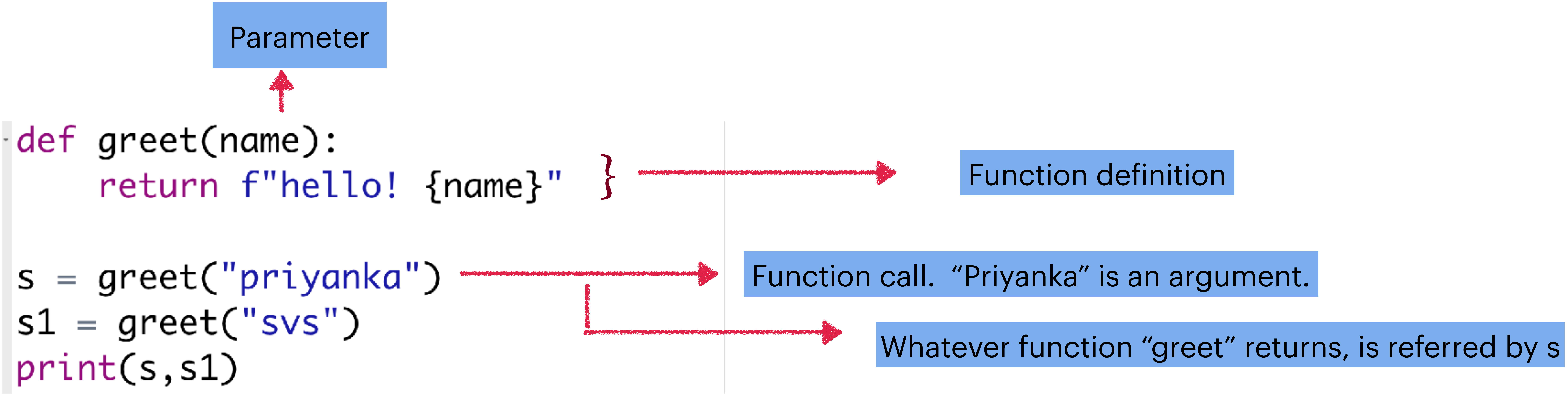
COL1000
Introduction to Programming

Priyanka Golia

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

Recall: Functions



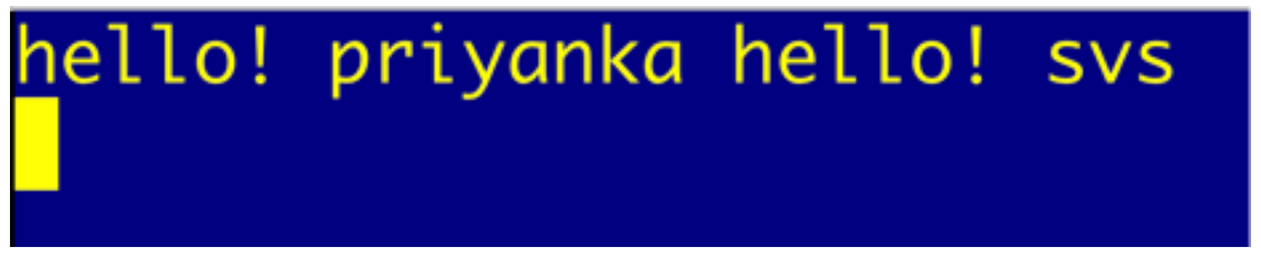
Parameter

```
def greet(name):
    return f"hello! {name}"

s = greet("priyanka")
s1 = greet("svs")
print(s,s1)
```

Function definition

Function call. “Priyanka” is an argument.

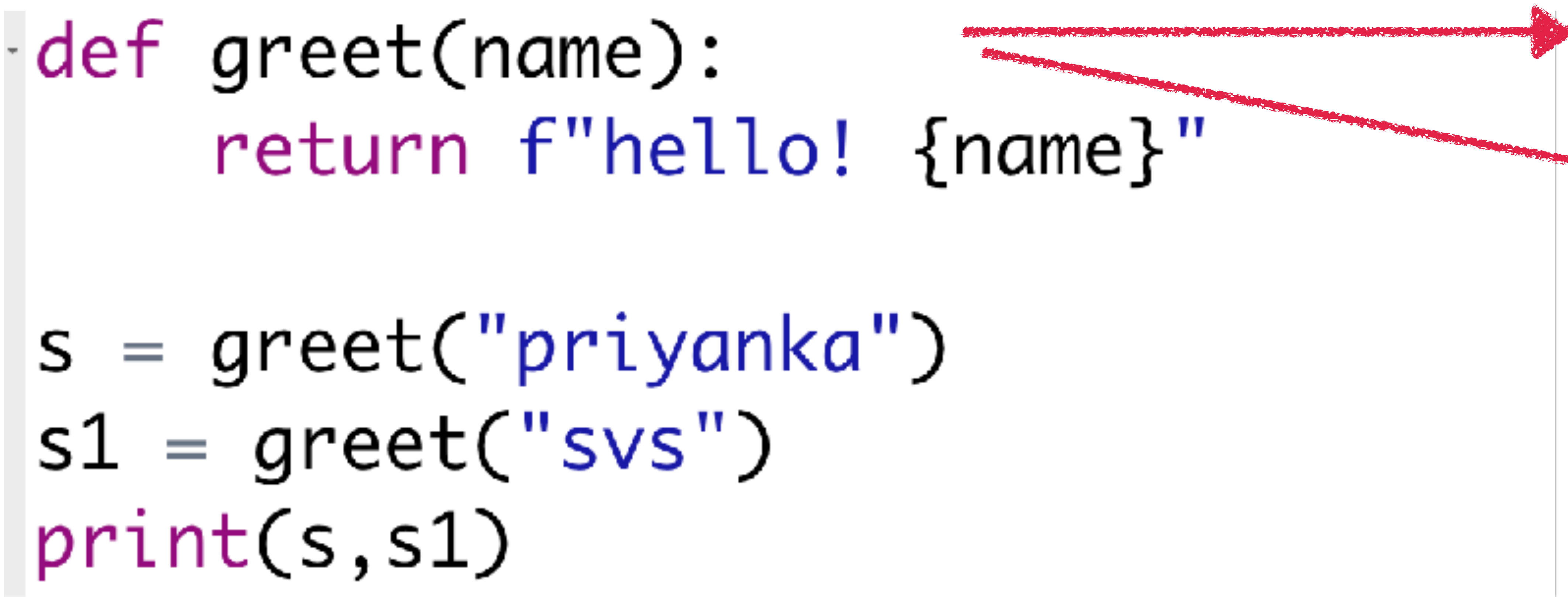
Whatever function “greet” returns, is referred by s



```
hello! priyanka hello! svs
```

- **def** starts a function definition.
- greet is the **function name** (same rules as variable names).
- (name) are the **parameters**.
- **:** ends the header line; the indented block is the function body.
- **return** sends a value back to the caller (optional).

Recall: Functions



```python
def greet(name):
    return f"hello! {name}"

s = greet("priyanka")
s1 = greet("svs")
print(s,s1)
```

Creates a function object.

The “greet” name is referring to that function.

```python
print(type(greet))   ##<class 'function'>
```

Recall: Functions

```
def outer():
    x = 10
    def inner():
        return x + 1
    return inner
outer()
```

- When outer is *defined*, Python just creates a function object. The code for inner is *inside* that function but not executed yet.

- Everytime outer is *called*, a **new local frame** is created with its own variable x = 10.

- Inside outer, Python creates another function object inner. While creating inner, it sees that inner uses a variable x which is *not local* to it, but exists in an **enclosing scope** (outer's frame).

- So Python attaches a **reference** to that variable x — not its value, but *reference (think how it will behave for mutable and immutable objects)*.

- When outer() returns, it returns the inner function object.
  But that returned function still carries with it a reference to the variable x — even though outer has finished executing. That's called "closure" — A closure is a function that "remembers" variables from its enclosing scope, even after that scope is gone.

Recall: Functions

```
def outer():
    x = 10
    def inner():
        return x + 1
    return inner

f = outer()
print(f())
print(f())
```

```
outer() frame:
    x → 10
    inner → function object
          └── closure cell referencing x
```

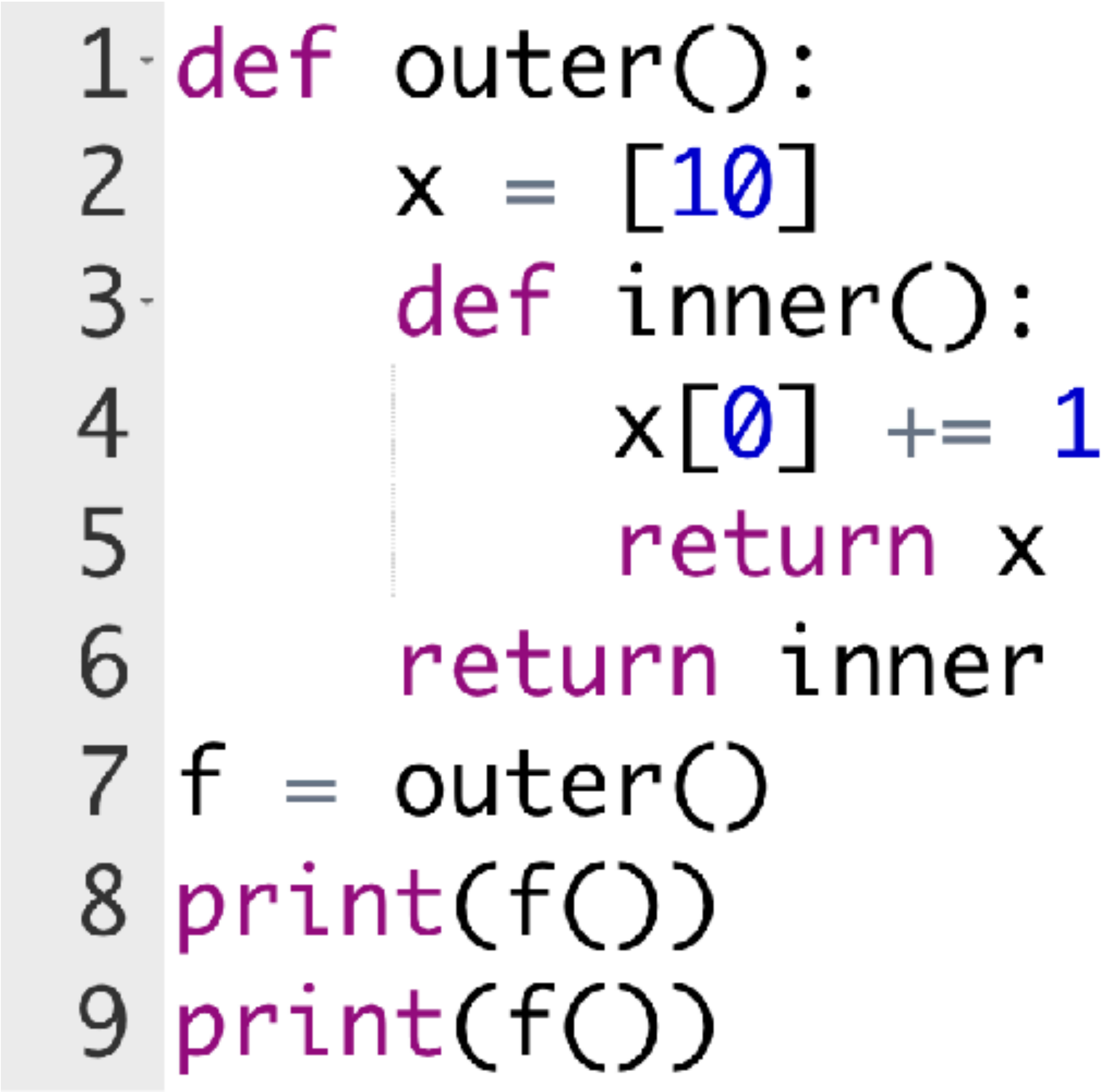
Recall: Functions

```
def outer():
    x = 10
    def inner():
        return x + 1
    return inner

f = outer()
print(f())
print(f())
```

```
11
11
```

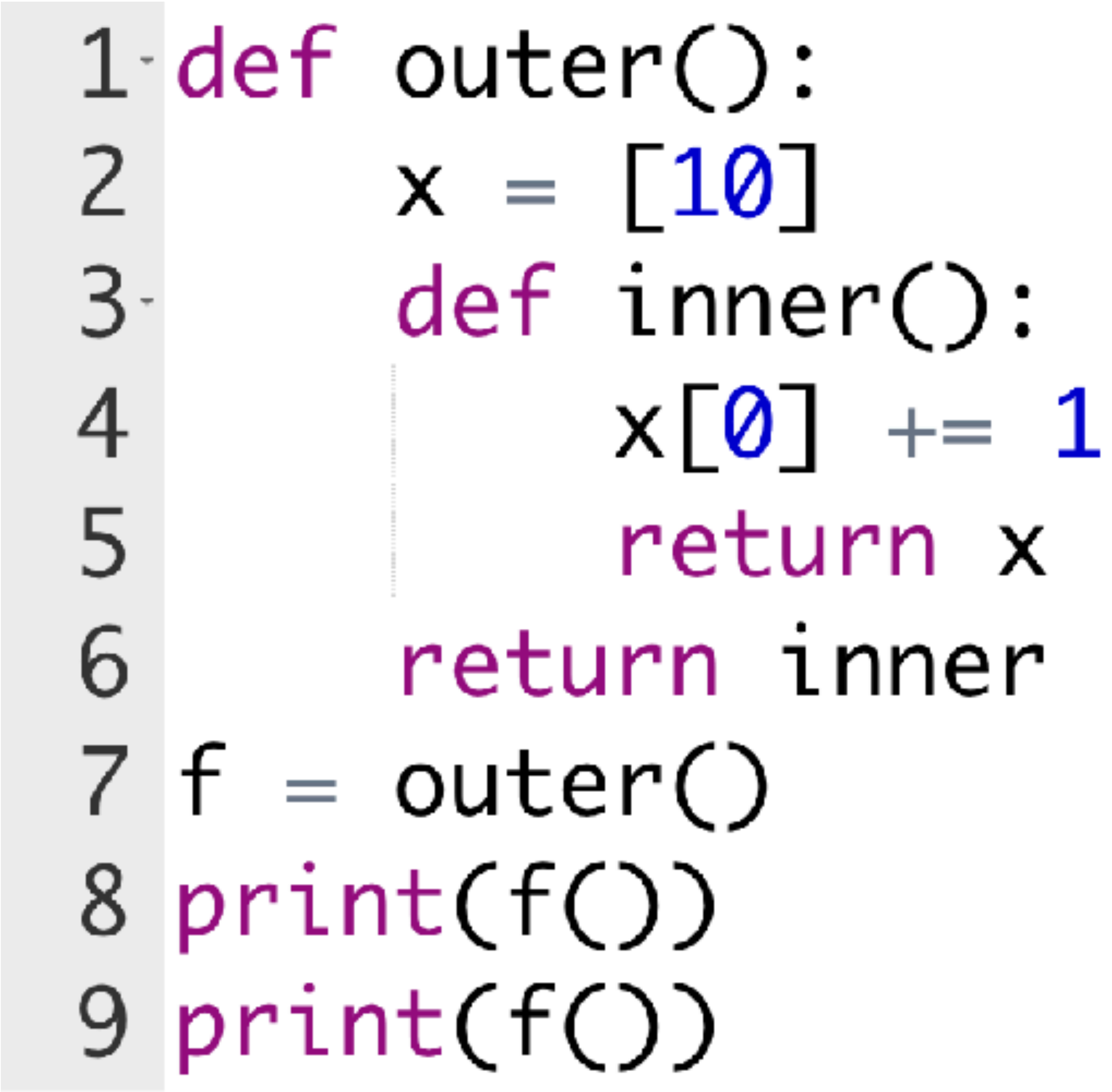
```
outer() frame:
    x → 10
    inner → function object
            └── closure cell referencing x
```

Recall: Functions



```python
def outer():
    x = [10]
    def inner():
        x[0] += 1
        return x
    return inner
f = outer()
print(f())
print(f())
```

Recall: Functions

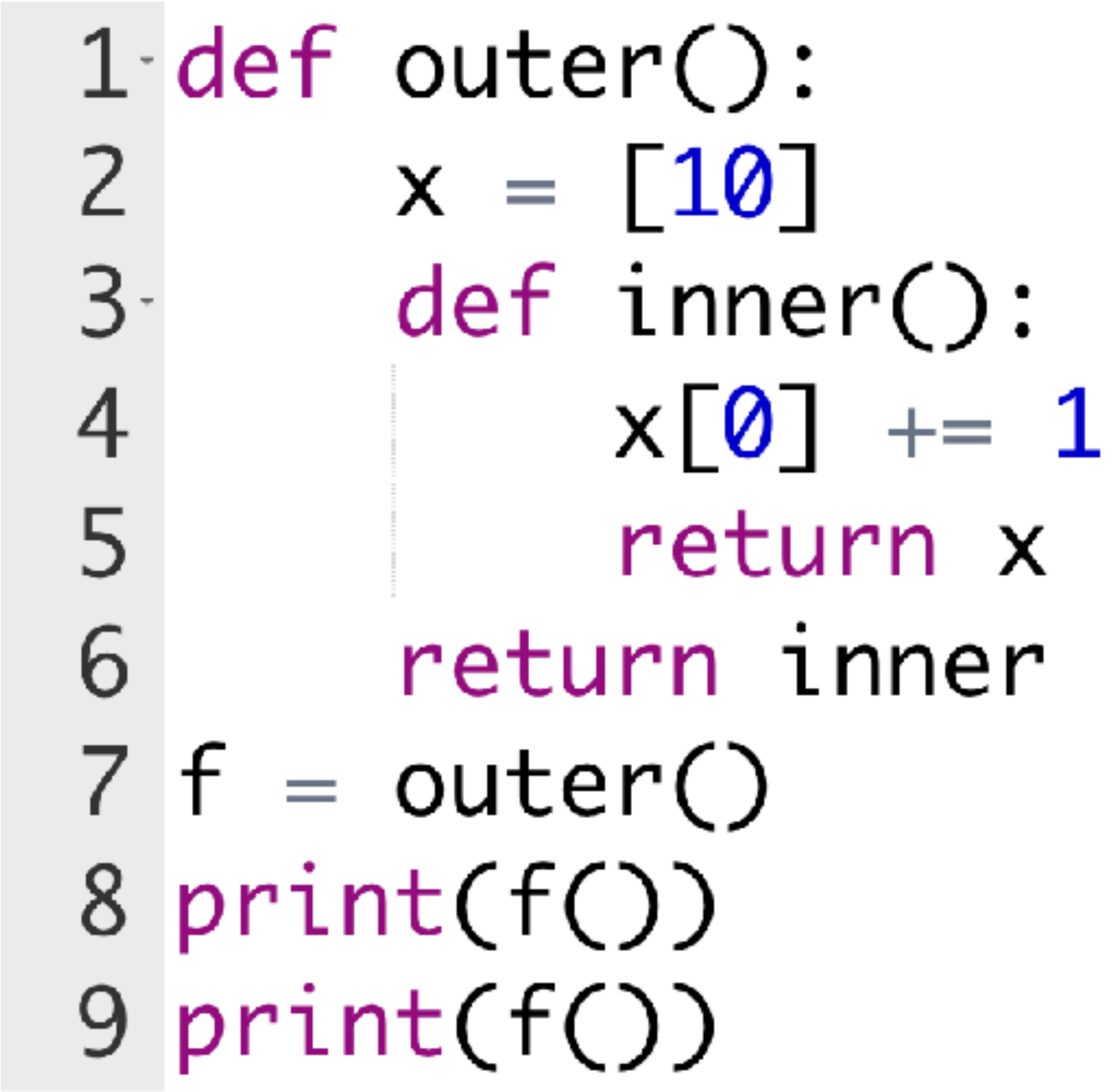


```python
def outer():
    x = [10]
    def inner():
        x[0] += 1
        return x
    return inner
f = outer()
print(f())
print(f())
```

```
[11]
[12]
```

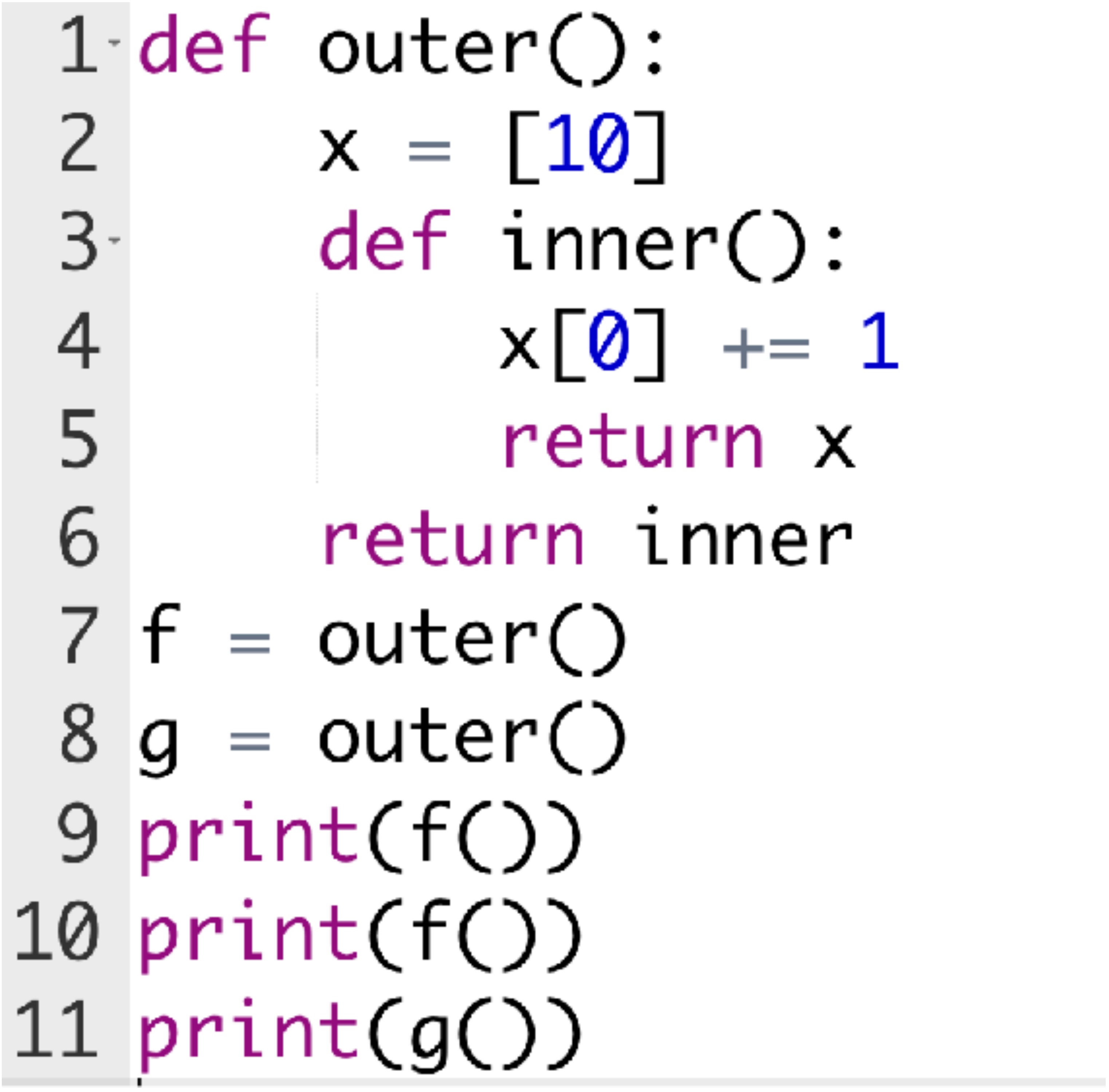
x is mutable, changes inside the inner function affect the same underlying object.

Recall: Functions



```python
def outer():
    x = [10]
    def inner():
        x[0] += 1
        return x
    return inner
f = outer()
print(f())
print(f())
```

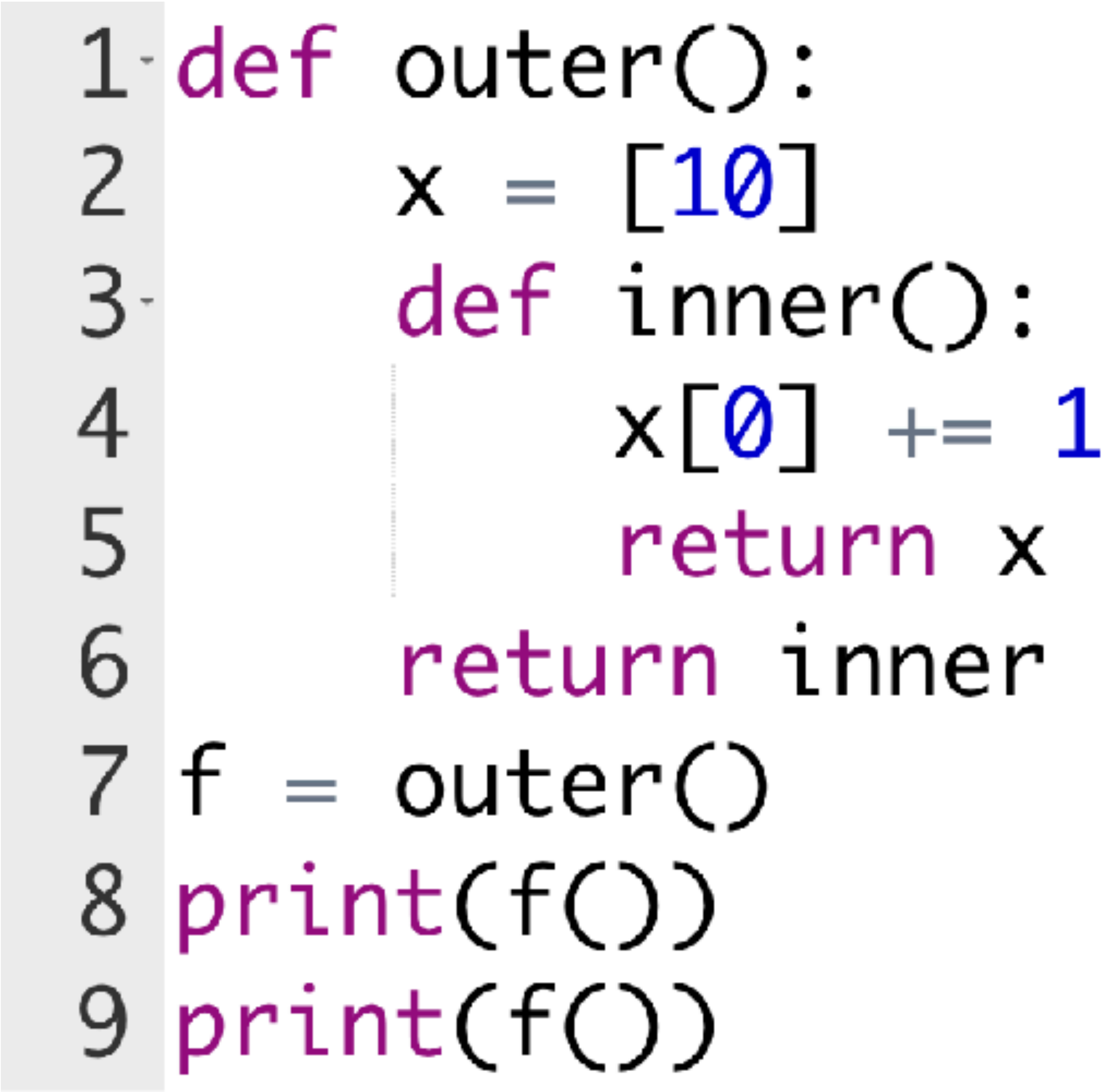
```
[11]
[12]
```



```python
def outer():
    x = [10]
    def inner():
        x[0] += 1
        return x
    return inner
f = outer()
g = outer()
print(f())
print(f())
print(g())
```

x is mutable, changes inside the inner function affect the same underlying object.

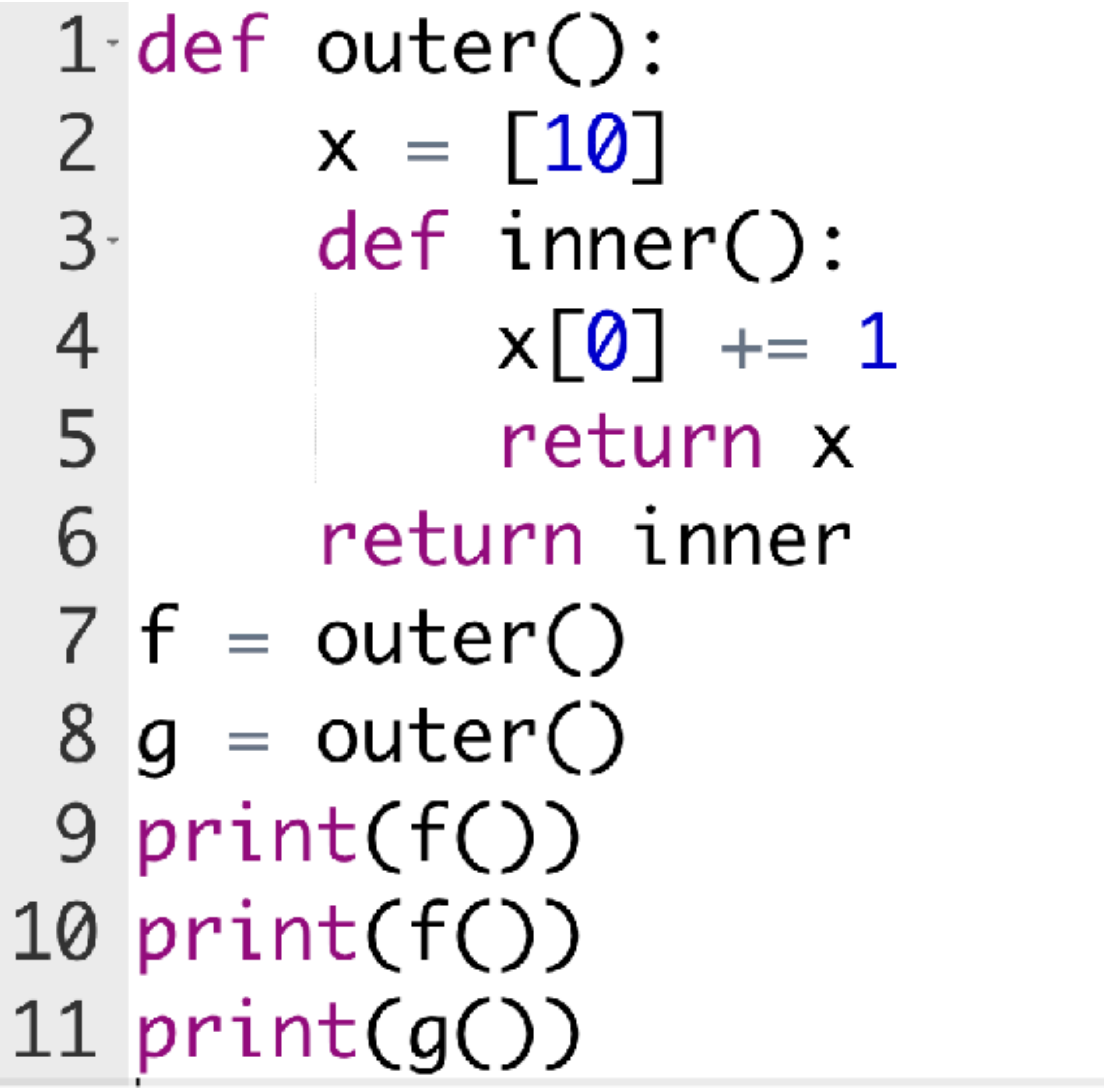
Recall: Functions



```python
def outer():
    x = [10]
    def inner():
        x[0] += 1
        return x
    return inner
f = outer()
print(f())
print(f())
```

[11]
[12]

x is mutable, changes inside the inner function affect the same underlying object.



```python
def outer():
    x = [10]
    def inner():
        x[0] += 1
        return x
    return inner
f = outer()
g = outer()
print(f())
print(f())
print(g())
```

[11]
[12]
[11]

F and g have independent internal states.

Revision!!

A function can return another function

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(f())
```

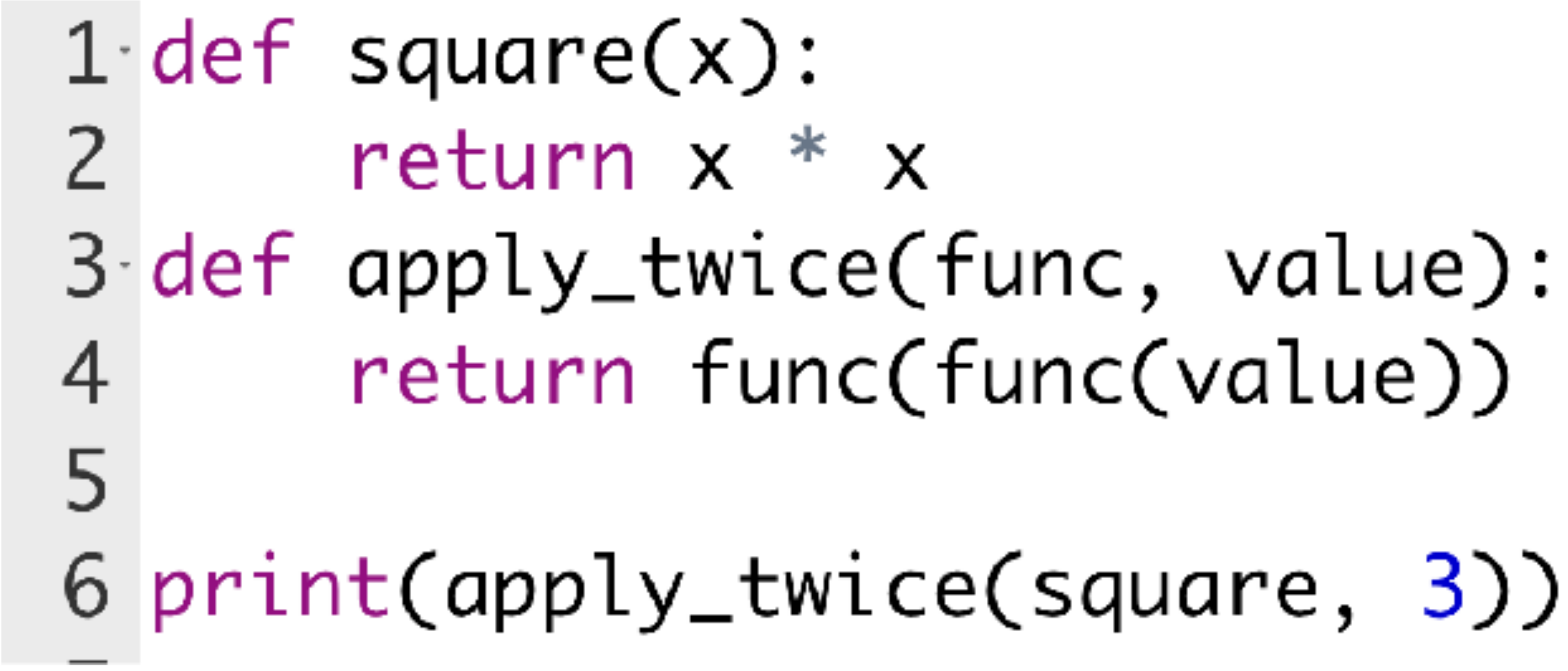
```
def power_factory(n):
    def power(x):
        return x ** n
    return power

square = power_factory(2)
cube = power_factory(3)

print(square(5))   # 25
print(cube(2))     # 8
```

The function power_factory *creates* and *returns* new functions — one that squares, one that cubes.
Each returned function remembers the value of n used when it was created.

A closure is a function that remembers the environment in which it was created, even after that environment is gone

Higher Order Functions

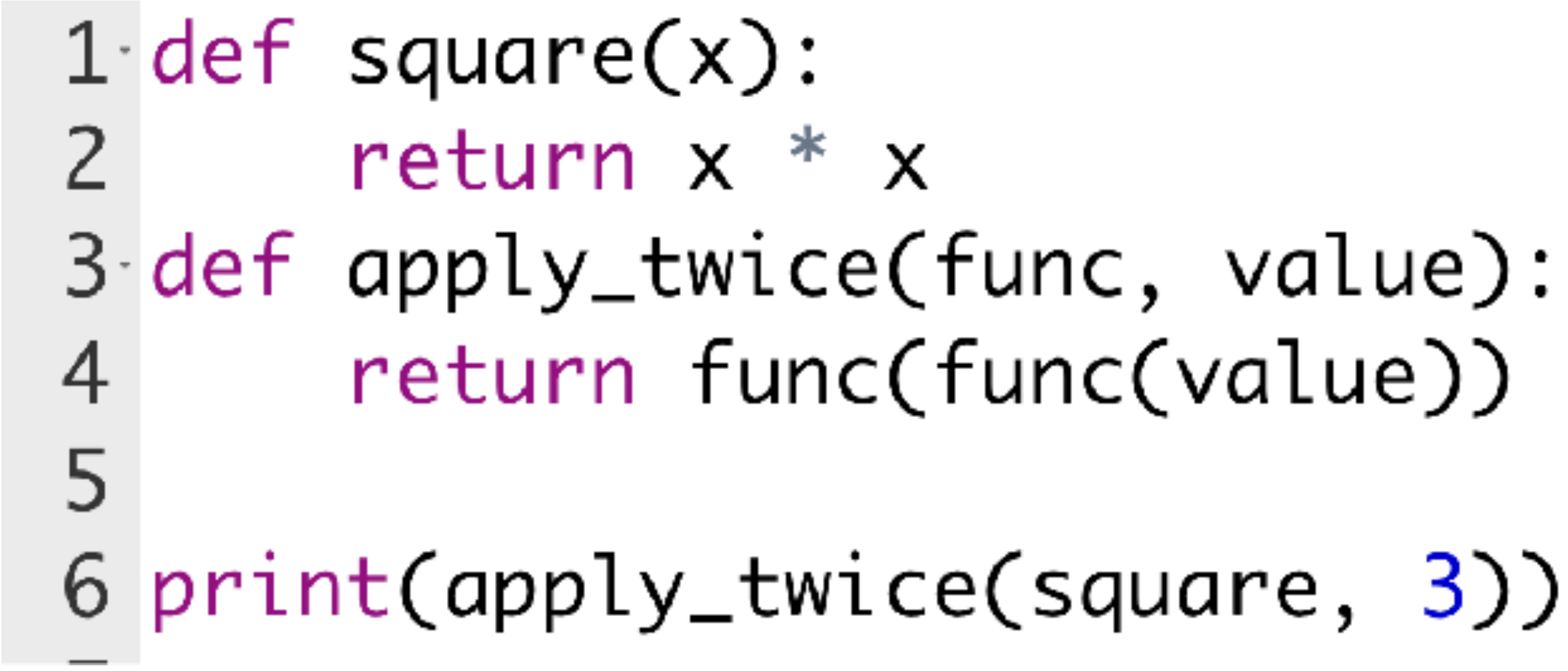
A higher-order function is a function that takes another function as input, or **returns a function as output**, or both



```python
def square(x):
    return x * x
def apply_twice(func, value):
    return func(func(value))

print(apply_twice(square, 3))
```

Higher Order Functions

A higher-order function is a function that takes another function as input, or **returns a function as output**, or both

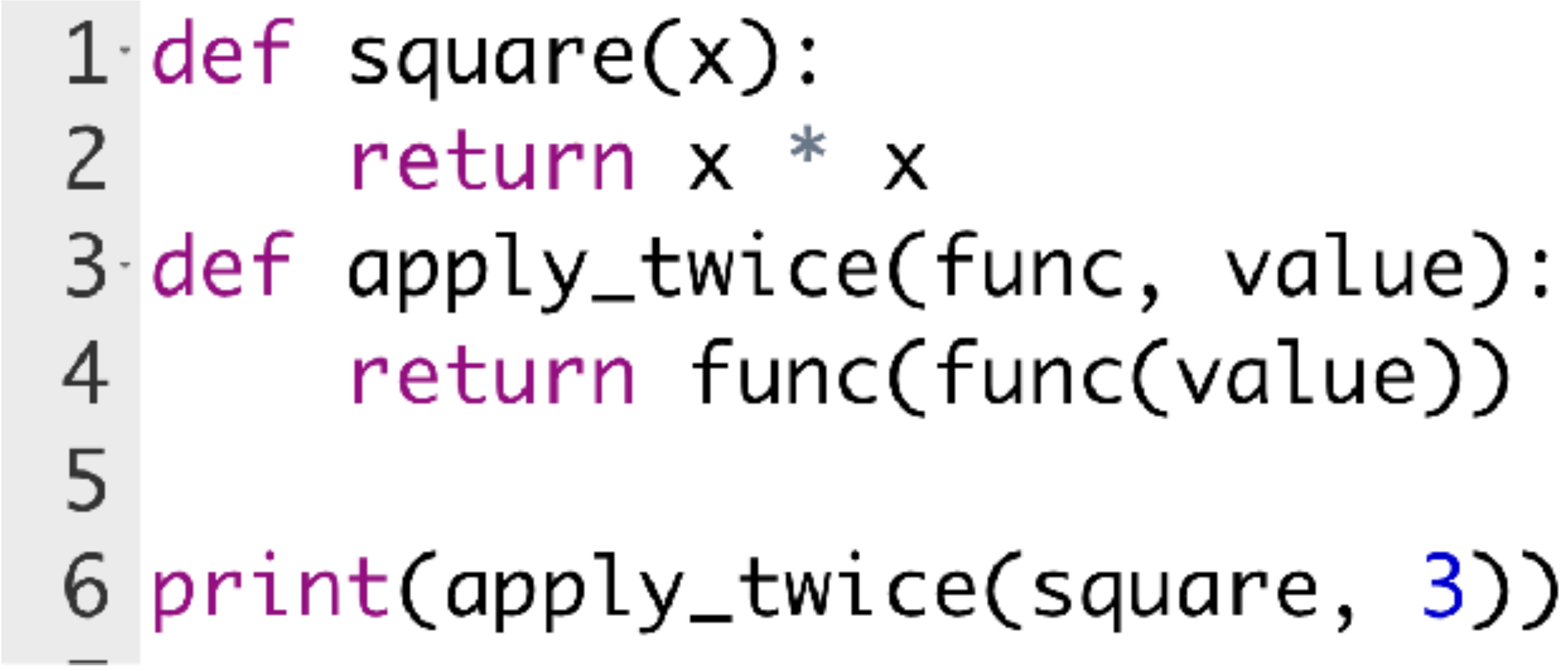


```
1 def square(x):
2     return x * x
3 def apply_twice(func, value):
4     return func(func(value))
5
6 print(apply_twice(square, 3))
```

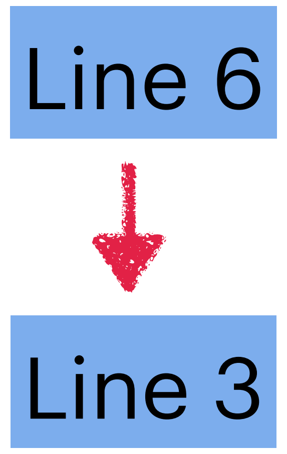
Line 6

Higher Order Functions

A higher-order function is a function that takes another function as input, or **returns a function as output**, or both



```
1 def square(x):
2     return x * x
3 def apply_twice(func, value):
4     return func(func(value))
5
6 print(apply_twice(square, 3))
```

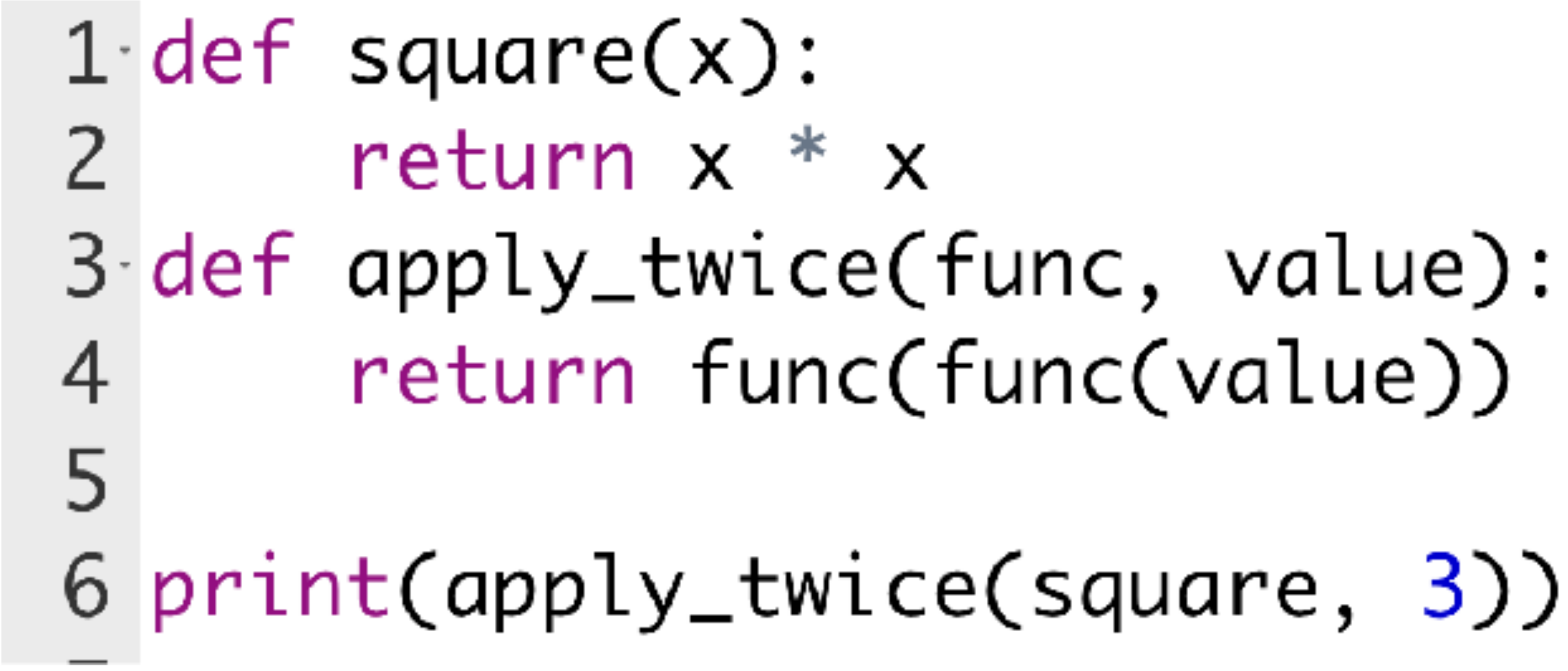


Line 6

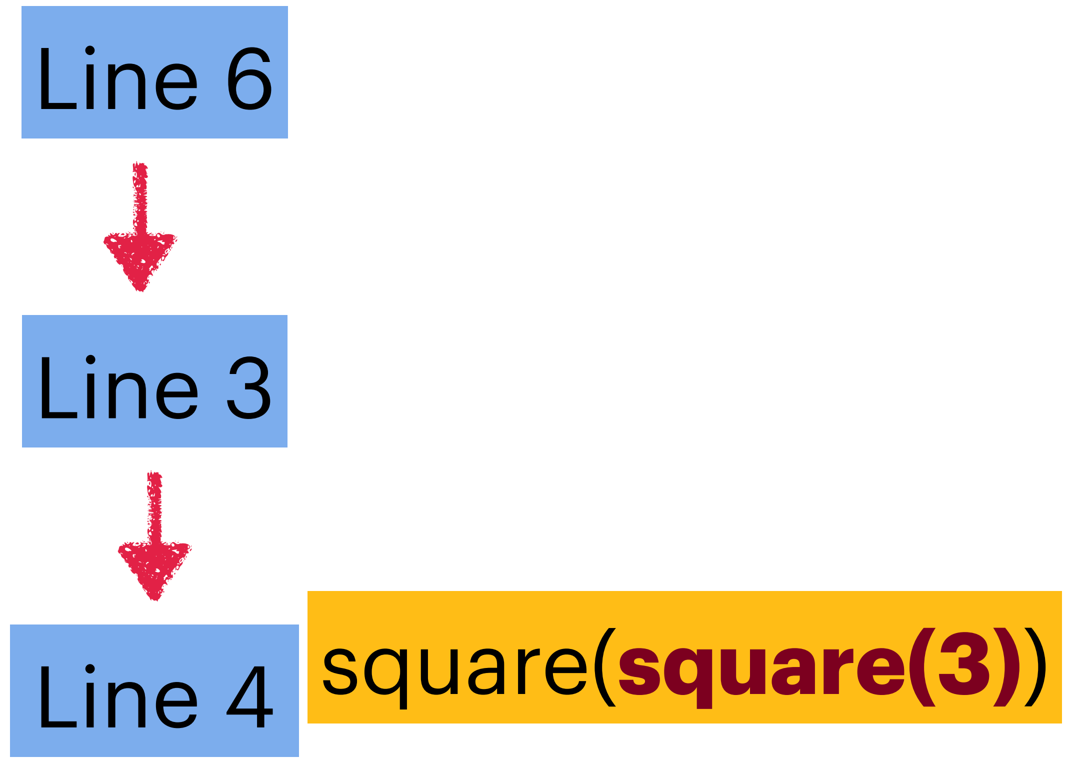
Line 3

Higher Order Functions

A higher-order function is a function that takes another function as input, or **returns a function as output**, or both



```
1 def square(x):
2     return x * x
3 def apply_twice(func, value):
4     return func(func(value))
5 
6 print(apply_twice(square, 3))
```



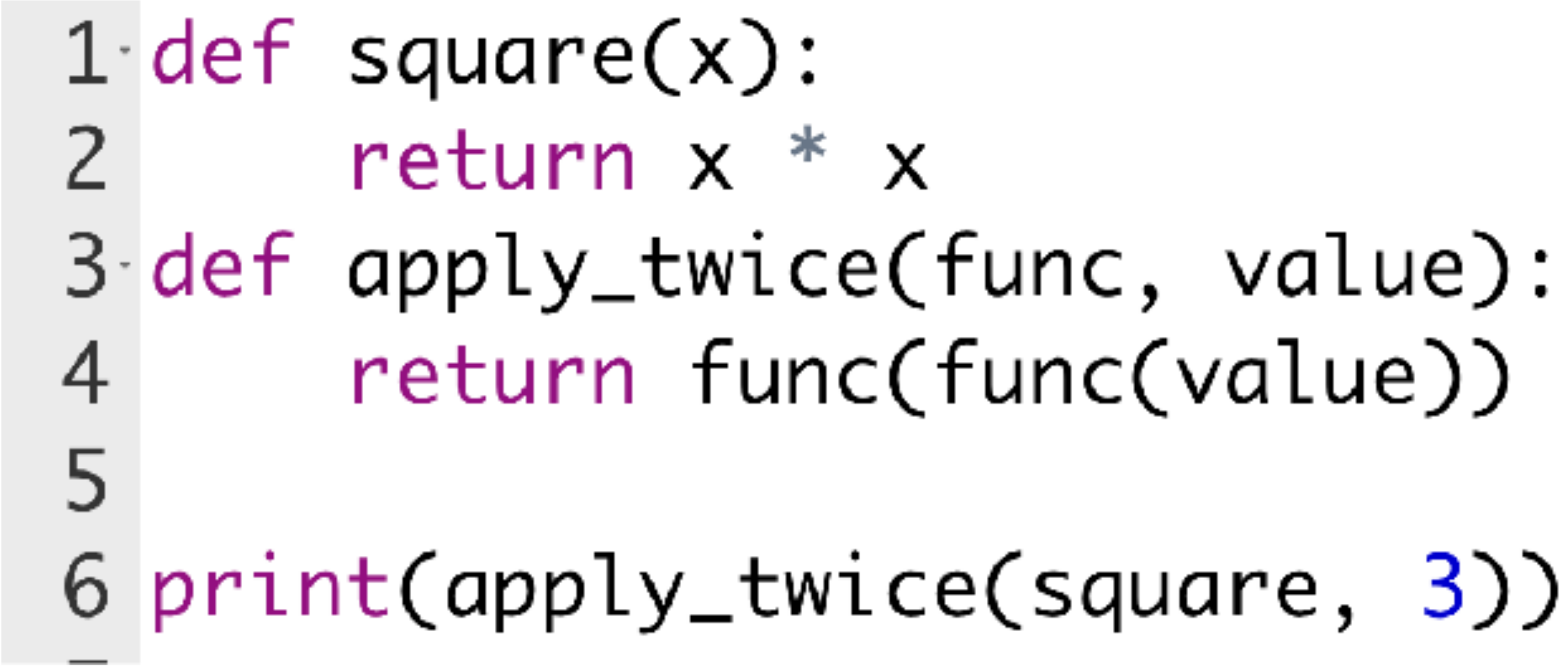
Line 6

Line 3

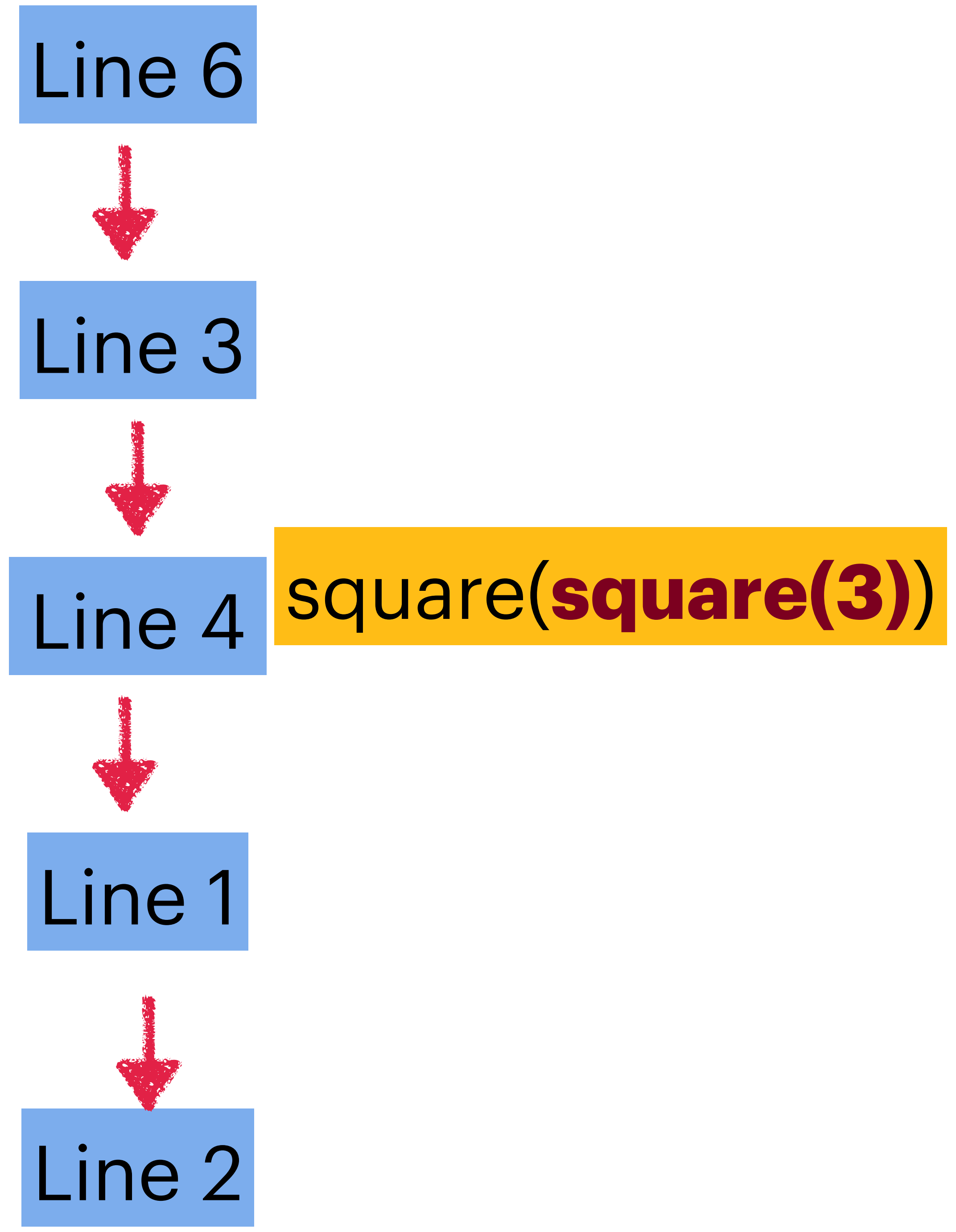
Line 4 square(**square(3)**)

Higher Order Functions

A higher-order function is a function that takes another function as input, or **returns a function as output**, or both



```
1 def square(x):
2     return x * x
3 def apply_twice(func, value):
4     return func(func(value))
5
6 print(apply_twice(square, 3))
```



Line 6

Line 3

Line 4

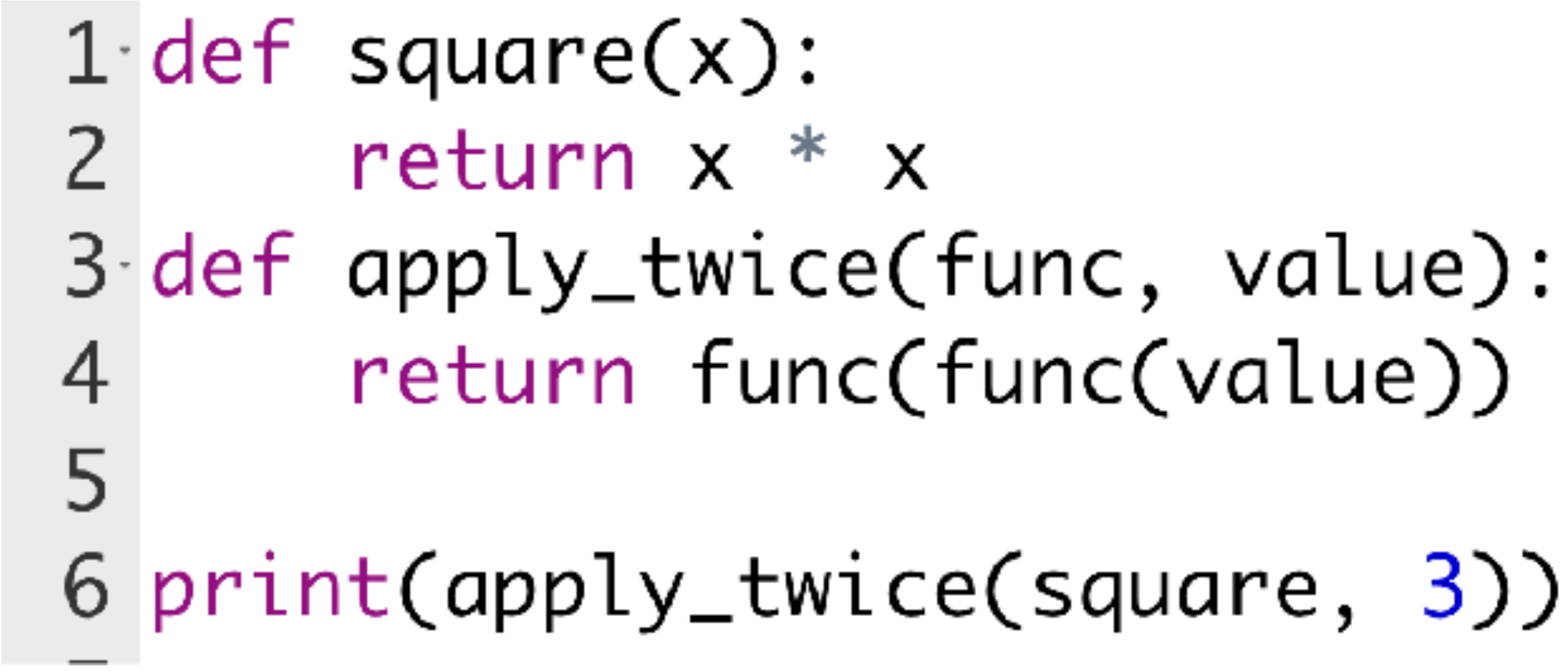
square(**square(3)**)

Line 1

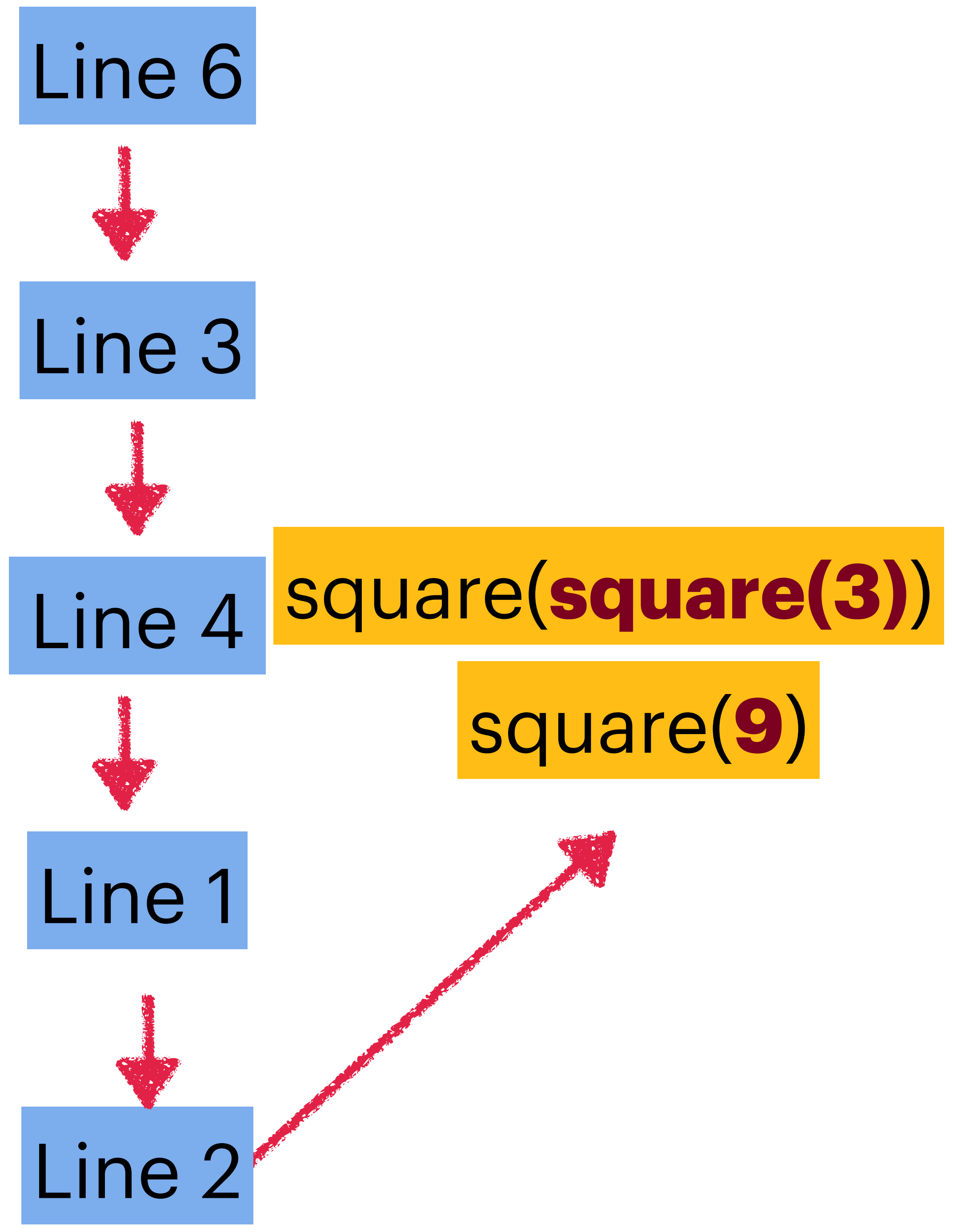
Line 2

Higher Order Functions

A higher-order function is a function that takes another function as input, or **returns a function as output**, or both



```
1 def square(x):
2     return x * x
3 def apply_twice(func, value):
4     return func(func(value))
5
6 print(apply_twice(square, 3))
```



Line 6

Line 3

Line 4 square(**square(3)**)

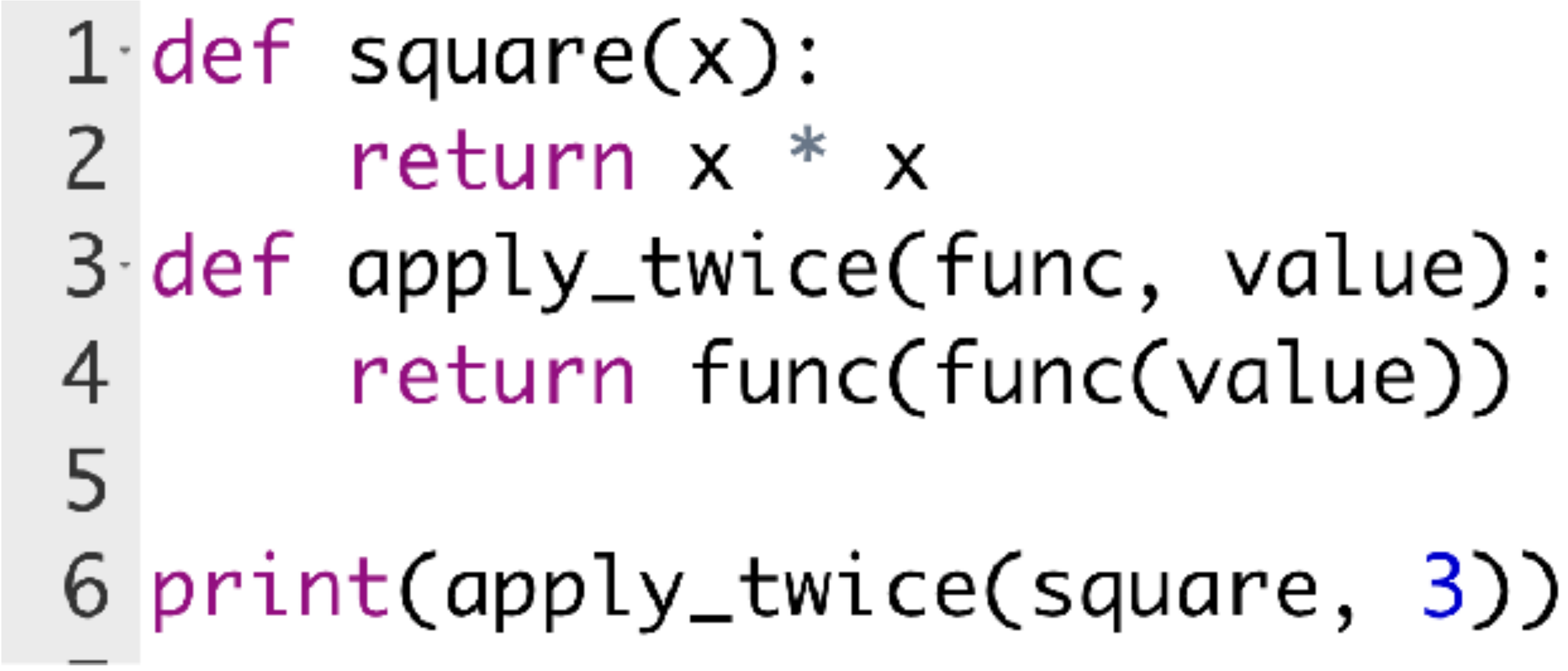
square(**9**)

Line 1

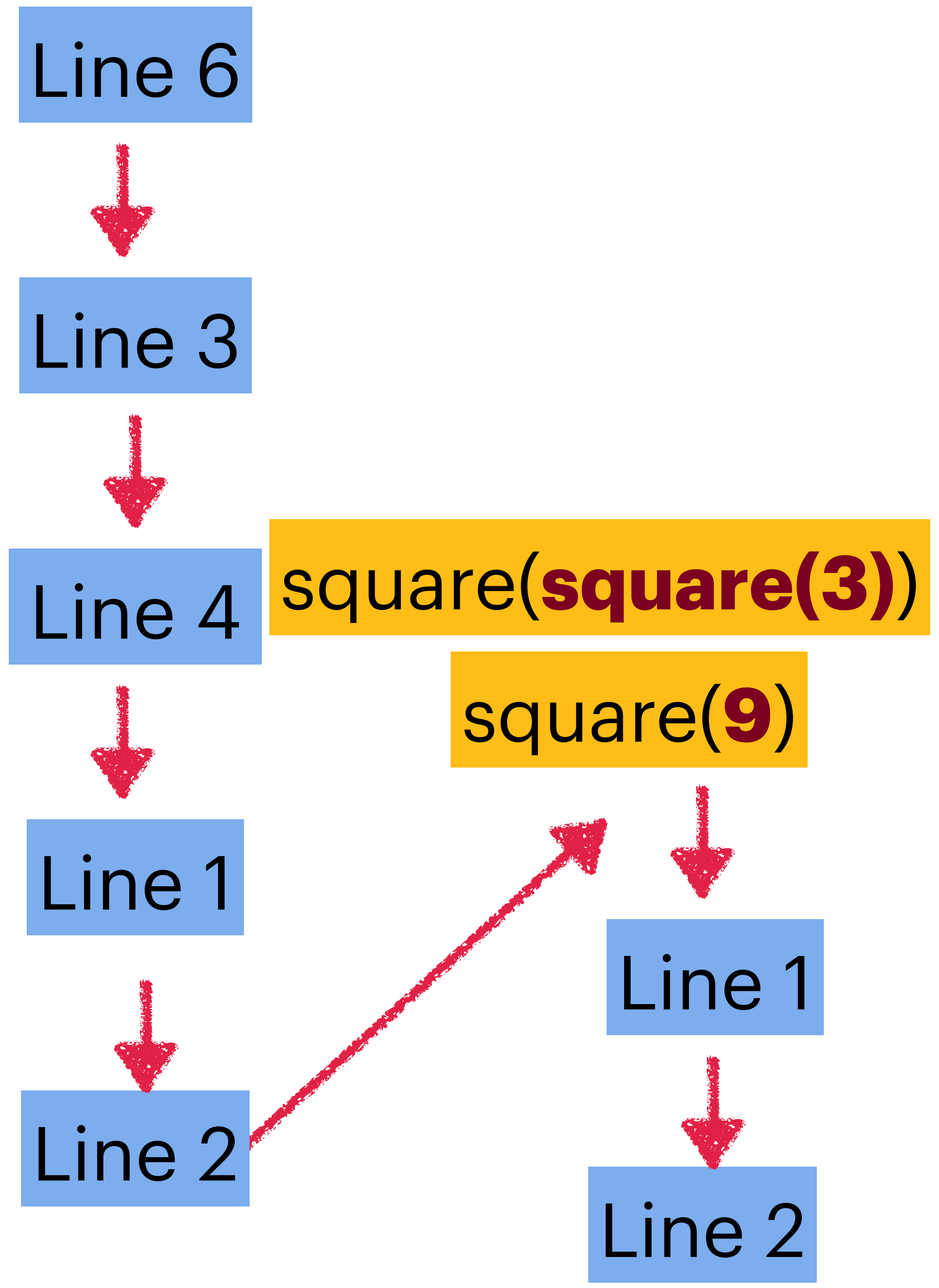
Line 2

Higher Order Functions

A higher-order function is a function that takes another function as input, or **returns a function as output**, or both

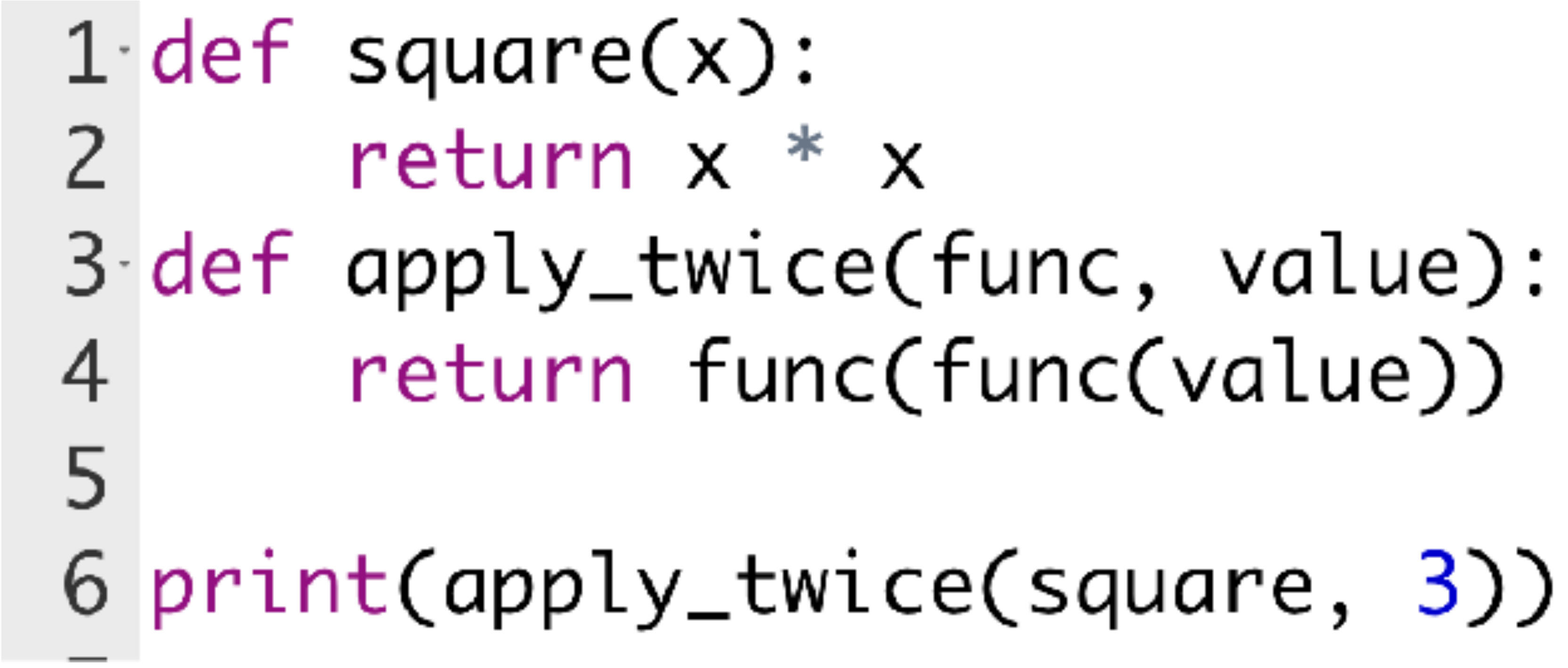


```
1 def square(x):
2     return x * x
3 def apply_twice(func, value):
4     return func(func(value))
5
6 print(apply_twice(square, 3))
```

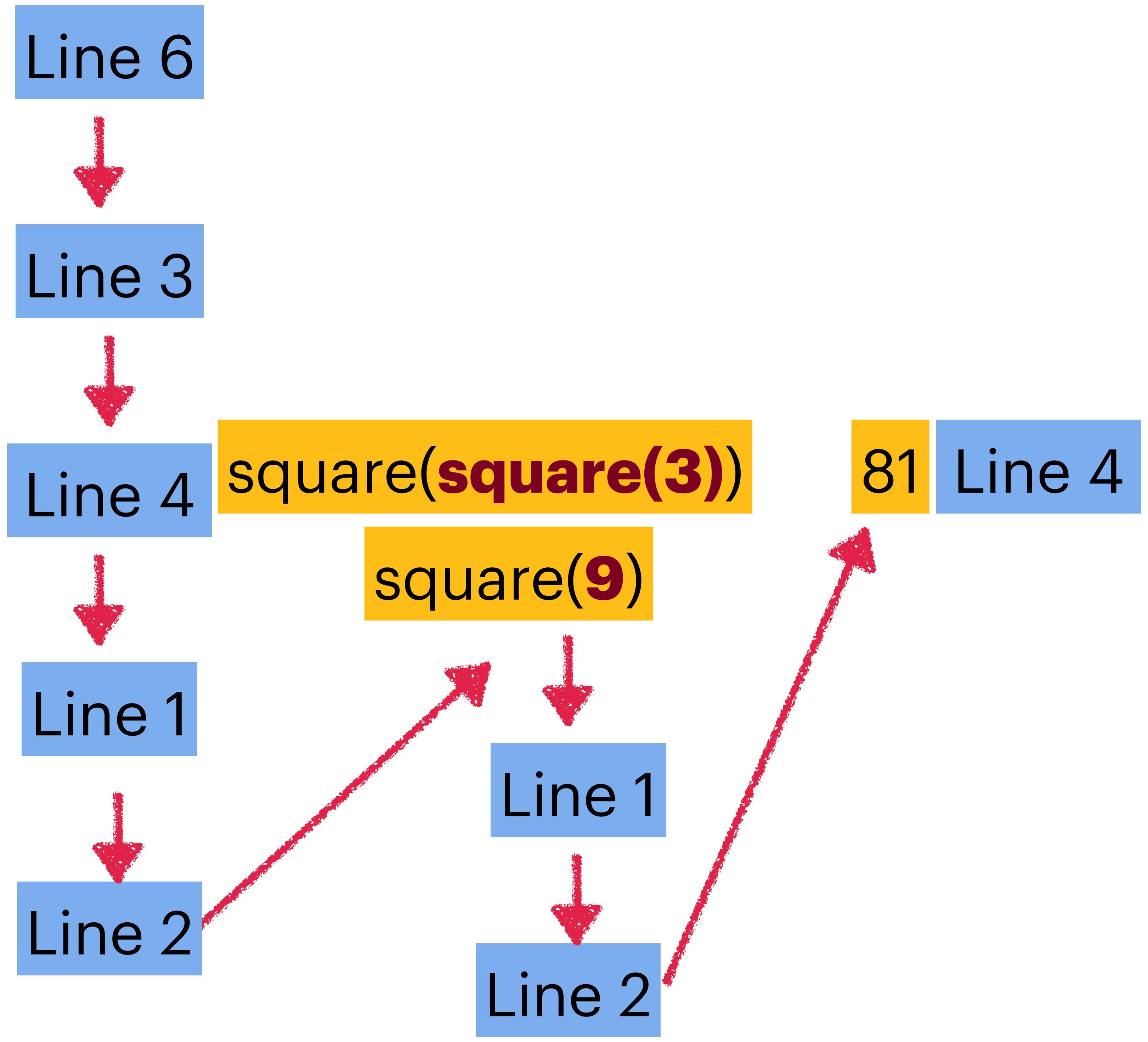


Higher Order Functions

A higher-order function is a function that takes another function as input, or **returns a function as output**, or both



```
1 def square(x):
2     return x * x
3 def apply_twice(func, value):
4     return func(func(value))
5 
6 print(apply_twice(square, 3))
```



Line 6

Line 3

Line 4 square(**square(3)**)

square(**9**)

Line 1

Line 2

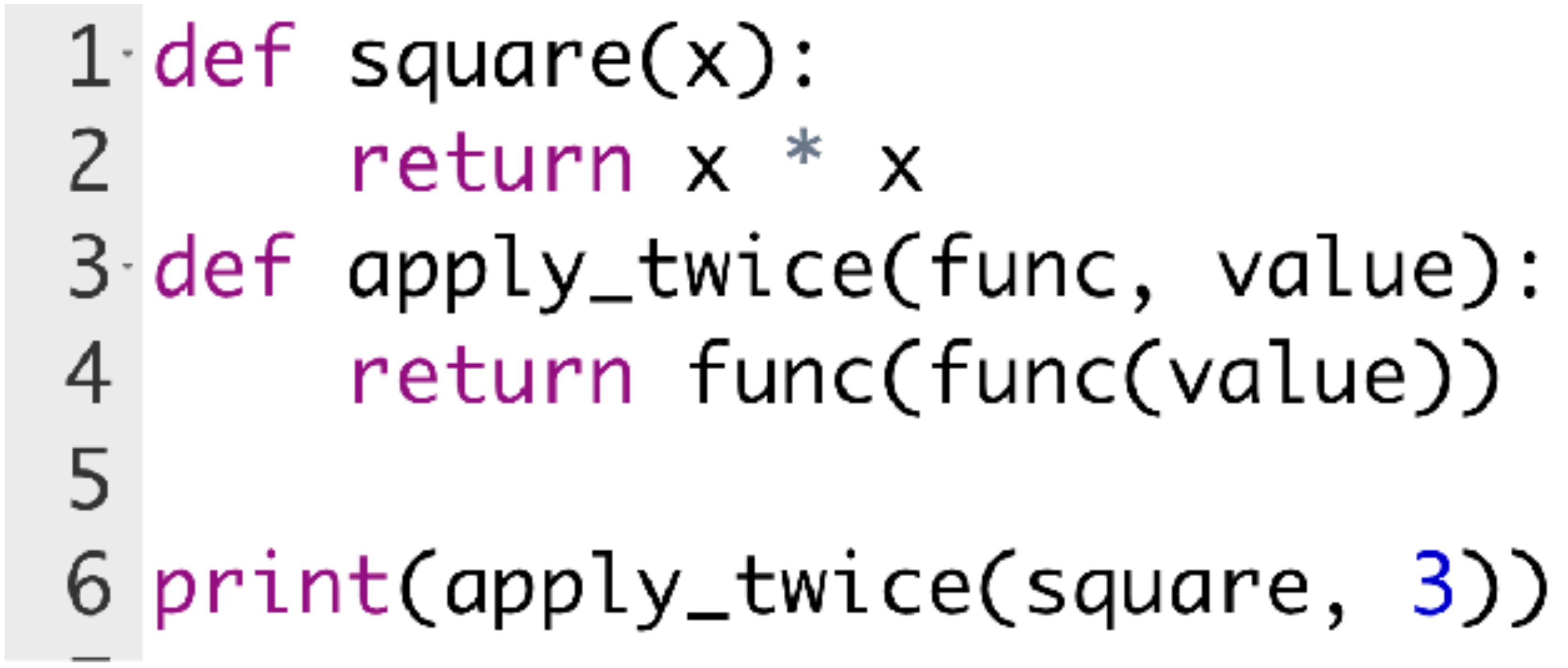
Line 1

Line 2

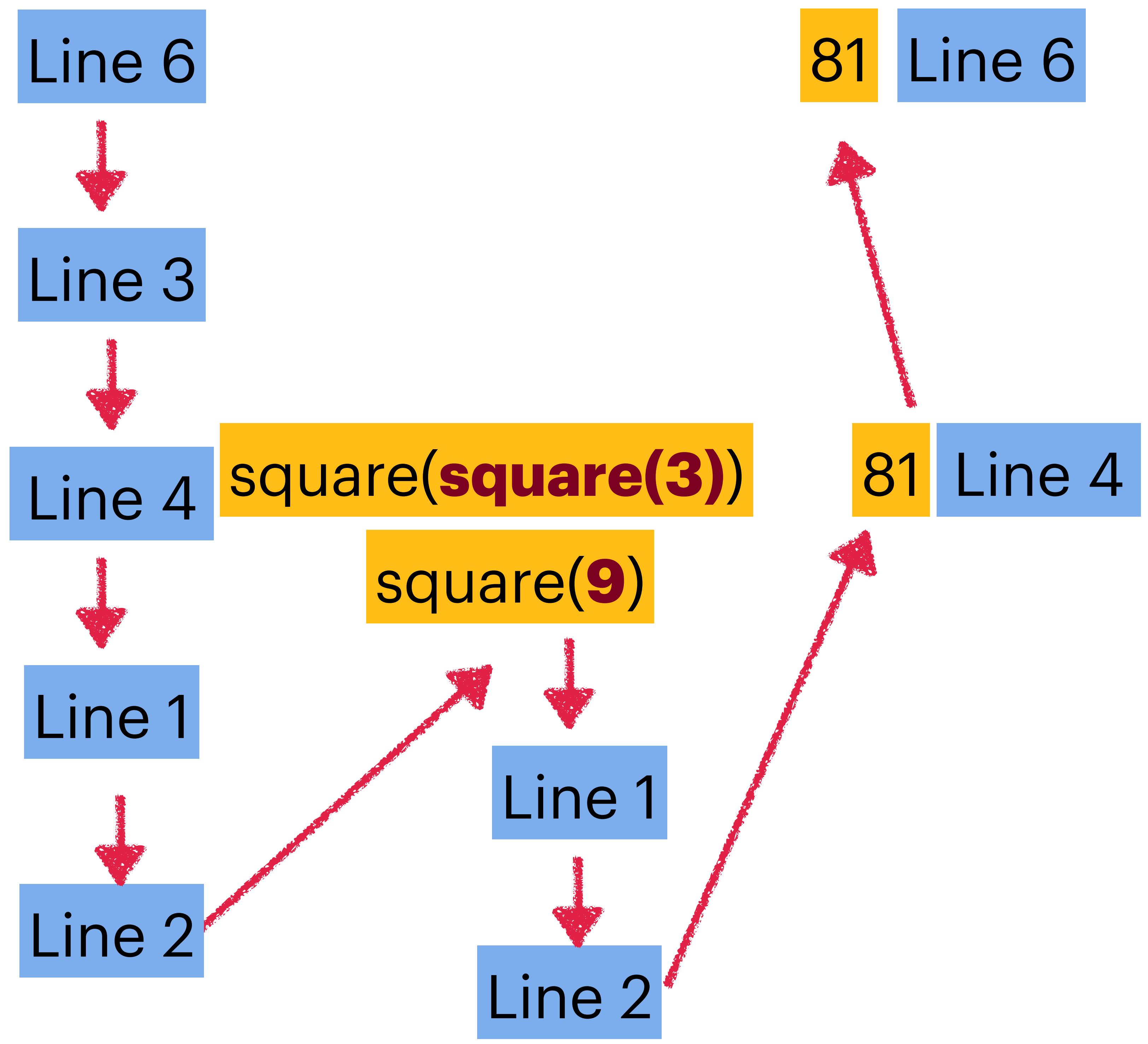
81 Line 4

Higher Order Functions

A higher-order function is a function that takes another function as input, or **returns a function as output**, or both



```
1 def square(x):
2     return x * x
3 def apply_twice(func, value):
4     return func(func(value))
5 
6 print(apply_twice(square, 3))
```



Higher Order Functions

We are using higher order functions since day 1 — print(sum(x)), print(len(x)) —

Higher Order Functions

We are using higher order functions since day 1 — print(sum(x)), print(len(x)) —

Built-in higher order functions!

map(function , iterable)

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
s = map(square, nums)
print(list(s))  # [1, 4, 9, 16, 25]
```

Higher Order Functions

We are using higher order functions since day 1 — print(sum(x)), print(len(x)) —

Built-in higher order functions!

map(function , iterable)

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
s = map(square, nums)
print(list(s))  # [1, 4, 9, 16, 25]
```

For each item of nuts, applies function square. creates an iterator (just like range), which can be converted to list.

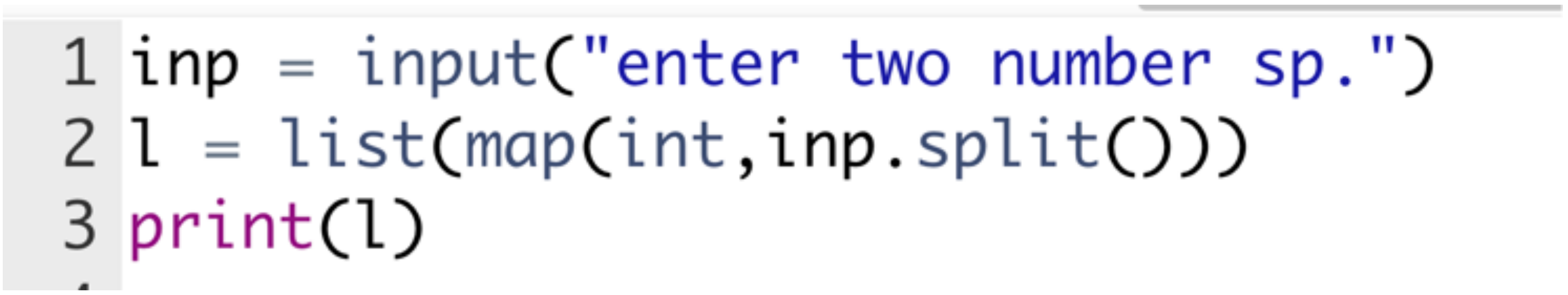
Higher Order Functions

We are using higher order functions since day 1 — print(sum(x)), print(len(x)) —

Built-in higher order functions!

map(function , iterable)

You have been using map in the labs!



```
inp = input("enter two number sp.")
l = list(map(int,inp.split()))
print(l)
```

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
s = map(square, nums)
print(list(s))  # [1, 4, 9, 16, 25]
```

For each item of nuts, applies function square. creates an iterator (just like range), which can be converted to list.

Higher Order Functions

filter(function, iterable)

```
def is_even(x):
    return x % 2 == 0

nums = [1, 2, 3, 4, 5, 6]
evens = filter(is_even, nums)
print(list(evens))  # [2, 4, 6]
```

Keeps only the elements where the function (is_even) returns True.

Higher Order Functions

reduce(function, iterable)

```
import functools

def add(a, b):
    return a + b

nums = [1, 2, 3, 4]
print(reduce(add, nums))  # (((1+2)+3)+4) = 10
```

Higher Order Functions

reduce(function, iterable)

```
import functools

def add(a, b):
    return a + b

nums = [1, 2, 3, 4]
print(reduce(add, nums))   # (((1+2)+3)+4) = 10
```

It reduces a sequence to a single value by repeatedly applying the function.

Higher Order Functions

reduce(function, iterable)

```
import functools

def add(a, b):
    return a + b

nums = [1, 2, 3, 4]
print(reduce(add, nums))   # (((1+2)+3)+4) = 10
```

Recall to use pi, we had to import math. Similarly, to use reduce, import functools.

It reduces a sequence to a single value by repeatedly applying the function.

Lambda Functions

```
f = lambda x: x + 1
print(f(5))    # 6
```

=

```
def f(x):
    return x + 1
print(f(5)) #6
```

Lambda Functions

```
f = lambda x: x + 1
print(f(5))    # 6
```

=

```
def f(x):
    return x + 1
print(f(5)) #6
```

Lambda is useful when the function is **short-lived** or **used only once**

Lambda Functions

```
f = lambda x: x + 1
print(f(5))    # 6
```

=

```
def f(x):
    return x + 1
print(f(5))  #6
```

Lambda is useful when the function is **short-lived** or **used only once**

```
lambda arguments: expression
```

Lambda Functions

```
f = lambda x: x + 1
print(f(5))    # 6
```

=

```
def f(x):
    return x + 1
print(f(5)) #6
```

Lambda is useful when the function is **short-lived** or **used only once**

```
lambda arguments: expression
```

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
print(list(map(square, nums)))
# [1, 4, 9, 16, 25]
```

Lambda Functions

```
f = lambda x: x + 1
print(f(5))    # 6
```

=

```
def f(x):
    return x + 1
print(f(5)) #6
```

Lambda is useful when the function is **short-lived** or **used only once**

```
lambda arguments: expression
```

```
print(list(map(lambda x: x * x, [1, 2, 3, 4])))
```

=

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
print(list(map(square, nums)))
# [1, 4, 9, 16, 25]
```