

Style Guidelines

Styleguidelines

Ensuring a clean style throughout your code and notebooks is good practice.

Also, it helps you to keep track of what you did already and it helps us in correcting your assignments - especially when tasks get a bit more complex.

Therefore, we would like you to adhere to the following style guidelines.

If certain criteria are repeatedly violated, we will deduct overall-points from an assignment, even though, solutions might be correct.

Comments and Markdown Cells

Comments

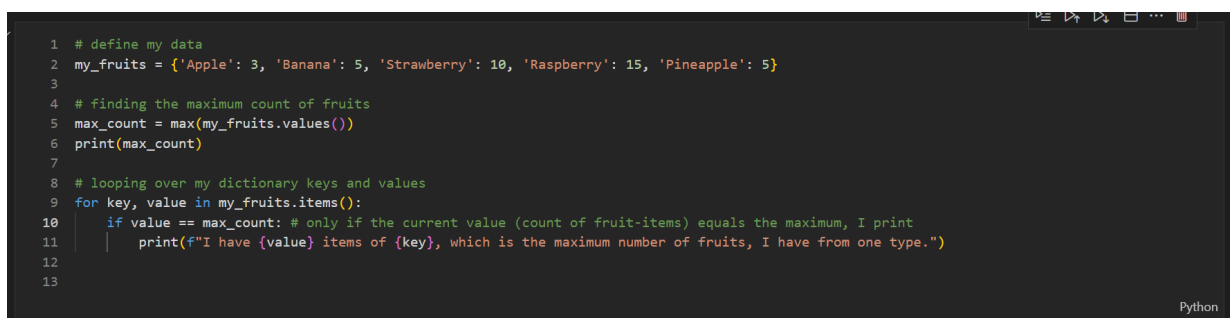
In general, we want you to write comments for your code. These comments should be **self-written** and formulated in your **own words**. This will help you to check if you know

1. **what** you code is computing and *at which location* (→ also helps in finding mistakes)
- and it helps us to **understand your conceptual approach** to a task (which enables us to allocate partial credits for the correct approach but maybe a erroneous code)

Comments can be added in code cells and always start with a "#"

→ input image of example comments on a loop with multiple steps

BUT of course, you do not need to comment every variable assignment or trivial function calls.



```
1 # define my data
2 my_fruits = {'Apple': 3, 'Banana': 5, 'Strawberry': 10, 'Raspberry': 15, 'Pineapple': 5}
3
4 # finding the maximum count of fruits
5 max_count = max(my_fruits.values())
6 print(max_count)
7
8 # looping over my dictionary keys and values
9 for key, value in my_fruits.items():
10     if value == max_count: # only if the current value (count of fruit-items) equals the maximum, I print
11         print(f"I have {value} items of {key}, which is the maximum number of fruits, I have from one type.")
12
13
```

Fig. 1: Comments.png

Interpretations

For interpretations of results, we would like you to **use markdown cells**, which are made for incorporating text components into notebooks.

Depending on the application (IDE) you are using, there are different ways of adding a markdown cell:

1. Jupyter Notebook

In Jupyter Notebook, you can find a small "+" sign in the top menu header. Clicking this inputs a new cell below the current cell, your cursor is on.

By default, the cell will be a code cell. To change the type to markdown, put the cursor into the new cell and see the menu header on the right. There you will find a drop-down menu, which allows to change the type to 'Markdown' (and back).

2. VS Code

In VS Code, hovering over the space between two code cells, or below the last cell will display two options: add a code-cell or add a markdown-cell

Using new Code Chunks

For a clear structure, we would like you to use **new code cells for intermediate results** (to show them) and for **every new subtask**.

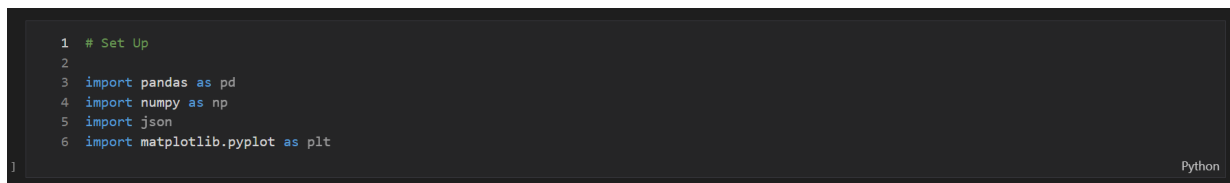
Especially in more complex or longer tasks this will help you in structuring and checking intermediate results of your step-by-step approach to a task.

Be aware, that Python does **not forget** variables, that have been defined before or packages, that have been imported beforehand. Therefore **no need to redefine/reimport** variables, data or packages!

Import of Packages

We will use multiple and different packages within an assignment. It is good practice, to have **one** set up code chunk in the beginning, which **contains all package imports**, that will be required in the current notebook.

Collecting all package imports in one cell also ensures, you are not importing a package multiple times (which is not needed and bad style).



```
1 # Set Up
2
3 import pandas as pd
4 import numpy as np
5 import json
6 import matplotlib.pyplot as plt
```

Fig. 2: Package_imports.png

Comprehensive Plots

When creating a visualization, different aspects need to be fulfilled in order to make the plot comprehensive.

Ideally, a plot should be designed in a way, that it can be understood solely by the information provided in the plot, without additional information.

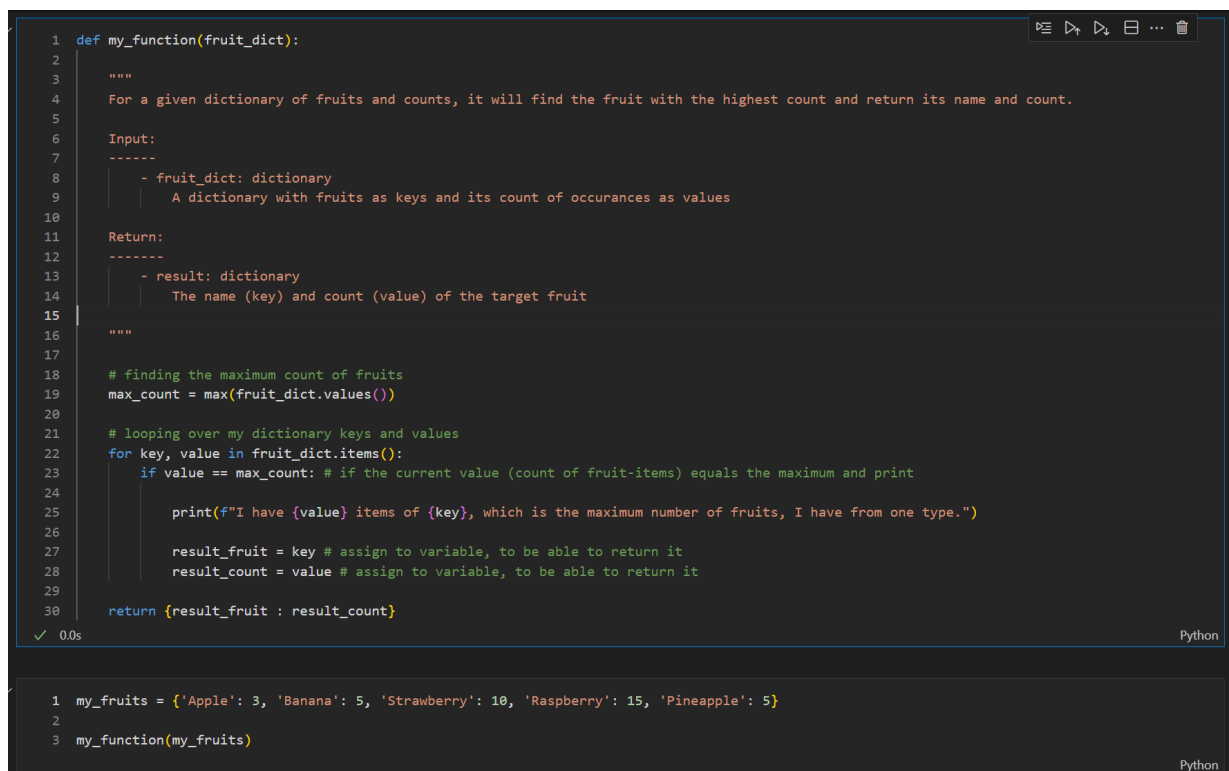
Therefore, a plot

- has a **meaningful header**, containing the overall information (i.e. what is compared against what) and for example filters, that have been applied (i.e. data only covers the year 2017)
- **all** axis should have a label to indicate, which variable is displayed here. Additionally, **indicate the scale** which is used to measure the variable (i.e. inhabitants in 100.000 or vote share in percentage) as well as potential **transformations** which have been applied (i.e. log-transformed values)
- if you use different color, line types or marks to indicate categorical variables, make sure to include a **legend**

Defining Functions and writing Docstrings

For self-defined functions, we expect you to write so called docstrings. These docstrings are meant to explain the *inputs* to a function, *what the functions is computing* and the *results* that will be returned. Such docstrings are super helpful, also for yourself, to remember what a function, you've coded, actually computes.

- Docstrings are added right below the `def my_own_function():` call and are surrounded by three quotation marks.
- They start with a short description, of what the function does/what it can be used for
- Afterwards, the required (and optional) input-parameters are listed, along with the data type they have to be
- Finally, we explain what the result contains and in which data type.



```

1 def my_function(fruit_dict):
2
3     """
4     For a given dictionary of fruits and counts, it will find the fruit with the highest count and return its name and count.
5
6     Input:
7     -----
8     - fruit_dict: dictionary
9       A dictionary with fruits as keys and its count of occurrences as values
10
11     Return:
12     -----
13     - result: dictionary
14       The name (key) and count (value) of the target fruit
15
16     """
17
18     # finding the maximum count of fruits
19     max_count = max(fruit_dict.values())
20
21     # looping over my dictionary keys and values
22     for key, value in fruit_dict.items():
23         if value == max_count: # if the current value (count of fruit-items) equals the maximum and print
24
25             print(f"I have {value} items of {key}, which is the maximum number of fruits, I have from one type.")
26
27             result_fruit = key # assign to variable, to be able to return it
28             result_count = value # assign to variable, to be able to return it
29
30     return {result_fruit : result_count}

```

✓ 0.0s Python

```

1 my_fruits = {'Apple': 3, 'Banana': 5, 'Strawberry': 10, 'Raspberry': 15, 'Pineapple': 5}
2
3 my_function(my_fruits)

```

Python

Fig. 3: Docstring_1.png

→ sticking to this exact syntax enables i.e. VS Code to later display the docstring, when hovering over the functions name, like below:

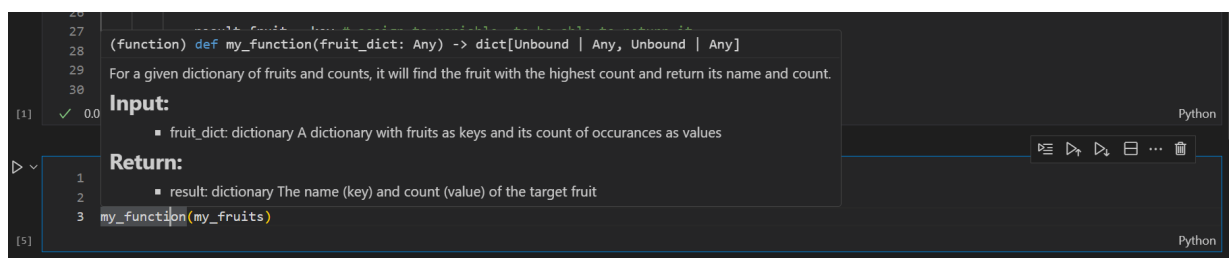


Fig. 4: Docstring_2.png