Python for Scientific Computing

A gentle introduction

Dr. Vishwanath H. Dalvi (Institute of Chemical Technology, Mumbai)

Contents

1. Introduction

This is a gentle introduction to a powerful tool for doing science. The main advantage of the Python programming platform is its almost intuitive syntax (indeed some lines of code are almost syntactically correct English!) and availability of a large number of libraries. The best scientific groups around the word provide a Python interface for the code they write. But more importantly, Python allows any scientist to unlock the power of the computer is a painless manner. With Python, *anybody* can be a programmer, *anybody* can make animated, interactive graphics and *anybody* can write code to interface with Excel.

2. The Programming Platform

A Python program can be written in any text editor (e.g. Notepad) as long as the file is saved with the *.py* extension. However, it is preferable to write the program in an *Integrated Development Environment (IDE)* with is a text editor designed *specifically* for writing programs: so that it will save you a lot of trouble. The IDE of choice in our case is *Spyder*.

2.1 Installation

To run the program, you need to have the Python interpreter installed on your computer.

To get the Python interpreter, the IDE and a host of other *libraries* (pre-written programs for your convenience), we can download the installer from the internet (http://python-xy.github.io/). The installer is called *"Python(x,y)-2.7.3.1.exe"*. It is already there on your machine. The icon for the version 2.7.3.1 looks like Figure 1. It is a 500 MB file. You can copy it to your pendrive and take it home to install on your own computers if you like. It runs in Windows.

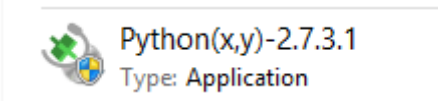


Figure 1: Icon of Python(x,y) version 2.7.3.1

Once you have this on your machine, *double-click* it to run. You will soon see the screen in Figure 2(a) which will invite you do execute a *Custom* install. *Do not do this*. From the drop down menu choose *Full* install (see Figure 2(b)). Then click on the *Next* button and wait for installation to complete. Should take about 5-10 minutes.

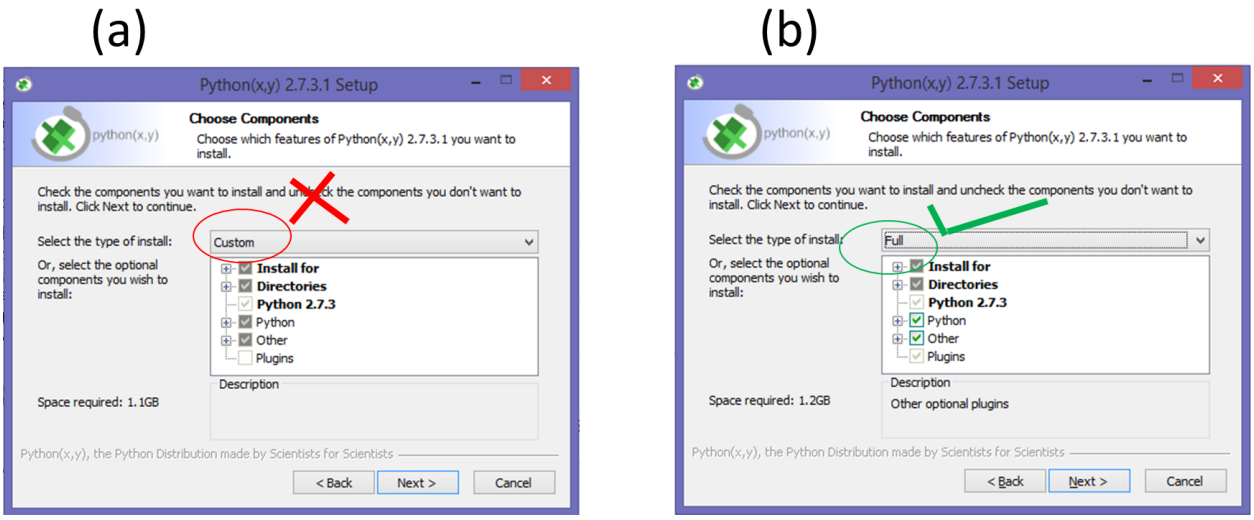


Figure 2: Wrong (a) and right (b) configurations for installing Python(x,y)

2.2 Spyder the IDE

Once Python(x,y) is installed, go to your start menu (press the Windows button) and start typing: "Spyder". You will see an icon that looks like this  . Click on it. Things will happen and then you will get a screen that looks like Figure 3 (without the annotation, of course).

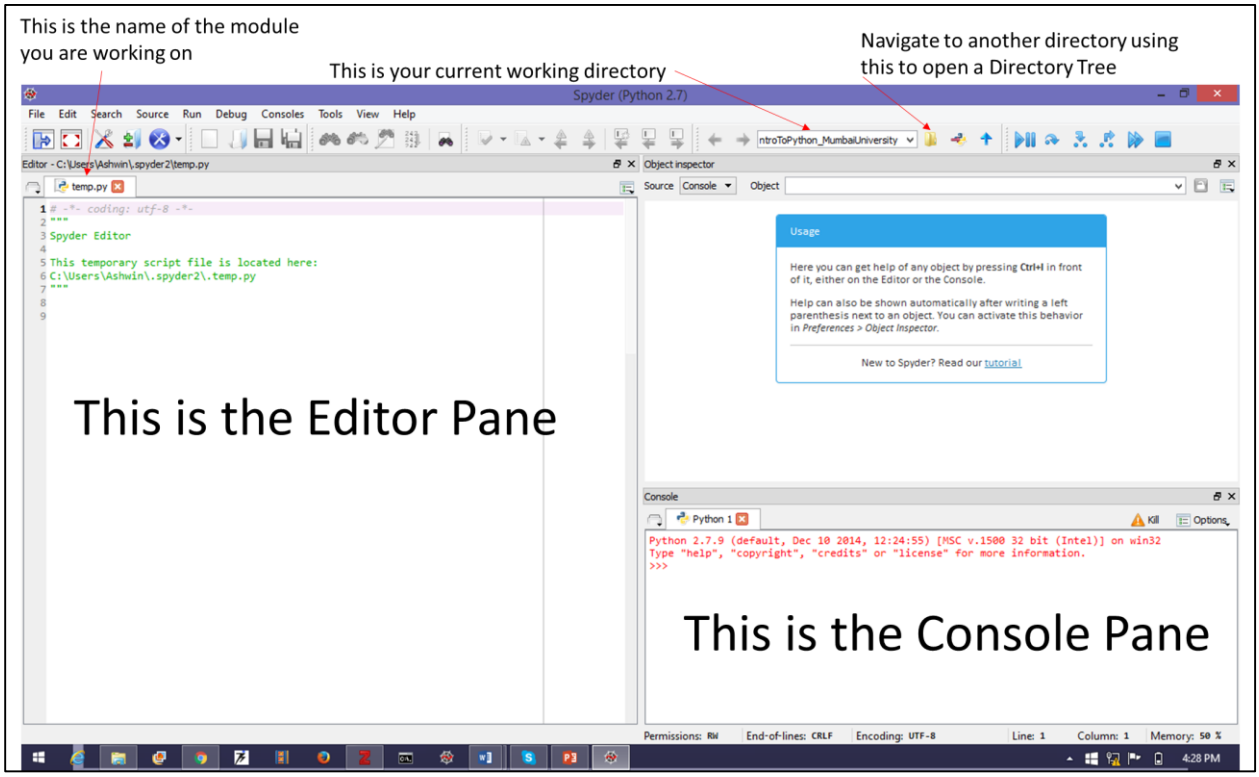


Figure 3: A first look at the Spyder IDE

In the Console Pane, you can start programming right away. *But* you can execute only one line at a time: so it quickly becomes a drag. To execute multiple lines automatically (as well as do more fancy coding), you need to use the Editor Pane. *But* it is not interactive. The trick for good fun programming is to use *both*. So let's begin …

2.3 Use of the Console Pane

In front of the prompt (">>>") in the Console Pane, type "a = 1" and press Enter/Return. Then type "b = 2" and press Enter. Then type the commands shown in Figure 4.

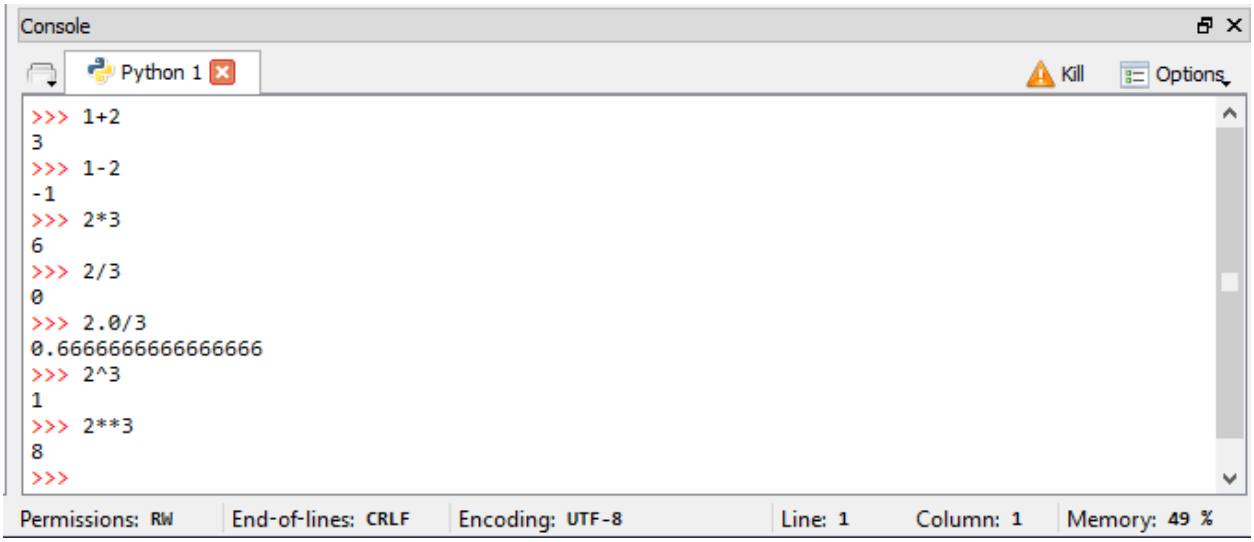


Figure 4: Simple programming in the Console Pane

Notice the difference between *integer* division (where 2/3=0) and *float* division (2.0/3.0 = 0.666). Also notice that raising to powers is done in the *C* manner (2**3 = 8) and *NOT* in the Excel Manner (*2^3=1!*). So can you use this Console Pane as a scientific calculator? Sure. But you have to *import* the appropriate *libraries*.

So let's do that. We would like to use *trigonometric functions, exponentials* and *logarithms*. All of these are held in the *Scientific Python* library which is called "scipy". Hence (see Figure 5) we first *import* the Scientific Python library using the command "import scipy". Now the variable "scipy" will be used to access all the mathematical functions we need.

For instance, if we wish to find the value of e^{-1}, we would us the *exp* function in the *scipy* module thus: *scipy.exp(-1.0)*.

Similarly if we wanted to find the *sine* of a number (say 30.0), we would use the *sin* function in the *scipy* module thus: *scipy.sin(30.0)*. Note that the *argument* to the function is assumed to be in *radians,* not degrees. If we wanted to find the *sine* of 30^{o}, we would have to first convert it to radians. We could simply multiply by $\pi/180$, but there is *already* an inbuilt function to save us the trouble. We will *nest* the function *scipy.radians* within *scipy.sin* thus: *scipy.sin(scipy.radians(30.0))*. Python supports nesting function.

There are a host of other mathematical functions that we can use. All the common trigonometric, the exponential, the hypergeometric, logarithmic etc. They are all there.

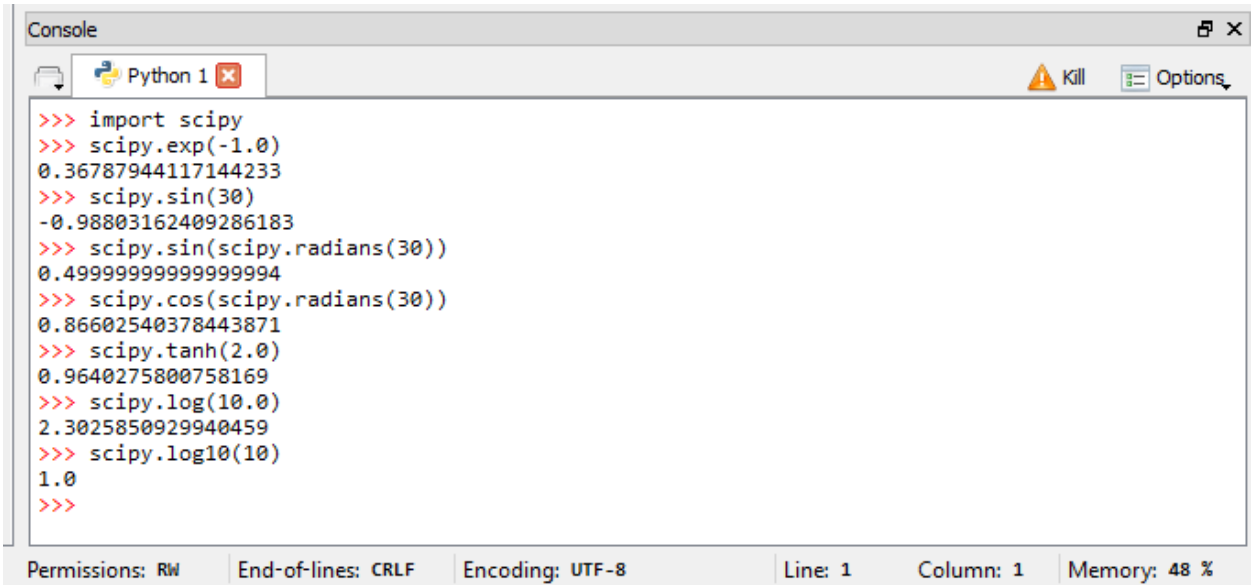


Figure 5: Use of the Scientific Python Library. We first import *the library using "import scipy" and then use the functions in it e.g. exponentiation is accessed using scipy.exp.*

Simple Problem Using Console Pane

Now, to be even more useful, we need to use variables. Say that you are considering a back-of-the-envelope calculation: How much land is necessary to capture enough solar radiation to run a 500 MW solar-thermal power plant? We know that the solar-insolation on the earth's surface is something like 800 W/m². Of this only 75% can be concentrated. Using parabolic trough collectors, only 50% of this concentrated radiation can be captured and used to raise steam in a boiler. This steam can be converted to electricity with an overall thermal efficiency of 30%. Also, the power plant operates for 24 hours per day but the sun shines for only 6 hours per day. How much thermal storage is required? If the cost of mirrors is Rs. 2 0,000/m² and the cost of land is Rs. 250,000/hectare, land use factor is 25% and the cost of a power plant is Rs. 6 crores/MW, what will be the cost of a solar-thermal installation? Note that 1 hectare is 10,000 m². What should be the price of electricity (in Rs/kWh) so that this returns a yearly return on investment of 15%. Assume 8000 operating hours per year. This is a *complex* problem requiring many steps. It can be solved straightforwardly, *but it is very easy to make mistakes*. You need to keep *track* of your numbers.

The easiest way of doing this is converting everything to algebraic symbols. Hence P is the capacity of the power plant, I is the solar insolation, η_{DNI} is the concentratable fraction, η_{cap} is the fraction that can be captured, η_{conv} is the fraction that can be turned to electricity. ϕ_{land} is the land use factor. D_{op} is the operating duration per day, D_{sun} is the sunny duration. Let C_{land} be cost of land, C_{mirror} be cost of mirrors and C_{plant} be cost of the power plant. Let RoI be the return on investment and C_{elec} be the cost of electricity. Let Y_{op} be the operating hours per year.

Hence, the electricity generated per unit mirror area is given by: $e_{sp} = I\eta_{DNI}\eta_{cap}\eta_{conv}$.

Quantity of electricity to be produced per day is: $E_{daily} = P \times D_{op}$.

Solar radiation corresponding to this has to be captured in D_{sun} time, hence the solar-field should be designed for a higher power production given by: $P_{sun} = \frac{E_{daily}}{D_{sun}}$.

Hence mirror area required is: $A_{mirror} = \frac{P_{sun}}{e_{sp}}$.

Hence land area required is: $A_{land} = \frac{A_{mirror}}{\phi_{land}}$

Hence cost of the project is: $C_T = A_{mirror} \times C_{mirror} + A_{land} \times C_{land} + P \times C_{plant}$.

The quantity of electricity produced per year is: $E_{yearly} = P \times Y_{op}$

The revenue generated from this electricity is: $R = E_{yearly} \times C_{elec}$.

Hence the return on investment is: $RoI = \frac{R}{C_T} \times 100$.

Hence $C_{elec} = \frac{RoI}{100} \times \frac{C_T}{E_{yearly}}$.

The storage required is equivalent to they electricity generated over the *non-sunny* portions of the day. Hence storage capacity required is: $E_{storage} = P \times (D_{op} - D_{sun})$.

Now lets do everything in Python. See Figure 6. We have converted all units to SI: Hence *J, W, s* etc. Notice that we can separate statements using semicolons (';') and that Python ignores anything on a line that follows a *hash* ('#').

You will also notice that Python has raised an error. That was because we forgot to put a '#' before the 'Rs/W' and, of course, 'Rs/W' is a meaningless term (we have defined neither 'Rs' nor 'W').

The result of the calculations is that $C_{elec} = Rs.\,11.9/kWh$ for the plant to get 10% return on investment and the storage capacity necessary is 9 million kWh.

(Incidentally, if we add the cost of electricity storage (Rs. 24,000/kWh) the cost of storage *alone* would be an unthinkable *Rs 22,000 crores!* The cost of the remaining plant is Rs 48,000 crores i.e. electricity storage would increase plant costs by nearly 50%! This is why people prefer *thermal* storage to electricity storage.)

Now, suppose we want to answer *another* problem: what is the cost of electricity if solar insolation were 1000 W/m^2? We would have to do the calculations all over again: it is very hard to separate the parts that would change from the parts that would not.

That is a bore! Even worse, that is a recipe for making mistakes. It would be nice if we could go *up* in the Console Pane and edit just one thing. Well, we can do that, but not in the Console Pane. For that we have to write the code in the *Editor Pane*. So let's do that …

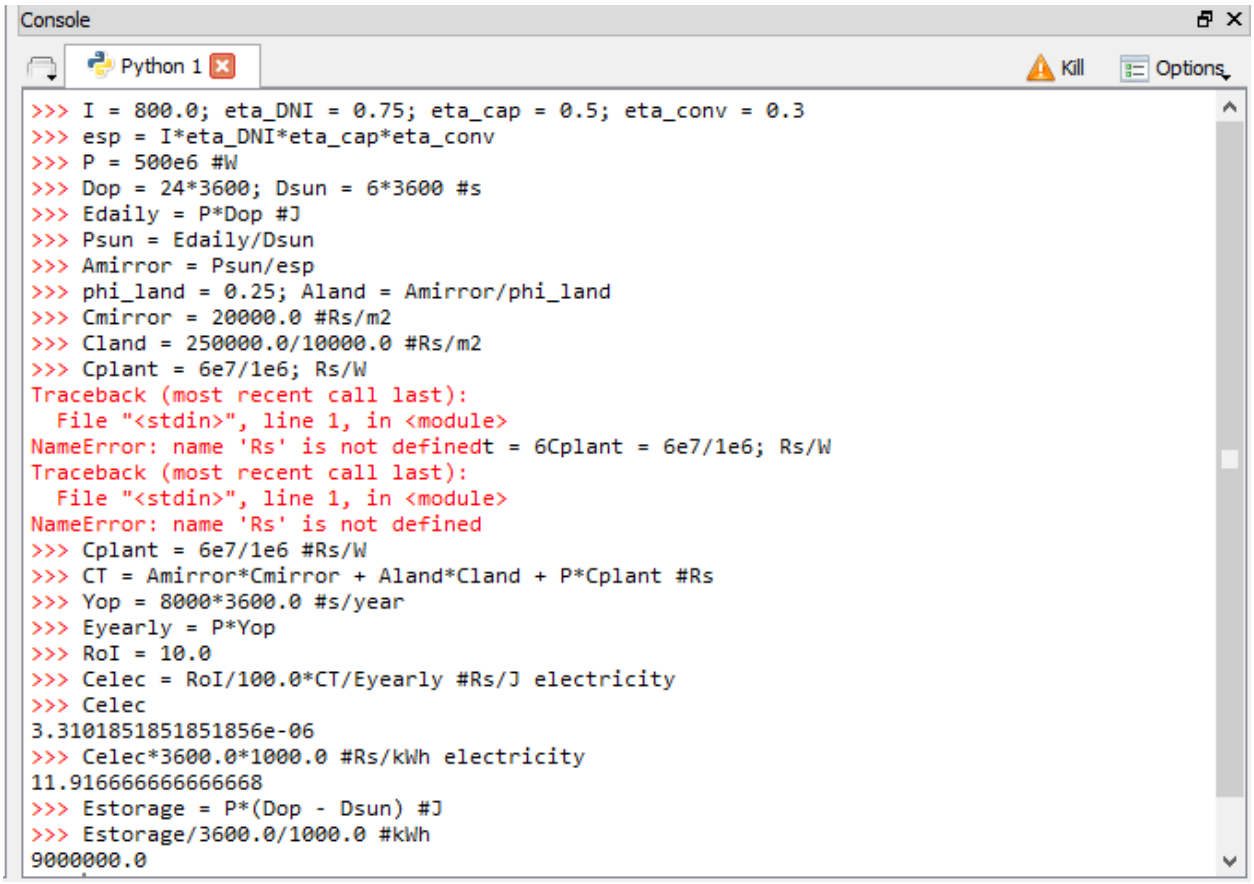


Figure 6: Back of the Envelope Calculations in Python

Interestingly: You can access the python console (something like the console pane) very quickly *without* using Spyder. Just open the command-prompt (press the 'Windows' button, type 'cmd' and press Enter) and type 'python' at the prompt and press Enter. There is your python Console ☺ (see Figure 7).

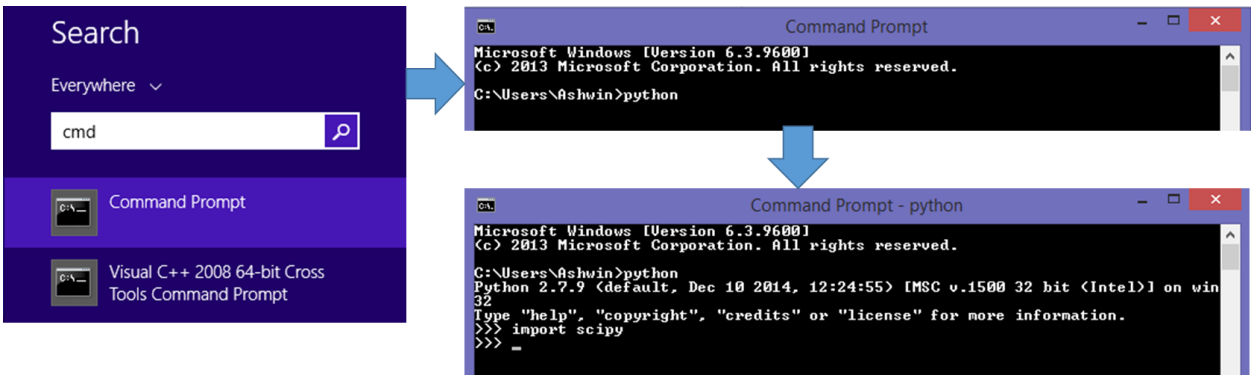


Figure 7: Quick access to the Python Console. Press 'Windows Button', type 'cmd', press Enter, type 'python', press Enter. Compare this is just one more step than accessing the 'Calculator' option in Windows, but it gives you far more calculating power ☺. Try to use it on a regular basis: why not?

2.4 Using the Editor Pane

Instead of executing code one-line-at-a-time like we did in the Console Pane, we can submit a bunch of code to the python interpreter via the Editor Pane. To use the Editor Pane effectively, we much first navigate to our *working folder*. Click the yellow folder icon (see Figure 3) which will open a file-explorer. Navigate to our working folder and click on the "Select Folder" button. Then click the multi-coloured icon *just to the right* of the yellow-folder icon.

Now look at the Editor Pane. You will see that there is a tab open named "temp.py". Close it (click on the 'x'). Now press 'Ctrl+N' to open a new file. A new tab opens in the Editor Pane. It is called "untitled0.py*'. The asterix ('*') indicates that it is unsaved. Press 'Ctrl-S' and a file-explorer window opens. Change the filename to 'solarthermal' in the File_name: field. You do not need to add the '.py' extension as long as the "Save as type:" field reads "Python files (*.py, *.pyw, *.ipy)". Now click on the 'Save' button. You will see that the name of the tab has changed to 'solarthermal.py' and that the '*' is gone.

(**Note:** You will also see some gray and green letters enclosed in triple quotes. Ignore them for now. If you like, you can delete them. They are not important for executing the code.)

Figure 8 shows the same calculation of section 2.3 except in the Editor Pane.

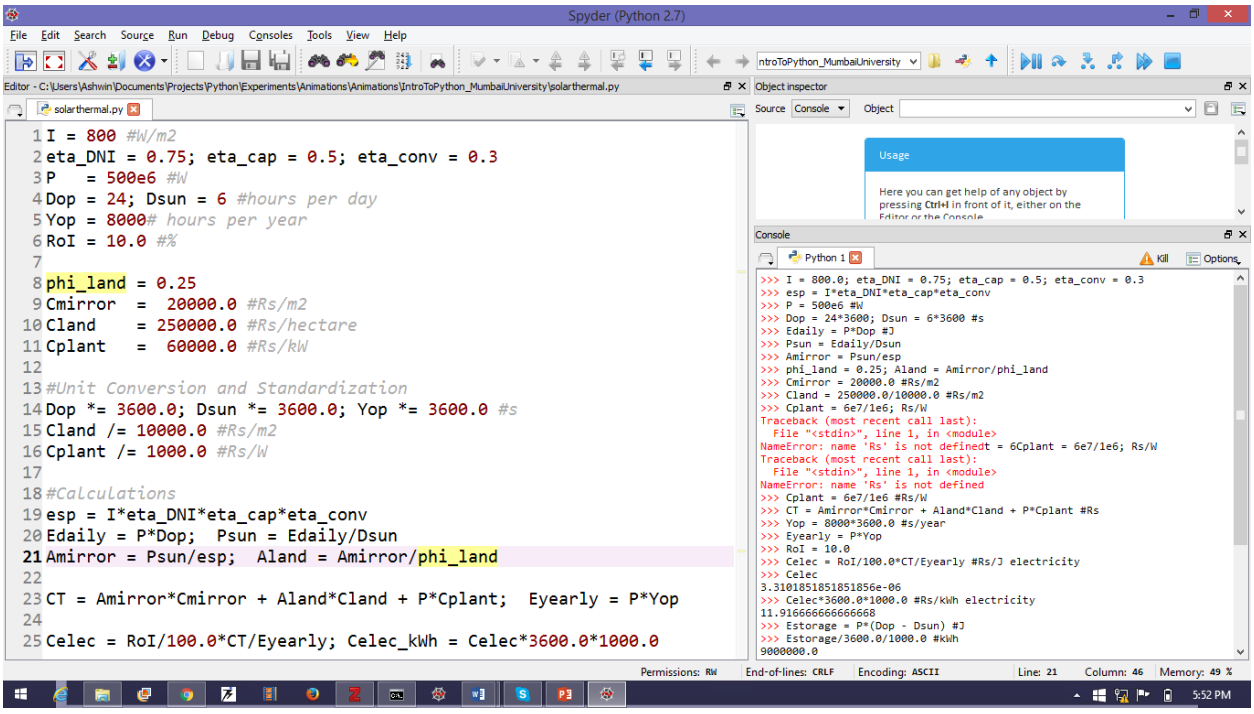


Figure 8: The same calculation of section 2.3 except now in the Editor Pane.

You can see that it allows much better organization of the code. To run the code, press the function key 'F5' on your keyboard. The first time you do this, you will see a dialog that lets you specify how you want the code to run (see Figure 9). Make sure you choose to "Execute in a new dedicated Python console". This will open up a new tab in the console pane. You should also check the option to "Interact with the Python console after execution" which makes it so that the new tab that opens has all the code in the Editor pane *pre-executed* (see Figure 10).

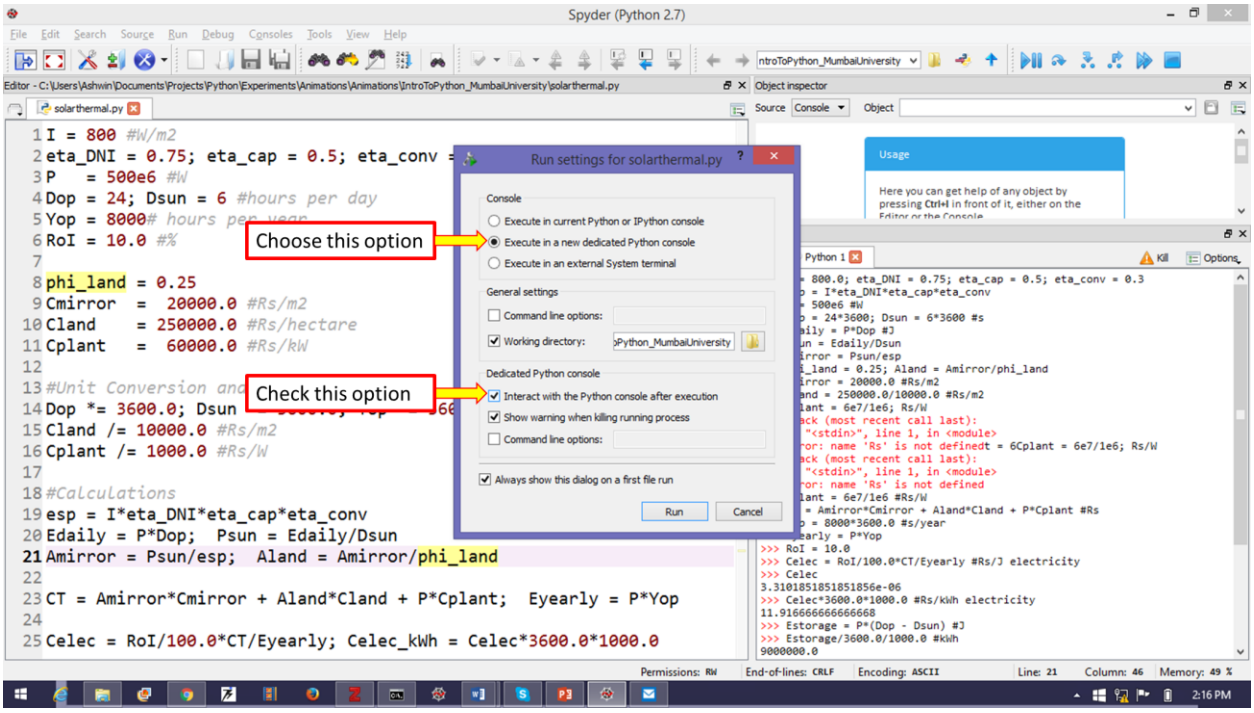


Figure 9: Dialog that appears first time you run a program from the Editor Pane. Make sure to choose to "Execute in a new dedicated Python console" and to "Interact with the Python console after execution".

Now click the "Run" button. You should see that a new tab has opened in the Console Pane (see Figure 10) which has the same name as the module we executed ('solarthermal.py'). You can see that this console has the line in the Editor Pane *pre-executed*.

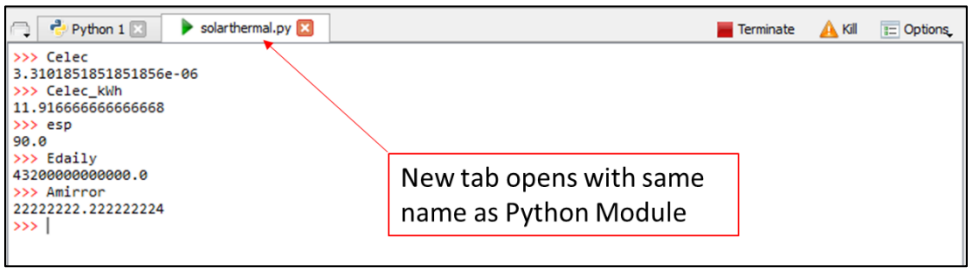


Figure 10: A new tab opens in the Console Pane. The name of the tab is the same as that of the module which was run. In this tab, all the lines in the Editor Pane are pre-executed. Hence you can access the variables in the Editor pane e.g. simply type Celec_kWh and press Enter to get the value of that variable.

Now, if we wanted to find out the cost of electricity for an *RoI* of 20%, we could just change the line 6 in the Editor Pane (see Figure 8) to "RoI = 20.0" and press 'F5' to run again. A dialog box will open asking whether you want to kill and existing process: say 'yes'. The 'solarthermal.py' tab in the console pane will refresh. Type 'Celec_kWh' and press Enter to get the new cost of electricity. It will be Rs. 23.8.

2.5 Writing Functions

This method is fine if you want only a few numbers. But when we do a sensitivity analysis, we really don't want keep changing line 6 and pressing 'F5' and noting down the answer. We want to repeatedly query a running program. For this we need to write functions. Writing functions in Python is really easy.

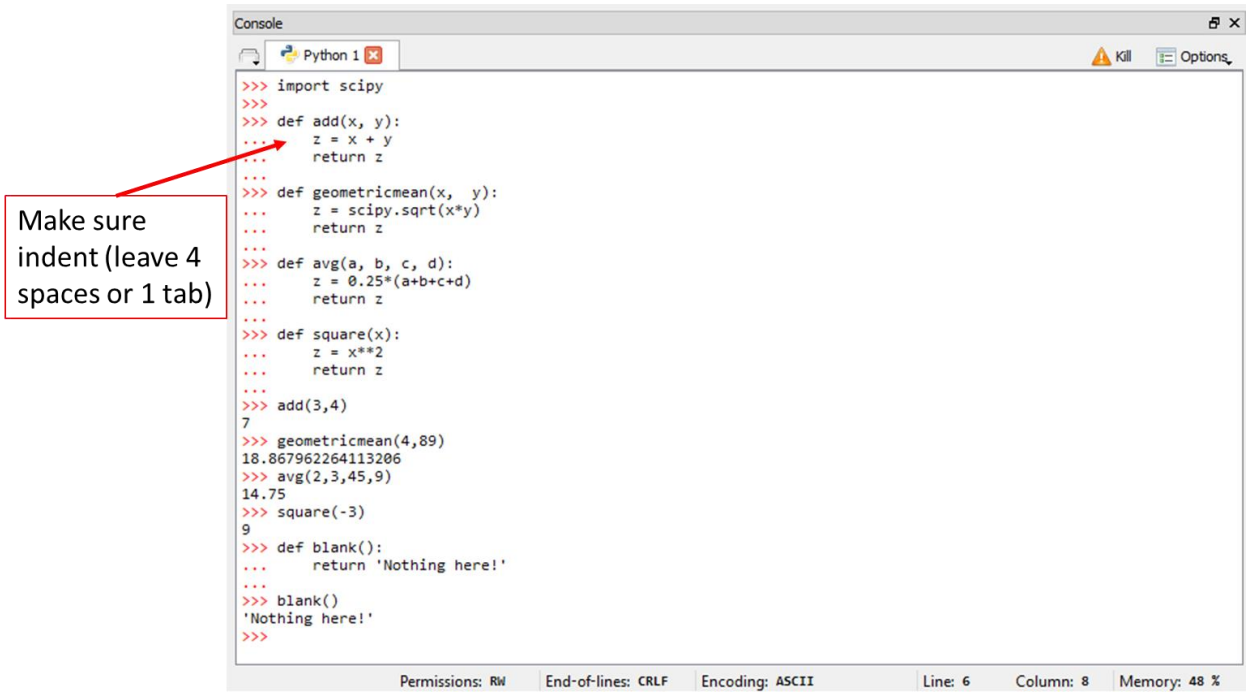


Figure 11: Writing simple functions in Python. Note that the block of code describing the function is indented *with respect to the def i.e. it is shifted by 4 spaces or 1 tab. Python does not use 'end' statements of curly brackets '{}' to define a function block. Instead Python considers all line indented with respect to the function definition to be part of the function. The first de-indent closes the function block.*

Figure 11 shows five simple functions written in the Console Pane. Each function has a "def" in front of its name. The name is *immediately* (no spaces) by an open parenthesis '(' which *may or may not be* followed by a list of *arguments* separated by commas and terminated by a closed parenthesis ')' and a colon ':'. This tells the Python interpreter that a function of the given name is being defined and it also tells how many arguments that function takes. What the interpreter is supposed to do to execute that function is written in the *indented block* on subsequent lines. Python uses *indentation* instead of 'end' statement or curly brackets ('{}') to delineate a function block. An indentation means that the line is 4 spaces to the left of the 'def' line. The function block often *returns* one or more values.

We can now modify the 'solarthermal.py' code to include a *function* to calculate *Celec_kWh*. We will save it as a new file called 'solarthermal_function.py'. The code is shown in Figure 12. What we have done is put everything from unit conversion downwards into a function block. The function name is *Celec* and it takes 13 arguments.

(**Note:** Notice that there is essentially no limit to the *number* of arguments that can be passed to a function. Also, Python allows you to continue a list of arguments over several lines.)

Run it in the same manner as we did before ('F5' then choose the appropriate options as shown in Figure 9 and then 'Run'). Again, a new tab called 'solarthermal_function.py' appears in the Console Pane (see Figure 13).

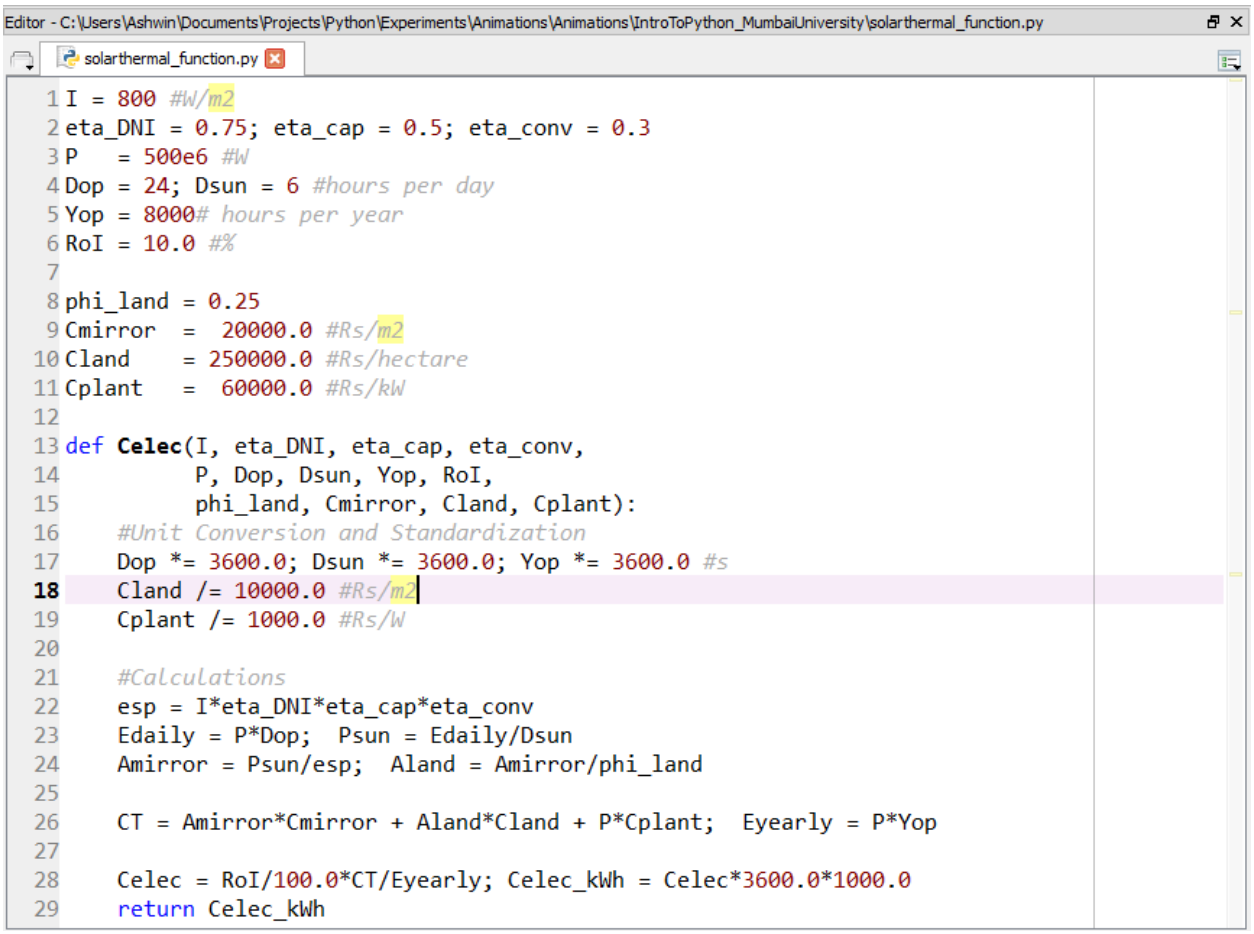


Figure 12: The code of Figure 8 written with a function incorporated.

We can now call the function *Celec* as shown in Figure 13. We have called it repeatedly with different values of RoI. This is a *much* quicker way than modifying the source code and running each time: especially when the code can be complex and involve pre-processing.

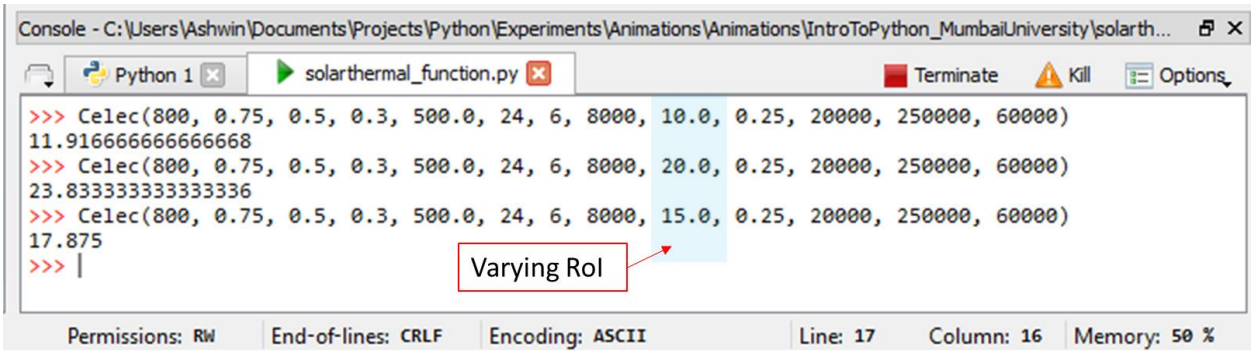


Figure 13: The code of Figure 12 being run in the Console Pane. The function Celec is called repeatedly for RoI = 10, 20 and 15. It returns, respectively, the values 11.91666..., 23.8333.., 17.875.

2.6 Writing Classes

Now I will have to ask you to bear with me for a bit. The next bit can be difficult for some to wrap their minds around. However, it is a major feature of Python, and you deserve to have, at least in passing, a look at it.

A problem with the code as written in Figure 12 and run in Figure 13 is the sheer number of arguments that need to be passed. This is particularly tedious when we need to change only *one* argument. There must be some way of telling Python to hold the values of all the other arguments constant while we change just one. There is. It just written in a weird way: look at Figure 14.

solarthermal_classes.py

```
class SolarThermal:
    I = 800.0 #W/m2
    eta_DNI = 0.75; eta_cap = 0.5; eta_conv = 0.3
    P = 500.0 #MW
    Dop = 24; Dsun = 6 #hours
    Yop = 8000 #hours per year
    RoI = 10.0 #%
    phi_land = 0.25
    Cmirror = 20000 #Rs/m2
    Cland = 250000 #Rs/hectare
    Cplant = 60000 #Rs/kW
    def Celec(self):
        I = self.I
        eta_DNI = self.eta_DNI; eta_cap = self.eta_cap; eta_conv = self.eta_conv
        P = self.P
        Dop = self.Dop; Dsun = self.Dsun
        Yop = self.Yop
        RoI = self.RoI

        phi_land = self.phi_land
        Cmirror = self.Cmirror
        Cland = self.Cland
        Cplant = self.Cplant

        #Unit Conversion and Standardization
        P *= 1e6 #W
        Dop *= 3600.0; Dsun *= 3600.0; Yop *= 3600.0 #s
        Cland /= 10000.0 #Rs/m2
        Cplant /= 1000.0 #Rs/W

        #Calculations
        esp = I*eta_DNI*eta_cap*eta_conv
        Edaily = P*Dop;  Psun = Edaily/Dsun
        Amirror = Psun/esp;  Aland = Amirror/phi_land

        CT = Amirror*Cmirror + Aland*Cland + P*Cplant;  Eyearly = P*Yop

        Celec = RoI/100.0*CT/Eyearly; Celec_kWh = Celec*3600.0*1000.0
        return Celec_kWh
```

Figure 14: Using classes to write more user friendly code. The power of this approach is illustrated in FIG

Here we define a *class* called 'SolarThermal'. In like manner to the function definition, this definition begins with the word 'class' followed by the name 'SolarThermal' followed immediately by a colon ':'. There are *no* parentheses!

Everything that follows is indented with respect to the class definition line (line 1 in Figure 14). We input the standard/default values of the various parameters as shown. Each parameter *e.g. I, Dop, RoI etc* are now called *attributes* of the class SolarThermal.

We now write the definition of the *Celec* function: only now it is *also* indented wrt the class definition. Hence *Celec* is also an *attribute* of this class: it is a *method* attribute.

Now when we write the *Celec* function, we have to pass *just* one argument. And that is the keyword *self*. This argument *self* is *implicit* i.e. the user does not have to pass it. To access the attributes of the class from within *Celec*, we use *self* e.g. to get the value of *I* we use *self.I (*see line 13 in Figure 14). Everything else is as before. Now lets run this ('F5', select appropriate options, press 'Run' button). A new tab called 'solarthermal_classes.py' opens in the Console Pane.

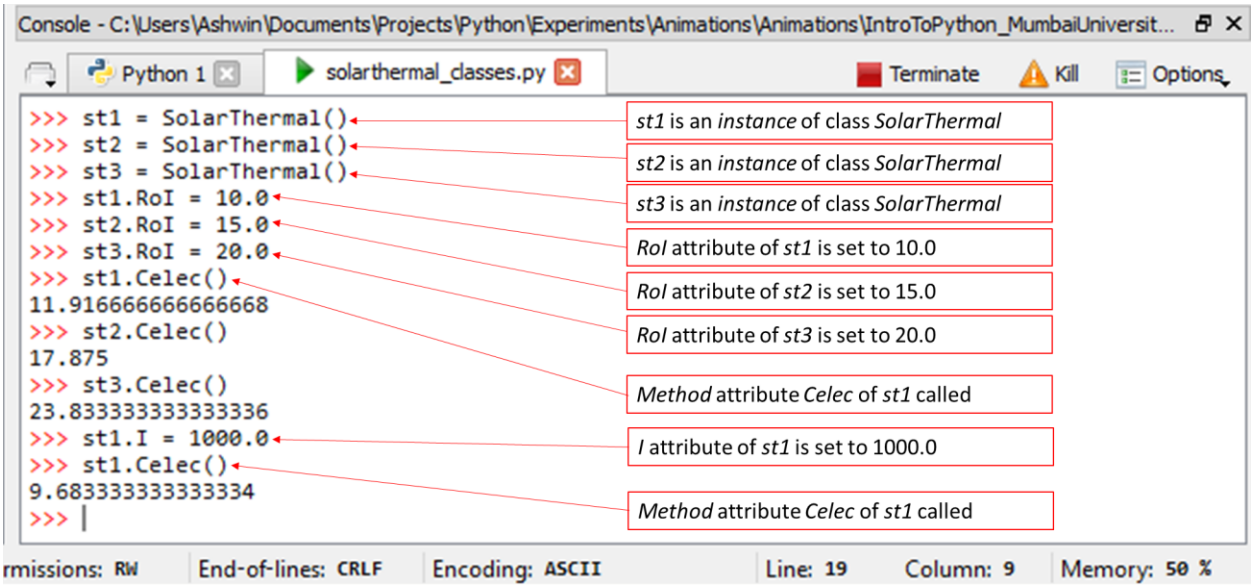


Figure 15: Use of classes. Shown are three instances *of class SolarThermal (st1, st2, st3) each with their RoI attributes set to different values. Then the Celec method of each class is called and the results are seen. Then the I attribute of st1 is modified and st1.Celec called again.*

Use of our newly minted *SolarThermal* class is illustrated in Figure 15. To use a class, we have to create an *instance* of that class. In Figure 15 we see three *instances* of class *SolarThermal*: *st1, st2, st3*. Each instance starts life with the *same* value for all attributes. But then we can modify the attributes of each class. Hence, we have changed the *RoI* value of each class to 10.0, 15.0 and 20.0 for *st1, st2* and *st3* respectively. Now when we call the method attribute *Celec* for each instance, we get different results (11.91666..., 17.875, 23.833... respectively).

We can change another attribute e.g. we have changed the *I* attribute of *st1* to 1000.0 and called *st1.Celec()* to yield 9.6833... .

Clearly this is a far more user friendly way of programming. Just as an example of its power, suppose you want to *add* a refinement to the model e.g. cost of thermal storage say, you would need several more parameters to be passed to the function in Figure 13 *by the user* i.e. you will have to go back and change several lines of code *every* time you decide to refine your model. However, with the use of classes (Figure 15) the user interface remains *exactly* the same. The code is now truly *modular*.

(**Note:** This type of programming is called *Object Oriented Programming*: a revolution in coding. Saved a lot of programmers a lot of time and trouble over the years. C++ is another Object Oriented language, as is Java.)

2.7 Lists and For Loops

Now, the purpose we did all this Object Oriented programming was because we wanted to automate *sensitivity* analysis i.e. we wanted the computer to automate test cases e.g. say we want to know the effect of *RoI* on *Celec* for different values of *Cmirror*. We have to have some way of telling the Python interpreter what values of *RoI* we want to work with. The simplest way of doing this is to make a *list* of the *RoI* values.

It is very easy to make a list in Python. The Figure 16 shows two lists: *list_of_RoI* which is a list of the *RoI* values that we would be interested in, and *list_of_Cmirror* which is a list of the *Cmirror* values we are interested in. Each list in enclosed in *brackets* ('[ ]') and the elements of the list are separated by commas.

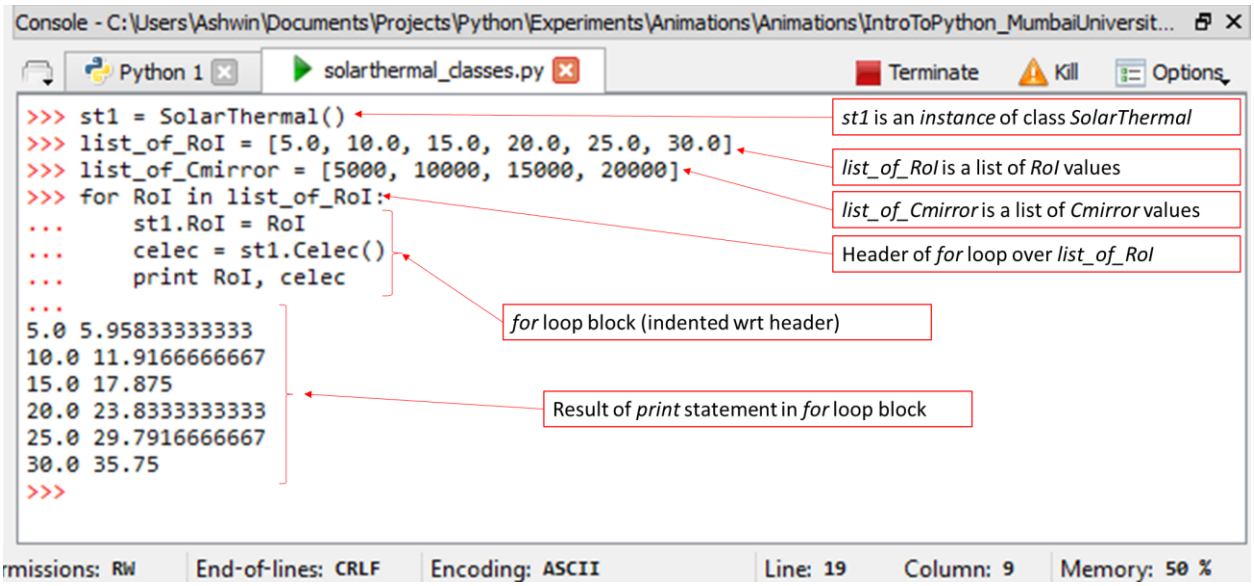


Figure 16: Use of lists and for-loops in Python.

The use of a *for* loop in Python is slightly different from other common languages. The *for* loop iterates over *elements* of a list. Hence in Figure 16, each element of *list_of_RoI* is successively stored in the variable *RoI*. In the *for* block, the value of the variable *RoI* is assigned to the *RoI* attribute of the class instance *st1*. Then the *Celec* method attribute of *st1* is called and the value returned assigned to the variable *celec*. Finally *RoI* and *celec* are printed out to the screen and the *next* element of *list_of_RoI* is assigned to the variable *RoI* and the process is repeated until all elements of the list are used up. Very simple and convenient.

But there is a problem: we can't *do* anything with the values of *celec*. We could copy them and paste them in Excel (for instance) and plot them etc. But that is tedious. And unnecessary. Python provides a simple way of making *another* list of values of *celec*. Figure 17 shows a way of creating an empty list (just '[]') and adding element after element to it. Two lists can be *concatenated* simply by adding them i.e. *[1,2,4,7] + [2,3,7,1] will result in [1,2,34,7,2,3,7,1]*.

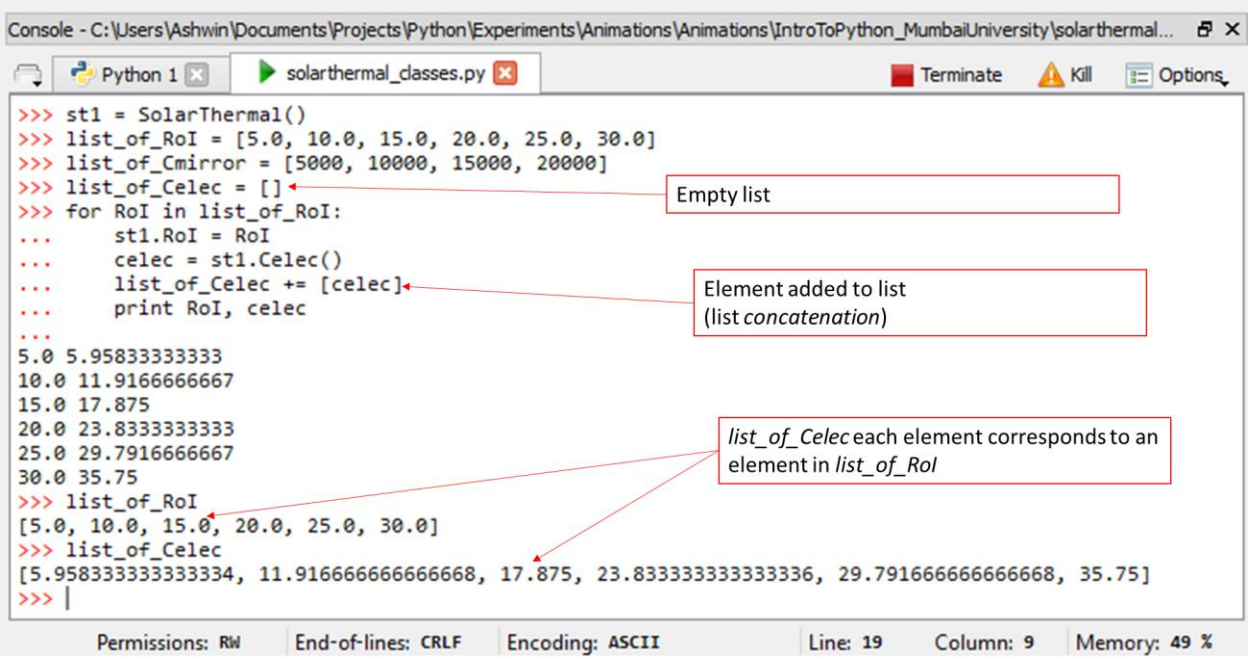


Figure 17: Creating an empty list and populating it.

You can also have a list *of* lists. See Figure 18.

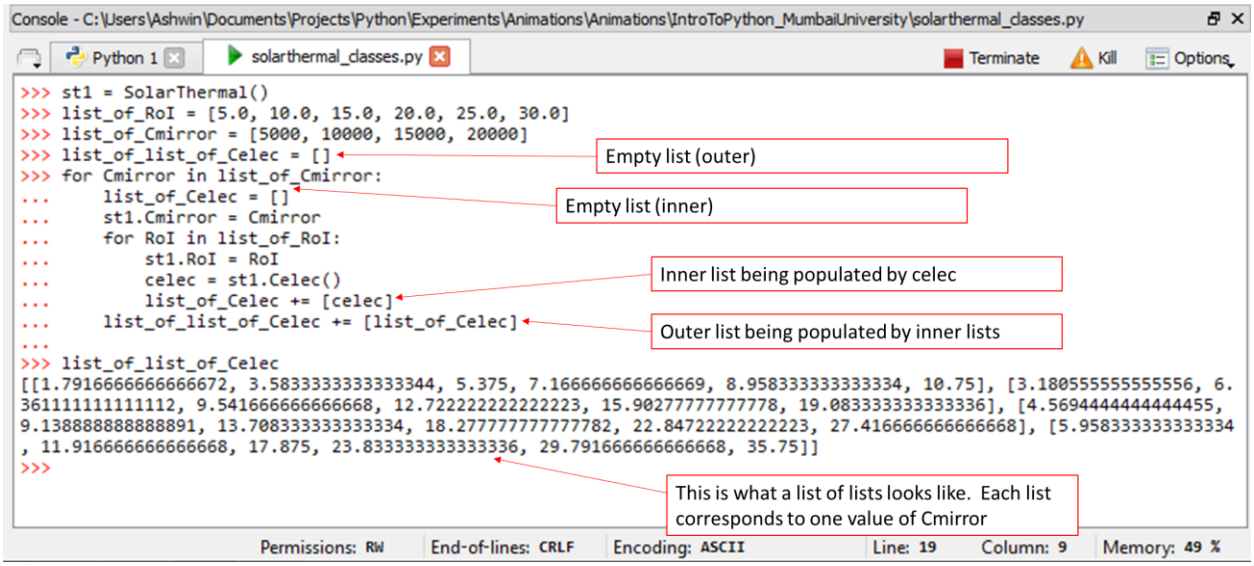


Figure 18: Making a list of lists and nested for loops.

In Figure 18 we see an example of *nested* for loops (one for loop inside the block of another) as well as a *list of lists*. This is a very powerful feature of Python. It enables us to handle information very efficiently. It also has other benefits, which we will see.

2.8 Plotting

A very useful feature for scientific analysis is the ability to plot data. Excel (well) *excels* at allows users to quickly plot and analyse data. Python does the same. In some ways, it is actually easier

to plot data in Python than it is in Excel. And we shall soon see that it need not be *either-or*. Our favourite word is *both*!

So what does it take to plot something in Python. Well, first we need something to plot: that is a set of x and y coordinates with a one-to-one correspondence i.e. there must be exactly the same number of x coordinates as there are y coordinates and both sets must be *ordered*. In other words, we need two lists: a list of x coordinates and a list of y coordinates.

Let's plot the *sine* function between 0 and 2π (both inclusive) in 20 points. So the points in the list of x coordinates are given by: $\frac{2\pi}{19}i$ where $i = 0,1,2,\ldots,19$. Hence the points in the list of y coordinates are given by: $\sin\left(\frac{2\pi}{19}i\right)$.

Now that we have our lists, we need to put them in a picture. This would have normally been very heavy work, but Python has a fully equipped library called *Matplotlib*. We will use the *pyplot* sub-module of the *matplotlib* library. The code is shown in Figure 19.

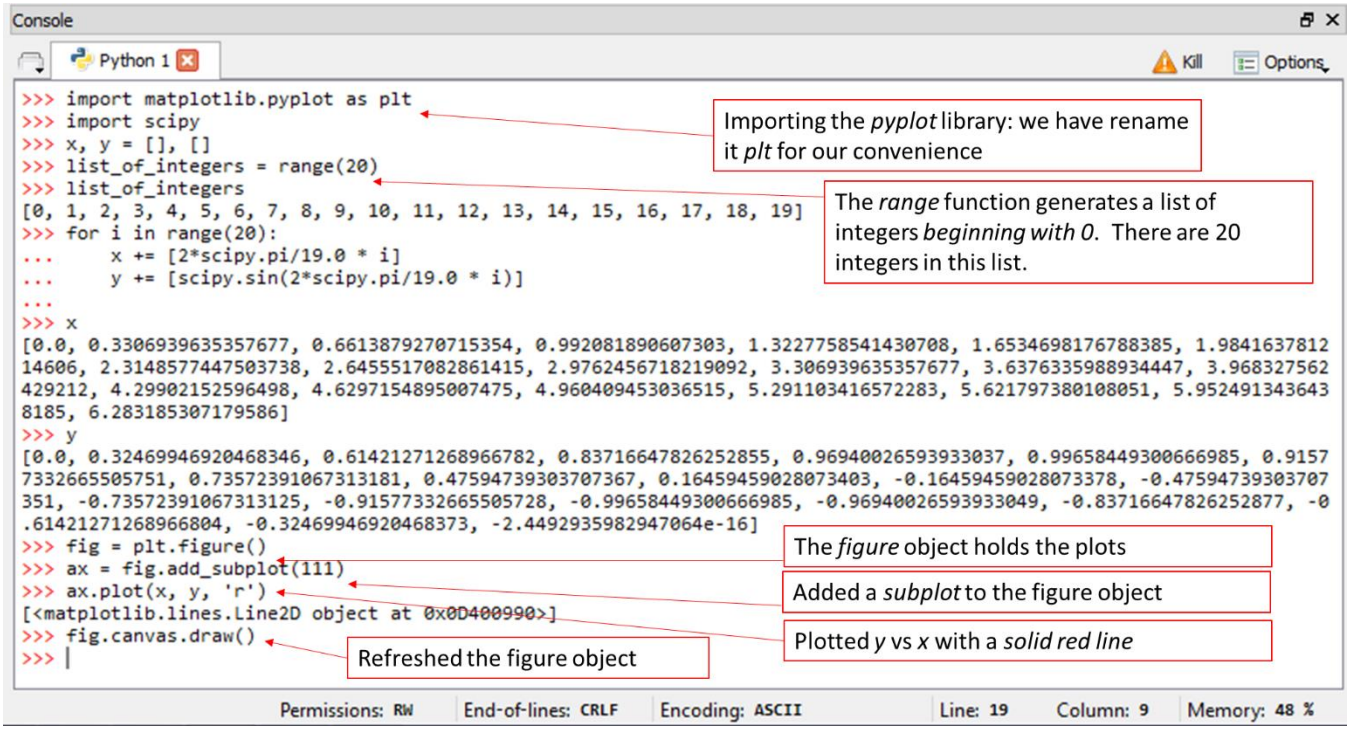


Figure 19: Plotting using matplotlib.pyplot

You will see a *new window* has opened on your screen (it might be minimized). Expand it and you should see something like Figure 20.

Now add the lines shown in Figure 21 and you should get Figure 22.

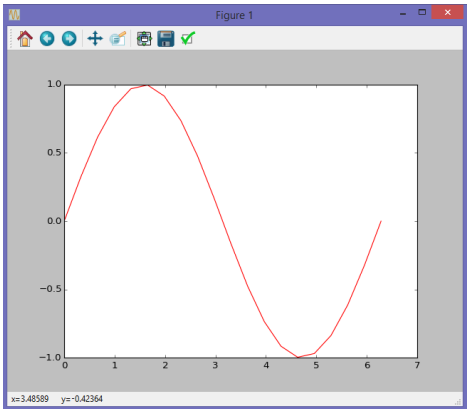


Figure 20: Your first plot in Python.

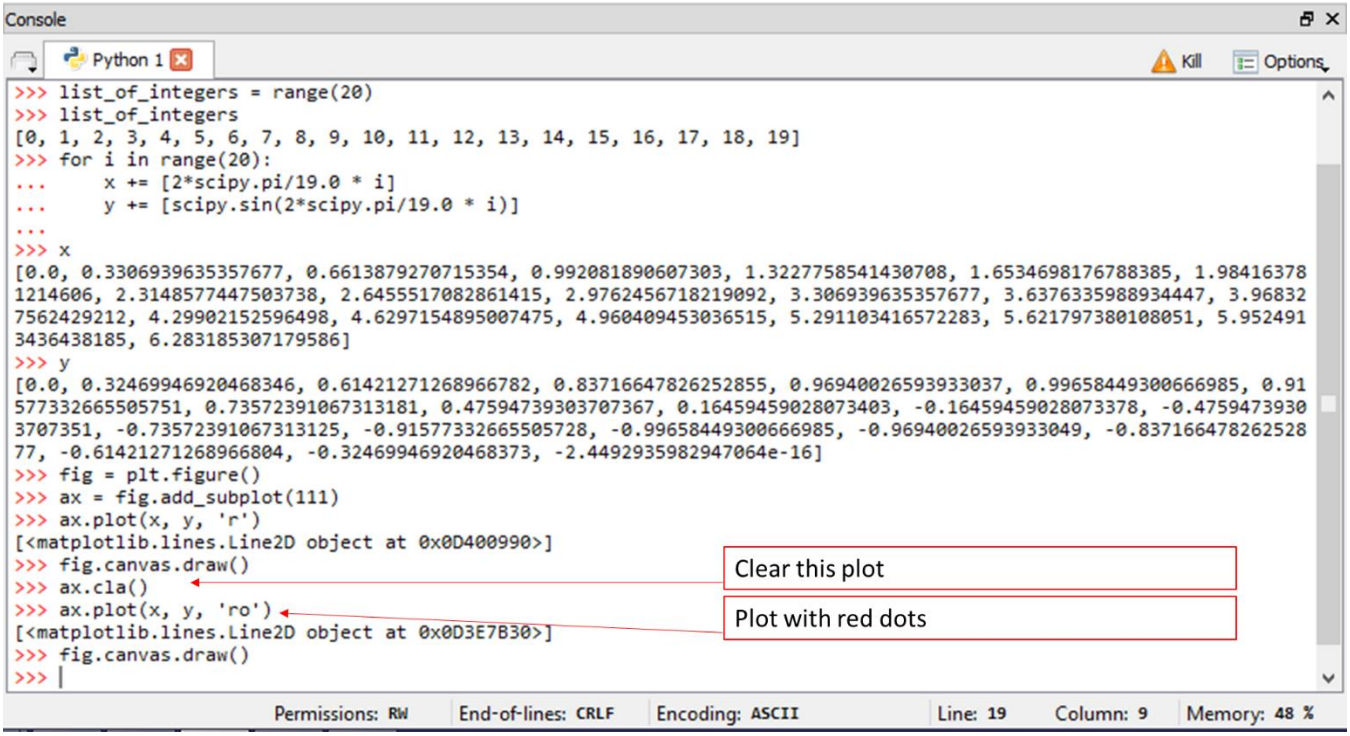


Figure 21: Plotting in red dots

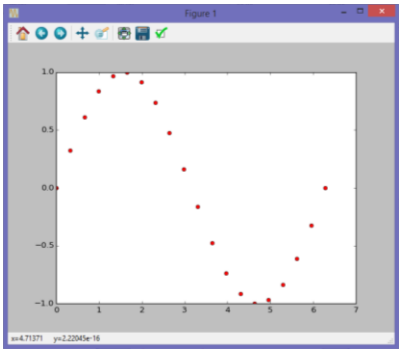


Figure 22: Plot with red dots

2.9 Vectors and Arrays

Now, this is all very well. But as the MATLAB tag-line says: "*Life is too short to write for loops.*" MATLAB became very successful and widely used because of its innovation of using *arrays* as the basic using of computation. Python offers the same functionality. We can turn any list into an array by *casting* it.

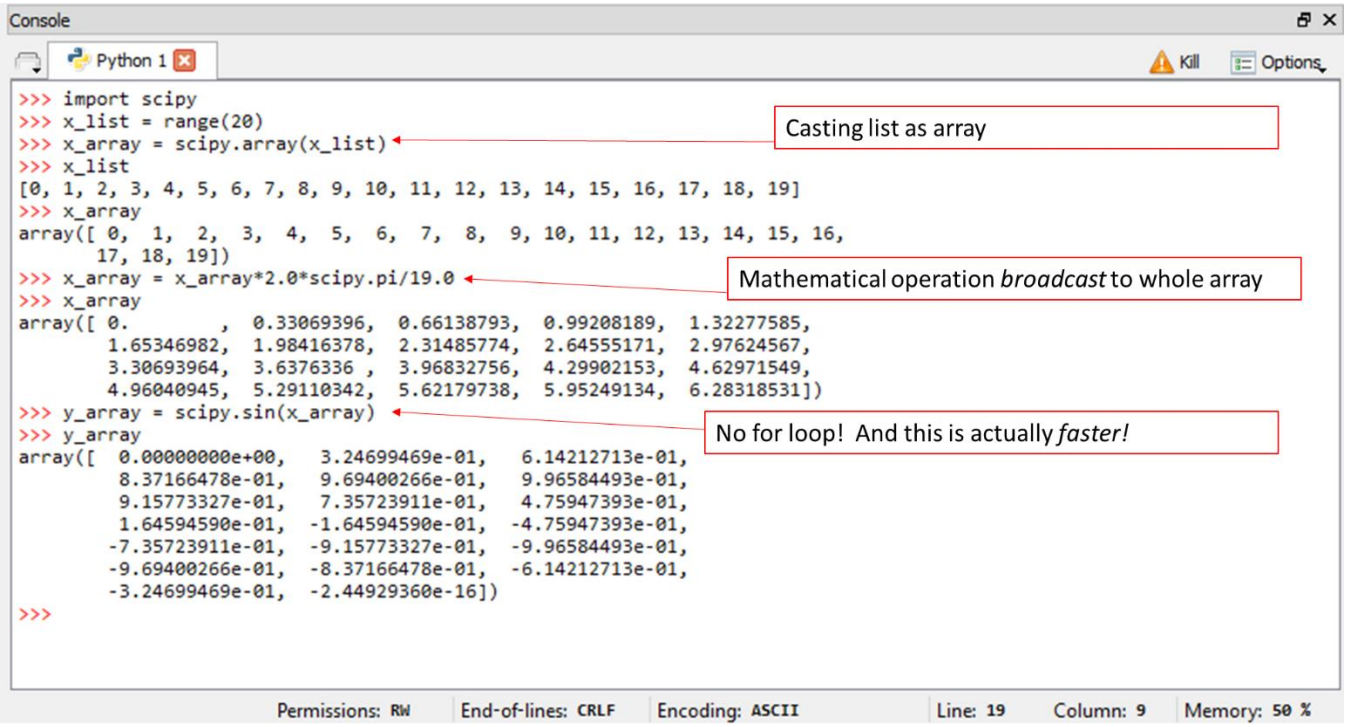


Figure 23: Casting a list into an array and the power of using arrays.

Figure 23 shows how to cast a list into an array and the power of using arrays. Mathematical operation performed on the array as a whole are *broadcast* to each element of the array. Hence we don't need *for* loops.

The Figure 21 shows a quick and elegant way of plotting the *sine* function between 0 and 2π. The resulting plot is shown in Figure 25.

Note the use of the *scipy.linspace* function which returns an evenly spaced grid of points between two end points.

Figure 26 shows some other standard array operations that are available to Python. There are *many many* more!

```
Console
Python 1                                                    Kill  Options
>>> import scipy
>>> import matplotlib.pyplot as plt
>>> x = scipy.linspace(0,2*scipy.pi, 20)
>>> y = scipy.sin(x)
>>> fig = plt.figure(); ax = fig.add_subplot(111)
>>> ax.plot(x,  y, 'r')
[<matplotlib.lines.Line2D object at 0x0D400710>]
>>> ax.plot(x, y, 'b^')
[<matplotlib.lines.Line2D object at 0x0D457E10>]
>>> fig.canvas.draw()
>>>
>>> x
array([ 0.        ,  0.33069396,  0.66138793,  0.99208189,  1.32277585,
        1.65346982,  1.98416378,  2.31485774,  2.64555171,  2.97624567,
        3.30693964,  3.6376336 ,  3.96832756,  4.29902153,  4.62971549,
        4.96040945,  5.29110342,  5.62179738,  5.95249134,  6.28318531])
>>> y
array([  0.00000000e+00,   3.24699469e-01,   6.14212713e-01,
         8.37166478e-01,   9.69400266e-01,   9.96584493e-01,
         9.15773327e-01,   7.35723911e-01,   4.75947393e-01,
         1.64594590e-01,  -1.64594590e-01,  -4.75947393e-01,
        -7.35723911e-01,  -9.15773327e-01,  -9.96584493e-01,
        -9.69400266e-01,  -8.37166478e-01,  -6.14212713e-01,
        -3.24699469e-01,  -2.44929360e-16])
>>>
Permissions: RW  End-of-lines: CRLF  Encoding: ASCII  Line: 19  Column: 9  Memory: 49 %
```

linspace is a *convenience* function for quickly getting an array of evenly spaced points

Figure 24: Using the power of arrays to greatly simplify the code of Figure 21

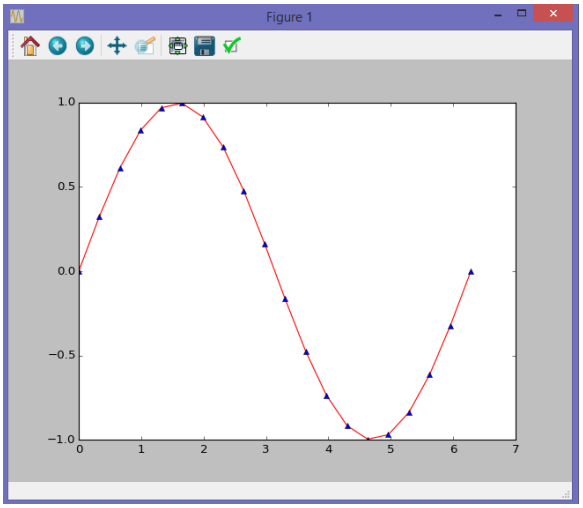


Figure 25: An example of multiple plots on the same canvas.

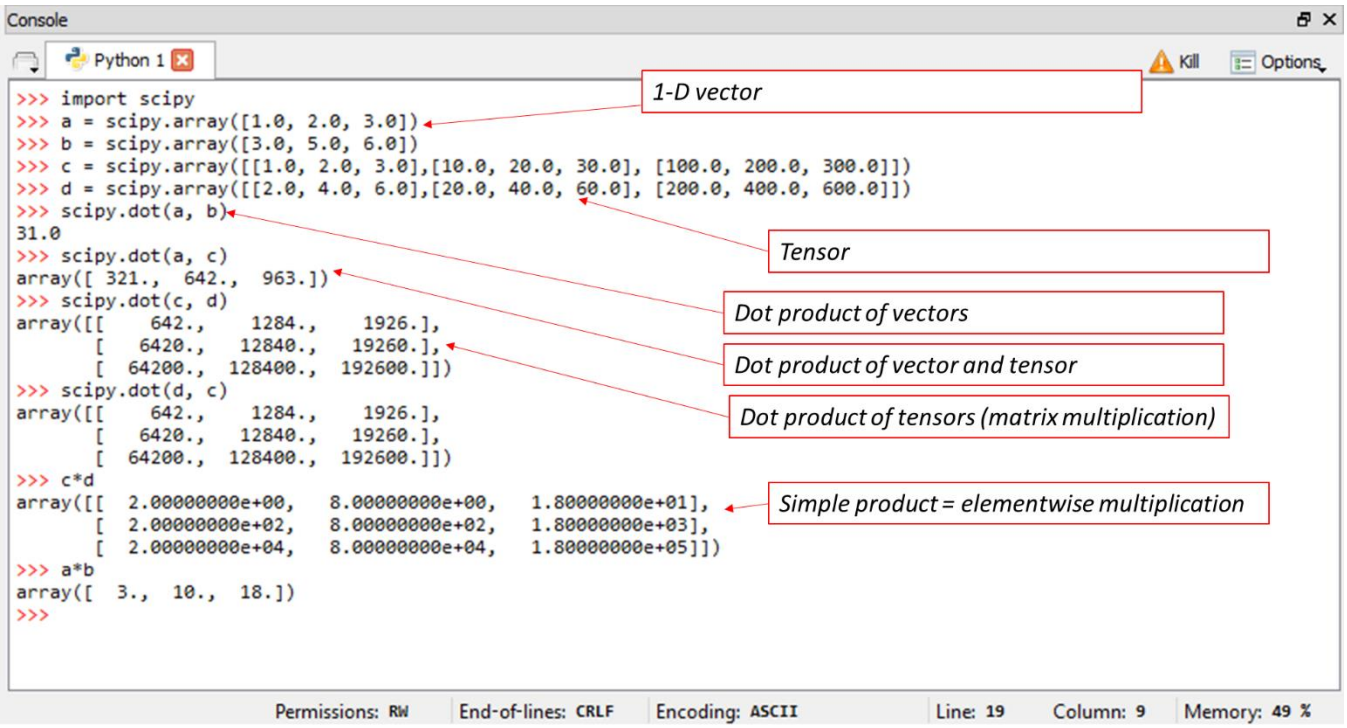


Figure 26: Vectors and tensors in Python

2.10 Importing Own Modules

In Figure 18 etc we saw how to use code written in the Editor Pane in the Console Pane. However, a file with .py extension is also a *Python Module* in its own right and can be imported into *other* modules.

Let's use this concept to extend the analysis of Figure 18. We will make a new file called 'solarthermal_plots.py' into which we will import *scipy*, *matplotlib.pyplot* and *solarthermal_classes*.

(**Note:** For user modules, the module you import *must* be in the same folder as your current program file.)

Figure 27 shows how it is done. It also shows some more sophisticated plotting, the result of which appear in Figure 28. Can you see where we wrote the code for giving the axis lables?

solarthermal_classes.py | solarthermal_plots.py

```
1 import scipy
2 import matplotlib.pyplot as plt
3 import solarthermal_classes as STC
4
5 st1 = STC.SolarThermal()
6 list_of_RoI = [5.0, 10.0, 15.0, 20.0, 25.0, 30.0]
7 list_of_Cmirror = [5000, 10000, 15000, 20000]
8 list_of_list_of_Celec = []
9
10 for Cmirror in list_of_Cmirror:
11     st1.Cmirror = Cmirror
12     list_of_Celec = []
13     for RoI in list_of_RoI:
14         st1.RoI = RoI
15         celec = st1.Celec()
16         list_of_Celec += [celec]
17     list_of_list_of_Celec += [list_of_Celec]
18
19 fig = plt.figure()
20 ax = fig.add_subplot(111)
21
22 list_lines = ['r','b','g','c']
23 list_legend = []
24 for i in range(len(list_of_Cmirror)):
25     Cmirror = list_of_Cmirror[i]
26     list_of_Celec = list_of_list_of_Celec[i]
27     ax.plot(list_of_RoI, list_of_Celec, list_lines[i])
28     list_legend += ['Cmirror=%f'%(Cmirror)]
29 ax.legend(tuple(list_legend))
30 ax.xaxis.label.set_text('RoI %')
31 ax.yaxis.label.set_text('Cost of Electricity in Rs/kWh')
32 fig.canvas.draw()
```

Module imported as STC

SolarThermal class accessed through STC

Permissi

Figure 27: Importing own modules into another module

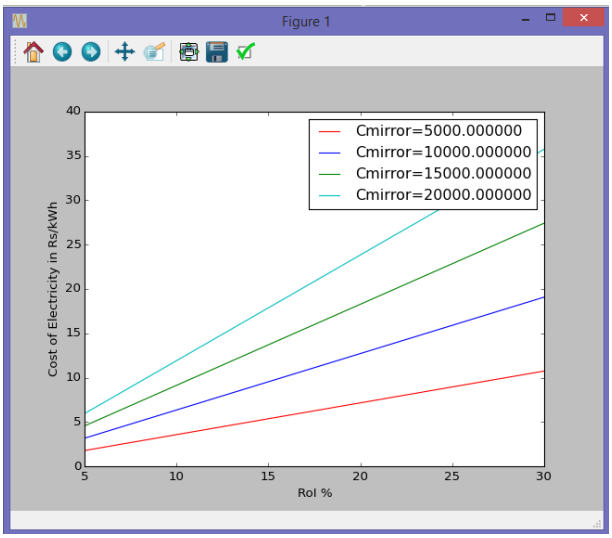
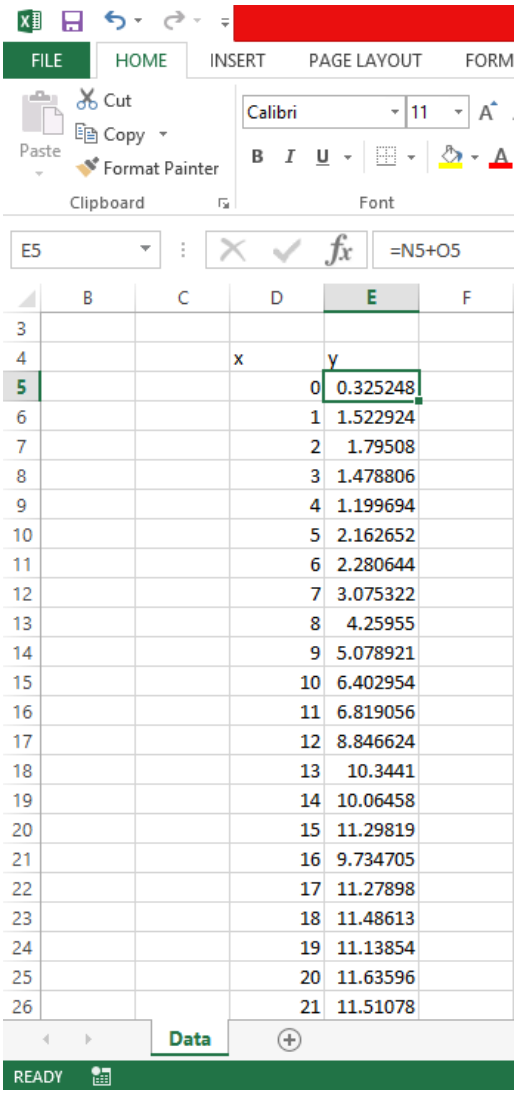


Figure 28: Result of code of Figure 27

2.11 Importing Data From Excel

Excel (or a spreadsheet tool) is by far the most effective way of communicating and manipulating information. Python can quickly and painlessly get data from Excel. Consider the data in Figure 29. The x-values are in the range "D5:D26" while y values are in the range "E5:E26".



	B	C	D	E	F
3					
4			x	y	
5			0	0.325248	
6			1	1.522924	
7			2	1.79508	
8			3	1.478806	
9			4	1.199694	
10			5	2.162652	
11			6	2.280644	
12			7	3.075322	
13			8	4.25955	
14			9	5.078921	
15			10	6.402954	
16			11	6.819056	
17			12	8.846624	
18			13	10.3441	
19			14	10.06458	
20			15	11.29819	
21			16	9.734705	
22			17	11.27898	
23			18	11.48613	
24			19	11.13854	
25			20	11.63596	
26			21	11.51078	

Figure 29: Data in Excel worksheet called 'Data' in workbook called 'Data.xlsx'.

The Figure 30 shows how to get data from the Excel sheet of Figure 29. The code is stored in the file 'exceldataimport.py'. Line 1 imports a special library called *win32com.client* using which we open the *Excel* application object as *xl* (line 5) and hence the workbook called 'Data.xlsx' (line 6) as *wb*. From *wb* we get the sheet 'Data' as *sheet* (line 7). The function *getdata* converts data in a column in excel into a vector in Python. The code also plots the data.

Run this code to get the graph shown in Figure 31.

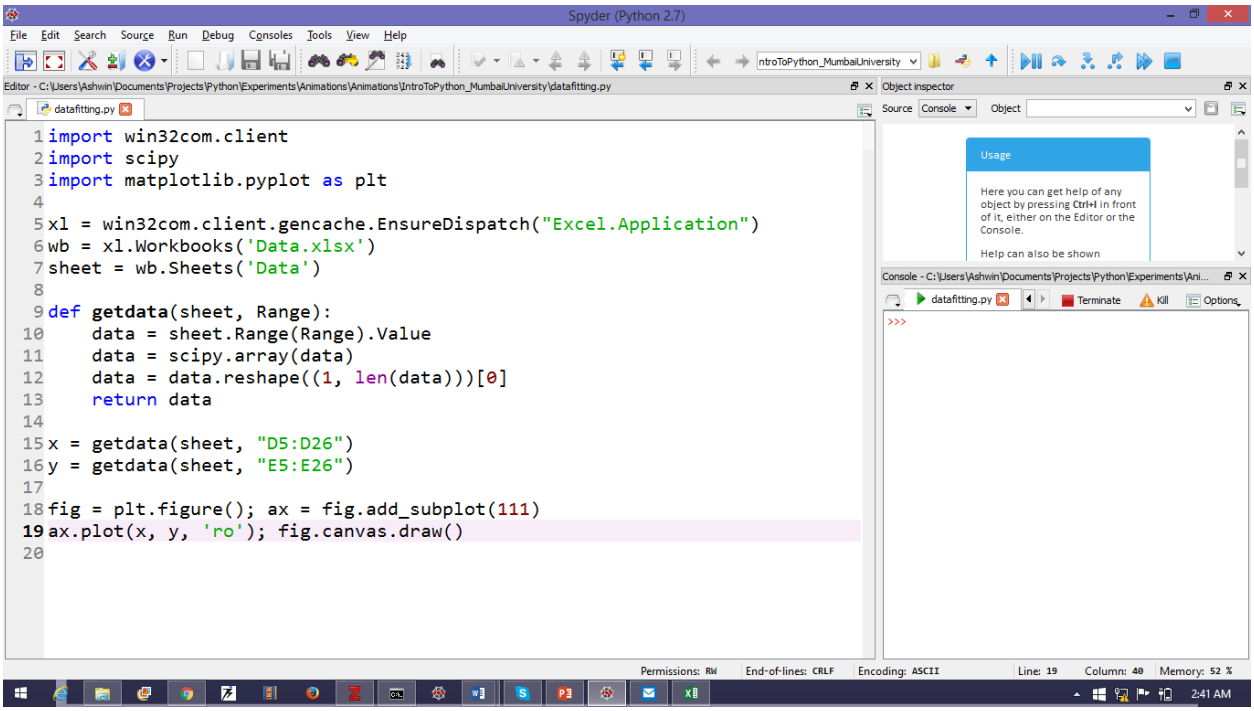


Figure 30: Getting data from the Excel sheet shown in Figure 29

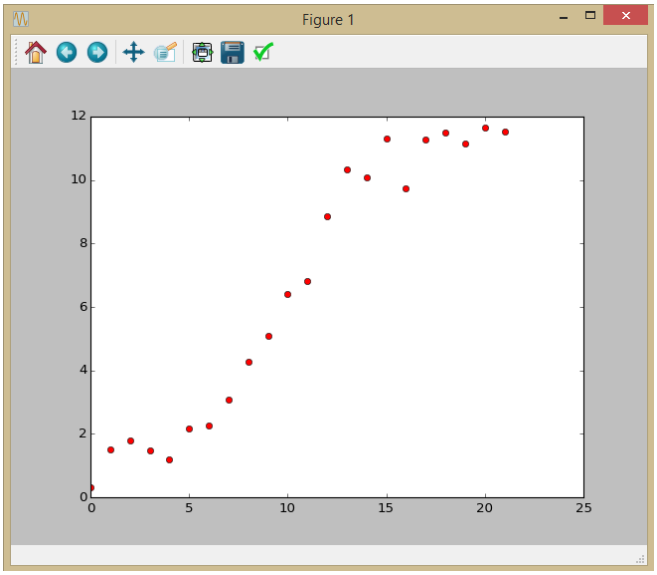


Figure 31: Graph from running code in Figure 30

2.12 Fitting Data

Now that we have imported data, it remains to fit it. For this we use the *Levenberg-Marquadt* algorithm available to us via *scipy.optimize.leastsq* function. The full code is written in the file called 'datafitting.py' (see Figure 32).

Editor - C:\Users\Ashwin\Documents\Projects\Python\Experiments\Animations\Animations\IntroToPython_Mumba

datafitting.py

```
1  import win32com.client
2  import scipy, scipy.optimize
3  import matplotlib.pyplot as plt
4  xl = win32com.client.gencache.EnsureDispatch("Excel.Application")
5  wb = xl.Workbooks('Data.xlsx')
6  sheet = wb.Sheets('Data')
7  def getdata(sheet, Range):
8      data = sheet.Range(Range).Value
9      data = scipy.array(data)
10     data = data.reshape((1, len(data)))[0]
11     return data
12 x = getdata(sheet, "D5:D26")
13 y = getdata(sheet, "E5:E26")
14 def curve(x, p):
15     [A, B, C, D] = p
16     y = A + B*scipy.tanh(C*(x-D))
17     return y
18 def error(p, x, yexp):
19     ycalc = curve(x, p)
20     err = ycalc - yexp
21     return err
22 def get_r2(x, y, ycalc):
23     ymean = scipy.average(y)
24     dymean2 = (y - ymean)**2
25     dycalc2 = (y - ycalc)**2
26     r2 = 1 - sum(dycalc2)/sum(dymean2)
27     return r2
28 pguess = [6.0, 6.0, 0.5, 12.0]
29 plsq = scipy.optimize.leastsq(error, pguess, args = (x, y))
30 p = plsq[0]
31 ycalc = curve(x, p)
32 r2 = get_r2(x, y, ycalc)
33
34 fig = plt.figure(); ax = fig.add_subplot(111)
35 ax.plot(x, y, 'ro');
36 ax.plot(x, ycalc, 'b')
37 ax.title.set_text('r2=%f'%(r2))
38 fig.canvas.draw()
```

Figure 32: Code for fitting data shown in Figure 29. The result of running this appears in Figure 33.

Let's unpack the code in Figure 32 a bit. Upto line 13 we have already seen in Figure 30. So we have our x and y arrays. Now we need a likely curve to fit these to. From the nature of the graph shown in Figure 31, we can hazard that the curve is a hyperbolic tangent i.e.

$$y = A + B \times \tanh\big(C(x - D)\big)$$

Here *A, B, C* and *D* are the parameters to be obtained from the fit.

First we write the function for the curve. Here it is called *curve* (line 14) and it takes two arguments: *x* and *p* where *p = [A, B, C, D]*. The function *curve* return the best estimate of *y* for a given *x*.

Next, we need to determine how good this estimate is. Hence, on line 18, we have defined the function *error*. The *error* function takes 3 arguments: *p*, the set of *x* and the set of *y*. In the error function, *curve* is called and the *ycalc* from *curve* compared with *y* and the set of errors is returned.

On line 28, we have given a best guess of the parameters based on a visual inspection of the Figure 31. This information is now given to the *scipy.optimize.leastsq* function (line 29). Notice that *p* is the *first* argument of *error* while *error* is the first argument of *leastsq*. This is because of the requirement of the *leastsq* function: it takes its second argument and treats it as the first argument of the function that is its first argument. The final argument of *leastsq* is *args = (x, y)*. Contained in the parantheses are the remaining arguments for *error*.

Note: In Python, you can pass *functions* as arguments.

leastsq returns a two-element tuple. The first element of this is the optimized set of the parameters *p*. Hence line 30.

Note: In Python, indexing begins at 0.

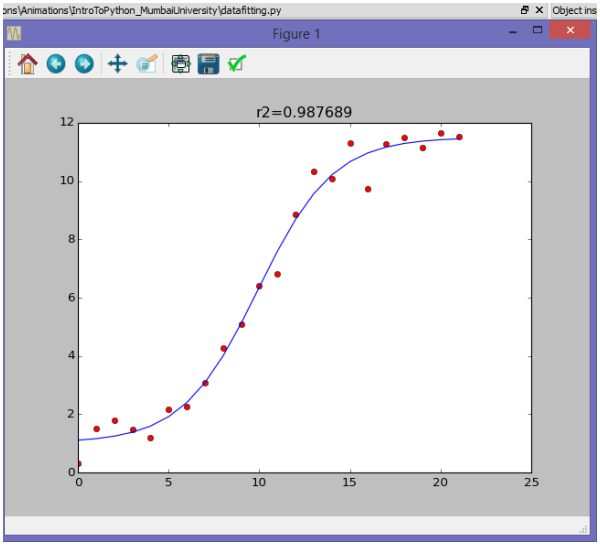


Figure 33: Fitted Curve to Data shown in Figure 29

3. Citation

If you use Python in your research, please cite the following somewhere in your paper:

For Scientific Python (scipy): Oliphant, T. E. "Python for Scientific Computing." Comput. Sci. Eng. 9, 10–20 (2007).

For Matplotlip (pyplot/plt): J. D. Hunter, "Matplotlib: A 2D Graphics Environment," Comput. Sci. Eng., vol. 9, no. 3, pp. 90–95, 2007.

4. Conclusion

There is lots more that Python can do. Python has modules for differential equation solution, data-analysis, machine learning, signal processing, wavelet transforms, computer vision, 3D visualization, efficient data storage and retrieval, web development, game development and any number of other fun and useful features. And more are being added every day.

"This is not the end. This is not even the beginning of the end. This is the end of the beginning"

■ *Winston Churchill*