lambda, map and filter Functions in Python

Anab Batool Kazmi

Lambda function in Python

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.
- Unlike regular functions defined with the def keyword, lambda functions do not have a formal name. They are often used for short, simple operations.

Syntax

lambda arguments : expression

Lambda function in Python

Syntax

lambda arguments : expression

- The **lambda keyword** is used to declare an anonymous function.
- **Arguments:** After the lambda keyword, **you can list one or more arguments, separated by commas**. These arguments **are similar to the parameters** you would define in a regular function.
- A **colon (:):** A colon is used to **separate the argument list from the expression** that defines what the lambda function does.
- **Expression:** After the colon, you specify **a single expression that the lambda function will evaluate and return.** This expression i**s the operation or computation** you want the lambda function to perform. It can be as simple or as complex as needed.

Sum of two variables

```
# Lambda function with two arguments (x a
add = lambda x, y: x + y
result = add(5, 3)  # result will be 8
print(result)
```

```
8
```

```
print(type(add))
print(add)
```

```
<class 'function'>
<function <lambda> at 0x00000227ECD31620>
```

- a lambda function that takes **two arguments, x and y,** and **returns their sum.**
- **add is a variable that is assigned a lambda function**. This lambda function takes two arguments, x and y.
- The **expression x + y** calculates the sum of x and y.
- You then **call the lambda function by providing the values 5 and 3 as arguments**. It's equivalent to calling add(x=5, y=3).
- The lambda function computes the sum of the provided values, which is 5 + 3, resulting in 8.
- The result of the lambda function call is stored in the variable result.

A lambda function that squares a number

```
# A lambda function that squares a number:
square = lambda x: x * x
result = square(5)
print(result)
```

```
25
```

```
print(type(square))
print(type(result))
```

```
<class 'function'>
<class 'int'>
```

- square is a variable that is assigned a lambda function.
- This lambda function takes one argument x.
- The lambda function's expression, x * x, calculates the square of the input value x.

A lambda function that converts a string to upper case

```
# A lambda function that converts a string to all uppercase:
uppercase = lambda s: s.upper()
result = uppercase("hello world!")
print(result)
```

```
HELLO WORLD!
```

- uppercase is a variable that is assigned a lambda function.
- This lambda function takes one argument s, which is expected to be a string.
- The lambda function's expression, s.upper(), uses the upper() method to convert the string s to all uppercase letters.

Sort Tuple based on second element

```
#sorted(iterable, key=key, reverse=reverse)
data = [5,8,2,1]
print(sorted(data,reverse=True))
```

```
[8, 5, 2, 1]
```

```
data = [(3, 5), (1, 9), (8, 2)]
sorted_data = sorted(data)
print(sorted_data)
```

```
[(1, 9), (3, 5), (8, 2)]
```

```
#sort the list based on the second element
#(the element at index 1) of each tuple.
data = [(3, 5), (1, 9), (8, 2)]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)
```

```
[(8, 2), (3, 5), (1, 9)]
```

- In this case, a lambda **function lambda x: x[1]** is provided as the key function.
- This lambda function takes each tuple x from the data list and returns its second element, which is x[1]. This is the value used for sorting.
- As a result, the sorted() function sorts the data list based on the second element of each tuple. It reorders the tuples in ascending order according to their second elements.

map() function in Python

- The map() function in Python is a built-in function used for applying a specified function to each item in an iterable (e.g., a list, tuple, or other iterable data structure).
- It returns an iterator, which can be converted to other iterable types like lists or tuples.
- The purpose of the map() function is to transform data in a concise and efficient way without the need for explicit loops.

syntax

```
map(function, iterable)
```

map() function in Python

syntax

map(function, iterable)

- **function:** This is the function that you want to apply to each item in the iterable.
- **iterable:** This is the sequence or collection of items (e.g., a list) that you want to process.

Squaring a list of numbers using map()

- the map() function is used to apply the lambda function lambda x: x**2 to each element in the numbers list. This lambda function squares each number in the list.
- map() returns an iterator, so squared is an iterator containing the squared values at this point.

```
#Squaring a list of numbers using map()
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(type(squared))
squared_list = list(squared)
print(squared_list)
```

```
<class 'map'>
[1, 4, 9, 16, 25]
```

Converting a list of strings to uppercase using map()

- The code first uses map() to apply the str.upper method to each string, then converts the iterator to a list to display the uppercase strings, which are ['APPLE', 'BANANA', 'CHERRY'].

```
#Converting a list of strings to uppercase using map()
words = ["apple", "banana", "cherry"]
uppercase_words = map(str.upper, words)
print(type(uppercase_words))
uppercase_list = list(uppercase_words)
print(uppercase_list)
```

```
<class 'map'>
['APPLE', 'BANANA', 'CHERRY']
```

Doubling a list of numbers using map()

- the map() function is used to apply the double_value function to each element in the numbers list. This function doubles each value in the list.
- map() returns an iterator, so doubled_numbers is an iterator containing the doubled values at this point.

```python
#Mapping a custom function to a list of values
def double_value(x):
    return 2 * x

numbers = [1, 2, 3, 4, 5]
doubled_numbers = map(double_value, numbers)
doubled_list = list(doubled_numbers)
print(type(doubled_numbers))
print(doubled_list)
```

```
<class 'map'>
[2, 4, 6, 8, 10]
```

filter() function in Python

- The filter() function in Python is a built-in function used **for filtering elements from an iterable** (e.g., a list, tuple, or other iterable data structure) based on a specified condition.
- The filter() function **applies the specified function to each item in the iterable** and retains only those items for which the function returns True.
- **It returns an iterator** containing the elements that meet the condition.
- The filter() function takes two arguments: a function that defines the condition and the iterable to be filtered.

Syntax

filter(function, iterable)

filter() function in Python

Syntax

filter(function, iterable)

- **function:** This is the function that defines the condition**. It should return True for the elements to be retained and False for elements to be filtered out.**
- **iterable:** This is the sequence or collection of items that you want to filter.

Filtering even numbers from a list using filter()

- filter() function is used to apply the lambda function lambda x: x % 2 == 0 to each element in the numbers list. This lambda function checks whether each number is even or not.
- filter() returns an iterator, so filtered_even_numbers is an iterator containing the elements that meet the condition at this point.

```python
#Filtering even numbers from a list using filter()
numbers = [1, 2, 3, 4, 5, 6]
filtered_even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(type(filtered_even_numbers))
even_numbers_list = list(filtered_even_numbers)
print(even_numbers_list)  # Output: [2, 4, 6]
```

```
<class 'filter'>
[2, 4, 6]
```

Filtering strings longer than 5 characters

- the filter() function is used to apply the lambda function lambda x: len(x) > 5 to each element in the words list. This lambda function checks whether the length of each string is greater than 5 characters.
- filter() returns an iterator, so filtered_long_words is an iterator containing the strings that meet the condition at this point.

```python
#Filtering strings longer than 5 characters
words = ["apple", "banana", "cherry", "date", "fig"]
filtered_long_words = filter(lambda x: len(x) > 5, words)
print(type(filtered_long_words))
long_words_list = list(filtered_long_words)
print(long_words_list)  # Output: ['banana', 'cherry']
```

```
<class 'filter'>
['banana', 'cherry']
```

Filtering positive values from a list

- the filter() function is used to apply the lambda function lambda x: x > 0 to each element in the values list. This lambda function checks whether each value is greater than 0 (i.e., positive).
- filter() returns an iterator, so filtered_positive_values is an iterator containing the positive values from the original list at this point.

```
#Filtering positive values from a list
values = [-2, 0, 3, -5, 7, -9]
filtered_positive_values = filter(lambda x: x > 0, values)
print(type(filtered_positive_values))
positive_values_list = list(filtered_positive_values)
print(positive_values_list)  # Output: [3, 7]
```

```
<class 'filter'>
[3, 7]
```

*args and **kwargs in python

- In Python, *args and **kwargs are special syntax **used to pass a variable number of positional and keyword arguments to functions.**
- They provide flexibility when defining functions that can accept a varying number of arguments.
- *args and **kwargs are often used when you want to **create flexible and generic functions that can handle different numbers of arguments**.
- They are particularly useful in cases **where the number of arguments or their names may vary**.

*args (Arbitrary Positional Arguments)

- The *args syntax is used in function definitions **to allow a function to accept a variable number of positional arguments.**
- It collects **all the additional positional arguments into a tuple**.
- You **can name it anything you like, but *args is a common convention**.

```python
#*args (Arbitrary Positional Arguments)
def example_function(arg1, *args):
    print("arg1:", arg1)
    print("args:", args)

example_function(1, 2, 3, 4)
```

```
arg1: 1
args: (2, 3, 4)
```

Sum of Arbitrary Numbers

```python
#Sum of Arbitrary Numbers:
def sum_numbers(*args):
    print(type(args))
    result = 0
    for num in args:
        result += num
    return result

total = sum_numbers(1, 2, 3, 4, 5)
print(total)  # Output: 15
```

```
<class 'tuple'>
15
```

Arbitrary List Concatenation

```python
#Arbitrary List Concatenation:

def concatenate_lists(*args):
    result = []
    for lst in args:
        result.extend(lst)
    return result

list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = [7, 8, 9]
combined_list = concatenate_lists(list1, list2, list3)
print(combined_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Arbitrary Number of String Concatenation:

```python
#Arbitrary Number of String Concatenation:
def concatenate_strings(*args):
    return " ".join(args)

result = concatenate_strings("Hello", "world", "in", "Python!")
print(result)
```

```
Hello world in Python!
```

**kwargs (Arbitrary Keyword Arguments):

- The **kwargs syntax allows **a function to accept a variable number of keyword arguments.**
- It collects all the **additional keyword arguments into a dictionary**, where **the keys are the argument names** and the **values are the associated values.**

```python
#**kwargs (Arbitrary Keyword Arguments):
def example_function(arg1, **kwargs):
    print("arg1:", arg1)
    print("kwargs:", kwargs)

example_function(1, name="Alice", age=30, city="New York")
```

```
arg1: 1
kwargs: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

Displaying Information About a Person

```python
#Displaying Information About a Person
def display_person_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_person_info(name="Alice", age=30, address="123 Main St")
```

```
name: Alice
age: 30
address: 123 Main St
```

Configuring a Logging Function

```python
#Configuring a Logging Function
def log_message(message, **kwargs):
    log_level = kwargs.get("log_level", "INFO")
    timestamp = kwargs.get("timestamp", "2023-10-17 12:00:00")
    print(f"[{timestamp}] [{log_level}] {message}")

log_message("An important message")
log_message("An error occurred", log_level="ERROR")
```

```
[2023-10-17 12:00:00] [INFO] An important message
[2023-10-17 12:00:00] [ERROR] An error occurred
```

Storing User Preferences

```python
#Storing User Preferences:
user_preferences = {}

def update_preferences(**kwargs):
    user_preferences.update(kwargs)

update_preferences(language="English", theme="Dark Mode")
update_preferences(font_size=16)

print(user_preferences)
```

```
{'language': 'English', 'theme': 'Dark Mode', 'font_size': 16}
```