Lambda Function

Premanand S

Assistant Professor
School of Electronics and Engineering
Vellore Institute of Technology
Chennai Campus

premanand.s@vit.ac.in

November 8, 2024

Definition and Concept of Lambda Functions

- A **lambda function** is a small anonymous function defined with the `lambda` keyword.
- It can take any number of arguments, but it can only have a single expression.
- The result of the expression is automatically returned.
- Syntax:

```
lambda arguments:  expression
```

- Example:

```
add = lambda x, y:  x + y
```

- You can call the function as:

```
result = add(3, 5)   Output:  8
```

Why Lambda Functions Are Used in Python

- **Conciseness:** Lambda functions are written in a single line, making the code more compact and easier to read.
- **Anonymous Functions:** Lambda functions do not need a name and are used for short-term or one-time use cases.
- **Functional Programming:** Useful in functional programming tasks where functions are passed as arguments (e.g., `map`, `filter`, `reduce`).
- **Readability in Short Tasks:** Lambda functions enhance readability when the logic is simple enough.

Difference Between Lambda Functions and Regular Functions

- **Syntax:**
  - `lambda x, y: x + y` (Lambda Function)
  - `def add(x, y): return x + y` (Regular Function)
- **Return:**
  - Lambda: Automatically returns the result of the expression.
  - Regular Function: Requires an explicit `return` statement.
- **Name:**
  - Lambda: Anonymous by default (can be assigned to a variable).
  - Regular Function: Always has a name.
- **Complexity:**
  - Lambda: Limited to a single expression.
  - Regular Function: Can contain multiple statements.

Lambda Functions vs Regular Functions

- **Lambda Functions**: Anonymous, concise, single-expression functions.
- **Regular Functions**: Named, multi-expression, reusable functions defined with `def`.

Lambda vs Regular Function Example

- **Lambda Function:**

```
square = lambda x:  x * x

print(square(4))   Output:  16
```

- **Regular Function:**

```
def square(x):  return x * x

print(square(4))   Output:  16
```

Example of a Lambda Function

- A simple example of a lambda function that adds two numbers:

```
add = lambda x, y:  x + y
```

- Calling the function:

```
print(add(5, 10))
```
Output: 15

Simplifying Code for One-Time Use

- Lambda functions are useful when a simple operation is required that doesn't need a full function definition.
- They help in simplifying code and reducing verbosity.
- Example:

```
sum = lambda x, y:  x + y

print(sum(10, 5))   Output:  15
```

Lambda Functions in Higher-Order Functions

- Lambda functions are ideal when used with higher-order functions like `map()`, `filter()`, and `reduce()`.
- **map()**: Applies a function to all items in an input list.
- **filter()**: Filters elements based on a condition.
- **reduce()**: Reduces a sequence to a single value.

Example of map()

- `map()` applies a function to each item in a list.
- Example:

```
numbers = [1, 2, 3, 4]

result = map(lambda x:  x**2, numbers)

print(list(result))   Output:  [1, 4, 9, 16]
```

Example of map() - Celsius to Fahrenheit

- `map()` can be used to apply a function to each element in a list.
- Example:

```
celsius = [0, 10, 20, 30, 40]

fahrenheit = map(lambda c:  (c * 9/5) + 32, celsius)

print(list(fahrenheit))
```

filter() - Filter Elements Based on a Condition

- `filter()` filters elements in an iterable based on a condition.
- Example to filter even numbers:

Example (Python Code)

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers)
Output: [2, 4, 6]
```

Example of reduce()

- `reduce()` reduces a list to a single value by applying a function cumulatively.
- Example:

```
from functools import reduce

numbers = [1, 2, 3, 4]

result = reduce(lambda x, y:  x + y, numbers)

print(result)   Output:  10
```

Example of reduce() - Find Product of Numbers

- `reduce()` reduces a list to a single value by applying a rolling computation.
- Example:

Example (Python Code)

```
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)
Output: 24
```

Example of reduce() - Find Maximum Value

- Example to find the maximum value in a list using `reduce()`:

Example (Python Code)

```
numbers = [1, 5, 3, 8, 2]
max_value = reduce(lambda x, y: x if x > y else y, numbers)
print(max_value)
Output: 8
```

sorted() - Sorting with Custom Key Functions

- `sorted()` sorts an iterable with a custom sorting key.
- Example to sort a list of tuples by the second element:

Example (Python Code)

```
data = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)

Output: [(1, 'apple'), (3, 'banana'), (2, 'cherry')]
```

When to Prefer Lambda Functions

- Short, single-use functions.
- Functional programming with `map()`, `filter()`, `reduce()`.
- Simple, one-liner expressions.

When to Prefer Regular Functions

- Complex logic requiring multiple expressions.
- Reusable functions that require naming and documentation.
- Functions with conditions, loops, or multiple return statements.

Advantages of Lambda Functions

- Concise syntax.
- Quick, anonymous function for one-time use.
- Ideal for functional programming (e.g., `map()`, `filter()`).

Limitations of Lambda Functions

- Limited to a single expression.
- Can harm readability if overused.
- Lack of documentation and names.

Syntax of Lambda Functions with Multiple Arguments

- `lambda arg1, arg2, ..., argN: expression`
- Accepts multiple arguments and returns the result of the expression.

Example: Lambda with Two Arguments

- `sum_two = lambda x, y:  x + y`
- `sum_two(5, 3)` $\Rightarrow$ `8`

Example: Lambda with Three Arguments

- `product_three = lambda a, b, c:  a * b * c`
- `product_three(2, 3, 4)` $\Rightarrow$ `24`

Example: Lambda with Four Arguments

- `weighted_avg = lambda w1, w2, w3, w4:  (w1 + w2 + w3 + w4) / 4`
- `weighted_avg(90, 85, 78, 92)` $\Rightarrow$ `86.25`

Lambda with Higher-Order Functions

- **map()**: `map(lambda x: x**2, [1, 2, 3, 4])`
- **filter()**: `filter(lambda x: x % 2 == 0, [1, 2, 3, 4])`
- **sorted()**: `sorted([(1, 'apple'), (3, 'banana')], key=lambda x: x[1])`

Sorting Lists of Tuples Using Lambda Functions

- Sort a list of tuples by the second element:
- `sorted_data = sorted(data, key=lambda x: x[1])`
- Example:
  - `data = [(1, 'apple'), (3, 'banana'), (2, 'cherry')]`
  - `sorted_data = sorted(data, key=lambda x: x[1])`
- Output: `[(1, 'apple'), (2, 'cherry'), (3, 'banana')]`

Sorting Lists of Dictionaries Using Lambda Functions

- Sort a list of dictionaries by a specific key:
- `sorted_students = sorted(students, key=lambda x: x['age'])`
- Example:
  - `students = 'name': 'Prem', 'age': 25, 'name': 'Anand', 'age': 22, 'name': 'Premanand', 'age': 23`
  - `sorted_students = sorted(students, key=lambda x: x['age'])`
- Output: `['name': 'Anand', 'age': 22, 'name': 'Premanand', 'age': 23, 'name': 'Prem', 'age': 25]`

Sorting by Multiple Keys

- Sort by multiple keys using a tuple:
- `sorted_students = sorted(students, key=lambda x: (x['age'], x['name']))`
- Output: `['name': 'Anand', 'age': 22, 'name': 'Premanand', 'age': 23, 'name': 'Prem', 'age': 25]`

Sorting Custom Objects Using Lambda Functions

- Sort a list of objects by an attribute:
- `sorted_people = sorted(people, key=lambda x: x.age)`
- Example:
  - `people = [Person('Alice', 25), Person('Bob', 22), Person('Charlie', 23)]`
  - `sorted_people = sorted(people, key=lambda x: x.age)`
- Output: `[Bob 22, Charlie 23, Alice 25]`

What is a Closure?

- A closure occurs when a function refers to variables from its surrounding scope.
- The inner function retains access to `x` even after `outer_function` finishes executing.

Example (Python Code)

```
def outer_function(x):
    def inner_function(y):
        return x+y
    return inner_function
```

Lambda Functions in Closures

- Lambda functions can be used inside closures to provide compact and dynamic functionality.
- Example:

Example (Python Code)

```
def multiplier(factor):
    return lambda x: x * factor
double = multiplier(2)
print(double(5))
Output: 10
```

Lambda Functions with Multiple Arguments in Closures

- Lambda functions can capture multiple variables and accept multiple arguments.
- Example:

Example (Python Code)

```
def create_adder(x):
    return lambda y, z: x + y + z
add_five = create_adder(5)
print(add_five(3, 2))
Output: 10
```

Lambda Functions and Functional Programming

- Lambda functions are essential tools in functional programming.
- They align with core principles such as:
  - Immutability
  - Higher-order functions
  - Declarative programming style

Lambda Functions and Immutability

- Lambda functions do not modify original data, preserving immutability.
- Example:
  - `numbers = [1, 2, 3, 4, 5]`
  - `squared_numbers = list(map(lambda x:  x**2, numbers))`
- The original `numbers` list remains unchanged.

Lambda Functions and Declarative Programming

- Lambda functions support a declarative approach.
- Example:
  - Imperative style: `for number in numbers: squared_numbers.append(number**2)`
  - Declarative style: `squared_numbers = list(map(lambda x: x**2, numbers))`