**Data scientist’s guide to efficient coding in Python**

Tips and tricks I use for writing clean codes on a day-to-day basis

In this article, I wanted to share a few tips for writing cleaner codes that I have absorbed in the last year — mainly from pair programming. Generally speaking, including them as part of my everyday coding routine has helped me generate supreme quality Python scripts, that are easily maintainable and scalable over time.

*Ever thought why****senior developer’s code look so much better in comparison to a junior developer****. Read on to bridge the gap….*

Rather than giving generic examples on how to use these techniques, I will be giving real-life coding scenarios where I have *actually* used them! Here is the [Jupyter Colab Notebook](https://colab.research.google.com/drive/1gSIJd_HY88A_bq-Z0zMMzFYb1hjRI8DO?usp=sharing) if you’d like to follow along!

**1. Use**tqdm**when working with**for **loops.**

Imagine looping over a *large* iterable (list, dictionary, tuple, set), and not knowing whether the code has finished running! *Bummer*, *right*! In such scenarios make sure to use tqdm construct to display a progress bar alongside.

For instance, to display the progress as I read through all the files present in 44 different directories (whose paths I have already stored in a list called fpaths):

from tqdm import tqdmfiles = list()  
fpaths = ["dir1/subdir1", "dir2/subdir3", ......]for fpath in tqdm(fpaths, desc="Looping over fpaths")):  
 files.extend(os.listdir(fpath))

https://miro.medium.com/max/700/1*5YzB1ok6hkRiM7wFcKtpHA.png

Using tqdm with “for“ loop

*Note: Use the desc argument to specify a small description for the loop.*

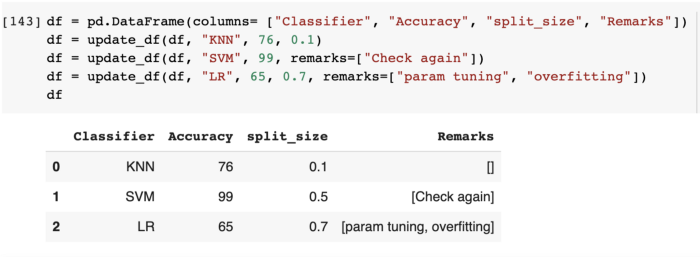
**2. Use**type hinting when writing functions.

In simple terms, it means explicitly stating the type of all the arguments in your Python function definition.

I wish there were specific use cases I could provide to emphasize *when* I use type hinting for my work, but the truth is, I use them more often than not.

Here’s a hypothetical example of a function update\_df(). It updates a given data frame by appending a row containing useful information from a simulation run — such as classifier used, accuracy scored, train-test split size, and additional remarks for that particular run.

def update\_df(**df: pd.DataFrame**,   
 **clf: str**,   
 **acc: float**,  
 **split:float** = 0.5,  
 **remarks: List[str] = []**  
 ) -> **pd.DataFrame**: new\_row = {'Classifier':clf,   
 'Accuracy':acc,   
 'split\_size':split,  
 'Remarks':remarks} df = df.append(new\_row, ignore\_index=True)  
 return df



Few things to note:

* The datatype following the -> symbol in the function definition (def update\_df(.......) **->** pd.DataFrame) indicates the type of the value returned by the functions, i.e. a Pandas’s dataframe in this case.
* The default value, if any, can be specified as usual in the form param:type = value . (For example: split: float = 0.5)
* In case a function does not return anything, feel free to use None. For example : def func(a: str, b: int) -> None: print(a,b)
* To return values of mixed types, for example, say a function could either print a statement if a flag optionwas set OR return an intif the flag was not set:

**from typing import Union**  
*def* dummy\_args(\**args*: list[int], *option* = True) -> **Union[None, int]**: if *option*: print(*args*) else: return 10

*Note: As of Python 3.10, Union is not required, so you can simply do:*

*def* dummy\_args(\**args*: list[int], *option* = True) -> **None | int**: if *option*: print(*args*) else: return 10

* You can go as specific as you’d like while defining the types of parameters, as we have done for remarks: List[str]. Not only do we specify it should be a List, but it should be a list of str only.  
  For fun, try passing a list of integers to remarkswhile calling the function. You’ll see no error returned! *Why is that?*Because Python interpreter doesn’t enforce any type checking based on your type-hints.

T[ype-hints have no impact on your code other than adding documentation](https://www.pythonlikeyoumeanit.com/Module5_OddsAndEnds/Writing_Good_Code.html#What-is-It-Good-For?-(Absolutely-Nothing)).

It’s still good practice, though, to include it! I feel it lends more clarity to oneself when writing a function. In addition, when someone makes a call to such a function, they get to see nice prompts for the arguments it would take as inputs.



Prompts when calling a function with type hinting

**3. Use args and kwargs for functions with unknown # of arguments**.

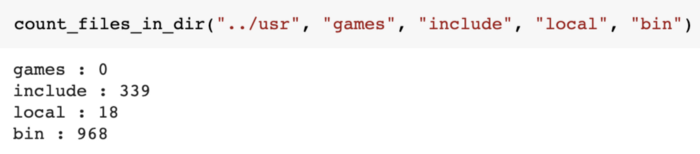
Imagine this: you want to write a function that takes as input *some* directory paths and prints the number of files within each. The problem is, we do not know *how* many paths the user would input! Could be 2 could be 20! So we are unsure how many parameters should we define in our function definition. Clearly, writing a function like def count\_files(file1, file2, file3, …..file20)would be silly. In such cases, args and (sometimes kwargs) come handy!

***Args****is used for specifying an unknown number of****positional****arguments.****Kwargs****is used for specifying an unknown number of****keyword****arguments.*

**Args**

Here’s an example of a function count\_files\_in\_dir()that takes in project\_root\_dir and an arbitrary number of folder paths within it (using \*fpaths in the function definition). As an output, it prints the number of files within each of these folders.

def count\_files\_in\_dir(project\_root\_dir, \*fpaths: str): for path in fpaths: rel\_path = os.path.join(project\_root\_dir, path)  
 print(path, ":", len(os.listdir(rel\_path)))



Counting the # of files in Google Colab directories

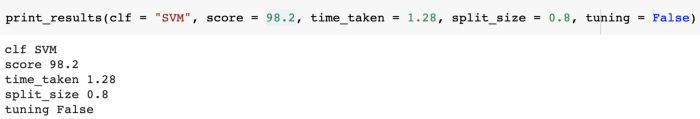
In the function call, we pass in 5 arguments. As the function definition expects one *required* positional arguments i.e. project\_root\_dir, it automatically knows "../usr" must be it. All the remaining arguments (four in this case) are “soaked up” by \*fpathsand are used for counting the files.

*Note: The proper terminology for this soaking up technique is “argument packing” i.e. remaining arguments are packed into \*fpaths.*

**Kwargs**

Let’s look at functions that must take an unknown number of *keyword* arguments. In such scenarios, we must use kwargs instead of args. Here’s a short (useless) example:

def print\_results(\*\*results): for key, val in results.items():  
 print(key, val)



The usage is quite similar to \*args but now we are able to pass an arbitrary number of *keyword* arguments to a function. These arguments get stored as key-value pairs in \*\*resultsdictionary. From hereon, it is easy to access the items within this dictionary using .items().

I have found two main applications for kwargs in my work:

* merging dictionaries (*useful but less interesting*)

dict1 = {'a':2 , 'b': 20}  
dict2 = {'c':15 , 'd': 40}merged\_dict = {\*\*dict1, \*\*dict2}\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
{'a': 2, 'b': 20, 'c': 15, 'd': 40}

* extending an existing method (*more interesting*)

def myfunc(a, b, flag, \*\*kwargs): if flag:  
 a, b = do\_some\_computation(a,b)  
   
 actual\_function(a,b, \*\*kwargs)

*Note: Checkout*[*matplotlib’s plot function using kwargs*](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib-pyplot-plot)*to specify optional embellishments for a plot such as linewidth and label.*

Here’s a real practical use-case for extending methods using \*\*kwargs from one of my recent projects:  
We often use Sklearn’s train\_test\_split() for splitting X and y. While working on one of the GANs projects, I had to split the generated synthetic images into the *same* train-test split that is used for splitting real images and their respective labels. In addition, I also wanted to be able to pass any other arguments that one would normally pass to the train\_test\_split(). Finally, stratify must always be passed because I was working on a face recognition problem (and wanted all labels to be present in both train and test set).

To achieve this, we created a function called custom\_train\_test\_split(). I’ve included a bunch of print statements to show what exactly is happening under the hood (and omitted some snippets for brevity).

def custom\_train\_test\_split(clf, y, \*X, stratify, \*\*split\_args): *print("Classifier used: ", classifier)  
 print("Keys:", split\_args.keys())  
 print("Values: ", split\_args.values())  
 print(X)  
 print(y)  
 print("Length of passed keyword arguments: ", len(split\_args))* trainx,testx,\*synthetic,trainy,testy = train\_test\_split(  
 \*X,  
 y,  
 stratify=stratify,  
 \*\*split\_args  
 ) *######### OMITTED CODE SNIPPET #############  
 # Train classifier on train and synthetic ims  
 # Calculate accuracy on testx, testy  
 ############################################* *print("trainx: ", trainx, "trainy: ",trainy, '\n', "testx: ",   
 testx, "testy:", testy) print("synthetic: ", \*synthetic)*

*Note: While calling this function, instead of using the actual image vectors and labels (see figure below), I have replaced them with dummy data for ease of understanding. The code, however, should work for real images as well!*

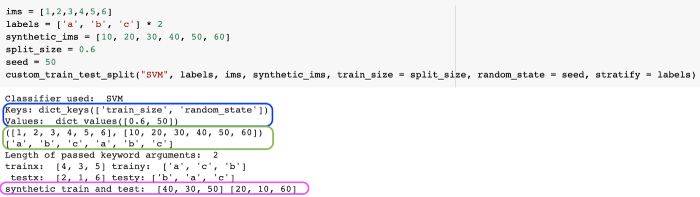


Figure A Calling a function with kwargs in function definition

Bunch of things to note:

* All the *keyword* arguments (**except**stratify), used in the function call statement (such as train\_size and random\_state) will be stored as a key-value pair in the \*\*split\_args dictionary. (To verify, check out the output in Blue.)  
  Why not stratify, you might ask? This is because according to the function definition, it is a *required*keyword-onlyargument and not an *optional* one.
* All the *non-keyword*(i.e. positional)arguments passed in the function call (such as "SVM", labels, etc.) would be stored in the first three parameters in function definition i.e.clf, y and \*X, (and yes the order in which they are passed matters). However, we have *four* arguments in function call i.e. "SVM", labels, ims , and synthetic\_ims. *Where do we store the fourth one?*  
  Remember that we used \*X as the third parameter in the function definition, hence all arguments passed to the function after the first two arguments are *packed*(soaked) into \*X. (To verify, check output in Green).
* When calling train\_test\_split() method within our function, we are essentially *unpacking* the X and split\_args arguments using the \* operator,(\*X and \*\*split\_args), so that all elements of it can be passed as different parameters.

That is,