KEB-45250 Numerical Techniques for Process Modelling
Exercise 0 - Python Tutorial
11.01.2020

Antti Mikkonen

1 Introduction

On this course we use two computational tools: Python and ANSYS. We will start with Python and start using ANSYS once we reach 3D fluid flow in the 4th period.

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 science and engineering tools. No prior experience with Python is necessary, but programming is not explicitly taught 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 TUNI basic programming course.

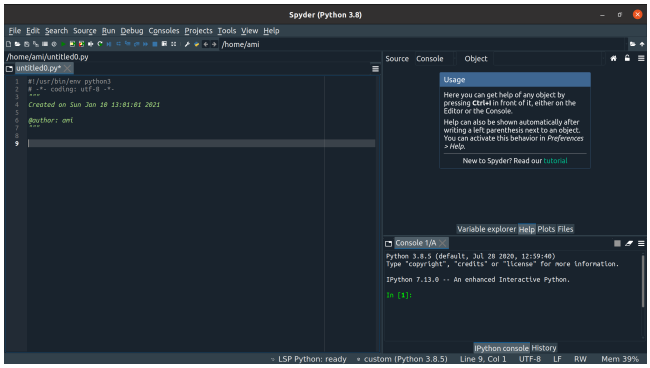

Figure 1: Spyder

2 First steps

There are many ways to access the power of Python but we use Spyder 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 we would recommend using your favorite package manager, for example apt.

Start up Spyder and you'll see something like that in Fig. 1.

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

```
1 3+2
2 4*2
3 3**3
```

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

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

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

This is convenient for some extremely simple cases but quickly becomes tiresome. So let's **start scripting**. Now type your simple math in the editor window on the left and press **"F5"**. The first time you run the code, a window similar to that in Fig. 2 will pop up. It is recommended to use the configuration shown in Fig. 2. "Remove all variables before execution" is the key setting. The script you just created will run in IPython just like if you had written it there. Now you have made your first Python script.

Figure 2: Recommended run configuration

3 First toy problem

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

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

$$\frac{1}{\sqrt{f}} = -1.8 \log_{10}\left(\frac{6.9}{\mathrm{Re}}\right) \tag{2}$$

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

where Δp is pressure drop, ρ is density, V is velocity, L is pipe length, d is pipe diameter, f is Darcy friction factor, Re is Reynolds number, and ν is kinematic viscosity. This is a straight forward problem to solve even with a hand held calculator but, arguably, programming is easier.

In this example we mostly make due with standard Python but note the log10-operator in Eq. 2. Log-operators is not available in the standard Python so we need to import a library. For such as simple operation, many suitable libraries exists, but we go straight for the standard "scientific Python library" numpy. Note that we need a 10-based logarithm.

Libraries are collections of pre-existing tools for some specific purpose. Python is such a widely used scientific tool, that you can find a free tool for most of your problems with a simple Google search.

To access the power of libraries in Python, use the command "*import library_name*". A library can be given a simpler name with the "*as*" keyword as shown in Fig. 3. Now we can access the log-function as "*np.log*".

Use the same input values as shown in Fig. 3 and solve the problem.

```
 9 import numpy as np
10
11 rho = 1000
12 L   = 10
13 d   = 0.05
14 nu  = 1e-6
15 V   = 10
16
17 Re = V*d/nu
18 f  = (-1.8*np.log10(6.9/Re))**-2
19 dp = 0.5*rho*V**2*L/d*f
20
21 print("Re =", Re)
22 print("f  =", f)
23 print("dp =", dp, "Pa")
```

Figure 3: 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 with next to zero probability of error.

4 Solving a large number of toy problems

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

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

Let's start simple by familiarizing us with 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 front of all the lines. You can also use the comment-hot key "*Ctrl-1*" to comment faster. Commented lines will be ignored by Python.

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

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

Note that the empty spaces in front of the code (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. 4. Running this code now prints the solution for all the velocities.

```
 7 import numpy as np
 8
 9 rho = 1000
10 L   = 10
11 d   = 0.05
12 nu  = 1e-6
13
14 Vs=[1,2,3,4,5,6,7,8,9,10]
15 for V in Vs:
16     Re = V*d/nu
17     f = (-1.8*np.log10(6.9/Re))**-2
18     dp = 0.5*rho*V**2*L/d*f
19
20     print("V  =", V)
21     print("Re =", Re)
22     print("f  =", f)
23     print("dp =", dp, "Pa")
24     print()
```

Figure 4: For-loop

Now you have the power to solve a large number of simple problems!

Try solving for a 10 times as many velocities. Or maybe 100 times. You can conveniently repeat the list 10 times by writing "Vs=[1,2,3,4,5,6,7,8,9,10]*10". If you end up with a process that takes too long to finish, you can terminate it by pressing the red square in the upper right corner of the interpreter window.

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) into a new list. Go back to the original 10 different values of velocity. First initialize an empty list as "dps=[]" and then append the resulting pressure drop to the new list as "dps.append(dp)". You can now access the pressure drops afterwards. See Fig. 5.

```
 7  import numpy as np
 8
 9  rho = 1000
10  L   = 10
11  d   = 0.05
12  nu  = 1e-6
13
14
15  Vs=[1,2,3,4,5,6,7,8,9,10]
16
17  dps = []
18  for V in Vs:
19      Re = V*d/nu
20      f = (-1.8*np.log10(6.9/Re))**-2
21      dp = 0.5*rho*V**2*L/d*f
22
23      dps.append(dp)
24
25      print("V  =", V)
26      print("Re =", Re)
27      print("f  =", f)
28      print("dp =", dp, "Pa")
29      print()
30
31  print(dps)
```

Figure 5: 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 Spyder "Plots" tab after running the scrip (F5), see Fig. 6. We can make the plot prettier by adding labels and grid.

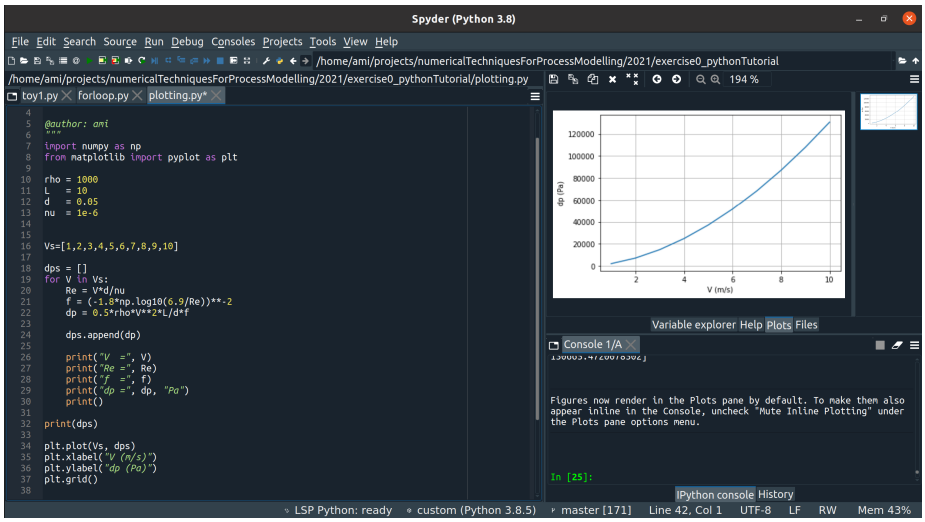

Figure 6: Plotting

6 Numpy Arrays

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

Numpy 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 numpy arrays of the same size, say: "a=np.array([1,2])" and "b=np.array([3,4])". The syntax for the numoy 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, np.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.

If you want to start indexing from the back, use negative numbers. For example, a[-1] is the last element in a and a[-2] is the second last.

Now let's use numpy arrays in our toy problem. First replace "Vs=[1,2,3,4,5,6,7,8,9,10]" with a equivalent numpy array "Vs=np.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 numpy 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. 7.

```
7  import numpy as np
8  from matplotlib import pyplot as plt
9
10 rho = 1000
11 L   = 10
12 d   = 0.05
13 nu  = 1e-6
14
15 V=np.array([1,2,3,4,5,6,7,8,9,10])
16
17 Re = V*d/nu
18 f = (-1.8*np.log10(6.9/Re))**-2
19 dp = 0.5*rho*V**2*L/d*f
20
21 print("V  =", V)
22 print("Re =", Re)
23 print("f  =", f)
24 print("dp =", dp, "Pa")
25
26 plt.plot(V, dp)
27 plt.xlabel("V (m/s)")
28 plt.ylabel("dp (Pa)")
29 plt.grid()
```

Figure 7: Numpy 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's 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. 8. This should run exactly as before.

```python
import numpy as np
from matplotlib import pyplot as plt

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

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

    Re = V*d/nu
    f = (-1.8*np.log10(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 8: Function definition

Now let's 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. 9.

```python
import numpy as np
from matplotlib import pyplot as plt

def solver(rho, L, d, nu, V):
    Re = V*d/nu
    f = (-1.8*np.log10(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=np.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: 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.

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 manually type them in. 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 numpy as np
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*np.log10(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   = np.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()
41
42
43     h = PropsSI("H", "T", 20+273.15, "Q", 0.4, "water")
```

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 $\nu = \frac{\mu}{\rho}$.

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, wanted to get steam enthalpy at $T = 20°C$, you could write

```
h = PropsSI("H", "T", 20+273.15, "Q", 0.4, "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 Additional features

In addition to the previous libraries, numpy, matplotlib, and coolprop, we often use an additional scientific library scipy - scientific python. Scipy extends the capabilities of numpy to include interpolation, optimizations, etc. typical tools for an engineer. Some of the course material may be out of date and use scipy in place of numpy in some instances, for example **sp**.linspace instead of **np**.linspace. This still works, but should be avoided as they will be removed in SciPy 2.0.0. An up-to-date scipy will give you a helpful warning if you do this. Just replace scipy with numpy in these instances.

The programming is kept very simple in this course. As a final touch, it is helpful to add a "if __name__ == '__main__':" test in the end section.

This line tests 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 accept it as a good programming practice and the reasons will become obvious with experience. Or search Internet for "unit test". Most of the example codes on this course use this convention.

The complete code is shown below in Fig. 11.

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

def solver(rho, L, d, nu, V):
    Re = V*d/nu
    f = (-1.8*np.log10(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")
    print()

    return dp

if __name__ == "__main__":
    rho = 1000
    L   = 10
    d   = 0.05
    nu  = 1e-6
    V=np.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 11: Unit test