COL1000
Introduction to Programming

Priyanka Golia

Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

Debugging

What is debugging: when you write a program, bugs (errors or unexpected behavior) are inevitable.

Debugging means finding and fixing those bugs.

You can't avoid bugs — but you can learn to fix them efficiently.

Debugging

What is debugging: when you write a program, bugs (errors or unexpected behavior) are inevitable.

Debugging means finding and fixing those bugs.

You can't avoid bugs — but you can learn to fix them efficiently.

```
def avg(x, y):
    return x + y / 2
```

$\longrightarrow$ Specification — take two positive integers and compute their avg.

Debugging

What is debugging: when you write a program, bugs (errors or unexpected behavior) are inevitable.

Debugging means finding and fixing those bugs.

You can't avoid bugs — but you can learn to fix them efficiently.

```
def avg(x, y):
    return x + y / 2
```

$\longrightarrow$ Specification — take two positive integers and compute their avg.

```
avg(10,20)
```

$\longrightarrow$ What should be the output?

Debugging

What is debugging: when you write a program, bugs (errors or unexpected behavior) are inevitable.

Debugging means finding and fixing those bugs.

You can't avoid bugs — but you can learn to fix them efficiently.

```
def avg(x, y):
    return x + y / 2
```

$\rightarrow$ Specification — take two positive integers and compute their avg.

```
avg(10,20)
```

$\rightarrow$ What should be the output?

It returns 20 instead of 15 — because division happens before additions.

Debugging with Print statement

The old way to do the debugging is to Print —
“when in doubt, print it out”

Debugging with Print statement

The old way to do the debugging is to Print —
“when in doubt, print it out”

```
def avg(x, y):
    print("x =", x, "y =", y)
    result = x + y / 2
    print("result =", result)
    return result
```

Debugging with Print statement

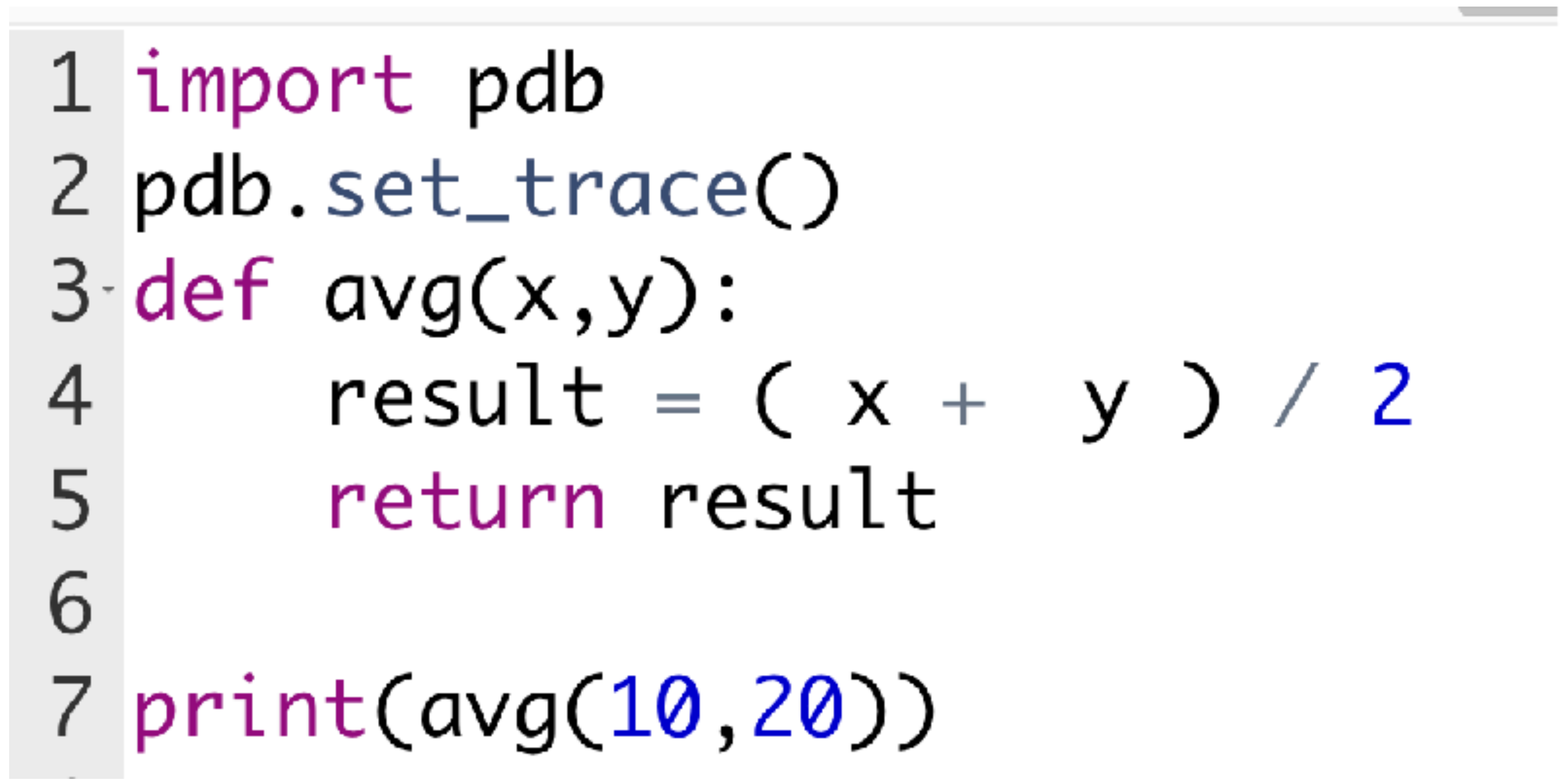
The old way to do the debugging is to Print —
“when in doubt, print it out”

```
def avg(x, y):
    print("x =", x, "y =", y)
    result = x + y / 2
    print("result =", result)
    return result
```

```
x = 10 y = 20
result = 20.0
```

Using Python's Built-in Debugger (pdb)

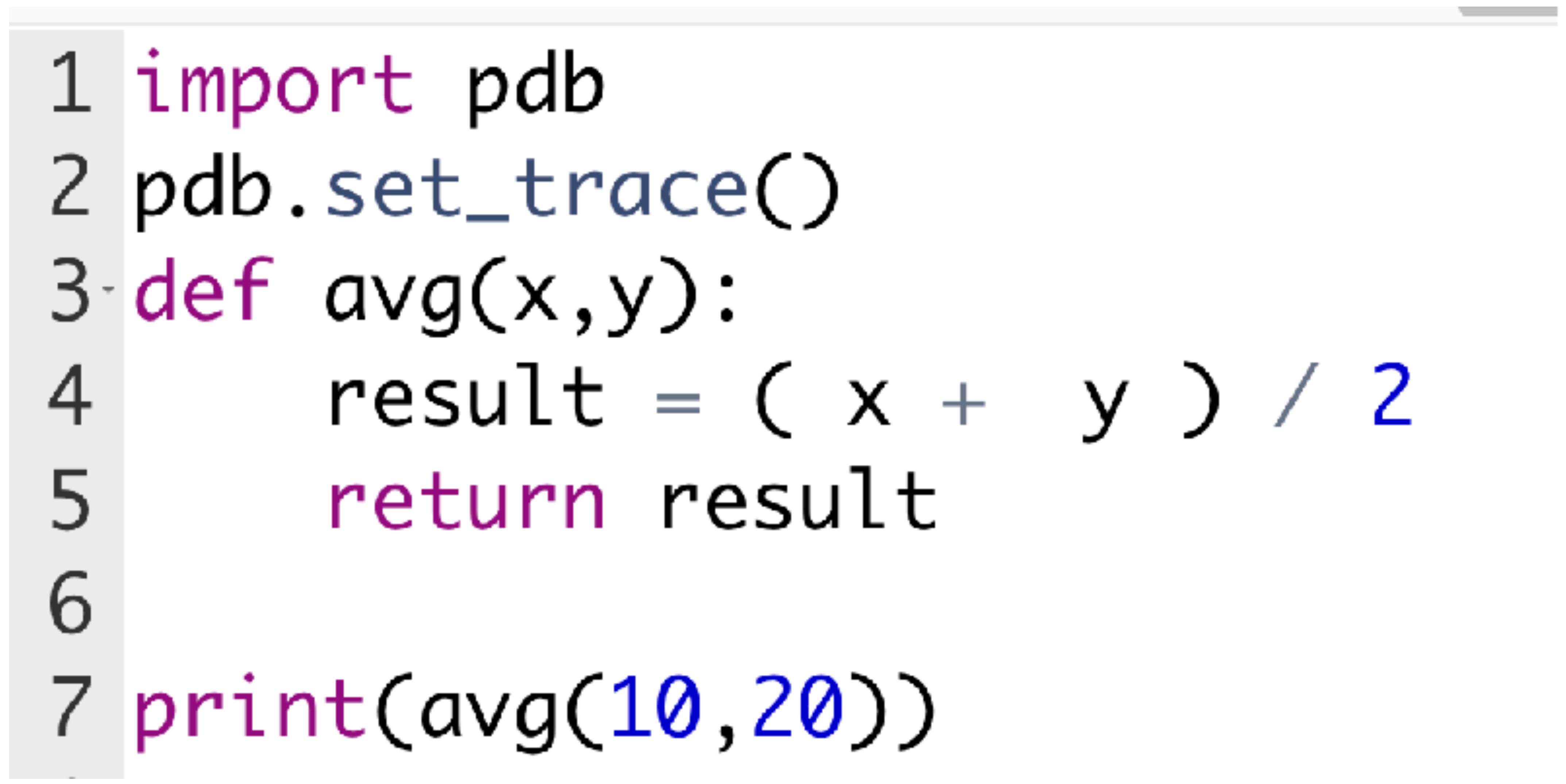
Instead of print(), we can pause the program and inspect variables interactively.



```python
import pdb
pdb.set_trace()
def avg(x,y):
    result = ( x +  y ) / 2
    return result

print(avg(10,20))
```

Using Python's Built-in Debugger (pdb)

Instead of print(), we can pause the program and inspect variables interactively.

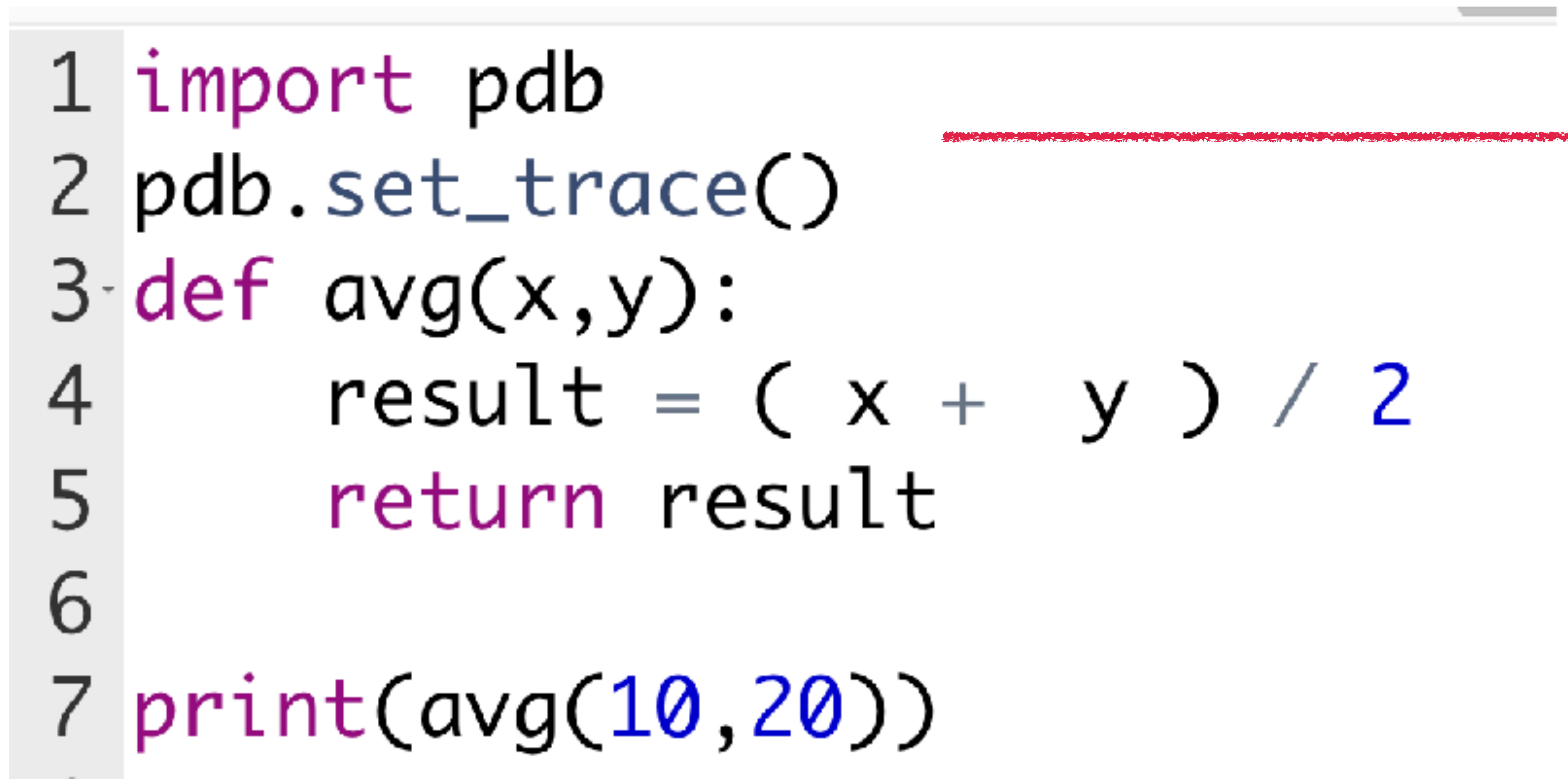


```python
import pdb
pdb.set_trace()
def avg(x,y):
    result = ( x +  y ) / 2
    return result

print(avg(10,20))
```

To tell python that you would like to be in the debug mode.

Using Python's Built-in Debugger (pdb)

Instead of print(), we can pause the program and inspect variables interactively.

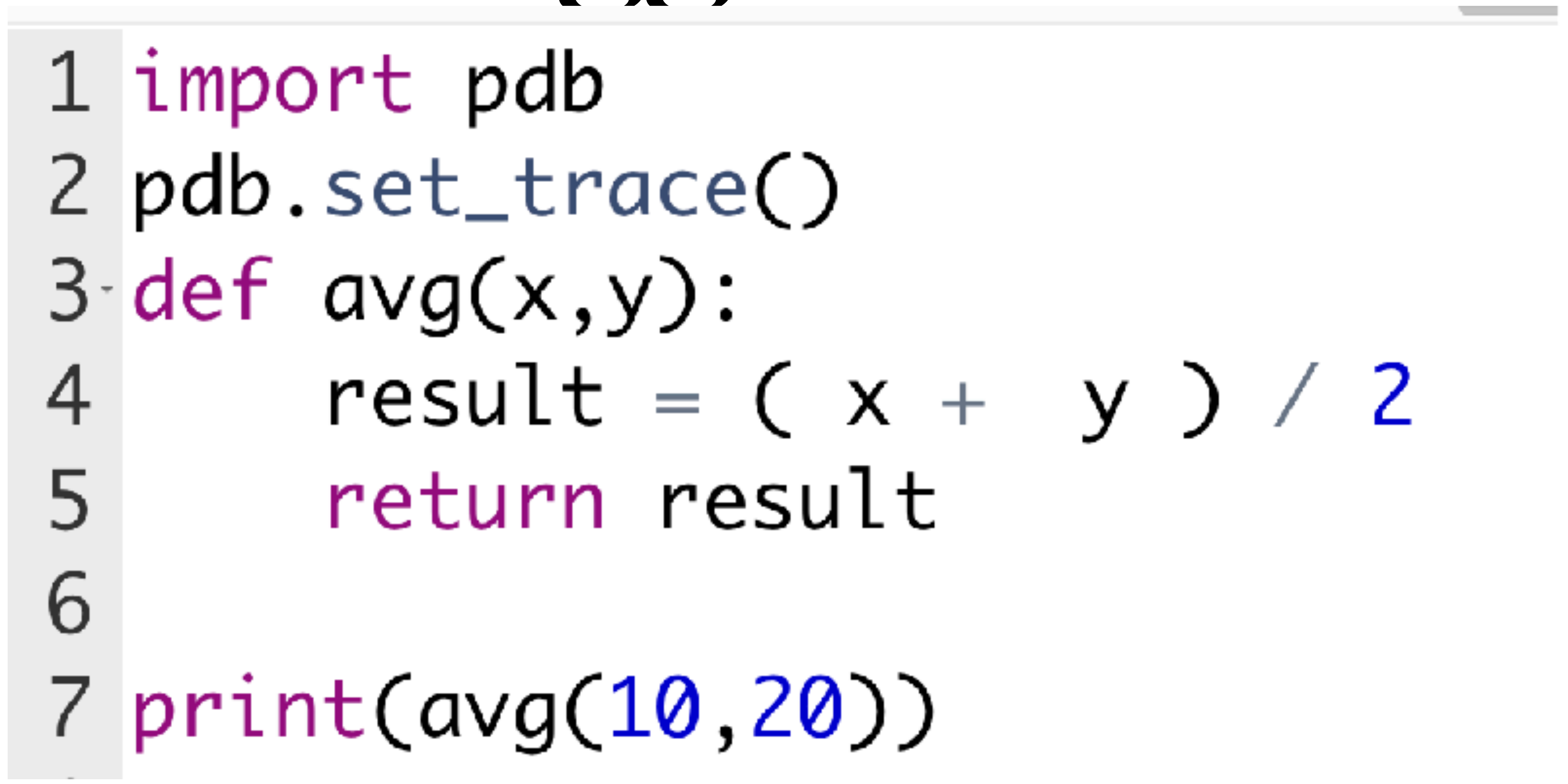


```python
import pdb
pdb.set_trace()
def avg(x,y):
    result = ( x +  y ) / 2
    return result

print(avg(10,20))
```

To tell python that you would like to be in the debug mode.

Debugger Pdb

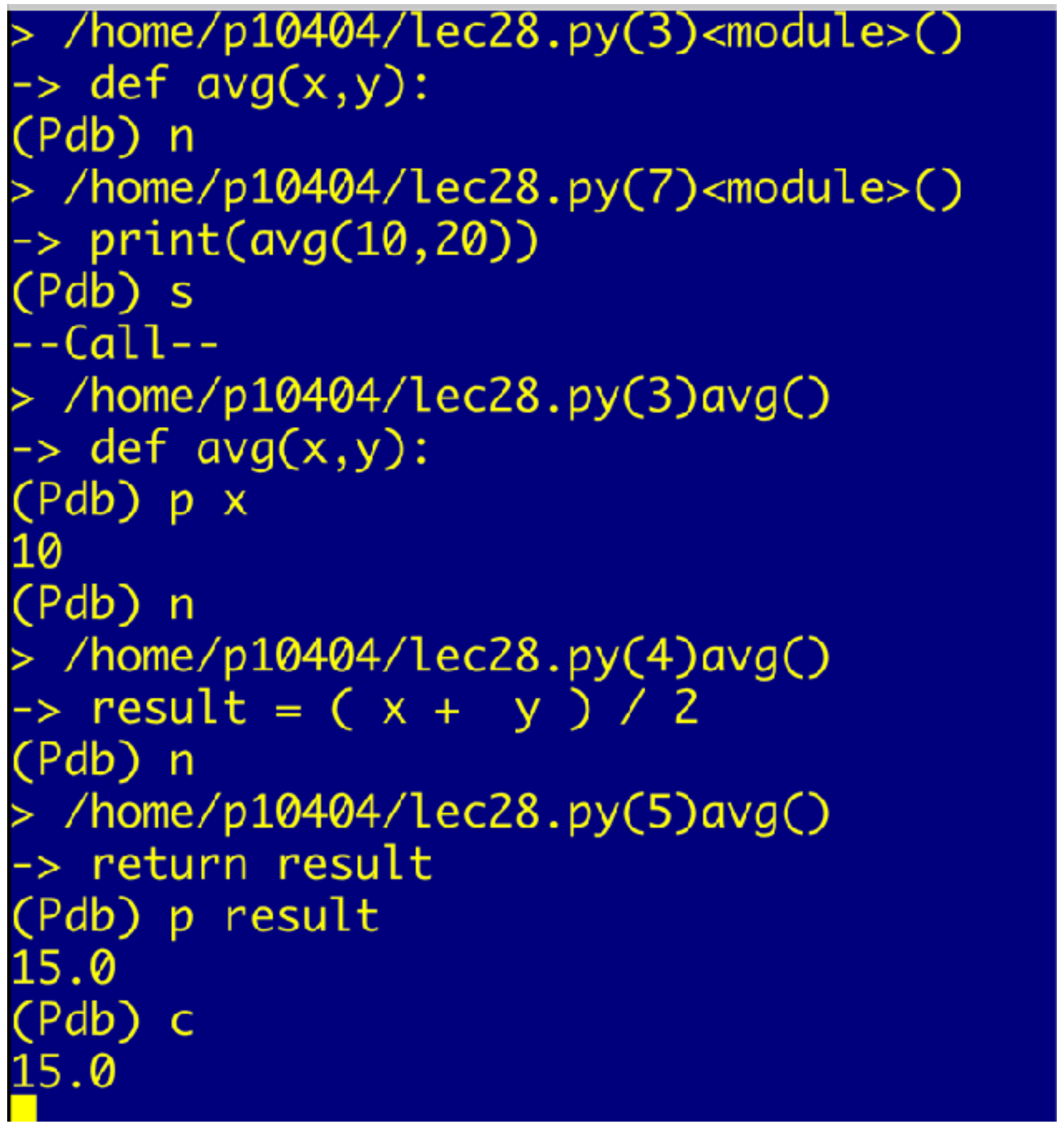
Command	Meaning
n	next line
s	step into function
c	continue execution
p variable	print the value
q	quit debugger



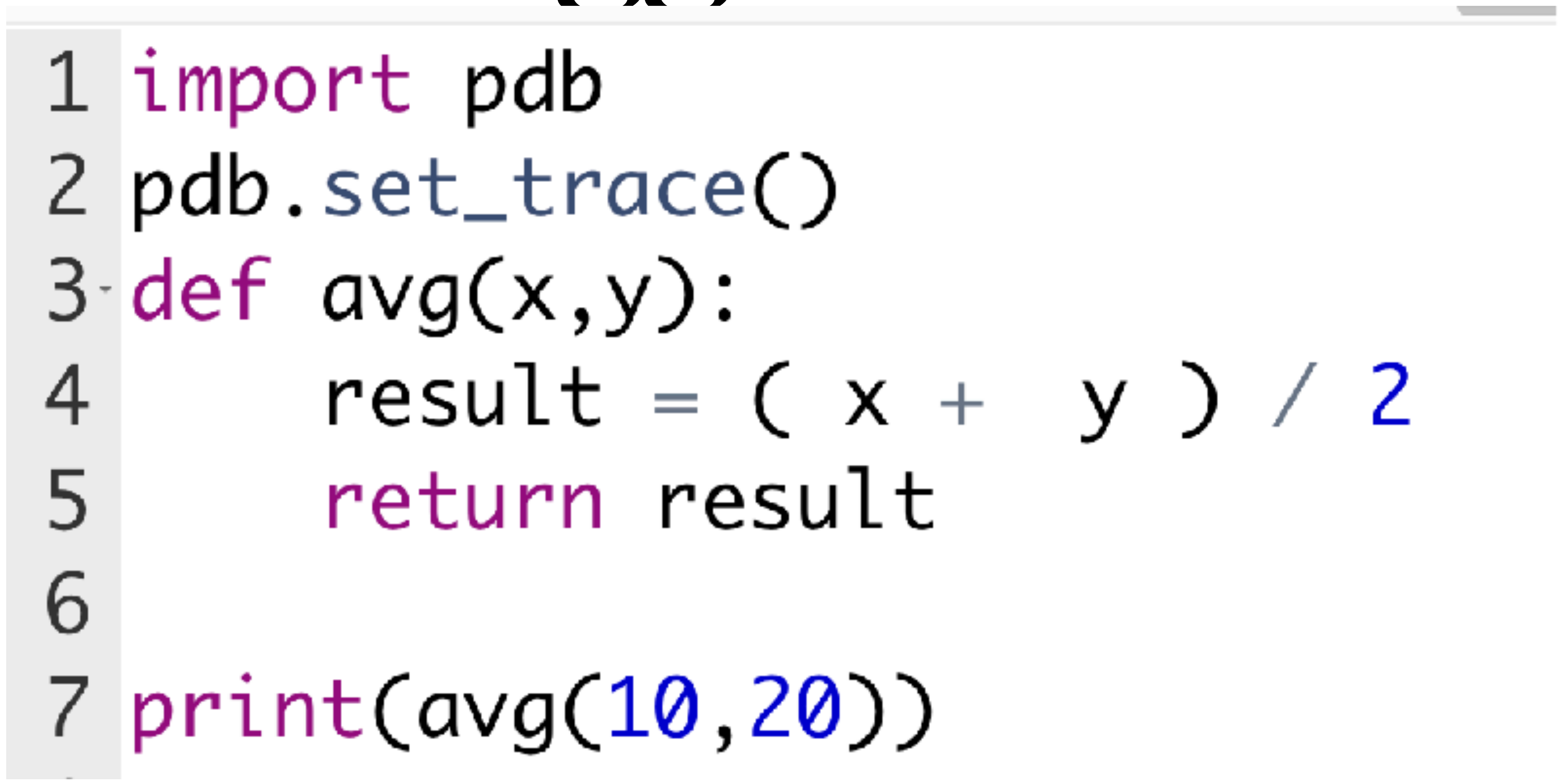
```python
import pdb
pdb.set_trace()
def avg(x,y):
    result = ( x +  y ) / 2
    return result

print(avg(10,20))
```

Debugger Pdb

Command	Meaning
n	next line
s	step into function
c	continue execution
p variable	print the value
q	quit debugger



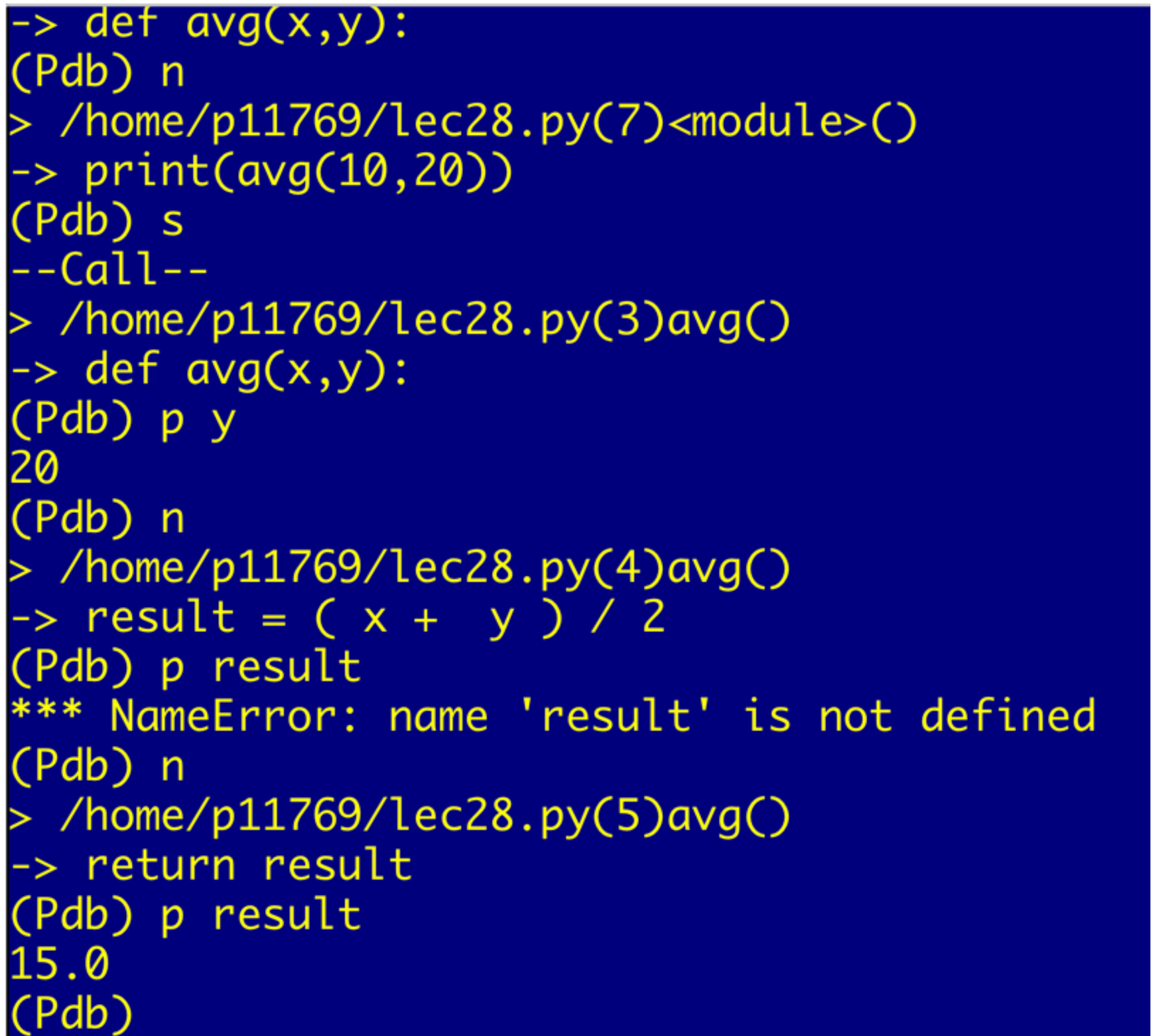
```
> /home/p10404/lec28.py(3)<module>()
-> def avg(x,y):
(Pdb) n
> /home/p10404/lec28.py(7)<module>()
-> print(avg(10,20))
(Pdb) s
--Call--
> /home/p10404/lec28.py(3)avg()
-> def avg(x,y):
(Pdb) p x
10
(Pdb) n
> /home/p10404/lec28.py(4)avg()
-> result = ( x +  y ) / 2
(Pdb) n
> /home/p10404/lec28.py(5)avg()
-> return result
(Pdb) p result
15.0
(Pdb) c
15.0
```



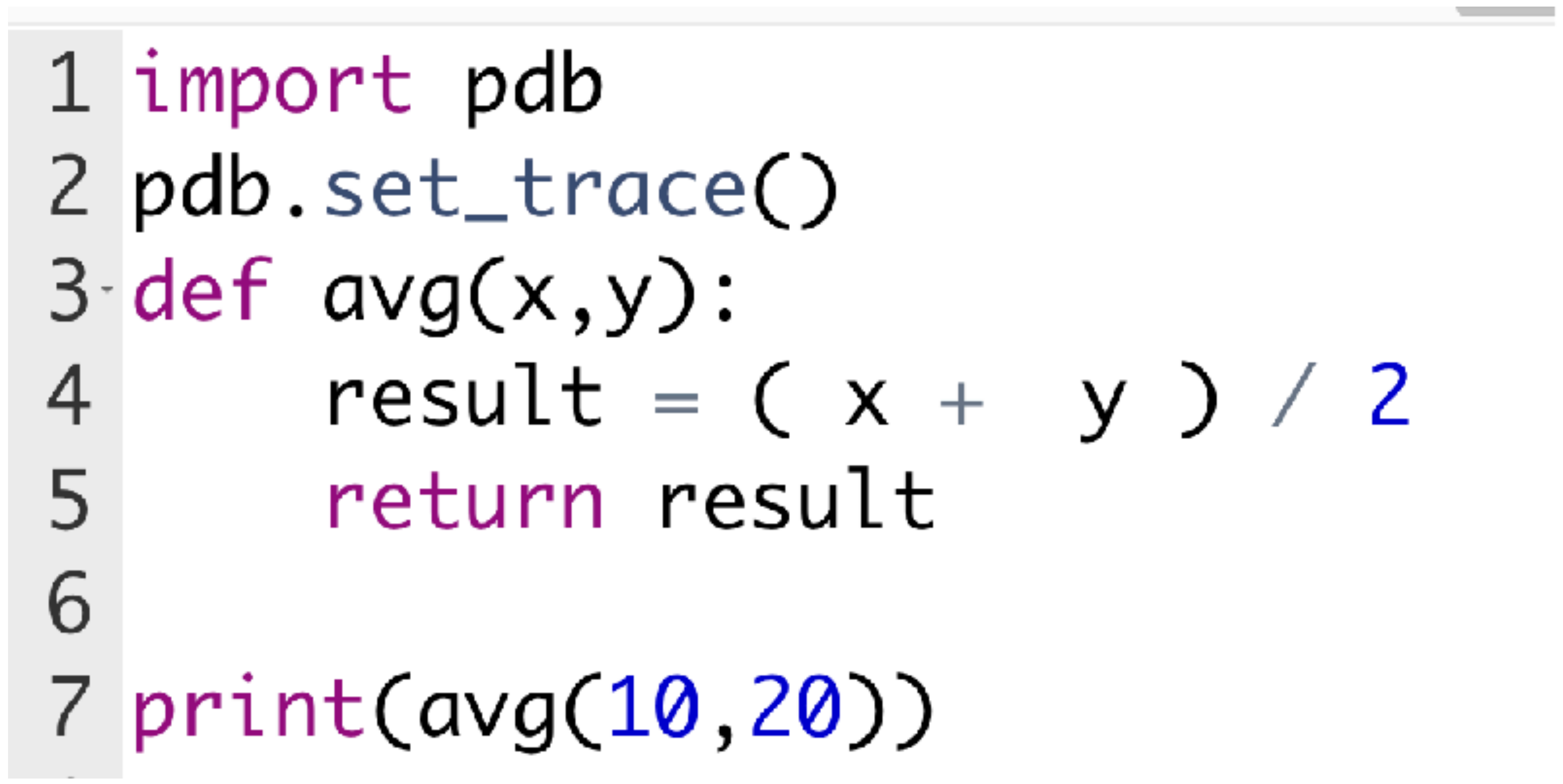
```
import pdb
pdb.set_trace()
def avg(x,y):
    result = ( x +  y ) / 2
    return result

print(avg(10,20))
```

Debugger Pdb

Command	Meaning
n	next line
s	step into function
c	continue execution
p variable	print the value
q	quit debugger



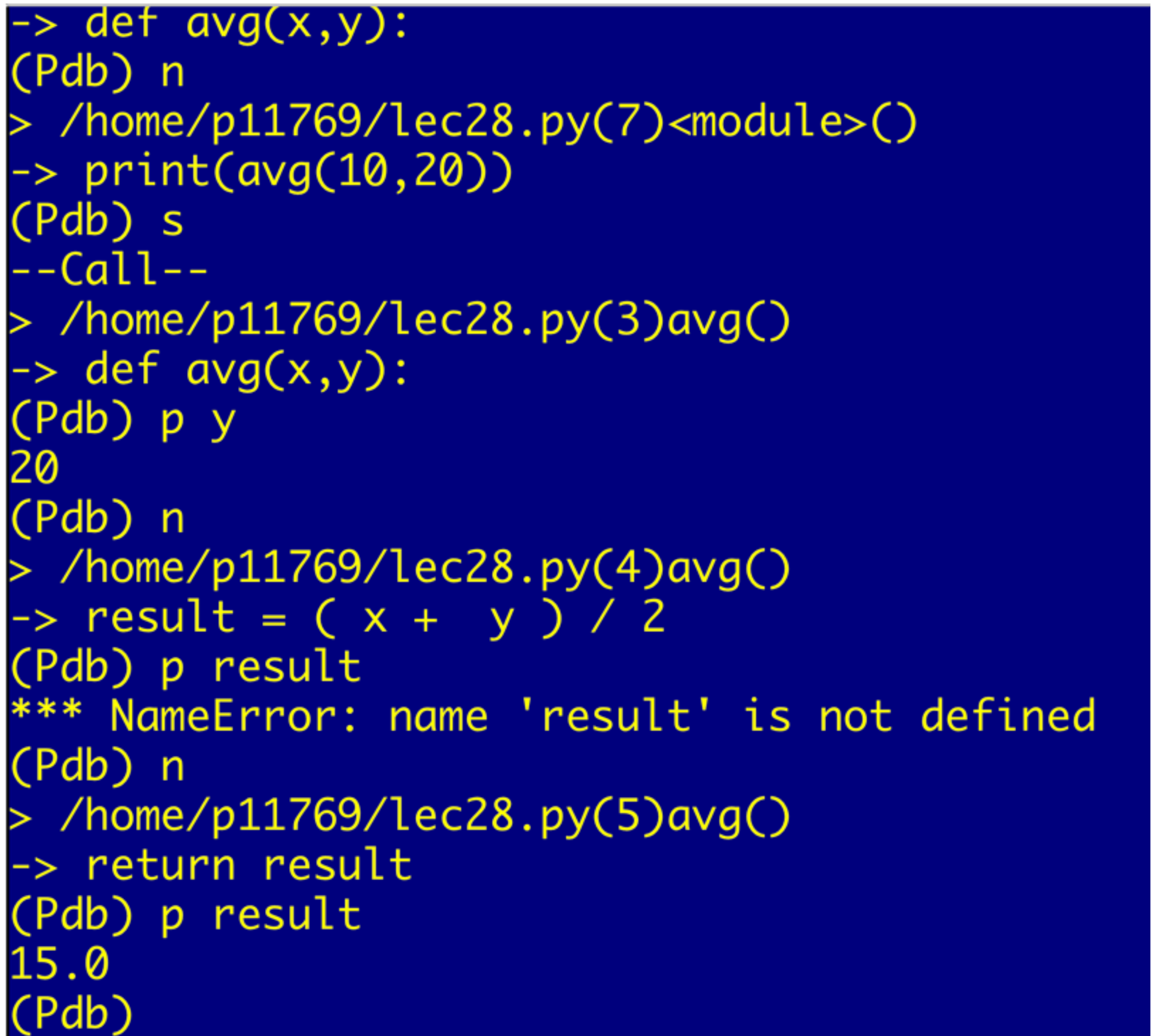
```
-> def avg(x,y):
(Pdb) n
> /home/p11769/lec28.py(7)<module>()
-> print(avg(10,20))
(Pdb) s
--Call--
> /home/p11769/lec28.py(3)avg()
-> def avg(x,y):
(Pdb) p y
20
(Pdb) n
> /home/p11769/lec28.py(4)avg()
-> result = ( x +  y ) / 2
(Pdb) p result
*** NameError: name 'result' is not defined
(Pdb) n
> /home/p11769/lec28.py(5)avg()
-> return result
(Pdb) p result
15.0
(Pdb)
```



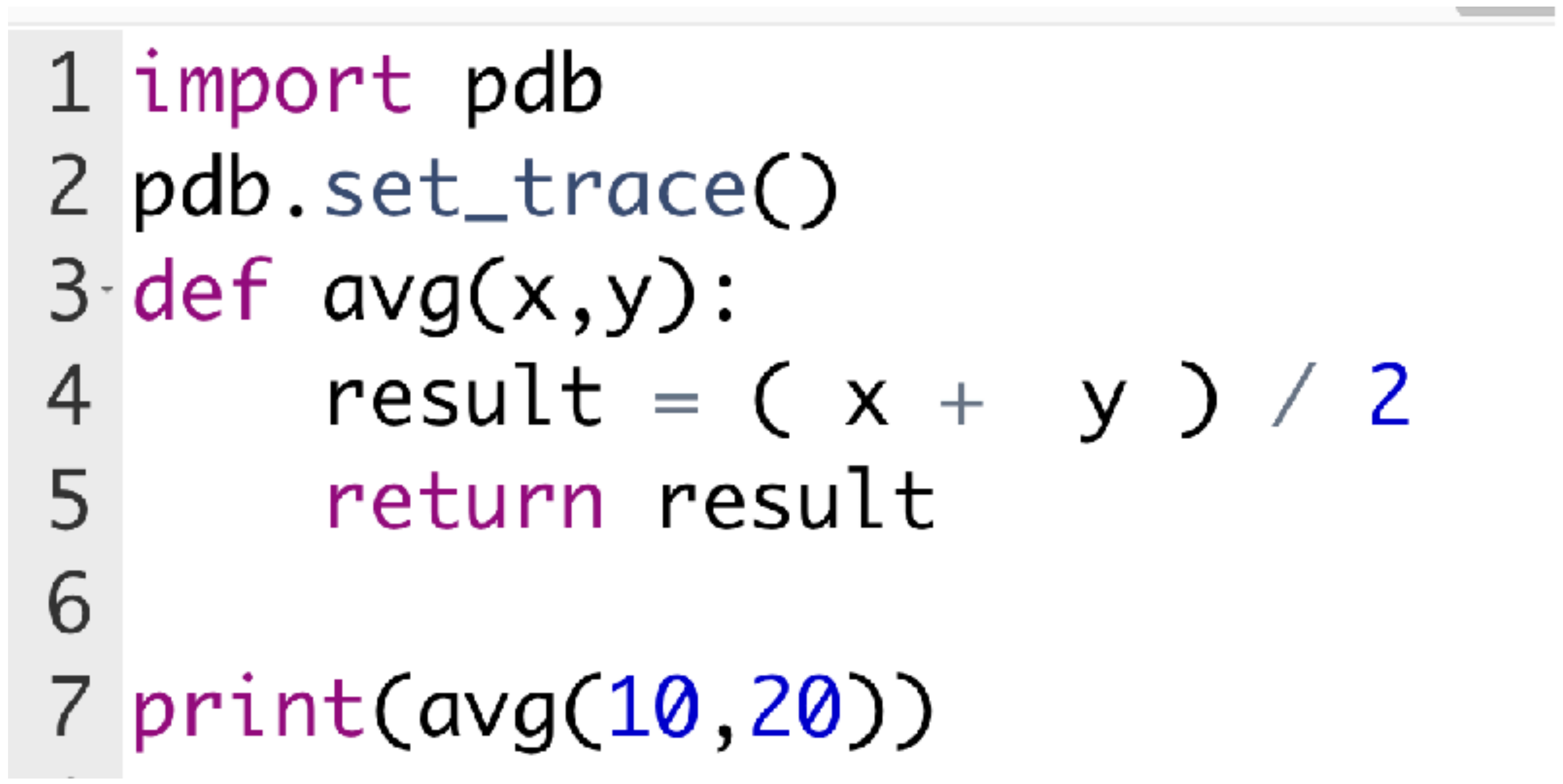
```
1 import pdb
2 pdb.set_trace()
3 def avg(x,y):
4      result = ( x +  y ) / 2
5      return result
6
7 print(avg(10,20))
```

Debugger Pdb

Command	Meaning
n	next line
s	step into function
c	continue execution
p variable	print the value
q	quit debugger



```
-> def avg(x,y):
(Pdb) n
> /home/p11769/lec28.py(7)<module>()
-> print(avg(10,20))
(Pdb) s
--Call--
> /home/p11769/lec28.py(3)avg()
-> def avg(x,y):
(Pdb) p y
20
(Pdb) n
> /home/p11769/lec28.py(4)avg()
-> result = ( x +  y ) / 2
(Pdb) p result
*** NameError: name 'result' is not defined
(Pdb) n
> /home/p11769/lec28.py(5)avg()
-> return result
(Pdb) p result
15.0
(Pdb)
```

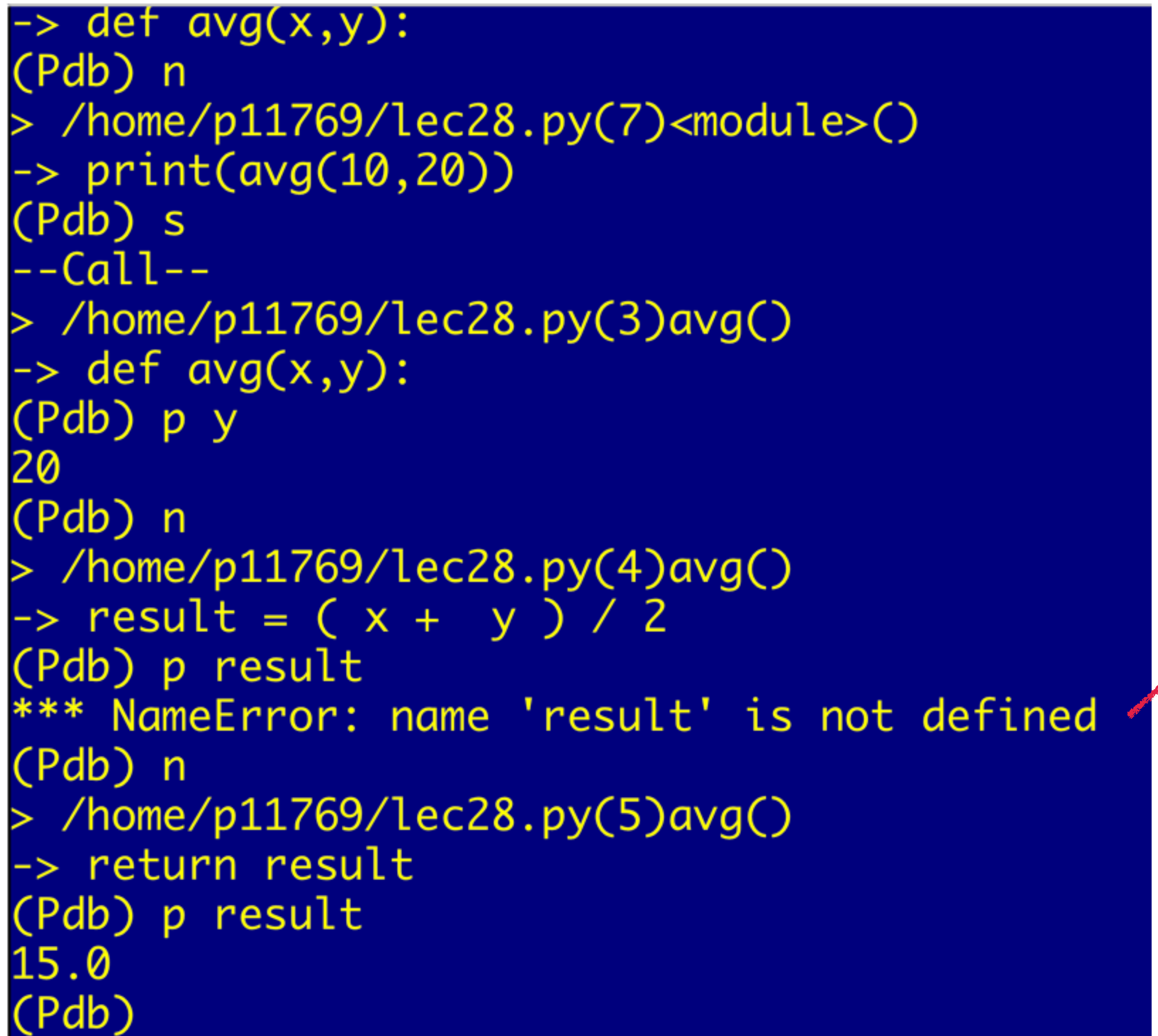


```
1 import pdb
2 pdb.set_trace()
3 def avg(x,y):
4     result = ( x +  y ) / 2
5     return result
6
7 print(avg(10,20))
```

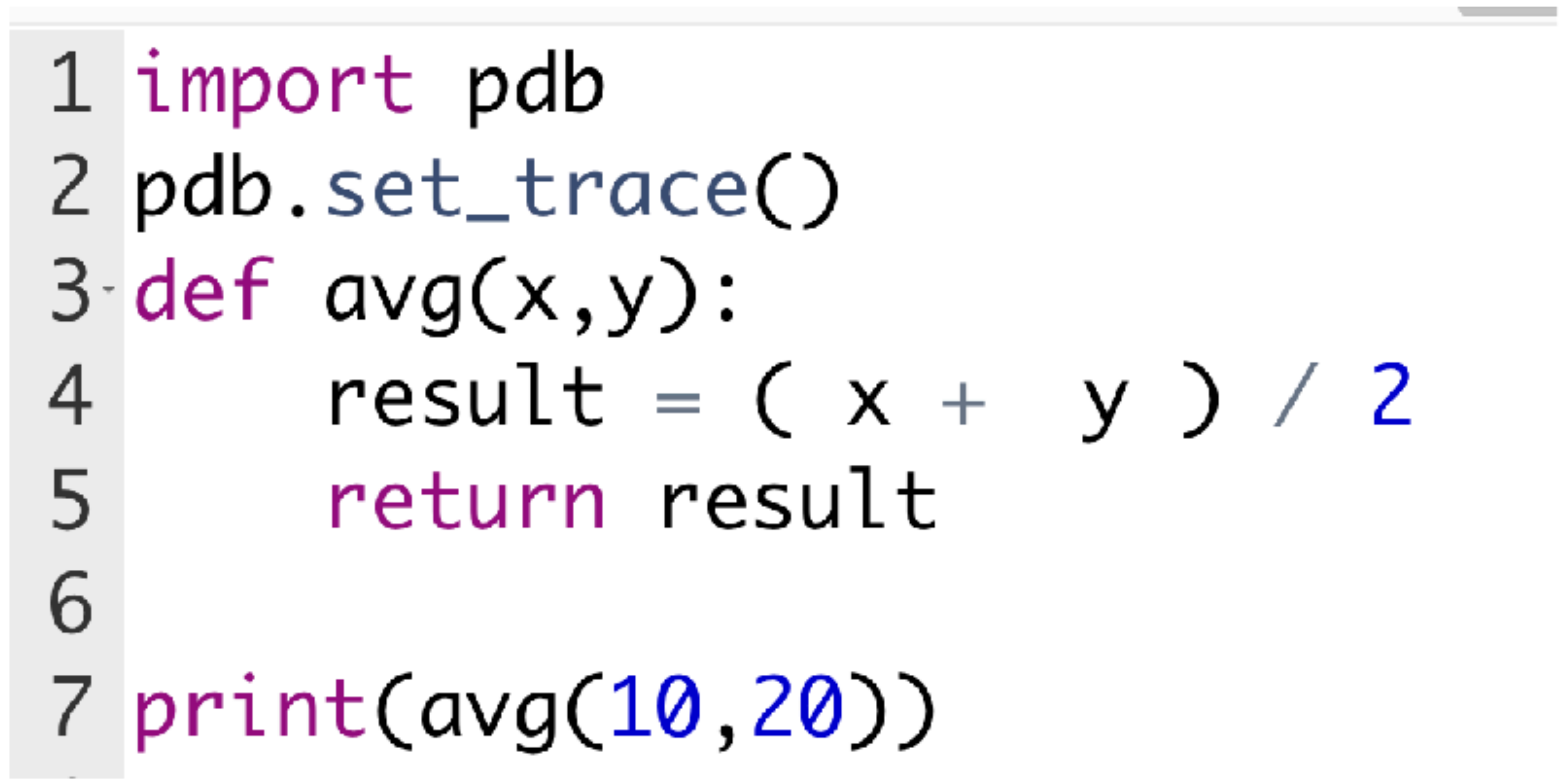
Notice the `name` error — n tells the cursor is at this line, it didn't execute the line, hence the error.

Debugger Pdb

Command	Meaning
n	next line
s	step into function
c	continue execution
p variable	print the value
q	quit debugger



```
-> def avg(x,y):
(Pdb) n
> /home/p11769/lec28.py(7)<module>()
-> print(avg(10,20))
(Pdb) s
--Call--
> /home/p11769/lec28.py(3)avg()
-> def avg(x,y):
(Pdb) p y
20
(Pdb) n
> /home/p11769/lec28.py(4)avg()
-> result = ( x +  y ) / 2
(Pdb) p result
*** NameError: name 'result' is not defined
(Pdb) n
> /home/p11769/lec28.py(5)avg()
-> return result
(Pdb) p result
15.0
(Pdb)
```

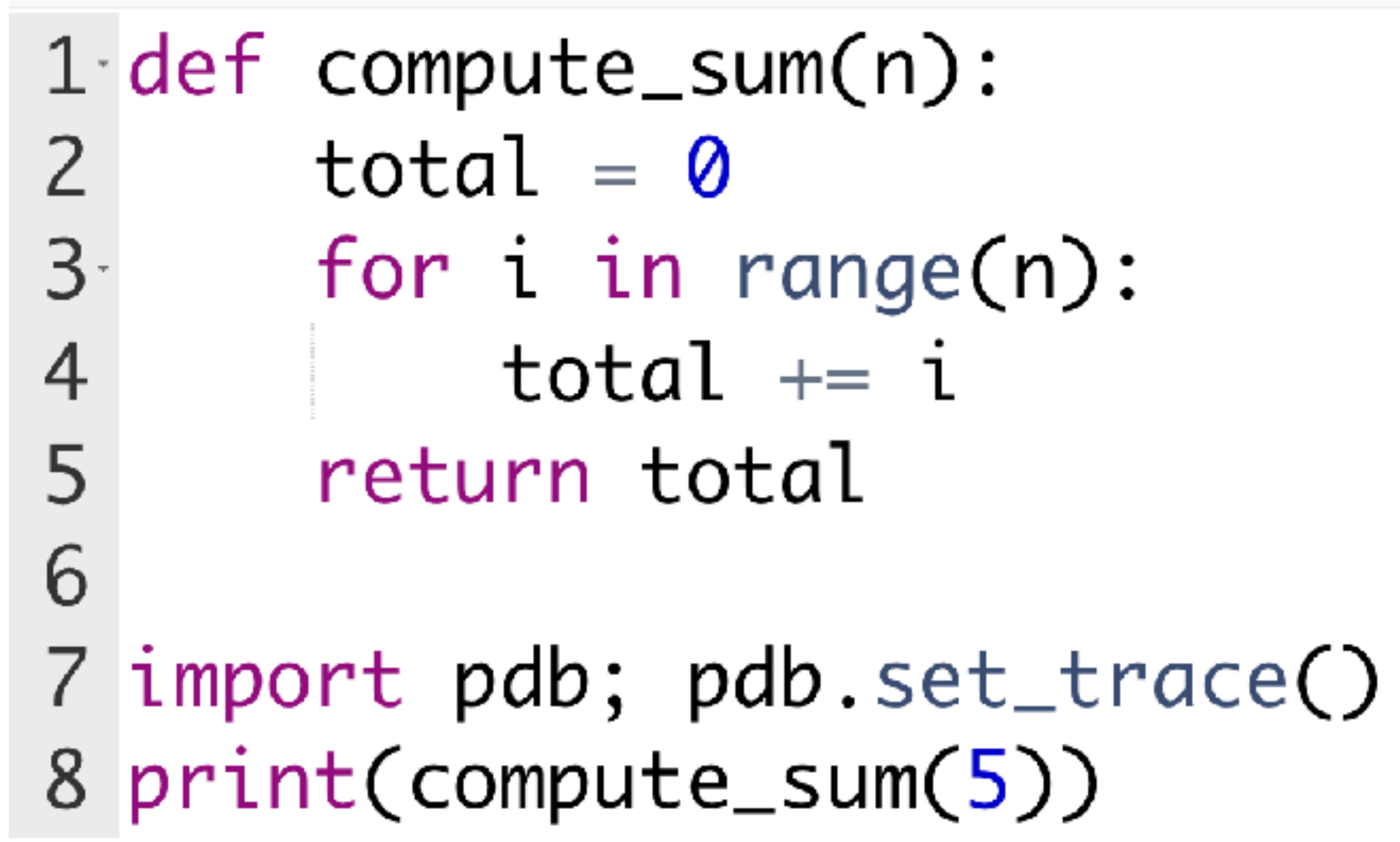


```
1 import pdb
2 pdb.set_trace()
3 def avg(x,y):
4     result = ( x +  y ) / 2
5     return result
6
7 print(avg(10,20))
```

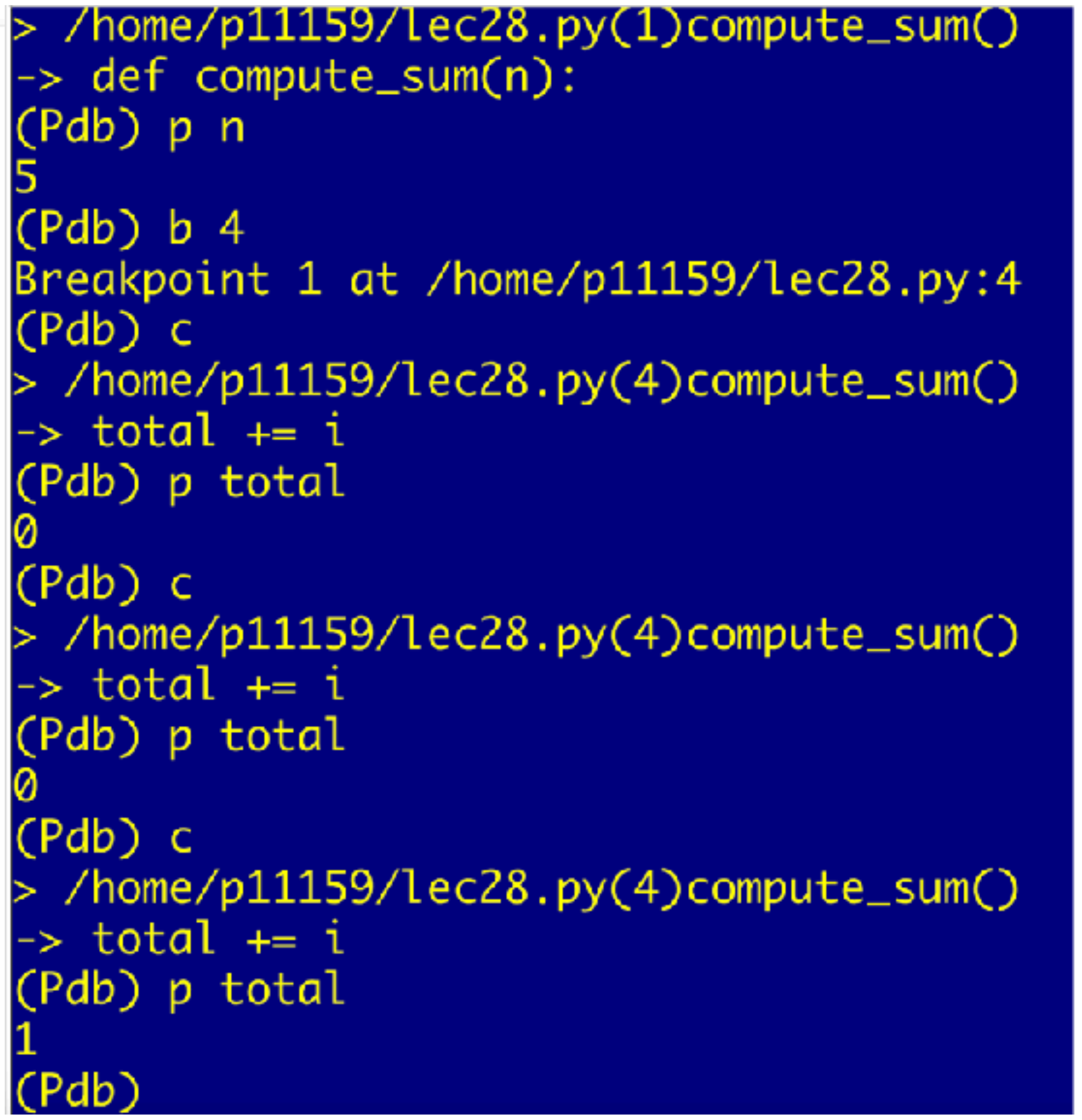
Notice the `name` error — n tells the cursor is at this line, it didn’t execute the line, hence the error.

We can define breakpoints as well in pdb

A breakpoint is a place in your program where Python will pause execution, so you can inspect what's happening at that moment — values of variables, flow of control, etc.



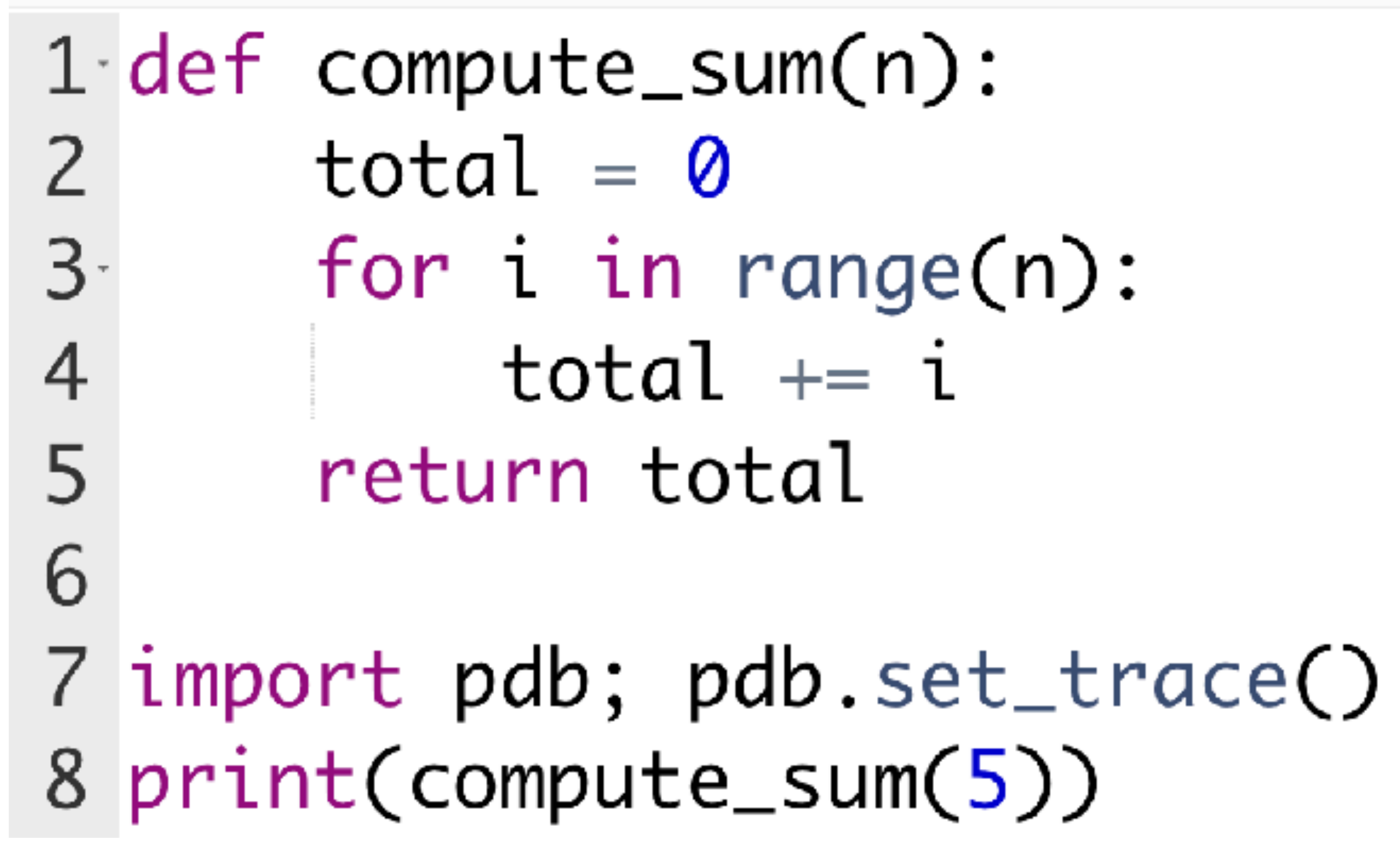
```python
def compute_sum(n):
    total = 0
    for i in range(n):
        total += i
    return total

import pdb; pdb.set_trace()
print(compute_sum(5))
```



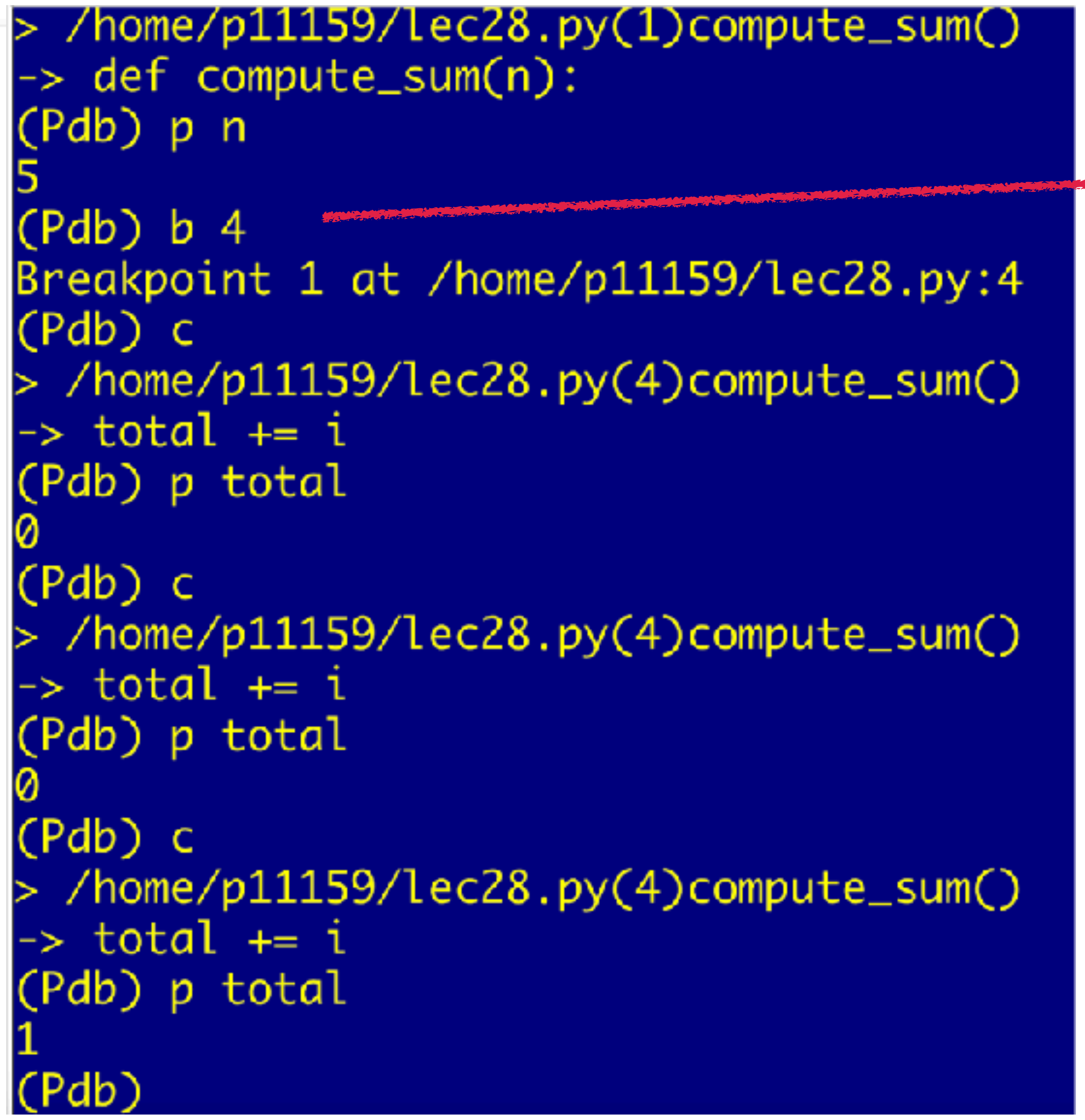
```
> /home/p11159/lec28.py(1)compute_sum()
-> def compute_sum(n):
(Pdb) p n
5
(Pdb) b 4
Breakpoint 1 at /home/p11159/lec28.py:4
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
0
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
0
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
1
(Pdb)
```

We can define breakpoints as well in pdb

A breakpoint is a place in your program where Python will pause execution, so you can inspect what's happening at that moment — values of variables, flow of control, etc.



```
1 def compute_sum(n):
2     total = 0
3     for i in range(n):
4         total += i
5     return total
6
7 import pdb; pdb.set_trace()
8 print(compute_sum(5))
```

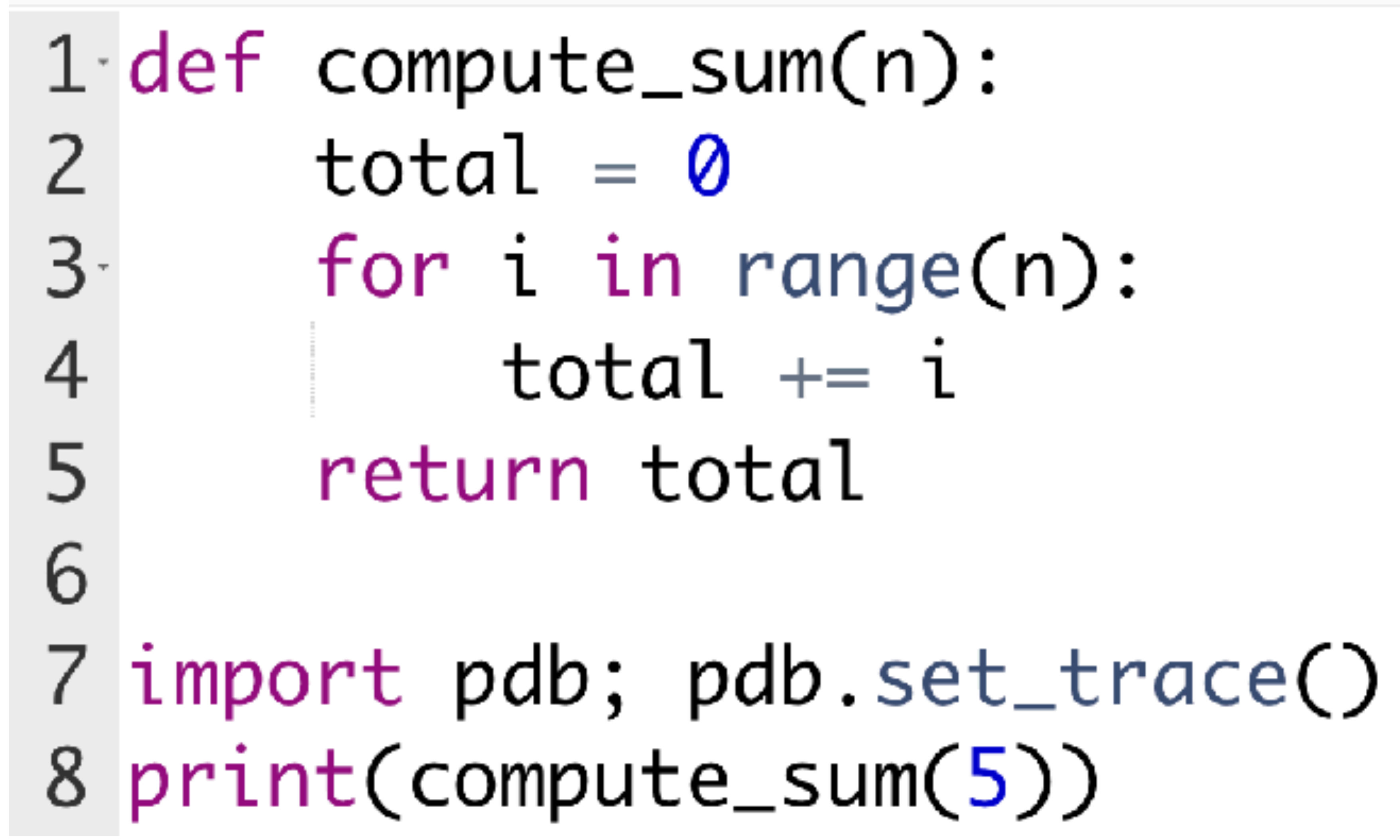


```
> /home/p11159/lec28.py(1)compute_sum()
-> def compute_sum(n):
(Pdb) p n
5
(Pdb) b 4
Breakpoint 1 at /home/p11159/lec28.py:4
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
0
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
0
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
1
(Pdb)
```

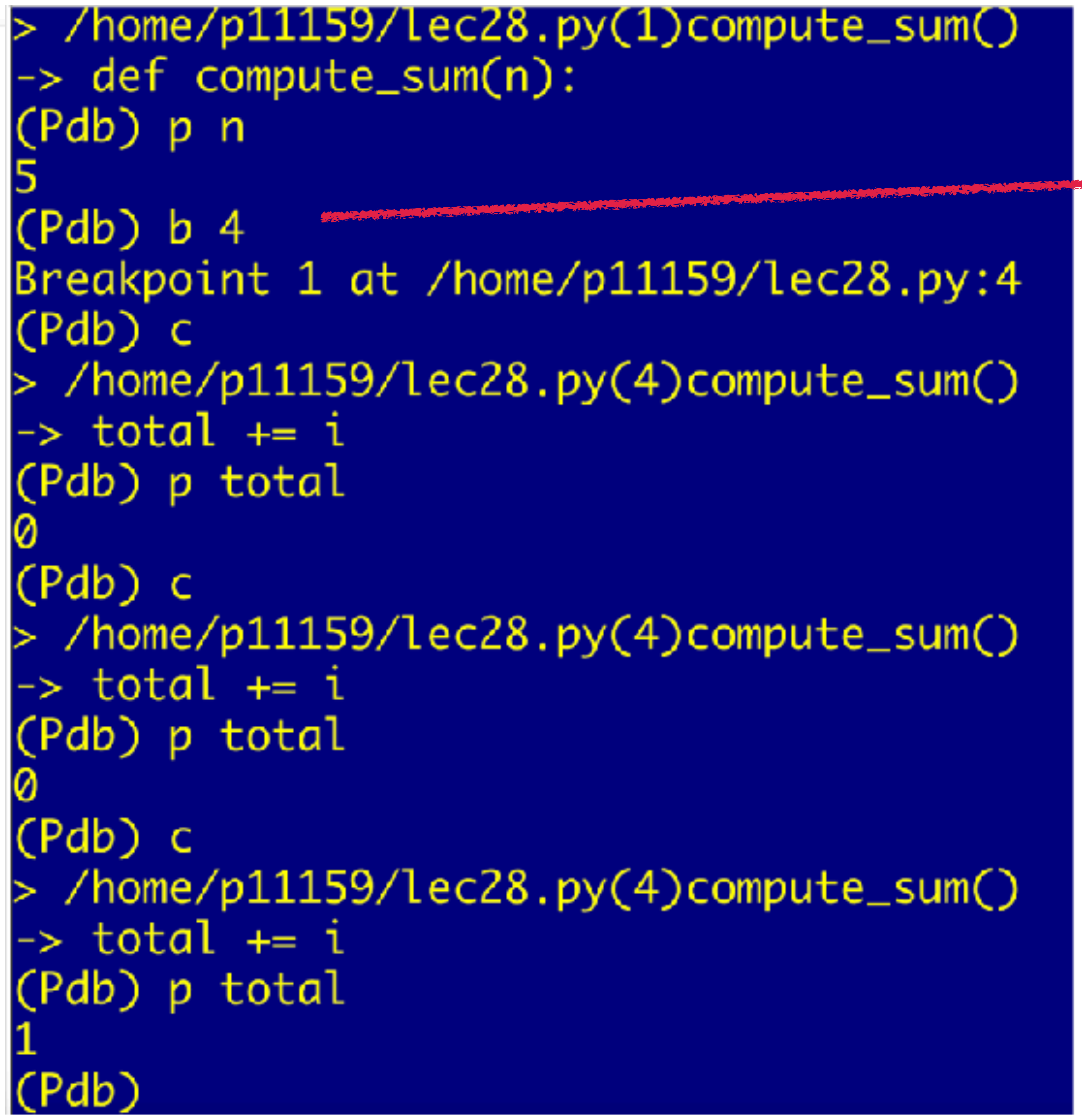
b <line number> to set the breakpoint

We can define breakpoints as well in pdb

A breakpoint is a place in your program where Python will pause execution, so you can inspect what's happening at that moment — values of variables, flow of control, etc.



```
1 def compute_sum(n):
2     total = 0
3     for i in range(n):
4         total += i
5     return total
6
7 import pdb; pdb.set_trace()
8 print(compute_sum(5))
```



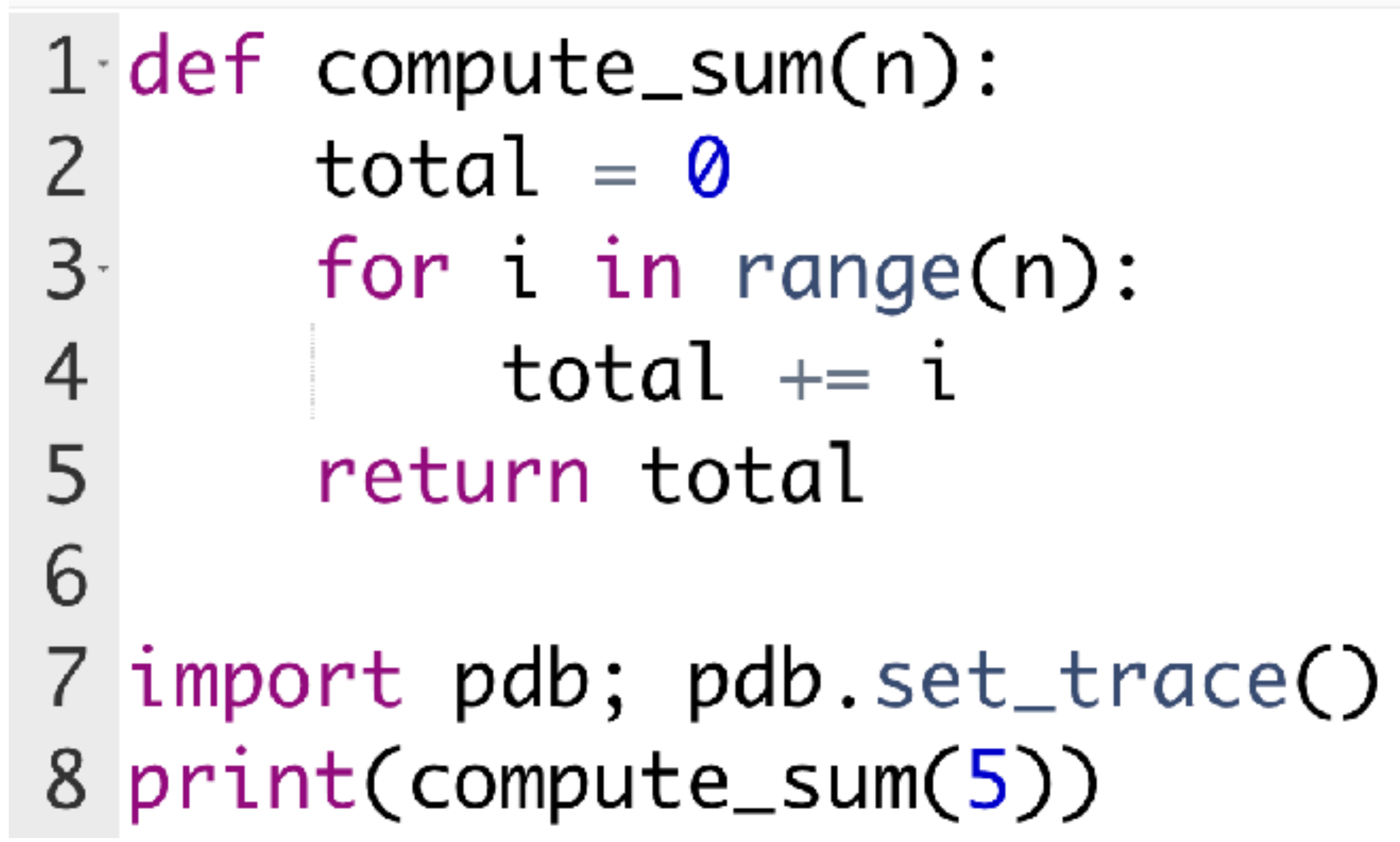
```
> /home/p11159/lec28.py(1)compute_sum()
-> def compute_sum(n):
(Pdb) p n
5
(Pdb) b 4
Breakpoint 1 at /home/p11159/lec28.py:4
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
0
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
0
(Pdb) c
> /home/p11159/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
1
(Pdb)
```

b <line number> to set the breakpoint

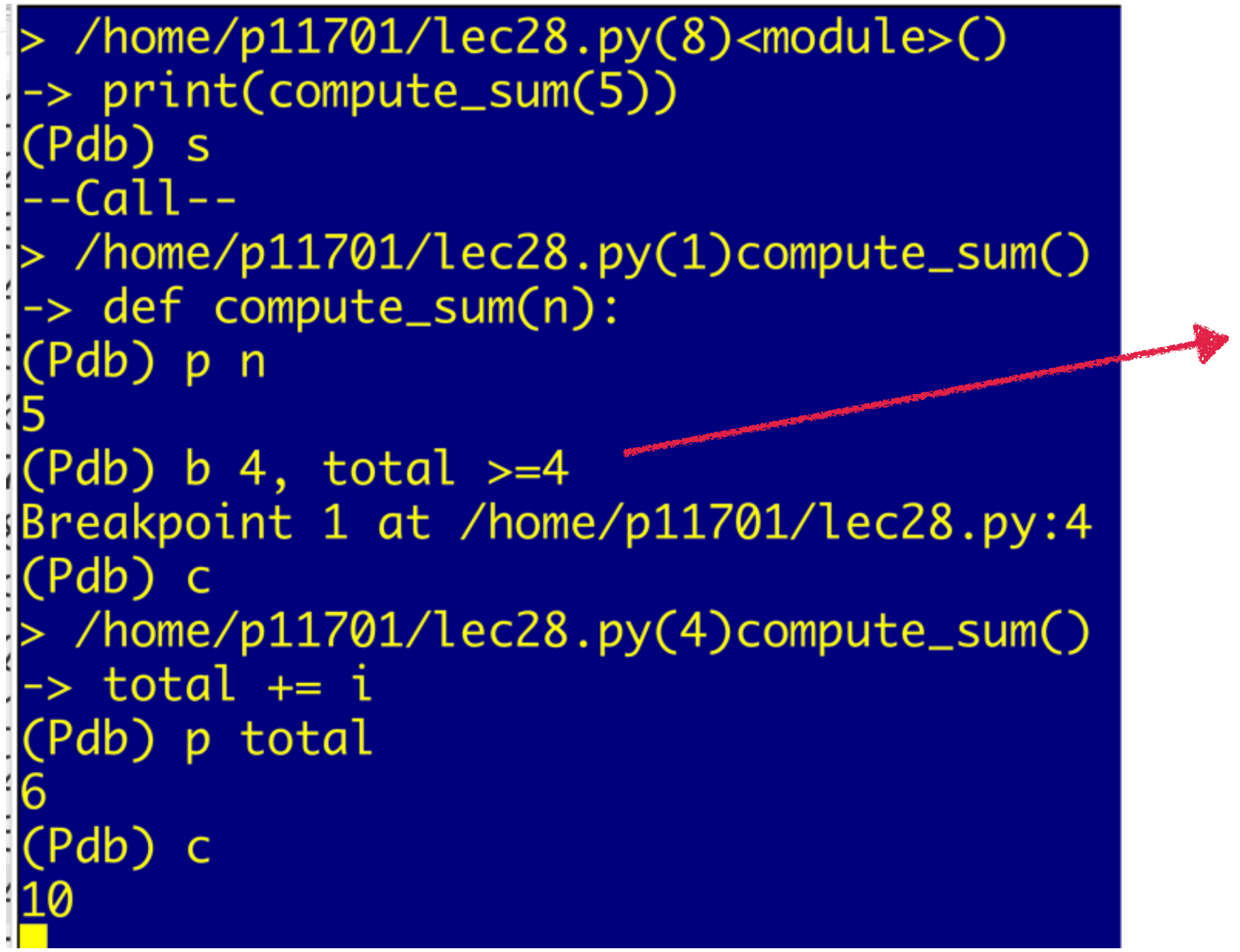
Notice everytime flow reaches line 4, it stops

We can define breakpoints as well in pdb

A breakpoint is a place in your program where Python will pause execution, so you can inspect what's happening at that moment — values of variables, flow of control, etc.



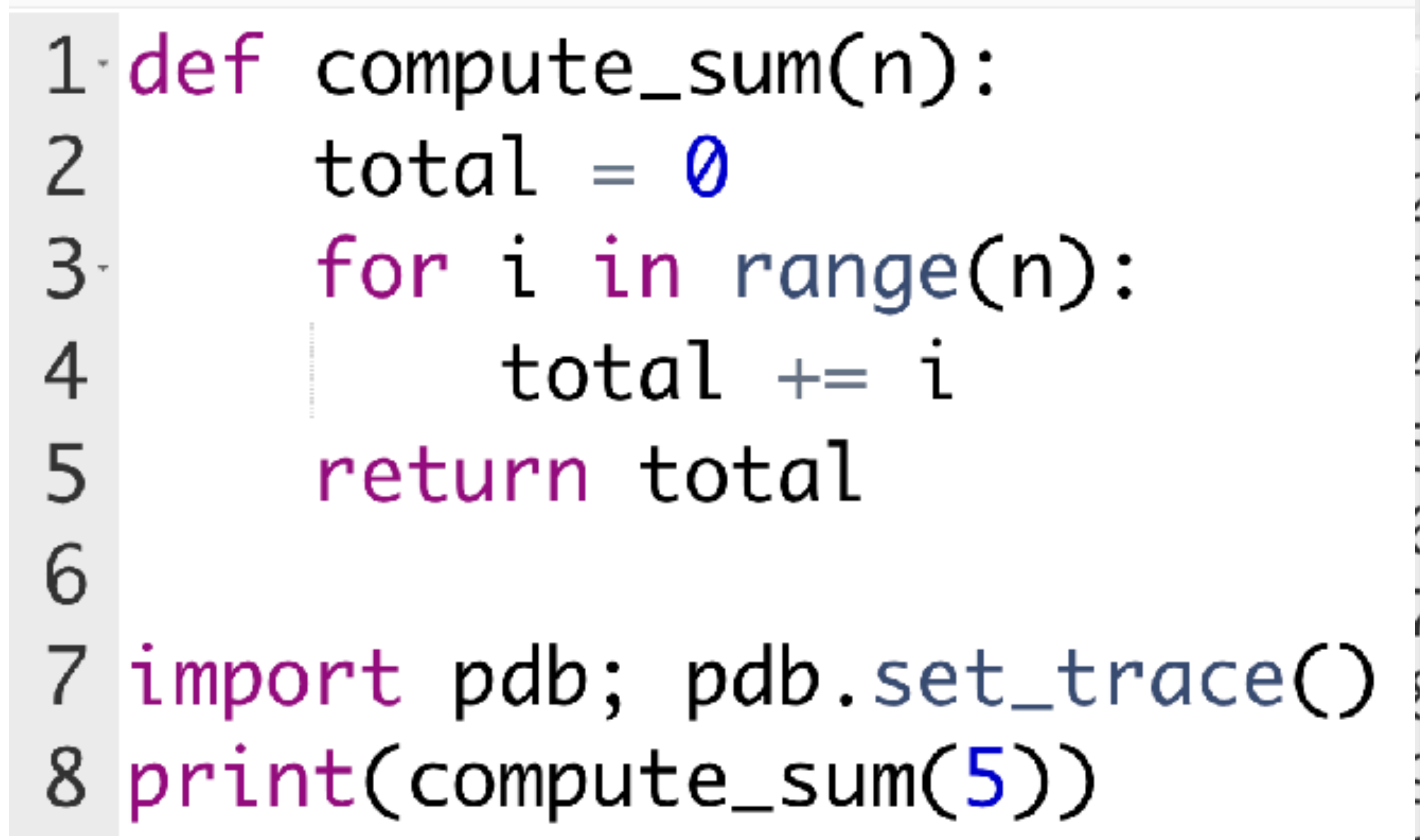
```python
def compute_sum(n):
    total = 0
    for i in range(n):
        total += i
    return total

import pdb; pdb.set_trace()
print(compute_sum(5))
```



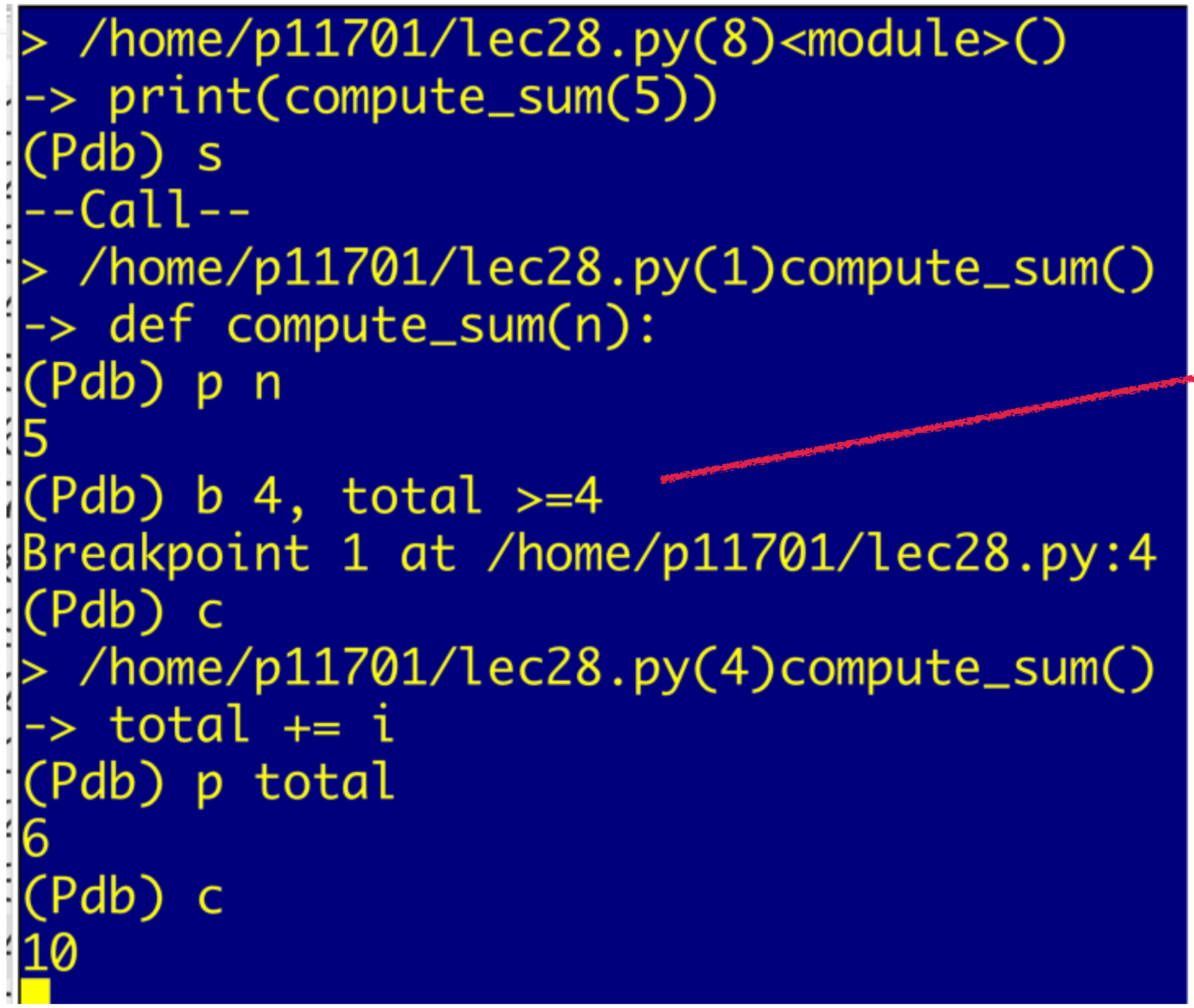
```
> /home/p11701/lec28.py(8)<module>()
-> print(compute_sum(5))
(Pdb) s
--Call--
> /home/p11701/lec28.py(1)compute_sum()
-> def compute_sum(n):
(Pdb) p n
5
(Pdb) b 4, total >=4
Breakpoint 1 at /home/p11701/lec28.py:4
(Pdb) c
> /home/p11701/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
6
(Pdb) c
10
```

We can define breakpoints as well in pdb

A breakpoint is a place in your program where Python will pause execution, so you can inspect what's happening at that moment — values of variables, flow of control, etc.



```python
def compute_sum(n):
    total = 0
    for i in range(n):
        total += i
    return total

import pdb; pdb.set_trace()
print(compute_sum(5))
```



```
> /home/p11701/lec28.py(8)<module>()
-> print(compute_sum(5))
(Pdb) s
--Call--
> /home/p11701/lec28.py(1)compute_sum()
-> def compute_sum(n):
(Pdb) p n
5
(Pdb) b 4, total >=4
Breakpoint 1 at /home/p11701/lec28.py:4
(Pdb) c
> /home/p11701/lec28.py(4)compute_sum()
-> total += i
(Pdb) p total
6
(Pdb) c
10
```

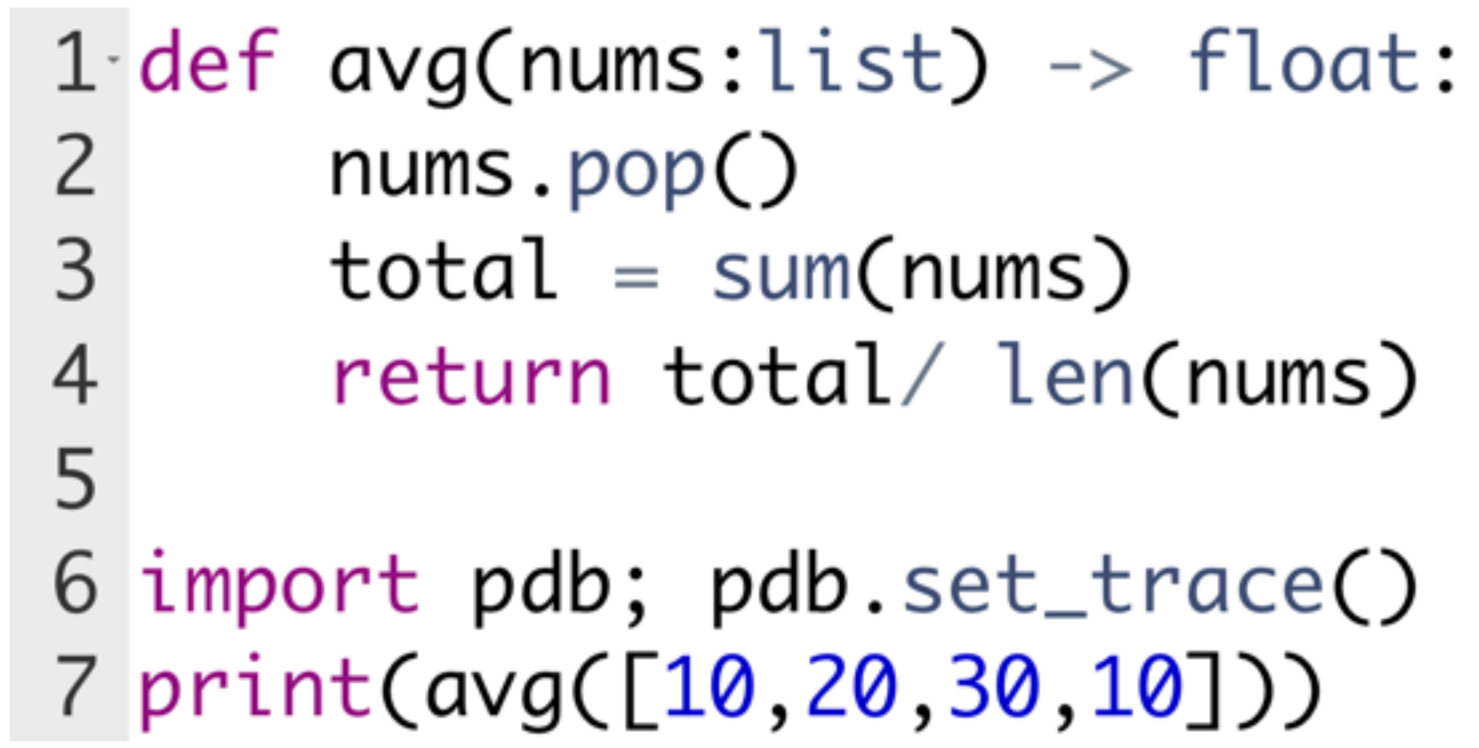
b <line number> , <condition>

breakpoint when condition is met.

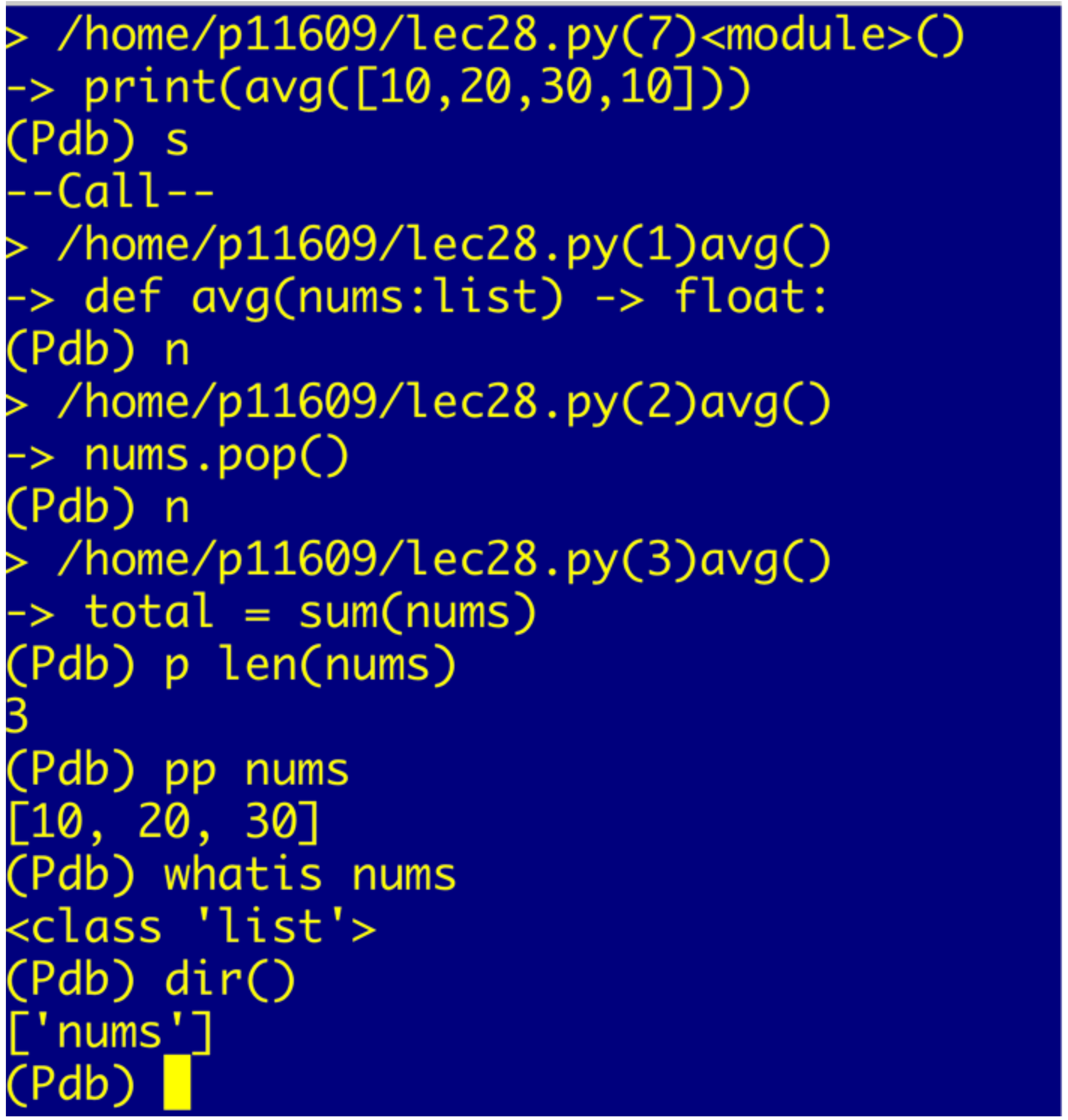
Breakpoints in pdb

Command	What it does
b 15	Sets a breakpoint at **line 15** of the current file
b myfunc	Breaks at the first line of function `myfunc`
b 30, x > 10	Conditional breakpoint — stops only when `x > 10`
b	Lists all current breakpoints
cl 15	Clear breakpoint at line 15
enable	Enable all breakpoints
disable	Temporarily disables breakpoints

More examples on pdb

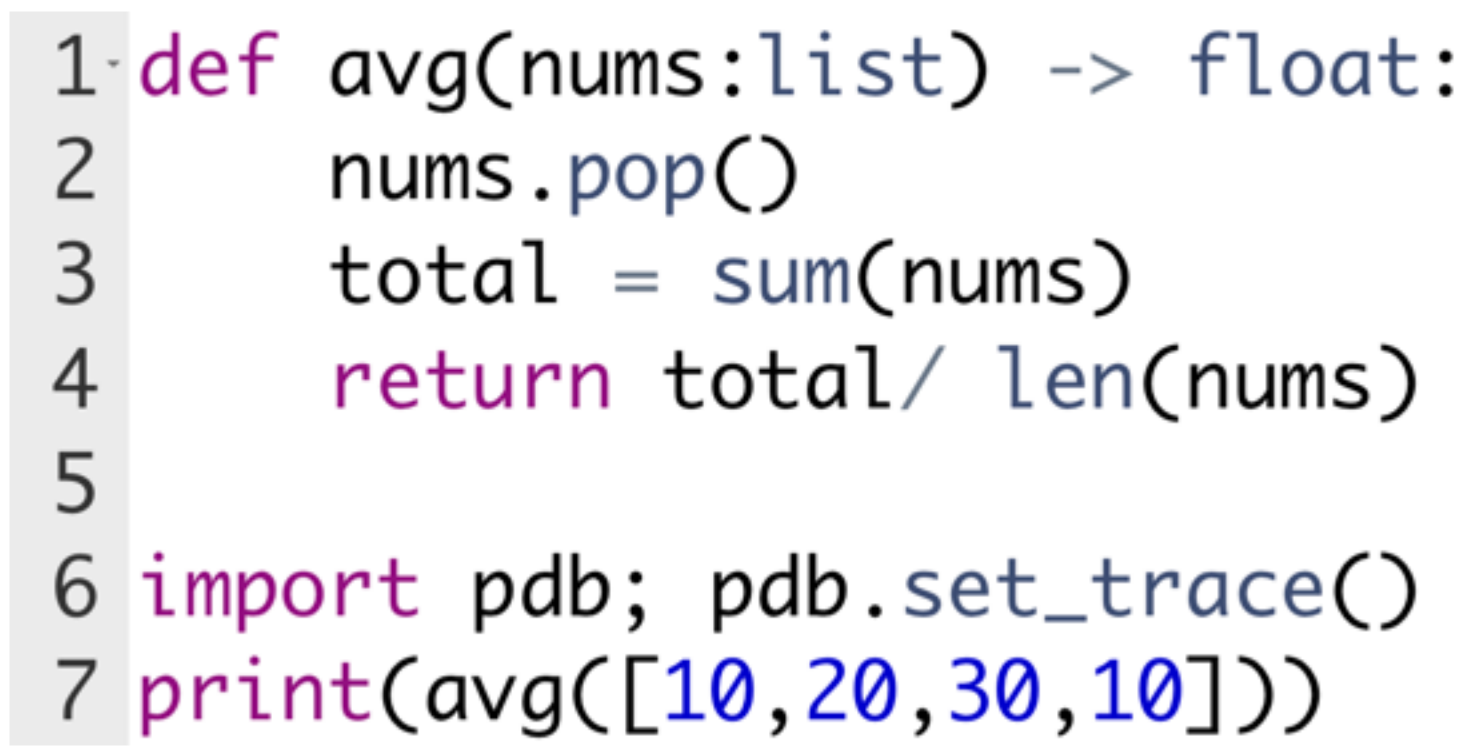


```
1 def avg(nums:list) -> float:
2     nums.pop()
3     total = sum(nums)
4     return total/ len(nums)
5
6 import pdb; pdb.set_trace()
7 print(avg([10,20,30,10]))
```

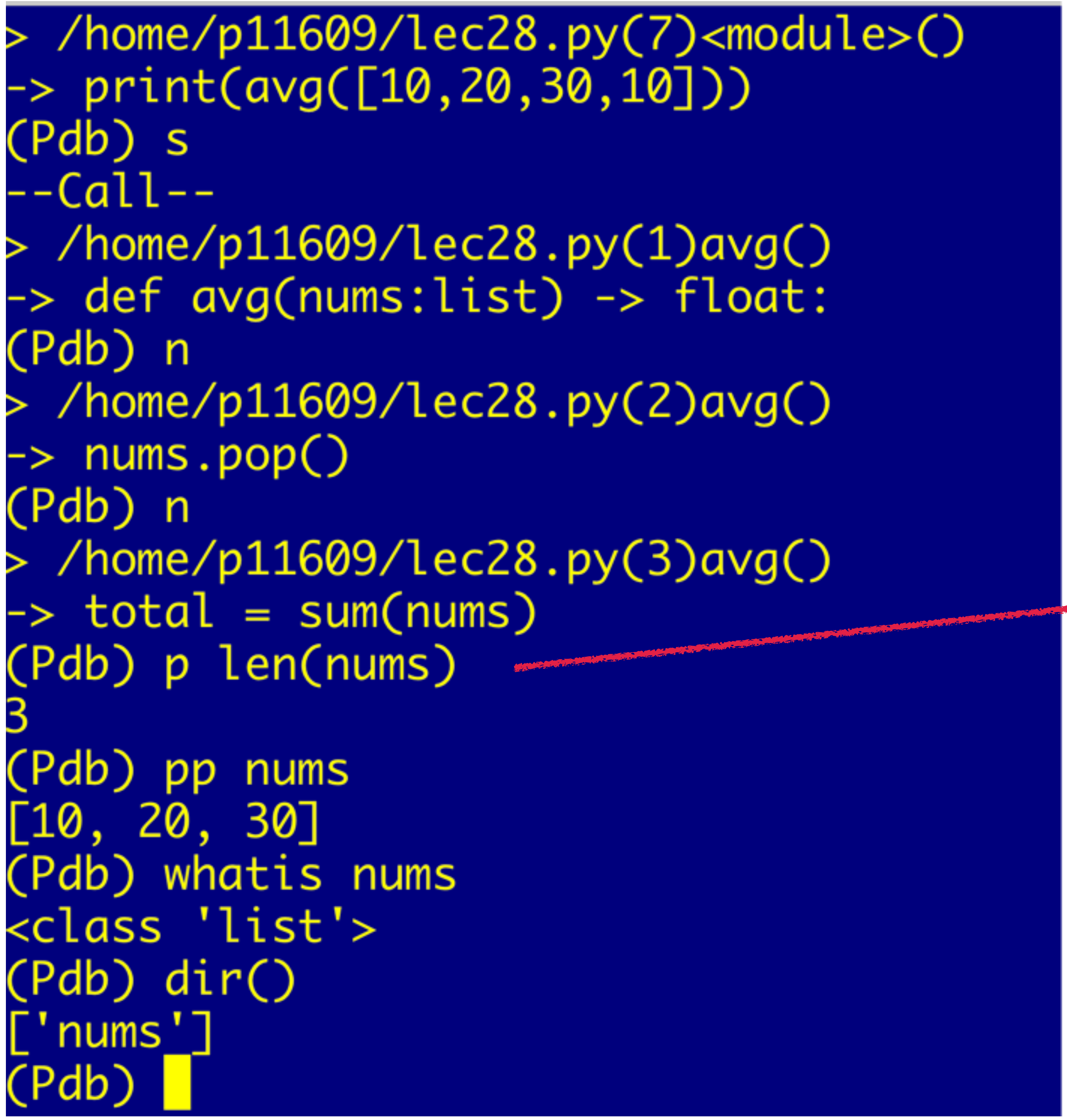


```
> /home/p11609/lec28.py(7)<module>()
-> print(avg([10,20,30,10]))
(Pdb) s
--Call--
> /home/p11609/lec28.py(1)avg()
-> def avg(nums:list) -> float:
(Pdb) n
> /home/p11609/lec28.py(2)avg()
-> nums.pop()
(Pdb) n
> /home/p11609/lec28.py(3)avg()
-> total = sum(nums)
(Pdb) p len(nums)
3
(Pdb) pp nums
[10, 20, 30]
(Pdb) whatis nums
<class 'list'>
(Pdb) dir()
['nums']
(Pdb)
```

More examples on pdb



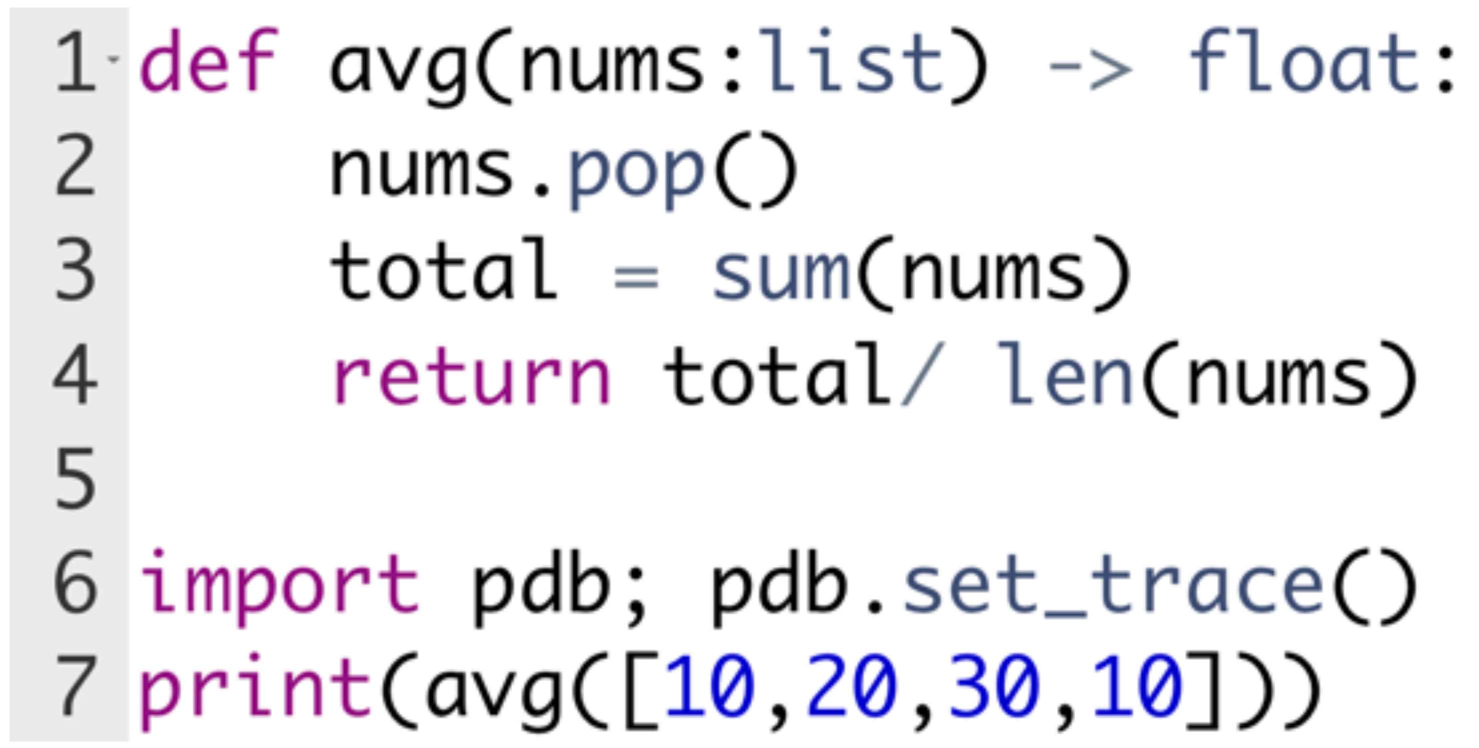
```python
def avg(nums:list) -> float:
    nums.pop()
    total = sum(nums)
    return total/ len(nums)

import pdb; pdb.set_trace()
print(avg([10,20,30,10]))
```



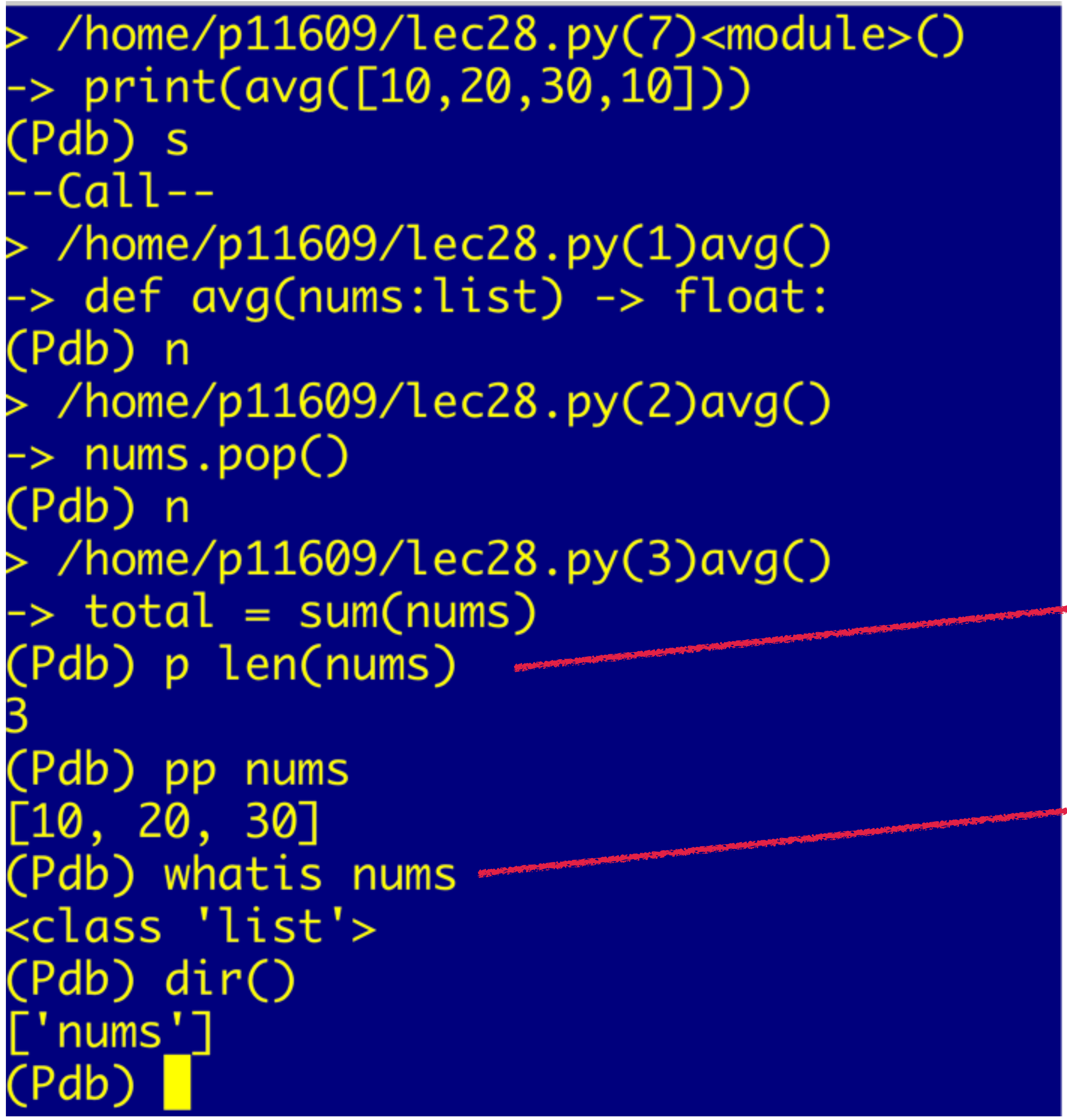
```
> /home/p11609/lec28.py(7)<module>()
-> print(avg([10,20,30,10]))
(Pdb) s
--Call--
> /home/p11609/lec28.py(1)avg()
-> def avg(nums:list) -> float:
(Pdb) n
> /home/p11609/lec28.py(2)avg()
-> nums.pop()
(Pdb) n
> /home/p11609/lec28.py(3)avg()
-> total = sum(nums)
(Pdb) p len(nums)
3
(Pdb) pp nums
[10, 20, 30]
(Pdb) whatis nums
<class 'list'>
(Pdb) dir()
['nums']
(Pdb) 
```

Could identify the bug.

More examples on pdb



```python
def avg(nums:list) -> float:
    nums.pop()
    total = sum(nums)
    return total/ len(nums)

import pdb; pdb.set_trace()
print(avg([10,20,30,10]))
```



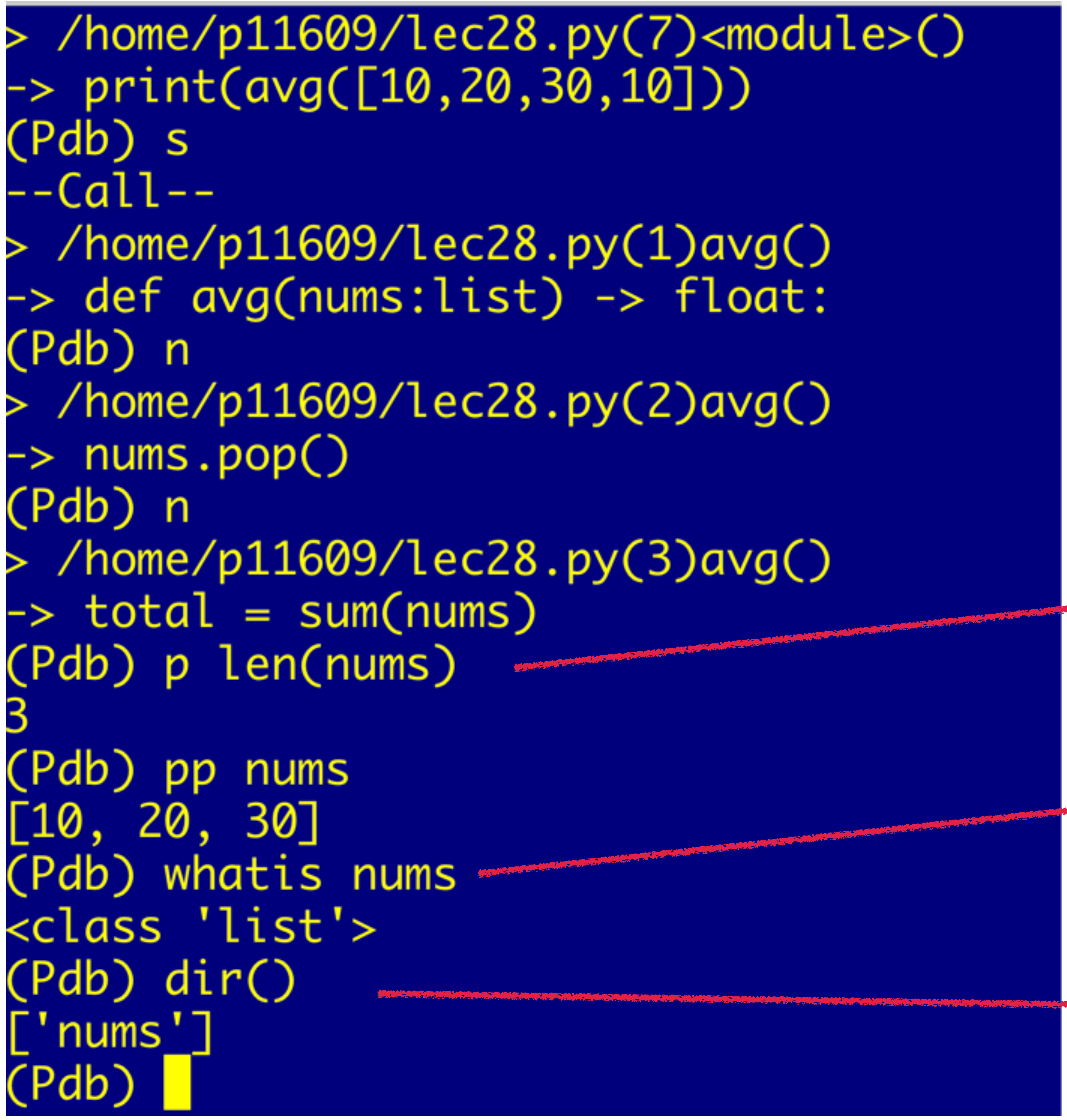
```
> /home/p11609/lec28.py(7)<module>()
-> print(avg([10,20,30,10]))
(Pdb) s
--Call--
> /home/p11609/lec28.py(1)avg()
-> def avg(nums:list) -> float:
(Pdb) n
> /home/p11609/lec28.py(2)avg()
-> nums.pop()
(Pdb) n
> /home/p11609/lec28.py(3)avg()
-> total = sum(nums)
(Pdb) p len(nums)
3
(Pdb) pp nums
[10, 20, 30]
(Pdb) whatis nums
<class 'list'>
(Pdb) dir()
['nums']
(Pdb)
```

Could identify the bug.

Tells you the type of object

More examples on pdb

```python
def avg(nums:list) -> float:
    nums.pop()
    total = sum(nums)
    return total/ len(nums)

import pdb; pdb.set_trace()
print(avg([10,20,30,10]))
```



```
> /home/p11609/lec28.py(7)<module>()
-> print(avg([10,20,30,10]))
(Pdb) s
--Call--
> /home/p11609/lec28.py(1)avg()
-> def avg(nums:list) -> float:
(Pdb) n
> /home/p11609/lec28.py(2)avg()
-> nums.pop()
(Pdb) n
> /home/p11609/lec28.py(3)avg()
-> total = sum(nums)
(Pdb) p len(nums)
3
(Pdb) pp nums
[10, 20, 30]
(Pdb) whatis nums
<class 'list'>
(Pdb) dir()
['nums']
(Pdb) 
```

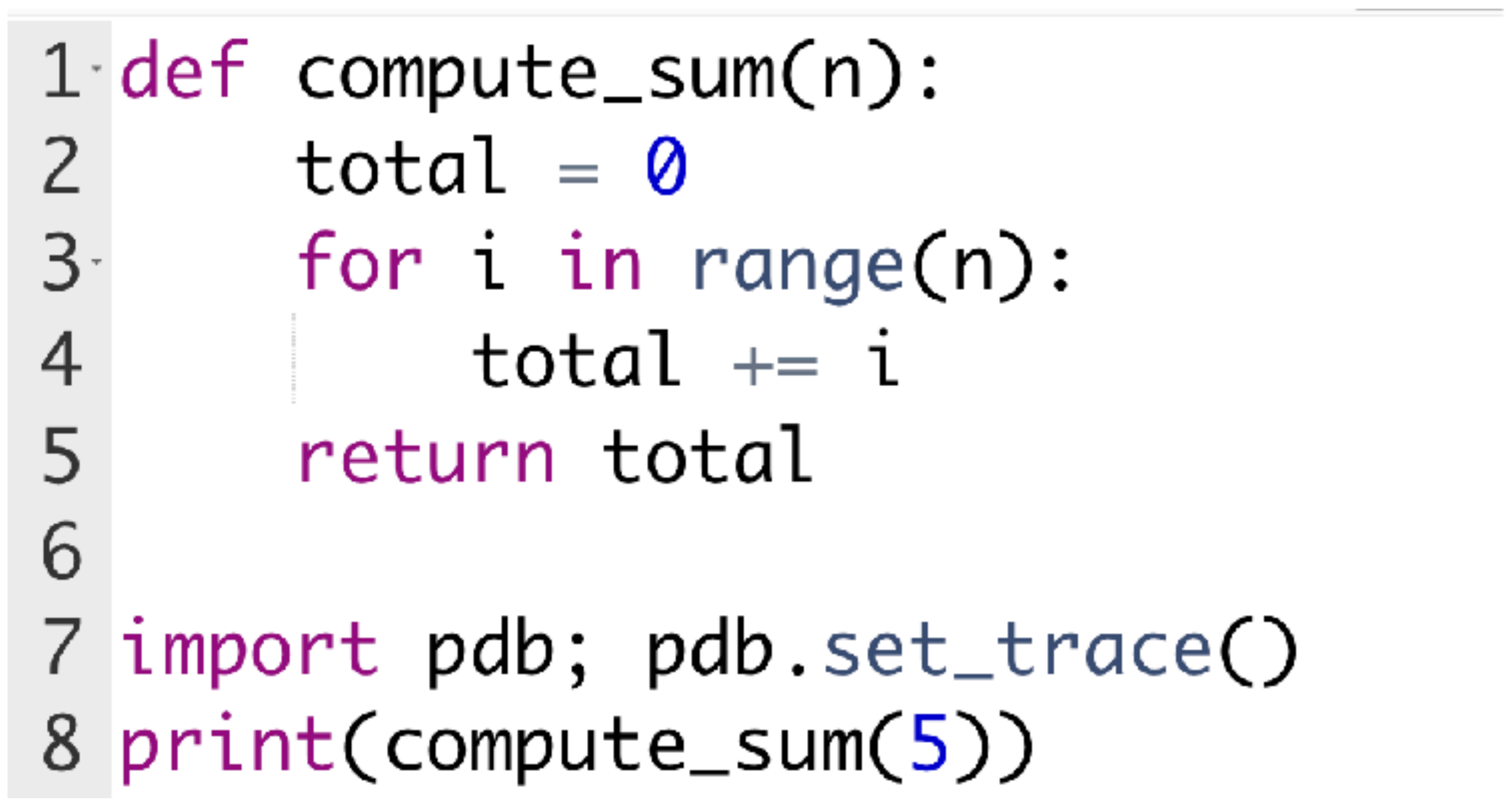
Could identify the bug.

Tells you the type of object

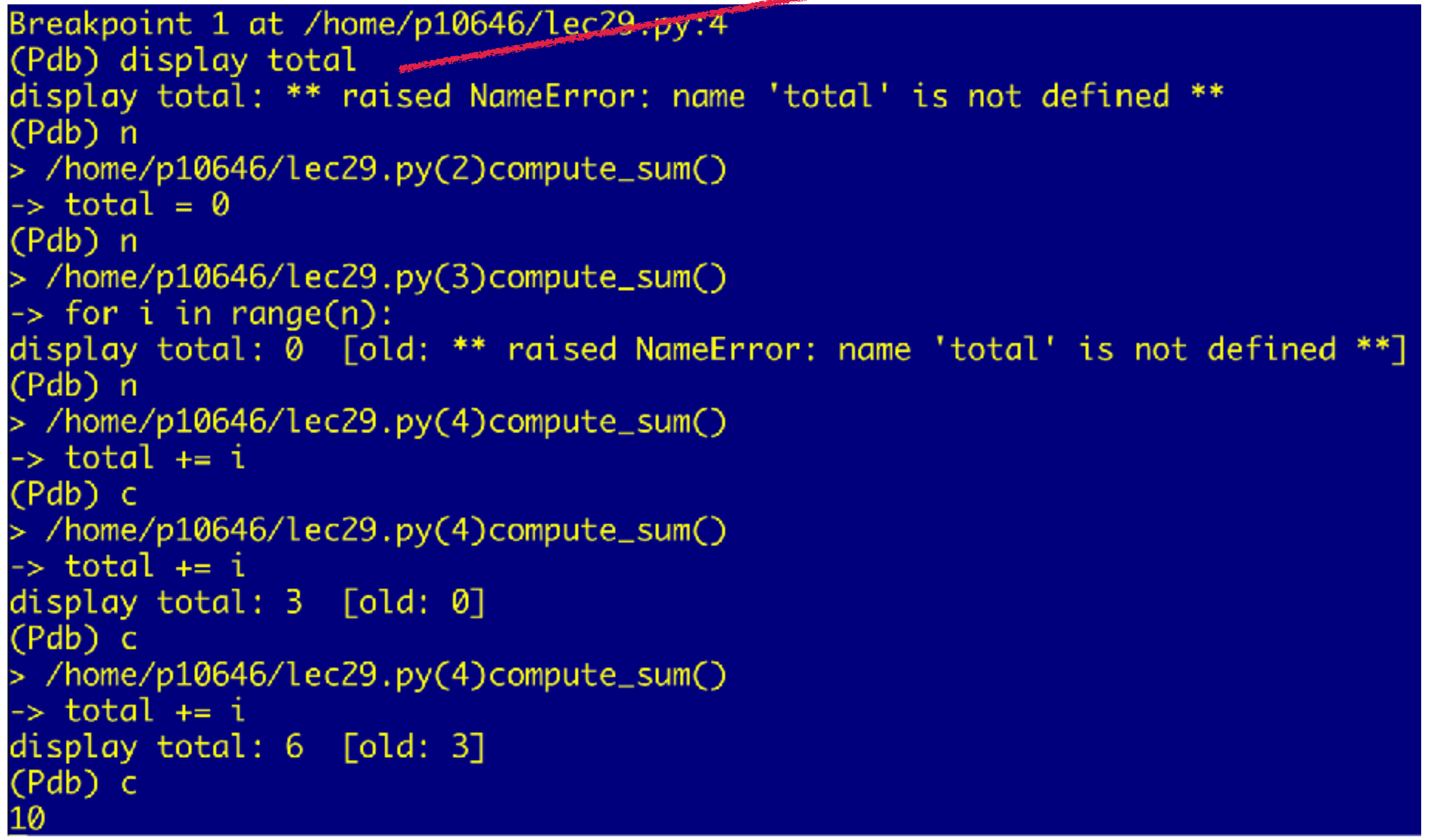
Lists the local variables.

More examples on pdb

Notice the use of “display”.



```
1 def compute_sum(n):
2     total = 0
3     for i in range(n):
4         total += i
5     return total
6
7 import pdb; pdb.set_trace()
8 print(compute_sum(5))
```



```
Breakpoint 1 at /home/p10646/lec29.py:4
(Pdb) display total
display total: ** raised NameError: name 'total' is not defined **
(Pdb) n
> /home/p10646/lec29.py(2)compute_sum()
-> total = 0
(Pdb) n
> /home/p10646/lec29.py(3)compute_sum()
-> for i in range(n):
display total: 0  [old: ** raised NameError: name 'total' is not defined **]
(Pdb) n
> /home/p10646/lec29.py(4)compute_sum()
-> total += i
(Pdb) c
> /home/p10646/lec29.py(4)compute_sum()
-> total += i
display total: 3  [old: 0]
(Pdb) c
> /home/p10646/lec29.py(4)compute_sum()
-> total += i
display total: 6  [old: 3]
(Pdb) c
10
```

display <expression>
Meaning: Register an expression to be automatically printed every time the debugger stops (e.g., after each step, breakpoint, or next).

You can manage displays:
undisplay <exp> — stop displaying that expression.
disable display <exp> — temporarily stop.
enable display <exp> — resume.

File handing

In any real-world application, data should not vanish when a program ends.
For example:

- A student management system must store student records permanently.
- A text editor saves your work on disk.
- A log file records errors and events over time.

This is where file handling comes in — it allows programs to **store**, **retrieve**, and **update** information on secondary storage (like hard drives).

In Python, files are objects that we can manipulate using built-in functions such as open(), read(), write(), and close().

File Paths: Relative vs Absolute

Absolute Path: The complete path from the **root directory** to the file.
Example (Windows):
C:\Users\Priyanka\Documents\data.txt
Example (Linux/Mac):
/home/priyanka/Documents/data.txt

Path relative to the current working directory.
Example:
If your script is in /home/priyanka/my.py, and the file is in /home/priyanka/data/marks.txt,
then the relative path is: data/marks.txt

```
import os
print(os.getcwd())  # get current working directory
```

```
os.chdir("/path/to/folder")
```

Now, the program will consider you in “folder”.

```
os.path.exists(“filename.txt”)
```

Return True if file is there, else return False.