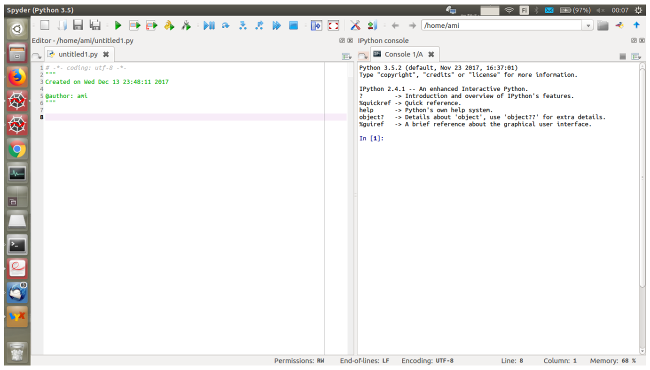
KEB-45250 Numerical Techniques for Process Modelling
Exercise 1 - Python Tutorial
11.01.2018

Antti Mikkonen

1 Introduction

On this course we use two computational tools: Python and ANSYS. In this exercise session, we concentrate on Python. An intensive hands-on-course on ANSYS will follow.

The purpose of this exercise session is to familiarize us with Python in the context of scientific calculations. We will start with the very basics and proceed to library usage. No prior experience with Python is necessary, but programming is not explicitly thought on this course. If you feel uncertain about your programming and/or Python skills in general we recommend the official Python documentation (https://docs.python.org/3/tutorial/) or the TUT basic programming course "TIE-02100, Introduction to Programming".



```
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 13 23:48:11 2017

@author: ami
"""

```

```
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]:
```

Figure 1: Spyder3

2 First steps

There are great many ways to access the power of Python but we use Spyder3 in this course. Note that we use Python3. Python2 is still widely in use but obsolete. If you want to install the necessary tools on your own computer, we recommend the Anaconda package (https://www.anaconda.com/download/#windows) on Windows and OSX. On Linux, you can also use Anaconda, but I would recommend using your favorite package manager.

Start up Spyder3 and you'll see something like that on Fig. 1.

On the right, you'll see IPython interpreter. **Start typing simple math in to the IPython**.

```
3+2
4*2
3**3
```

Note that ** is the exponent operator in Python. i.e. 3**3 means 3^3. You'll notice that the IPython acts like a calculator.

Now, create some variables and perform simple math on them. To print a variable on screen use the "print" command.

```
a=3
b=2
c=a*b
print(c)
```

This is convenient for extremely simple cases but quickly becomes tiresome. So let us **start scripting**. Now type your simple math in the editor window on the left and press **"F5"**. The script you such created will run in IPython just like if you had written it there.

Now you have made your first Python script.

3 First toy problem

Now that we have familiarized ourselves with Python scripting, lets do something useful. Consider the pressure drop in a pipe

$$\Delta p = \frac{1}{2}\rho V^2 \frac{L}{d} \mathrm{f} \tag{1}$$

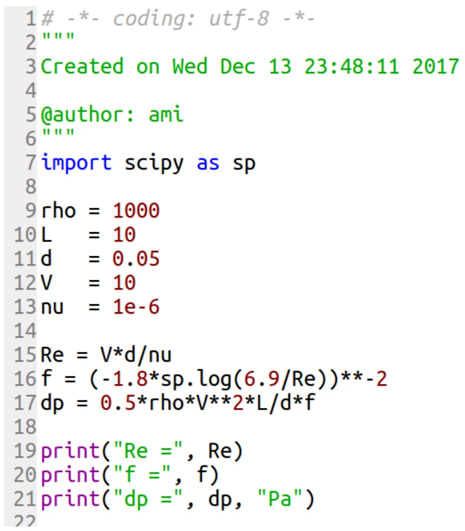
$$\frac{1}{\sqrt{f}} = -1.8 \log\left(\frac{6.9}{\mathrm{Re}}\right) \tag{2}$$

$$\mathrm{Re} = \frac{Vd}{\nu} \tag{3}$$

where Δp is pressure drop, ρ is density, V is velocity, L is pipe length, d is pipe diameter, f is Darcy friction factor, Reis Reynolds number, and ν is kinematic viscosity. This is a straight forward problem to solve, but note the log-operator in Eq. 2. Log-operator is not available in the standard Python so we need to import a library. Libraries are pre-existing collections of code that we can use in our own code. With Python, you can usually find a free library to do almost exactly what you want.

Libraries are imported with "import library_name" command in Python. A library can be given a simpler name with a "as" keyword as shown in Fig. 2. Now we can access the log-function as "sp.log".

Use the same input values as shown in Fig. 2 and solve the problem.



```python
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 13 23:48:11 2017

@author: ami
"""
import scipy as sp

rho = 1000
L   = 10
d   = 0.05
V   = 10
nu  = 1e-6

Re = V*d/nu
f = (-1.8*sp.log(6.9/Re))**-2
dp = 0.5*rho*V**2*L/d*f

print("Re =", Re)
print("f =", f)
print("dp =", dp, "Pa")
```

Figure 2: First toy problem

What we just did could have been easily done with a hand a held calculator. Manual calculations are, however, slow and error prone. One you write a script you can easily vary the input parameters. Try to play.

4 Solving a large number of toy problems

Now let us solve the toy problem in Section 3 for ten different values of velocity and plot the resulting pressure drop. Let $V = 1, 2, 3, ..., 10 m/s$.

One way to solve this problem is to create a list of the velocity values and the loop through them in a for-loop.

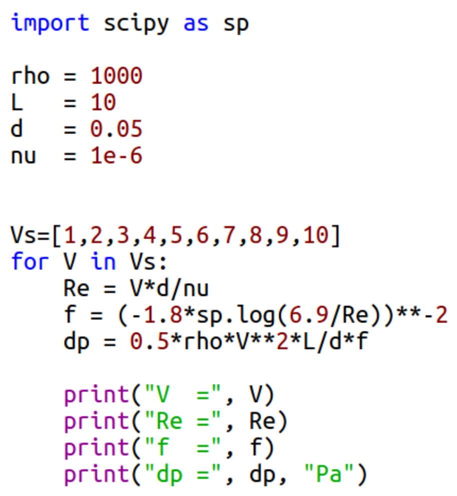
Let us start simple by familiarizing us with the list and loop first and ignore the toy problem for a now. Comment out the previous lines from your script by adding a # symbol in from of all the lines. You can also use the commenthot key “Ctrl-1” to comment faster. Commented lines will be ignored by the interpreter.

Lists are created with square brackets []. Type “Vs=[1,2,3,4,5,6,7,8,9,10]” to create the list and print it with “print(Vs)” command. Now to loop through all the values in the list use the for-loop as

```
Vs=[1,2,3,4,5,6,7,8,9,10]
for V in Vs:
    print(V)
```

Note that the empty spaces (indent) are part of the Python syntax. Running the code should print all the individual V values in Vs, i.e. 1,2,3,4...10.

Now uncomment the input values from before (rho, L,...) and copy-paste the relevant code inside the for-loop, as shown in Fig. 3. Running this code now prints the solution for all the velocities.



```
import scipy as sp

rho = 1000
L   = 10
d   = 0.05
nu  = 1e-6


Vs=[1,2,3,4,5,6,7,8,9,10]
for V in Vs:
    Re = V*d/nu
    f = (-1.8*sp.log(6.9/Re))**-2
    dp = 0.5*rho*V**2*L/d*f

    print("V  =", V)
    print("Re =", Re)
    print("f  =", f)
    print("dp =", dp, "Pa")
```

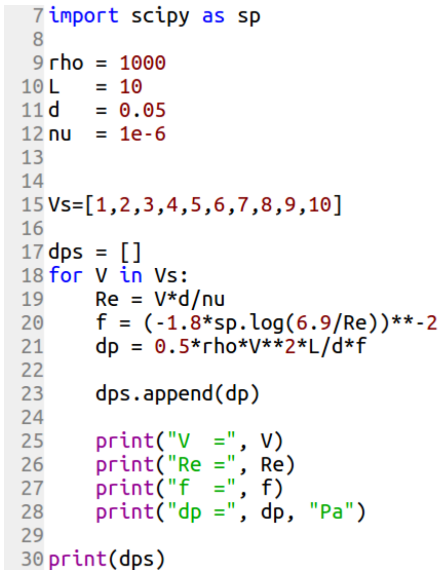
Figure 3: For-loop

Now you have the power to solve a large number of simple problems. Play around.

5 Plotting the results

The print outs in Section 4 are convenient until a certain number of cases but quickly become cumbersome. We’ll now learn how to plot the results in a graphical manner instead.

First we need to collect the resulting pressure drops (dp) in a new list. First initialize an empty list before the for-loop as “dps=[]” and the append the resulting pressure drop to the new list as “dps.append(dp)”. You can now access the pressure drops afterwards. See Fig. 4.

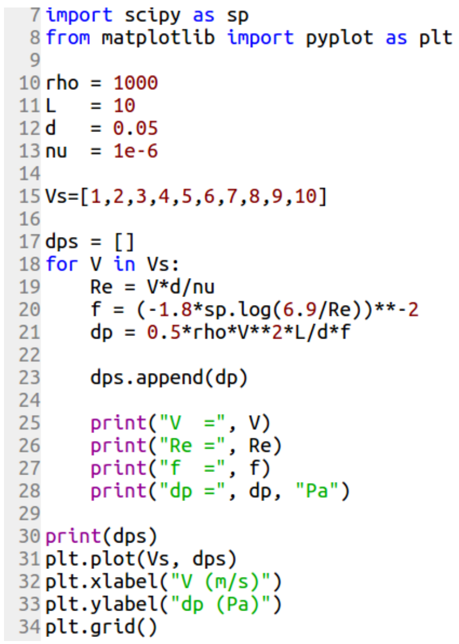


```python
import scipy as sp

rho = 1000
L   = 10
d   = 0.05
nu  = 1e-6


Vs=[1,2,3,4,5,6,7,8,9,10]

dps = []
for V in Vs:
    Re = V*d/nu
    f = (-1.8*sp.log(6.9/Re))**-2
    dp = 0.5*rho*V**2*L/d*f

    dps.append(dp)

    print("V  =", V)
    print("Re =", Re)
    print("f  =", f)
    print("dp =", dp, "Pa")

print(dps)
```

Figure 4: List of pressure drops

The standard tool for scientific plotting in Python is “matplotlib”. Import the library and give it a shorter name by adding “from matplotlib import pyplot as plt” in the beginning of the script. Matplotlib is a very large library and we only need the “pyplot” part so we only import that.

We can now plot the results as “plt.plot(Vs, dps)”. The resulting plot should be visible in the IPython after running the scrip (F5). We can make the plot prettier by adding labels and grid. See the complete code in Fig. 5.



```python
import scipy as sp
from matplotlib import pyplot as plt

rho = 1000
L   = 10
d   = 0.05
nu  = 1e-6

Vs=[1,2,3,4,5,6,7,8,9,10]

dps = []
for V in Vs:
    Re = V*d/nu
    f = (-1.8*sp.log(6.9/Re))**-2
    dp = 0.5*rho*V**2*L/d*f

    dps.append(dp)

    print("V  =", V)
    print("Re =", Re)
    print("f  =", f)
    print("dp =", dp, "Pa")

print(dps)
plt.plot(Vs, dps)
plt.xlabel("V (m/s)")
plt.ylabel("dp (Pa)")
plt.grid()
```

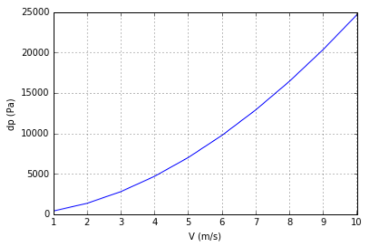


Figure 5: Plotting

6 Scipy Arrays

The lists and loops used in the previous Section work just fine, but it is often preferable to use scipy arrays instead. This usually results in faster and simpler code. So let us convert our script to scipy array format. You may want to make a back up of your script at this point.

Scipy arrays are basically lists on steroids. They can do pretty much everything lists can do, and a lot more.

In your IPython interpreted define a couple of scipy arrays of the same size, say: “a=sp.array([1,2])” and “b=sp.array([3,4])”. The syntax for the scipy array is a little longer that for the list but quickly pays out.

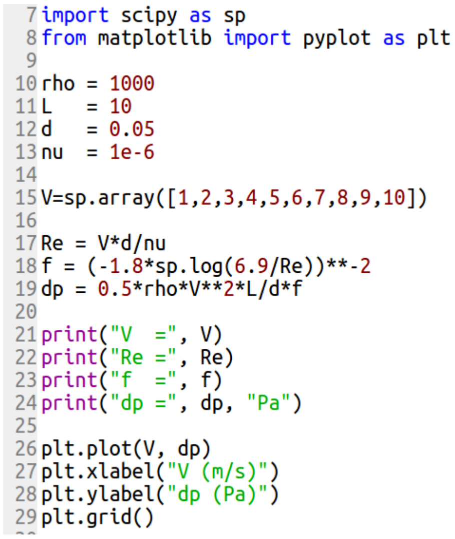
Now start doing simple math with the newly created a and b. Such as a*b, a**2, sp.log(b),... You'll notice that all the operations are done element vice. In other words, a*b results in two different operations 1*3 and 2*4.

You can also access elements inside your arrays using indexing. For example, “a[0]” is the first element in “a” and “b[1]” is the second element in “b”. Therefore “a[0]*b[1]” would give “1*4” with the example values.

Now let's use scipy arrays in our toy problem. First replace “Vs=[1,2,3,4,5,6,7,8,9,10]” with a equivalent scipy array “Vs=sp.array([1,2,3,4,5,6,7,8,9,10])“. You can re-run the code at this point. Nothing should change.

Now, we can remove the loop from our script by using the scipy array instead of single value in our calculations. Also remove the other unnecessary variables.

I also removed the “s” letters from the variable names to make the code prettier. See complete code in Fig. 6.



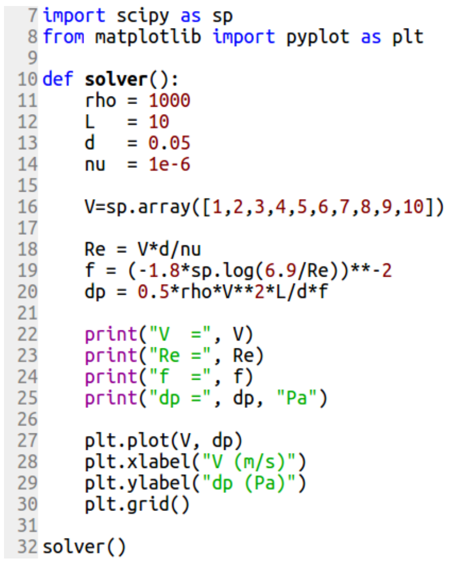
```
import scipy as sp
from matplotlib import pyplot as plt

rho = 1000
L   = 10
d   = 0.05
nu  = 1e-6

V=sp.array([1,2,3,4,5,6,7,8,9,10])

Re = V*d/nu
f = (-1.8*sp.log(6.9/Re))**-2
dp = 0.5*rho*V**2*L/d*f

print("V  =", V)
print("Re =", Re)
print("f  =", f)
print("dp =", dp, "Pa")

plt.plot(V, dp)
plt.xlabel("V (m/s)")
plt.ylabel("dp (Pa)")
plt.grid()
```

Figure 6: Scipy array

7 Defining a function

In order to make our code more structured and reusable it is often useful to define a lot of functions. In this toy problem, the function definition is a little artificial but let us do it for the practice.

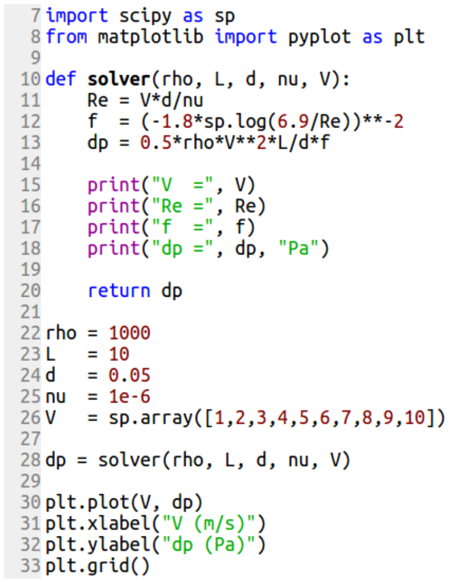
In Python, functions are defined with the keyword “def”. Let’s first turn our whole script into a function and call it with no added functionality. See Fig. 7. This should run exactly as before.



```python
import scipy as sp
from matplotlib import pyplot as plt

def solver():
    rho = 1000
    L   = 10
    d   = 0.05
    nu  = 1e-6

    V=sp.array([1,2,3,4,5,6,7,8,9,10])

    Re = V*d/nu
    f = (-1.8*sp.log(6.9/Re))**-2
    dp = 0.5*rho*V**2*L/d*f

    print("V  =", V)
    print("Re =", Re)
    print("f  =", f)
    print("dp =", dp, "Pa")

    plt.plot(V, dp)
    plt.xlabel("V (m/s)")
    plt.ylabel("dp (Pa)")
    plt.grid()

solver()
```

Figure 7: Function definition

Now let us turn the physical input parameters of our problem as parameters for the function. We also return the pressure drop from the function. Returning a value from a function makes it accessible outside of the function. See Fig. 8.



```python
import scipy as sp
from matplotlib import pyplot as plt

def solver(rho, L, d, nu, V):
    Re = V*d/nu
    f  = (-1.8*sp.log(6.9/Re))**-2
    dp = 0.5*rho*V**2*L/d*f

    print("V  =", V)
    print("Re =", Re)
    print("f  =", f)
    print("dp =", dp, "Pa")

    return dp

rho = 1000
L   = 10
d   = 0.05
nu  = 1e-6
V   = sp.array([1,2,3,4,5,6,7,8,9,10])

dp = solver(rho, L, d, nu, V)

plt.plot(V, dp)
plt.xlabel("V (m/s)")
plt.ylabel("dp (Pa)")
plt.grid()
```

Figure 8: Function with parameters

Now we have a solver function that takes all the physical parameters of the pressure drop as input parameters and return the pressure drop. In a more complicated problem, this would likely be a small part of the complete solution and repeated for many different cases.

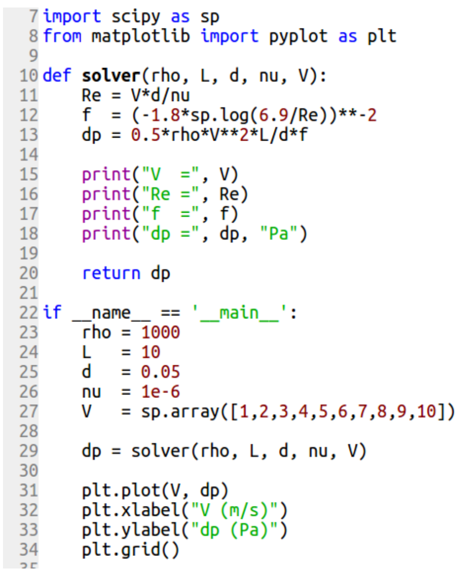
Now, as a final touch, let's turn our script in to a ready-to-deploy state by adding a "if __name__ == '__main__':" test in the end section.

This line test if the current file is the "main" file running the larger program. The technical term for this is "unit testing" and allows us to test small pieces of code individually without the test code affecting the larger program.

If the unit testing part doesn't make sense to you, never mind. Just believe it's a good programming practice and the reasons will become obvious with experience. Or search Internet for "unit test".

In the following Python exercises you will often be given a template file to work with.

The complete code below in Fig. 9.



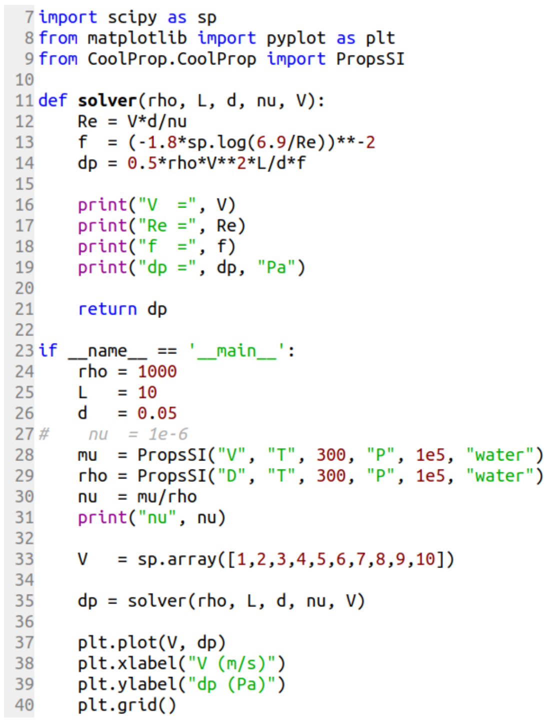
```
import scipy as sp
from matplotlib import pyplot as plt

def solver(rho, L, d, nu, V):
    Re = V*d/nu
    f  = (-1.8*sp.log(6.9/Re))**-2
    dp = 0.5*rho*V**2*L/d*f

    print("V  =", V)
    print("Re =", Re)
    print("f  =", f)
    print("dp =", dp, "Pa")

    return dp

if __name__ == '__main__':
    rho = 1000
    L   = 10
    d   = 0.05
    nu  = 1e-6
    V   = sp.array([1,2,3,4,5,6,7,8,9,10])

    dp = solver(rho, L, d, nu, V)

    plt.plot(V, dp)
    plt.xlabel("V (m/s)")
    plt.ylabel("dp (Pa)")
    plt.grid()
```

Figure 9: Unit test

8 Fluid properties

A very common feature of a fluid related engineering problem is the fluid properties. It quickly becomes cumbersome to look for to fluid properties from tables and type the values in the solver. Luckily, there are libraries for this too.

On this course, we use CoolProp library. The syntax is a little old fashioned but easy to use. And after this, you can always just copy-paste it. For the complete code see Fig. 10.



```
 7 import scipy as sp
 8 from matplotlib import pyplot as plt
 9 from CoolProp.CoolProp import PropsSI
10 
11 def solver(rho, L, d, nu, V):
12     Re = V*d/nu
13     f  = (-1.8*sp.log(6.9/Re))**-2
14     dp = 0.5*rho*V**2*L/d*f
15 
16     print("V  =", V)
17     print("Re =", Re)
18     print("f  =", f)
19     print("dp =", dp, "Pa")
20 
21     return dp
22 
23 if __name__ == '__main__':
24     rho = 1000
25     L   = 10
26     d   = 0.05
27 #    nu  = 1e-6
28     mu  = PropsSI("V", "T", 300, "P", 1e5, "water")
29     rho = PropsSI("D", "T", 300, "P", 1e5, "water")
30     nu  = mu/rho
31     print("nu", nu)
32 
33     V   = sp.array([1,2,3,4,5,6,7,8,9,10])
34 
35     dp = solver(rho, L, d, nu, V)
36 
37     plt.plot(V, dp)
38     plt.xlabel("V (m/s)")
39     plt.ylabel("dp (Pa)")
40     plt.grid()
```

Figure 10: CoolProp

First we need to import the CoolProp library. We only use the *PropsSI* part of the library so we only import that. See line 9 in Fig. 10.

The kinematic viscosity is not directly available from PropsSI, so we get dynamic viscosity μ and density ρ first. Kinematic viscosity is then calculated as $\nu = \frac{\mu}{\rho}$. See lines 28-30 in Fig. 10.

The first parameter in the PropsSI call is the property we want: "V" for viscosity and "D" for density. The next four are the temperature and pressure where we want the value. The units are Kelvins and Pascals. The last one is the fluid.

The parameters are quite flexible. If you, for example, would want to get saturated water viscosity at $T = 300K$, you could write

```
mu  = PropsSI("V", "T", 300, "Q", 1, "water")
```

where "Q" is quality.

Like programming in general, using a fluid properties library is radically faster and less error prone than manual labor.

9 Importing

We have already imported plenty of stuff but let us now look a little into what it means.

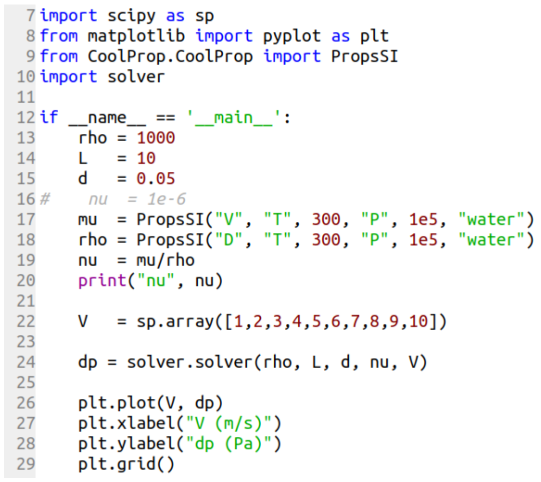
As previously stated, libraries are pre-existing collections of code. They can be written by somebody else or by yourself.

Where do libraries come from and how does Python find them? The simplest explanation is that you wrote them and placed them in the same folder as the main script. If the library only consist of one file, it is usually called a module. There is no fundamental difference.

Let us now turn our code into a module format. You may leave your existing code as it is. If you want your code to look the same as the example in Fig. 11, name your existing code "solver.py".

Now, create a new python file in the same folder as you "solver.py" file. You may name it "main.py".

Copy paste everything expect the solver implementation from your "solver.py" into "main.py".



```
 7 import scipy as sp
 8 from matplotlib import pyplot as plt
 9 from CoolProp.CoolProp import PropsSI
10 import solver
11 
12 if __name__ == '__main__':
13     rho = 1000
14     L   = 10
15     d   = 0.05
16 #    nu  = 1e-6
17     mu  = PropsSI("V", "T", 300, "P", 1e5, "water")
18     rho = PropsSI("D", "T", 300, "P", 1e5, "water")
19     nu  = mu/rho
20     print("nu", nu)
21 
22     V   = sp.array([1,2,3,4,5,6,7,8,9,10])
23 
24     dp = solver.solver(rho, L, d, nu, V)
25 
26     plt.plot(V, dp)
27     plt.xlabel("V (m/s)")
28     plt.ylabel("dp (Pa)")
29     plt.grid()
```

Figure 11: Module

You may now access your solver implementation from "solver.py" in "main.py" by importing it, see line 10 in Fig. 11. You also need to add the solver. prefix to the solver call on line 24 in Fig. 11. This is highly useful in more complex programs.

It is likely that you will not need to make your own modules in this course. You may, however, be given modules written by the staff.

What about scipy, matplotlib and the other libraries we've been importing?

If Python doesn't find any modules or libraries in the same folder, it will look into the default install locations of libraries. The exact procedure is not relevant to us. The TUT computers used on this course have the needed libraries pre-installed.

If you want to add something to your Python, the easiest way to do it is with PIP. You simply write "pip install package_name". PIP comes with Python and works on all platforms.