# Theory: Debugging in shell

10 minutes 0 / 5 problems solved Skip this topic Start practicing

1259 users solved this topic. Latest completion was about 8 hours ago.

You've already learned how to start the Python shell and experiment there with modules and functions. But the thing is, the more code you write, the higher chances to get a **bug**. A bug is an unexpected error in your code that is usually hard to find. The process of finding and fixing bugs is called **debugging**, and it's another thing the Python shell can be used for.

IDE (Integrated Development Environment) is thought to be better for debugging, and we'll study how to use JetBrains PyCharm for this purpose in another topic. However, IDLE also provides useful tools for debugging, so let's see how it can be done.

## §1. Debugging

There may be different reasons to check your program: e.g. you get an exception in your code and don't understand where it comes from, or the code doesn't work as intended. In such cases it's a good idea to use the shell to find out what's going wrong.

Let's see how the debugger works in action. Say, we have written a function to generate passwords:

```
import random
import string


def password_generator(length):
    chars = string.ascii_letters + string.digits + string.punctuation
    password = ''
    for _ in range(length):
        password += random.choice(chars)

    return password
```

It works fine but let's improve it a bit so that it doesn't use confusing characters for a password, such as “0” and “O” or “1” and “l”:

```
def password_generator(length):
    chars = string.ascii_letters + string.digits + string.punctuation
    confusing_chars = ('O', '0', '1', 'l')
    password = ''
    while not len(password) == length:
        char = random.choice(chars)
        if char in confusing_chars:
            password += char
    return password
```

After these changes our code doesn't work correctly: every password we get contains only those characters.

```
>>> password_generator(6)
'11l00O'
```

Using IDLE's debugger will help us find out what's happening inside the function.

## §2. Theory

First, import the modules *random* and *string*, then copy the function to IDLE and press *Enter* to see the prompt >>> again. To initialize the debugging mode, click "*Debug*" --> "*Debugger*" from the menu above. You'll see “*DEBUG ON*” in IDLE, and a new window will appear. Check all checkboxes so that we can investigate all options the IDLE debugger provides.

Current topic:

Debugging in shell Stage 2

Topic depends on:

✗ Bugs and debugging Stage 2

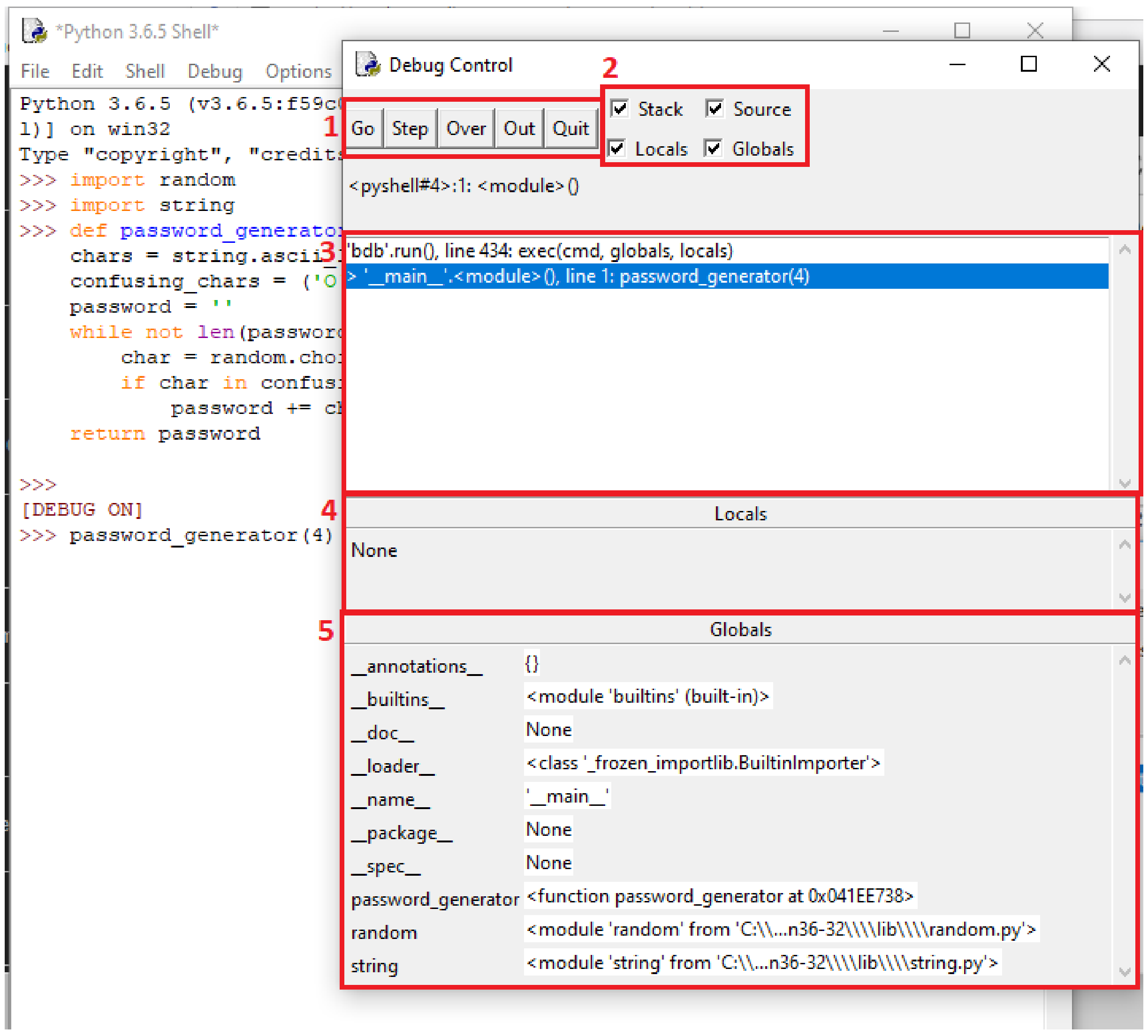
✗ Experiments with Python shell Stage 2

✓ Errors 3★ Stage 1

Table of contents:

To start the debugging, just call the function from the shell, i.e. type `password_generator(4)` and press *Enter*. The debugger window will change: it hasn't executed any code yet but we can see the line of code it has paused at, as well as global variables.

Now, a few words about what can be found in this window.



```
Python 3.6.5 (v3.6.5:f59c
1)] on win32
Type "copyright", "credit:
>>> import random
>>> import string
>>> def password_generato
    chars = string.ascii
    confusing_chars = ('O
    password = ''
    while not len(passwor
        char = random.cho
        if char in confus
            password += c
    return password

>>>
[DEBUG ON]
>>> password_generator(4)
```

```
<pyshell#4>:1: <module>()
```

```
'bdb'.run(), line 434: exec(cmd, globals, locals)
> '__main__'.<module>(), line 1: password_generator(4)
```

```
None
```

| | |
|---|---|
| __annotations__ | {} |
| __builtins__ | <module 'builtins' (built-in)> |
| __doc__ | None |
| __loader__ | <class '_frozen_importlib.BuiltinImporter'> |
| __name__ | '__main__' |
| __package__ | None |
| __spec__ | None |
| password_generator | <function password_generator at 0x041EE738> |
| random | <module 'random' from 'C:\\\...n36-32\\\\lib\\\\random.py'> |
| string | <module 'string' from 'C:\\\...n36-32\\\\lib\\\\string.py'> |

In area **1**, there are a number of buttons. With their help, we can control the process of debugging,

- **Go** runs the program as usual until an input is requested or until the program finishes. In our case, there would be no difference with the regular run of the program.
- **Step** serves to go through the code line by line: the most helpful action when debugging. If the line to be executed calls a function, the debugger will go to the first line of the function definition ("stepping into" the function). If we press this button now, we will find ourselves at the line where the `chars` variable is defined.
- **Over** is similar to the "Step" button, except that if the statement to be executed has a function call in it, the debugger executes the function without showing any details: it "steps over" the function, returning the result and pausing again. However, in our example, if we press "Over" in the beginning, the result will be the same as with the regular run of the program: debugger will execute the `password_generator()` function and pause; but this function is the only piece of code we have for debugging right now.
- **Out** is used when we are inside some function's code. It finishes execution of the function as it would do regularly, returns the result and pauses: we "step out" of the function. Instead of clicking "Step" repeatedly to jump out of the function, we can simply use "Out".
- **Quit** stops the execution of the entire program. This is helpful if we must start debugging again from the beginning of the program.

Area **2** contains checkboxes:

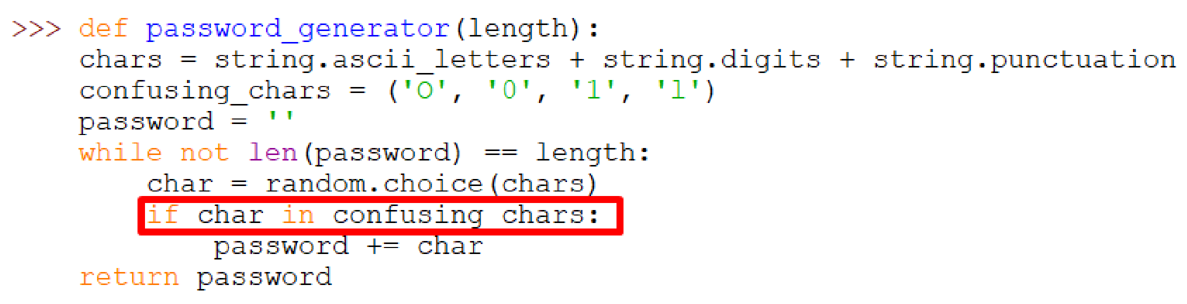
- "**Stack**" is what we can see in area **3**. It shows which line and from where is executed.
- "**Locals**" (area **4**) and "**Globals**" (area **5**) contain lists of local and global variables as they change. Remember, local variables are those created inside a function whereas global variables are those created outside of any functions.
- "**Source**" is helpful when we use some functions or classes from other modules. If it is checked, the module source file will open when we go through lines addressing it, and the corresponding line in the module

will be highlighted in gray. In our example, when we reach the line `char = random.choice(chars)`, the file "*random.py*" will open so that we can see what is happening inside the function `choice()` there.

## §3. Practice

Once we understand what's in this window, let's proceed to debugging our function.

Press the "Step" button several times and carefully look at what is happening in the debugger window. Once you have reached the line where our function initializes the `char` variable, check the value of this variable. If you see that some line wasn't executed, it means that you need to double check the line with `if` condition:



```
>>> def password_generator(length):
    chars = string.ascii_letters + string.digits + string.punctuation
    confusing_chars = ('O', '0', '1', 'l')
    password = ''
    while not len(password) == length:
        char = random.choice(chars)
        if char in confusing_chars:
            password += char
    return password
```

Ah, that's it! The condition is wrong, we need to check if it's NOT in `confusing_chars`. Sometimes we need to sleep more, sometimes we need a coffee-break, otherwise, we can make errors in our code.

```
def password_generator(length):
    chars = string.ascii_letters + string.digits + string.punctuation
    confusing_chars = ('O', '0', '1', 'l')
    password = ''
    while not len(password) == length:
        char = random.choice(chars)
        if char not in confusing_chars:
            password += char
    return password
```

After changing this line your function works perfectly!

```
>>> password_generator(5)
'wBxu9'
```

All glory to the Python shell and debugging!

## §4. Recap

The debugging process in the Python shell is quite simple and clear: it allows a user to run a code line by line to find errors and fix them.

Report a typo

87 users liked this theory. 3 didn't like it. **What about you?**

Start practicing

Comments (3)   Hints (2)   Useful links (1)   Show discussion