Function_Assignment3

August 31, 2024

[1]:
```
#Q1 What is the difference between a function and a method in Python?
#Ans Function
#Standalone: Exist independently of classes.
#Called directly: Invoked by their name.
#No implicit arguments: Don't have an automatic self parameter.
#Purpose: Perform general-purpose tasks that can be used across different parts␣
 ↪of a program.
# Example
def greet(name):
  print("Hello,", name)

greet("Alice")
```

```
Hello, Alice
```

[2]:
```
# Methods
#Bound to classes: Defined within a class.
#Called on objects: Invoked using dot notation on an object of the class.
#Implicit self argument: The first parameter is always self, referring to the␣
 ↪object itself.
#Purpose: Perform actions specific to the class and its instances.
# Example
class Dog:
  def bark(self):
    print("Woof!")

my_dog = Dog()
my_dog.bark()
```

```
Woof!
```

[4]:
```
#Q2 Explain the concept of function arguments and parameters in Python.
# Parameters
# Parameters are the variables defined within the parentheses of a function␣
 ↪definition.
#They act as placeholders for the values that will be passed to the function␣
 ↪when it's called.
```

```
# Arguments
#Arguments are the actual values passed to a function when it's invoked.
#These values are assigned to the corresponding parameters in the function␣
 ↪definition.
# Example
def greet(name):  # 'name' is a parameter
    print("Hello,", name)

greet("Alice")  # "Alice" is an argument
```

```
Hello, Alice
```

[13]:
```
#Q3 What are the different ways to define and call a function in Python?
# To define a function in Python, you use the def keyword followed by the␣
 ↪function name, parentheses for parameters, and a colon.
#The function body is indented.
#def function_name(parameters):
  # Function body
  # Statements to be executed
# To call a function, you use its name followed by parentheses. If the function␣
 ↪requires arguments, you pass them within the parentheses.
#function_name(arguments)
#Examples
def greet(name):
    print("Hello,", name)
greet("Alice")
```

```
Hello, Alice
```

[15]:
```
#Different Ways to Define and Call Functions
#1Functions with No Parameters
def greet():
    print("Hello, world!")
greet()
```

```
Hello, world!
```

[16]:
```
#2 Functions with Parameters
def add(x, y):
  return x + y

result = add(3, 4)  # Calling the function with arguments
print(result)
```

```
7
```

[17]:
```
#3 Functions with Default Parameters
def greet(name="World"):
  print("Hello,", name)

greet()  # Uses the default parameter
greet("Alice")  # Overrides the default parameter
```

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

[18]:
```
#4 Functions with Variable-Length Arguments
def my_function(*args, **kwargs):
  print("Positional arguments:", args)
  print("Keyword arguments:", kwargs)

my_function(1, 2, 3, a=4, b=5)
```

```
Positional arguments: (1, 2, 3)
Keyword arguments: {'a': 4, 'b': 5}
```

[19]:
```
#Q4 What is the purpose of the return statement in a Python function?
#The return statement in Python is used to:

#End the execution of a function: Once the return statement is encountered, the␣
 ↪function terminates immediately.
#Send a value back to the caller: The value following the return keyword is␣
 ↪returned to the code that called the function.
#This value can be used for further calculations, assignments, or other␣
 ↪operations.
#Example:
def add(x, y):
  result = x + y
  return result

sum = add(3, 4)
print(sum)
```

```
7
```

[1]:
```
#Q5 What are iterators in Python and how do they differ from iterables?
#Iterables
#An iterable is any object that can be iterated over. It's essentially a␣
 ↪container that holds a collection of elements.
#When you pass an iterable to the iter() function, it returns an iterator.

#Examples of iterables:
```

```
#Lists
#Tuples
#Strings
#Dictionaries
#Sets

#Iterators
#An iterator is an object that implements the iterator protocol. It has two␣
 ↪methods:

#__iter__(): Returns the iterator object itself.
#__next__(): Returns the next item in the sequence. If there are no more items,␣
 ↪it raises a StopIteration exception.
#Iterators are used internally by for loops to iterate over elements.

#Key differences:

#Every iterator is an iterable, but not every iterable is an iterator.
#Iterables represent a collection of data, while iterators provide a way to␣
 ↪access elements one by one.
#Iterables are created directly, while iterators are created from iterables␣
 ↪using the iter() function.
```

[3]:
```
#Q6 Explain the concept of generators in Python and how they are defined.
# Generators are a special type of function in Python that return an iterator.␣
 ↪Unlike regular functions that return a single value and then terminate,␣
 ↪generators can yield multiple values over time.
#This makes them extremely useful for creating iterators on the fly, especially␣
 ↪for large or infinite sequences.

#Defining a Generator
#To create a generator function, you use the yield keyword instead of return.
#The yield keyword pauses the function's execution and returns a value to the␣
 ↪caller.
#When the function is called again, it resumes execution from where it left off.
def my_generator():
    for i in range(3):
        yield i

# Create a generator object
generator_object = my_generator()

# Iterate over the generator
for value in generator_object:
    print(value)
```

```
#How Generators Work
#When you call a generator function, it returns a generator object.
#The generator object is an iterator that can be used in a for loop or with the␣
 ↪next() function.
#Each time next() is called, the generator function resumes execution until it␣
 ↪reaches the next yield statement.
#The value yielded is returned, and the function is paused again.
#If there are no more values to yield, a StopIteration exception is raised.
```

```
0
1
2
```

[4]:
```
#Q7 What are the advantages of using generators over regular functions?
#1Memory Efficiency
#Lazy evaluation: Generators produce values on the fly, as needed, rather than␣
 ↪creating and storing all values in memory upfront.
#This is especially beneficial when dealing with large datasets.
#Reduced memory footprint: By avoiding the creation of large intermediate data␣
 ↪structures, generators can significantly improve memory efficiency.
#2 Performance Optimization
#Faster iteration: In many cases, generators can outperform functions that␣
 ↪return lists or other data structures, especially when dealing with large␣
 ↪datasets or computationally expensive operations.
#Efficient resource utilization: Generators can be used to process data in␣
 ↪chunks, allowing for better resource management.
#3Infinite Sequences
#Handling infinite data streams: Generators can be used to create iterators␣
 ↪that produce an infinite sequence of values,
#which is not possible with regular functions.
#Processing large or unbounded datasets: Generators provide a way to handle␣
 ↪datasets that are too large to fit into memory.
#4Simplified Code
#Concise syntax: Generator expressions often provide a more concise way to␣
 ↪create iterators compared to traditional function implementations.
#Improved readability: The yield keyword can make code more readable in certain␣
 ↪situations.
```

[1]:
```
#Q8 What is a lambda function in Python and when is it typically used?
#A lambda function is a small, anonymous function defined using the lambda␣
 ↪keyword.
#It's a concise way to create functions that are intended for single-use or␣
 ↪short-lived operations.
#Syntax:
#lambda arguments: expression
#arguments: A comma-separated list of arguments.
```

```
#expression: The function body, which should evaluate to a single value.
#When to Use Lambda Function
#simple functions: When you need a function that can be expressed in a single␣
 ↪line.
#Functions as arguments: As arguments to higher-order functions like map,␣
 ↪filter, reduce, and sorted.
#Inline functions: When you need a function for a specific task within another␣
 ↪function.

#Common Use Cases
#Sorting with custom keys:
data = [(1, 'apple'), (3, 'banana'), (2, 'orange')]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)  # Output: [(3, 'banana'), (1, 'apple'), (2, 'orange')]
```

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

[3]:
```
#Q9 Explain the purpose and usage of the map() function in Python.
#The map() Function in Python
#Purpose:

#The map() function in Python applies a given function to each item of an␣
 ↪iterable (like a list, tuple, or string)
#and returns a new iterable with the results.

#Syntax:

#map(function, iterable, iterable2, ...)
#function: The function to apply to each item.
#iterable: The iterable whose elements are to be processed.
#iterable2, ...: Additional iterables (optional).

#Key points:
#The map() function is often used with lambda functions for concise expressions.
#The returned map object is an iterator, so it's usually converted to a list or␣
 ↪tuple for further processing.
#For more complex operations, list comprehensions might be more readable.

#When to use map():
#When you need to apply the same transformation to all elements of an iterable.
#When you want to create a new iterable with the transformed values.
#When you prefer a functional approach to problem-solving.
```

[5]:
```
#Q10 What is the difference between map(), reduce(), and filter() functions in␣
 ↪Python?
#map()
```

```
#Purpose: Applies a given function to each item of an iterable and returns a␣
 ↪new iterable with the results.
#Syntax: map(function, iterable, iterable2, ...)

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

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

[9]:
```
#filter()
#Purpose: Creates a new iterable with elements from the original iterable that␣
 ↪satisfy a given condition.
#Syntax: filter(function, iterable)

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

```
[2, 4]
```

[2]:
```
#reduce()
#Purpose: Applies a function to the first two elements of an iterable and then␣
 ↪applies the same function to the result and the next element, and so on,␣
 ↪until the iterable is reduced to a single value.
#Syntax: reduce(function, iterable, initializer)

#Example
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)
```

```
24
```

[1]:
```
#Q11 Using pen & Paper write the internal mechanism for sum operation using ␣
 ↪reduce function on this given
#Problem: Calculate the sum of the list [47, 11, 42, 13] using the reduce()␣
 ↪function.

#Steps:

#1Import reduce:

#from functools import reduce
```

```
#2Define the function:

#def add(x, y):
    #return x + y
#3Apply reduce:

#result = reduce(add, [47, 11, 42, 13])

#Internal Mechanism:

#reduce() iterates over the list from left to right.
#For each pair of elements:
#Applies the add function to the elements.
#The result becomes the first argument for the next iteration.
#Breakdown:

#Iteration 1:

#add(47, 11) is called.
#Result: 58

#Iteration 2:
#add(58, 42) is called (using the previous result).
#Result: 100

#Iteration 3:
#add(100, 13) is called (using the previous result)

#Result: 113
```

[ ]:
```
PRACTICAL QUESTION
```

[3]:
```
 #Practical Question

#Q1 Write a Python function that takes a list of numbers as input and returns␣
 ↪the sum of all even numbers in

#def sum_of_even_numbers(numbers):
#  """Calculates the sum of even numbers in a given list.

#  Args:
#    numbers: A list of numbers.

#  Returns:
#   The sum of all even numbers in the list.
#  """
```

```
#  even_sum = 0
#  for num in numbers:
#    if num % 2 == 0:
#      even_sum += num
#  return even_sum


#my_list = [1, 2, 3, 4, 5, 6, 7, 8]
#result = sum_of_even_numbers(my_list)
#print(result)
 # Output: 20
```

[4]:
```
#Q2 Create a Python function that accepts a string and returns the reverse of␣
 ↪that string

#```python
#def reverse_string(string):
#  """Reverses a given string.

#  Args:
#    string: The input string to be reversed.

#  Returns:
#    The reversed string.
#  """

#  reversed_string = ""
#  for char in string:
#    reversed_string = char + reversed_string
#  return reversed_string


#my_string = "hello"
#result = reverse_string(my_string)
#print(result)
 # Output: olleh
```

[6]:
```
#Q3  Implement a Python function that takes a list of integers and returns a␣
 ↪new list containing the squares of each number

#```python
#def square_list(numbers):
#  """Calculates the squares of numbers in a list.

#  Args:
#    numbers: A list of integers.

```

```
#  Returns:
#    A new list containing the squares of each number.
#  """

#  squared_list = []
#  for num in numbers:
#    squared_list.append(num * num)
#  return squared_list

#my_list = [1, 2, 3, 4, 5]
#result = square_list(my_list)
#print(result)  # Output: [1, 4, 9, 16, 25]
```

[9]:
```
#Q4 Write a Python function that checks if a given number is prime or not from 1 to 200.

#def is_prime(num):
#  """Checks if a given number is prime.

#  Args:
#    num: The number to check.

#  Returns:
#    True if the number is prime, False otherwise.
#  """

#  if num <= 1:
#    return False
#  if num <= 3:
#    return True
#  if num % 2 == 0 or num % 3 == 0:
#    return False

#  i = 5
#  while i * i <= num:
#    if num % i == 0 or num % (i + 2) == 0:
#      return False
#    i += 6

#  return True

# Example usage:
#for num in range(1, 201):
#  if is_prime(num):
#    print(num, "is prime")
```

[8]:
```
#Q5 Create an iterator class in Python that generates the Fibonacci sequence up␣
 ↪to a specified number of terms

#class FibonacciIterator:
#  """An iterator for generating the Fibonacci sequence."""

#  def __init__(self, max_terms):
#    """Initializes the Fibonacci iterator.

 #   Args:
 #     max_terms: The maximum number of terms to generate.
#    """
#   self.max_terms = max_terms
#    self.current_term = 0
#   self.next_term = 1

#  def __iter__(self):
#    return self

#  def __next__(self):
#    if self.current_term >= self.max_terms:
#      raise StopIteration

#    fib = self.current_term
#   self.current_term, self.next_term = self.next_term, self.current_term +␣
 ↪self.next_term
#   return fib

# Example usage:
#max_terms = 10
#fib_iterator = FibonacciIterator(max_terms)
#for num in fib_iterator:
#  print(num)
```

[10]:
```
#Q6 Write a generator function in Python that yields the powers of 2 up to a␣
 ↪given exponent

#def powers_of_two(exponent):
#  """Generates powers of 2 up to a given exponent.

#  Args:
#    exponent: The maximum exponent for the powers of 2.

#  Yields:
#    The next power of 2.
#  """
```

```
#  power = 1
#  for _ in range(exponent + 1):
#    yield power
#    power *= 2

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

[11]:
```
#Q7 Implement a generator function that reads a file line by line and yields␣
 ↪each line as a string

#def read_file_lines(filename):
#  """Reads a file line by line and yields each line as a string.

#  Args:
 #   filename: The name of the file to read.

#  Yields:
#    The next line from the file as a string.
#  """

#  with open(filename, 'r') as file:
#    for line in file:
#      yield line.strip()  # Remove trailing newline characters

## Example usage:
#file_path = 'your_file.txt'
#for line in read_file_lines(file_path):
#  print(line)
```

[12]:
```
#Q8 Use a lambda function in Python to sort a list of tuples based on the␣
 ↪second element of each tuple.

#my_list = [(3, 'apple'), (1, 'banana'), (2, 'orange')]

#sorted_list = sorted(my_list, key=lambda x: x[1])
#print(sorted_list)  # Output: [(1, 'banana'), (3, 'apple'), (2, 'orange')]
```

[13]:
```
#Q9. Write a Python program that uses map() to convert a list of temperatures␣
 ↪from Celsius to Fahrenheit.

#def celsius_to_fahrenheit(celsius):
#  """Converts Celsius to Fahrenheit.

#  Args:
```

```
#    celsius: Temperature in Celsius.

#  Returns:
#    Temperature in Fahrenheit.
#  """
#  return (celsius * 9/5) + 32

#temperatures_celsius = [0, 10, 20, 30]
#temperatures_fahrenheit = list(map(celsius_to_fahrenheit,␣
 ↪temperatures_celsius))
#print(temperatures_fahrenheit)  # Output: [32.0, 50.0, 68.0, 86.0]
```

[14]:
```
#Q10 Create a Python program that uses filter() to remove all the vowels from a␣
 ↪given string.

#def remove_vowels(char):
#  """Removes vowels from a character.

#  Args:
#    char: The character to check.

#  Returns:
#    The character if it's not a vowel, otherwise an empty string.
#  """
#  vowels = "aeiouAEIOU"
#  return "" if char in vowels else char

#string = "hello world"
#result = "".join(filter(remove_vowels, string))
#print(result)  # Output: hll wrld
```

[ ]:
```
#Q11 Imagine an accounting routine used in a book shop. It works on a list with␣
 ↪sublists,

#Write a Python program, which returns a list with 2-tuples. Each tuple␣
 ↪consists of the order number and the product of the price per item and the␣
 ↪quantity. The product should be increased by 10,- € if the value of the␣
 ↪order is smaller than 100,00 €.

#Write a Python program using lambda and map.

#def process_orders(orders):
#  """Processes orders and returns a list of tuples with order number and␣
 ↪adjusted price.

#  Args:
```

```
#    orders: A list of lists, where each inner list represents an order with
#            [order_number, book_title, author, quantity, price_per_item].

#  Returns:
#    A list of tuples, where each tuple contains the order number and the␣
 ↪adjusted price.
#  """

#  min_order_value = 100

#  def calculate_price(order):
#    order_number, _, _, quantity, price_per_item = order
#    total_price = quantity * price_per_item
#    adjusted_price = total_price if total_price >= min_order_value else␣
 ↪total_price + 10
#    return order_number, adjusted_price

#  return list(map(calculate_price, orders))

# Example usage:
#orders = [
#    [34587, "Learning Python, Mark Lutz", 4, 40.95],
#    [98762, "Programming Python, Mark Lutz", 5, 56.80],
#    [77226, "Head First Python, Paul Barry", 3, 32.95],
#    [88112, "Einführung in Python3, Bernd Klein", 3, 24.99]

#result = process_orders(orders)
#print(result)
```