Functions theory and practical

September 8, 2024

question 1 What is the difference between a function and a method in Python?

[15]:
```
# In Python, both functions and methods are used to perform specific tasks, but␣
 ↪they have some key differences>>
# Functions
# Independent>> Functions are blocks of code that are defined independently and␣
 ↪can be called by their name.
# Parameter>> They can accept parameters, which are passed explicitly.
# Return Values>> Functions may or may not return a value.
# No Class Association>> Functions are not associated with any class or object.
```

[1]:
```
def add(a, b):
    return a + b

result = add(5, 3)
print(result)
```

```
8
```

[14]:
```
# Methods
# Dependent>> Methods are similar to functions but are associated with an␣
 ↪object or class.
# Self Parameter>>They always include self as their first parameter, which␣
 ↪refers to the instance of the class.
# Class Association>>Methods can access and modify the data within the object␣
 ↪they belong to.
```

[3]:
```
class Calculator:
    def add(self, a, b):
        return a + b

calc = Calculator()
result = calc.add(5, 3)
print(result)
```

```
8
```

[ ]:
```

```

[ ]:

[ ]:

question 2 . Explain the concept of function arguments and parameters in Python.

[13]:
```
# When we define and call a Python function, the term parameter and argument is␣
 ↪used to pass information to the function.

# parameter>> It is the variable listed inside the parentheses in the function␣
 ↪definition.
# argument>>It is a value sent to the function when it is called. It is data on␣
 ↪which function performs some action and returns the result.
```

[16]:
```
##In this example, the function sum_marks() is defined with three parameters,␣
 ↪a, b, c, and print the sum of all three values of the arguments passed␣
 ↪during a function call.
```

[5]:
```
# a, b, c are arguments of the function
def my_sum(a, b, c):
    s = a + b + c
    return s

print('Total is:', my_sum(30, 40, 50))
```

```
Total is: 120
```

[ ]:

[ ]:

[ ]:

question 3 What are the different ways to define and call a function in Python?

[ ]:
```
# Basic Function Definition and Call>>>
# You can define a function using the def keyword, followed by the function␣
 ↪name and parentheses. To call the function, simply use its name followed by␣
 ↪parentheses.
```

[6]:
```
# Define a function
def greet():
    print("Hello, World!")

# Call the function
greet()
```

```
Hello, World!
```

[ ]:
```
# Function with Parameters>>>
# Functions can take parameters, which are specified within the parentheses.
```

[7]:
```
# Define a function with parameters
def greet(name):
    print(f"Hello, {name}!")

# Call the function with an argument
greet("Alice")
```

```
Hello, Alice!
```

[ ]:
```
# Function with Return Value>>>
# Functions can return a value using the return statement.
```

[8]:
```
# Define a function that returns a value
def add(a, b):
    return a + b

# Call the function and store the result
result = add(3, 5)
print(result)
```

```
8
```

[ ]:
```
# Default Parameter Values>>>
# You can provide default values for parameters, which are used if no argument␣
 ↪is provided
```

[9]:
```
# Define a function with default parameter values
def greet(name="World"):
    print(f"Hello, {name}!")

# Call the function without an argument
greet()  # Output: Hello, World!

# Call the function with an argument
greet("Alice")
```

```
Hello, World!
Hello, Alice!
```

[ ]:
```
# Keyword Arguments>>>
# When calling a function, you can specify arguments by name, allowing you to␣
 ↪pass them in any order.
```

ments by name, allowing you to pass them in any order.

[10]:
```
# Define a function with multiple parameters
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

# Call the function using keyword arguments
describe_pet(animal_type="dog", pet_name="Buddy")
describe_pet(pet_name="Whiskers", animal_type="cat")b
```

```
I have a dog named Buddy.
I have a cat named Whiskers.
```

[ ]:

[ ]:

[ ]:

question 4 What is the purpose of the `return` statement in a Python function?

The return statement in a Python function is used to exit the function and send a value back to the caller. This allows the function to pass back a specific result or data, which can then be used for further computation or processing in the calling code

[12]:
```
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 7)
print(result)
```

```
12
```

[ ]:

[ ]:

[ ]:

question 5 What are iterators in Python and how do they differ from iterables?

[18]:
```
# In Python, iterators and iterables are closely related but serve different␣
 ↪purposes:

# Iterables
# Definition: An iterable is any Python object capable of returning its␣
 ↪elements one at a time. Examples include lists, tuples, dictionaries, sets,␣
 ↪and strings.
# How it works: An iterable object has an __iter__() method that returns an␣
 ↪iterator.
```

```
# Usage: You can loop over an iterable using a for loop or other iteration␣
 ↪tools.


# Iterators
# Definition: An iterator is an object that represents a stream of data. It is␣
 ↪used to iterate over an iterable object.
# How it works: An iterator has a __next__() method that returns the next item␣
 ↪in the sequence. When there are no more items, it raises a StopIteration␣
 ↪exception.
# Creation: You can create an iterator from an iterable using the iter()␣
 ↪function.


# Key Differences
# Nature: All iterators are iterables, but not all iterables are iterators. For␣
 ↪example, a list is an iterable but not an iterator. However, you can obtain␣
 ↪an iterator from a list using iter(list).
# Methods: Iterables have an __iter__() method, while iterators have both␣
 ↪__iter__() and __next__() methods
```

[17]:
```
# Iterable example
my_list = [1, 2, 3]
for item in my_list:
    print(item)

# Creating an iterator from an iterable
my_iterator = iter(my_list)
print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
```

```
1
2
3
1
2
3
```

[ ]:

[ ]:

[ ]:

question 6 Explain the concept of generators in Python and how they are defined

Generators in Python are a special type of function that allow you to iterate over a sequence of

values without storing the entire sequence in memory. This makes them more memory-efficient, especially when dealing with large datasets or streams of data.

How Generators Work Generators are defined using the def keyword, just like regular functions, but they use the yield keyword instead of return to produce a series of values. When a generator function is called, it returns a generator object that can be iterated over.

[20]:
```
def simple_generator():
    yield 1
    yield 2
    yield 3

# Using the generator
for value in simple_generator():
    print(value)
    #In this example, the simple_generator function yields three values, one at␣
 ↪a time. Each time the yield statement is encountered,
    #  the function's state is saved, and the value is returned to the caller.␣
 ↪When the function is called again, it resumes execution
    # right after the last yield statement.
```

```
1
2
3
```

[ ]:

[ ]:

[ ]:

question 7 What are the advantages of using generators over regular functions?

[21]:
```
# Generators offer several advantages over regular functions, especially when␣
 ↪dealing with large data sets or streams of data. Here are some key benefits:

# Memory Efficiency: Generators yield items one at a time and only when␣
 ↪requested, rather than generating all items at once and storing them in␣
 ↪memory.
#  This is particularly useful when working with large datasets or when memory␣
 ↪resources are limited.

# Lazy Evaluation: Generators use lazy evaluation, meaning they produce items␣
 ↪on-the-fly rather than computing all items upfront. This can lead to␣
 ↪significant
#  performance improvements and reduced computation time, especially if not all␣
 ↪generated values are needed.
```

```
# State Preservation: Generators maintain their state between calls. This␣
 ↪allows you to resume execution from where you left off, without having to␣
 ↪re-compute
# previous values. This can simplify complex iterations and state management.

# Reduced Overhead: Because generators only compute values as needed, they␣
 ↪often have less overhead compared to regular functions that might need to␣
 ↪precompute
#  and store all results before returning them.

# Stream Processing: Generators are ideal for processing data streams where you␣
 ↪want to handle data as it arrives. This can be useful for tasks like reading
# lines from a large file or processing real-time data.

# Simpler Code: Using generators can simplify code that deals with iteration␣
 ↪and state management, reducing the need for explicit loop control variables␣
 ↪and temporary storage.

# Pipelining: Generators can be easily used in pipelines where each generator␣
 ↪feeds into the next. This chaining capability allows for efficient data␣
 ↪processing
# and transformation in a clean and readable manner.
```

[22]:
```
#regular function example
def get_numbers(n):
    numbers = []
    for i in range(n):
        numbers.append(i)
    return numbers

for num in get_numbers(10):
    print(num)
```

```
0
1
2
3
4
5
6
7
8
9
```

[24]:
```
#generator
def get_numbers(n):
    for i in range(n):
```

```
        yield i

for num in get_numbers(10):
    print(num)
```

```
0
1
2
3
4
5
6
7
8
9
```

question 8 What is a lambda function in Python and when is it typically used?

A lambda function in Python is a small, anonymous function defined with the lambda keyword. Unlike regular functions defined with the def keyword, lambda functions are typically used for short, throwaway functions that are needed temporarily.

[25]:
```
lambda arguments: expression
```

[25]:
```
<function __main__.<lambda>(arguments)>
```

[26]:
```
# Here's a breakdown of how it works:

# lambda: This keyword is used to indicate that a lambda function is being␣
 ↪defined.
# arguments: These are the parameters passed to the lambda function, similar to␣
 ↪parameters in a regular function.
# expression: This is a single expression that the lambda function computes and␣
 ↪returns.
```

[27]:
```
# Regular function
def add(x, y):
    return x + y

# Lambda function
add_lambda = lambda x, y: x + y

print(add(2, 3))
print(add_lambda(2, 3))
```

```
5
5
```

[28]:
```
# Typical Uses of Lambda Functions:

# Short-Term Functions: Lambda functions are often used when a function is␣
 ↪needed for a short period and doesn't require a name.
# They are particularly useful when a function is used as an argument to␣
 ↪another function.

# Functional Programming Constructs: Lambda functions are frequently used with␣
 ↪functions like map(), filter(), and sorted() where small functions
#  are needed for short-term operations.
```

[29]:
```
#map() Example: Applies a function to all items in an input list.
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

[30]:
```
#filter() Example: Filters items in a list based on a function.
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

```
[2, 4]
```

[31]:
```
#sorted() Example: Sorts items based on a custom key.
data = [('apple', 3), ('banana', 2), ('cherry', 1)]
sorted_data = sorted(data, key=lambda item: item[1])
print(sorted_data)  # Output: [('cherry', 1), ('banana', 2), ('apple', 3)]
```

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

[ ]:
```
# Inline Use: Lambda functions are often used when a function is needed only in␣
 ↪a specific context, such as in event handling or in a single-use scenario.

# Simple Operations: They are ideal for simple operations where defining a full␣
 ↪function might be overkill. If you need a quick function to perform a small
# task, lambda can be a concise solution
```

question 9 . Explain the purpose and usage of the `map()` function in Python.

[32]:
```
# The map() function in Python is a built-in function used to apply a given␣
 ↪function to each item of an iterable
#  (such as a list, tuple, or set) and return an iterable (specifically, a map␣
 ↪object) with the results.
# It's a convenient way to perform transformations or operations on all␣
 ↪elements of a collection without needing explicit loops.
```

```
# Purpose:

# The map() function is designed to:

# Apply a Function to Each Item: Transform each item in an iterable based on a␣
 ↪given function.
# Simplify Code: Reduce the need for explicit loops and make the code more␣
 ↪readable and concise.
# Enhance Functional Programming: Support functional programming paradigms by␣
 ↪allowing functions to be passed as arguments and applied across collections.

#map() Example: Applies a function to all items in an input list.
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)  # Output: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

question 10 What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

[33]:
```
# The map(), reduce(), and filter() functions in Python are all used for␣
 ↪functional programming, but they serve different purposes and operate in␣
 ↪distinct ways.
# Here's a breakdown of their differences:

# 1. map()
# Purpose: Apply a function to each item in an iterable and return an iterable␣
 ↪(map object) with the results.

# Usage:

# Function: Takes a function and one or more iterables as arguments.
# Return Value: A map object, which is an iterator, and can be converted to a␣
 ↪list, tuple, etc.

numbers = [1, 2, 3, 4]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))

# Characteristics:

# Transforms each element of the iterable based on the function provided.
# The function can be a named function or a lambda function.
# Can work with multiple iterables if the function accepts more than one␣
 ↪argument.
```

```
[1, 4, 9, 16]
```

[34]:
```
# 2. reduce()
# Purpose: Apply a binary function (a function that takes two arguments)␣
 ↪cumulatively to the items of an iterable, reducing the iterable to a single␣
 ↪accumulated result.

# Usage:

# Function: Takes a function and an iterable as arguments. The function must␣
 ↪accept two arguments.
# Return Value: A single value that is the result of applying the function␣
 ↪cumulatively to the iterable's items.

from functools import reduce

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

# Characteristics:

# Reduces an iterable to a single value by applying a function cumulatively.
# Requires importing from the functools module.
# The function should be associative to ensure correct results.
```

24

[35]:
```
# 3. filter()
# Purpose: Filter elements from an iterable based on a predicate function (a␣
 ↪function that returns True or False), returning only those elements that␣
 ↪satisfy the condition.

# Usage:

# Function: Takes a predicate function and an iterable as arguments.
# Return Value: A filter object, which is an iterator, and can be converted to␣
 ↪a list, tuple, etc.

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

# Characteristics:

# Filters elements based on the boolean outcome of the predicate function.
# The predicate function returns True or False for each element.
# Can be used to remove elements that do not satisfy a condition.
```

```
[2, 4]
```

Practical Questions

1. Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list.

[9]:
```
def even_sum(numbers):
    sum=0
    for i in numbers:
        if i%2==0:
            sum=sum+i
    return sum
even_sum([1,2,3,4,5,6,7,8])
```

[9]:
```
20
```

[ ]:

2. Create a Python function that accepts a string and returns the reverse of that string.

[11]:
```
def reverse_string(s):
    return s[::-1]
# Example usage
string1 = "Hello, World!"
reversed_string = reverse_string(string1)
print(reversed_string)
```

```
!dlroW ,olleH
```

[ ]:

3. Implement a Python function that takes a list of integers and returns a new list containing the squares of each number.

[15]:
```
l = [1,2,3,4,5,6,7]
def square(l):
    sq = []
    for i in l:
        sq.append(i**2)
    return sq
square(l)
```

[15]:
```
[1, 4, 9, 16, 25, 36, 49]
```

[ ]:

4. Write a Python function that checks if a given number is prime or not from 1 to 200

[23]:
```python
def check_prime(num):

    if num <= 1:
        print("Not Prime")
        return

    is_prime = True

    for i in range(2, num):
        if num % i == 0:
            is_prime = False
            break

    if is_prime:
        print("Prime number")
    else:
        print("Not Prime")

num = 10
check_prime(num)

num = 13
check_prime(num)
```

```
Not Prime
Prime number
```

5. Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms

[25]:
```python
class FibonacciIterator:
    def __init__(self, num_terms):

        self.num_terms = num_terms
        self.current = 0
        self.a, self.b = 0, 1

    def __iter__(self):
        return self

    def __next__(self):

        if self.current >= self.num_terms:
            raise StopIteration


        if self.current == 0:
            result = self.a
```

```
        elif self.current == 1:
            result = self.b
        else:
            result = self.a + self.b
            self.a, self.b = self.b, result


        self.current += 1

        return result


num_terms = 10
fib_iterator = FibonacciIterator(num_terms)

for num in fib_iterator:
    print(num)
```

```
0
1
1
2
3
5
8
13
21
34
```

6. Write a generator function in Python that yields the powers of 2 up to a given exponent

[28]:
```
def powers_of_two(exponent):

    for i in range(exponent + 1):
        yield 2 ** i


max_exponent = 5
for power in powers_of_two(max_exponent):
    print(power)
```

```
1
2
4
8
16
32
```

7. Implement a generator function that reads a file line by line and yields each line as a string.

[30]:
```
def read_lines(filename):

    try:
        with open(filename, 'r') as file:
            for line in file:
                yield line.rstrip('\n')
    except FileNotFoundError:
        print(f"Error: The file {filename} was not found.")
    except IOError as e:
        print(f"Error: An IOError occurred. Details: {e}")

filename = 'example.txt'
for line in read_lines(filename):
    print(line)
```

```
Error: The file example.txt was not found.
```

8. Use a lambda function in Python to sort a list of tuples based on the second element of each tuple

[32]:
```
tuples_list = [(1, 5), (2, 3), (3, 4), (4, 1)]
sorted_list = sorted(tuples_list, key=lambda x: x[1])
print(sorted_list)
```

```
[(4, 1), (2, 3), (3, 4), (1, 5)]
```

9. Write a Python program that uses map() to convert a list of temperatures from Celsius to Fahrenheit

[34]:
```
celsius_temps = [0, 10, 20, 30, 40, 100]
def celsius_to_fahrenheit(celsius):
    """Convert Celsius to Fahrenheit."""
    return (celsius * 9/5) + 32
fahrenheit_temps = list(map(celsius_to_fahrenheit, celsius_temps))
print(fahrenheit_temps)
```

```
[32.0, 50.0, 68.0, 86.0, 104.0, 212.0]
```

[ ]:
```

```

10. Create a Python program that uses filter() to remove all the vowels from a given string.

[36]:
```
def is_not_vowel(char):

    return char.lower() not in

# Given string
input_string = "Hello, World!"
```

```
# Use filter() to remove vowels
filtered_chars = filter(is_not_vowel, input_string)

# Convert the filter object to a string and print it
result_string = ''.join(filtered_chars)
print(result_string)
```

```
Hll, Wrld!
```

[38]:
```
def calculate_totals(orders):

    return list(map(lambda order: (
        order[0],  # order number
        max(order[1] * order[2] + 10, order[1] * order[2]) if order[1] *␣
 ↪order[2] < 100 else order[1] * order[2]
    ), orders))

# Example usage:
orders = [
    [1, 20.0, 3],   # 1st order
    [2, 5.0, 10],   # 2nd order
    [3, 50.0, 2],   # 3rd order
    [4, 15.0, 8]    # 4th order
]

results = calculate_totals(orders)
print(results)
```

```
[(1, 70.0), (2, 60.0), (3, 100.0), (4, 120.0)]
```

[ ]:
```

```