Exercise 1

Molecular Statistic, Week 1

2014

1 Introduction

This is the first exercise in the course Molecular Statistics. The exercises throughout the course are split into two parts. First part is a general introduction to new concepts and programming techniques. Second part is to use Python to simulate physical systems.

All concepts and syntax can be seen in the curriculum ”Think Python” for each week, or using the powerful tool called ”Google”.

2 Python

To execute a program written in Python, we save a file with the extension '.py' and run it via the terminal using the following syntax;

```
python example_program.py
```

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

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`

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, e.g. numbers, strings, lists of numbers and so forth. Variables are assigned values by specifying a name and a value, e.g.

```
my_first_variable = 5.0
```

where a variable named `my_first_variable` has been assigned the value of 5.0. We are not restricted in our naming of variables, but 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.

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 execute, and so it is fairly useful to create a Python **list** of numbers. A list is defined in 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, and are called *methods*. These methods is executed using 'dot'. For instances, the variable of type list has the method `sort()`, and can be called like this;

```
simple_list.sort()
```

to sort the list.

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. eg.

```
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 sorted, but it is quite tedious to enter the data manually, especially if there is a lot of it. Luckily, Python provides us with the means to construct lists using various other approaches. The `range()` command is one of them, probably the command you will spend 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.

Now if we want to access the i'th element in a list you 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. *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 list?

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, then we want to iterate over the list elements and create a 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 variables containing x-values, `x_list`, then the `y_list` is filled out 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 have a 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.

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 head 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(180.0)
print math.exp(180.0)
```

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

17. Use the pylab-manual (found on the website) to save plots of task 14 and 15.

3 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.

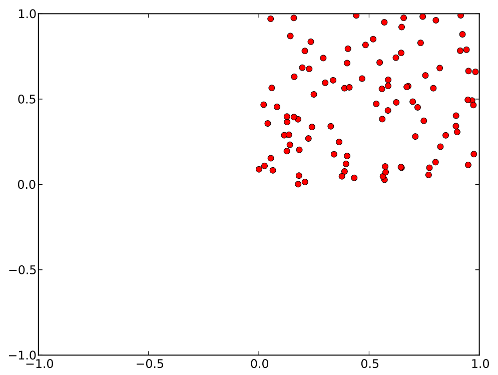
```python
# Import pylab and a function to create random numbers
import random
import pylab

# initialize some variables
n_particles = 100
n_steps = 1
dt = 0.001

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

# plot the x- and y-coordinates in a figure.
pylab.plot(x_positions, y_positions, 'ro')
pylab.axis((-1, 1, -1, 1))
pylab.savefig('coordinates_start.png')
```

As you can see, a lot of variables have 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 a 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.

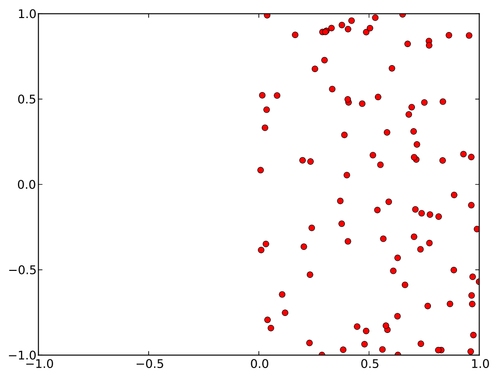
1. Inspect and understand all lines in the above code. Line by line, explain what each line does. You should print the `random.random()` function to understand what the output is.

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

See figure 3.1a to see what we are trying to simulate. Now we want to change the code so that it looks like 3.1b.



(a)



(b)

Figure 3.1: Pylab 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 named `vel_x` and `vel_y` which we shall use to store the velocities for the particles. They should be random, as the positions were and equal possibility to go in all directions.

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

```
pylab.quiver(pos_x, pos_y, vel_x, vel_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 `X` list? *Hint;* check out task (12) from part 1 again.

In this weeks simulation there are no forces to act on the particles, only its initial velocity, to influence its position. Hence the location of the i'th particle at time $n+1$ can be calculated at the current time n's coordinate and the velocity of the particle.

$$x_i^{(n+1)} = x_i^n + v_i^n \cdot dt \tag{3.1}$$

which tells us that the i'th particle will be at its previous position plus its velocity at the previous position times a time step.

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 to verify that your particles indeed 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? Make a plot of the region (-1, 1, -1, 1) as well as (-10, 10, -10, 10) to help answer the question.

What you simulate is how particles in vacuum would behave if they cannot feel each other and have kinetic energy. 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 $x \in [-1, 1]$ and $y \in [-1, 1]$. For simplicity we start out with the x-coordinates first and then, when we have confirmed that it is working, we move onto the y-coordinates.

8. Modify your code to check if the position plus velocity of particle i is outside the boundary of the box. If it is outside the box, change the sign of the velocity of that 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.
9. Repeat for the y-direction.

Now we would like to visualize the simulation we have made, because it is a bit hard to see if 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 (and as any good cooking show), we have prepared a method for you.

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

```
import video

...

for n in range(n_steps):

    ...

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

...

video.save('week1_video')
```

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

10. Run simulation from task 9, for `n_steps` = 1000 and save a video. Is the video of the simulation as you expect?