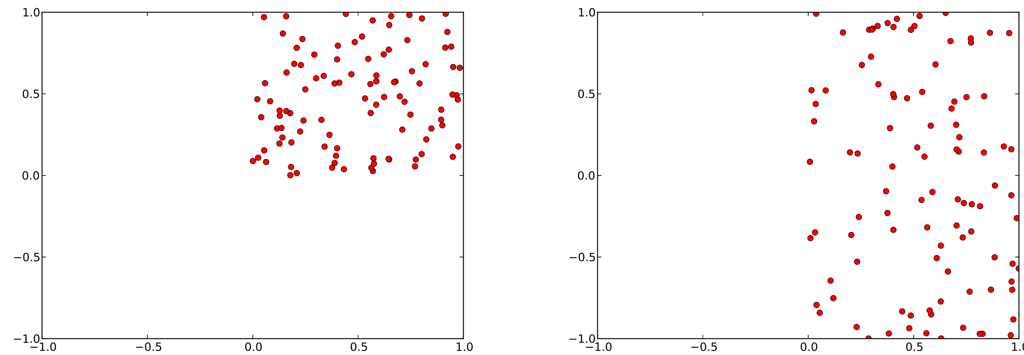
```python
# import modules
import numpy as np
import matplotlib.pyplot as plt

# initialize simultation variables
n_particles = 50
box_width = 10.0
n_steps = 1
dt = 0.001

# create the x- and y-coordinates
positions_x = [np.random.random() for i in range(n_particles)]
positions_y = [np.random.random() for i in range(n_particles)]

# plot the x- and y-coordinates in a figure.
plt.plot(positions_x, positions_y, 'ro')
plt.axis((-box_width, box_width, -box_width, box_width))
plt.savefig('coordinates_start.png')
```

As you can see, variables have been already been declared such as `n_particles`, which is the number of particles that we want to generate and simulate. `n_steps` is the number of steps we want to take in the simulation and `dt` controls how large a time-step we will be taking. Lastly we have defined the x-coordinates and y-coordinates for `n_particles` random particles using list-comprehensions.

1. Inspect and understand all lines in the above code. Line by line, explain what the code does. Use `#` to comment your own code. You should print the `np.random.random()` function to understand what the output is.

See figure 2.1a to see what we are trying to simulate. Now we want to change the code so that it looks like 2.1b.

1. Correct the y-coordinates of the particle positions by making them initialize in the range $y \in [-10, 10]$ instead of the standard $y \in [0, 10]$.



(a) **(b)**

Figure 2.1: Pyplot plot of particles in 2 dimensions confined in a box, spanning $x, y \in [0, 1]$ for (a) and $y \in [-1, 1]$ for (b).

When you have corrected the particle positions, it is time to give the particles random velocities to allow the particles to move around.

3. Create two new lists, `velocities_x` and `velocities_y`, which we shall use to store the velocities for the particles. The velocities should be a random number between -10 and 10.

If you want to visualize the velocity vectors for each particle you can use the following function.

```
plt.quiver(positions_x, positions_y, velocities_x, velocities_y)
```

We are now ready to loop over each particle in our system, but before we do this, you should make it really clear to yourself how we can obtain the coordinates and velocities for the i'th particle.

4. How do we access the value of the i'th element in the `positions_x` list? *Hint;* check out task (12) from part 1 again.

In this weeks simulation there are no forces that acts on the particles, and only their initial velocities influence their positions. Hence the location of the i'th particle at time $n + 1$ can be calculated from the current time n's coordinates and the velocity of the particle.

$$x_i^{(n+1)} = x_i^n + v_i^n \cdot dt \tag{2.1}$$

That is, the i'th particle will be at its previous position plus its velocity v_i times a time step dt.

5. Modify your script to loop over each particle and update the x- and y-coordinates with the respective particle velocities. Plot the particle positions after you have changed them to a file called stepcoords.png. Does the particles appear to have have moved?

6. Modify your code to repeat the displacement of the particles `n_step` times. Remember to update the positions for each step.

7. Where are the particles after 10 steps? 100 steps? 1000 steps? 10000 steps?

What you simulate is how particles in vacuum behave if they can not feel each other and have kinetic energy.

Now we would like to visualize the simulation we have made, because it is a bit hard to see if the simulation has been implemented correctly by just looking at a few images. What we would like to do is to save a video of what we have done. To make things easy (as any good cooking show), we have prepared a method for you.

On the course website, there is a 'md_video.py' file. Save this file, and put it in the same folder as the simulation file. The function is implemented as following;

```python
import video

...

for n in range(n_steps):

    ...

    # Save a frame every 10 steps
    if n % 10 == 0:
        video.add_frame(positions_x, positions_y)

...

video.save('week1_video')
```

where the import statement should be positioned in the head of the file. A video will be created called `week1_video.mp4`

8. Run the simulation from task 9 for `n_steps` = 5000 and save a video. Is the video of the simulation as you expect?

What remains is to keep the particles inside a box, i.e. they should make an elastic reflection on the walls and change direction.

To change the direction on the i'th particle, we must change the sign of the velocity for that particle. For simplicity, we shall add a box which corresponds to the region we are plotting, that is $x \in [-10, 10]$ and $y \in [-10, 10]$. We start out with the x-coordinates and then, when we have confirmed that it is working, we move on to the y-coordinates.

9. Modify your code to check if the updated position of particle i would violate $x \in [-10, 10]$. If the particle is outside this boundary, change the sign of the x-velocity of that particular particle before the displacement is made. *Hint*; you will need if-statements to solve this problem. Also, it is a good idea to draw your plan for what happens with a pen and paper.

10. Repeat for the y-direction.

Is the simulation running as it should be?