Some advanced topics

Aksel Hiorth

University of Stavanger

Jan 12, 2024

Contents

In this chapter we have collected some topics that are slightly more advanced.

1 Running Python files from command line

So far we have used notebooks to write Python code. Jupyter notebooks ends with an extension `.ipynb`, and if you open them in a text editor like notepad or emacs it does not look like it does in your web browser. On the other hand you can also write Python code in files that have an extension `.py`, then you would typically use an IDE such as Visual Studio Code. In the `.py` files you cannot

write markdown text. Sometimes it can be advantageous to use Python files and not Jupyter notebooks, the notebooks can take time to open and then you have to manually execute the cells you want to run.

It can be very convenient to have some small programs that could do some simple tasks for you, e.g.

1. search for files of a specific type
2. remove files of a specific type
3. remove or print out files that are large
4. download data from the web or a server

The only thing you have to do is to write your code in a suitable editor and save it, if you use Visual Studio Code or Spyder you can of course run the Python file in the editor. Next, you open the anaconda power shell, use `cd` to move where your Python file is located and write

Terminal

```
(base) PS C:\Users\Aksel Hiorth> python <name_of_file>
```

remember to include the `.py` extension. Below are an example

1.1 Search for files of a specific type

Put the following content in a file called `find_files.py`

```
#%%
#content of find_files1.py
import pathlib as pt
def find_files_of_type(dir='.',extensions=['.xlsx','.txt','.INC']):
    """
    return a list files of type defined in extensions
    """
    p=pt.Path(dir)
    files=[]
    for ext in extensions:
        for x in p.rglob("*"+ext):
            if x.is_file():
                files.append(x.absolute())
    return files
if __name__ == '__main__':
    files=find_files_of_type()
    for f in files:
        print(f)
# %%
```

Then open the anaconda power shell where this file is and run

Terminal

```
(base) PS C:\Users\Aksel Hiorth> python find_files.py
```

The reason we use the statement `if __name__ == '__main__':` is to only run the code if this is the main file. This means that we can import the file `find_files.py` in another program without running the last part, because then `find_files.py` is not the main program.

When running files from command line it would be useful to give some command line arguments. The `sys` module and the `sys.argv` can be used to access command line argument, the `sys.argv` command returns a list of strings, where the first element is always the name of the program. In the code below we have made it possible to also to give in the directory name of where to start the search.

```
#%%
#content of find_files1.py
import pathlib as pt
def find_files_of_type(dir='.',extensions=['.xlsx','.txt','.INC']):
    """
    return a list files of type defined in extensions
    """
    p=pt.Path(dir)
    files=[]
    for ext in extensions:
        for x in p.rglob("*"+ext):
            if x.is_file():
                files.append(x.absolute())
    return files
if __name__ == '__main__':
    files=find_files_of_type()
    for f in files:
        print(f)
# %%
```

To start the search two directories above

Terminal

```
(base) PS C:\Users\Aksel Hiorth> python find_files2.py ../../
```

2 Creating executable programs from Python file

Even if someone has not Python installed on their computer, they could still be interested in your programs. It is actually possible to create executable files from your Python scripts, where all libraries are included in the executable file. Here we show the bare minimum and only the `pyinstaller` library, for more information check out this blog[1].

Install packages: Open anaconda power shell, activate your environment and install `pyinstaler`

Terminal

```
(mod322) PS C:\Users\Aksel Hiorth> conda install pyinstaller
```

Create executable: Next, we must create an executable, run the following command (where your python file is located)

Terminal

```
(mod322) PS C:\Users\Aksel Hiorth> pyinstaller find_files2.py --onefile
```

`pyinstaller` now have created two folder, a `build` folder and a `dist` folder. Inside the `dist` folder there should be a single file called `find_files2.exe`. This file can be shared with other people. You can run this file from command line as before, and it should behave as your python script.

3 Passing functions to functions

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 at multiple points, that is straightforward since we have used the Numpy version of $\sin x$.

[1] https://www.blog.pythonlibrary.org/2021/05/27/pyinstaller-how-to-turn-your-python-code-into-an-exe-on-w

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

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('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, 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):
    '''
    t  : time
    s0 : initial starting point
    v0 : initial velocity
    a  : acceleration
    returns the distance traveled
    '''
    return s0+v0*t+a*t*t*0.5 #multiplication (0.5)is general faster
                     #than division (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 to avoid this error is to use global variable, that is to define `s0, v0`, and `a` at the top of the code. This is not always the best 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 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))
```

4 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 the variable is available, first some simple examples

A variable created inside a function is only available within the function: “

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

Doing `print(a)` outside the function will throw an error: `name 'a' is not defined`. What happens if we define 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.

The `global` keyword can be used to pass and access variables in functions: “

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

In this case `print(a)` *before* calling `f(x)` will give the answer `2` and *after* calling `f(x)` will 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 potentially be modified by several functions.

5 Passing arrays and lists to functions

In the previous section, we looked at some simple examples regarding the scope of variables, and what happened with that variable inside and outside a function. The examples used integer or floats. However in most applications you will pass an array or a list to a function, and then you need to be aware that the behavior is not always would you might expect.

Unexpected behavior.

Sometimes functions do not do what you expect, this might be because the function does not treat the arguments as you might think. The best advice is to make a very simple version of your function and test it for yourself. Is the behavior what you expect? Try to understand why or why not.

Let us look at some examples, and try to understand what is going on and why.

```
x=3
def f(x):
    x = x*2
    return x
print('x =',x)
print('f(x) returns ', f(x))
print('x is now ', x)
```

In the example above we can use `x=3`, `x=[3]`, `x=np.array([3])`, and after execution `x` is unchanged (i.e. same value as before `f(x)`) was called. Based on what we have discussed before, this is maybe what you would expect, but if we now do

```
x=[3]
def append_to_list(x):
```

```
    return x.append(1)
print('x = ',x)
print('append_to_list(x) returns ', append_to_list(x))
print('x is now ', x)
```

(Clearly this function will only work for lists, due to the append command.) After execution, we get the result

```
x = [3]
append_to_list(x) #returns  [3 1], x is now  [3, 1]
```

Even if this might be exactly what you wanted your function to do, why does `x` change here when it is a list and not in the previous case when it is a float? Before we explain this behavior let us rewrite the function to work with Numpy arrays

```
x=np.array([3])
def append_to_np(x):
    return np.append(x,1)
print('x = ',x)
print('append_to_np(x) returns ', append_to_np(x))
print('x is now ', x)
```

The output of this code is

```
x =  np.array([3])
append_to_np(x) #returns  [3 1], x is now  [3]
```

This time `x` was not changed, what is happening here? It is important to understand what is going on because it deals with how Python handles variables in the memory. If `x` contains million of values, it can slow down your program, if we do

```
N=1000000
x=[3]*N
%timeit append_to_list(x)
x=np.array([3]*N)
%timeit append_to_np(x)
```

On my computer I found that `append_to_list` used 76 nano seconds, and `append_to_np` used 512 micro seconds, the Numpy function was about 6000 times slower! To add to the confusion consider the following functions

```
x=np.array([3])
def add_to_np(x):
    x=x+3
    return x

def add_to_np2(x):
    x+=3
```

```
    return x
print('x = ',x)
print('add_to_np(x) returns ', add_to_np(x))
print('x is now ', x)

print('x = ',x)
print('add_to_np2(x) returns ', add_to_np2(x))
print('x is now ', x)
```

The output is

```
x =  np.array([3])
add_to_np(x) #returns [6], x is now [3]
x =  np.array([3])
add_to_np2(x) #returns  [6], x is now  [6]
```

In both cases the function returns what you expect, but it has an unexpected (or at least a different) behavior regarding the variable `x`. What about speed?

```
N=10000000
x=np.array([3]*N)
%timeit add_to_np(x)
x=np.array([3]*N)
%timeit add_to_np2(x)
```

`add_to_np` is about twice as slow as `add_to_np2`. In the next section we will try to explain the difference in behavior.

Avoiding unwanted behavior of functions.

The examples in this section are meant to show you that if you pass an array to a function, the array can be altered outside the scope of the function. If this is not what you want, it could lead to bugs that are hard to detect. Thus, if you experience unwanted behavior pick out the part of function involving list or array operations and test one by one in the editor.

6 Call by value or call by reference

For anyone that has programmed in C or C++ call by reference or value is something one need to think about constantly. When we pass a variable to a function there are two choices, should we pass a copy of the variable or should we pass information about where the variable is stored in memory?

Value and reference.

In C and C++ pass by value means that we are making a copy in the memory of the variable we are sending to the function, and pass by reference means that we are sending the actual parameter or more specific the address to the memory location of the parameter. In Python all variables are passed by object reference.

In C and C++ you always tell in the function definition if the variables are passed by value or reference. Thus if you would like a change in a variable outside the function definition, you pass the variable by reference, otherwise by value. In Python we always pass by (object) reference.

6.1 Floats and integers

To gain a deeper understanding, we can use the `id` function, the `id` function gives the unique id to a variable. In C this would be the actual memory address, lets look at a couple of examples

```
a=10.0
print(id(a)) #gives on my computer 140587667748656
a += 1
print(id(a)) #gives on my computer 140587667748400
```

Thus, after adding 1 to `a`, `a` is assigned *a new place in memory*. This is very different from C or C++, in C or C++ the variable, once it is created, *always has the same memory address*. In Python this is not the case, it works in the opposite way. The statement `a=10.0`, is executed so that *first* 10.0 is created in memory, secondly `x` is assigned the reference to 10.0. The assignment operator `=` indicates that `a` should point to whatever is on the right hand side. Another example is

```
a=10.0
b=10.0
print(a is b) # prints False
b=a
print(a is b ) # prints True
```

In this case 10.0 is created in two different places in the memory and a different reference is assigned to `a` and `b`. However if we put `b=a`, `b` points to the same object as `a` is pointing on. More examples

```
a=10
b=a
print(a is b) # True
a+=2
print(a is b) # False
```

When we add 2 to `a`, we actually add 2 to the value of 10, the number 12 is assigned a new place in memory and `a` will be assigned that object, whereas `b` would still points the old object 10.

6.2 Lists and arrays

Yous should think of lists and arrays as containers (or a box). If we do

```
x=[0,1,2,3,4]
print(id(x))
x[0]=10
print(id(x)) # same id value as before and x=[10,1,2,3,4]
```

First, we create a list, which is basically a box with the numbers 0, 1, 2, 3, 4. The variable `x` points to *the box*, and `x[0]` points to 0, and `x[1]` to 1 etc. Thus if we do `x[0]=10`, that would be the same as picking 0 out of the box and replacing it with 10, but *the box stays the same*. Thus when we do `print(x)`, we print the content of the box. If we do

```
x=[0,1,2,3,4]
y=x
print(x is y) # True
x.append(10)  # x is now [0,1,2,3,4,10]
print(y)      # y=[0,1,2,3,4,10]
print(x is y) # True
```

What happens here is that we create a box with the numbers 0, 1, 2, 3, 4, `x` is referenced that box. Next, we do `y=x` so that `y` is referenced the *same box* as `x`. Then, we add the number 10 to that box, and `x` and `y` still points to the same box.

Numpy arrays behave differently, and that is basically because if we want to add a number to a Numpy array we have to do `x=np.array(x,10)`. Because of the assignment operator `=` , we take the content of the original box add 10 and put it into a *new* box

```
x=np.array([0,1,2,3,4])
y=x
print(x is y)     # True
x=np.append(x,10) # x is now [0,1,2,3,4,10]
print(y)          # y=[0,1,2,3,4]
print(x is y)     # False
```

The reason for this behavior is that the elements in Numpy arrays (contrary to lists) have to be continuous in the memory, and the only way to achieve this is to create a new box that is large enough to also contain the new number. This also explains that if you use the `np.append(x,some_value)` inside a function where `x` is large it could slow down your code, because the program has to delete `x` and create a new very large box each time it would want to add a new element.

A better way to do it is to create `x` *large enough* in the beginning, and then just assign values `x[i]=a`.

7 Mutable and immutable objects

What we have explained in the previous section is related to what is known as mutable and immutable objects. These terms are used to describe objects that have an internal state that can be changed (mutable), and objects that have an internal state that cannot be changed after they have been created. Example of mutable objects are lists, dictionaries, and arrays. Examples of immutable objects are floats, ints, tuples, and strings. Thus if we create the number 10 its value cannot be changed (and why would we do that?). Note that this is *not the same as saying that* `x=10` and that the internal state of `x` cannot change, this is *not* true. We are allowed to make `x` reference another object. If we do `x=10`, then `x is 10` will give true and they will have the same value if we use the `id` operator on `x` and `10`. If we later say that `a=11` then `a is 10` will give false and `id(a)` and `id(10)` give different values, but * `id(10)` will have the same value as before*.

Lists are mutable objects, and once a list is created, we can change the content without changing the reference to that object. That is why the operations `x=[]` and `x.append(1)`, does not change the id of x, and also explain that if we put `y=x`, `y` would change if `x` is changed. Contrary to immutable objects if `x=[]`, and `y=[]` then `x is y` will give false. Thus, whenever you create a list it will be an unique object.

A final tip.

You are bound to get into strange, unwanted behavior when working with lists, arrays and dictionaries (mutable) objects in Python. Whenever, you are unsure, just make a simple version of your lists and perform some of the operations on them to investigate if the behavior is what you want.

Finally, we show some "unexpected" behavior, just to demonstrate that it is easy to do mistakes and one should always test code on simple examples.

```
x_old=[]
x  = [1, 2, 3]
x_old[:] = x[:] # x_old = [1, 2, 3]
x[0] = 10
print(x_old) # "expected" x_old = [10, 2, 3], actual [1, 2, 3]
```

Comment: We put the *content* of the `x` container into `x_old`, but `x` and `x_old` reference different containers.

```
def add_to_list(x,add_to=[])
```

```
    add_to.append(x)
    return add_to

print(add_to_list(1)) # "expected" [1] actual [1]
print(add_to_list(2)) # "expected" [2] actual [1, 2]
print(add_to_list(3)) # "expected" [3] actual [1, 2, 3]
```

Comment: `add_to=[]` is a default argument and it is created once when the program starts and not each time the function is called.

```
x = [10]
y = x
y = y + [1]
print(x, y) # prints [10] [10, 1]

x = [10]
y = x
y += [1]
print(x, y) # prints [10, 1] [10, 1]
```

Comment: In the first case `y + [1]` creates a new object and the assignment operator `=` assign `y` to that object, thus `x` stays the same. In the second case the `+=` adds `[1]` to the `y` container without changing the container, and thus `x` also changes.

References