How to Use Python Lambda Functions

The identity function, a function that returns its argument, is expressed with a standard Python function definition using the keyword `def` as follows:

```
>>>
>>> def identity(x):return x
```

`identity()` takes an argument x and returns it upon invocation.

identity function

In contrast, if you use a Python lambda construction, you get the following:

```
>>>
>>> lambda x: x
```

In the example above, the expression is composed of:

- **The keyword:** `lambda`
- **A bound variable:** `x`
- **A body:** `x`

Note: In the context of this article, a **bound variable** is an argument to a lambda function.

In contrast, a **free variable** is not bound and may be referenced in the body of the expression. A free variable can be a constant or a variable defined in the enclosing scope of the function.

You can write a slightly more elaborated example, a function that adds 1 to an argument, as follows:

identity function

You can write a slightly more elaborated example, a function that adds 1 to an argument, as follows:

```
>>>
>>> lambda x: x + 1
```

You can apply the function above to an argument by surrounding the function and its argument with parentheses:

```
>>>
>>> (lambda x: x + 1)(2)
3
```

REDUCTION

Reduction is a lambda calculus strategy to compute the value of the expression. In the current example, it consists of replacing the bound variable x with the argument 2:

```
(lambda x: x + 1)(2) = lambda 2: 2 + 1

                     = 2 + 1

                     = 3
```

Because a lambda function is an expression, it can be named. Therefore you could write the previous code as follows:

```
>>>

>>> add_one = lambda x: x + 1

>>> add_one(2)

3
```

The above lambda function is equivalent to writing this:

```
def add_one(x): return x + 1
```

Multi-argument functions

These functions all take a single argument. You may have noticed that, in the definition of the lambdas, the arguments don’t have parentheses around them. Multi-argument functions (functions that take more than one argument) are expressed in Python lambdas by listing arguments and separating them with a comma (`,`) but without surrounding them with parentheses:

```
>>>

>>> full_name = lambda first, last: f'Full name: {first.title()} {last.title()}'
>>> full_name('guido', 'van rossum')
'Full name: Guido Van Rossum'
```

The lambda function assigned to `full_name` takes two arguments and returns a string interpolating the two parameters `first` and `last`. As expected, the definition of the lambda lists the arguments with no parentheses, whereas calling the function is done exactly like a normal Python function, with parentheses surrounding the arguments.

F-string functions

https://note.nkmk.me/en/python-f-strings/

```
s = 'abc'
(f'right : {s:*>8}')
(f'center: {s:*^8}')
(f'left : {s:*<8}’)
# right : *****abc
# center: **abc***
# left : abc*****
```

Python Lambda Map Functions

```
l=[1,2,3,4,5];
sumOfList=0

for i in l:
    sumOfList+=i*i;
print sumOfList
```

```
l = [1,2,3,4,5]
print(sum(i*i for i in l))
```

```
sum(map(lambda x:x*x,l))
```

l=[1,2,3,4,5]
sum(map(lambda x:x*x,l))

Python Lambda Reduce Functions

The **reduce(fun,seq)** function is used to **apply a particular function passed in its argument to all of the list elements** mentioned in the sequence passed along. This function is defined in "**functools**" module.

Working :

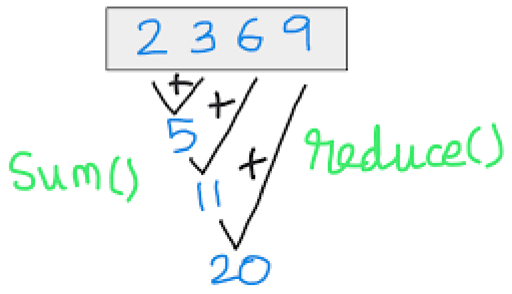
At first step, first two elements of sequence are picked and the result is obtained.

Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.

This process continues till no more elements are left in the container.

The final returned result is returned and printed on console.

```
from functools import reduce
reduce(lambda x,y:x+y, [2,3,6,9])
```



Python Lambda Map-Reduce Functions

```
from functools import reduce
reduce(lambda x,y:x+y, [2,3,6,9])
reduce(lambda x, y: x+ y*y, [1, 2, 3, 4, 5])
reduce(lambda x,y: x+y*y,l)
```

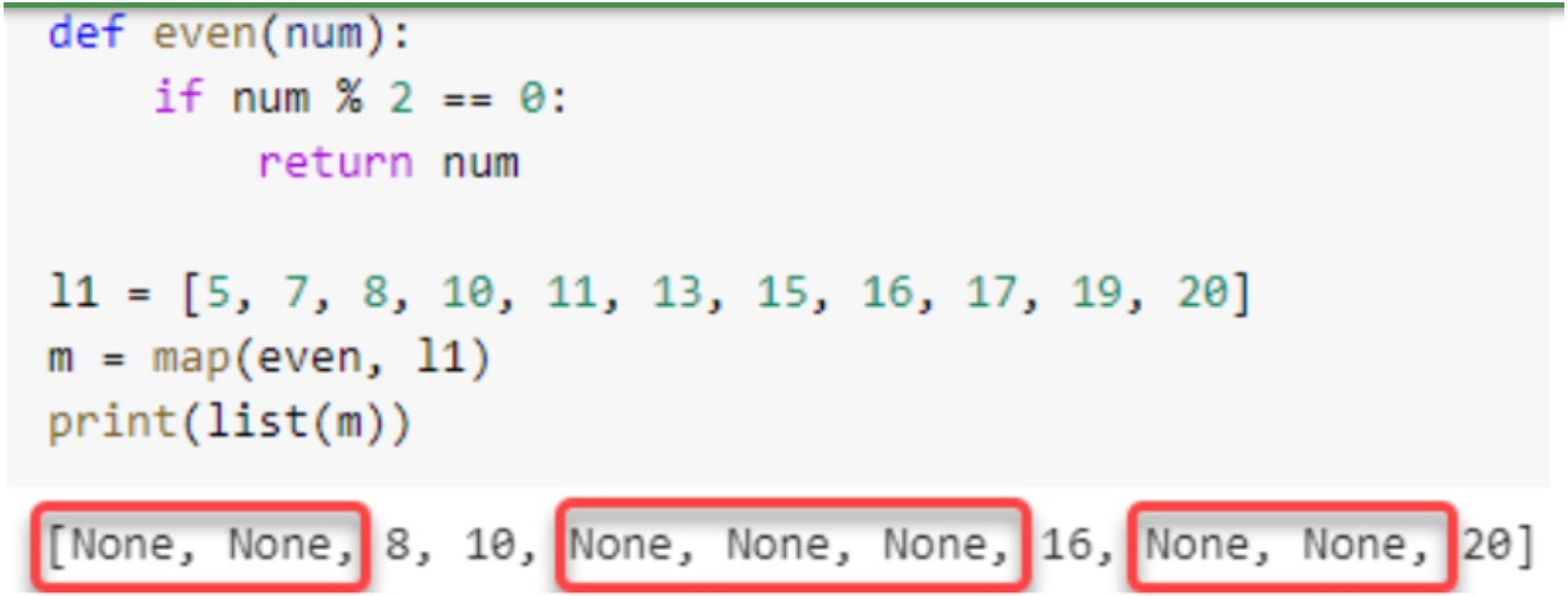
```
 l=[1,2,3,4,5]
sum(map(lambda x:x*x,l))
from functools import reduce
reduce(lambda x,y: x+y*y,l)
```

Map(f(x),L): Το x σαρώνει όλα τα στοιχεία της λίστας και για κάθε x εφαρμόζεται η f()→ f.e..x=x*x for each x in L

Reduce(f(x,y), L): Το x είναι ο accumulator με αρχική τιμή 0, το y σαρώνει όλα τα στοιχεία της λίστας και για κάθε y εφαρμόζεται η f() η τιμή της οποίας προστίθεται στην προηγούμενη τιμή του x και το αποτέλεσμα ξαναποθηκεύεται στο x.

f.e...x=0, x=x+y*y for each y in L

Map and filter() Functions

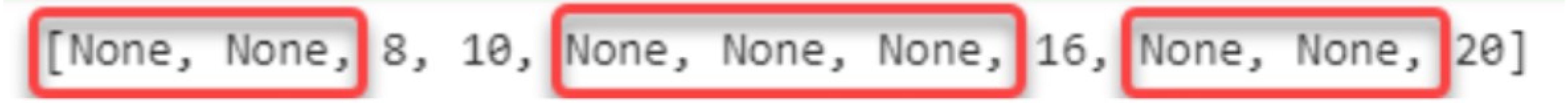


```
def even(num):
    if num % 2 == 0:
        return num


l1 = [5, 7, 8, 10, 11, 13, 15, 16, 17, 19, 20]
m = map(even, l1)
print(list(m))
```

```
[None, None, 8, 10, None, None, None, 16, None, None, 20]
```

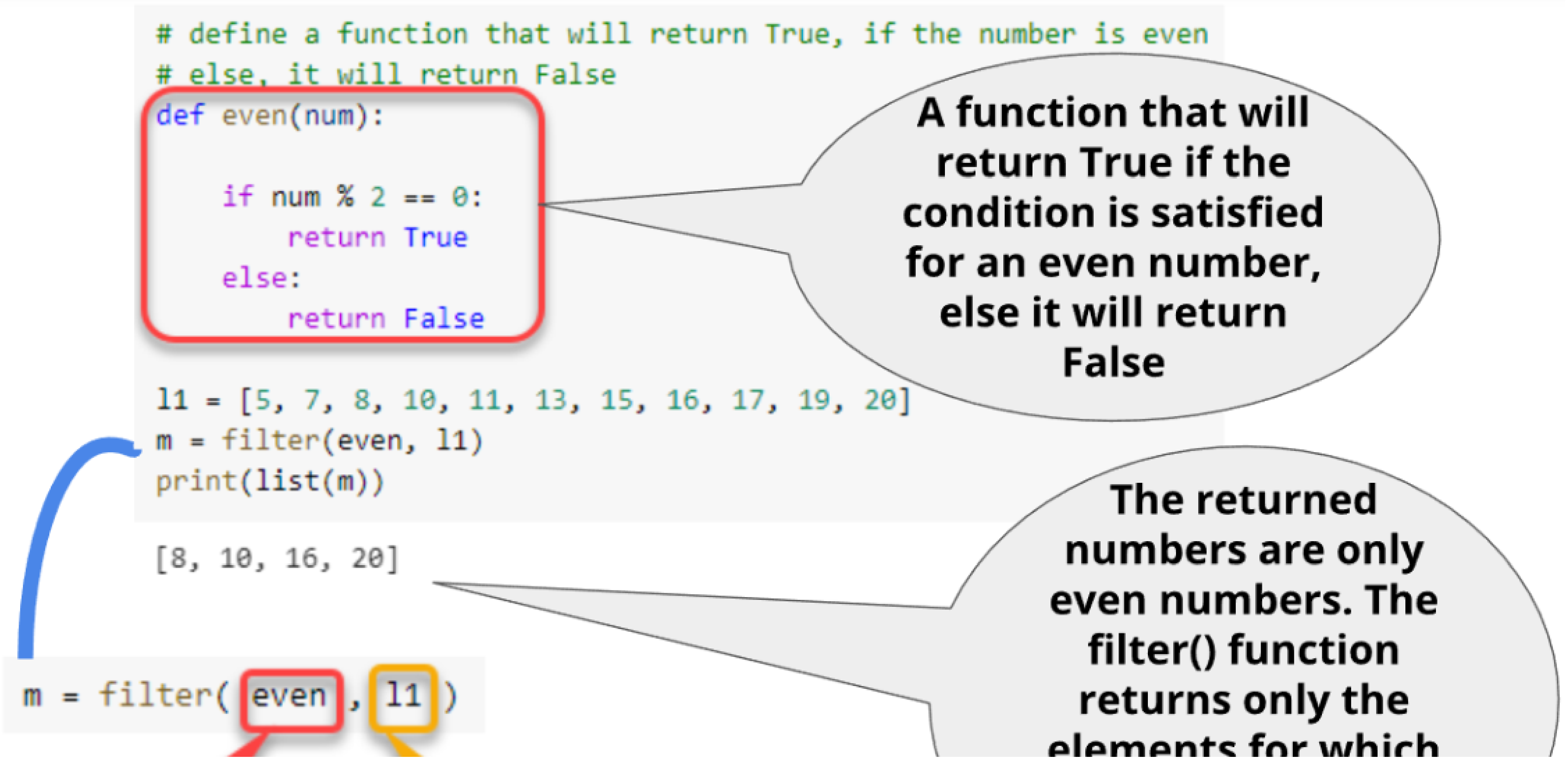
Map and filter() Functions



```
[None, None, 8, 10, None, None, None, 16, None, None, 20]
```

We are getting 'None' if the number is not even. Do we want this? No. We want to get only the even numbers.

Map and filter() Functions



```python
# define a function that will return True, if the number is even
# else, it will return False
def even(num):

    if num % 2 == 0:
        return True
    else:
        return False

l1 = [5, 7, 8, 10, 11, 13, 15, 16, 17, 19, 20]
m = filter(even, l1)
print(list(m))
```

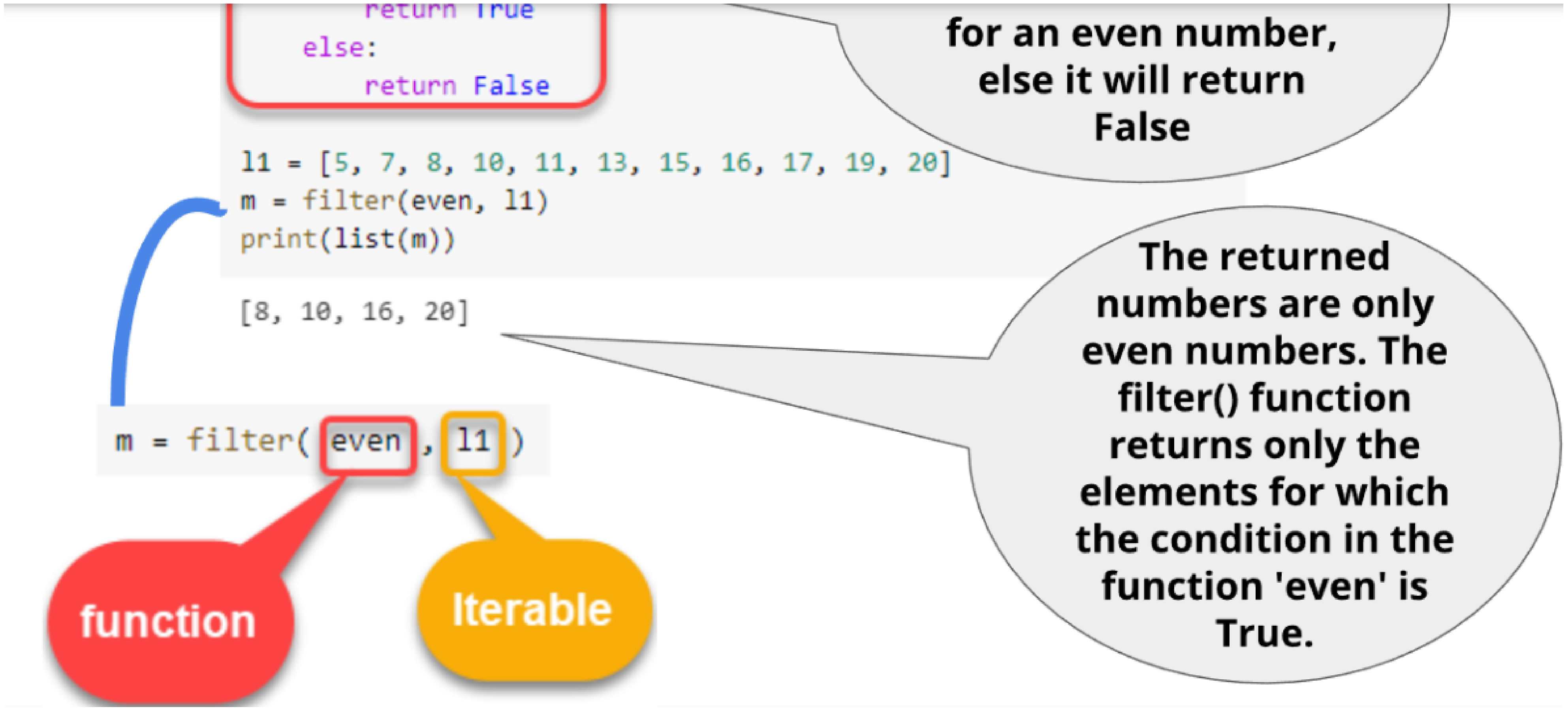
```
[8, 10, 16, 20]
```

A function that will return True if the condition is satisfied for an even number, else it will return False

The returned numbers are only even numbers. The filter() function returns only the elements for which

```python
m = filter( even , l1 )
```

Map and filter() Functions



```
        return True
    else:
        return False

l1 = [5, 7, 8, 10, 11, 13, 15, 16, 17, 19, 20]
m = filter(even, l1)
print(list(m))
```

```
[8, 10, 16, 20]
```

```
m = filter( even , l1 )
```

for an even number, else it will return False

The returned numbers are only even numbers. The filter() function returns only the elements for which the condition in the function 'even' is True.