جامعة الانبار
كلية علوم الحاسوب وتكنولوجيا المعلومات
قسم أنظمة شبكات الحاسوب

برمجة كيانية OOP
المرحلة الثانية
الفصل الدراسي الأول والثاني

مدرس المادة
م.د. سميه عبدالله حمد

Lambda Functions

Although Python is an object-oriented programming language, lambda functions are handy when you are doing various kinds of functional programming.

In Python, functions can take in one or more positional or keyword arguments, a variable list of arguments, a variable list of keyword arguments, and so on. They can be passed into a higher-order function and returned as output. Regular functions can have several expressions and multiple statements. They also always have a name.

A Python lambda function is simply an anonymous function. It could also be called a nameless function. Normal Python functions are defined by the **def** keyword.

Lambda functions in Python are usually composed of the lambda keyword, any number of arguments, and one expression.

Lambda functions are mostly used as one-liners. They are used very often within higher-order functions like map() and filter(). This is because anonymous functions are passed as arguments to higher-order functions, which is not only done in Python programming.

Because Python is an object-oriented programming language, everything is an object. Python classes, class instances, modules and functions are all handled as objects.

A function object can be assigned to a variable. It is common to assign variables to regular functions in Python. This behavior can also be applied to lambda functions. This is because they are function objects, even though they are nameless:

```
def greet(name):
    return f'Hello {name}'
greetings = greet
greetings('Clint')
<<<<
Hello Clint
```

Higher-order functions like map(), filter(), and reduce()

It's likely you'll need to use a lambda function within built-in functions such as filter() and map(), and also with reduce() — which is imported from the functools module in Python, because it's not a built-in function. **By default, higher-order functions are functions that receive other functions as arguments.**

As seen in the code examples below, the normal functions can be replaced with lambdas, passed as arguments into any of these higher-order functions:

```
#map function
names = ['Clint', 'Lisa', 'Asake', 'Ada']
greet_all = list(map(greet, names))
print(greet_all)
<<<<
['Hello Clint', 'Hello Lisa', 'Hello Asake', 'Hello Ada']
```

```
#filter function
numbers = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
def multiples_of_three(x):
    return x % 3 == 0
print(list(filter(multiples_of_three, numbers)))
<<<<
[12, 15, 18]
#reduce function
numbers =[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
def add_numbers(x, y):
    return x + y
print(reduce(add_numbers, numbers))
<<<<
55
```

The difference between a statement and an expression

A common point of confusion amongst developers is differentiating between a statement and an expression in programming.

- A statement is any piece of code that does something or performs an action such as if or while conditions.
- An expression is made of a combination of variables, values, and operators and evaluates to a new value.

This distinction is important as we explore the subject of lambda functions in Python. An expression like the one below returns a value:

```
square_of_three = 3 ** 2
print(square_of_three)
<<<<
9
```

A statement looks like this:

```
for i in range(len(numbers) , 0, -1):
    if i % 2 == 1:
        print(i)
    else:
        print('even')
<<<<
even 9 even 7 even 5 even 3 even 1
```

How to Use Python Lambda Functions

The Python lambda function must begin with the keyword lambda (unlike normal functions, which begin with the def keyword). The syntax for a lambda function generally goes like this:

```
lambda arguments : expression
```

Lambda functions can take any number of positional arguments, keyword arguments, or both, followed by a colon and only one expression. There cannot be more than one expression, as it is syntactically restricted. Let's examine an example of a lambda expression below:

```
add_number = lambda x, y : x + y
print (add_number(10, 4))
<<<<
14
```

From the example above, the lambda expression is assigned to the variable add_number. A function call is made by passing arguments, which evaluates to 14.

Let's take another example below:

```
discounted_price = lambda price, discount = 0.1, vat = 0.02 :    price * (1 -
                                discount) * (1 + vat)
print(discounted_price(1000, vat=0.04, discount=0.3))
<<<<
728.0
```

As seen above, the lambda function evaluates to 728.0. A combination of positional and keyword arguments are used in the Python lambda function. While using positional arguments, we cannot alter the order outlined in the function definition. However, we can place keyword arguments at any position only after the positional arguments.

This is mostly used with a Python interpreter, as shown in the following example:

```
print((lambda x, y: x - y)(45, 18))
<<<<
27
```

The lambda function object is wrapped within parentheses, and another pair of parentheses follows closely with arguments passed. The expression is evaluated and the function returns a value that is assigned to the variable. Python lambda functions can also be executed within a list comprehension. A list comprehension always has an output expression, which is replaced by a lambda function. Here are some examples:

```
my_list = [(lambda x: x * 2)(x) for x in range(10) if x % 2 == 0]
print(my_list)
<<<<
[0, 4, 8, 12, 16]
```

```
value = [(lambda x: x % 2 and 'odd' or 'even')(x) for x in my_list]
print(value)
<<<<
['even', 'even', 'even', 'even', 'even']
```

Lambda functions can be used when writing ternary expressions in Python. A ternary expression outputs a result based on a given condition. Check out the examples below:

```
test_condition1 = lambda x: x / 5 if x > 10 else x + 5
print(test_condition1(9))
<<<<
14
test_condition2 = lambda x: f'{x} is even' if x % 2 == 0 else (lambda x: f'{x} is odd')(x)
print(test_condition2(9))
<<<<
9 is odd
```

Lambda functions within higher-order functions

The concept of higher-order functions is popular in Python, just as in other languages. They are functions that accept other functions as arguments and also return functions as output.

In Python, a higher-order function takes two arguments: a function, and an iterable. The function argument is applied to each item in the iterable object. Since we can pass a function as an argument to a higher-order function, we can equally pass in a lambda function.

Here are some examples of a lambda function used with the map() function:

```
square_of_numbers = list(map(lambda x: x ** 2, range(10)))
print(square_of_numbers)
<<<<
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
strings = ['Nigeria', 'Ghana', 'Niger', 'Kenya', 'Ethiopia', 'South Africa', 'Tanzania', 'Egypt', 'Morocco', 'Uganda']
length_of_strings = list(map(lambda x: len(x), strings))
print(length_of_strings)
<<<<
[7, 5, 5, 5, 8, 12, 8, 5, 7, 6]
```

Here are some lambda functions used with the filter() function:

```
length_of_strings_above_five = list(filter(lambda x: len(x) > 5, strings))
print(length_of_strings_above_five)
<<<<
 ['Nigeria', 'Ethiopia', 'South Africa', 'Tanzania', 'Morocco', 'Uganda']
fruits_numbers_alphanumerics = ['apple', '123', 'python3', '4567', 'mango', 'orange', 'web3', 'banana', '890']
fruits = list(filter(lambda x: x.isalpha(), fruits_numbers_alphanumerics))
numbers = list(filter(lambda x: x.isnumeric(),
fruits_numbers_alphanumerics))
print(fruits)
print(numbers)
<<<<
['apple', 'mango', 'orange', 'banana']
['123', '4567', '890']
```

Here are some lambda functions used with the reduce() function:

```
values= [13, 6, 12, 23, 15, 31, 16, 21]
max_value = reduce(lambda x,y: x if (x > y) else y, values)
print(max_value)
<<<<
31
Nums=[1, 2, 3, 4, 5, 6]
multiplication_of_nums = reduce(lambda x,y: x*y, nums)
print(multiplication_of_nums)
<<<<
720
```

Conclusion

Although Python lambdas can significantly reduce the number of lines of code you write, they should be used sparingly and only when necessary. The readability of your code should be prioritized over conciseness. For more readable code, always use a normal function where suited over lambda functions, as recommended by the Python Style Guide.

Lambdas can be very handy with Python ternary expressions, but again, try not to sacrifice readability. Lambda functions really come into their own when higher-order functions are being used.

In summary:

Python lambdas are good for writing one-liner functions.

Lambdas should not be used when there are multiple expressions, as it makes code unreadable.

Python is an object-oriented programming language, but lambdas are a good way to explore functional programming in Python.