Those functions allow you to apply a function across an iterable quickly. They all return a generator that can be converted to a list using the list built-in function.

**1. The map() Function**

The map() function takes a function and a list (or any other iterable) and executes the function on every object of the list. It then returns the sequence containing all the results.

**For example:**

def upper\_string(s):

return s.upper()

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]

map\_object = map(upper\_string, fruit)

print(list(map\_object))

>> ['APPLE', 'BANANA', 'PEAR', 'APRICOT', 'ORANGE']

The returned map\_object will contain every fruit in capital letters due to every fruit passed to the upper\_string function.

**2. The filter() function**

Similar to map(), filter() takes a function object and an iterable and creates a new list.

As the name suggests, filter() forms a new list that contains only the elements that satisfy a particular condition, this condition is the function we passed as an argument, if it returns True, the element will be added to the new list, else it won’t. This needs to return a True or False to work.  
For example, the following snippet will return only the strings starting with an A.

def starts\_with\_A(s):

return s[0] == "A"

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]

filtered\_object = filter(starts\_with\_A, fruit)

print(list(filtered\_object))

>> ['Apple', 'Apricot']

**3. The functools.reduce() Function**

reduce() works differently than map() and filter(). It does not return a new list based on the function and iterable we’ve passed. Instead, it returns a single value.

For example, the sum function is a reducing operation, it takes a list as an argument and returns a single number.

A function passed to reduce needs to receive two arguments that will represent the first and the second element of the iterable, and then the second and the third, and then the third and the fourth, etc…

For example, a snippet that recreates the sum function:

from functools import reduce

def sum\_numbers(first, second):

return first+second

my\_list = [1, 3, 5, 7]

reduced\_list = reduce(sum\_numbers, my\_list)

print(reduced\_list)

>> 16

**4. Lambda functions**

Lambda functions are “one-line functions”, they only receive arguments and return a value, and they don’t need to be stored in variables (but they can!).

Here is the syntax of a lambda function:

lambda arg1, arg2: value\_returned

For example, a function that returns a string in capital letters:

lambda s: s.upper()

You can also store this function into a variable if you want to reuse it:

my\_function = lambda s: s.upper()

# This is the same as doing:

def my\_function(s):

return s.upper()

Those functions can be handy when using maps, filter, and reduce operations because they allow you to create it quickly. For example, let’s recreate the previous snippets using lambda functions.

The map example:

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]

map\_object = map(lambda s: s.upper(), fruit)

print(list(map\_object))

The filter example:

fruit = ["Apple", "Banana", "Pear", "Apricot", "Orange"]

filtered\_object = filter(lambda s: s[0] == "A", fruit)

print(list(filtered\_object))

The reduce example:

from functools import reduce

my\_list = [1, 3, 5, 7]

reduced\_list = reduce(lambda first, second: first+second, my\_list)

**Exercise**

people = ["Rick", "Morty", "Beth", "Jerry", "Snowball"]

Using map and filter, try to say hello to everyone who's name is less than or equal to 4 letters

people = ["Rick", "Morty", "Beth", "Jerry", "Snowball"]

def less\_than\_4(s):

    return len(s) <= 4

short\_name = filter(less\_than\_4,people)

def say\_hello(name):

    return f"Hello {name}"

display\_names = map(say\_hello, short\_name)

print(list(display\_names))

# with lambda:

short\_name = filter(lambda name: len(name)<=4, people)

display\_name = map(lambda name: f"Hello {name}!!", short\_name)

for name in list(display\_name):

    print(name)