Debugging with Thonny

Page Content

Programming mistakes

Everyone makes mistakes—even seasoned professional developers!

Python is good at catching mistakes like syntax errors and run-time errors. Unfortunately, there is a third type of error that you may have already experienced. **Logic errors** occur when a valid program doesn’t do what the programmer intended.

For example, type the code below and save it has `buggy_code.py`.

For this tutorial download **`buggy_code.py`** file and save it to your lesson folder.

```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word

phrase = "hello"
print(add_underscores(phrase))
```

The expected output is `_h_e_l_l_o_`. Run it and you will see it actually outputs `o_`.

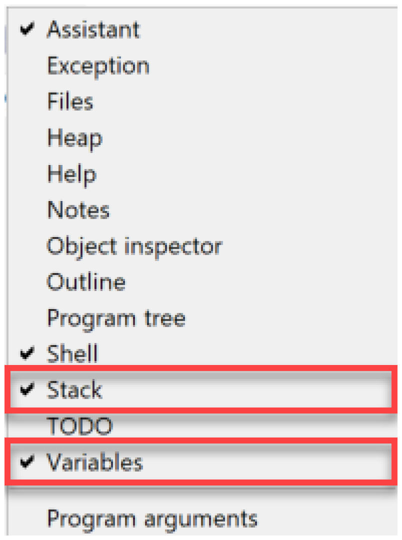
Logic errors cause unexpected behaviours called **bugs**. Debugging is process of removing bugs. A **debugger** is a tool that helps the programmer hunt down bugs and understand what’s happening.

Knowing how to find and fix bugs in your code is a skill that you will use for your entire coding career!

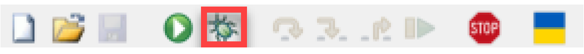
Using Thonny’s Debugger

To debug `buggy_code.py` we need to understand the debugging tools we have at our disposal. Thonny has a debugger built-in, but before we explore it, we need to make sure you have the correct setup.

Open the **View** menu and ensure there is a tick beside both **Stack** and **Variables**.



To enter into Thonny's Debugger click on the **Debug** button.



Controlling the debugger

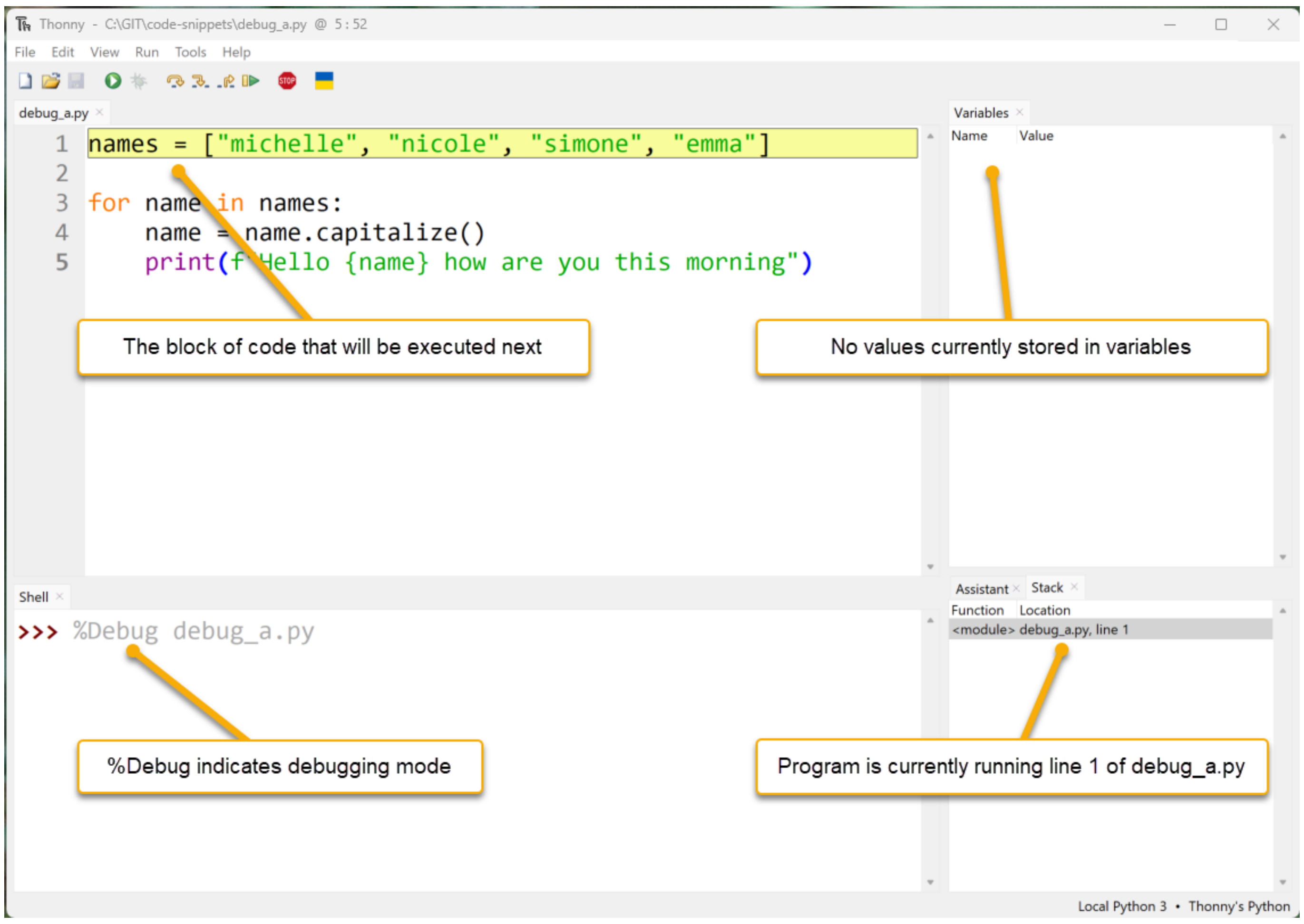
To see how the debugger works, let's start by writing a simple program without any bugs.

Type the following into Thonny and save it as `debug_a.py`:

```
1 names = ["michelle", "nicole", "simone", "emma"]
2
3 for name in names:
4     name = name.capitalize()
5     print(f"Hello {name} how are you this morning")
```

Now start Thonny's debugger.

Your Thonny should now look like the image below:


Thonny - C:\GIT\code-snippets\debug_a.py @ 5 : 52
File Edit View Run Tools Help
debug_a.py
Variables
Name Value
Shell
Assistant Stack
Function Location
<module> debug_a.py, line 1
Local Python 3 • Thonny's Python

```
1  names = ["michelle", "nicole", "simone", "emma"]
2
3  for name in names:
4      name = name.capitalize()
5      print(f"Hello {name} how are you this morning")
```

```
>>> %Debug debug_a.py
```

The block of code that will be executed next

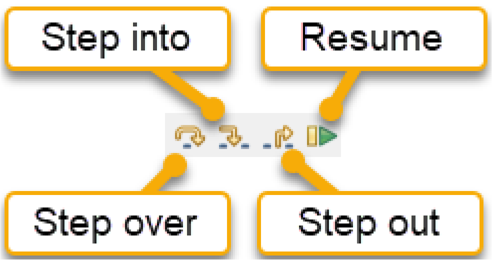
No values currently stored in variables

%Debug indicates debugging mode

Program is currently running line 1 of debug_a.py

- **Code Panel:** Thonny has paused the execution of the code. The yellow highlight shows the code that Python will execute next.
- **Variables Panel:** Since the program hasn't assigned any values, it shows no variables.
- **Shell Panel:** %Debug launches is the command that launches the program (debug_a.py).
- **Stack Panel:** Shows the current function and module that is running.

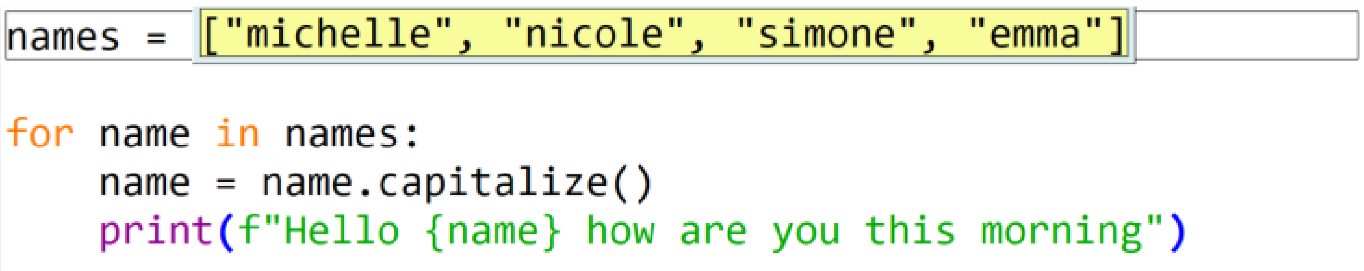
Something else has happened, other debugging buttons are now available.



Lets see how they work.

Step into button

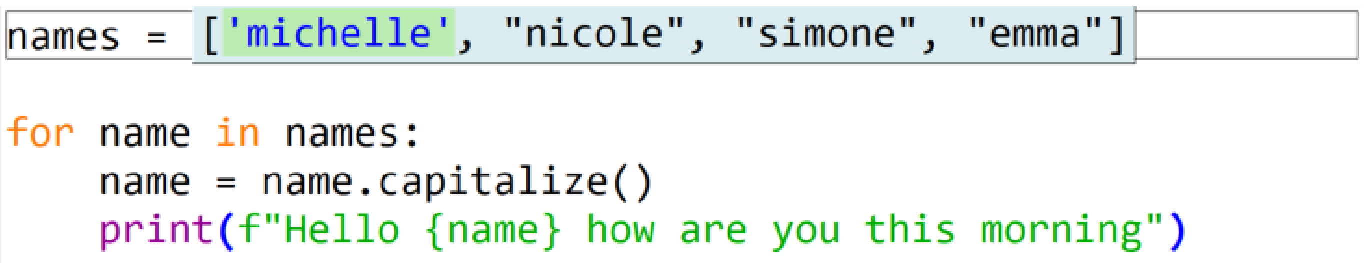
Click the **Step into** button. Thonny will now execute the previous highlighted code block. The new highlighted code indicates the part of the code to Python will execute next. In this case it is the list `["michelle", "nicole", "simone", "emma"]`.



```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

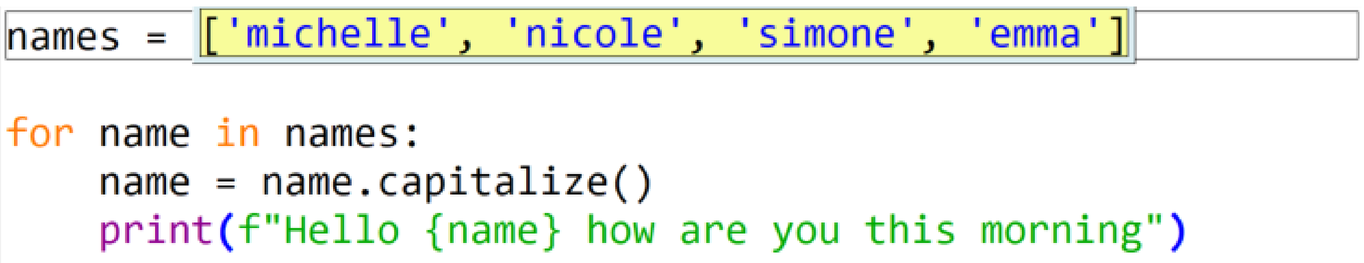
Clicking **Step into** again highlights `"michelle"`. This shows that Python will now evaluating this item.

Continuing and the `"michelle"` turns blue indicating that Python has read it.



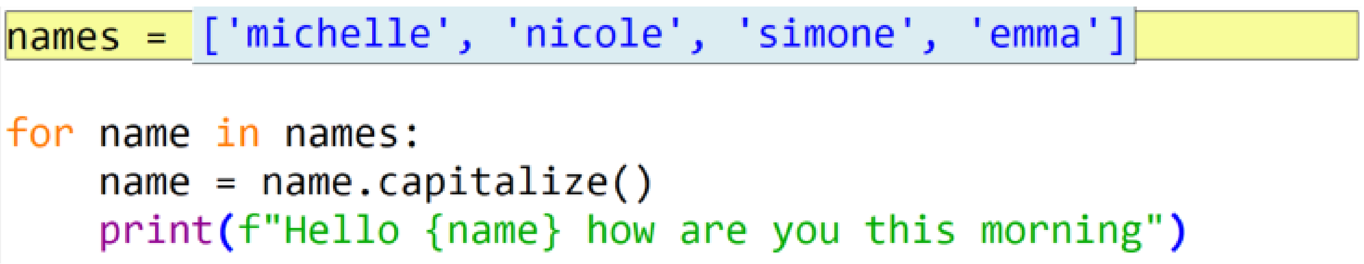
```
names = ['michelle', "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

Four more **Step into** clicks (or pressing **F7** on your keyboard), and Python has read all the strings.



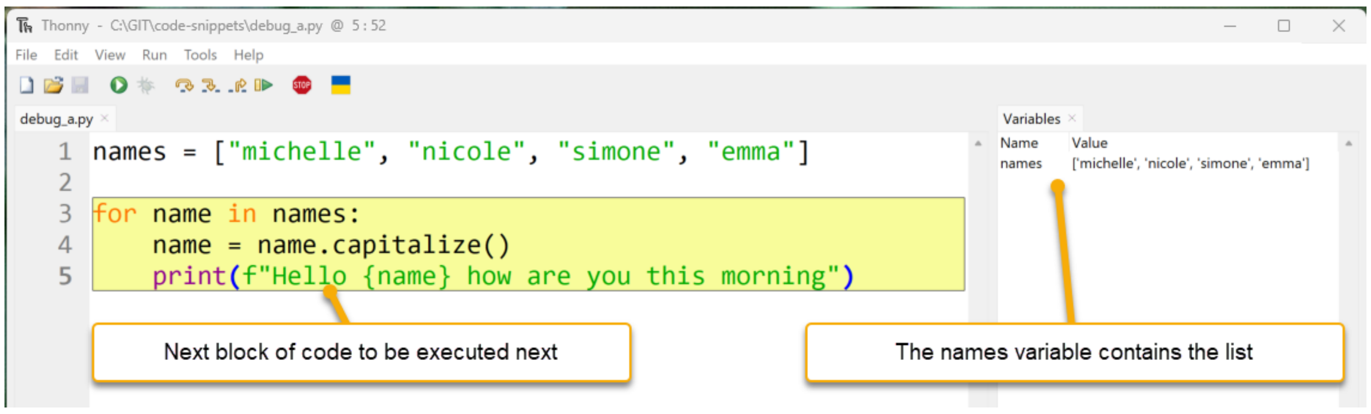
```
names = ['michelle', 'nicole', 'simone', 'emma']

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

The next **Step into** shows that Python is now ready to write the list into the variable `names`.



```
names = ['michelle', 'nicole', 'simone', 'emma']

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

The next **Step into** highlights the next block of code Python will process. The **Variables panel** shows that `names` now stores `["michelle", "nicole", "simone", "emma"]`.


Thonny - C:\GIT\code-snippets\debug_a.py @ 5 : 52
File Edit View Run Tools Help
debug_a.py
Variables

```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

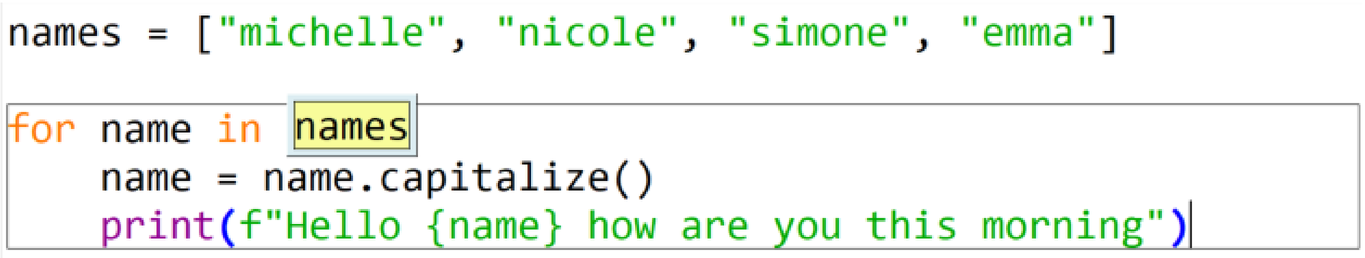
Name	Value
names	['michelle', 'nicole', 'simone', 'emma']

Next block of code to be executed next

The names variable contains the list

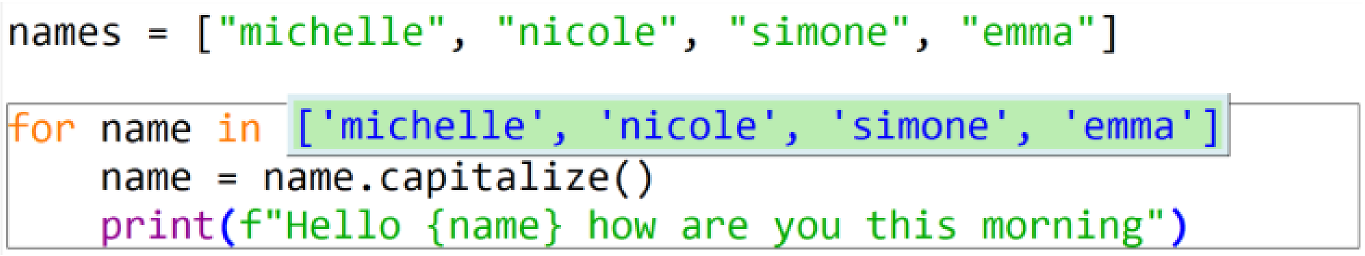
Notice that the highlight covers more than one line. Thonny is highlighting all the code that is part of the `for` loop. We'll now step through it.

Click **Step into** and it shows that the next executed code is `names`. This is the first element of the `for` statement.



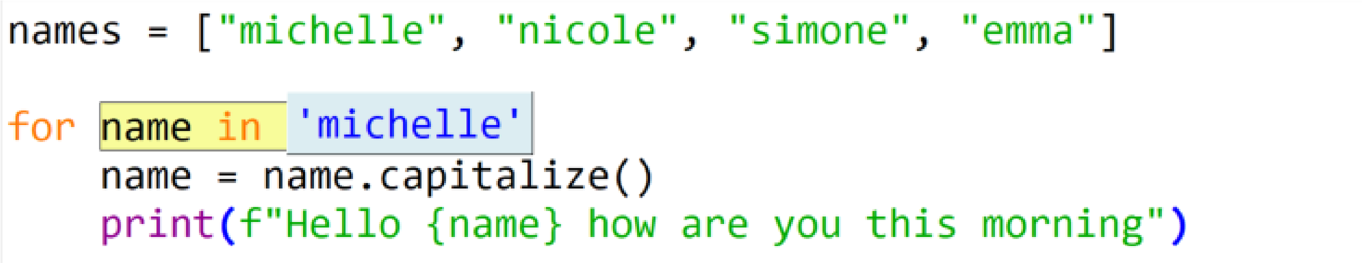
```
names = ["michelle", "nicole", "simone", "emma"]

for name in names
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

A further **Step into** replaces `names` with the list stored in `names`.



```
names = ["michelle", "nicole", "simone", "emma"]

for name in ['michelle', 'nicole', 'simone', 'emma']
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

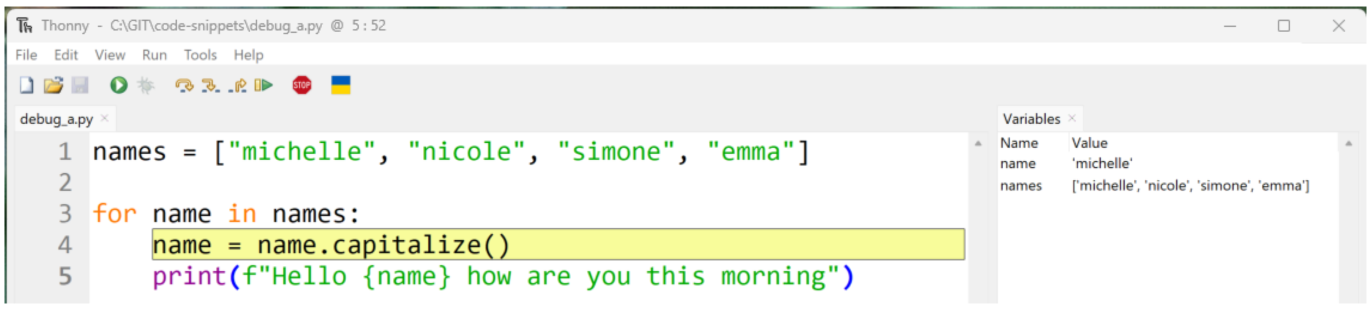
Since this is a for loop, the next **Step into** reads the first element of the list ('michelle').



```
names = ["michelle", "nicole", "simone", "emma"]

for name in 'michelle'
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

Clicking **Step into** highlights `line 4`. `michelle` is now stored in the `name` variable as displayed in the **Variables panel**.

> **Note**
>
> Do not confuse `name` with `names`. They're very close, but Python is very precise.



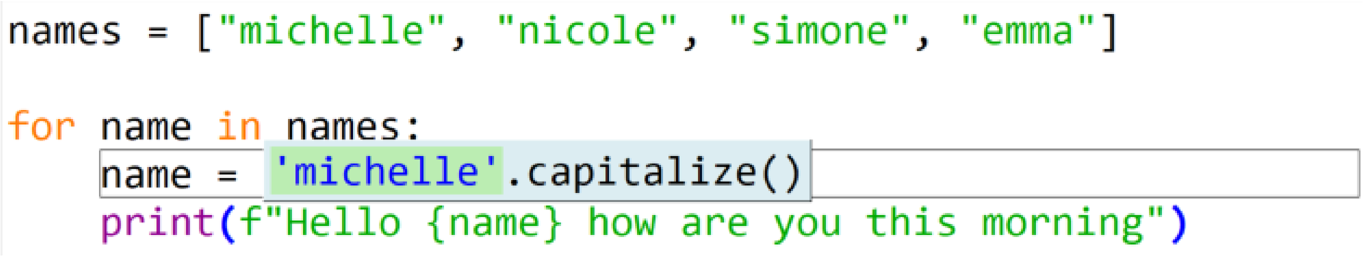
Thonny - C:\GIT\code-snippets\debug_a.py @ 5 : 52
File Edit View Run Tools Help
debug_a.py
Variables

```
1 names = ["michelle", "nicole", "simone", "emma"]
2
3 for name in names:
4     name = name.capitalize()
5     print(f"Hello {name} how are you this morning")
```

Name	Value
name	'michelle'
names	['michelle', 'nicole', 'simone', 'emma']

The next three **Step into** clicks:

1. highlights the `name.capitalize()`
2. highlights `name`
3. replaces `name` with `'michelle'`



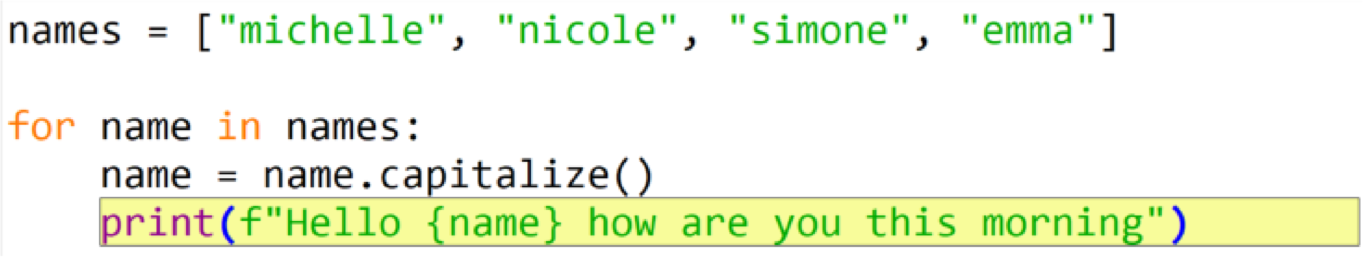
```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = 'michelle'.capitalize()
    print(f"Hello {name} how are you this morning")
```

The next three **Step into** clicks:

1. applies the `capitalize()` method to `'michelle'`

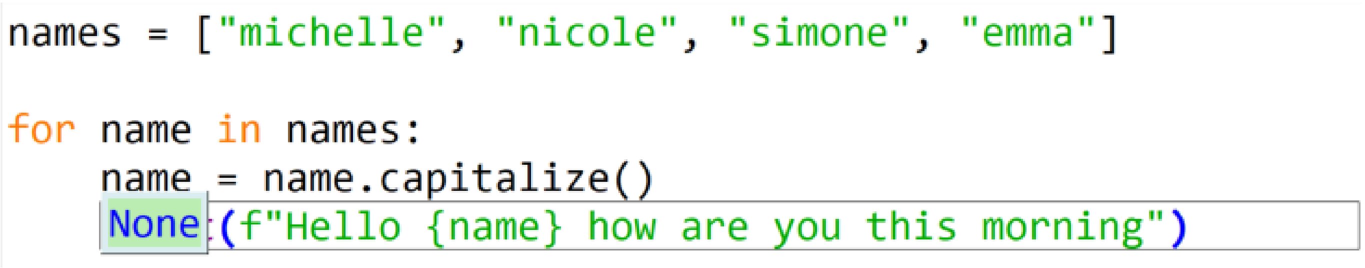
2. changes `'michelle'` to `'Michelle'`
3. overwrites the value in `name` with `'Michelle'`

`Line 4` is now complete, so `line 5` is now highlighted.



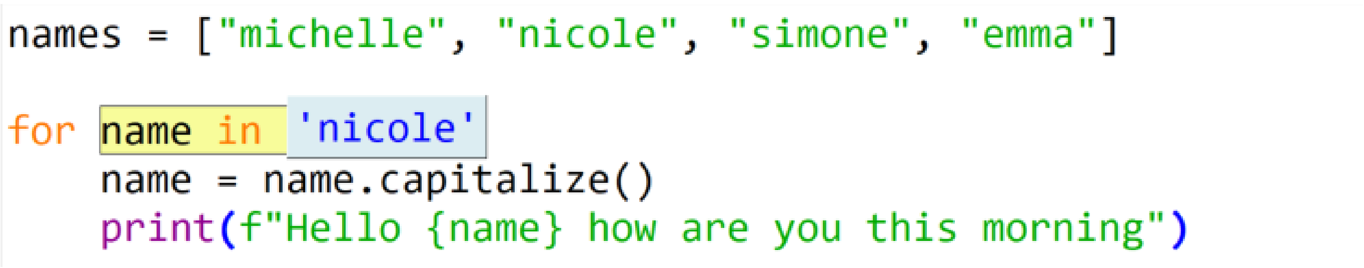
```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

The next five **Step into** clicks show how Python processes an f-string and prints it to the terminal. It also shows that a call to the `print()` function returns a `None` value.



```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    None(f"Hello {name} how are you this morning")
```

One more click on **Step into** returns to `line 3`. The `for` statement, has taken the next element from the names list (`'Nicole'`).



```
names = ["michelle", "nicole", "simone", "emma"]

for name in 'nicole'
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

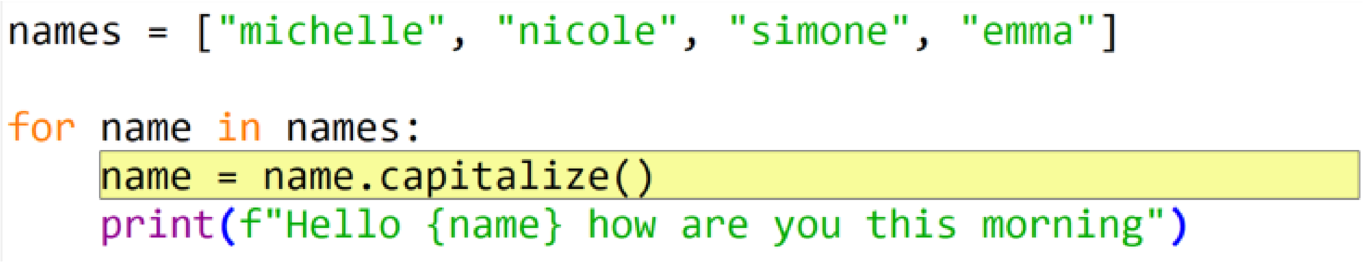
Click **Step into** one more time and Python stores `'Nicole'` in `name`.

We will use the next iteration of the `for` loop investigate how **Step over** works.

Step over button

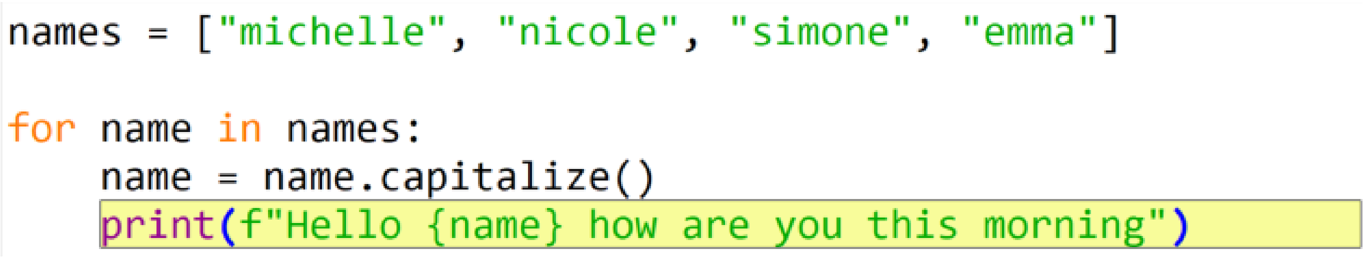
The **Step over** function processes the highlighted code without going into the details. The next click of **Step over:**

1. takes the value stored in name ('nicole')
2. capitalizes it to 'Nicole'
3. writes it back to name.



```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

Click the **Step over** button and you will see the results below. Notice the value stored in `name`.

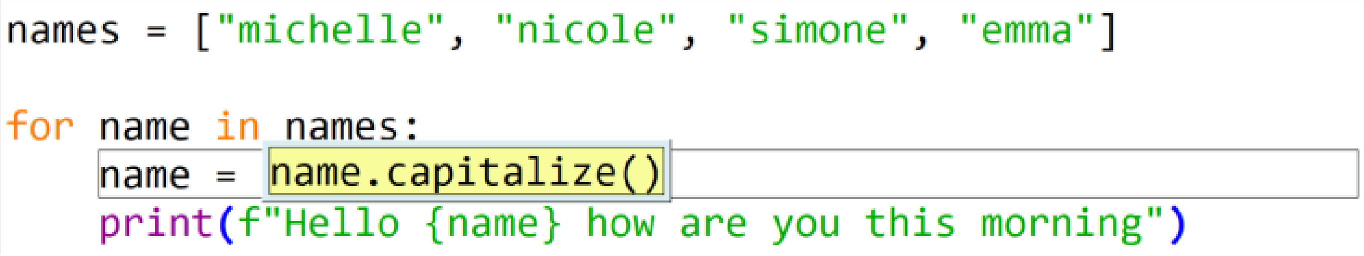


```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

Clicking **Step over** again executes `line 5`. The highlight then returns to the `line 3` for statement.

> **When to use Step over**
>
> Use **Step over** when you know that the highlighted code is bug free. Executing working code helps find the bug location quicker.

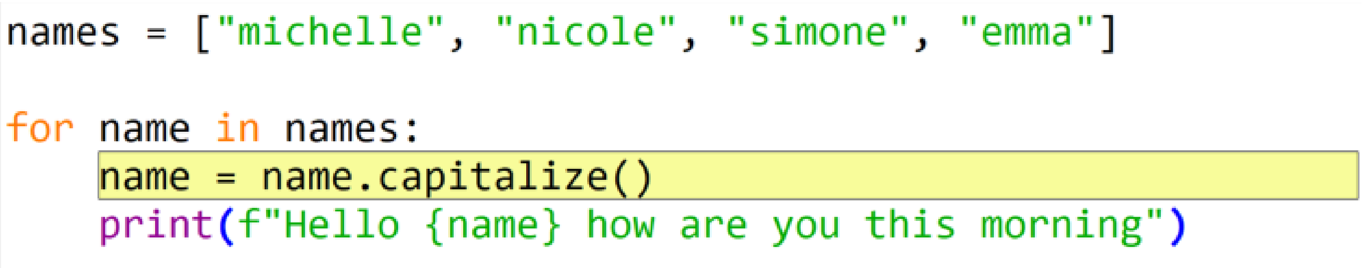
Click **Step over** and then **Step into** to move your code to the position below, so we can now look at the **Step out**.



```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

Step out button

The **Step out** completes the rest of the currently highlighted code. Look back at the last example, and notice the grey box around `line 4`? This box indicates that we have stepped into `line 4` and are processing it. Click **Step out** and Thonny will move back up a level to highlight all `line 4`.



```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

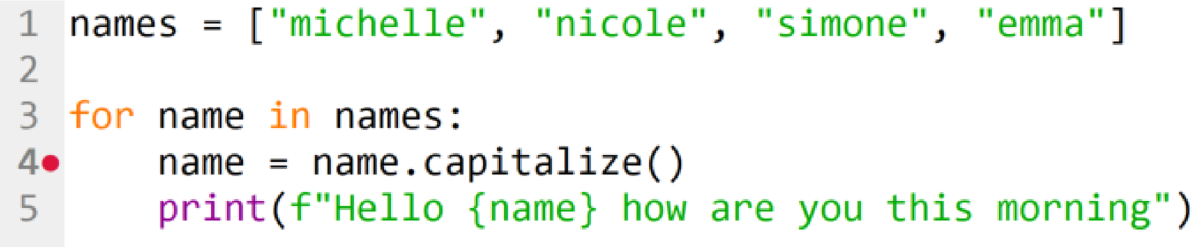
Clicking **Step out** again will move back up one level. The debugger is now to outside the for loop and the program will finish.

Resume button and breakpoints

The last button we need to look at is the **Resume** button. This works in conjunction with **breakpoints**. The **Resume** button will execute the code until it finds a **breakpoint**, then it will pause**.**

To add a **breakpoint** to your code, click on the line number that you wish to pause the program on.

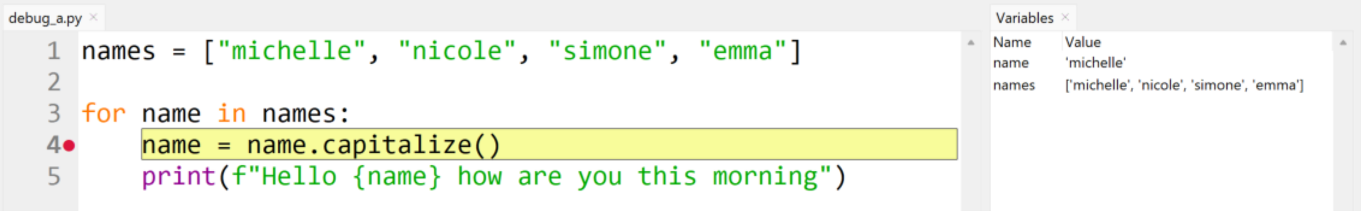
Let’s try this out. Click on the line number 4. A red dot should appear between the number and the code panel (as below):



```
1 names = ["michelle", "nicole", "simone", "emma"]
2
3 for name in names:
4●    name = name.capitalize()
5     print(f"Hello {name} how are you this morning")
```

Now click **Debug**.

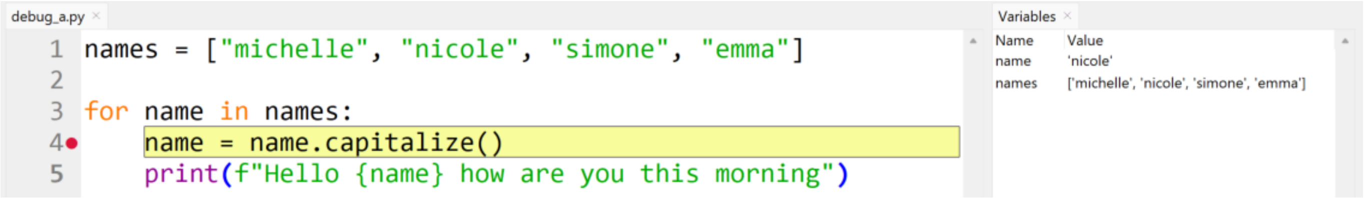
Notice that the program runs and then pauses at the **breakpoint**. At this point, you can check the current values of the variables.



```
1 names = ["michelle", "nicole", "simone", "emma"]
2
3 for name in names:
4●    name = name.capitalize()
5     print(f"Hello {name} how are you this morning")
```

Name	Value
name	'michelle'
names	['michelle', 'nicole', 'simone', 'emma']

The next **Resume** click will restart the program. The debugger will pause at the next **breakpoint.** This is also `line 4` on the second iteration of the for loop.



```
names = ["michelle", "nicole", "simone", "emma"]

for name in names:
    name = name.capitalize()
    print(f"Hello {name} how are you this morning")
```

Name	Value
name	'nicole'
names	['michelle', 'nicole', 'simone', 'emma']

Hint

Notice the changed values in the variables.

Now that we know how to control Thonny's debugger, let's go back and debug `buggy_code.py`.

Debugging a Logic Error

First let's look closely at `buggy_code.py`:

```
1 def add_underscores(word):
2     new_word = "_"
3     for index in range(len(word)):
4         new_word = word[index] + "_"
5     return new_word
6
7 phrase = "hello"
8 print(add_underscores(phrase))
```

Step 1: Make a guess about where the bug is located

The first step is to identify the section of code that is likely to contain the bug. You may not be able to identify exactly where the bug. So, make a reasonable guess about which section of your code has an error.

Notice that there are two distinct sections of the program:

- a function definition → `lines 1` to `5`
- main code block
    - `line 7` → defines a variable `phrase` with the value `"hello"`
    - `line 8` → then prints the result of calling `add_underscores(phrase)`

Look at the main section:

```
7 phrase = "hello"
8 print(add_underscores(phrase))
```

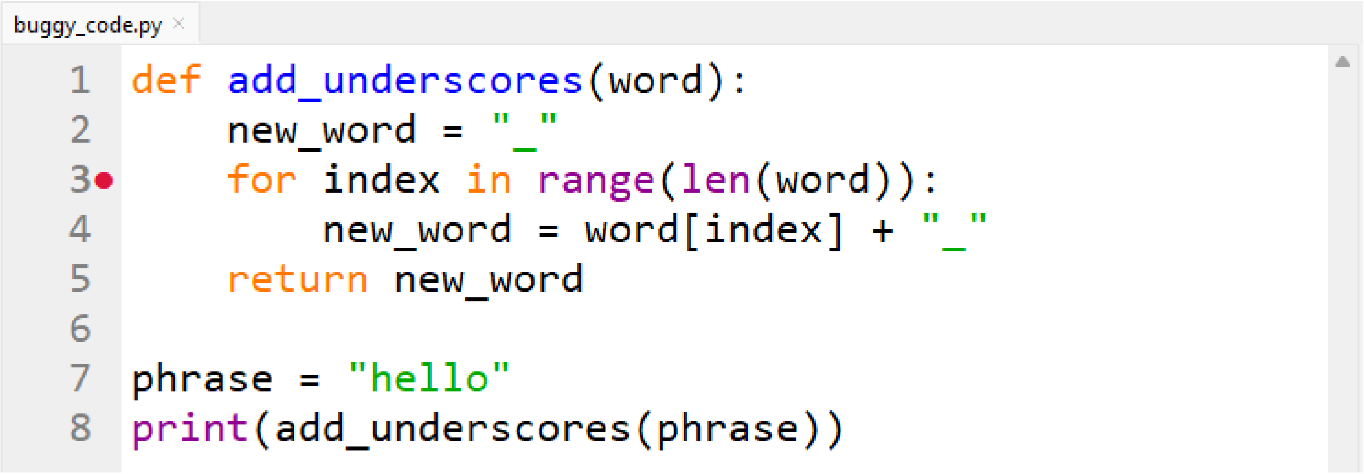
Do you think the problem could be here? It doesn't look like it, right? Everything about those two lines of code looks good. So, the problem must be in the function definition:

```
1 def add_underscores(word):
2     new_word = "_"
3     for index in range(len(word)):
4         new_word = word[index] + "_"
5     return new_word
```

The first line of code inside the function creates a variable `new_word` with the value `"_"`. You're all good there, so you can conclude that the problem is somewhere in the body of the `for` loop.

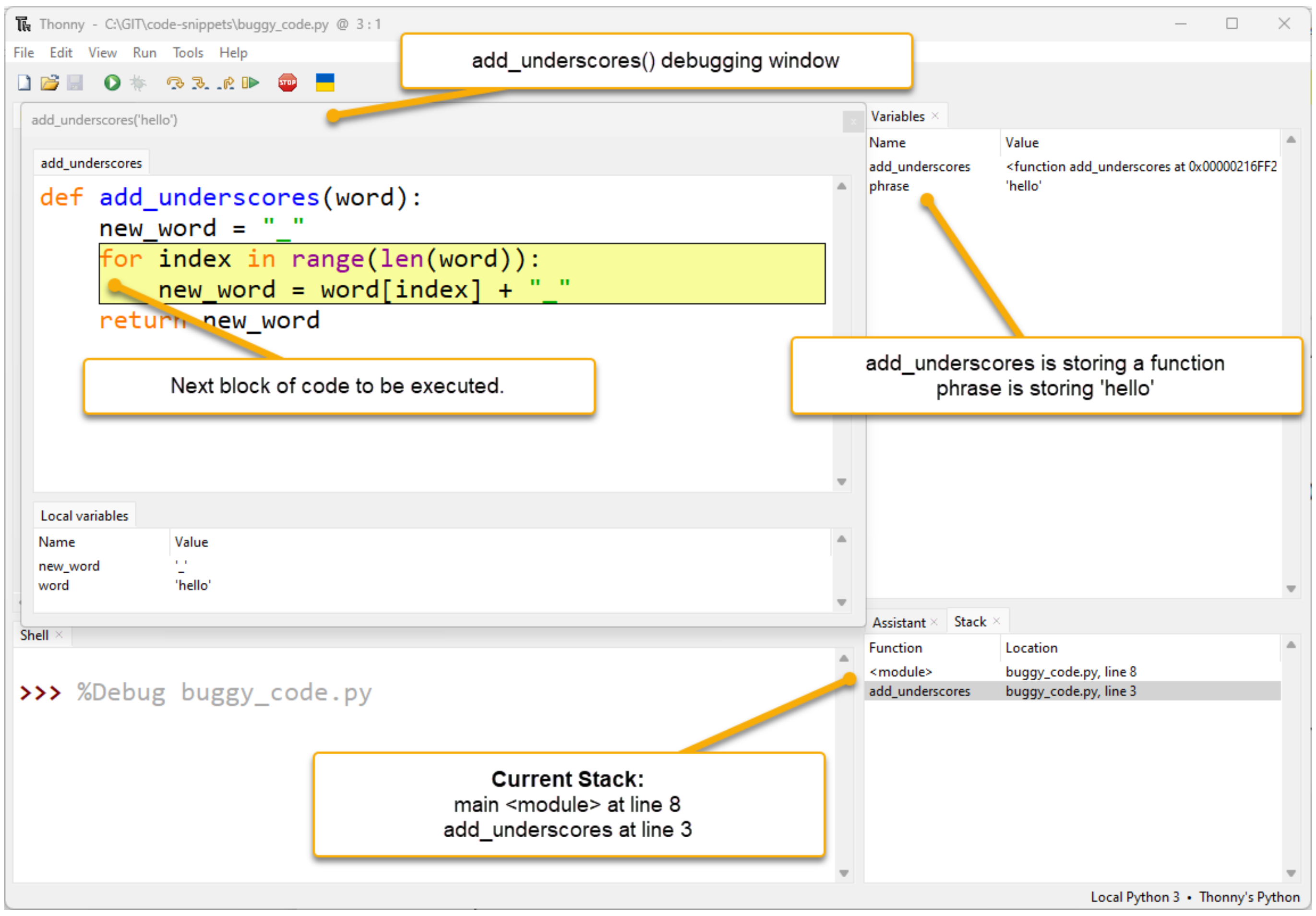
Step 2: Set a breakpoint and inspect the code

Now that we've identified where the bug must be, set a breakpoint at the start of the for loop. This way we can trace out exactly what's happening inside the loop with the debugging tool:



```
1  def add_underscores(word):
2      new_word = "_"
3      for index in range(len(word)):
4          new_word = word[index] + "_"
5      return new_word
6
7  phrase = "hello"
8  print(add_underscores(phrase))
```

Click the **Debug** button to launch Thonny's debugger. Thonny will run the code until it hits the breakpoint. Your IDE should look like the image below. There are some new features we haven't seen before.


Thonny - C:\GIT\code-snippets\buggy_code.py @ 3 : 1
File Edit View Run Tools Help
add_underscores('hello')
add_underscores
Variables
Local variables
Shell
Assistant
Stack
Local Python 3 • Thonny's Python

add_underscores() debugging window

```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

Next block of code to be executed.

Name	Value
add_underscores	<function add_underscores at 0x00000216FF2
phrase	'hello'

add_underscores is storing a function
phrase is storing 'hello'

Name	Value
new_word	'_'
word	'hello'

```
>>> %Debug buggy_code.py
```

Function	Location
<module>	buggy_code.py, line 8
add_underscores	buggy_code.py, line 3

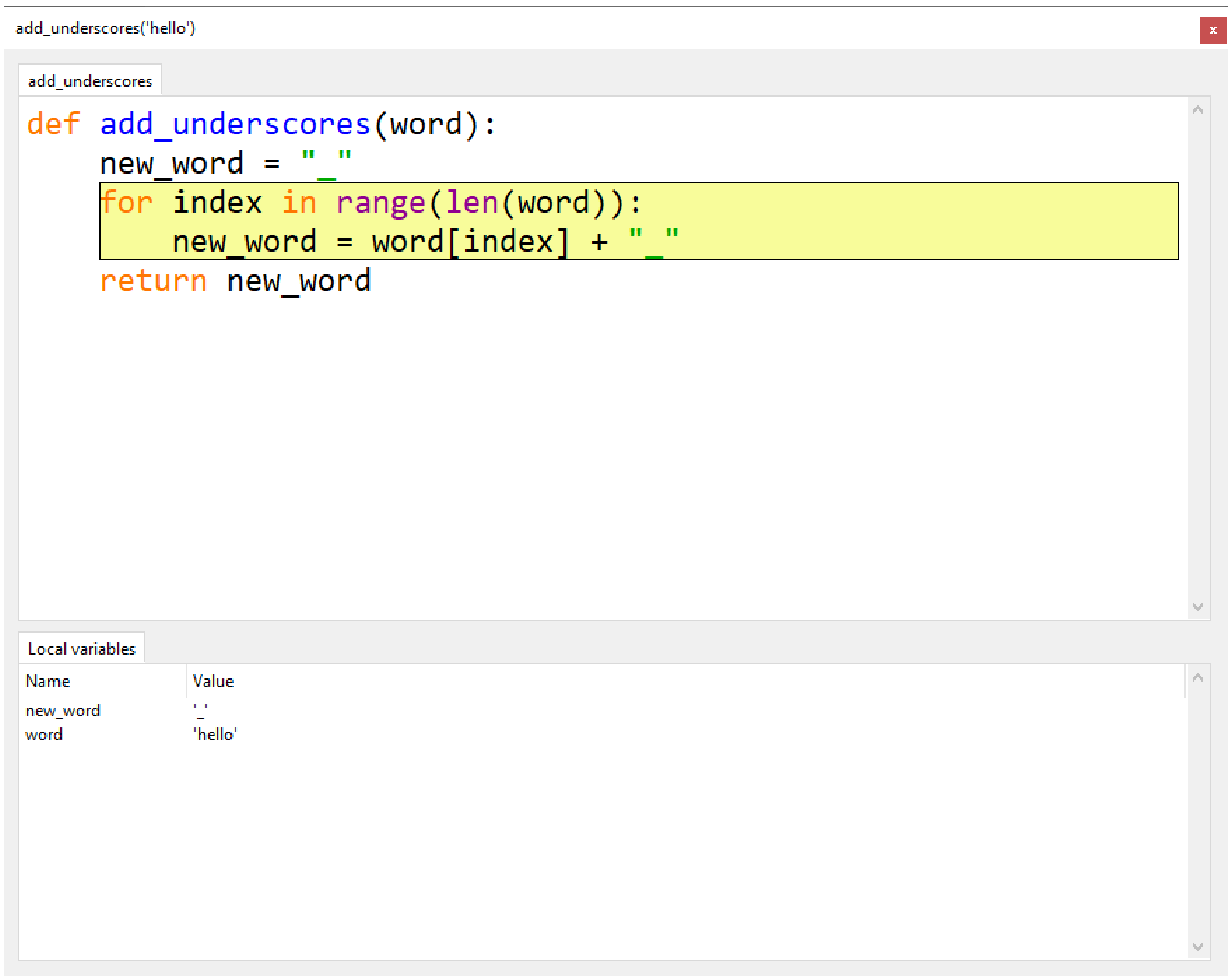
Current Stack:
main <module> at line 8
add_underscores at line 3

- **additional debugging window:**
  - notice that Thonny has launched a new debugging window for `add_underscores('hello')` function.
  - Whenever a Python enters a new scope Thonny will launch a debugging window for that scope.
  - The values stored in the **local variables** are at the bottom of the new window.
  - **Local variables** are variables that only the current function can see.
- **multiple stack values:**
  - in the stack panel you will now see two values.
    - first value refers to the main section
    - second value refers to the add_underscores function section
  - This shows us that the program is at:
    - `line 8` in the main program
    - `line 3` of the add_underscores function section.

> ### ❗ Stack timeline
>
> 1. `line 8` in the main module called the `add_underscores` function
> 2. Python pauses the main section at `line 8` at wait for the `add_underscores` function to finish
> 3. When the function finishes, the main section will continue from `line 8` onwards.

Back to debugging. Notice the `add_underscores` window displays the word and new_word variables. Currently, `word` has the value `'hello'` and `new_word` has the value `'_'`, as expected.

Let's look further.



add_underscores('hello')

add_underscores

```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

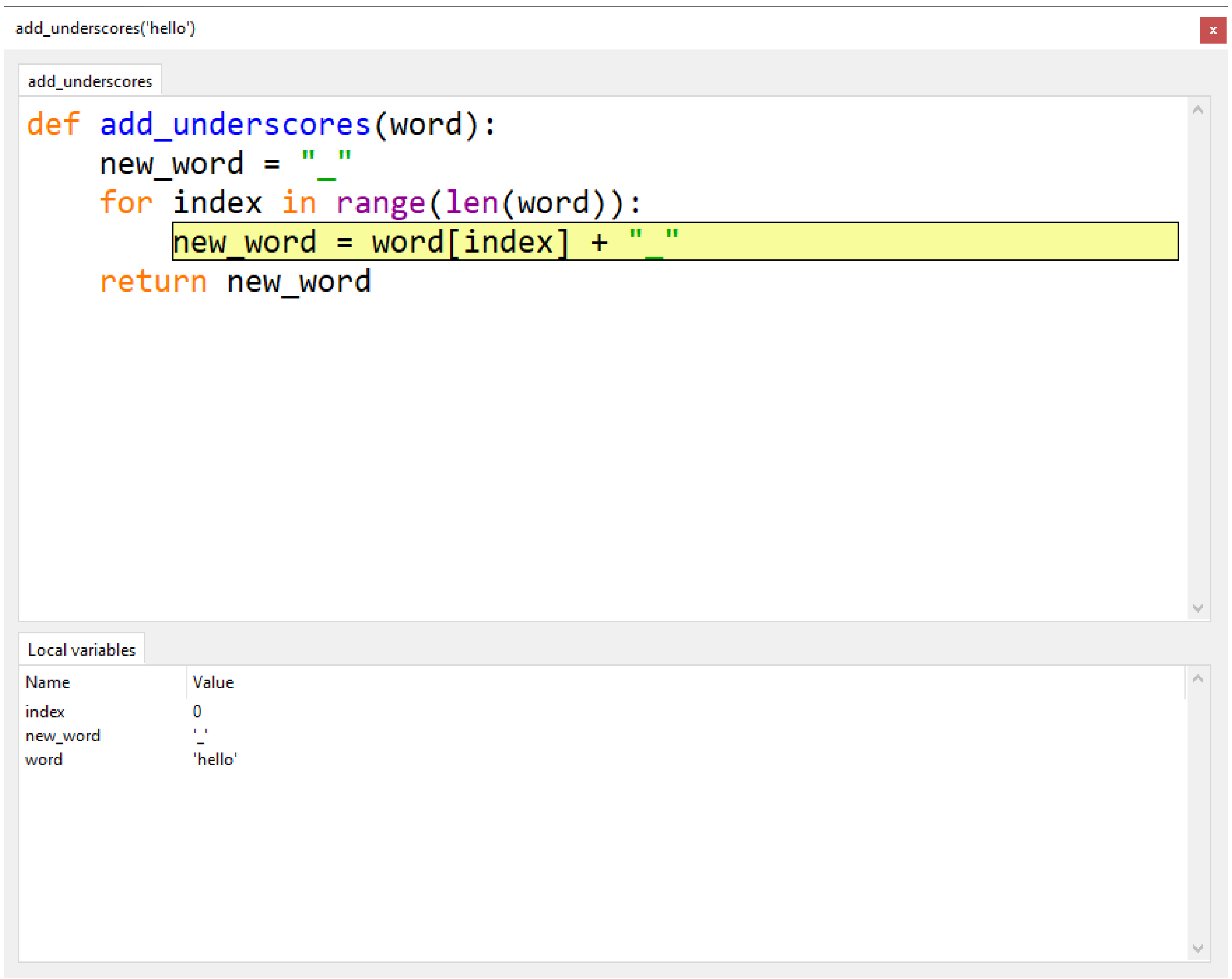
Local variables

Name	Value
new_word	'_'
word	'hello'

Click:

- **Step into** once
- **Step over** twice

You should end up with `new_word = word[index] + "_"` highlighted and ready to process (like below).



add_underscores('hello')

add_underscores

```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

Local variables

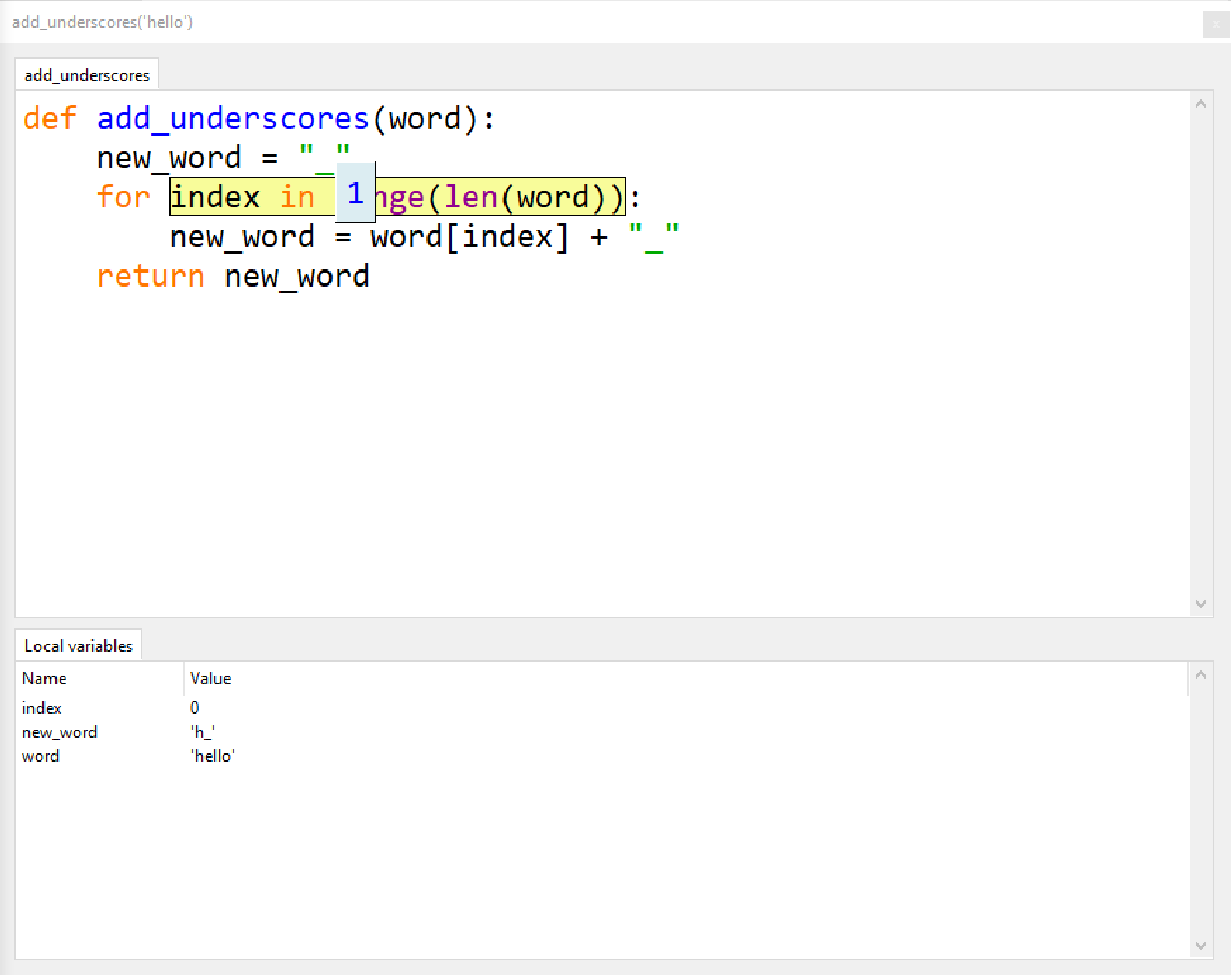
Name	Value
index	0
new_word	'_'
word	'hello'

Notice the local variable index is storing 0. This is correct for the first iteration of the loop.

Hint

If you can't see the index variable, you may need to resize the *Local variables* panel.

Now click **Step over** to execute `new_word = word[index] + "_"` and have a look at the results.



```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

```
Name          Value
index         0
new_word      'h_'
word          'hello'
```

Notice that new_word is now storing `'h_'`, whereas we want it to be storing `'_h_'`. What happened there? **We have found our error location.**

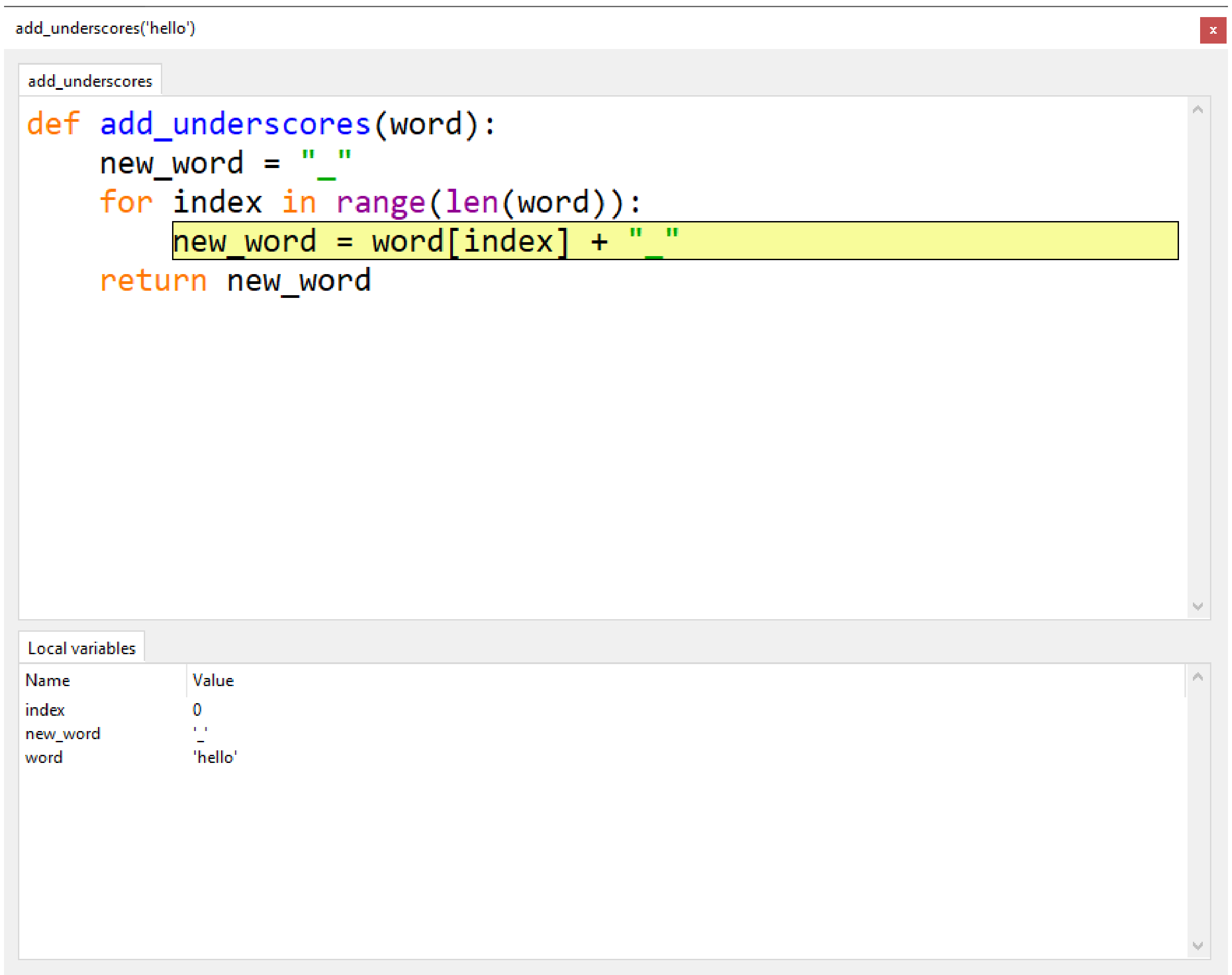
Now that we know the error is in the new_word = word[index] + "_" code. Let's investigate that code and see exactly what happened.

First click **Stop** and then **Debug** again.

Click:

- **Step into** once
- **Step over** twice

You should now have the problem code highlighted (like below).

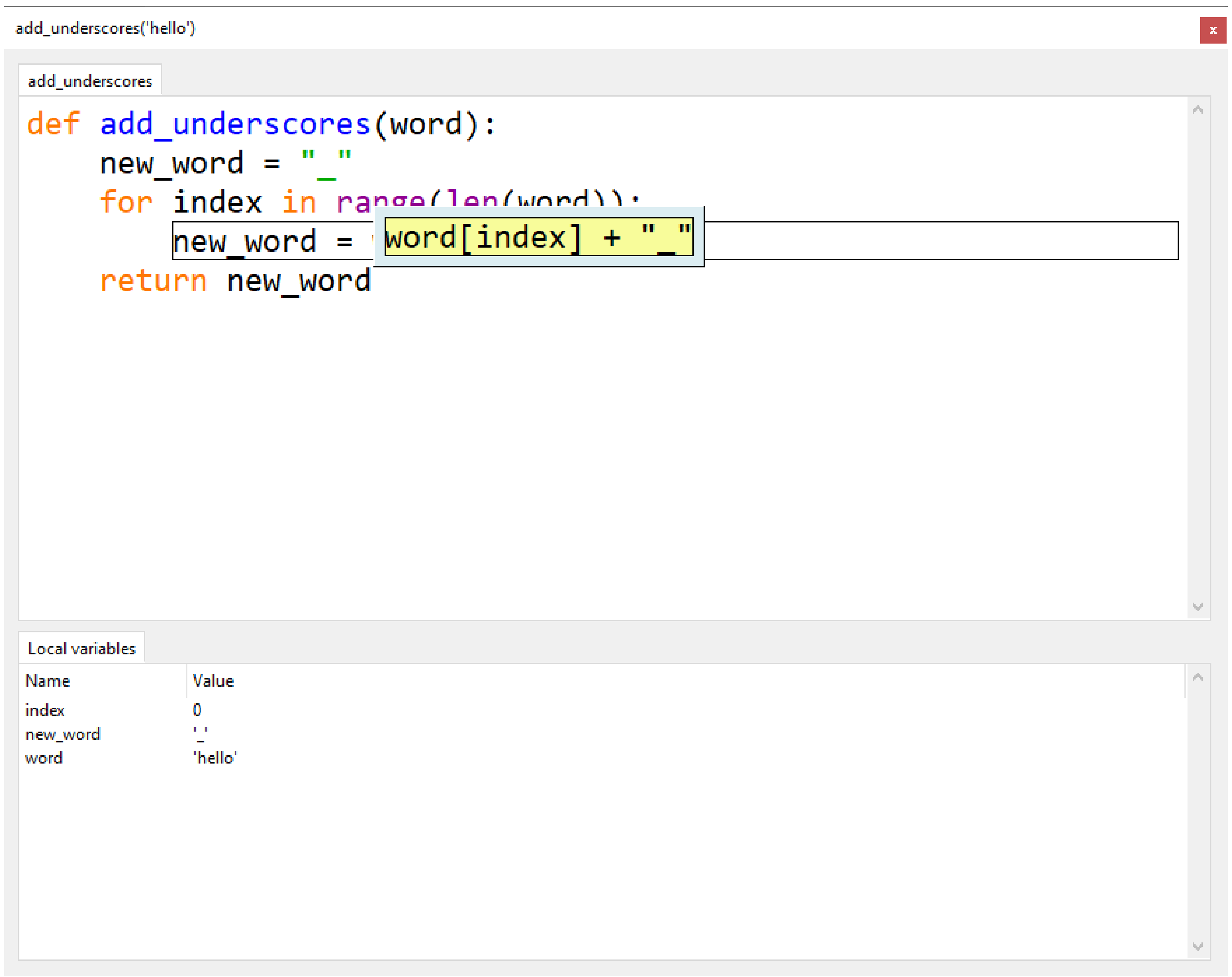


```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

```
Name          Value
index         0
new_word      '_'
word          'hello'
```

This time we will step into the code and see what happens.

First **Step into** and everything looks good.

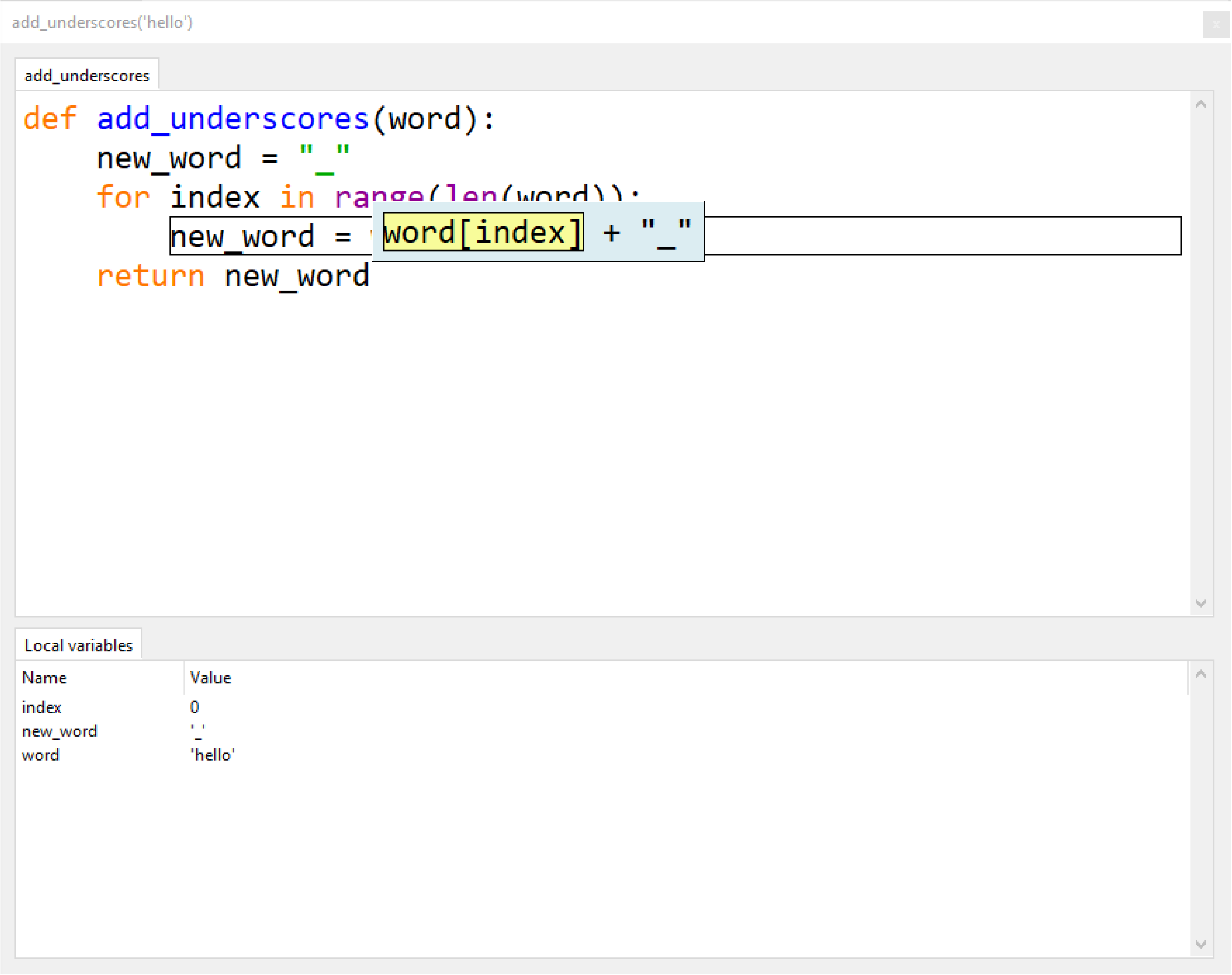


add_underscores('hello')

```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

Name	Value
index	0
new_word	'_'
word	'hello'

Second **Step into** everything is still fine.

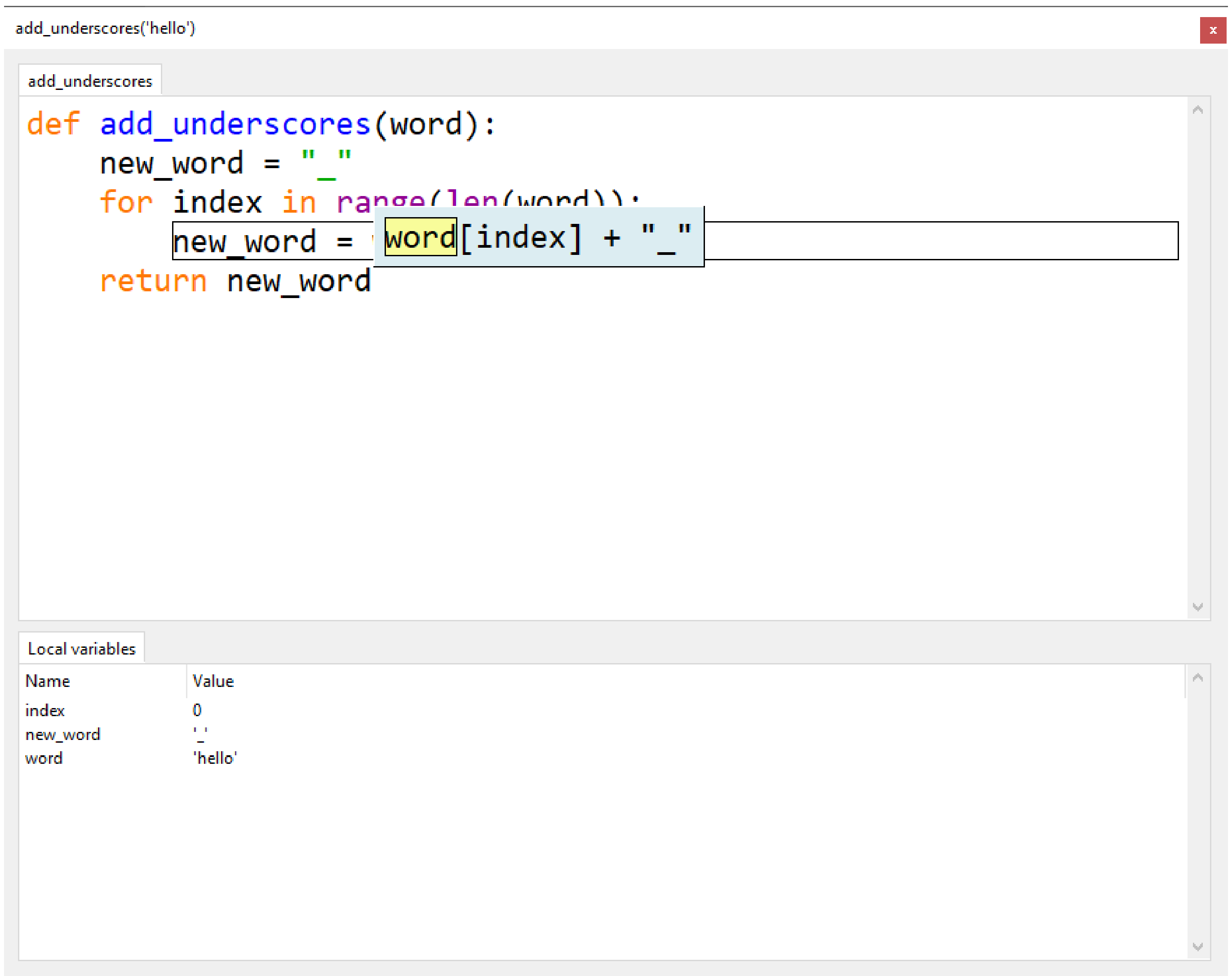


add_underscores('hello')

```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

Name	Value
index	0
new_word	'_'
word	'hello'

Third **Step into**, all good.


add_underscores('hello')

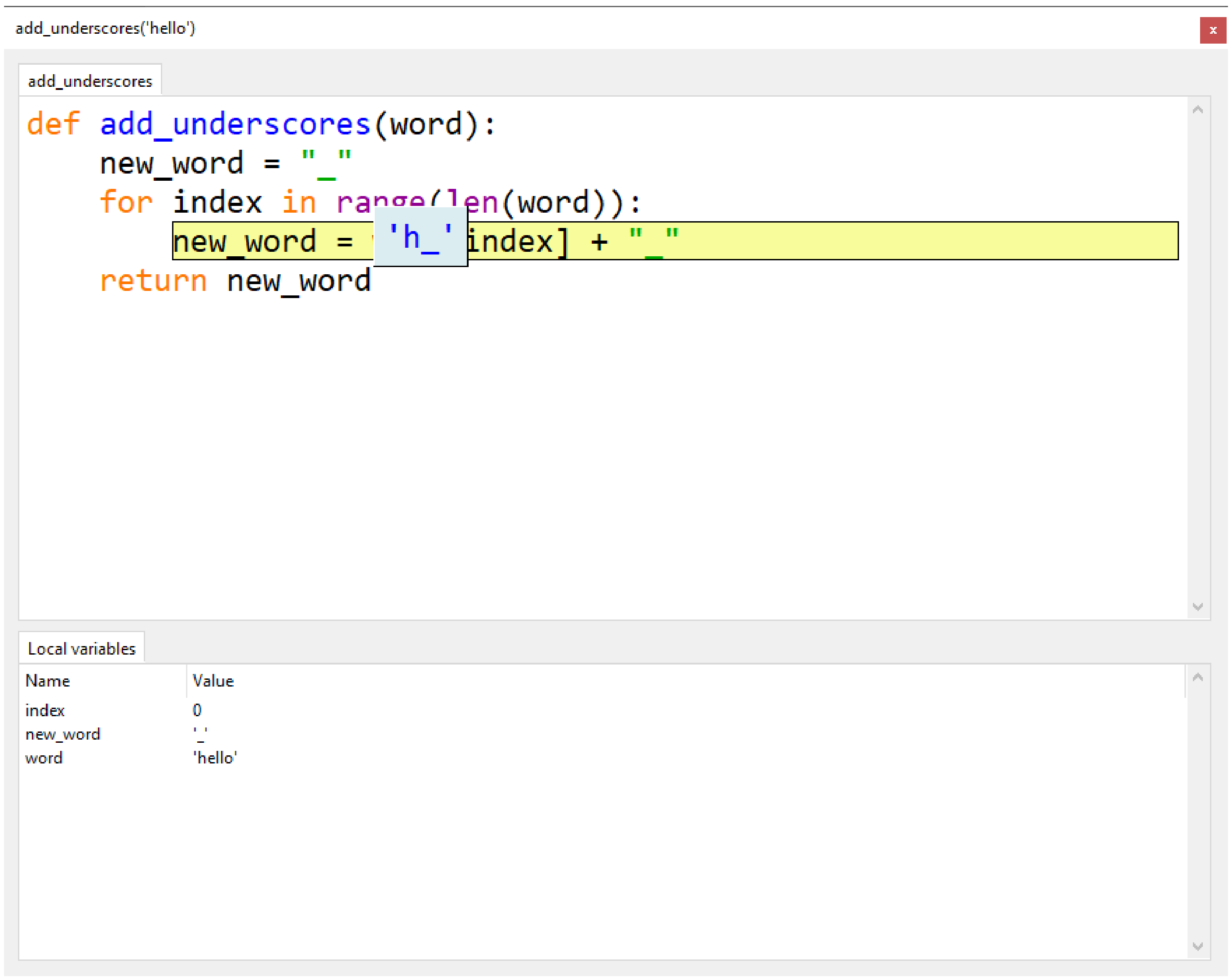
add_underscores

```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

Local variables

Name	Value
index	0
new_word	'_'
word	'hello'

Keep click **Step into** and follow what is happening in the Local variables. Stop when your `add_underscores('hello')` is in the state below:


add_underscores('hello')

add_underscores

```
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = word[index] + "_"
    return new_word
```

Local variables

Name	Value
index	0
new_word	'_'
word	'hello'

Looking closely at the debugging code. You will notice:

- Python is about to assign the value of `'h_'` to `new_word`
- This is not right!
- We want it to assign `'_h_'` to `new_word`

Now we know **exactly** where the problem is, we need to work out why this is a problem.

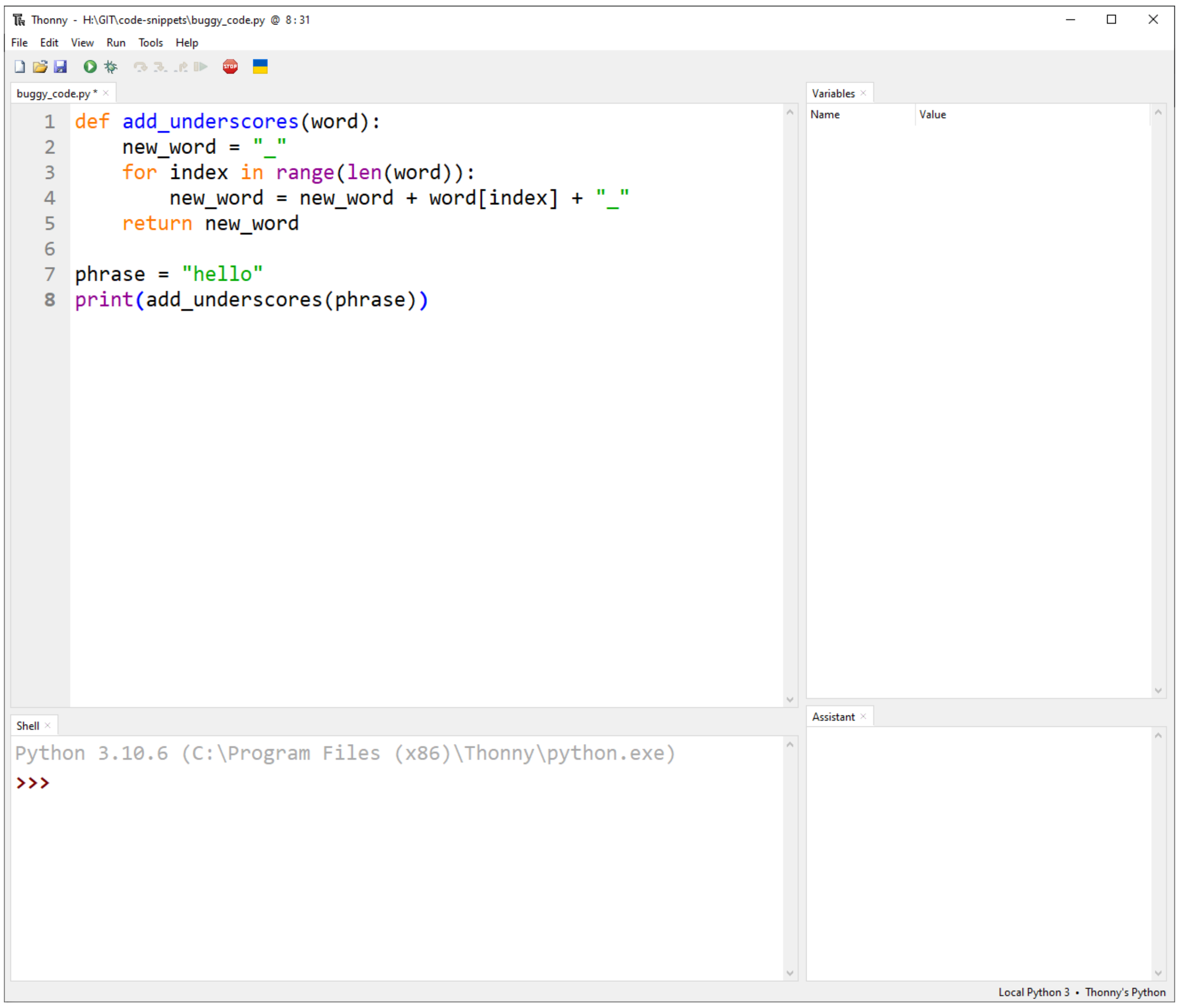
`add_underscores()` is supposed to insert a _ between each letter. It does this by repetitively concatenating the next letter and _ to the value stored in new_word.

But our code is overwriting `new_word`. We are loosing all the previously processed letters.

What we need to do is concatenate the current value of `new_word` in front of the processed letter and _.

The line should read `new_word = new_word + word[index] + "_"`

Stop the debugging, make the change to the code below.


Thonny - H:\GIT\code-snippets\buggy_code.py @ 8 : 31
File Edit View Run Tools Help
buggy_code.py *
Variables
Name
Value
Shell
Assistant
Local Python 3 • Thonny's Python

```python
def add_underscores(word):
    new_word = "_"
    for index in range(len(word)):
        new_word = new_word + word[index] + "_"
    return new_word

phrase = "hello"
print(add_underscores(phrase))
```

```
Python 3.10.6 (C:\Program Files (x86)\Thonny\python.exe)
>>>
```

Run the program normally. Is the output `_h_e_l_l_o_`?

Problem solved. You have now successfully debugged faulty code.

By Damien Murtagh