Clean Code in Python

Sergio Bugallo

Dec 10, 2019

CONTENTS

CHAPTER

ONE

DOCSTRINGS AND ANNOTATIONS

1.1 1. Docstrings

Docstrings are basically documentation embedded in the source code. A **docstring** is basically a literal string, placed somewhere in the code, with the intention of documenting that part of the logic. This information it's meant to represent explanation, not justification.

Having comments in the code is a bad practice for multiple reasons. First, they represent our failure to express our ideas in the code. Second, it can be misleading. Worst than having to spend some time reading a complicated section is to read a comment on how it is supposed to work and figuring out that the code actually does something different.

Sometimes, we cannot avoid having comments (maybe there is an error on a third-party library). In those cases, placing a small but descriptive comment might be acceptable.

The reason why docstrings are a good thing to have in the code is that Python is dynamically typed. Python will not enforce, nor check, anything like the value for any function's input parameters. Documenting the expected input and output of a function is a good practice that will help the readers of that function understand how it is supposed to work.

```
Add some example
```

This information is crucial for someone that hats to learn and understand how a new code works, and how they can take advantage of it.

The docstring is not something separated or isolated from the code. It becomes part of the code, and you can access it. When an object has a docstring defined, this becomes part of it via its `__doc__` attribute:

```
def sample():
    """Sample docstring"""
    return

>>> sample.__doc__
'Sample docstring'
```

There is, unfortunately, one downside to docstrings, and it is that, as it happens with all documentation, it requires manual and constant maintenance. As the code changes, it will have to be updated. Another problem is that for docstrings to be really useful, they have to be detailed, which requires multiple lines.

1.2 2. Annotations

The basic idea is to hint to the readres of the code about what to expect as values of arguments in functions. Annotations enable type hinting.

Annotations let you specify the expected type of some variables that have been defined. It is actually not only about the types, but any kind of metadata that can help you get a better idea of what that variable actually represents.

```
class Point:
    def __init__(self, lat, lon):
        self.lat = lat
        self.lon = lon

def locate (latitude: float, longitude: float) -> Point:
    """..."""
    ...
```

Here, we use `float` to indicate the expected types of input parameters. This is merely informative for the reader, Python will not check these types nor enforce them. We can also specify the expected type of the returned value of the function. In this case, `Point` is a user-defined class, so it will mean that whatever is returned will be an instance of `Point`.

With the introduction of annotations, a new special attribute is also included, and it is `__annotations__`. This will give us access to a dictionary that maps the n ame of the annotations with their corresponding values, which are those we have defined for them:

```
>>> locate.__annotations__
{'latitude': float, 'longitude': float, 'return': __main__.Point}
```

The idea of type hinting is to have extra tools to check and assess the correct use of types throughout the code and to hint to the user in case any incompatibilities are detected.

Starting with Python 3.5, the new typing module was introduced, and this significantly improved how hwe define the types and the annotations in our Python code. The basic idea is that now the semantics extend to more meaningful concepts. For example, you could have a function that worked with lists of tuples in one of its parameters, and you would have put one of these two types as the annotation, or even a string explaining it. But with this module, it is possible to tell Python that it expects an iterable or a sequence. You can even identify the type or the values on it.

There is one extra improvement made in regards to annotations starting from Python 3.6. It is possible to annotate variables directly, not just function parameters and return types. The idea is that you can declare the types of some variables defined without necessarily assigning a value to them:

```
class Point:
    lat: float
    lon: float

>>> Point.__annotations__
{'lat': <class 'float'>, 'lon': <class 'float'>}
```

1.2.1 2.1. Do annotations replace docstrings?

The short answer is no, and this is because they complement each other. It is true that a part of the information previously contained on the docstring can now be moved to the annotations. But this should only leave more room for a better documentation on the docstring. In particular, for dynamic and nested data types, it is always a good idea to provide examples of the expected data so that we can get a better idea of what we are dealing with.

```
def data_from_response(response: dict) -> dict:
    """
    If the response is OK, return its payload.

    Arguments
    ---------
    response: A dict like::
        {
            "status": 200, # <int>
            "timestamp": "...", # <date time>
            "payload": {...} # <dict>
```

(continues on next page)

(continued from previous page)

```
        }

    Returns
    -------
    result: A dict like::
        {"data": {...}}

    Raises
    ------
    ValueError: if the HTTP status is not 200.
    """
    if response["status"] != 200:
        raise ValueError

    return {"data": response["payload"]}
```

Now, we have a complete idea of what is expected to be received and returned by this function. The documentation serves as valuable input, not only for understanding and getting an idea of what is being passed around, but also as a valuable source for unit tests. We can derive data like this to use as input, and we know what would be the correct and incorrect values to use on the tests.

The benefit is that now we know what the possible values of the keys are, as well as their types, and we have a more concrete interpretation of what the data looks like. The cost is that, as we mentioned earlier, it takes up a lot of lines and it needs to be verbose and detailed to be effective.