Writing Python code

Aksel Hiorth, the National IOR Centre & Institute for Energy Resources,

University of Stavanger

Feb 8, 2022

Contents

In this chapter we cover some aspects of how to write (good) Python code. As you might have discovered tasks can be solved in many different ways in Python. This is clearly a strength because you would most likely be able to solve any task thrown at you. On the other hand it is a weakness, because code can get messy and hard to follow, especially if you solve the same task in different part of your code using different libraries or syntax.

In this chapter I will explain how I tend to solve some common tasks, in this process we will also cover some stuff that you should know. If you need more information on each topic below, there are plenty of online sources.

The code examples are meant as an inspiration, and maybe you do not agree and have solutions that you think are much better. If that is the case I would love to know, and I can update this chapter.

Speed and readability.

Many people are concerned about speed or execution time. My advice is to focus on readable code, and get the job done. When the code is working it is very easy to go back and change out parts that are slow. Particularly you can use the magic commands `%timeit` to check performance of different functions. There is also the option of using Numba, which translate python code into optimized machine code.

1 Personal Guidelines

It is important to have a some guidelines when coding, and for Python there are clear style guides PEP 8. Take a look at the official guidelines, and make some specific rules for yourself, and stick to them. The reason for this is that if you make a large code, people will recognize your style and it is easier to understand the code. If you are working in team, it is even more important - try to agree on some basic rules, e.g.

Code Guidelines:

- Variable names should be meaningful
- Naming of variables and functions, should you write `def my_fynction(...):` or `def MyFunction(..)`, i.e are words separated by underscore or capital letter. Personally I use capital letters for class definition, and underscore for function definitions.
- (Almost) always use doc string, you would be amazed how easy it is to forget what a function does. Shorter (private) functions usually do not need comments or doc strings, *if you use good variable names* - it should then be easy to understand what is happening by just looking at the code.
- Inline comments should be used sparingly, only where strictly necessary.
- Strive to make code general, in particular not type specific. In Python it is easy to make functions that will work if a list (array) or a single value is passed.
- Use exception handling, in particular for larger projects.

- DRY - Do not Repeat Yourself [1]. If you need to change the code more than one place to extend it, you will forget.
- The DRY principle also applies to *knowledge sharing*, it is not only about copy and paste line of code, but knowledge should only be represented in one place.
- Import libraries using the syntax `import library as ..`, Numpy would typically be `import numpy as np`. The syntax `from numpy import *` could lead to conflicts between modules as functions could have the same name in two different modules.

Work Guidelines:

- Do not copy and paste code without understanding it. It is OK to be inspired of others, but in some cases the code example are unnecessary complicates, but perhaps more important you will get a code with a mix of different styles.
- Stick to a limited number of libraries, I try to do as much as possible with Numpy, Pandas, and Matplotlib. I only do plotting with Matplotlib, and do not use the build in functionality of Pandas.
- Unexpected behavior of functions, functions should be able to discover if something are wrong with arguments, and give warnings.

1.1 Code editor

You would like to use an editor that gives you some help. It is particularly useful when you do not remember a variable or function name, you can guess at the name and a drop down list will appear which will let you pick the name or function you want. If you enter a function name, the editor will write some useful information about the function, some screenshots are shown in figure 1.

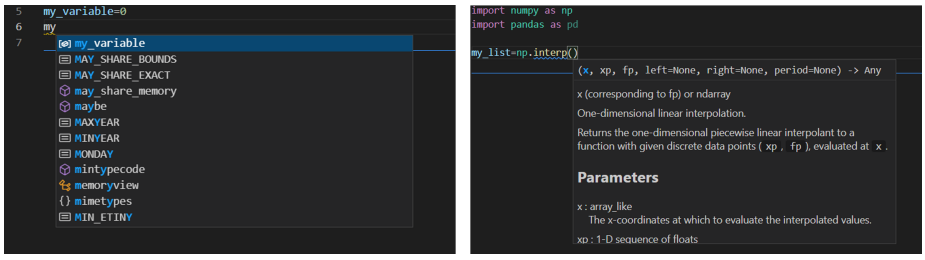

Figure 1: A screenshot of vscode (left) the editor helps to identify which variable name you mean, (right) the editor show relevant of the function you would like to call.

Currently my favorite editor is vscode, it can be used for any language, and there are a lot of add ins that can be installed to make the coding experience more pleasant. Spyder is also a very good alternative, but this editor is mainly for Python. It takes some time to learn how an editor work, so it is good if it can be used for multiple purposes. However, always be open to new ideas and products, it will only make you more efficient. As an example, in some cases you will have a particular difficult error in the code, and then it could help to open and run that code in a different editor, you might get a slightly different error messages, which could help you locate the error.

2 Types in Python

In Python you do not need to define types as in a compiled language, this an advantage as you need to do less writing, and it is also easier to create functions that will work with any kind of type.

2.1 Basic types

I will assume that you are familiar with the common types like floats (for real numbers), strings (text, lines, word, a character), integer (whole numbers), Boolean (True, False). What is sometimes useful is to be able to test what kind of type a variable is, this can be done with `type()`

```
my_float = 2.0
my_int   = 3
my_bool  = True
print(type(my_float))
print(type(my_int))
print(type(my_bool))
```

The output of the code above will be `float, int, bool`. If you want to test the value of a variable you can do

```
if isinstance(my_int,int):
    print('My variable is integer')
else:
    print('My variable is not integer')
```

Python also has build in support for complex number, they are written `1+2j`, `j` is used as the complex number. Note there is no multiplication between the number 2 and `j`.

2.2 Lists

Lists are extremely useful, and they have some very nice syntax that in my mind is more elegant than Numpy arrays. Whenever you want to do more than one thing with only a slight change, you should think of lists. Lists are defined using the square bracket `[]` symbols

```
my_list = []      # an empty list
my_list = []*10   # still an empty list ...
my_list = [0]*10  # a list with 10 zeros
my_list = ['one', 'two','three'] # a list of strings
my_list = ['one']*10 # a list with 10 equal string elements
```

Notice.

To get the first element in a list, we do e.g. `my_list[0]`. In a list with 10 elements the last element would be `my_list[9]`, the length of a list can be found by using the `len()` function, i.e. `len(my_list)=10`. Thus, the last element can also be found by doing `my_list[len(my_list)-1]`. However, in Python you can always get the last element by doing `my_list[-1]`, the second last element would be `my_list[-2]` and so on.

Sometimes you do not want to initialize the list with everything equal, and it can be tiresome to write everything out yourself. If that is the case you can use *list comprehension*

```
my_list = [i for i in range(10)] # a list from 0,1,..,9
my_list = [i**3 for i in range(10)] # a list with elements 0,1,8, ..,729
```

We will cover for loop below, but basically what is done is that the statement `i in range(10)`, gives `i` the value 0, 1, ..., 9 and the first `i` inside the list tells python to use that value as the element in the list. Using this syntax, there are plenty of opportunities to initialize. Maybe you want to pick from a list words that contain a particular subset of characters

```
my_list  = ['hammer', 'nail','saw','lipstick','shirt']
new_list = [i for i in my_list if 'a' in i]
```

Now `new_list=['hammer', 'nail', 'saw']`.

List arithmetic. I showed you some examples above, where we used multiplication to create a list with equal copies of a single element, you can also join two lists by using addition

```
my_list  = ['hammer','saw']
my_list2 = ['screw','nail','glue']
new_list = my_list + my_list2
```

Now `new_list=['hammer', 'saw', 'screw', 'nail', 'glue']`, we can also multiply the list with an integer and get a larger list with several copies of the original list.

List slicing. Clearly we can access elements in a list by using the index to the element, i.e. first element is `my_list[0]`, and the last element is `my_list[-1]`. Python also has very nice syntax to pick out a subset of a list. The syntax is `my_list[start:stop:step]`, the step makes it possible to skip elements

```
my_list=['hammer', 'saw', 'screw', 'nail', 'glue']
my_list[:]      # ['hammer', 'saw', 'screw', 'nail', 'glue']
my_list[1:]     # ['saw', 'screw', 'nail', 'glue']
my_list[:-1]    # ['hammer', 'saw', 'screw', 'nail']
my_list[1:-1]   # ['saw', 'screw', 'nail']
my_list[1:-1:2] # ['saw','nail']
my_list[::1]    # ['hammer', 'saw', 'screw', 'nail', 'glue']
my_list[::2]    # ['hammer', 'screw', 'glue']
```

Sometimes you have lists of lists, if you want to get e.g. the first element of each list you cannot access those elements using list slicing, you have to use a for loop or list comprehension

```
my_list  = ['hammer','saw']
my_list2 = ['screw','nail','glue']
new_list=[my_list,my_list2]
# extract the first element of each list
new_list2 = [ list[0] for list in new_list]
```

```
new_list2=['hammer','screw']
```

When to use lists.

Use lists if you have mixed types, and as storage containers. Be careful when you do numerical computation to mix lists and Numpy arrays, adding two lists e.g. `[1,2]+[1,1]`, will give you `[1,2,1,1]`, whereas adding two Numpy arrays will give you `[2,3]`.

2.3 Numpy arrays

Numpy arrays are awesome, and should be your preferred choice when doing numerical operations. We import Numpy as `import numpy as np`, some examples of initialization

```
my_array=np.array([0,1,2,3]) # initialized from list
my_array=np.zeros(10) # array with 10 elements equal to zero
my_array=np.ones(10)  # array with 10 elements equal to one
```

A typical use of Numpy arrays is when you want to create equal spaced numbers to evaluate a function, this can be done in (at least) two ways

```
my_array=np.arange(0,1,0.2) # [0, 0.2, 0.4, 0.6, 0.8]
my_array=np.linspace(0,1,5) # [0., 0.25, 0.5, 0.75, 1.]
```

Note that in the last case, the edges of the domain (0,1) are included, and is probably the outcome you want in most cases.

Do not mix Numpy arrays and lists in functions.

If a function is written to use Numpy arrays as *arguments*, make sure that it *returns* Numpy arrays. If you have to use a list inside the function to e.g. store the results of a calculation, convert the list to a Numpy array before returning it by `np.array(my_list)`.

Array slicing. As with lists you can access elements in Numpy arrays in the same way as lists, the syntax is `my_array[start,stop,step]`

```
my_array=np.arange(0,6,1)
my_array[:]      # [0,1,2,3,4,5]
my_array[1:]     # [1,2,3,4,5]
my_array[:-1]    # [0,1,2,3,4]
my_array[1:-1]   # [1,2,3,4]
my_array[1:-1:2] # [1,3]
my_array[::2]    # [0,2,4]
```

However, as opposed to lists all the basic mathematical operations addition, subtraction, multiplication are meaningful (*if the arrays have equal length, or shape*)

```
my_array  = np.array([0,1,2])
my_array2 = np.array([3,4,5])
my_array+my_array2 # [3,5,7]
my_array*my_array2 # [0,4,10]
my_array/my_array2 # [0,.25,.4]
```

Note that the operations does what you would expect them to do. If we have arrays of arrays, we can easily access elements in the arrays

```
my_array  = np.array([[0,1,2],[3,4,5]])
my_array[0,:] # [0,1,2]
my_array[1,:] # [3,4,5]
my_array[:,0] # [0,3]
my_array[:,1] # [1,4]
```

Not the extra `[]` in the definition of `my_array`. Numpy arrays have a shape property, which makes it very easy to create different matrices. The array `[0,1,2,3,4,5]` has shape (6,), but we can change the shape to create e.g. a 2×3 matrix

```
my_array  = np.array([0,1,2,3,4,5])
my_array.shape = (2,3) # [[0,1,2],[3,4,5]]
my_array.shape = (3,2) # [[0,1],[2,3],[4,5]]
```

3 Looping

There are basically two ways of iterating through lists or to do a series of computations, using a for-loop or a while-loop. During a numerical computation

we typically iterate through time, from time zero to the end time to calculate e.g. the position of an object.

3.1 For loops

A typical example of a for loop is to loop over a list and do something, and maybe during the execution we would like to store the results in a list

```
numbers=['one','two','three','one','two']
result=[] # has to be declared as empty
for number in numbers:
    if number == 'one':
       result.append(1)
```

After executing this code `result=[1, 1]`. The `number` variable changes during the iteration, and takes the value of each element in the list. Note that I use `numbers` for the list and `number` as the iterator, this makes it quite easy to read and understand the code. In many cases you want to have the index, not only the element in the list

```
numbers  = ['one','two','three','one','two']
numerics = [  1   ,  2  ,   3   , 1   , 2   ]
result=[] # has to be declared as empty
for idx,number in enumerate(numbers):
    if number == 'one':
       result.append(numerics[idx])
```

After executing this code `result=[1, 1]`. In this case the function `enumerate(numbers)` returns two values: the index, which is stored in `idx`, and the value of the list element, which is stored in `number`.

In many cases you might be in a situation that you want to plot more than one function in a plot. It is then very tempting to copy and paste the previous code, but it is more elegant to use a for loop and lists

```
import numpy as np
import matplotlib.pyplot as plt
x_val   = np.linspace(0,1,100) # 100 equal spaced points from 0 to 1
y_vals  = [x_val,x_val*x_val]
labels  = [r'x', r'$x^2$']
cols    = ['r','g']
points  = ['-*','-^']
for idx,y_val in enumerate(y_vals):
    plt.plot(x_val,y_val,points[idx],c=cols[idx],label=labels[idx])
plt.grid()
plt.legend()
plt.show()
```

3.2 While loops

In most cases a for loop can also be written as a while loops and vice versa. In python you would prefer to use a for loop whenever you are iterating over a

fixed number of elements. This makes the code easy to read. In cases where we are waiting for input or time is involved it may make more sense to use a while loop. Typically you would use a while loop when you do not know at the start when to stop iterating. The syntax of the while loop is to do something while a condition is true

```
import numpy as np
finished = False
sum =0
while not finished:
      sum += np.random.random()
      if sum >= 10.:
               finished = True
```

In some cases we are iterating from t_0, t_1, etc. to a final time t_f, if we use a fixed time step, Δt, we can calculate the number of steps at the beginning i.e $N = \text{int}((t_f - t_0)/\Delta t)$, and use a for loop. On the other hand, in the more fancy algorithm we change the time step as the simulation proceeds and then we would choose a while loop, e.g. `while t0 <= tf:`.

3.3 Functions in Python

When to use functions? There is no particular rule, *but whenever you start to copy and paste code from one place to another, you should consider to use a function*. Functions makes the code easier to read. It is not easy to identify which part of a program is a good candidate for a function, it requires skill and experience. Most likely you will end up changing the function definitions as your program develops.

Use short functions.

Short functions makes the code easier to read. Each function has a particular task, and it does only one thing. If functions does too many tasks there is a chance that you will have several functions doing some of the same operations. Whenever you want to extend the program you might have to make changes several places in the code. The chance then is that you will forget to do the change in some of the functions and introduce a bug.

3.4 Defining a mathematical function

Throughout this course you will write many functions that does mathematical operations. In many cases you would also pass a function to another function to make your code more modular. Lets say we want to calculate the derivative of $\sin x$, using the most basic definition of a derivative $f'(x) = f(x+\Delta x) - f(x)/\Delta x$, we could do it as

```
def derivative_of_sine(x,delta_x):
    ''' returns the derivative of sin x '''
    return (np.sin(x+delta_x)-np.sin(x))/delta_x

print('The derivative of sinx at x=0 is :', derivative_of_sine(0,1e-3))
```

If we would like to calculate the derivative in multiple points, that is straight forward since we have used the Numpy version of $\sin x$.

```
x=np.array([0,.5,1])
print('The derivative of sinx at x=0,0.5,1 is :', derivative_of_sine(x,1e-3))
```

We will return in a later chapter why $\Delta x = 10^{-3}$ is a reasonable choice. However, the challenge with our implementation is that if we want to calculate the derivative of another function we have to implement the derivative rule again for that function. It is better to have a separate function that calculates the derivative

```
def f(x):
    return np.sin(x)

def df(x,f,delta_x=1e-3):
    ''' returns the derivative of f '''
    return (f(x+delta_x)-f(x))/delta_x
print('The derivative of sinx at x=0 is :', df(0,f))
```

Note also that we have put `delta_x=1e-3` as a *default argument*. Default arguments have to come at the end of the argument lists, `df(x,delta_x=1e-3,f)` is not allowed. All of this looks well and good, but what you would experience is that your functions would not be as simple as $\sin x$. In many cases your functions need additional arguments to be evaluated e.g.:

```
def s(t,s0,v0,a):
    '''
    s0 : initial starting point
    v0 : initial velocity
    a  : acceleration
    returns the distance traveled
    '''
    return s0+v0*t+a*t*t/2
```

How can we calculate the derivative of this function? If we try to do `df(1,s)` we will get the following message

```
TypeError: s() missing 3 required positional
           arguments: 's0', 'v0', and 'a'
```

This happens because the `df` function expect that the function we send into the argument list has a call signature `f(x)`. What many people do is that they use global variable, that is to define `s0, v0, a` at the top of the code. This is not a good solution. Python has a special variable `*args` which can be used to pass multiple arguments to your function, thus if we rewrite `df` like this

```
def f(x,*args):
    return np.sin(x)

def df(x,f,*args,delta_x=1e-3):
    ''' returns the derivative of f '''
    return (f(x+delta_x,*args)-f(x,*args))/delta_x
```

we can do (assuming `s0=0`, `v0=1`, and `a=9.8`)

```
print('The derivative of sinx at x=0 is :', df(0,f))
print('The derivative of s(t) at t=1 is :', df(0,s,0,1,9.8))
```

3.5 Scope of variables

In small programs you would not care about scope, but once you have several functions, you will easily get into trouble if you do not consider the scope of a variable. By scope of a variable we mean where it is available, first some simple examples

```
def f(x):
    a=10
    b=20
    return a*x+b
```

A variable created inside a function is only available within the function: Doing `print(a)` outside the function would create an error: `name 'a' is not defined`. What happens if we defined a variable `a` outside the function

```
a=2
def f(x):
    a=10
    b=20
    return a*x+b
```

If we first call the function `f(0)`, and then do `print(a)` Python would give the answer `2`, *not* `10`. A *local* variable `a` is created inside `f(x)`, that does not interfere with the variable `a` defined outside the function.

```
global a
a=2
def f(x):
    global a
    a=10
    b=20
    return a*x+b
```

The `global` keyword can be used to pass and access variables in functions: In this case `print(a)` *before* calling `f(x)` would give the answer `2` and *after* calling `f(x)` would give `10`.

Use of global variables.

Sometimes global variables can be very useful, and help you to make the code simpler. But, make sure to use a *naming convention* for them, e.g. end all the global variables with an underscore. In the example above we would write `global a_`. A person reading the code would then know that all variables ending with an underscore are global, and can be altered by many functions.

References

[1] David Thomas and Andrew Hunt. *The Pragmatic Programmer: Your Journey to Mastery*. Addison-Wesley Professional, 2019.

Index