Chapter 11 - Functional Programming

11 Functional Programming

In this section, we will be learning the about the functions `map` and `reduce`. These functions are paradigms (meaning programs constructed by applying and composing other functions) of functional programming. They are used to shorten codes without needing to worry about loops and conditionals (`if` statements).

11.1 Map

The Python `map()` function has the following syntax

```
map(func, iterables, ...)
```

The first parameter `func` stands for function and the second parameter `*iterables` means that you can add as many iterable objects like elements (lists, tuples, strings).

Example

```
# to convert a tuple of strings to upper case
names = ('john', 'peter', 'william', 'ben')

# storing the map output into a list
uppered_names = list(map(str.upper, names))
print(uppered_names)
```

the output is

```
['JOHN', 'PETER', 'WILLIAM', 'BEN']
```

From the example, the function we gave to `map` is `str.upper`. `str.upper` is the `upper()` function from the String class and it converts all characters to upper case. Note that we do not use this `str.upper()` syntax in the `func` parameter as that is called internally by the map function on **each element**. Let's see how this task is to be done without the `map()` function.

```
names = ('john', 'peter', 'william', 'ben')
uppered_names = []

for name in names:
    # change the string to uppercase
    name_ = name.upper()
    # add the result to the list
    uppered_names.append(name_)

print(uppered_names)
```

What happens if the function that you want to use requires more than one input parameters? Let's take the `round()` function, for example. It is a built-in Python function that takes accepts 2 parameters (a number to round up and the number of decimal places to round the number up to) thus we need to supply the `map()` with 2 sequences.

```
# list of numbers to round up to 2 decimal places
nums = [5.852185, 9.1555562, 71.159213, 215.15632523]
# creating a uniform list of decimal places w.r.t the length of numbers
dec_places = [2] * len(nums)

# storing the map output into a list
rounded_nums = list(map(round, nums, dec_places))
print(rounded_nums)
```

When we run that, the output is

```
[5.85, 9.16, 71.16, 215.16]
```

If the sequences passed to the `map()` does not match, meaning that the length of `nums` differs from the length of `dec_places`, Python will just execute `map` based on the sequence with the **shortest** length.

```
# list of numbers to round up to 2 decimal places
nums = [5.852185, 9.1555562, 71.159213, 215.15632523]
# this list is no longer uniform to nums
dec_places = [2,2]

# storing the map output into a list
rounded_nums = list(map(round, nums, dec_places))
print(rounded_nums)
```

When we run that, the output is

```
[5.85, 9.16]
```

Let's try the `map()` function with a custom function. We have a function that supposed to return the square of a given number but it does something weird to it. In the code block below, this function is called `sneaky_func()` and it has 1 argument. We can see that custom functions are used the same way as any built-in functions for `map()`.

```
def sneaky_func(num):
    return num*num - 5

numbers = [1,2,3,4,5,6]
print(list(map(sneaky_func, numbers)))
# output: [-4, -1, 4, 11, 20, 31]
```

11.2 Reduce

The Python `reduce()` function has the following syntax:

```
reduce(func, iterable[, initial])
```

This function is located in the `functools` library in Python thus we have to remember to import this library when using this function. The purpose of this `reduce()` function is to apply a function of **2 parameters** cumulatively to the elements of a sequence so as to reduce the sequence into **a single value**.

From the syntax, it looks like the function takes in 3 parameters but the parameter in the square brackets `[]` is optional. The first parameter `func` is a function that requires 2 input parameters. The second `iterable` is any iterable object and the third (optional) `initial` parameter is an element that is placed before all elements of the iterable object used in the calculation. The `initial` parameter also serves as a default when the `iterable` is empty.

Internally, the `reduce()` function performs the following steps:

1. **Apply** the function to the first 2 element of the iterable then generate a partial result.
2. **Use** this partial result together with the 3rd element of the iterable to generate another partial result.
3. **Repeat** the process until the last element of the iterable then return the single cumulative value.

This can be shown using the procedures, code example and figure 1 below and on the next page.

```
# importing the reduce function from the functool library
from functools import reduce
# list of nums
nums = [2, 8, 9, 3, 4]

def custom_sum(x, y):
    return x + y

# adding the elements in the list cumulatively
result = reduce(custom_sum, nums)
print("Result is: ", result)
```

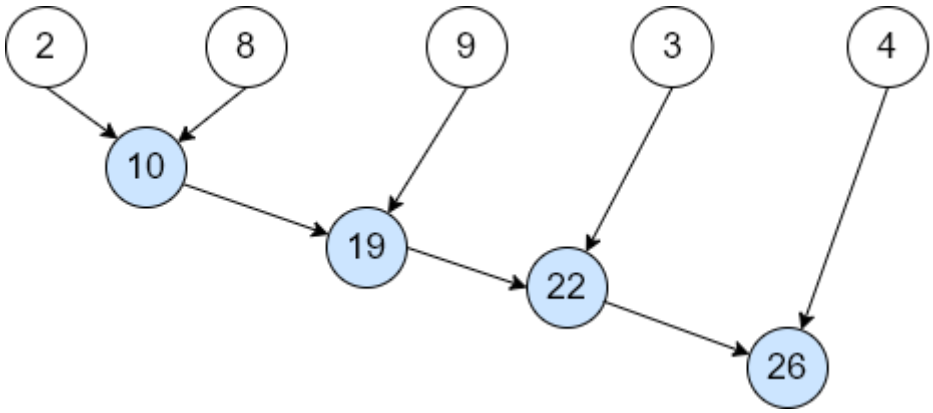

Figure 1: Pictorial represenation of the internal steps of the `reduce()` function.

Step 1: 2 and 8 gets passed to the function `custom_sum()`.
Step 2: It gets added and the result (10) is returned.
Step 3: The `reduce()` function will store 10 as the new or updated value of `x` (the blue circles in figure 1 above)
Step 4: The next element 9 and the stored result 10 is then passed to the function `custom_sum()`.
Step 5: Repeat Step 2 to Step 4 till the last element of the iterable object has been reached then return the final result.

When we run the code, the output is

```
Result is:  26
```

Without the `reduce()` function a loop would have to be used as shown below.

```
nums = [2, 8, 9, 3, 4]

def custom_sum(list_of_nums):
    total = 0
    # add the elements in the list cumulatively
    for num in list_of_nums:
        total += num

    return total

result = custom_sum(nums)
print("Result is: ", result)
```

But how does the `initial` parameter changes the output? Try using the same steps from above along with figure 2 and the code example below and on the next page to help with your understanding.

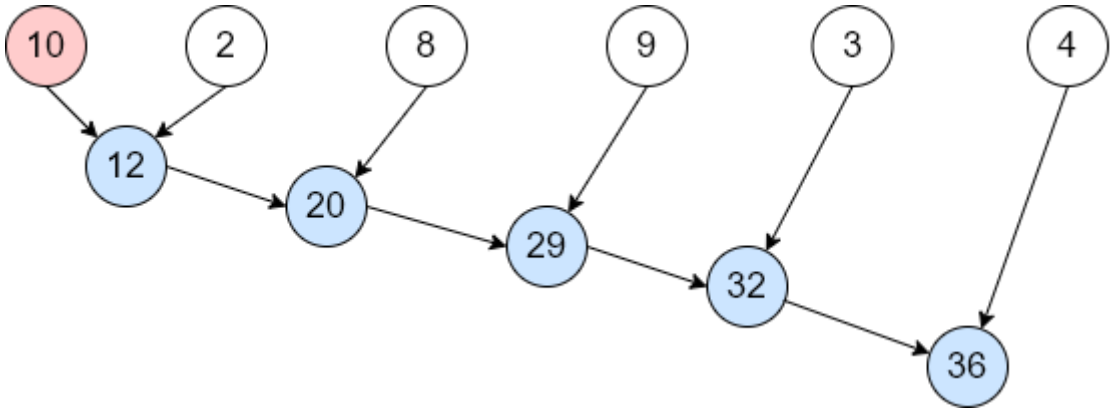

Figure 2: Pictorial represenation of the internal steps of the `reduce()` function with `initial` value of 10.

```python
# importing the reduce function from the functool library
from functools import reduce
# list of nums
nums = [2, 8, 9, 3, 4]

def custom_sum(first, second):
    return first + second
# we are add the elements in the list cumulatively
# with the inital value, the list now looks like
# [10, 3, 4, 6, 9, 5, 12] when passed to the function
result = reduce(custom_sum, nums, 10)
print("Result is: ", result)
```

The output is

```
Result is:  49
```

Note that the `initial` parameter for the `reduce()` function **must always** be a single value.

11.3 References

1. Ramos, June 2020, Python's reduce(): From Functional to Pythonic Style, https://realpython.com/python-reduce-function/
2. Sideris, Tutorial: Python Functions and Functional Programming, https://www.dataquest.io/blog/introduction-functional-programming-python/