Functions as Arguments

This lesson teaches us how to use functions as parameters for other functions.

We'll cover the following

In Python, one function can become an argument for another function. This is useful in many cases.

Let's make a `calculator` function that requires the `add`, `subtract`, `multiply`, or `divide` function along with two numbers as arguments.

For this, we'll have to define the four arithmetic functions as well.

Using Simple Functions

```
def add(n1, n2):
    return n1 + n2


def subtract(n1, n2):
    return n1 - n2


def multiply(n1, n2):
    return n1 * n2


def divide(n1, n2):
    return n1 / n2


def calculator(operation, n1, n2):
    return operation(n1, n2)  # Using the 'operation' argument as a function


result = calculator(multiply, 10, 20)
print(result)
print(calculator(add, 10, 20))
```

Python automatically understands that the `multiply` argument in **line 16** is a function, and so, everything works perfectly.

Using Lambdas

In the last lesson, we were discussing the purpose of lambdas. Well, now it's their time to shine.

For the `calculator` method, we needed to write four extra functions that could be used as the argument. This can be quite a hassle.

Why don't we just pass a lambda as the argument? The four operations are pretty simple, so they can be written as lambdas.

Let's try it:

```
def calculator(operation, n1, n2):
    return operation(n1, n2)  # Using the 'operation' argument as a function


# 10 and 20 are the arguments.
result = calculator(lambda n1, n2: n1 * n2, 10, 20)
# The lambda multiplies them.
print(result)

print(calculator(lambda n1, n2: n1 + n2, 10, 20))
```

The code looks much cleaner now! We can define the `operation` on the go whenever we want.

This is the beauty of lambdas. They work really well as arguments for other functions.

More Examples

The built-in `map()` function creates a **map object** using an existing list and a function as its parameters. This object can be converted to a list using the `list()` function (more on this later).

The template for `map()` is as follows:

```
map(function, list)
```

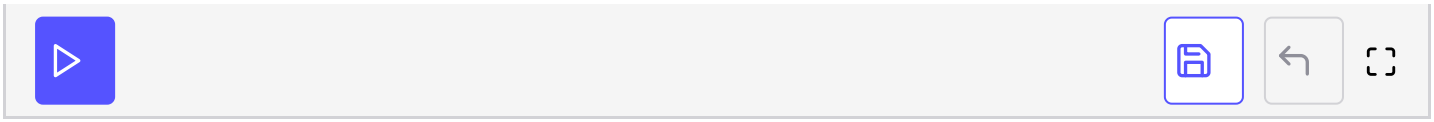
The `function` will be applied, or *mapped*, to all the elements of the `list`.

Below, we'll use `map()` to double the values of an existing list:

```
num_list = [0, 1, 2, 3, 4, 5]

double_list = map(lambda n: n * 2, num_list)

print(list(double_list))
```



This creates a new list. The original list remains unchanged.

We could have created a function that doubles a number and used it as the argument in `map()`, but the lambda made things simpler.

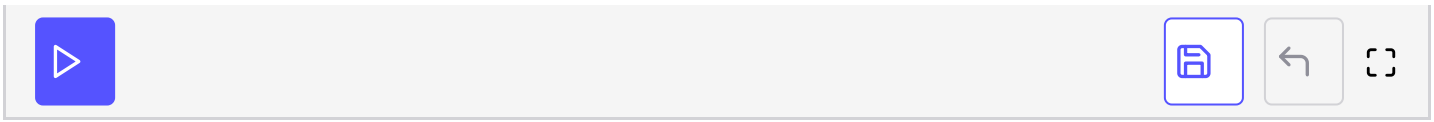
Another similar example is the `filter()` function. It requires a function and a list.

`filter()` *filters* elements from a list if the elements satisfy the condition that is specified in the argument function.

Let's write a `filter()` function that filters all the elements which are greater than `10`:

```
numList = [30, 2, -15, 17, 9, 100]

greater_than_10 = list(filter(lambda n: n > 10, numList))
print(greater_than_10)
```



The function returns a **filter object** which can be converted to a list using `list()`.

just like `map()`, `filter()` returns a new list without changing the original one.

By now, we have a better understanding of how functions can become arguments and why lambdas are helpful in that situation.

In the next lesson, we'll explore another powerful feature of functions: **recursion**.