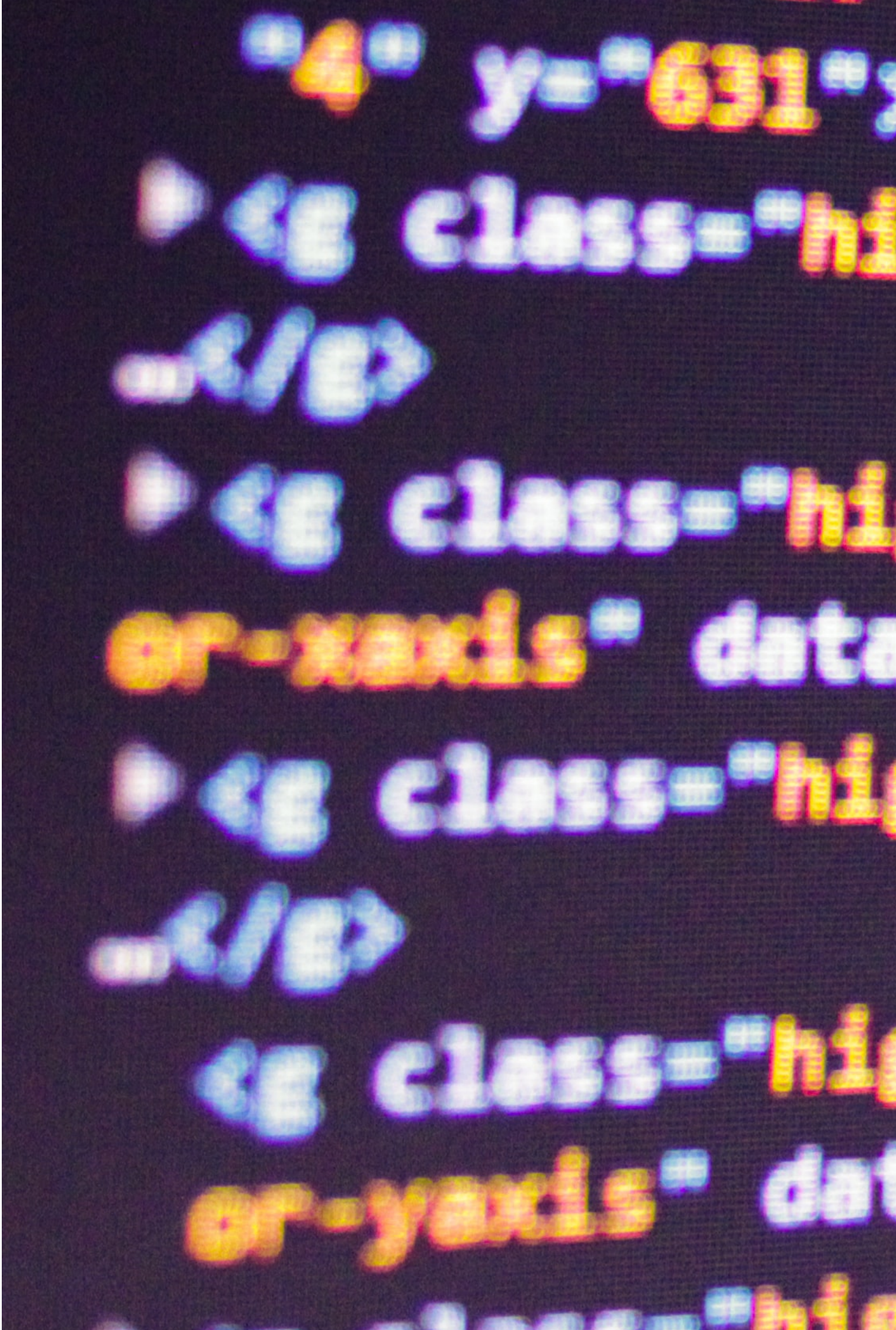
5 ways for Data Scientists to Code Efficiently in Python

BEGINNER PROGRAMMING PYTHON

This article was published as a part of the Data Science Blogathon

Introduction

Writing a code is like a piece of art and in this article, I am going to share some tips and tricks for Data Scientists and Python enthusiasts who want to write better and cleaner codes. I personally also use these coding techniques in my life and they have improved my style a lot. So I want to share them with everyone so that everyone can learn about them. You will learn how to write some epic quality Python scripts which are easily scalable and also easy to maintain.



Link of image: https://unsplash.com/photos/Q9-QEy1_jYI

You might have often wondered why your code does not look like the senior developer's code? Well, there might be various reasons for that and this article will help you bridge the gap between you two. Let me be clear, what I will not do in this article is to give you some generic examples of using these techniques but I will try to give you real-life coding scenarios where you will be using these techniques. So let's begin the process.

1. Try using tqdm when your code has for loops

Looping over a small list might not be much but imagine when you have to loop over a large dataset. You will not be able to know if your code has finished or not, which is unfair right? Hence to save you the doubt of whether there is an infinite loop or just time taking the task, use tqdm to display a progress bar alongside your code.

Code :

```
for i in tqdm(range(100000000),desc="Looping through the loop"): pass
```

Output :

The above code will produce a progress bar along with the execution to show you the loop execution and other details like the speed of iteration per second and also the total time is taken to complete the loop. The “DESC” parameter allows us to pass a suitable description for the loop to be displayed.

2. Use Type hinting when writing any function

Type hinting means that explicitly state all the arguments of a python function. These are very helpful for my day-to-day work and I almost always use them. Did you know that you can specify return types in the function definition of a python function? Well, we do not use it much but it exists and can be considered as a good coding standard. The following code demonstrates it :

Code :

```
def update_df(df: pd.DataFrame, classifier: str, accuracy: float, split_size:float = 0.5, remarks: List[str] = []) -> pd.DataFrame: new_row = {'Classifier':classifier, 'Accuracy':accuracy, 'split_size':split_size, 'Remarks': remarks} df = df.append(new_row, ignore_index=True) return df df = pd.DataFrame(columns= ["Classifier", "Accuracy", "split_size", "Remarks"]) df = update_df(df, "KNN", 88, 0.1) df = update_df(df, "SVM", 90, remarks=["Check again"]) df = update_df(df, "LR", 80, 0.8, remarks=["param tuning", "overfitting"]) df
```

Output :

The above function returns a dataframe where we can append a new row to an existing Dataframe. Here the code might seem a bit weird but you should consider the following points about the above code

- The “->” symbol after the function defines the type of value which the function will return. In this particular case, the function returns a pandas Dataframe. This is also known as the return type in other programming languages
- The Default value for any function can be defined in the format “ parameter: type = value “, for example in the above function, the “split: float = 0.5 “ sets a default value of 0.5 to the split variable of Datatype Float.
- The default function return type in python, unlike void for other languages, is “None”.
- If you want to have a mixed type of return, for example, depending upon the work, a function can either print something or return a value. The function can also be either an integer or a float value. For such cases, you can use the “Union” functionality of python like shown below :
    - For normal python function, this works for all python versions and is deprecated from Python 3.10

```
def func(val : int = 0) - > Union[ None , int ]:
```

- In python 3.10 and above, you do not need the union keyword and can directly use an “OR” operator. Example :

```
def func (val: int = 0) - > None | int
```

- You can go as specific as you what while type hinting in a function definition. For example, in our case for “Remarks” we specify that it will be a list and the list will be of “str” or string format only. Here you should remember one thing that python does not provide an inbuilt checking for type error in type hints and passing a list of integers will not throw an error. At least not in 3.10 and below versions as of now. Hence you need to write a manual error checking code for it.

3. Unknown no of arguments require kwards and Argos to tackle it

You can imagine a scenario where the user can pass an undisclosed or uncertain number of parameters or you want to input directory paths which can either be 1 or 50 different paths through the user input. Clearly writing “ def func(para1,para2…) “ might not be the best case of action and in that case, you can use “args” and “kwargs”. You should remember the following purpose of both the keywords.

- Args is used to specify an unknown number of positional arguments

- Kwargs is used to specify an unknown number of keyword arguments

Args:

The below function takes the root directory of a path and then takes an arbitrary number of folder paths using “ *fpaths ” as the input and then prints the number of files within each of the folders.

Code :

```
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))) count_files_in_dir("../usr", "games", "include", "local", "bin")
```

Output :

Note: Here we are counting the number of files in the Google Collab directory

In the function call, we are actually passing 5 arguments and the function definition is expecting one mandatory positional argument and the remaining ones are “soaked up” with “ *fpaths ” and then we can count the number of files. The technical term for soaking up is actually “argument packing”.

Kwargs :

Now we will look at functions that take an unknown number of arguments and in such scenarios we must use kwargs instead of Argos. The following code will explain the situation better.

Code :

```
def print_results(**results): for key, val in results.items(): print(key, val) print_results(clf = "SVM", score = 98.2, time_taken = 1.28, split_size = 0.8, tuning = False)
```

Output :

The use does look similar to *args but now we are able to pass an arbitrary number of keywords and their arguments to any function. These values will by default get stored as key and value pairs which can be accessed using the .items() function of the dictionary. You can effectively use it for 2 different purposes which I have found, if you find more, feel free to reach out to me via the contact details provided at the end of the article. The two useful ways are :

- Merging dictionaries
- Extending the existing method with more functionalities

Code for Merging Dictionaries :

```
dict1 = {'a':2 , 'b': 20} dict2 = {'c':15 , 'd': 40} merged_dict = {**dict1, **dict2} Merged_dict
```

Output :

Code for Extending methods :

```
def custom_train_test_split(classifier: str, y: List, *X: List, 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)) # *train_all, labels_all, size, seed = split_args.values() trainx, testx, *synthetic, trainy, testy = train_test_split(*X, y, stratify = stratify, **split_args) print("trainx: ", trainx, "trainy: ",trainy, 'n', "testx: ", testx, "testy:", testy) print("synthetic train and test: ", *synthetic) ims = [1,2,3,4,5,6] labels = ['a', 'b', 'c'] * 2 synthetic_ims = [10, 20, 30, 40, 50, 60] split_size = 0.6 seed = 50 custom_train_test_split("SVM", labels, ims, synthetic_ims, train_size = split_size, random_state = seed, stratify = labels)
```

Output :

I often use Sklearn's train_test_split() for splitting the x and y and while working on one of my GAN projects, I had to split the synthetic pictures into the same test and train split which is used to split real-life images and the labels. I also wanted to pass extra arguments to the split function. Stratify was also used as I was working with face recognition. So the custom train and test split function was coded for this purpose. I have replaced the image vectors and labels with dummy data but it will work with real-life images as well.

4. Pre-Commit Hooks

The Code we often write can lack proper formatting or be messy at times. Trailing whitespaces, extra comma, unsorted import statements and the spaces in indentation, and more can be a nuisance to solve. You can definitely solve this all manually but you can save yourself a lot of time if you use a pre-commit hook. These hooks can do auto-formatting with just one line of code which is: " pre-commit run "

You need to create your own " pre-commit-config.yaml " file which will contain all information regarding the formatting rules you want to follow. You should stage the file with git add before you do a pre-commit run or else all files will be skipped.

5. Useful VS-Code Extensions

Hands down, I have been using VS-Code for almost everything and it has been very helpful but with these extensions by your side, you can amplify your productivity and I personally also feel these extensions to be very useful. They are as follows :

- Bracket Pair Colorizer – It pairs matching brackets with the same color
- Path Intellisense – It allows the auto-complete of filenames
- Python Dockstring. generator – Allows generation of docstring for python
- Python Indent – It allows the proper indentation of code that runs on multiple lines
- Python Type Hint – It allows for auto-completion for the type hints while writing functions
- TODO Tree – Keeps track of all the TODO's in one place which was written as comments and has the word TODO in it
- Pylance – Helps with your code completion and parameter suggestions

Bonus Tip :

Docstring of any function can be accessed with the "function_name.__doc__ " format. The following code shows this with an example :

Code :

```
def func(): '''This is a sample function for docstring''' pass func.__doc__
```

Output :

Endnotes:

So these few tips and tricks will help you become a better Data Scientist and help you write better and more efficient code with less effort which will maximize your productivity. I hope all tips were something

that was valuable and feel free to reach out to me to share your tips for a future part 2 of the article. Stay Safe and get vaccinated everyone.

Arnab Mondal

Data Engineer | Python Developer

https://www.linkedin.com/in/arnab1408/

Collab Notebook Link :

https://colab.research.google.com/drive/1gSIJd_HY88A_bq-Z0zMMzFYb1hjRI8DO?usp=sharing

---

Article Url - https://www.analyticsvidhya.com/blog/2021/08/5-ways-for-data-scientists-to-code-efficiently-in-python/

Arnab1408