COL1000
Introduction to Programming

Priyanka Golia

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

Parameters vs. Arguments

Parameter (at definition time) — A parameter is a placeholder you write in a function definition. It says: “This function expects a value here.”

```
def greet(name):    # ← "name" is a parameter
return f”hello {name}”
```

Here name is just a symbol (variable) inside the function — no actual value yet

An argument is the actual value (object) you pass when calling the function. It “fills” the parameter.

```
S = greet(“priyanka”)    # ← "priyanka" is an argument.
```

Definition doesn’t execute until unless it is called.

Parameters vs. Arguments

When you pass a variable to a function, the reference to the object is passed

```
def greet(name):    # ← "name" is a parameter
return f”hello {name}”
```

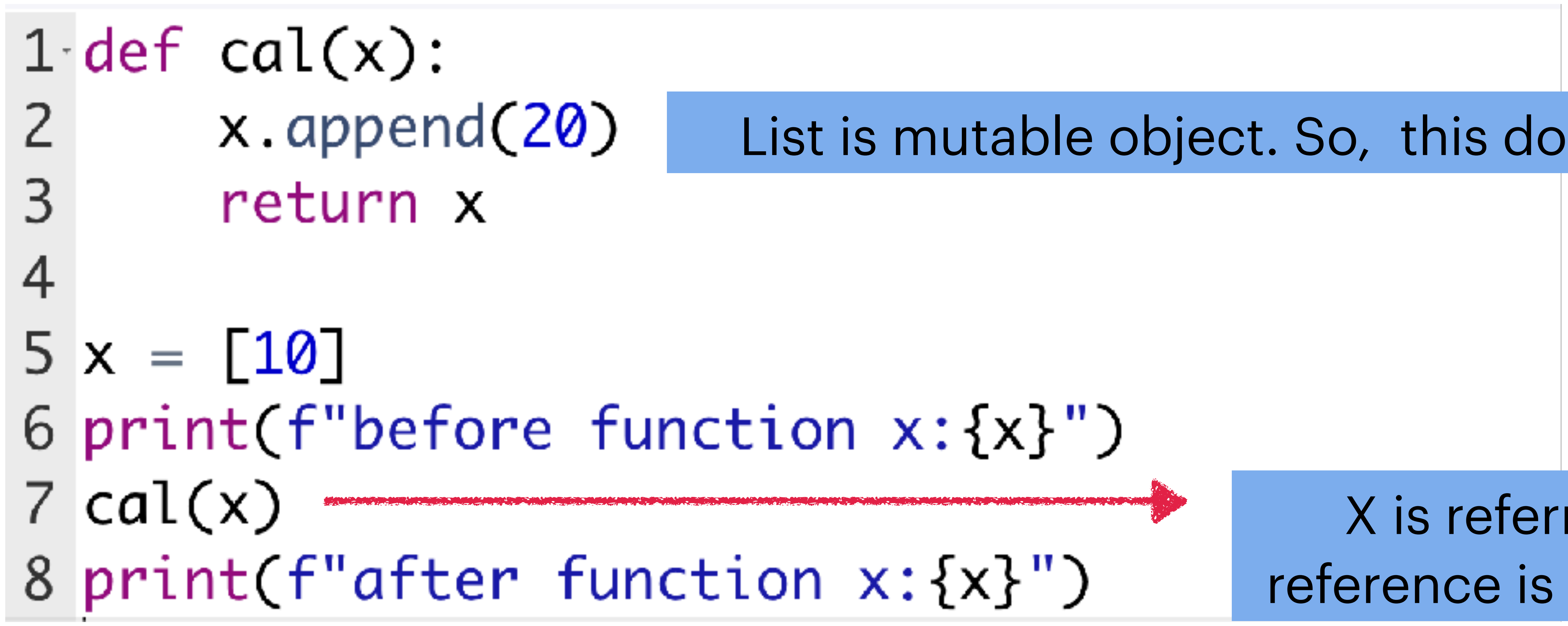
Variable name is now referencing to object “priyanka”

```
S = greet(“priyanka”)    # ← "priyanka" is an argument.
```

An object “priyanka” is created, and reference to that object is passed in “greet”

Parameters vs. Arguments

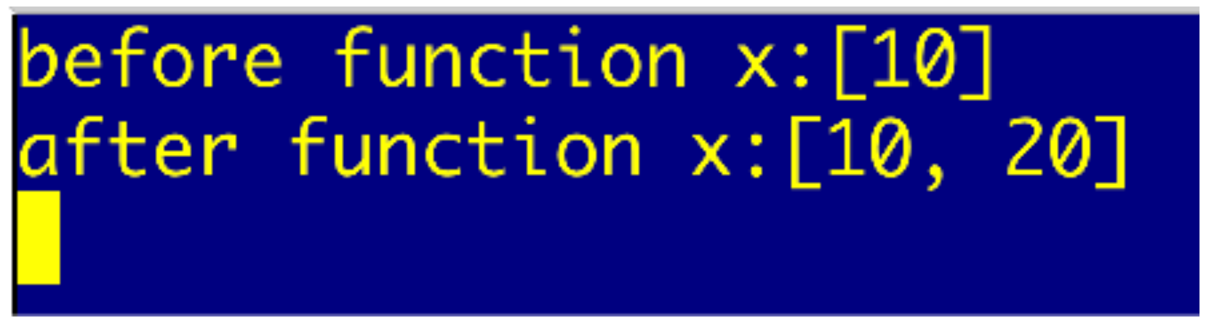
When you pass a variable to a function, the reference to the object is passed



```python
def cal(x):
    x.append(20)
    return x

x = [10]
print(f"before function x:{x}")
cal(x)
print(f"after function x:{x}")
```

List is mutable object. So, this doesn't create a new object.

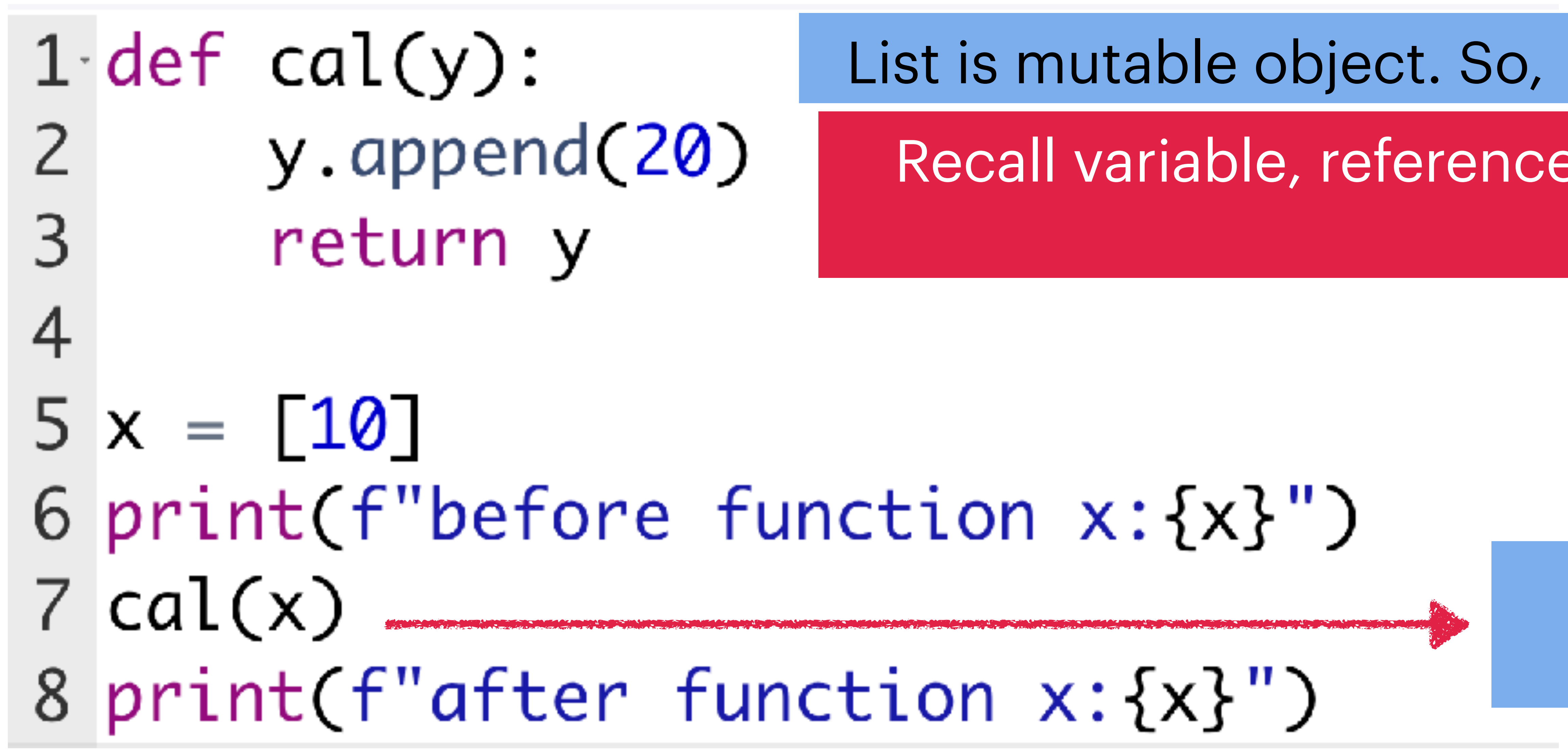
X is referring to a list object [10]. The reference is passed as an argument in `cal'.



```
before function x:[10]
after function x:[10, 20]
```

Parameters vs. Arguments

When you pass a variable to a function, the reference to the object is passed

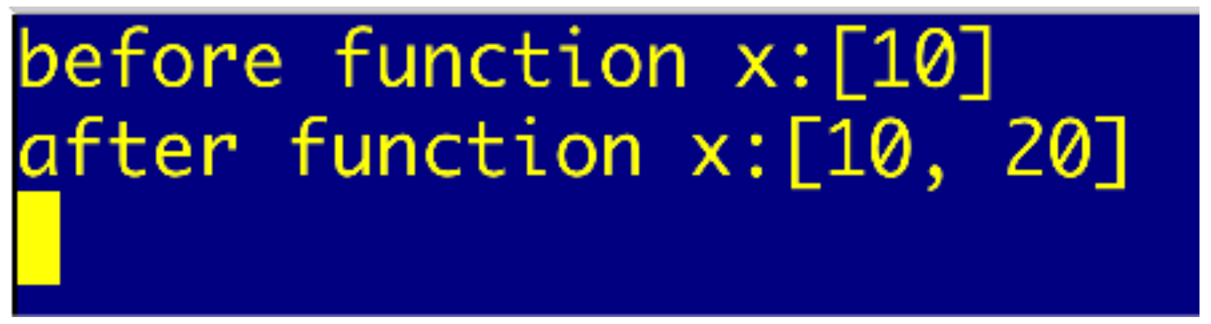


```python
def cal(y):
    y.append(20)
    return y

x = [10]
print(f"before function x:{x}")
cal(x)
print(f"after function x:{x}")
```

List is mutable object. So, this doesn't create a new object.

Recall variable, reference and object. Doesn't matter what is the `name` of the variable, as far as it is referring to same object.

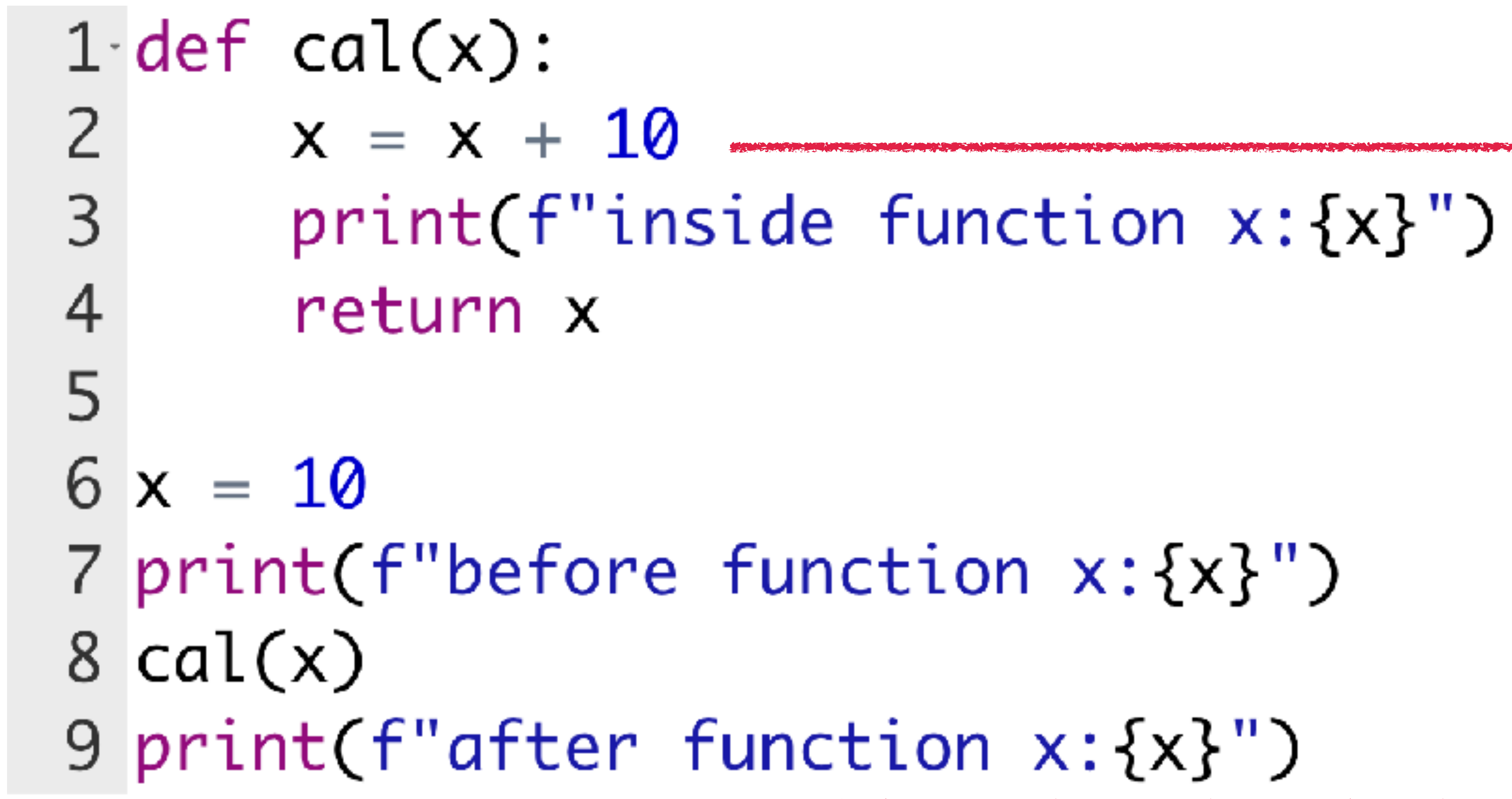
X is referring to a list object [10]. The reference is passed as an argument in `cal'.



```
before function x:[10]
after function x:[10, 20]
```

Parameters vs. Arguments

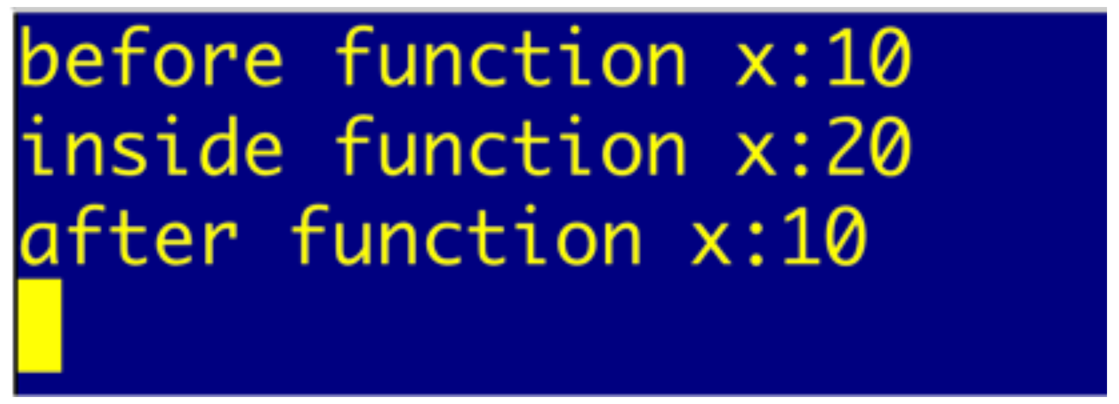
When you pass a variable to a function, the reference to the object is passed



```python
def cal(x):
    x = x + 10
    print(f"inside function x:{x}")
    return x

x = 10
print(f"before function x:{x}")
cal(x)
print(f"after function x:{x}")
```

Integer is immutable object. So, it will create a new object 20, which is referred to as by a local variable x (which is different than global x).

Integer is immutable object. This is still refereeing to 10.



```
before function x:10
inside function x:20
after function x:10
```

Parameters vs. Arguments

Function parameters are **references to objects**.

Inside the function, the parameter becomes a **new local name** bound to the same object.

Behavior depends on whether the object is mutable (changes reflect outside) or immutable (changes don't).

Practice with different mutable and immutable objects   list, tuple, dic, sets, integer, list with tuple, tuple with lists, strings, etc.

Scope!

```python
x = 10    # Global variable

def func():
    x = 5    # Local variable
    print("Inside function:", x)

func()
print("Outside function:", x)
```

```
Inside function: 5
Outside function: 10
```

```python
x = 10    # Global variable

def func():
      print("Inside function:", x)

func()
print("Outside function:", x)
```

Can access global variables.

```
Inside function: 10
Outside function: 10
```

Scope!

```
x = 10

def func():
    x = x + 1    # ❌ UnboundLocalError
    print(x)

func()
```

As soon as you attempt to change the reference, it creates a new local variable. Now, this “x” is considered local to func(). Since, we are accessing this local x, without defining it, hence the error.

```
x = 10

def func():
     global x
     x = x + 1
     print(x)

func()
```

Now, it is changing the reference for global x, which will be visible outside func()

Scope!

```
def outer():
    y = 20
    def inner():
        print("Inner sees:", y)  # y from enclosing function
    inner()
outer()
```

```
def outer():
    y = 20
    def inner():
          y = y + 20 # ❌ UnboundLocalError
     inner()
outer()
```

```
Inner sees : 20
```

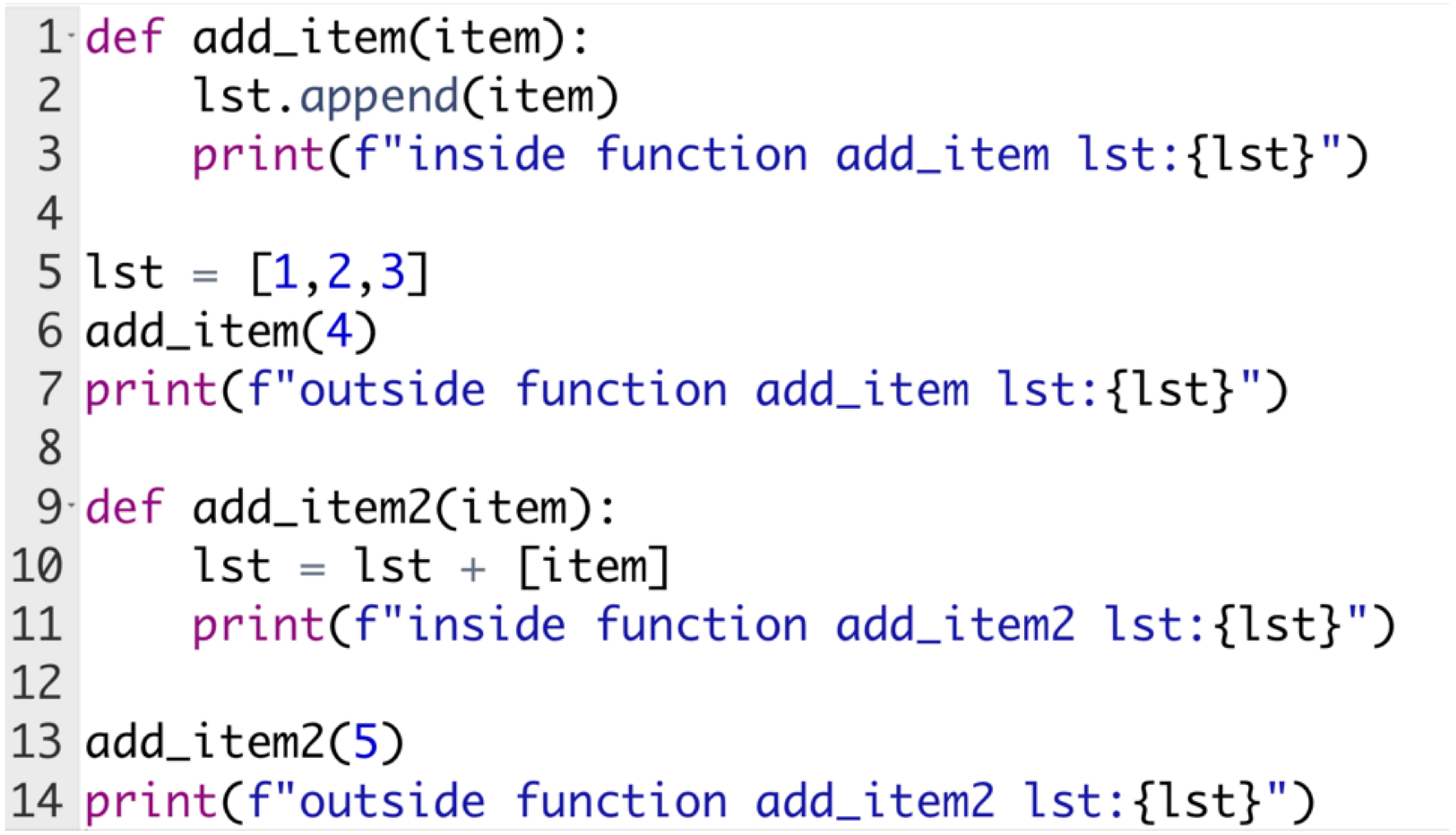
Scope!

```
def outer():
    y = 20
    def inner():
         y = y + 20 # ❌ UnboundLocalError
     inner()
outer()
```

```
def outer():
    y = 20
    def inner():
         nonlocal y
         y = y + 20
         print(f“inside inner y:{y}”)
     inner()
     print(f“inside outer y:{y}”)
outer()
```

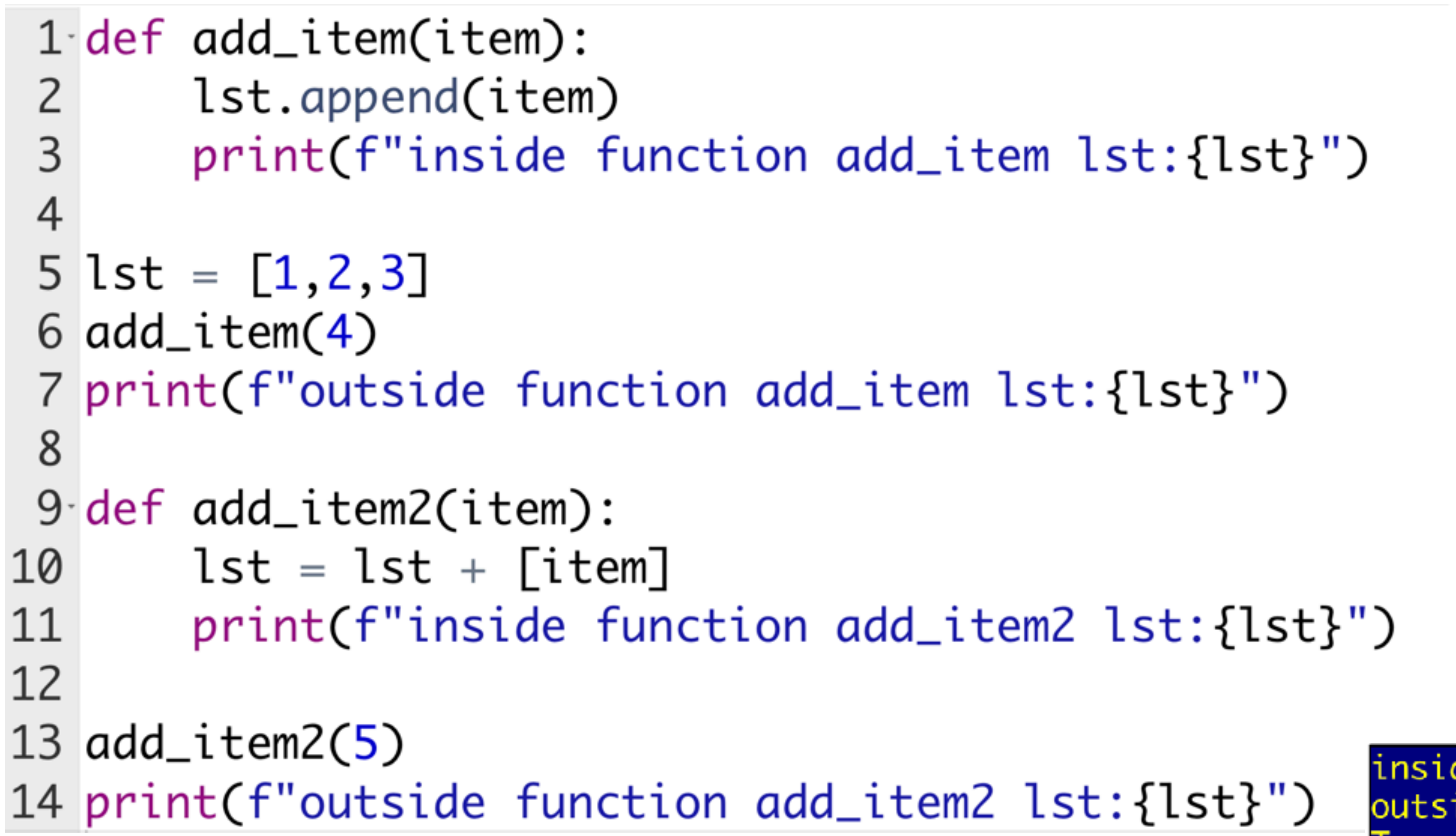
```
Inside inner: 40
inside outer: 40
```

Scope!

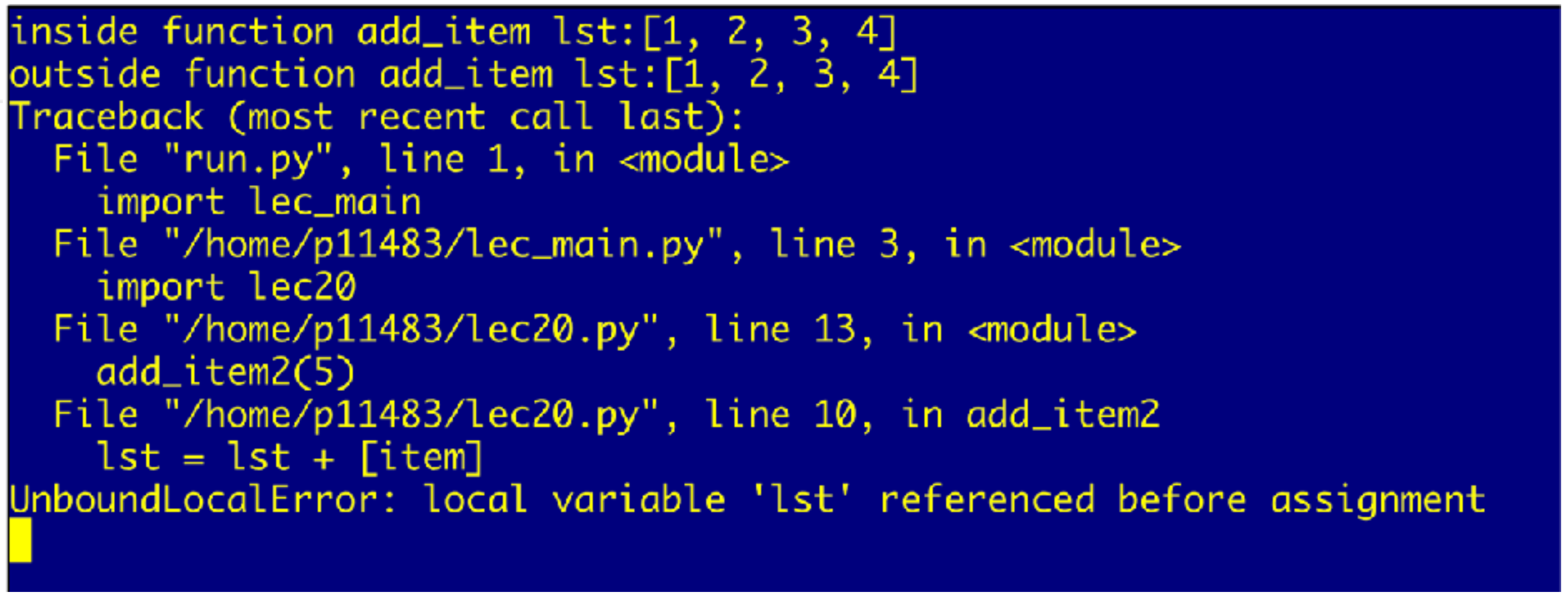


```python
def add_item(item):
    lst.append(item)
    print(f"inside function add_item lst:{lst}")

lst = [1,2,3]
add_item(4)
print(f"outside function add_item lst:{lst}")

def add_item2(item):
    lst = lst + [item]
    print(f"inside function add_item2 lst:{lst}")

add_item2(5)
print(f"outside function add_item2 lst:{lst}")
```

Scope!

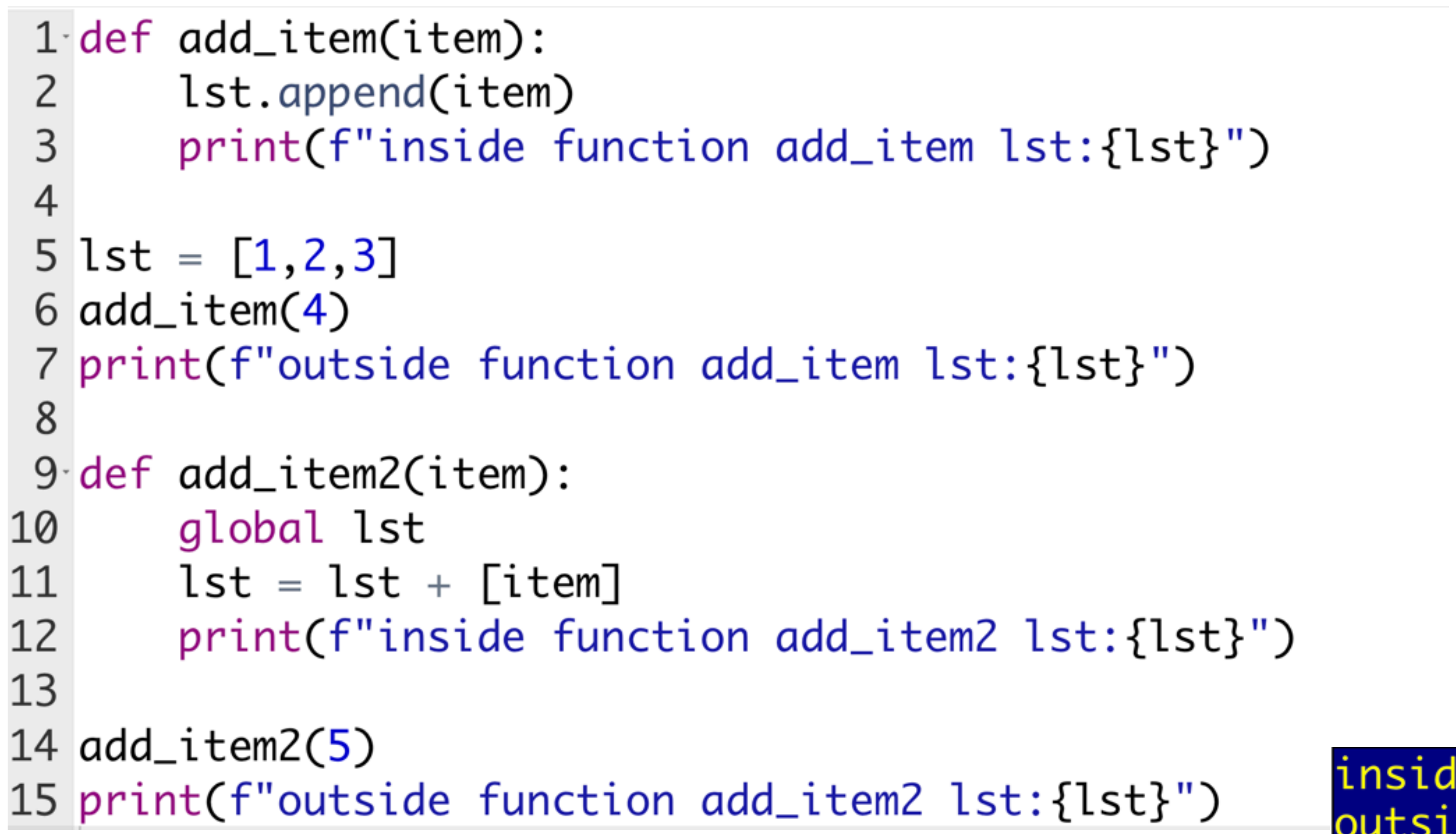


```
1  def add_item(item):
2      lst.append(item)
3      print(f"inside function add_item lst:{lst}")
4
5  lst = [1,2,3]
6  add_item(4)
7  print(f"outside function add_item lst:{lst}")
8
9  def add_item2(item):
10     lst = lst + [item]
11     print(f"inside function add_item2 lst:{lst}")
12
13 add_item2(5)
14 print(f"outside function add_item2 lst:{lst}")
```

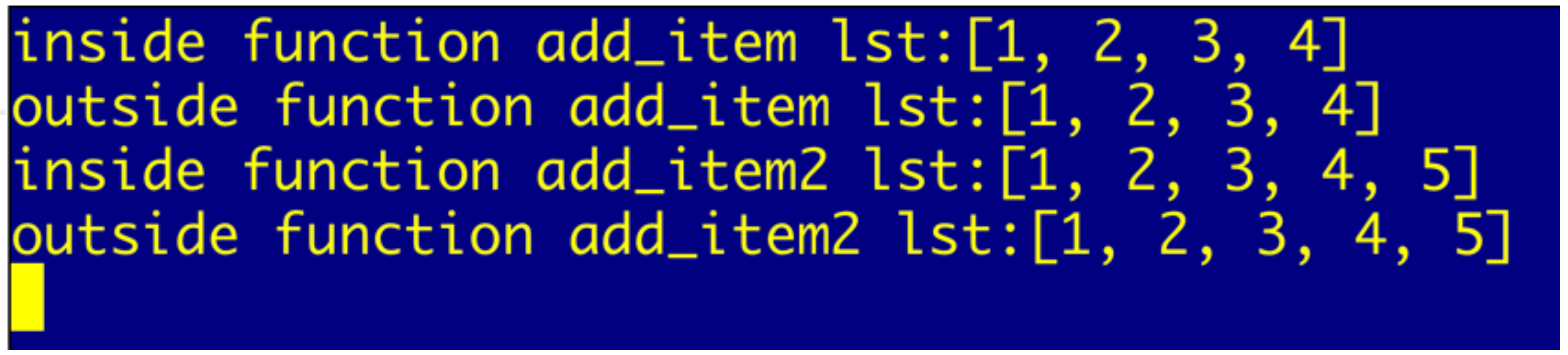


```
inside function add_item lst:[1, 2, 3, 4]
outside function add_item lst:[1, 2, 3, 4]
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p11483/lec_main.py", line 3, in <module>
    import lec20
  File "/home/p11483/lec20.py", line 13, in <module>
    add_item2(5)
  File "/home/p11483/lec20.py", line 10, in add_item2
    lst = lst + [item]
UnboundLocalError: local variable 'lst' referenced before assignment
```

Scope!



```python
def add_item(item):
    lst.append(item)
    print(f"inside function add_item lst:{lst}")

lst = [1,2,3]
add_item(4)
print(f"outside function add_item lst:{lst}")

def add_item2(item):
    global lst
    lst = lst + [item]
    print(f"inside function add_item2 lst:{lst}")

add_item2(5)
print(f"outside function add_item2 lst:{lst}")
```



```
inside function add_item lst:[1, 2, 3, 4]
outside function add_item lst:[1, 2, 3, 4]
inside function add_item2 lst:[1, 2, 3, 4, 5]
outside function add_item2 lst:[1, 2, 3, 4, 5]
```

Scope!

```
def func():
    print(len([1, 2, 3]))  # len is built-in

func()
```

Types of Scope (LEGB Rule)

Python follows the LEGB rule for variable resolution:

L – Local: Variables defined inside a function.

E – Enclosing: Variables in the enclosing function (nested functions).

G – Global: Variables defined at the top level of a module or declared as global.

B – Built-in: Predefined names like len, print.

Python searches in this order (L → E → G → B).

Quiz

```
x = 5

def foo():
    print(x)

foo()
```

Quiz

```
x = 5

def foo():
    print(x)

foo()
```

5. The function only *reads x*. Since there's no assignment, Python looks in the global scope and finds x.

Quiz

```
x = 5

def foo():
    print(x)

foo()
```

5. The function only *reads* x. Since there's no assignment, Python looks in the global scope and finds x.

```
x = 5

def foo():
    print(x)
    x = 10

foo()
```

Quiz

```
x = 5

def foo():
    print(x)

foo()
```

5. The function only *reads* x. Since there's no assignment, Python looks in the global scope and finds x.

```
x = 5

def foo():
    print(x)
    x = 10

foo()
```

Error: UnboundLocalError. Because of x = 10, Python assumes x is local. But print(x) tries to read the local x before it's assigned.

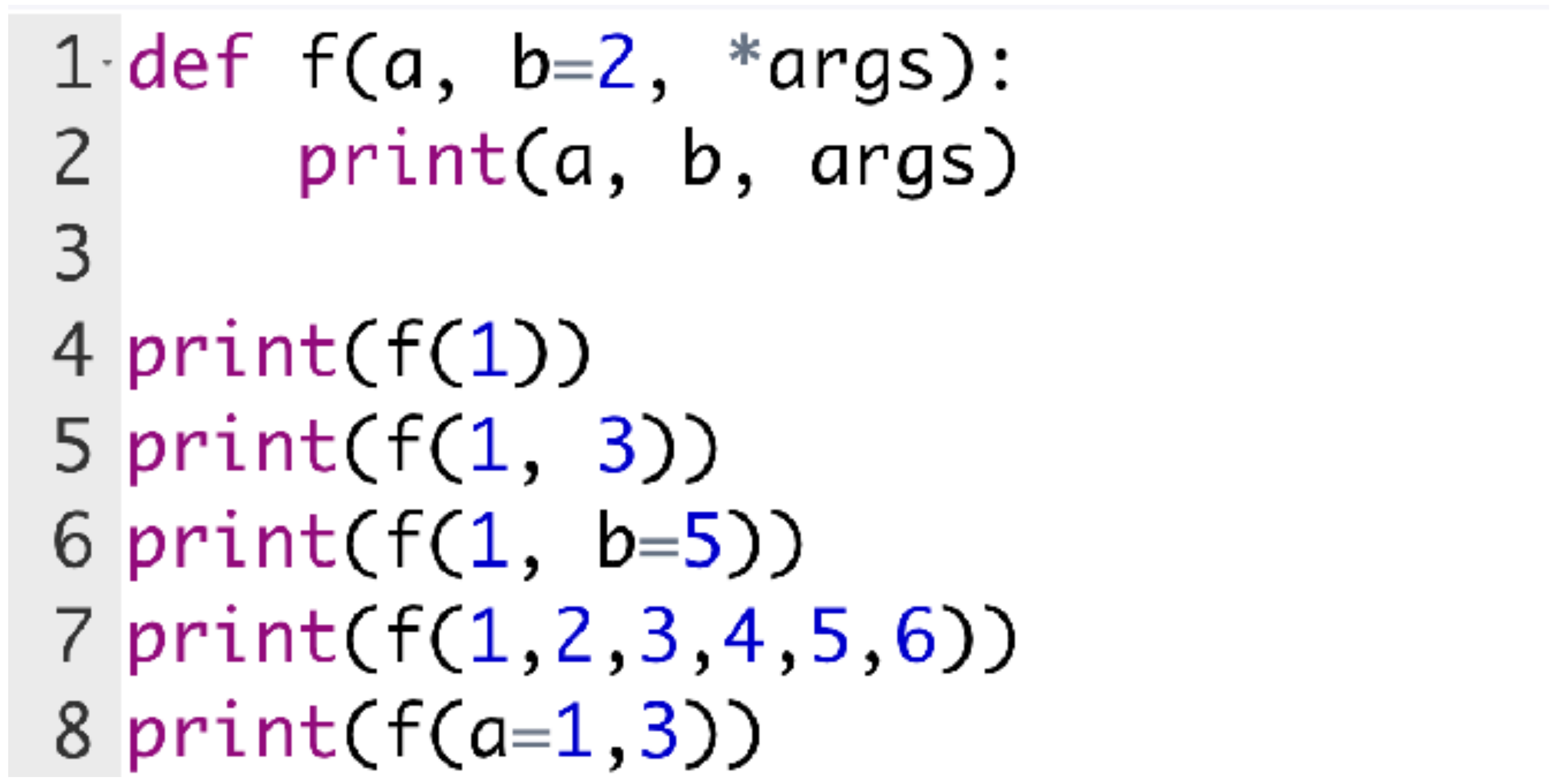
Quiz

```
x = 5

def foo():
    print(x)

foo()
```

5. The function only *reads* x. Since there's no assignment, Python looks in the global scope and finds x.

```
x = 5

def foo():
    print(x)
    x = 10

foo()
```

Error: UnboundLocalError. Because of x = 10, Python assumes x is local. But print(x) tries to read the local x before it's assigned.



```python
def f(a, b=2, *args):
    print(a, b, args)

print(f(1))
print(f(1, 3))
print(f(1, b=5))
print(f(1,2,3,4,5,6))
print(f(a=1,3))
```

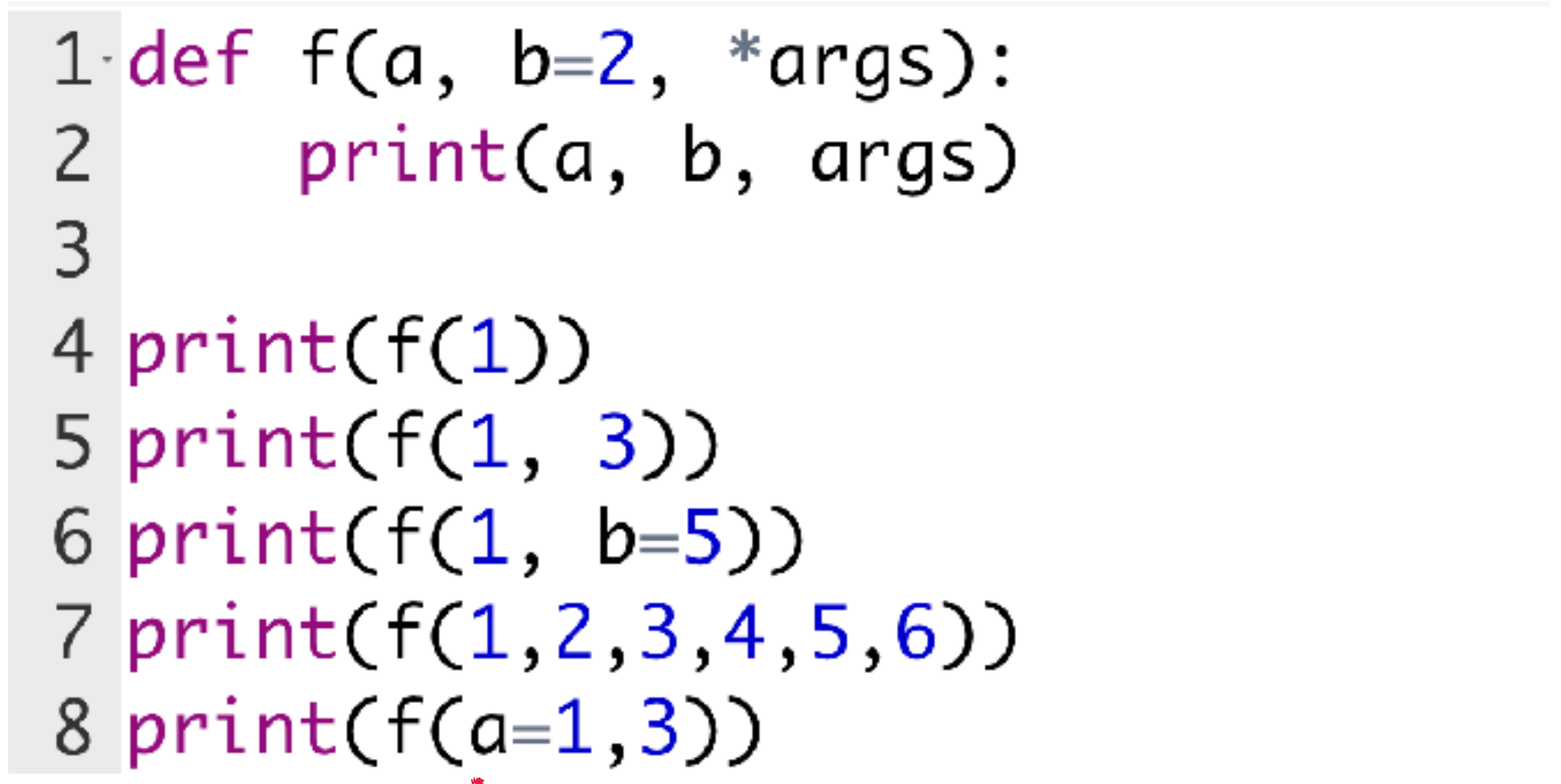
Quiz

```
x = 5

def foo():
    print(x)

foo()
```

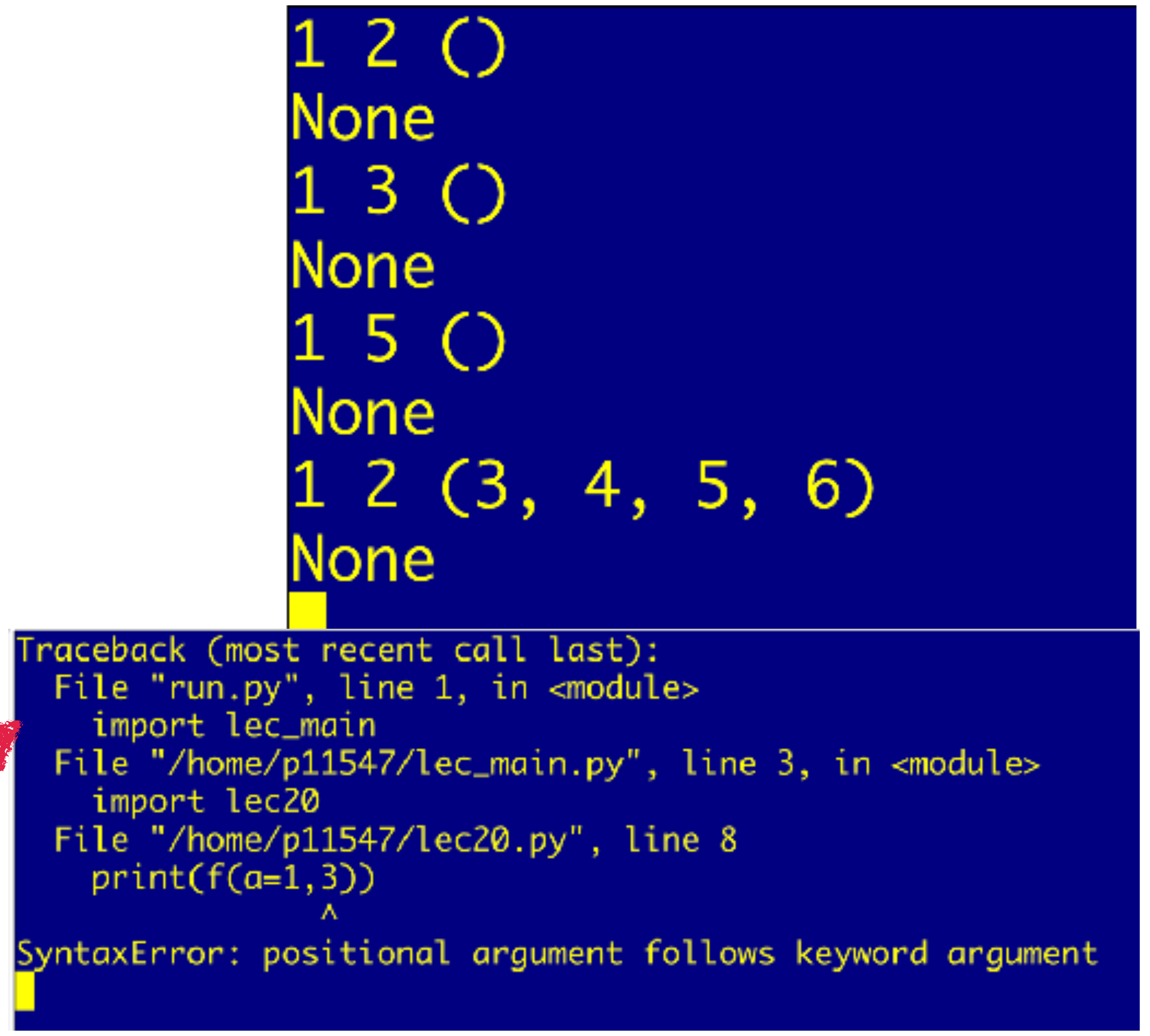
5. The function only *reads* x. Since there's no assignment, Python looks in the global scope and finds x.

```
x = 5

def foo():
    print(x)
    x = 10

foo()
```

Error: UnboundLocalError. Because of x = 10, Python assumes x is local. But print(x) tries to read the local x before it's assigned.



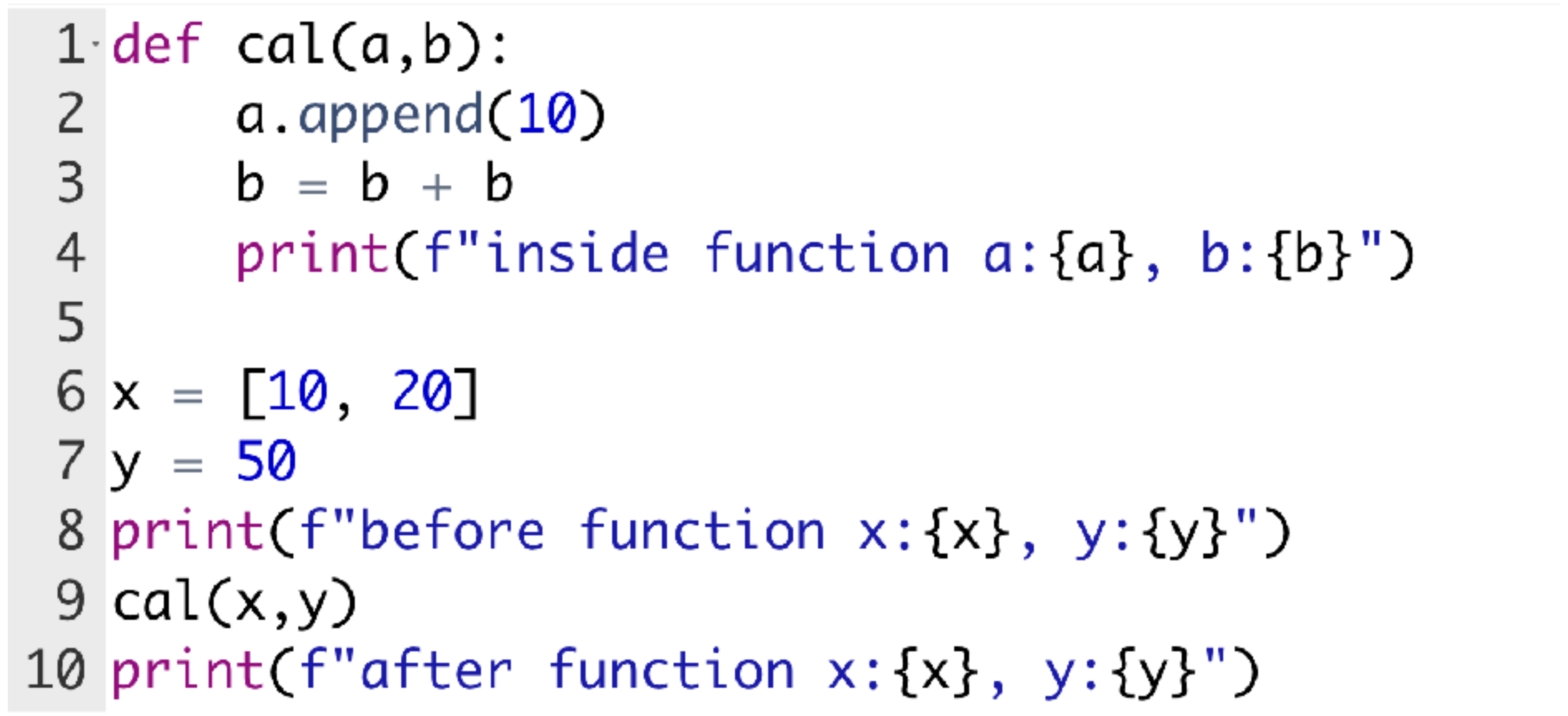
```
def f(a, b=2, *args):
    print(a, b, args)

print(f(1))
print(f(1, 3))
print(f(1, b=5))
print(f(1,2,3,4,5,6))
print(f(a=1,3))
```



```
1 2 ()
None
1 3 ()
None
1 5 ()
None
1 2 (3, 4, 5, 6)
None
```

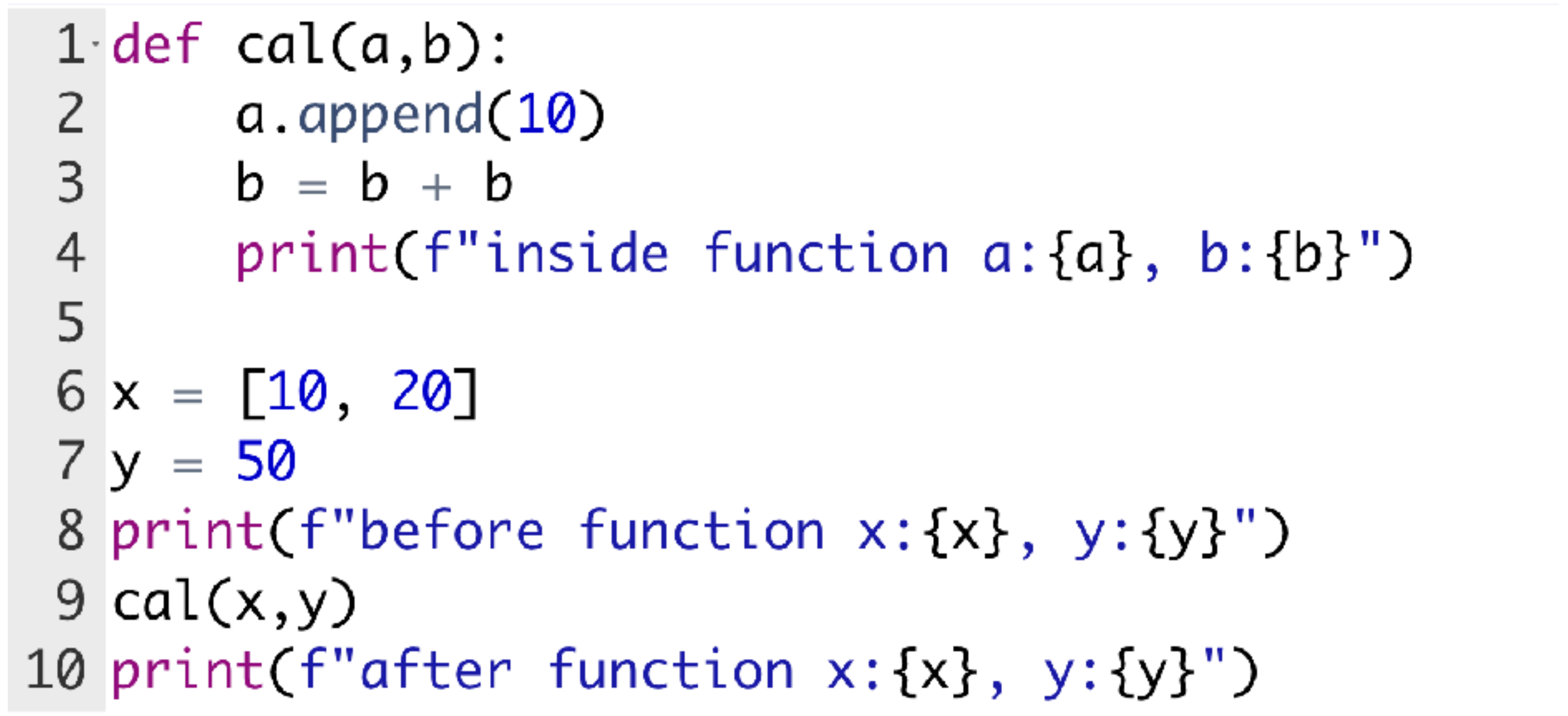
```
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p11547/lec_main.py", line 3, in <module>
    import lec20
  File "/home/p11547/lec20.py", line 8
    print(f(a=1,3))
               ^
SyntaxError: positional argument follows keyword argument
```

Quiz

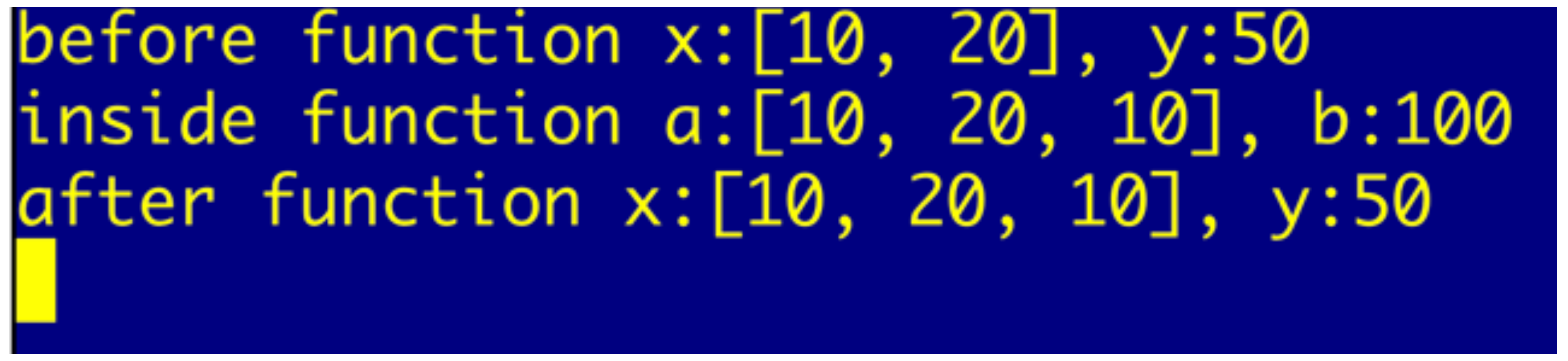


```python
def cal(a,b):
    a.append(10)
    b = b + b
    print(f"inside function a:{a}, b:{b}")

x = [10, 20]
y = 50
print(f"before function x:{x}, y:{y}")
cal(x,y)
print(f"after function x:{x}, y:{y}")
```

Quiz

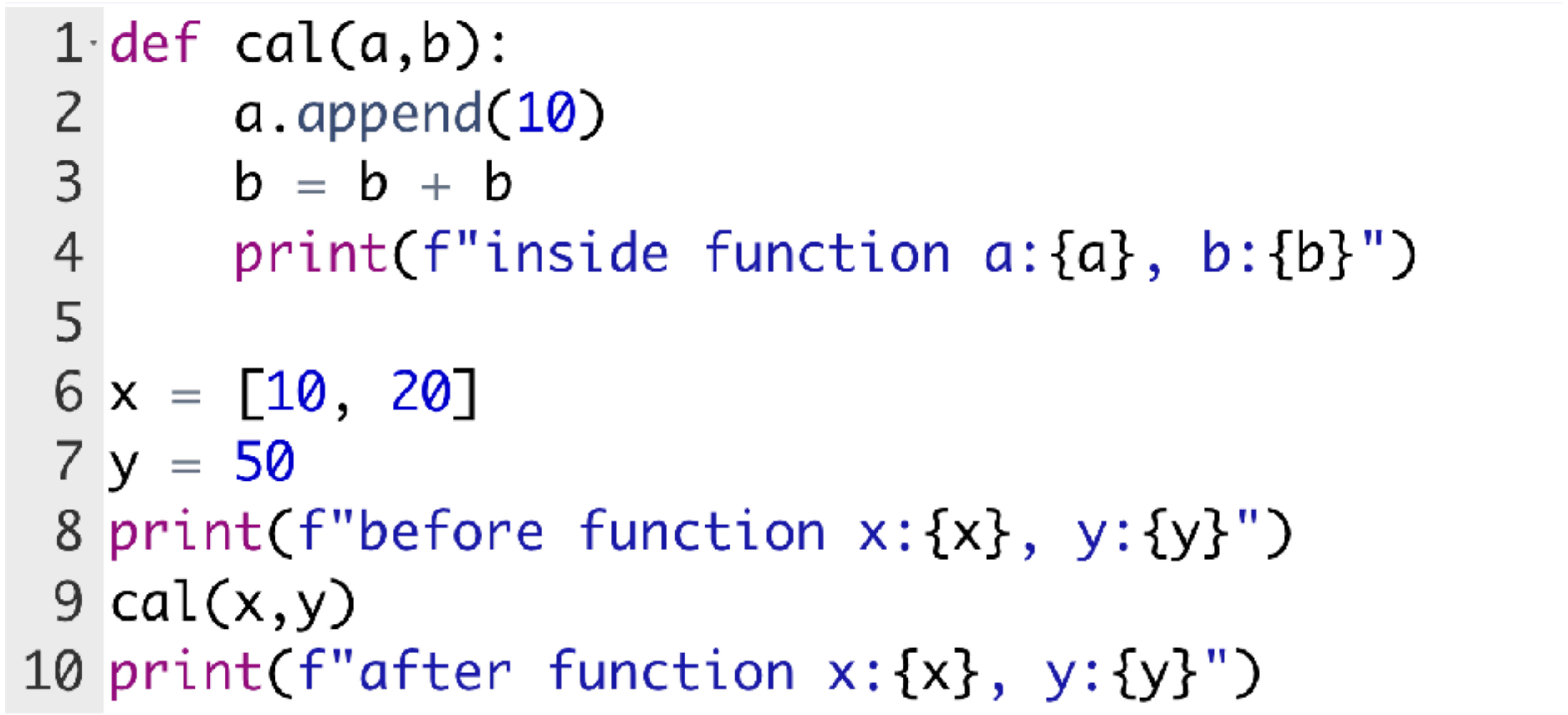


```python
def cal(a,b):
    a.append(10)
    b = b + b
    print(f"inside function a:{a}, b:{b}")

x = [10, 20]
y = 50
print(f"before function x:{x}, y:{y}")
cal(x,y)
print(f"after function x:{x}, y:{y}")
```

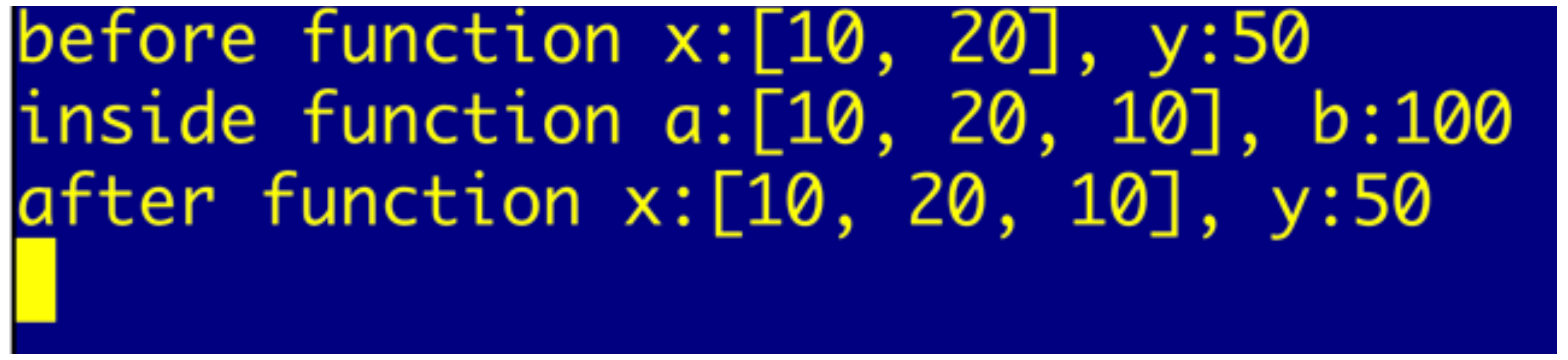


```
before function x:[10, 20], y:50
inside function a:[10, 20, 10], b:100
after function x:[10, 20, 10], y:50
```

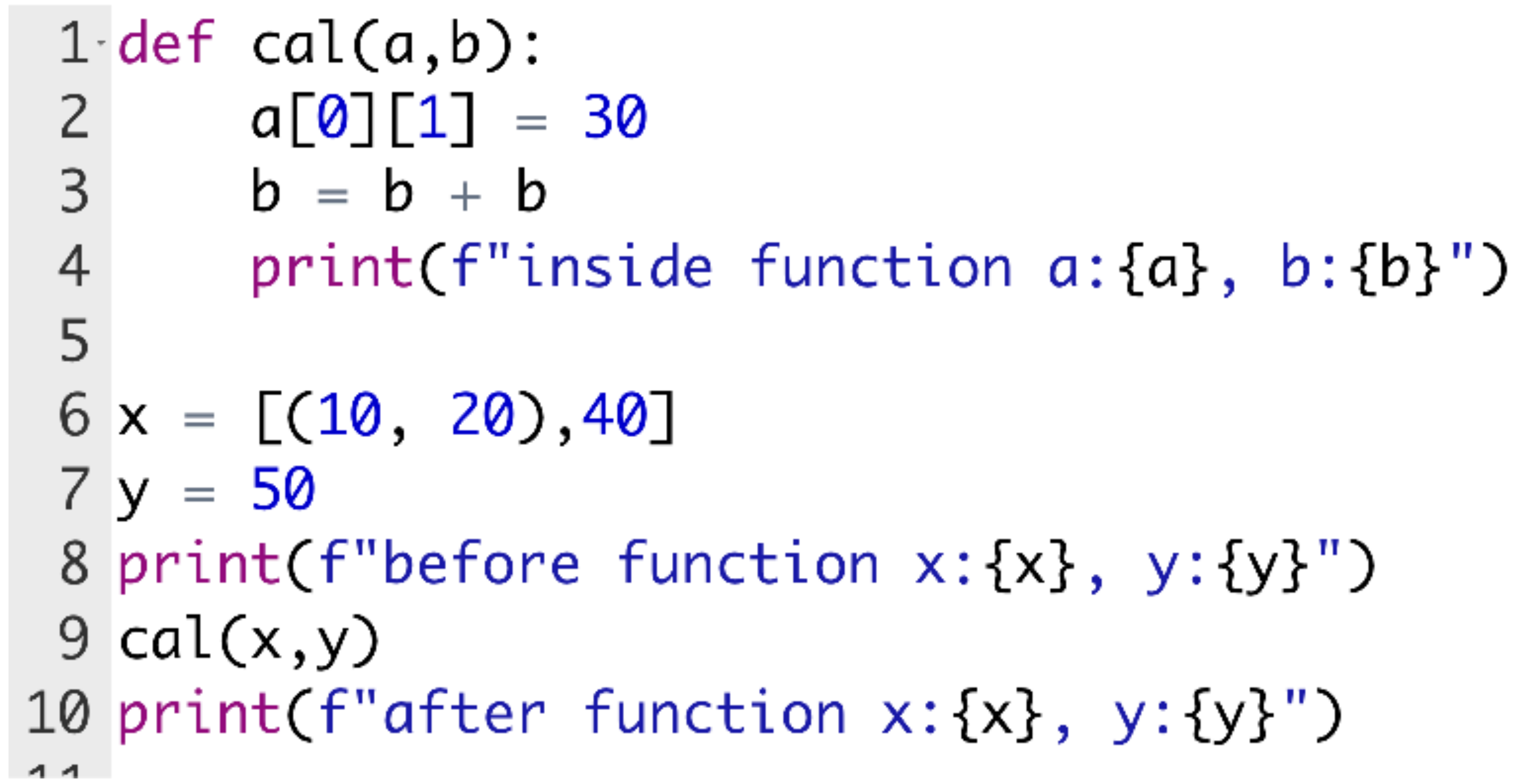
Quiz



```python
def cal(a,b):
    a.append(10)
    b = b + b
    print(f"inside function a:{a}, b:{b}")

x = [10, 20]
y = 50
print(f"before function x:{x}, y:{y}")
cal(x,y)
print(f"after function x:{x}, y:{y}")
```

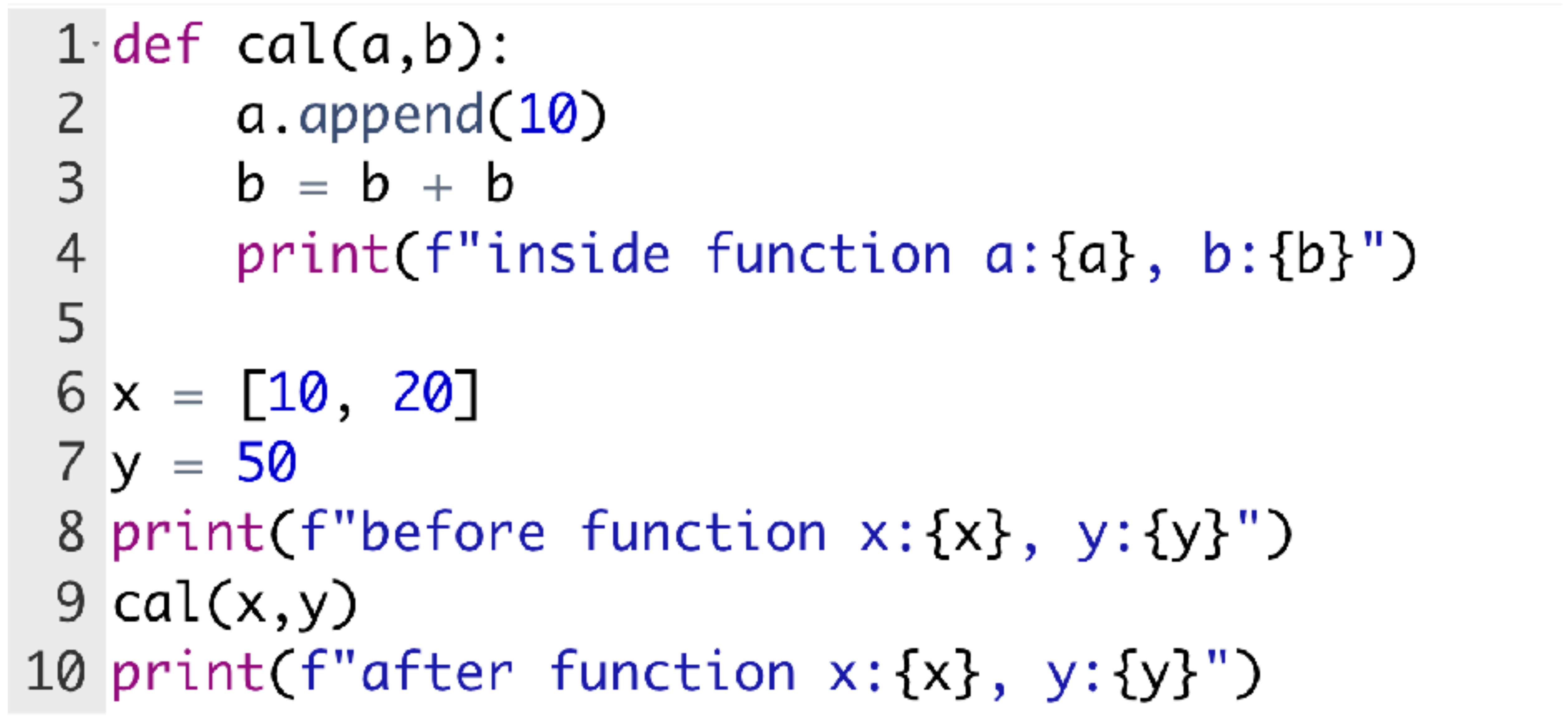


```
before function x:[10, 20], y:50
inside function a:[10, 20, 10], b:100
after function x:[10, 20, 10], y:50
```

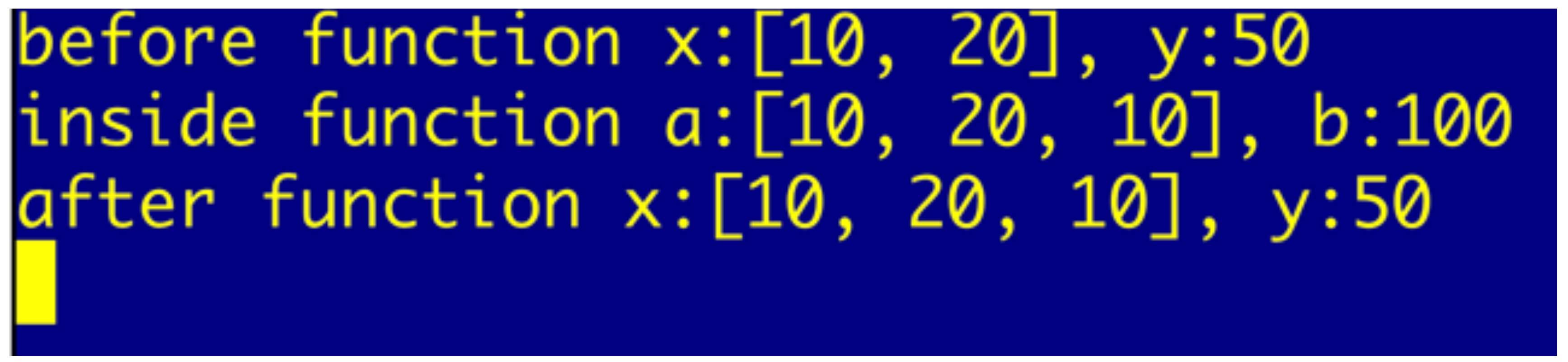


```python
def cal(a,b):
    a[0][1] = 30
    b = b + b
    print(f"inside function a:{a}, b:{b}")

x = [(10, 20),40]
y = 50
print(f"before function x:{x}, y:{y}")
cal(x,y)
print(f"after function x:{x}, y:{y}")
```

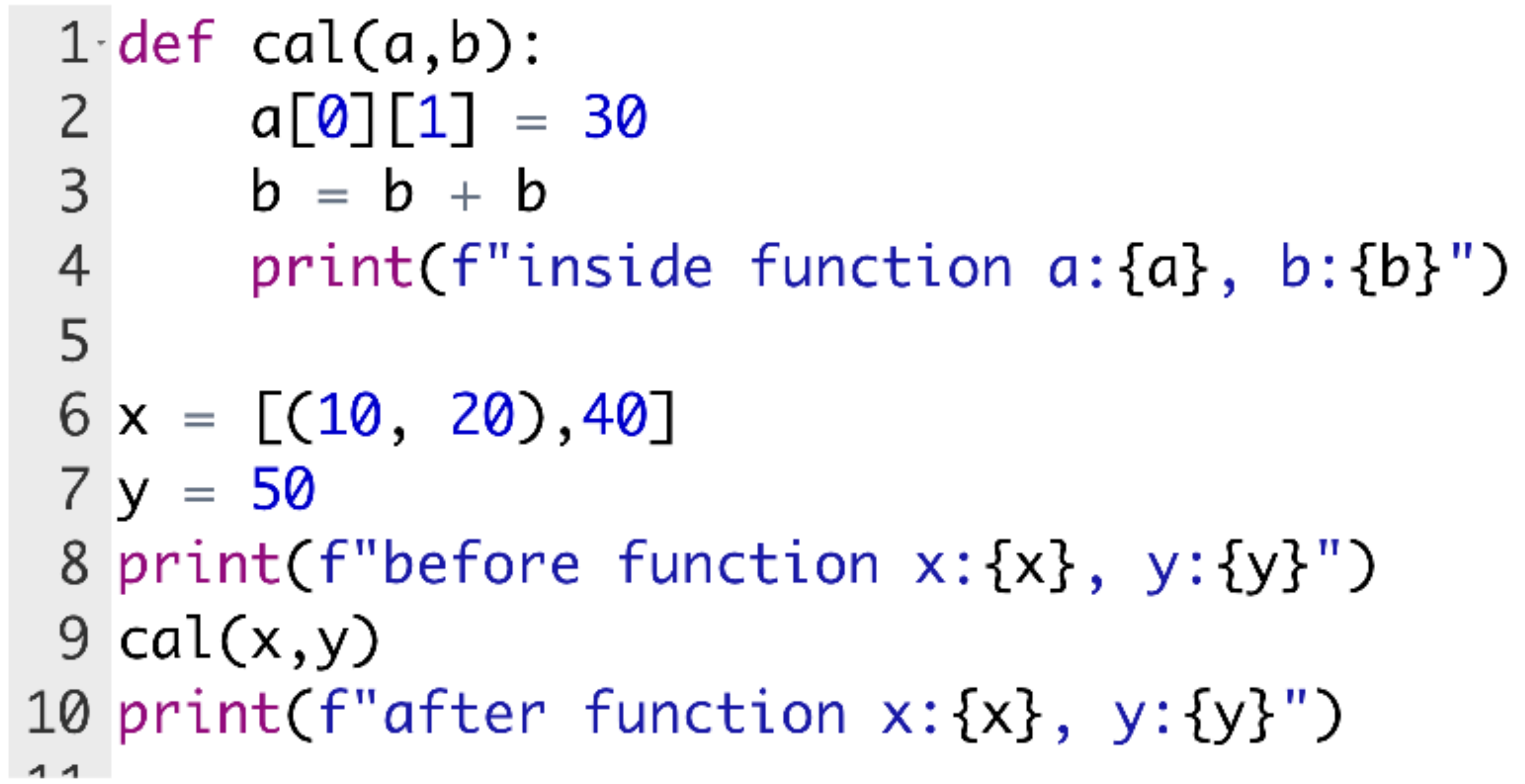
Quiz



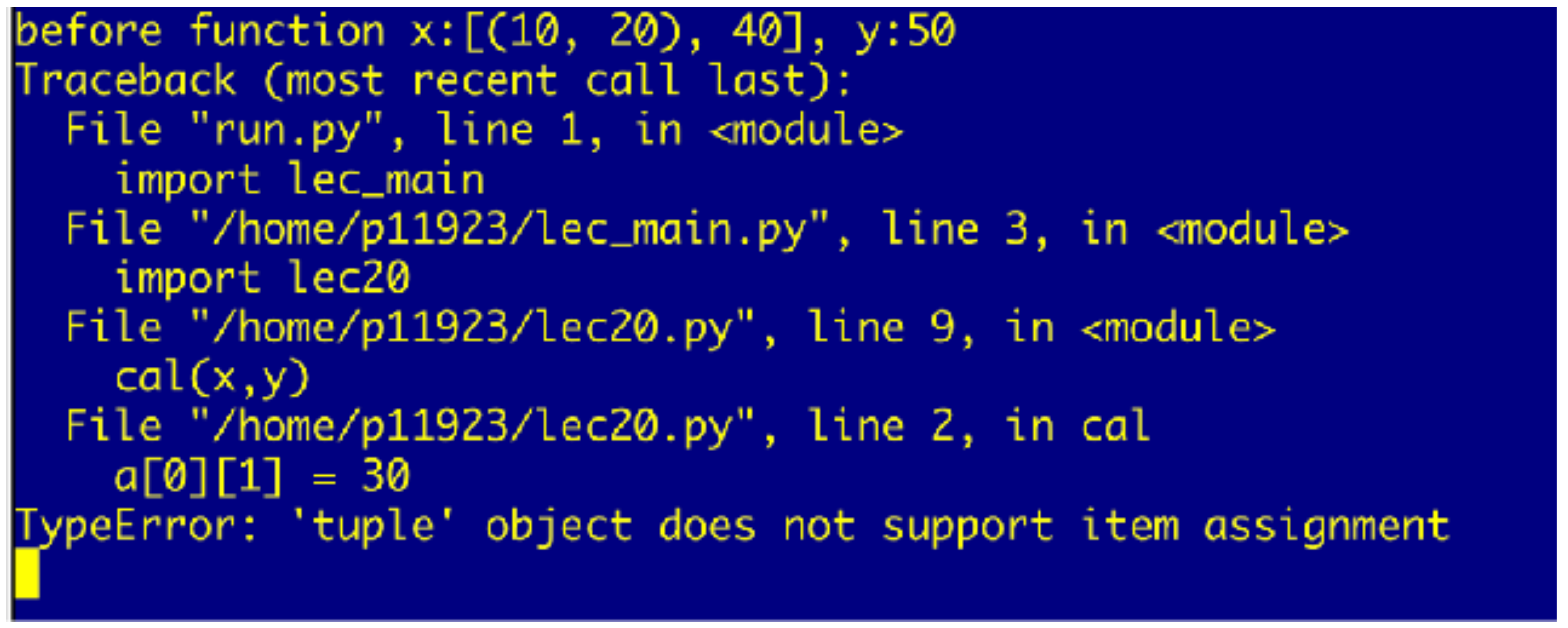
```python
def cal(a,b):
    a.append(10)
    b = b + b
    print(f"inside function a:{a}, b:{b}")

x = [10, 20]
y = 50
print(f"before function x:{x}, y:{y}")
cal(x,y)
print(f"after function x:{x}, y:{y}")
```



```
before function x:[10, 20], y:50
inside function a:[10, 20, 10], b:100
after function x:[10, 20, 10], y:50
```

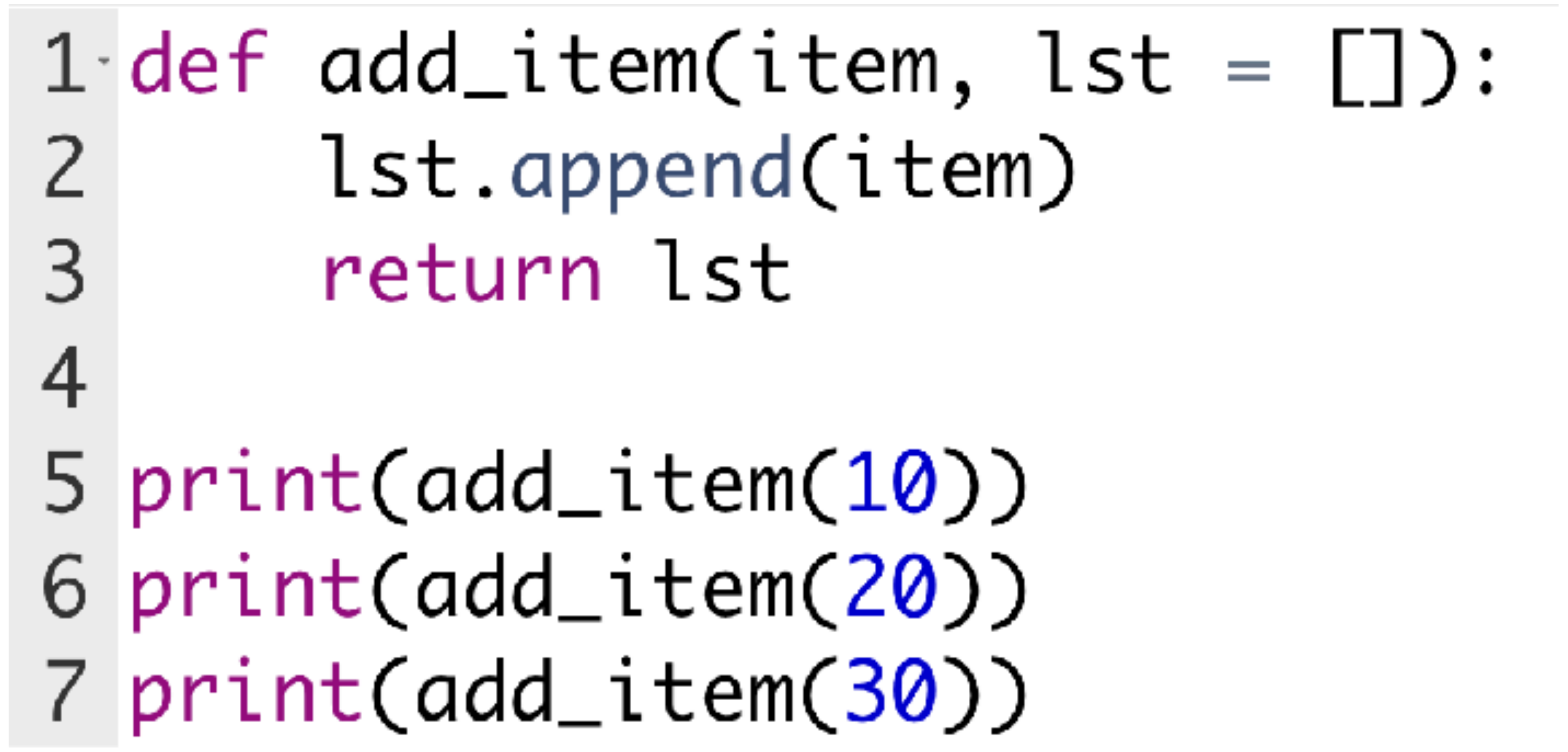


```python
def cal(a,b):
    a[0][1] = 30
    b = b + b
    print(f"inside function a:{a}, b:{b}")

x = [(10, 20),40]
y = 50
print(f"before function x:{x}, y:{y}")
cal(x,y)
print(f"after function x:{x}, y:{y}")
```



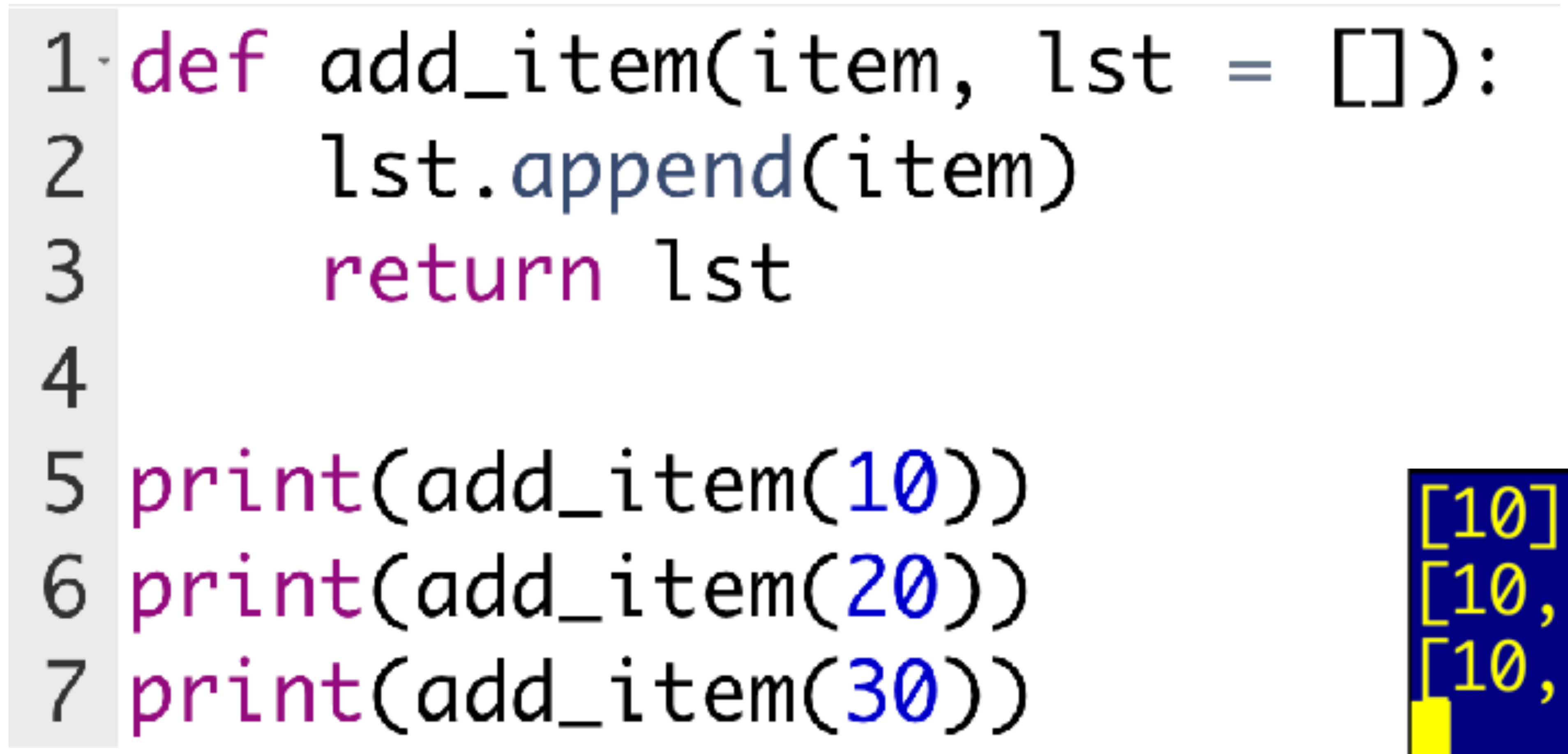
```
before function x:[(10, 20), 40], y:50
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p11923/lec_main.py", line 3, in <module>
    import lec20
  File "/home/p11923/lec20.py", line 9, in <module>
    cal(x,y)
  File "/home/p11923/lec20.py", line 2, in cal
    a[0][1] = 30
TypeError: 'tuple' object does not support item assignment
```

Quiz

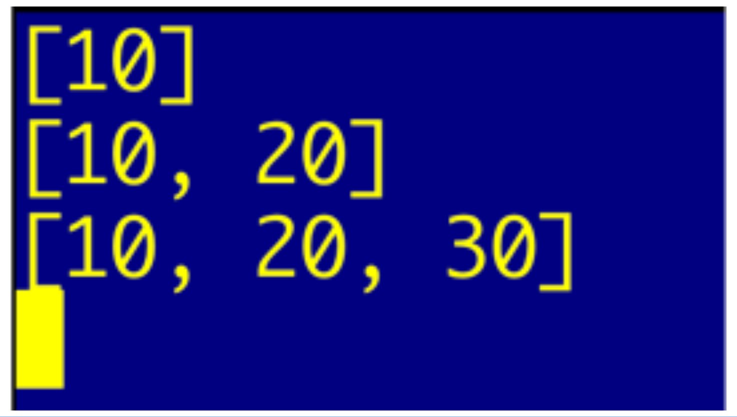


```python
def add_item(item, lst = []):
    lst.append(item)
    return lst

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

Quiz



```python
def add_item(item, lst = []):
    lst.append(item)
    return lst

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

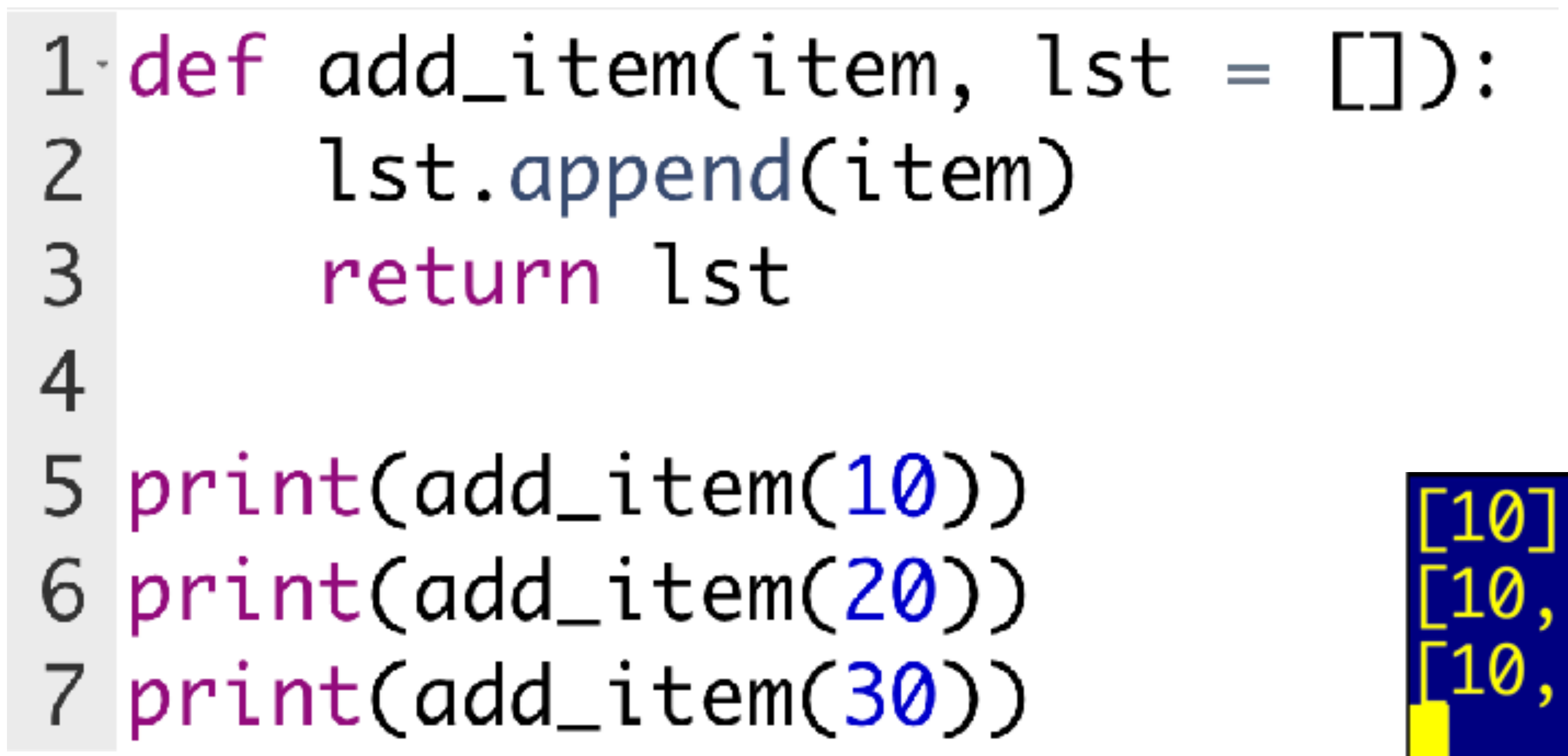


```
[10]
[10, 20]
[10, 20, 30]
```

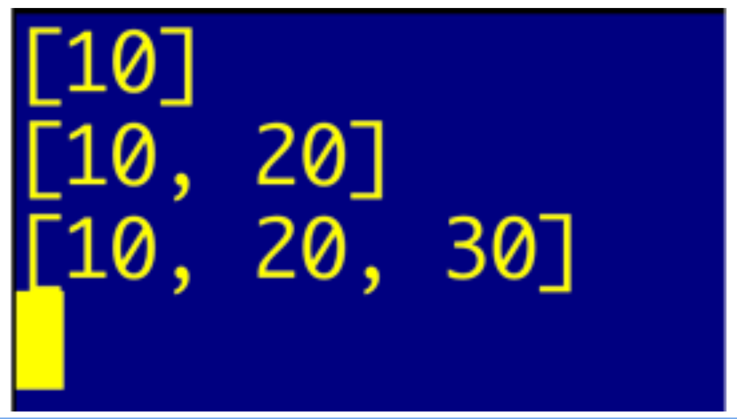
mutable default arguments (like lists/dicts) are shared across calls.

There is **one list object** created when Python defined the function. Every time you don’t pass a list, the parameter lst references that same list object. So each call mutates the same object

Quiz



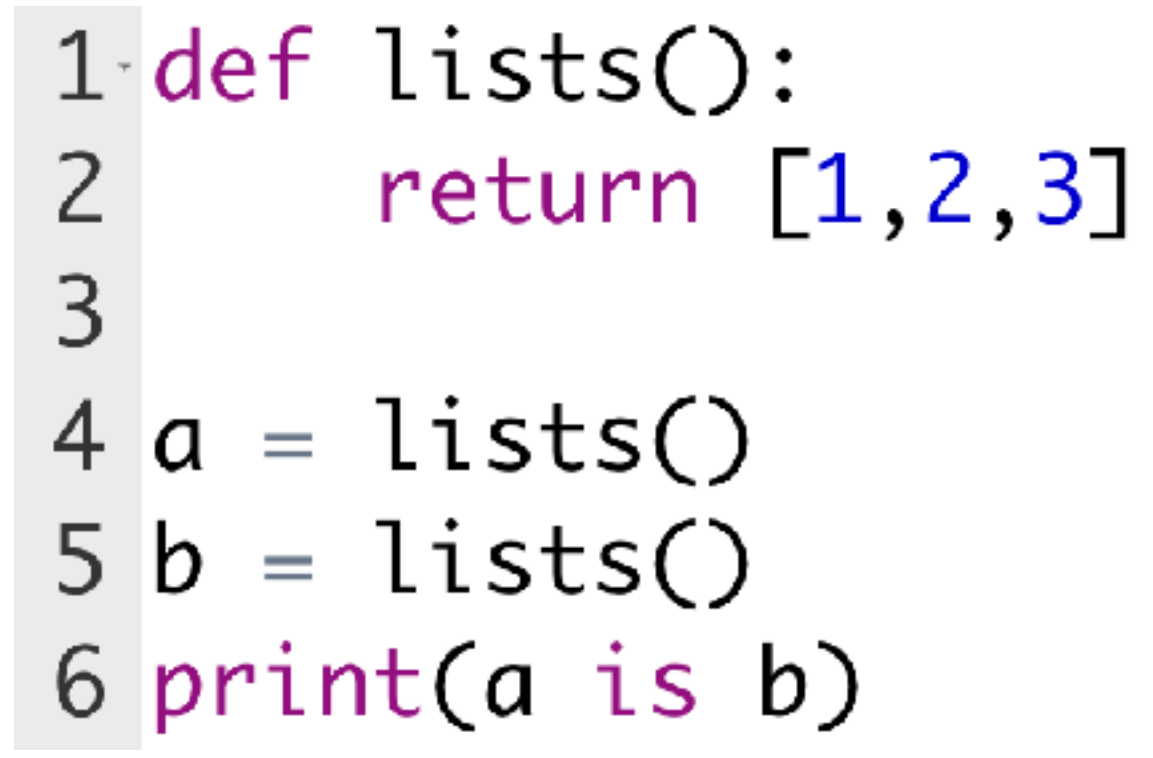
```python
def add_item(item, lst = []):
    lst.append(item)
    return lst

print(add_item(10))
print(add_item(20))
print(add_item(30))
```



```
[10]
[10, 20]
[10, 20, 30]
```

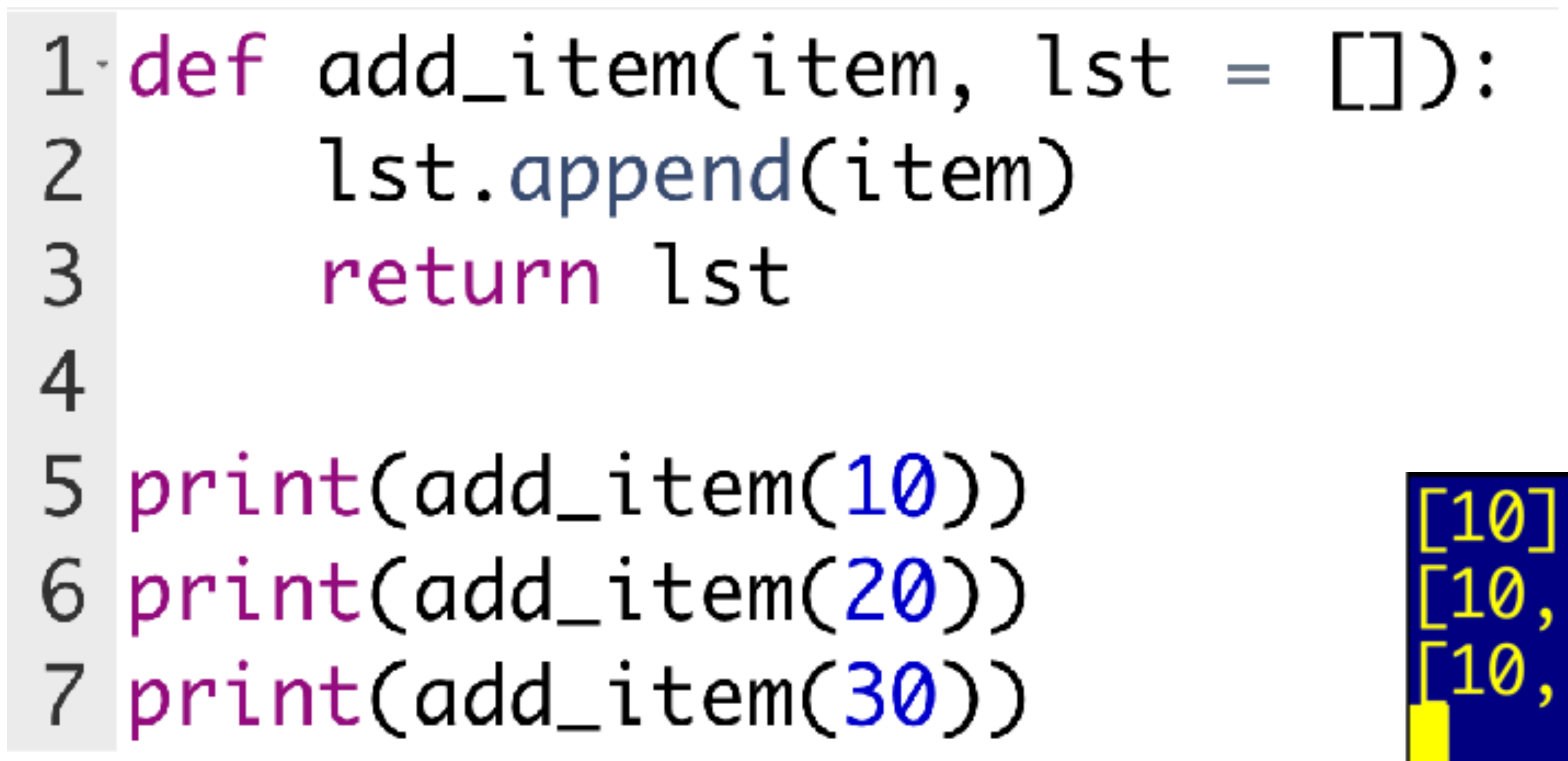
mutable default arguments (like lists/dicts) are shared across calls.

There is **one list object** created when Python defined the function. Every time you don't pass a list, the parameter lst references that same list object. So each call mutates the same object

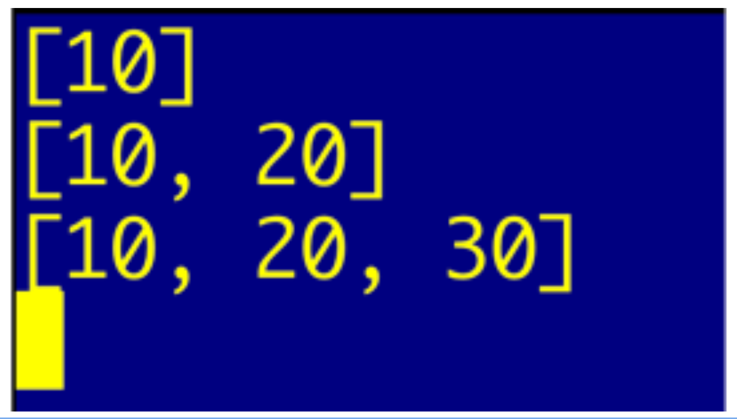


```python
def lists():
    return [1,2,3]

a = lists()
b = lists()
print(a is b)
```

Quiz



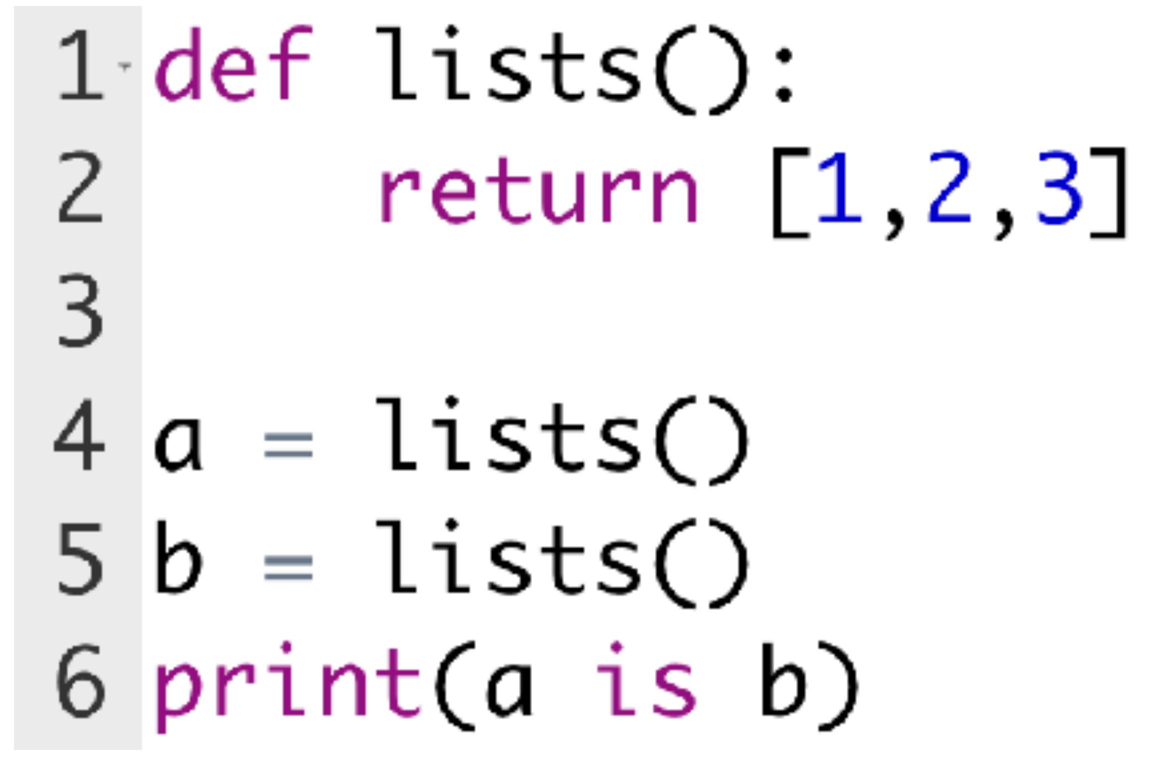
```python
def add_item(item, lst = []):
    lst.append(item)
    return lst

print(add_item(10))
print(add_item(20))
print(add_item(30))
```



```
[10]
[10, 20]
[10, 20, 30]
```

mutable default arguments (like lists/dicts) are shared across calls.

There is **one list object** created when Python defined the function. Every time you don't pass a list, the parameter lst references that same list object. So each call mutates the same object

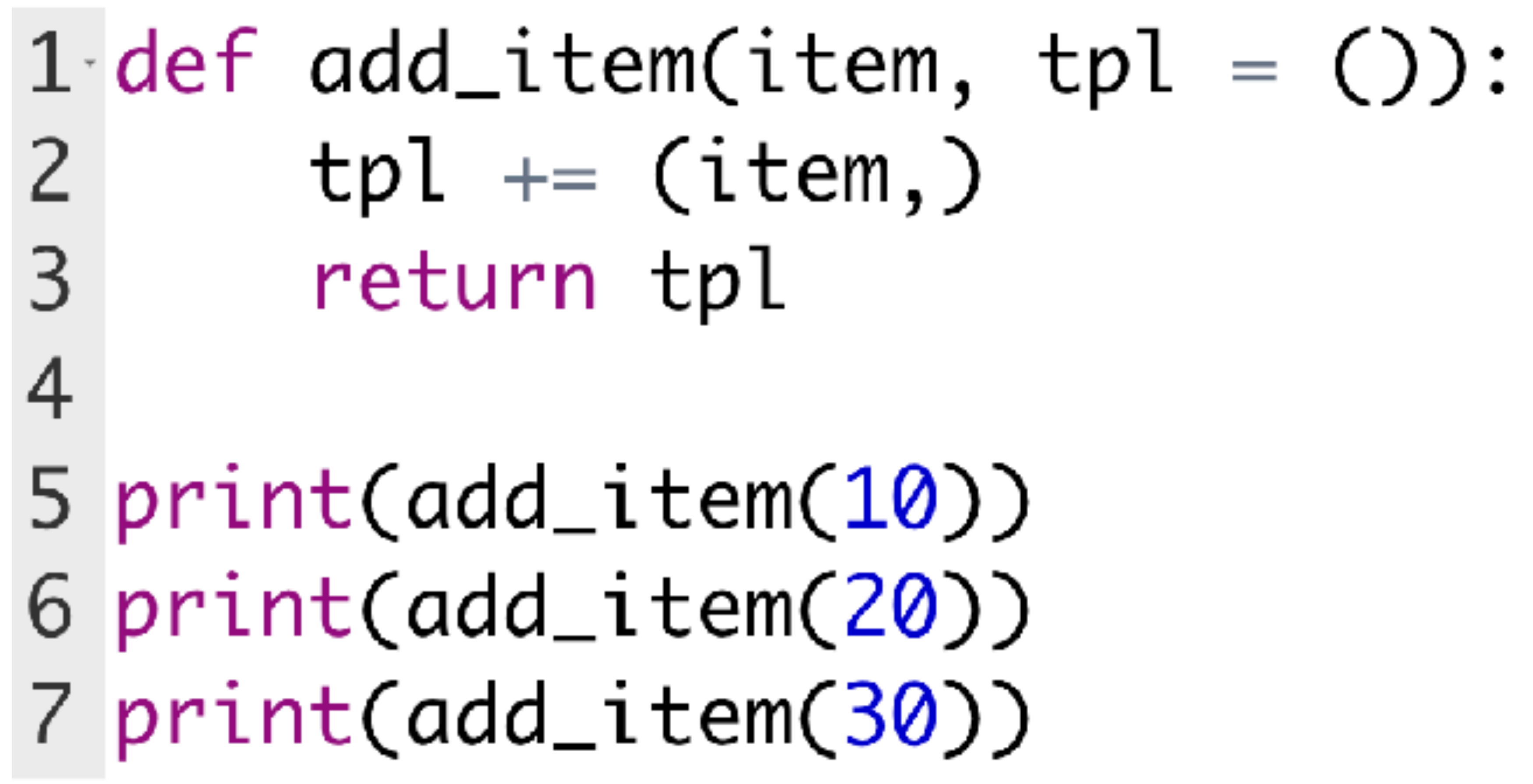


```python
def lists():
    return [1,2,3]

a = lists()
b = lists()
print(a is b)
```

Don't get confuse. Here, in each call, we are creating a new object.

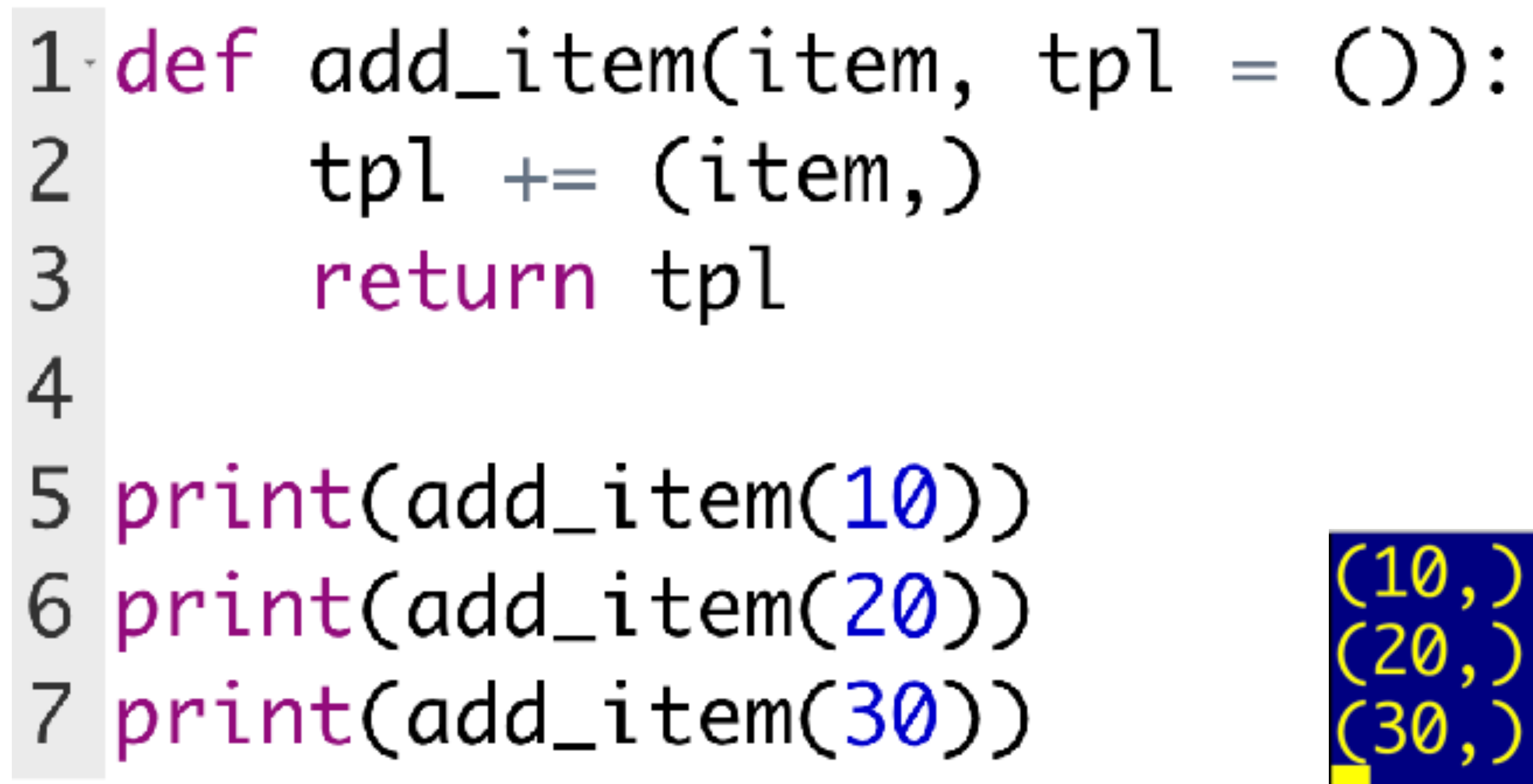
False

Quiz

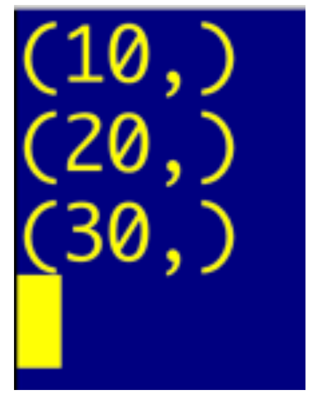


```python
def add_item(item, tpl = ()):
    tpl += (item,)
    return tpl

print(add_item(10))
print(add_item(20))
print(add_item(30))
```

Quiz



```python
def add_item(item, tpl = ()):
    tpl += (item,)
    return tpl

print(add_item(10))
print(add_item(20))
print(add_item(30))
```



```
(10,)
(20,)
(30,)
```

Type Annotations!

Python is **dynamically typed**, but since Python 3.5+, you can add **type hints** (annotations) to improve readability and help tools like *mypy* or IDEs check types.

Type annotations: Provide hints about parameter and return types but are not enforced at runtime. Tools like mypy and IDEs enforce it.

```
def add(x: int, y: int) -> int:
    return x + y
```