Python Lambda Expressions

Summary: in this tutorial, you'll learn about Python lambda expressions and how to use them to write anonymous functions.

Sometimes, you need to write a simple function (https://www.pythontutorial.net/python-basics/python-functions/) that contains one expression. However, you need to use this function once. And it'll unnecessary to define that function with the `def` keyword.

That's where the Python lambda expressions come into play.

What are Python lambda expressions

Python lambda expressions allow you to define anonymous functions.

Anonymous functions are functions without names. The anonymous functions are useful when you need to use them once.

A lambda expression typically contains one or more arguments, but it can have **only one expression**.

The following shows the lambda expression syntax:

```
lambda parameters: expression
```

It's equivalent to the following function without the `"anonymous"` name:

```
def anonymous(parameters):
    return expression
```

Python lambda expression examples

In Python, you can pass a function to another function or return a function from another function.

1) Functions that accept a function example

The following defines a function called `get_full_name()` that format the full name from the first name and last name:

```
def get_full_name(first_name, last_name, formatter):
    return formatter(first_name, last_name)
```

The `get_full_name()` function accepts three arguments:

- First name (`first_name`)
- Last name (`last_name`)
- A function that will format the full name (`formatter`). In turn, the `formatter` function accepts two arguments first name and last name.

The following defines two functions that return a full name from the first name and last name in different formats:

```
def first_last(first_name, last_name):
    return f"{first_name} {last_name}"


def last_first(first_name, last_name):
    return f"{last_name}, {first_name}"
```

And this shows you how to call the `get_full_name()` function by passing the first name, last name, and `first_last` / `last_first` functions:

```
full_name = get_full_name('John', 'Doe', first_last)
print(full_name) # John Doe

full_name = get_full_name('John', 'Doe', last_first)
print(full_name) #  Doe, John
```

Output:

```
John Doe
Doe, John
```

Instead of defining the `first_last` and `last_first` functions, you can use lambda expressions.

For example, you can express the `first_last` function using the following lambda expression:

```
lambda first_name,last_name: f"{first_name} {last_name}"
```

This lambda expression accepts two arguments and concatenates them into a formatted string in the order `first_name` , space, and `last_name` .

And the following converts the `last_first` function using a lambda expression that returns the full name in the format: last name, space, and first name:

```
lambda first_name, last_name: f"{last_name} {first_name}";
```

By using lambda expressions, you can call the `get_full_name()` function as follows:

```
def get_full_name(first_name, last_name, formatter):
    return formatter(first_name, last_name)


full_name = get_full_name(
    'John',
```

```
    'Doe',
    lambda first_name, last_name: f"{first_name} {last_name}"
)
print(full_name)

full_name = get_full_name(
    'John',
    'Doe',
    lambda first_name, last_name: f"{last_name} {first_name}"
)
print(full_name)
```

Output:

```
John Doe
Doe, John
```

2) Functions that return a function example

The following `times()` function returns a function which is a lambda expression:

```
def times(n):
    return lambda x: x * n
```

And this example shows how to call the `times()` function:

```
double = times(2)
```

Since the `times()` function returns a function, the `double` is also a function. To call it, you place parentheses like this:

```
result = double(2)
print(result)
```

```
result = double(3)
print(result)
```

Output:

```
4
6
```

The following shows another example of using the `times()` function:

```
triple = times(3)

print(triple(2))  # 6
print(triple(3))  # 9
```

Python lambda in a loop

See the following example:

```
callables = []
for i in (1, 2, 3):
    callables.append(lambda: i)

for f in callables:
    print(f())
```

How it works.

- First, define a list with the name callables.
- Second, iterate from 1 to 3, create a new lambda expression in each iteration, and add it to the callables list.
- Third, loop over the callables and call each function.

The expected output will be:

```
1
2
3
```

However, the program shows the following output:

```
3
3
3
```

The problem is that all the there lambda expressions reference the `i` variable, not the current value of `i`. When you call the lambda expressions, the value of the variable `i` is 3.

To fix this, you need to bind the `i` variable to each lambda expression at the time the lambda expression is created. One way to do it is to use the default argument (https://www.pythontutorial.net/python-basics/python-default-parameters/):

```
callables = []
for i in (1, 2, 3):
    callables.append(lambda a=i: a)

for f in callables:
    print(f())
```

In this example, the value of a is evaluated at the time the lambda expression is created. Therefore, the program returns the expected output.

Summary

- Use Python lambda expressions to create anonymous functions, which are functions without names.
- A lambda expression accepts one or more arguments, contains an expression, and returns the result of that expression.
- Use lambda expressions to pass anonymous functions to a function and return a function from another function.