Lambda Functions

CLARUSWAY©
WAY TO REINVENT YOURSELF

Table of Contents

- Defining a Lambda Function
- Uses of the Lambda Functions
- Lambda within Built-in (map()) Functions-1
- Lambda within Built-in (filter()) Functions-2
- Lambda within User-Defined Functions

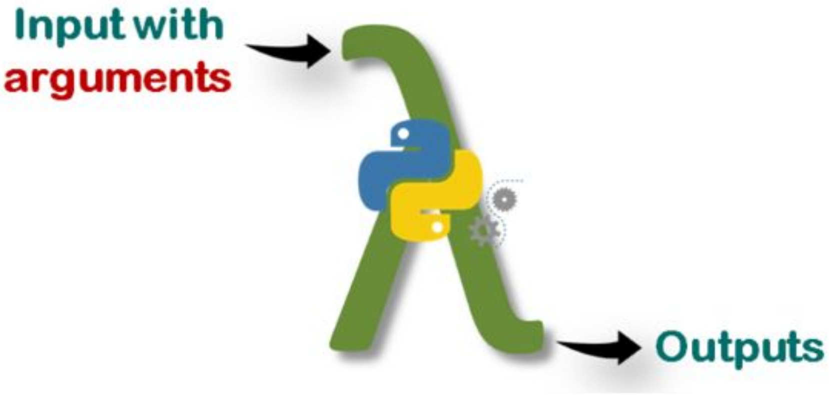
1

Defining a Lambda Function

Defining a `lambda` Function(review)

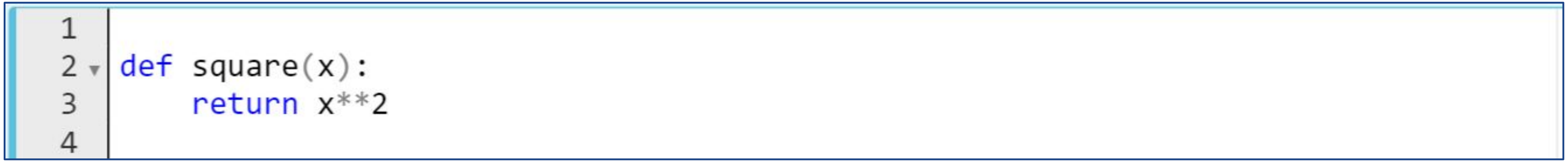
- Another way to define functions in Python is lambda functions. Lambda functions are also called **anonymous** functions since they have no name. We use keyword `lambda` to define a function.

```
The formula syntax is : lambda parameters : expression
```



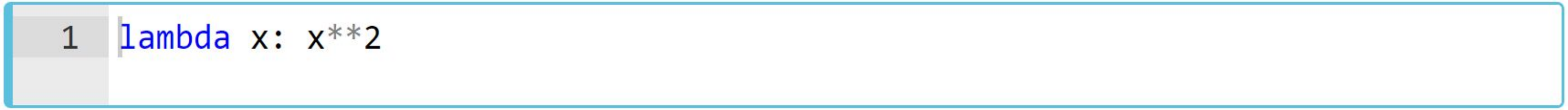
Why we need `lambda` functions?

- If you need to use a one-time function, defining a lambda function is the best option. In some cases, you may need to define a function only once without having to use it later. For instance; let's square given numbers with a function. First, we're going to use `def` :



```python

def square(x):
    return x**2

```

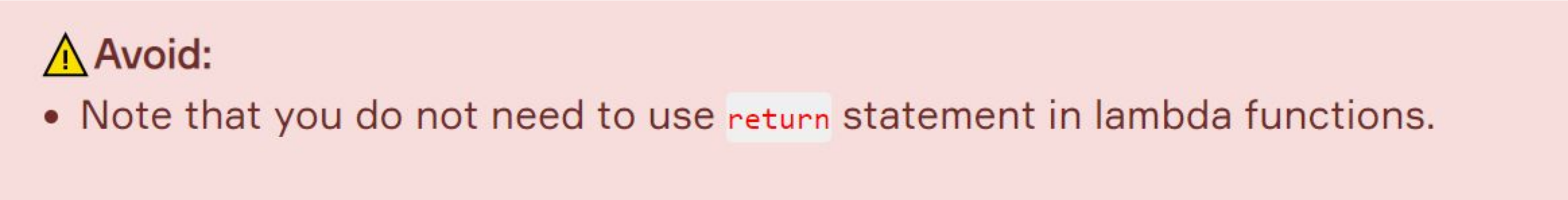
- And now we'll define `lambda` function to do the same.



```python
lambda x: x**2
```

Why we need `lambda` functions?

- As you see, `lambda` is very simple and has a single line with a single expression. On the other hand, these two functions do exactly the same thing.
- A lambda function can take **multiple arguments** *separated by **commas***, but it must be defined with a single expression. This expression is evaluated and the result is returned.

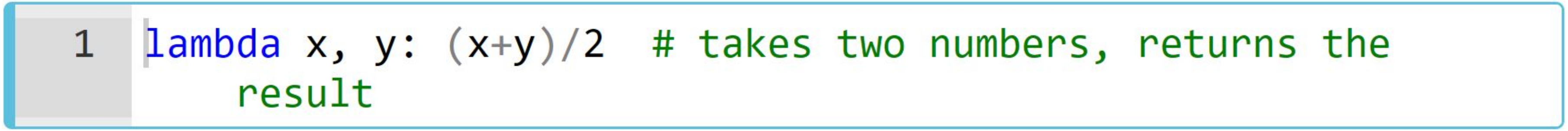


⚠ **Avoid:**

- Note that you do not need to use `return` statement in lambda functions.

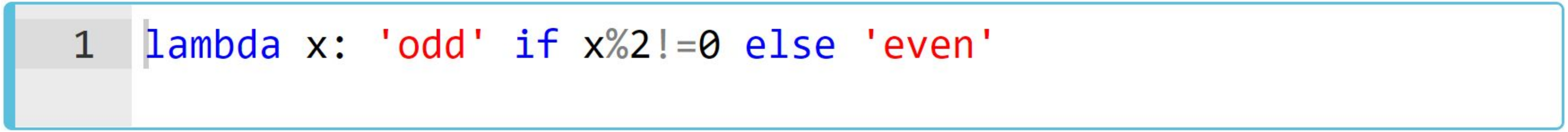
Defining a `lambda` Function(review)

- Consider the following example of multiple arguments. Let's calculate the arithmetic mean of two numbers :



```
lambda x, y: (x+y)/2  # takes two numbers, returns the result
```

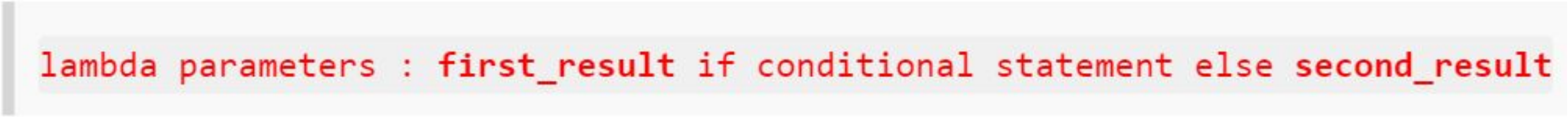
- What if we need to use conditional statements within the lambda definition? Here how we do it :



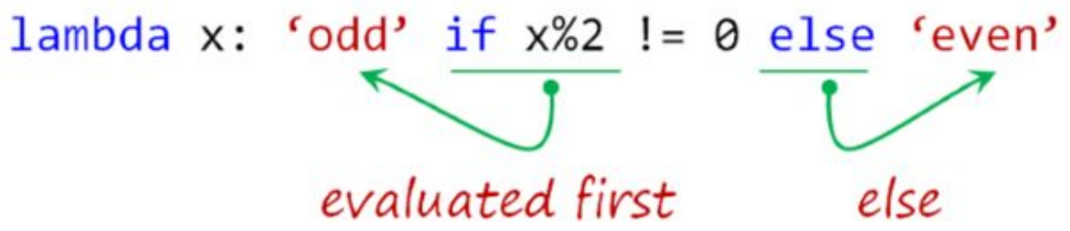
```
lambda x: 'odd' if x%2!=0 else 'even'
```

Defining a `lambda` Function(review)

- The formula syntax of conditional lambda statement is :



```
lambda parameters : first_result if conditional statement else second_result
```



```
lambda x: ‘odd’ if x%2 != 0 else ‘even’
```

⚠ **Avoid:**

- Note that you can't use the usual conditional statement with lambda definition.

2

Uses of the `lambda` Functions

Uses of the `lambda` Functions(review)

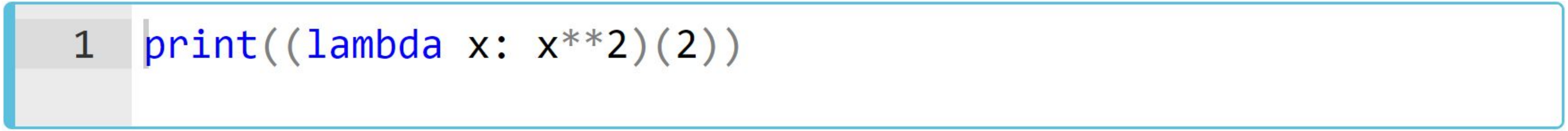
- So far you have seen the definition of lambda function and some of its features. Well, unlike `def`, where do we use lambda? If we need, how do we use the `lambda` functions in our code stream? Moreover, they don't even have names, so how can we call them? In this and the next lesson, we're going to try to find out the answer to these questions.
- Lambda's most important advantages and uses are:
  - You can use it with its own syntax using parentheses,
  - You can also assign it to a variable,
  - It can be useful inside user-defined functions (def),
  - You can use it in several built-in functions,

Uses of the `lambda` Functions(review)

- **By enclosing the function in parentheses :**

First use

```
The formula syntax is :
(lambda arguments : expression)(arguments)
```



```
print((lambda x: x**2)(2))
```

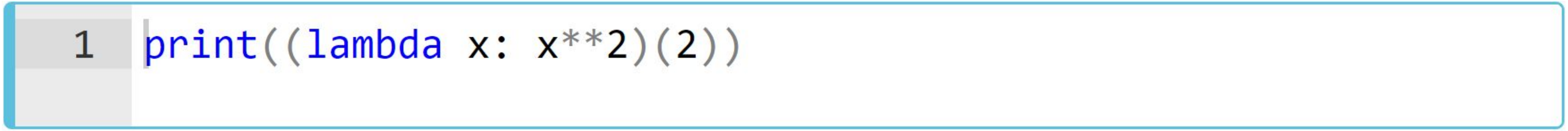
What is the output? Try to figure out in your mind...

Students, write your response!

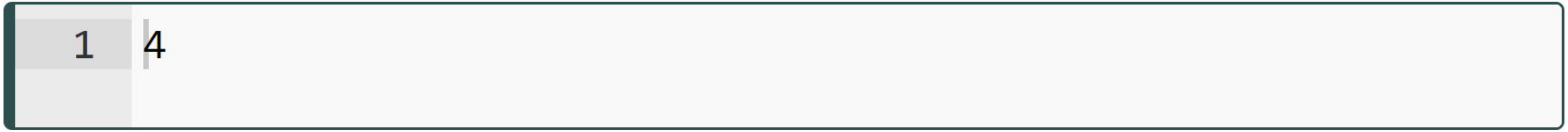
Uses of the `lambda` Functions(review)

- By enclosing the function in parentheses :

```
The formula syntax is :
(lambda parameters : expression)(arguments)
```



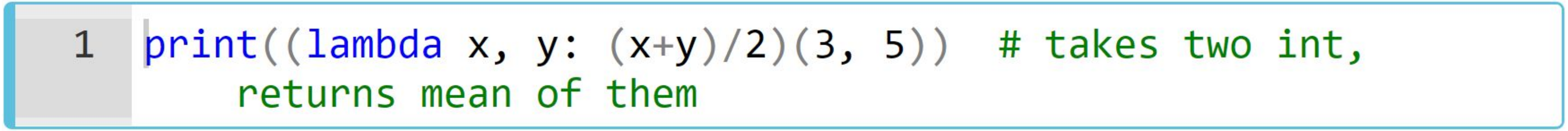
```python
print((lambda x: x**2)(2))
```



```
4
```

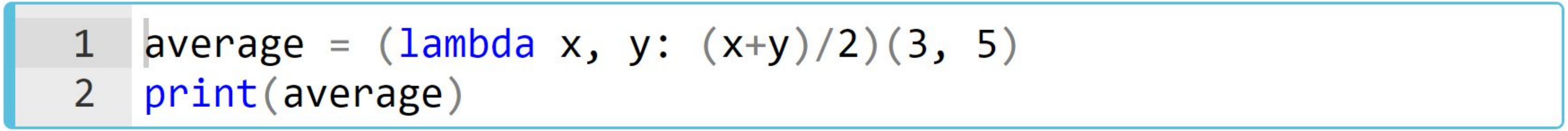
Uses of the `lambda` Functions(review)

- Or you can use multiple arguments using the same syntax :



```python
print((lambda x, y: (x+y)/2)(3, 5))  # takes two int, returns mean of them
```

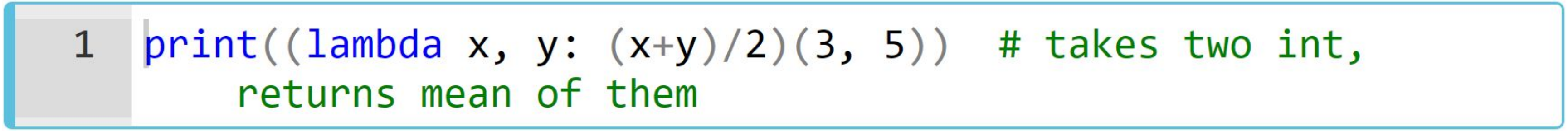
- You can also assign the lambda statement in parentheses to a variable :



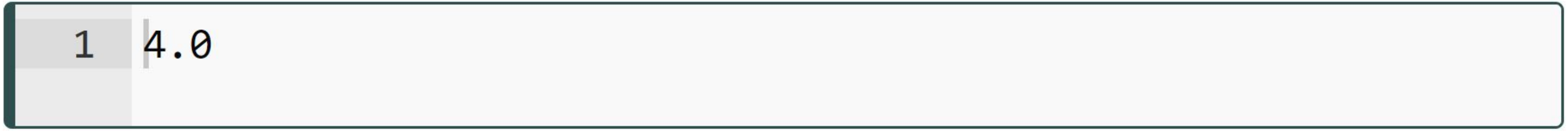
```python
average = (lambda x, y: (x+y)/2)(3, 5)
print(average)
```

Uses of the `lambda` Functions(review)

- Or you can use multiple arguments using the same syntax :

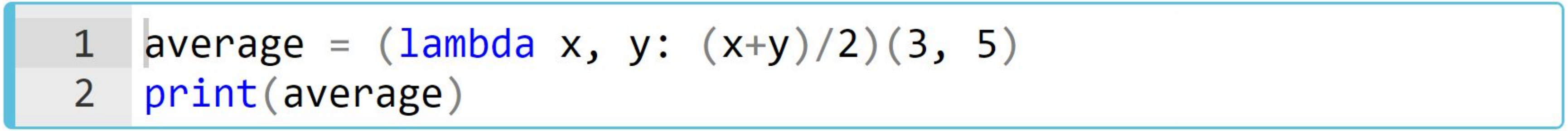


```
print((lambda x, y: (x+y)/2)(3, 5))  # takes two int, returns mean of them
```



```
4.0
```

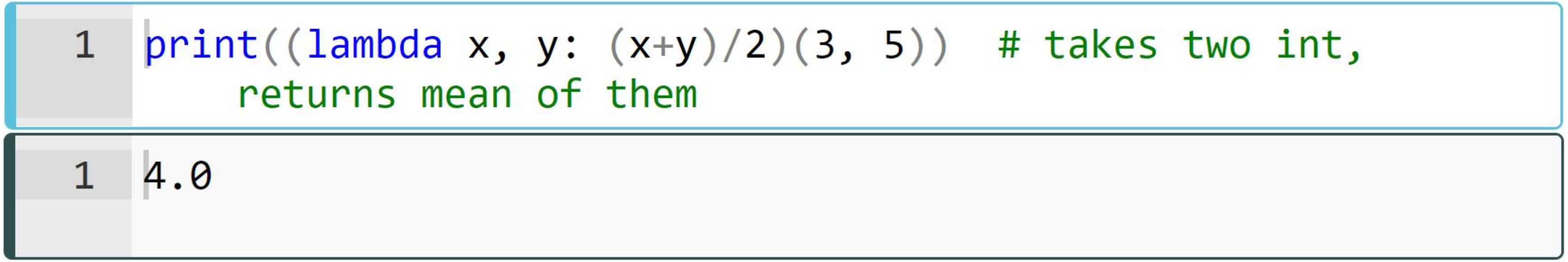
- You can also assign the lambda statement in parentheses to a variable :



```
average = (lambda x, y: (x+y)/2)(3, 5)
print(average)
```

Uses of the `lambda` Functions (review)

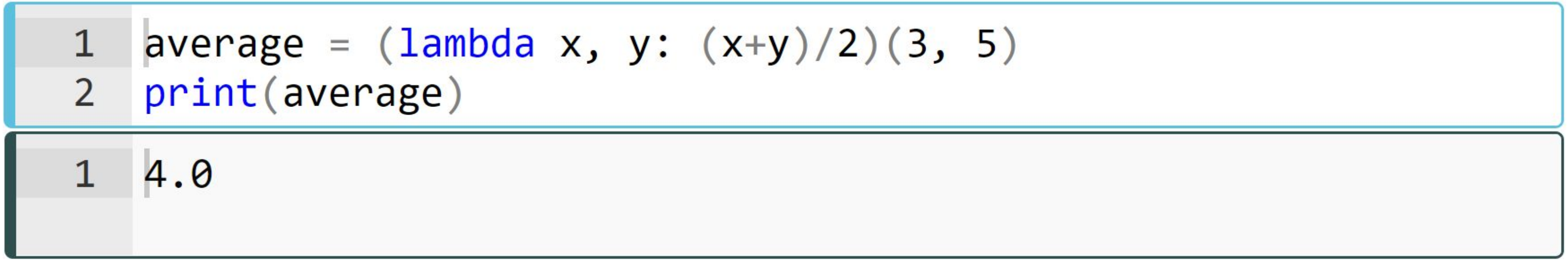
- Or you can use multiple arguments using the same syntax :



```
print((lambda x, y: (x+y)/2)(3, 5))  # takes two int, returns mean of them
```

```
4.0
```

- You can also assign the lambda statement in parentheses to a variable :



```
average = (lambda x, y: (x+y)/2)(3, 5)
print(average)
```

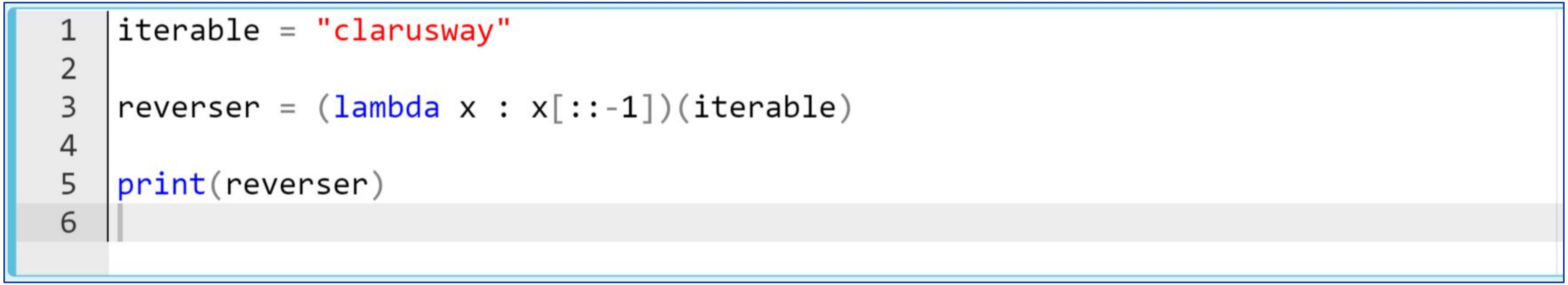
```
4.0
```

Uses of the `lambda` Functions

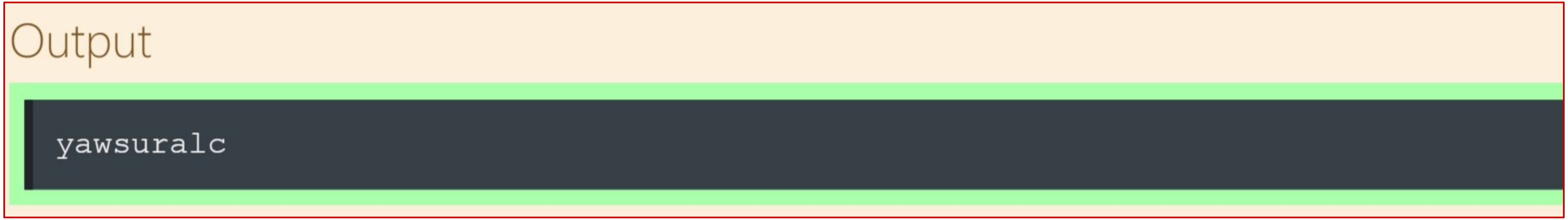
- **Task** :
  - Define a `lambda` function to reverse the elements of any iterables.
  - Use **parentheses** for arguments and print the result.

Uses of the `lambda` Functions

- **The code can be as :**



```python
iterable = "clarusway"

reverser = (lambda x : x[::-1])(iterable)

print(reverser)

```

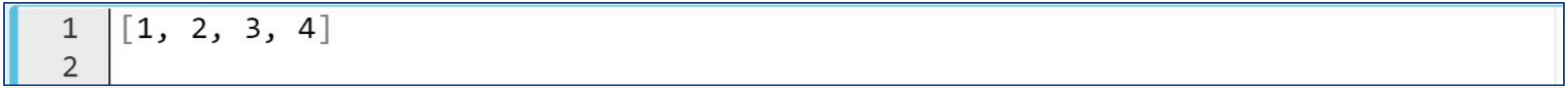


Output

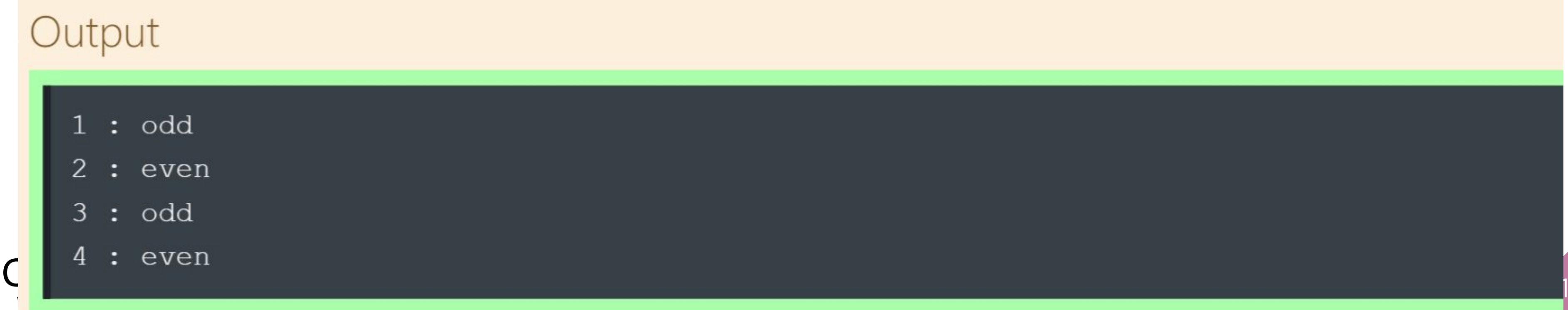
```
yawsuralc
```

Uses of the `lambda` Functions

- **Task** :
  - Write a Python program that types ‘**even**’ or ‘**odd**’ in accordance with the numbers in a `list`.
  - Use `lambda` function and loop.
  - Your code must contain no more than 2 lines.
  - The sample `list` and desired output are as follows :



```
[1, 2, 3, 4]

```

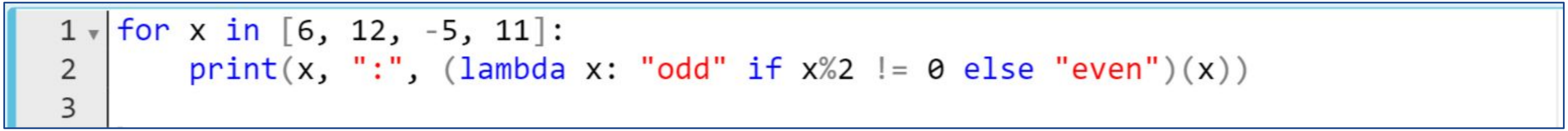


Output

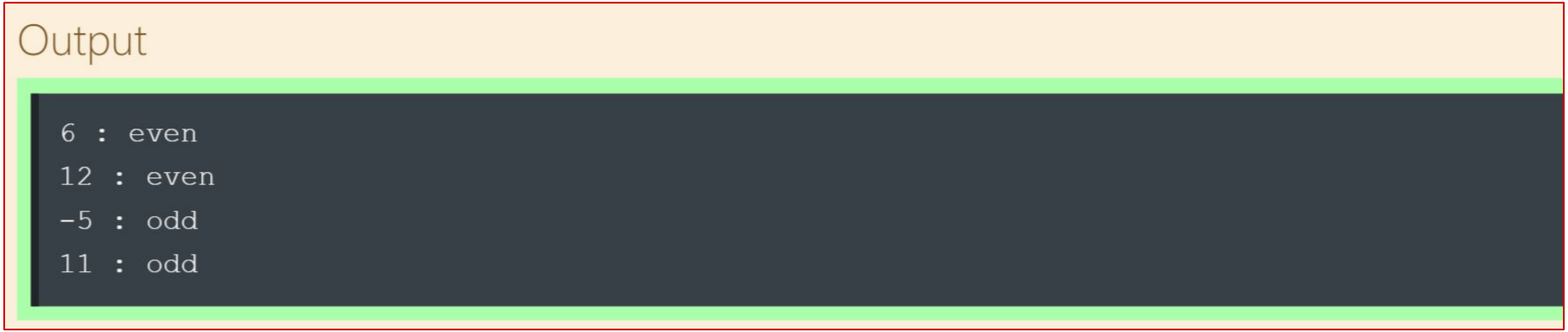
```
1 : odd
2 : even
3 : odd
4 : even
```

Uses of the `lambda` Functions

- **The code can be as :**



```python
for x in [6, 12, -5, 11]:
    print(x, ":", (lambda x: "odd" if x%2 != 0 else "even")(x))

```



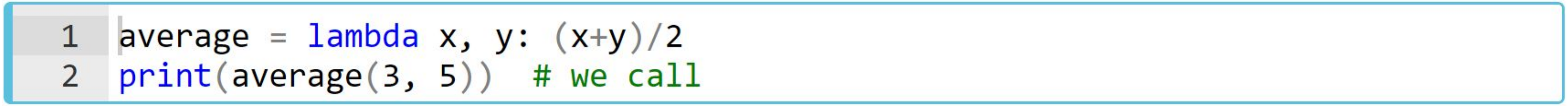
Output

```
6 : even
12 : even
-5 : odd
11 : odd
```

Uses of the `lambda` Functions (review)

By assigning a function object to a variable : **Second use**

- Alternatively, you can assign the lambda function definition to a variable then you can call it :



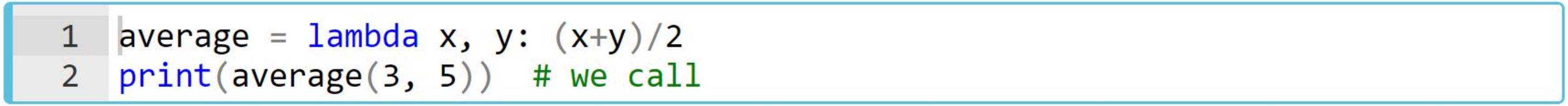
```python
average = lambda x, y: (x+y)/2
print(average(3, 5))  # we call
```

What is the output? Try to figure out in your mind...

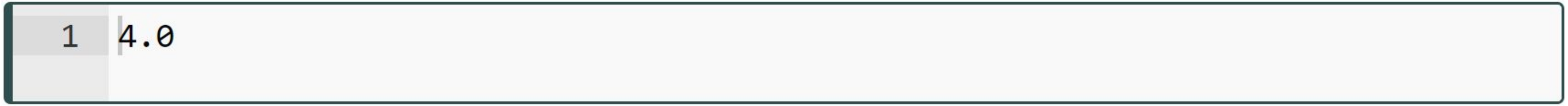
Students, write your response!

Uses of the `lambda` Functions (review)

- Alternatively, you can assign the lambda function definition to a variable then you can call it :



```
average = lambda x, y: (x+y)/2
print(average(3, 5))  # we call
```



```
4.0
```

Uses of the `lambda` Functions (review)

- **Task** :
  - Define a `lambda` function to reverse the elements of any iterables.
  - Use **variable** for arguments and print the result.

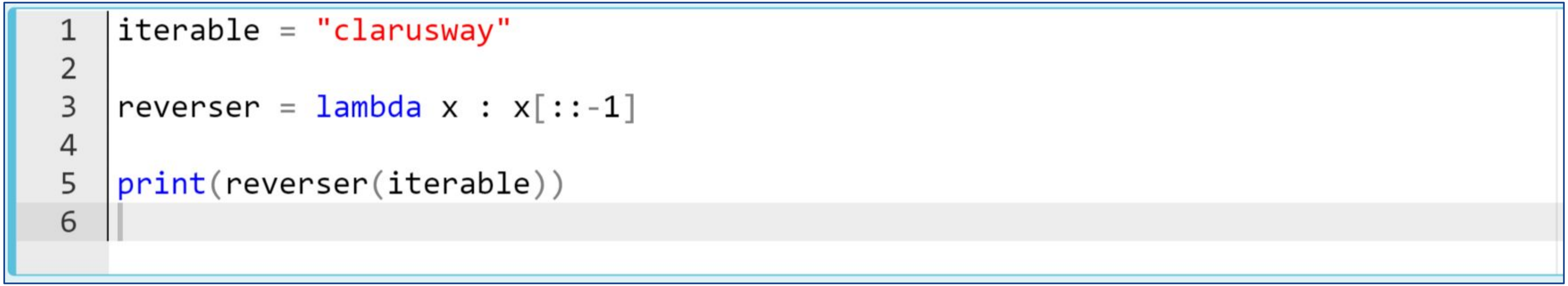
Third use

3

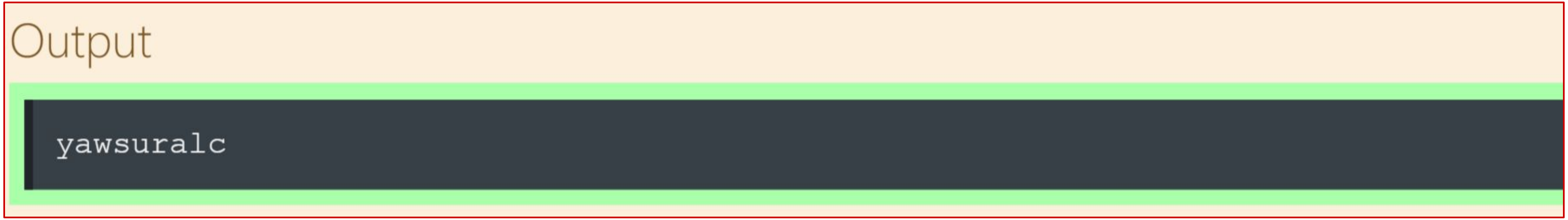
Lambda within Built-in (`map()`) Functions-1

Uses of the `lambda` Functions (review)

- **The code can be as :**



```python
iterable = "clarusway"

reverser = lambda x : x[::-1]

print(reverser(iterable))

```



Output

```
yawsuralc
```

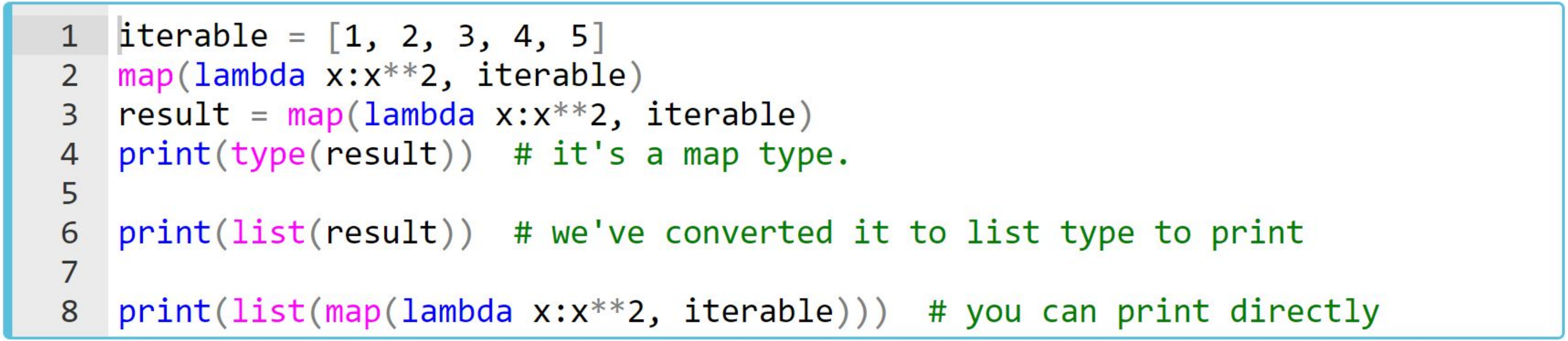
Lambda within Built-in (`map()`) Functions-1

- When using some built-in functions we may need additional functions inside them. This can be done by using def, but when we do the same thing with lambda we save both time and additional lines of code and we make it clear to read.
- **Lambda within `map()` function :**
  - `map()` returns a list of the outputs after applying the given function to *each element* of a given *iterable object* such as `list`, `tuple`, etc.

```
The basic formula syntax is : map(function, iterable)
```

Lambda within Built-in (`map()`) Functions-1

- Let's square all the numbers in the list using `map()` and `lambda`. Consider this *pre-class* example :



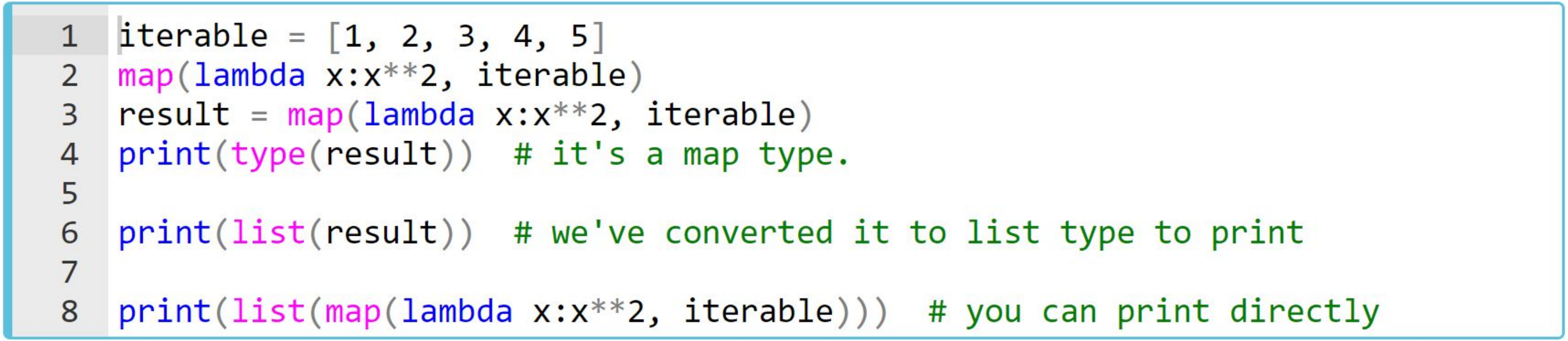
```
iterable = [1, 2, 3, 4, 5]
map(lambda x:x**2, iterable)
result = map(lambda x:x**2, iterable)
print(type(result))  # it's a map type.

print(list(result))  # we've converted it to list type to print

print(list(map(lambda x:x**2, iterable)))  # you can print directly
```

What is the output? Try to figure out in your mind...

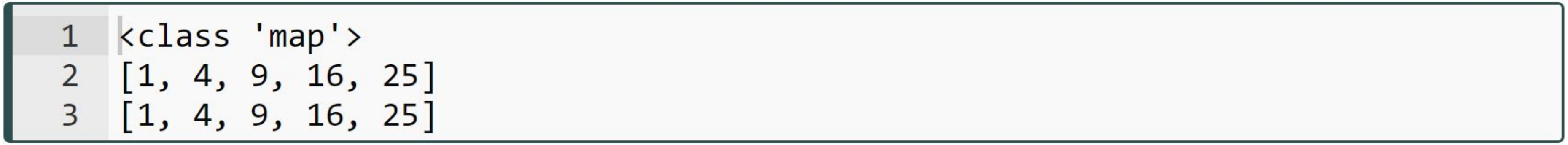
Students, write your response!

Lambda within Built-in (`map()`) Functions-1

- The output of this *pre-class* example :



```python
iterable = [1, 2, 3, 4, 5]
map(lambda x:x**2, iterable)
result = map(lambda x:x**2, iterable)
print(type(result))  # it's a map type.

print(list(result))  # we've converted it to list type to print

print(list(map(lambda x:x**2, iterable)))  # you can print directly
```



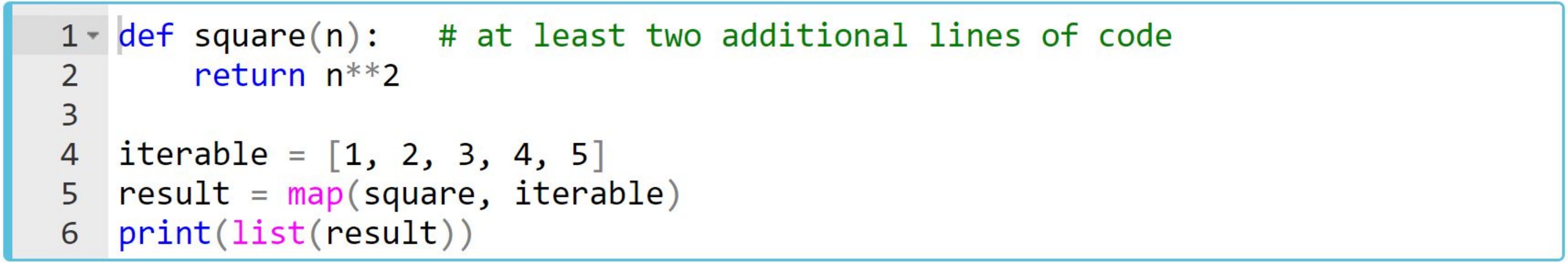
```
<class 'map'>
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
```

Lambda within Built-in (`map()`) Functions-1

- **Task :**
  - Do the same thing using user-defined function (`def`).
  - Use the `def` in `map()` function.

Lambda within Built-in (`map()`) Functions-1

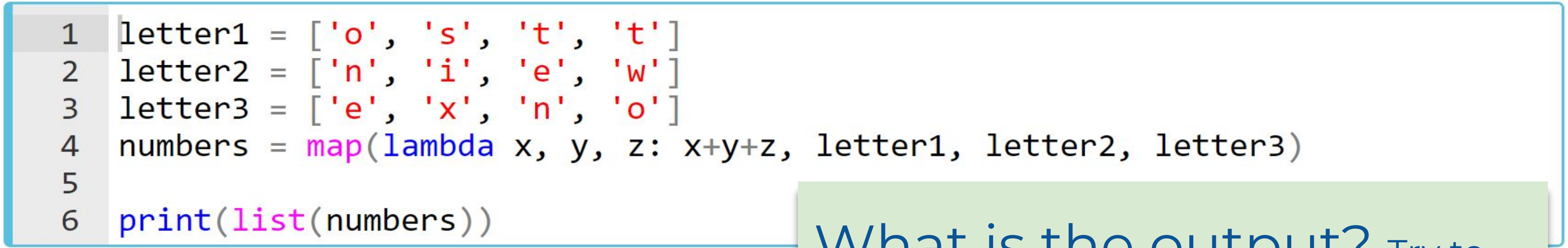
- If you try to do the same thing using `def`, it is likely that the lines of code similar to the following occur. As you can see below, there are at least two additional lines of code. Moreover, we will not use the `square` function again because we only need to use it inside the `map()` function.



```python
def square(n):   # at least two additional lines of code
    return n**2

iterable = [1, 2, 3, 4, 5]
result = map(square, iterable)
print(list(result))
```

Lambda within Built-in (`map()`) Functions-1

- Now, let's try to give an example with multiple arguments in `lambda` function using `map()` :



```
letter1 = ['o', 's', 't', 't']
letter2 = ['n', 'i', 'e', 'w']
letter3 = ['e', 'x', 'n', 'o']
numbers = map(lambda x, y, z: x+y+z, letter1, letter2, letter3)

print(list(numbers))
```

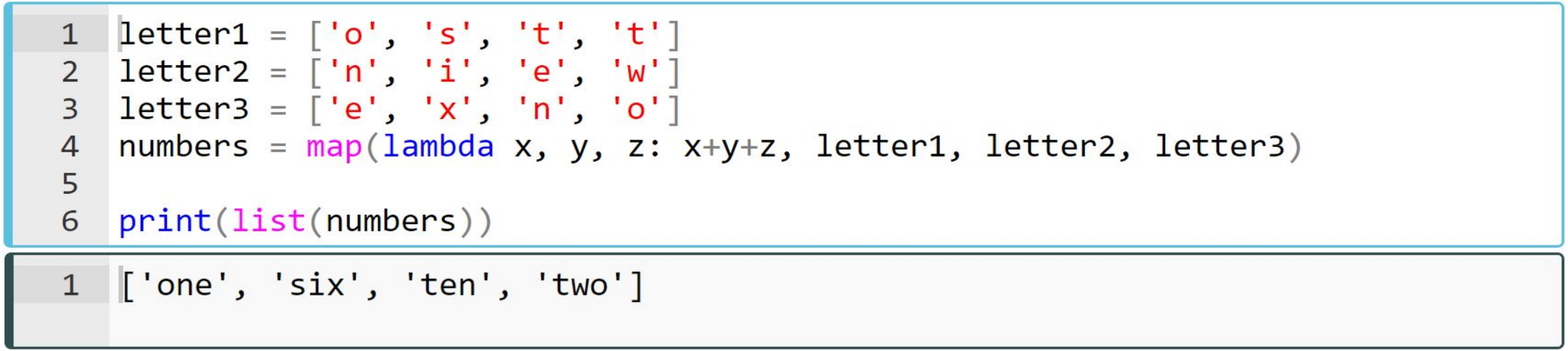
What is the output? Try to figure out in your mind...

- In the above example, we have combined three strings using 👉+ operator in `lambda` definition.

Students, write your response!

Lambda within Built-in (`map()`) Functions-1

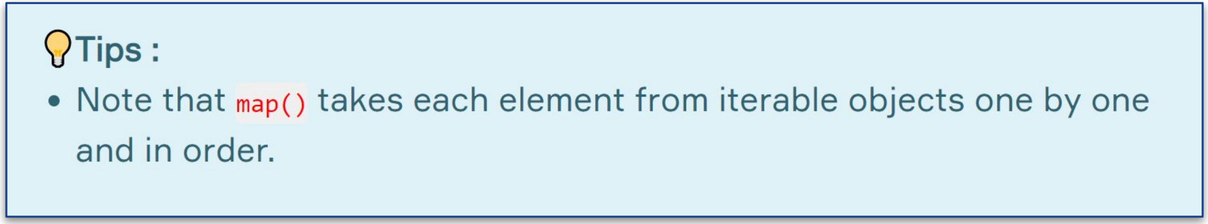
- The output :



```
letter1 = ['o', 's', 't', 't']
letter2 = ['n', 'i', 'e', 'w']
letter3 = ['e', 'x', 'n', 'o']
numbers = map(lambda x, y, z: x+y+z, letter1, letter2, letter3)

print(list(numbers))
```

```
['one', 'six', 'ten', 'two']
```

- In the above example, we have combined three strings using 👉+ operator in lambda definition.

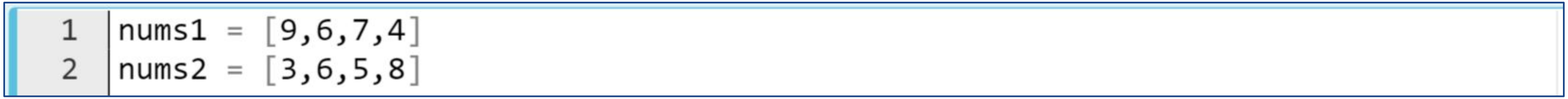


💡Tips :

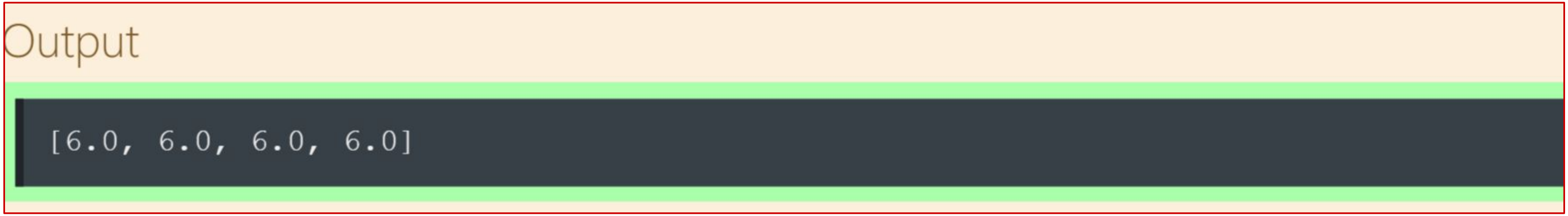
- Note that `map()` takes each element from iterable objects one by one and in order.

Lambda within Built-in (`map()`) Functions-1

- **Task :**
  - Using `lambda` in `map()` function, Write a program that calculates the arithmetic mean of the elements in the following two `list`s in accordance with their order and collects them into a `list`.



```
nums1 = [9,6,7,4]
nums2 = [3,6,5,8]
```

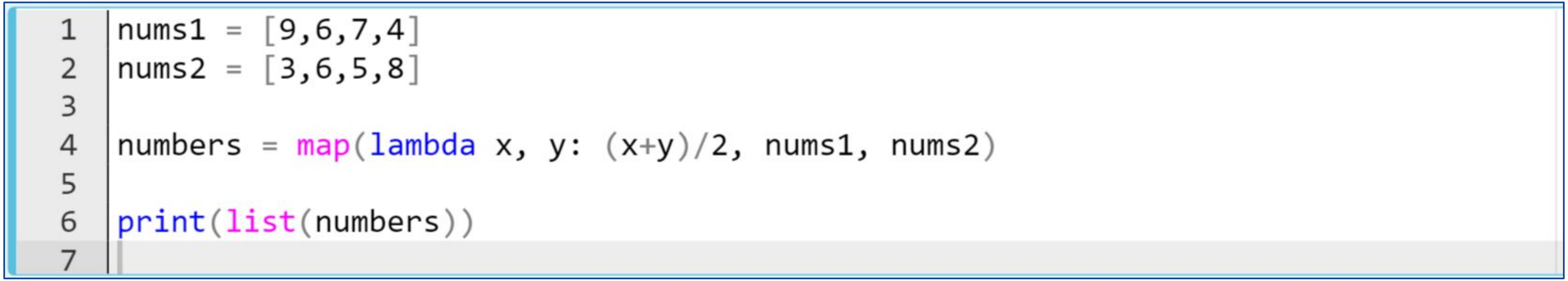


Output

```
[6.0, 6.0, 6.0, 6.0]
```

Lambda within Built-in (`map()`) Functions-1

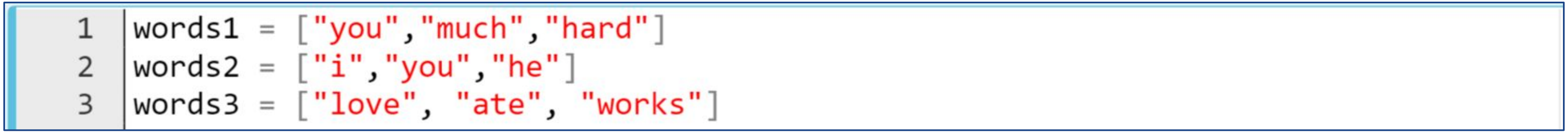
- **The code can be as follows :**



```python
nums1 = [9,6,7,4]
nums2 = [3,6,5,8]

numbers = map(lambda x, y: (x+y)/2, nums1, nums2)

print(list(numbers))

```

Lambda within Built-in (`map()`) Functions-1

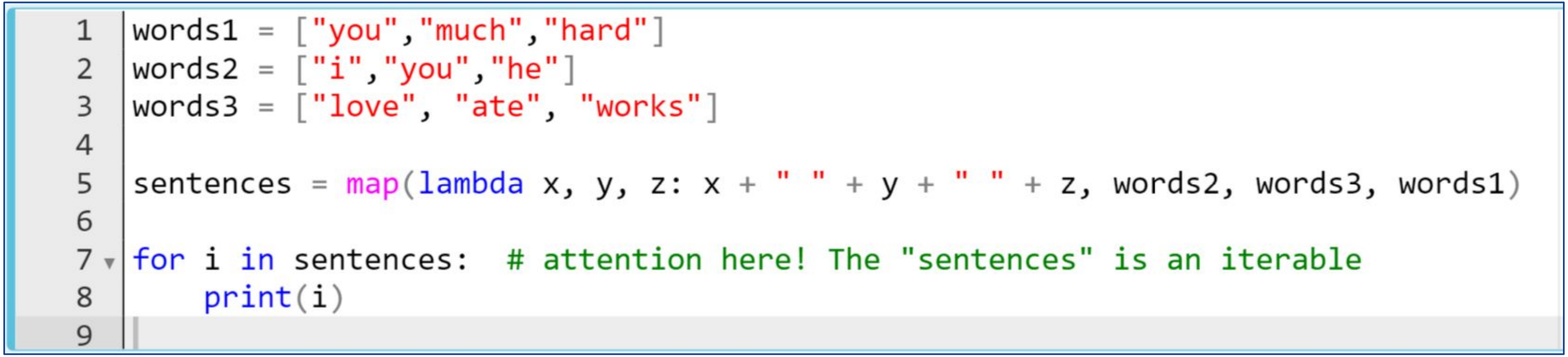
- **Task :**
  - Using `lambda` in `map()` function, write a program that sets three meaningful sentences derived from the elements in the following three `list`s in accordance with their order.
  - Print these sentences on separate lines.



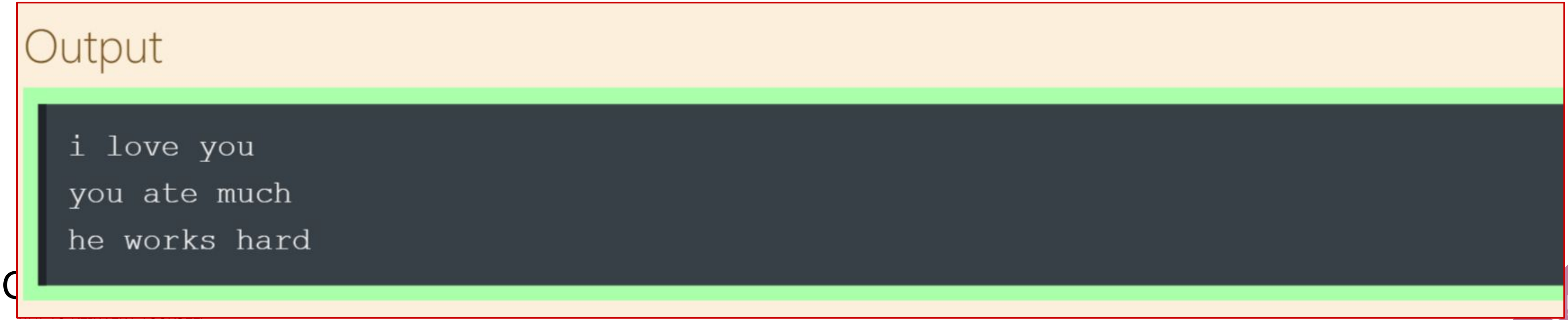
```
words1 = ["you","much","hard"]
words2 = ["i","you","he"]
words3 = ["love", "ate", "works"]
```

Lambda within Built-in (`map()`) Functions-1

- **The code can be as follows :**



```python
words1 = ["you","much","hard"]
words2 = ["i","you","he"]
words3 = ["love", "ate", "works"]

sentences = map(lambda x, y, z: x + " " + y + " " + z, words2, words3, words1)

for i in sentences:  # attention here! The "sentences" is an iterable
    print(i)

```



Output

```
i love you
you ate much
he works hard
```

Third use(...continued)

4

Lambda within Built-in (`filter()`) Functions-2

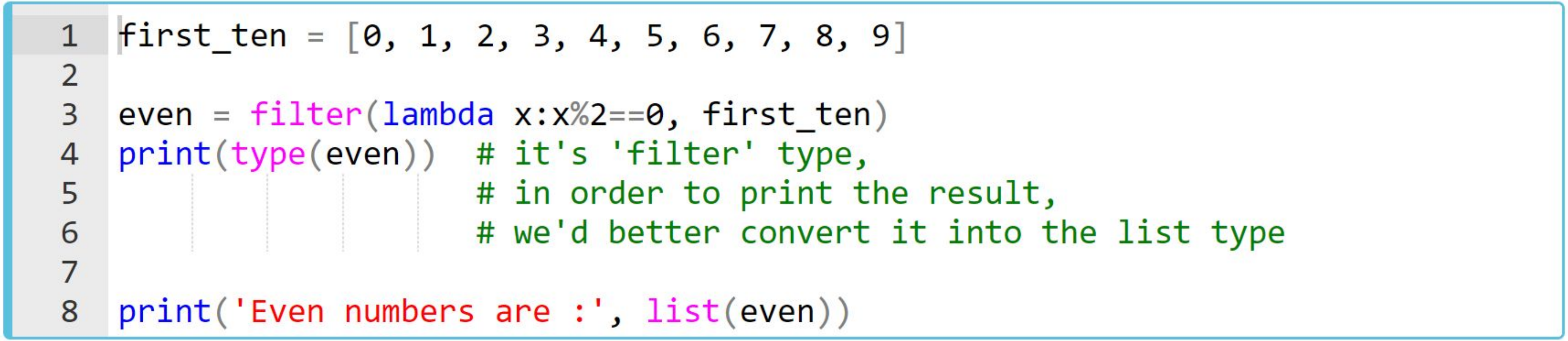
Lambda within Built-in (`filter()`) Functions-2

- **Lambda within filter() function :**
  - `filter()` filters the given sequence (iterable objects) with the help of a function (`lambda`) that tests each element in the sequence to be `True` or not.

```
The basic formula syntax is : filter(function, sequence)
```

Lambda within Built-in (`filter()`) Functions-2

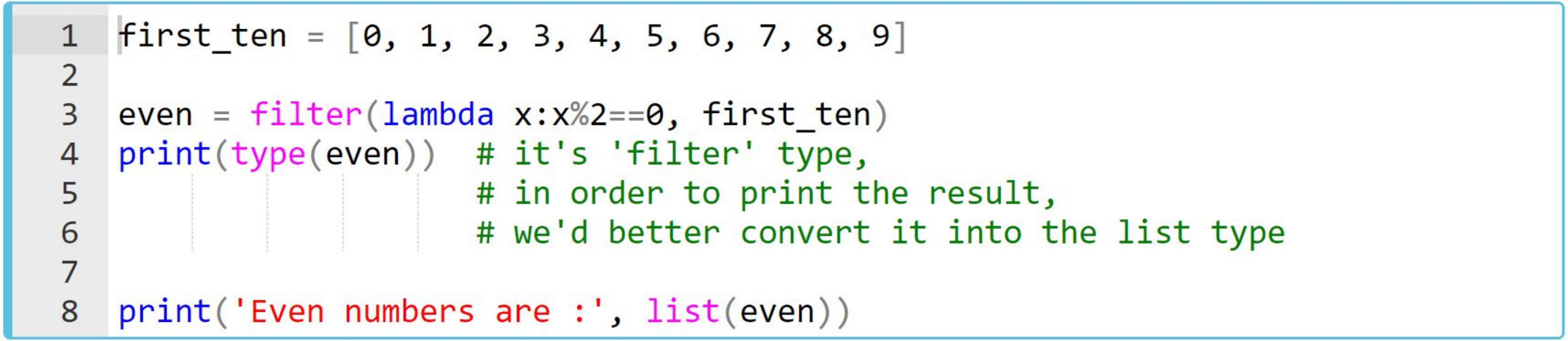
- Let's grasp the subject with a *pre-class* example in which we'll filter the even numbers in a `list`.



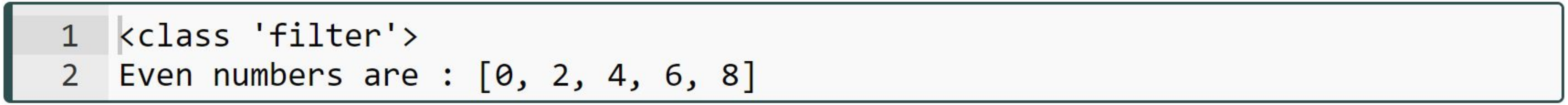
```
first_ten = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

even = filter(lambda x:x%2==0, first_ten)
print(type(even))  # it's 'filter' type,
                   # in order to print the result,
                   # we'd better convert it into the list type

print('Even numbers are :', list(even))
```

Lambda within Built-in (`filter()`) Functions-2

- The output :



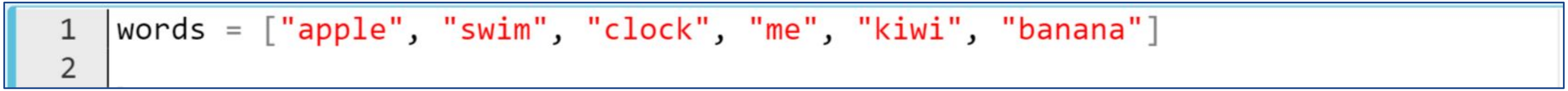
```python
first_ten = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

even = filter(lambda x:x%2==0, first_ten)
print(type(even))  # it's 'filter' type,
                   # in order to print the result,
                   # we'd better convert it into the list type

print('Even numbers are :', list(even))
```



```
<class 'filter'>
Even numbers are : [0, 2, 4, 6, 8]
```

Lambda within Built-in (`filter()`) Functions-2

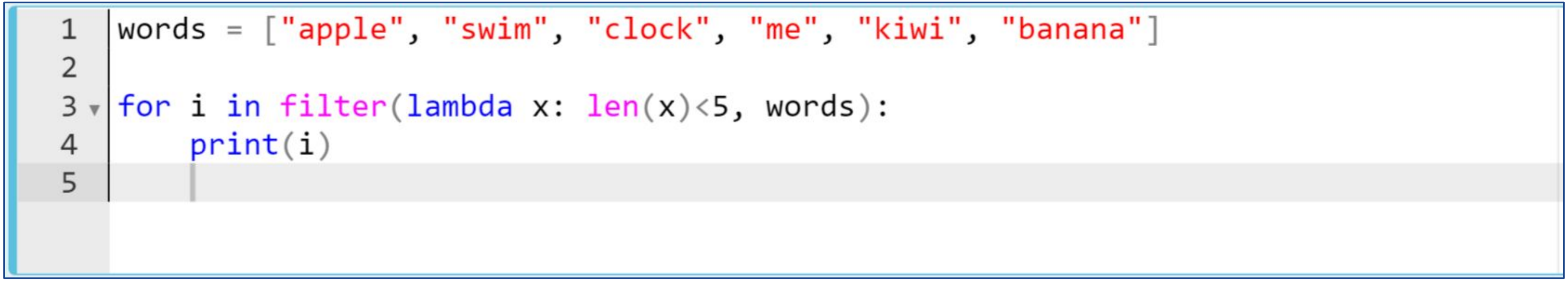
- **Task :**
  - Using `lambda` in `filter()` function, write a program that filters out words (elements of the given `list`) with less than 5 chars.
  - Print these words which has less than 5 chars on separate lines.



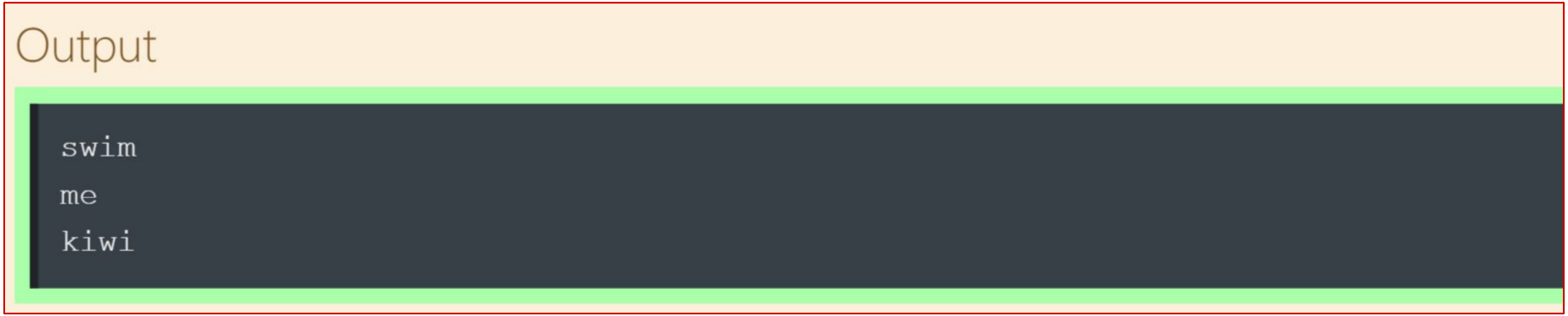
```python
words = ["apple", "swim", "clock", "me", "kiwi", "banana"]

```

Lambda within Built-in (`filter()`) Functions-2

- **The code can be as follows :**



```python
words = ["apple", "swim", "clock", "me", "kiwi", "banana"]

for i in filter(lambda x: len(x)<5, words):
    print(i)

```

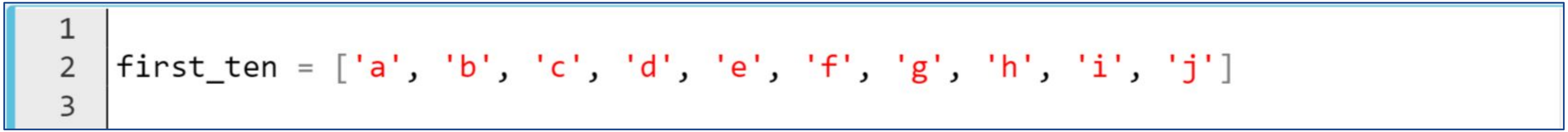


Output

```
swim
me
kiwi
```

Lambda within Built-in (`filter()`) Functions-2

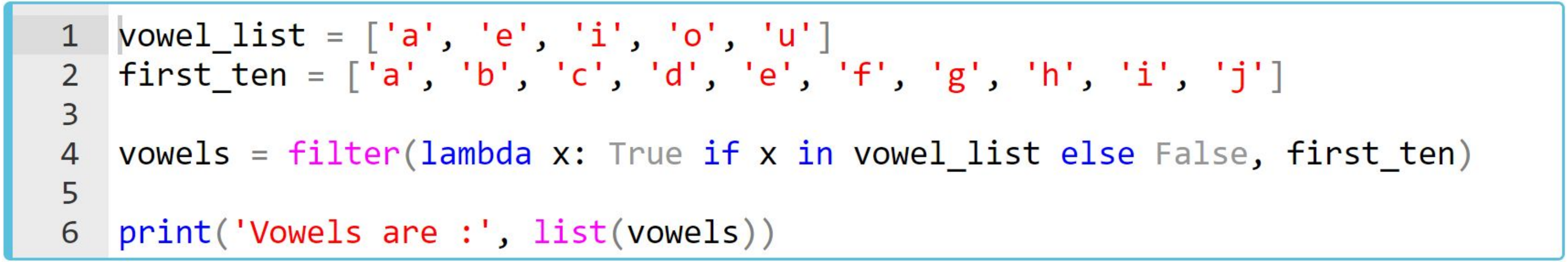
- **Task :**
  - This time, let’s filter the vowels from the given letters in a `list`.
  - Print these letters in a `list`.



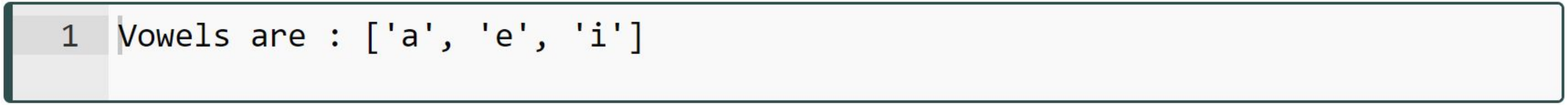
```

first_ten = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

```

Lambda within Built-in (`filter()`) Functions-2

- The code should look like :



```python
vowel_list = ['a', 'e', 'i', 'o', 'u']
first_ten = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

vowels = filter(lambda x: True if x in vowel_list else False, first_ten)

print('Vowels are :', list(vowels))
```



```
Vowels are : ['a', 'e', 'i']
```

- We draw your attention to this issue that *lambda definition* we use in this example gives only `True` or `False` as a result.

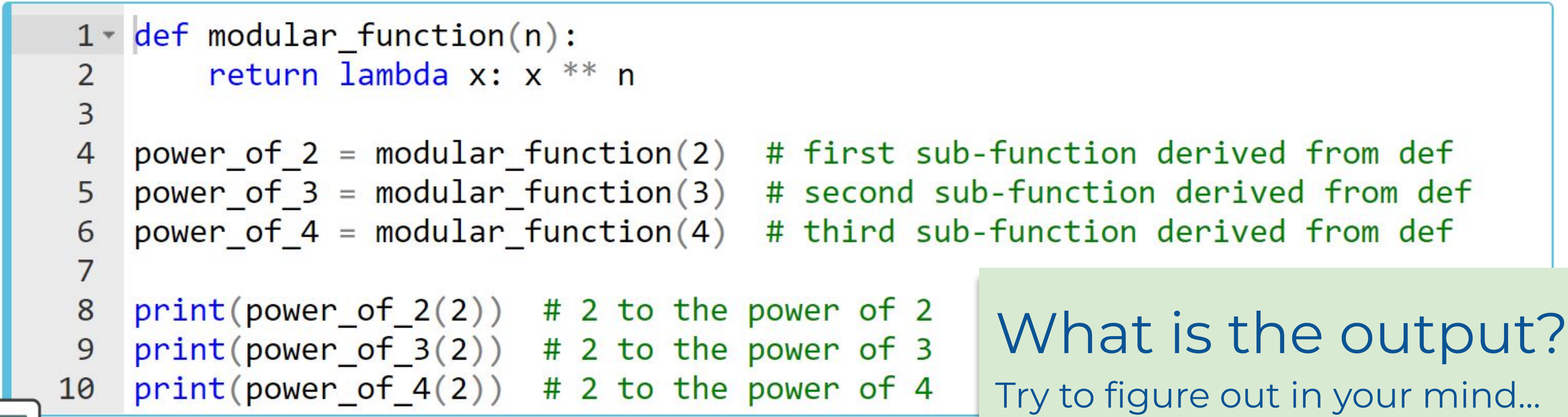
Last use

4

Lambda within User-Defined Functions

Lambda within User-Defined Functions

- **Lambda within def :**
  - Using a lambda statement in a user-defined function provides us useful opportunities. We can define a group of functions that we may use later in our program flow.
- Take a look at the following *pre-class* example :



```python
def modular_function(n):
    return lambda x: x ** n

power_of_2 = modular_function(2)  # first sub-function derived from def
power_of_3 = modular_function(3)  # second sub-function derived from def
power_of_4 = modular_function(4)  # third sub-function derived from def

print(power_of_2(2))  # 2 to the power of 2
print(power_of_3(2))  # 2 to the power of 3
print(power_of_4(2))  # 2 to the power of 4
```

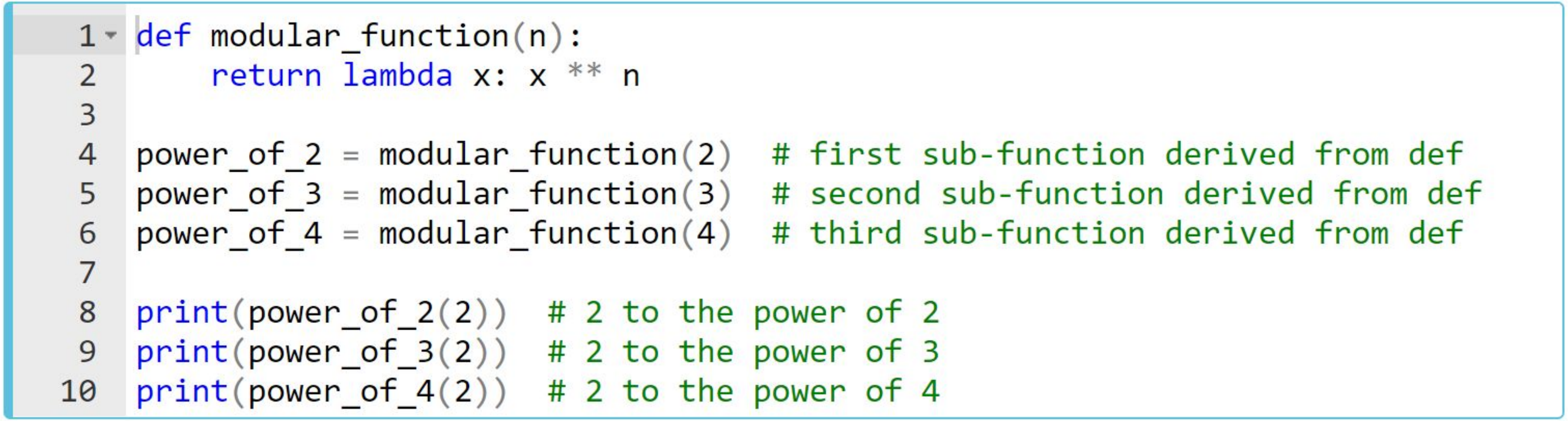
What is the output?

Try to figure out in your mind...

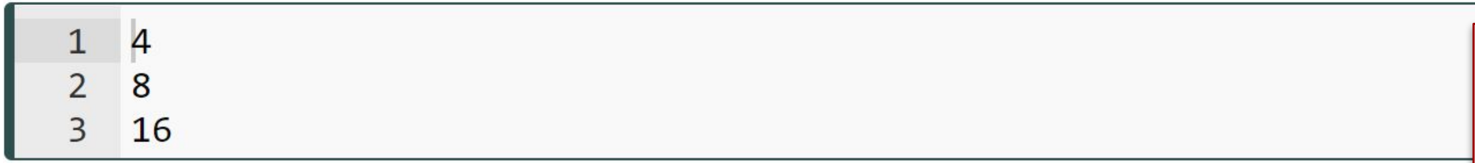
Students, write your response!

Lambda within User-Defined Functions

- **Lambda within def :**



```python
def modular_function(n):
    return lambda x: x ** n

power_of_2 = modular_function(2)  # first sub-function derived from def
power_of_3 = modular_function(3)  # second sub-function derived from def
power_of_4 = modular_function(4)  # third sub-function derived from def

print(power_of_2(2))  # 2 to the power of 2
print(power_of_3(2))  # 2 to the power of 3
print(power_of_4(2))  # 2 to the power of 4
```



```
4
8
16
```

Read the descriptions on the next slide

Lambda within User-Defined Functions

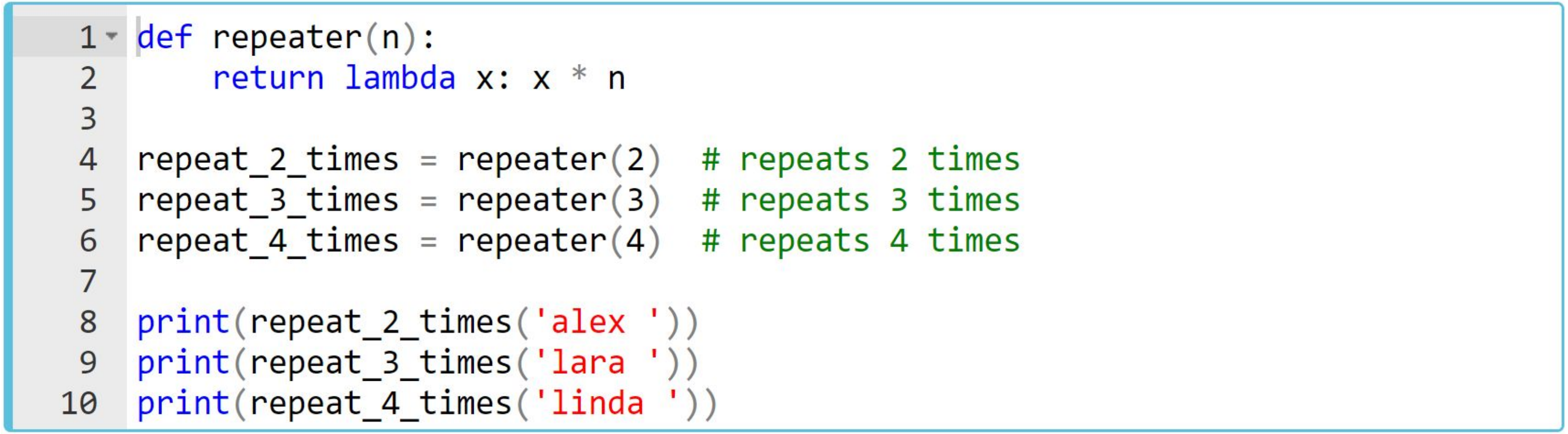
- The `modular_function` takes one argument, number n, and returns a function that takes the power of any given number `x` by that `n`.
- This usage enabled us to use a **function** as **flexible**. Thanks to `lambda`, we could use a single `def` in different ways with the arguments we wanted. We've created three **sub-functions** derived from a single `def`. This is flexibility!

Lambda within User-Defined Functions

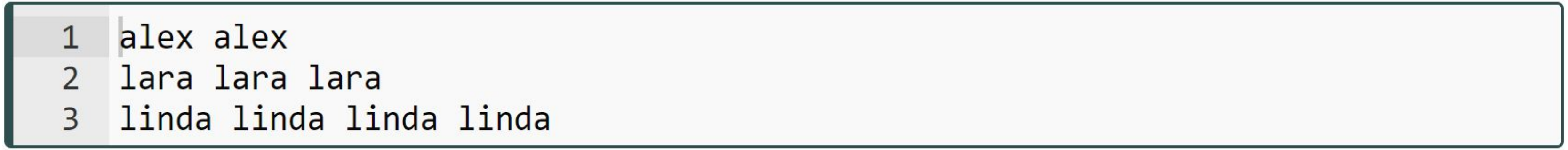
- **Task :** (pre-class content)
  - We can define a function with the same logic as the previous example that repeats the string passed into it.
  - Define a function (`def`) named `repeater` using `lambda` to print the string `n` times.

Lambda within User-Defined Functions

- The sample code and the output :



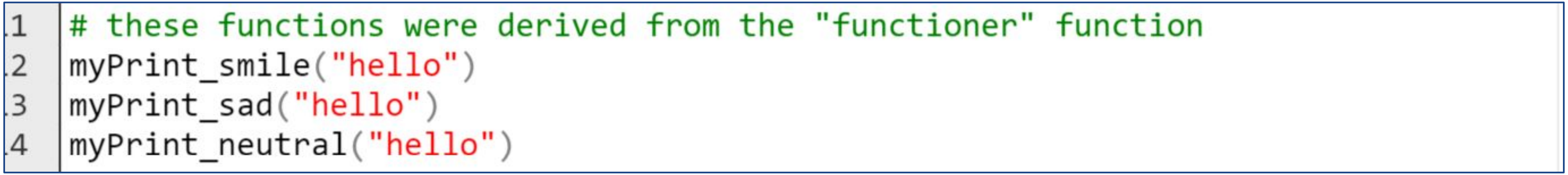
```python
def repeater(n):
    return lambda x: x * n

repeat_2_times = repeater(2)  # repeats 2 times
repeat_3_times = repeater(3)  # repeats 3 times
repeat_4_times = repeater(4)  # repeats 4 times

print(repeat_2_times('alex '))
print(repeat_3_times('lara '))
print(repeat_4_times('linda '))
```



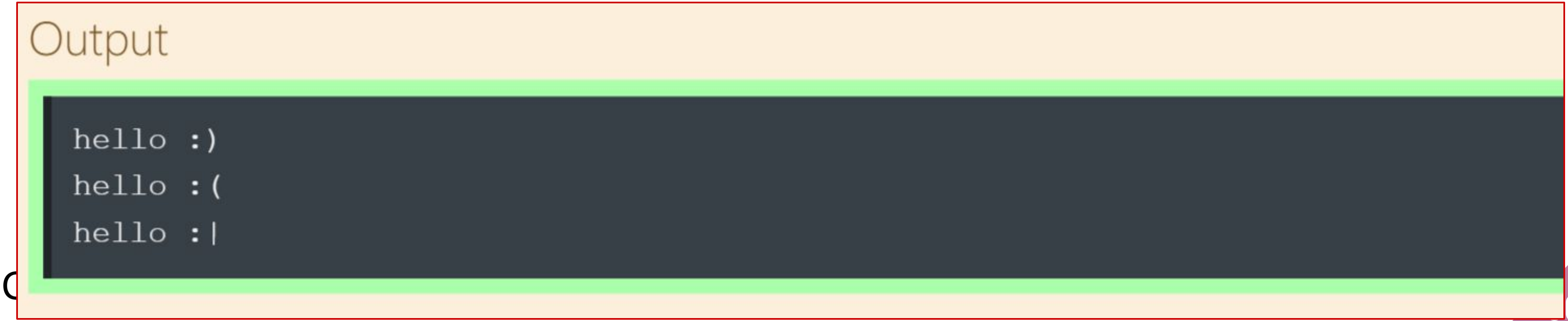
```
alex alex
lara lara lara
linda linda linda linda
```

Lambda within User-Defined Functions

- **Task :**
  - Define a *simple* function (`def`) named `functioner` using `lambda` to create your own `print` function with ***emoji faces***. Such as :



```python
# these functions were derived from the "functioner" function
myPrint_smile("hello")
myPrint_sad("hello")
myPrint_neutral("hello")
```

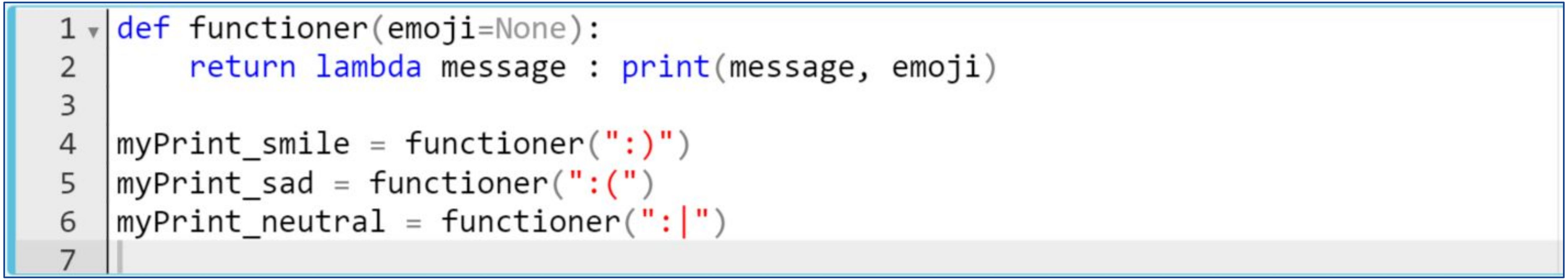


Output

```
hello :)
hello :(
hello :|
```

Lambda within User-Defined Functions

- The sample code and the output :



```
def functioner(emoji=None):
    return lambda message : print(message, emoji)

myPrint_smile = functioner(":)")
myPrint_sad = functioner(":(")
myPrint_neutral = functioner(":|")

```

THANKS!

Any questions?

You can find me at:

- @joseph
- joseph@clarusway.com