ADVANCED PYTHON

Day 16: First Class Functions and Lambda Expressions

Welcome to day 16 of the 30 Days of Python series! Today we're going to be talking about the concept of first class functions. All functions in Python are first class functions, and we're going to discuss a little bit about what that means, and why it's useful.

We're also going to be looking at a different way of defining simple functions using lambda expressions.

What are first class functions?

A programming language is said to have first class functions when functions in that language are treated like any other type. They're said to be "first class citizens" of the language.

Generally speaking, this means we can pass functions in as arguments to other functions, return them from functions, and assign them to variables.

In Python, functions are indeed first class citizens, and we can do everything I listed above. Let's look at some examples.

Assigning functions to variables

First, let's talk about assigning functions to variables, because this is actually a vital part of creating a function in Python.

When we define a function using `def`, we provide a name for our function. Two things happen when we do this.

The function gets named according to our specification.

Looking back at day 13, when we printed out the global namespace, we found entries like this:

```
<function add at 0x7fae512021f0>
```

The function name becomes part of this representation of the function.

Python creates a variable for us using this name, and it assigns the function to this name.

These are actually two very different things, and we can demonstrate this by assigning a function to a different variable name, and printing the values associated with these different names.

```
def add(a, b):
        print(a + b)

adder = add
```

```
print(add)
print(adder)
```

If we run the code, we'll get some output that looks something like this:

```
<function add at 0x7f11bc6451f0>
<function add at 0x7f11bc6451f0>
```

There are a couple of interesting things to note here.

Firstly, both of the functions are the exact same object. We can see this because their memory addresses (on the far right) are identical.

Secondly, both of the functions are still called `add` when we print this string representation of the functions. The second output didn't change to this:

```
<function adder at 0x7f11bc6451f0>
```

That's because the variable name we use to refer to the function, and the function's name, are different things.

We can show this very clearly by doing the following:

```
def add(a, b):
        print(a + b)

adder = add
del add

add(5, 4)  # NameError
```

Here we've deleted the variable name, `add`, so we can no longer call the function by this name. If we try, we get a `NameError`:

```
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    add(5, 4)
NameError: name 'add' is not defined
```

Despite this, we can still call `adder`, and the name of the function is still `add`:

```
def add(a, b):
        print(a + b)

adder = add
del add

adder(5, 4)   # 9
print(adder)  # <function add at 0x7f03780ae1f0>
```

Functions as arguments

Since we can assign functions to variables, it makes sense that we can also pass in functions as arguments. Why? Because parameters are really just variables, and arguments are the values we assign to those variables.

Let's look at an interesting case where we can pass in a function as an argument.

Python has a function called `max` which lets us find the largest value in some iterable.

For an iterable where the elements have a natural order, it's very easy to use: we just pass the iterable to max, and it works its magic.

```
numbers = [56, 3, 45, 29, 102, 30, 73]
highest_number = max(numbers)

print(highest_number)  # 102
```

However, not every set of values has an obvious order. For example, let's say I have a list of dictionaries like this:

```
students = [
        {"name": "Hannah", "grade_average": 83},
        {"name": "Charlie", "grade_average": 91},
        {"name": "Peter", "grade_average": 85},
        {"name": "Rachel", "grade_average": 79},
        {"name": "Lauren", "grade_average": 92}
]
```

Realistically we could order this alphabetically, or by grade average. However, in this case, `max` is actually trying to compare a whole dictionary to another dictionary using the `>` operator, and that's not even a legal operation.

In cases like this, we can help the `max` function out by providing a function to a keyword only parameter called `key`. This function gets called for every item in the iterable, and each item is passed to the function, one at a time.

What `max` expects from this function is a sortable value that is can use to determine the order of a given item.

For example, we could provide a function like this:

```
def get_grade_average(student):
        return student["grade_average"]
```

This function accepts a dictionary, and it returns the value associated with the `"grade_average"` key of that dictionary. In our case this is an integer, and integers can be legally compared using `>`.

We can now do something like this:

```
def get_grade_average(student):
        return student["grade_average"]
```

```
students = [
        {"name": "Hannah", "grade_average": 83},
        {"name": "Charlie", "grade_average": 91},
        {"name": "Peter", "grade_average": 85},
        {"name": "Rachel", "grade_average": 79},
        {"name": "Lauren", "grade_average": 92}
]

valedictorian = max(students, key=get_grade_average)
print(valedictorian)
```

Here we provided the `get_grade_average` function as an argument for `max`, and `max` called this function for us internally to determine the order of the items in our list. It then correctly returned this dictionary as the student with the highest grade average:

```
{"name": "Lauren", "grade_average": 92}
```

There are other functions and methods which expect similar sort of functions. For example, the `sorted` function and `sort` methods both accepts keys, which must be functions.

Returning functions from other functions

At this point, I think it's fairly clear that we *can* return functions from other functions, but the question is, why would we want to do this?

Here I want to show you a pattern using the `get` method for dictionaries, which is going to allow us to have a user select a function to run. Methods are really just special types of functions, so `get` will serve as a good demonstration.

First, we need some functions, so I'm just going to use the operator functions we defined in the [exercises for day 12](#).

```
def add(a, b):
        print(a + b)
```

```
def subtract(a, b):
        print(a - b)

def multiply(a, b):
        print(a * b)

def divide(a, b):
        if b == 0:
                print("You can't divide by 0!")
        else:
                print(a / b)
```

These are nice examples, because they all take in two arguments, and they're thematically similar.

The next step is to define a dictionary like this:

```
operations = {
        "a": add,
        "s": subtract,
        "m": multiply,
        "d": divide
}
```

It's really important to note that we're not calling the functions here: we're just referencing the functions using their variable names.

Now we have a dictionary where the keys are strings and the values associated with those keys are functions. We can now ask the user to select from these options, and we'll run the function for them by looking up their option in the dictionary:

```
def add(a, b):
        print(a + b)

def subtract(a, b):
        print(a - b)

def multiply(a, b):
```

```
        print(a * b)

def divide(a, b):
        if b == 0:
                print("You can't divide by 0!")
        else:
                print(a / b)

operations = {
        "a": add,
        "s": subtract,
        "m": multiply,
        "d": divide
}

selected_option = input("""Please select one of the fo
llowing options:

a: add
s: subtract
m: multiply
d: divide

What would you like to do? """)

operation = operations.get(selected_option)

if operation:
        a = int(input("Please enter a value for a: "))
        b = int(input("Please enter a value for b: "))

        operation(a, b)
else:
        print("Invalid selection")
```

When we call `get`, we look up the user's string in our dictionary, and (assuming nothing went wrong) what we get back is a function. We can then assign this function to a variable name and call it just like any other function.

If something did go wrong, we have some protection, because the `get` method returns `None` when the key isn't found, and we can

check for this using a conditional statement to make sure we didn't get a falsy value. If we did, we know that we didn't get a function back, so we can let the user know their selection was invalid.

Lambda expressions

Lambda expressions are an alternative syntax for defining simple functions. One of the very useful features of lambda expressions is the fact that they are *expressions*, and the value they evaluate to is the function we want to create.

In contrast, the function definitions using the `def` keyword are statements. They don't have a value, which is why Python goes through the trouble of assigning the function to a name for us. One major limitation of this is that we always have to define our functions separate from where we want to use them.

For example, we have to define our key function in advance of using it in `max`:

```
def get_grade_average(student):
        return student["grade_average"]

students = [
        {"name": "Hannah", "grade_average": 83},
        {"name": "Charlie", "grade_average": 91},
        {"name": "Peter", "grade_average": 85},
        {"name": "Rachel", "grade_average": 79},
        {"name": "Lauren", "grade_average": 92}
]

valedictorian = max(students, key=get_grade_average)
print(valedictorian)
```

There was no way to define the function as part of the `max` call when using `def`. However, we have no such limitation when using lambda expressions, which makes them very handy for things like key functions, which generally have a one line function body.

Okay, so how do we write a lambda expression?

The first part of any lambda expression is the `lambda` keyword. Directly after the lambda keyword, we optionally specify any parameters for the function we want to create. Unlike in regular function definitions, these values are not put inside parentheses. This section of the lambda expression is closed off with a colon.

After the colon comes an expression, and this expression is what we want to return from the function.

Let's look at a few examples, starting with our `add` function from before.

```
lambda a, b: a + b
```

Here we have two parameters, `a` and `b`, and these are placed after the `lambda` keyword, and before the colon. The colon marks the end of the parameters we want to define, and everything which comes afterwards is what we want to return from the function. In this case, what we want to return is `a + b`.

Just for comparison, here is the same function written using `def`:

```
def add(a, b):
        return a + b
```

Now let's look at our `get_grade_average` that we used as our key function for `max`:

```
def get_grade_average(student):
        return student["grade_average"]
```

This function takes in a single parameter, `student`, which we expect to be a dictionary. It returns the result of a subscription expression

that looks up the value associated with the key `"grade_average"` in the provided dictionary.

Written as a lambda expression, this function would look like this:

```
lambda student: student["grade_average"]
```

We can use it in our little application to find the student with the highest grades like so:

```
students = [
        {"name": "Hannah", "grade_average": 83},
        {"name": "Charlie", "grade_average": 91},
        {"name": "Peter", "grade_average": 85},
        {"name": "Rachel", "grade_average": 79},
        {"name": "Lauren", "grade_average": 92}
]

valedictorian = max(students, key=lambda student: stud
ent["grade_average"])
print(valedictorian)
```

As you can see, the lambda expression just replaces the reference to a function, because the lambda expression itself yields a function.

Another place we can put lambda expressions to good use is in our operator mapping dictionary from before:

```
def divide(a, b):
        if b == 0:
                return "You can't divide by 0!"
        else:
                return a / b

operations = {
        "a": lambda a, b: a + b,
        "s": lambda a, b: a - b,
        "m": lambda a, b: a * b,
        "d": divide
```

```
}

selected_option = input("""Please select one of the fo
llowing options:

a: add
s: subtract
m: multiply
d: divide

What would you like to do? """)

operation = operations.get(selected_option)

if operation:
        a = int(input("Please enter a value for a: "))
        b = int(input("Please enter a value for b: "))

        print(operation(a, b))
else:
        print("Invalid selection")
```

The only function we can't replace here is `divide`, because `divide` has a conditional statement inside. Lambda expressions are limited to single expressions and cannot contain statements.

Remember, all functions in Python are first class functions, so we can assign the functions we create with lambda expressions to variables if we want.

```
add = lambda a, b: a + b

print(add(7, 8))  # 15
```

Generally speaking this isn't very useful though. In situations like this, it's generally more appropriate to define a function with `def`. They tend to be more readable, and saving a few lines by writing a lambda expression is not a great trade-off in that case.

Style note

Be very careful not to abuse lambda expressions in your code. While we can write very long and complicated expressions as return values, it's not a good idea.

If you need a function that performs any really complex logic, it's better to define a function using `def`, and to break the operations up into simpler steps so that you can use descriptive variables and comments to document what's happening.

Exercises

1) Use the `sort` method to put the following list in alphabetical order with regards to the students' names:

```
students = [
        {"name": "Hannah", "grade_average": 83},
        {"name": "Charlie", "grade_average": 91},
        {"name": "Peter", "grade_average": 85},
        {"name": "Rachel", "grade_average": 79},
        {"name": "Lauren", "grade_average": 92}
]
```

You're going to need to pass in a function as a key here, and it's up to you whether you use a lambda expression, or whether you define a function with `def`.

2) Convert the following function to a lambda expression and assign it to a variable called `exp`.

```
def exponentiate(base, exponent):
        return base ** exponent
```

3) Print the function you created using a lambda expression in previous exercise. What is the name of the function that was created?

You can find our solutions to the exercises here.