Theory: Default arguments

26 minutes 6 / 10 problems solved

Start practicing

2211 users solved this topic. Latest completion was about 1 hour ago.

In addition to several ways that you can use to pass arguments into functions, Python also has special syntax for accepting these values from a function call. So, while in earlier topics we have learned how to work with arguments, now we will focus on parameters, the ones with **default values** in particular, and look into them in more detail.

Current topic:

✓ Default arguments

Topic depends on:

✓ Operations with list Stage 3 6★

✓ Objects in Python

Topic is required for:

✓ Args

Table of contents:

§1. Defaults

In Python, functions can have parameters with **default values**. Default parameters are specified in the function definition and contain default values for arguments in case they are not provided with a function call. Have a look at this code:

```
def locate(place, planet="Earth"):
    print(place, "on", planet)


locate("Berlin")                       # Berlin on Earth
locate("Breakfast", planet="Pluto")  # Breakfast on Pluto
locate("Craters", "Mercury")         # Craters on Mercury
```

Here we have two parameters, `place` and `planet`. The first one has no default value, so we should always specify it when calling the function. The second one, in contrast, can be omitted, in which case the function will simply take the default value.

Parameters with default values, such as `planet`, are optional in some way. You can easily call a function without them and rely on preset values. As in the example above, most of the places we might want to find are most likely on Earth. However, new values can be assigned to them either by name or by position.

When you declare this function, place non-default parameters first and then those with default values. If you try doing the opposite, `SyntaxError` will crop up:

```
def greet(greeting="Hello,", name):
    print(greeting, name)

# SyntaxError: non-default argument follows default argument
```

In this case, you will not be able to use the default value at all. As the second parameter still requires a value, we would always have to write both values in a call, which makes not much sense. So, when you declare a function, pay attention to the order of the parameters.

§2. Mutable objects as defaults

When it comes to mutable objects, things are getting trickier. Let's set a list as a default value and see how it works:

```
def add_player(player, team=[]):
    ...
    team.append(player)
    return team
```

As you can see, the function simply adds a new player to a team. First, we will give both arguments to it:

```
ice_hockey_team = add_player("Chris", ["Robert", "Alice"])
print(ice_hockey_team)    # ['Robert', 'Alice', 'Chris']
```

Everything looks fine. However, when we call it relying on the default value, the function's behavior would differ from what you might have expected:

```
rugby_team = add_player("Robin")
print(rugby_team)     # ['Robin']

football_team = add_player("Andrew")
print(football_team)  # ['Robin', 'Andrew']
print(rugby_team)     # ['Robin', 'Andrew']
```

Instead of two separate lists for teams, surprisingly, you got just one. With every subsequent call, the function will append a new item to this list. Why so? It turns out that default parameter values are evaluated *once*.

After you have declared a function, a new object for a default value is created. It will be used from this point on. This means that if the function modifies this object in some way, the default value in the mutable will change too. For this reason, you should use mutable default values carefully.

Here is a common workaround to fix the function from our earlier example:

```
def add_player(player, team=None):
    if team is None:
        team = []
    team.append(player)
    return team
```

Setting the default value to `None` and explicitly reassigning the value of the `team` parameter allows you to create a new list each time this function is called.

§3. PEP time

Look at the declared functions shown in this topic one more time, for example, `def locate(place, planet="Earth"): ...`. Have you noticed missing spaces around the equals sign? Their absence is not accidental. By PEP 8 convention, you should not put spaces around `=` when indicating a keyword argument. This holds true for parameters with default values.

§4. Recap

Let's go over the main points we have discussed:

- Python Functions can be quite flexible, you can use them passing fewer arguments in a call (thanks to **default values**).
- You should pay close attention to the **order** of parameters when you declare functions. Place non-default parameters first and those with default values afterward.
- **Mutable defaults** may work contrary to your intentions, as their values are created once at the runtime. If so, a common way to avoid trouble is by using `None` by default and changing the parameter's value in the function's body.

Report a typo

171 users liked this theory. 2 didn't like it. **What about you?**

Start practicing

Comments (3) Hints (0) Useful links (0) Show discussion