SOFT40161_Lab04

October 25, 2024

Nottingham Trent University

Department of Computer Science

1 SOFT40161 - Introduction to Computer Programming: Lab 04

Note: Try to buid up the concepts! Don't just execute the codes without knowing it's meaning.

1.1 Functions, Modules, Argument Passing, Recursion and Exceptions!

Please visit the lecture materials first!

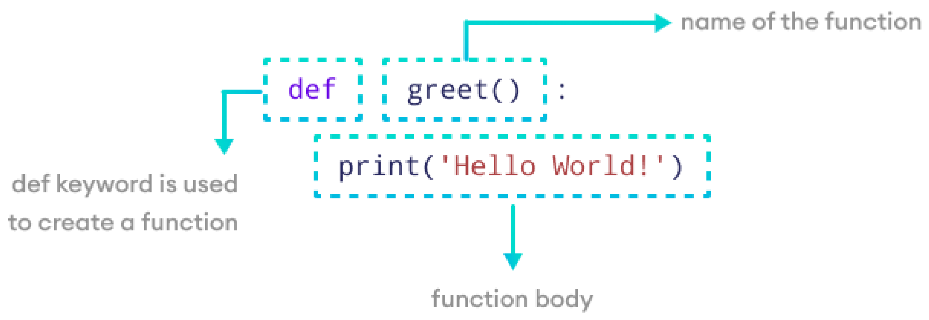
1.2 Lab Learning Outcomes

By the end of this lab, you will be able to: 1. **Remembering:** Identify and define functions, modules, arguments, recursion, and exceptions. 2. **Understanding:** Explain how to use functions, modules, and handle exceptions. 3. **Applying:** Implement functions, modules, recursion, and error handling in Python. 4. **Analyzing:** Differentiate between recursion and iteration for problem-solving. 5. **Evaluating:** Assess code solutions for efficiency and error handling. 6. **Creating:** Design programs integrating functions, modules, recursion, and exception handling.

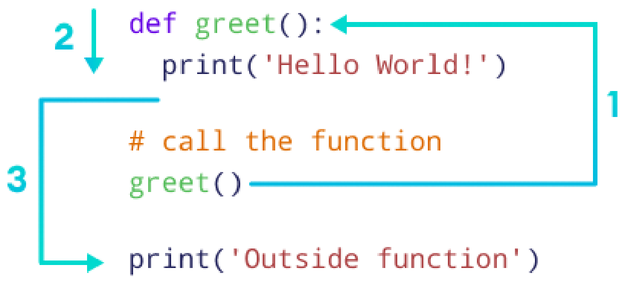
2 Example, Explanation and Exercise

2.1 Example 1: Basic Function in Python

[ ]:
```
def greet():
    print('Hello World!')
```



```
[ ]: greet()
     print ('Outside Function')
```



```
def greet():
  print('Hello World!')

# call the function
greet()

print('Outside function')
```

```
[ ]:
```

Explanation: - This is a simple function named greet. - The function takes no parameters; it just prints a greeting message when called. - To call the function, you simply use `greet()`.

This example introduces the concept of defining and calling a function, focusing on the basics of using def to declare a function in Python.

Exercise 1: Basic Function Example

Create a Python function called `display_message` that takes no arguments and prints a message about a hobby or interest. Inside the function: - Print a message like "I enjoy reading science fiction books."

At the end of your code, call the `display_message` function three times to repeat the message. Hints: - Use the def keyword to define the function. - The function should not have any parameters (keep the parentheses empty). - Include a `print()` statement inside the function to show your message.

Make sure to call the function three times to display the message.

Exercise 1: Solution

[ ]:
```

```

2.2 Example 2: Function with Positional Arguments

A Function with Positional Arguments in Python refers to a function where the arguments are passed in a specific order, and this order determines which values are assigned to the function's parameters. - In a function call, the values (arguments) are assigned to the function's parameters based on their position. - The first argument in the function call goes to the first parameter, the second to the second, and so on. The order of arguments is crucial because the parameters will receive the values in the sequence they are given.

[ ]:
```
def add_numbers(num1, num2):
    resultOfSum = num1 + num2
    return resultOfSum
```

[ ]:
```
# Call the function with positional arguments
result = add_numbers(5, 3)
print("Sum:", result)
```

Exercise 2: Function with Positional Arguments

Write a Python function called calculate_rectangle_area that takes two parameters: *length* and *width*. The function should return the area of a rectangle using the formula: $Area = length$ x $width$ Instructions: Use positional arguments to pass the values for length and width when calling the function. Call the function twice: - First, pass length as 5 and width as 10. - Second, reverse the order of arguments by passing length as 10 and width as 5.

Print the result for both function calls to see if the order affects the output.

Exercise 2: Solution

[ ]:
```

```

2.3 Example 3: Function with Default Arguments

A function with default arguments in Python allows you to set a default value for one or more parameters. This means if the caller does not provide a specific argument, the function will use the default value. It helps to make parameters optional while still allowing them to be overridden if needed.

[ ]:
```
def calculate_area(length, height=10):
    return length * height

# Call the function with and without the default parameter
area1 = calculate_area(5)
area2 = calculate_area(5, 15)

print("Area with default height:", area1)
print("Area with custom height:", area2)
```

Explanation:

This function, calculate_area, calculates the area of a rectangle. - It has two parameters: *length* and *height*. The *height* parameter has a default value of 10. - If *height* is not specified when calling the function, it defaults to 10. - In the example, $area1$ uses the default *height*, while $area2$ overrides it with 15.

This shows how to use default arguments, providing flexibility in function calls.

Exercise 3: Calculating Areas of various Shapes

Write a Python function that calculates the area of a rectangle or a circle. The function should use default arguments to specify the shape to calculate the area for.

Define a Function: Create a function named `calculate_area` that takes the following parameters: - *shape* (string, default: "rectangle"): the type of shape ("rectangle" or "circle"). - *length* (float, default: 1): the length of the rectangle or the radius of the circle. - *width* (float, default: 1): the width of the rectangle (not applicable for circles). The function should return the area based on the specified shape:

For a rectangle, the area is calculated as: $Area = length$ x $width$ For a circle, the area is calculated as: $Area = \pi$ x $radius^2$

Test the Function: - Call the $calculate_area $ function with different combinations of arguments to verify that the default values are applied correctly when not specified.

Output the Results: - Print the results of each function call to the console, including appropriate messages.

Exercise 3: Solution

[ ]:

```

```

2.4 Example 4: Function with variable Arguments

Variable Arguments: The $*args$ syntax enables a function to accept an arbitrary number of arguments. Instead of specifying each parameter explicitly, you can use $*args$ to collect all the additional positional arguments into a tuple.

When defining a function, you can include $*args$ as a parameter. This tells Python to pack any extra positional arguments into a tuple named args.

[ ]:

```python
def my_function(*args):
    for arg in args:
        print(arg)
```

[ ]:

```python
print("\n Output from the 1st function call:")
my_function(1, 2, 3)

print("\n Output from the 2nd function call:")
my_function()

print("\n  Output from the 3rd function call:")
```

```
my_function('apple', 'banana', 'orange', 'mango')
```

Key Points to Remember

**args* must be placed after any required positional parameters in the function definition. - You can name the parameter anything you like (not just `args`), but the * is essential. - If you need to accept both positional and keyword arguments, you can do so by combining `*args` with `**kwargs` in the function definition.

By using `*args`, you can create more flexible and reusable functions that can handle varying numbers of inputs.

Exercise 4: Creating a Custom Average Function with `*args`

Write a Python function that calculates the average of a variable number of numerical arguments using `*args`.

Define a Function: - Create a function named `calculate_average` that accepts any number of numerical arguments using `*args`. - The function should calculate and return the average of the provided numbers. If no numbers are provided, it should return 0.

Implementation Details: - If the number of arguments is greater than 0, compute the average by summing the arguments and dividing by the number of arguments. - Ensure your function handles cases where no arguments are provided gracefully.

Test the Function: - Call the `calculate_average` function with different sets of numbers to verify that it works correctly.

Print the results of each function call to the console.

Exercise 4: Solution

[ ]:
```

```

2.5 Example 5: Function with arbitrary number of keyword-value pairs using `**kwargs`

Write a Python function that takes a variable number of keyword arguments (using `**kwargs`) to display details about a product. This exercise will show how `**kwargs` can be used to accept an arbitrary number of keyword-value pairs.

[ ]:
```
def display_product_details(**kwargs):
    print("Product Details: \n")
    for key, value in kwargs.items():
        print(f"{key.capitalize()}: {value}")

# Test the function with different product details
display_product_details(name="Laptop", brand="Dell", price=850, warranty="2␣
 ↪years")

display_product_details(name="Smartphone", model="iPhone 16", price=1299)
display_product_details(name="iPad", manufacturingYear=2023, condition =␣
↪"Refurbished")
```

Function Definition with `**kwargs`:

`def display_product_details(**kwargs):` The `**kwargs` syntax allows the function to accept any number of keyword arguments, which are collected into a dictionary named kwargs. The keys of the dictionary are the names of the arguments (like name, brand, etc.), and the values are the corresponding values provided when the function is called.

Printing the Product Details:

`print("Product Details:") for key, value in kwargs.items(): print(f"{key.capitalize()}: {value}")`

The function first prints a header: “Product Details:”. It then iterates over the `kwargs` dictionary using a for loop. `kwargs.items()` returns a list of (key, value) pairs. - Each key is capitalized using `.capitalize()` to ensure a consistent display format. - The key and value are printed together in a formatted string.

Test the Function: `display_product_details(name="Laptop", brand="Dell", price=850, warranty="2 years")` - This function call passes four keyword arguments: name, brand, price, and warranty. - The function handles them dynamically and displays the details correctly.

Exercise: Building a User Profile Using `**kwargs`

Write a Python function that takes a variable number of keyword arguments using `**kwargs` to create a user profile (name, location, job, email, etc.). The function should allow users to provide any set of attributes (like name, age, location, job, etc.) and display the complete profile information.

[ ]:
```

```

2.5.1 Example: 6 Calculating Factorial Using a Recursive Function

Write a recursive function called factorial to compute the factorial of a given non-negative integer. A factorial of a number n (denoted as $n!$ is the product of all positive integers from 1 to n.

[ ]:
```
def factorial(n):
    # Base case: if n is 0 or 1, return 1
    if n == 0 or n == 1:
        return 1
    # Recursive case: multiply n by the factorial of (n-1)
    else:
        return n * factorial(n - 1)

# Test the function
print(factorial(5))  # Expected output: 120
print(factorial(3))  # Expected output: 6
print(factorial(0))  # Expected output: 1
```

Explanation: In this example, we use a recursive function called factorial to calculate the factorial of a given number n. A recursive function is a function that calls itself in its definition, allowing it to solve problems by breaking them down into smaller sub-problems.

** Function Definition with Base Case and Recursive Case: **

`def factorial(n):` - The function factorial accepts a single argument n, which is a non-negative integer.

Base Case:

`if n == 0 or n == 1: >    return 1`

The base case is a condition that stops the recursion. For the factorial calculation, if n is 0 or 1, the factorial is 1. This is because '0! = 1! = 1'. When n is 0 or 1, the function returns 1.

Recursive Case:

`else: >    return n * factorial(n - 1)`

- The recursive case is the condition under which the function calls itself to break the problem into smaller parts.
- The function calls itself with the value $n - 1$ and multiplies the result by n.
- This process repeats until the base case is reached, at which point the function stops calling itself and starts returning the accumulated results.

To compute factorial(5), the function proceeds as follows:

factorial(5) calls 5 * factorial(4) >factorial(4) calls 4 * factorial(3) »factorial(3) calls 3 * factorial(2) »>factorial(2) calls 2 * factorial(1) »»factorial(1) returns 1 (base case)

factorial(2) returns 2 * 1 $\rightarrow$ 2

factorial(3) returns 3 * 2 $\rightarrow$ 6

factorial(4) returns 4 * 6 $\rightarrow$ 24

factorial(5) returns 5 * 24 $\rightarrow$ 120

Key Points:

- Recursion: A technique in which a function calls itself to solve smaller instances of the same problem.
- Base Case: The stopping condition for the recursive calls, which prevents infinite recursion.
- Recursive Case: The part of the function where the recursion takes place, allowing the function to break down the problem.

This example provides a clear illustration of how recursion can be used to solve problems like factorial calculations, where the solution involves repeatedly reducing the problem size until a simple case is reached.

2.6 Error Handling Section

Division with Error Handling Task: Write a program that prompts the user for two numbers and divides them. Handle cases where the input is invalid or division by zero is attempted. Print a relevant error message for each exception.

```
[ ]: # Step 0: Build the function
def tryExceptFunction(num1,num2):

    try:
        # num1 = float(input("Enter the first number: "))
        # num2 = float(input("Enter the second number: "))

        # Step 2: Perform division
        result = num1 / num2
        print(f"Result: {num1} divided by {num2} is {result}")

    # Step 3: Handle division by zero
    except ZeroDivisionError:
        print("Error: Division by zero is not allowed.")

    # Step 4: Handle invalid input (e.g., entering a non-numeric value)
    except ValueError:
        print("Error: Please enter valid numbers.")

    # Step 5: Handle any other unexpected errors
    except Exception as e:
        print(f"An unexpected error occurred: {e}")


# Step 1: Input two numbers from the user
inpNum1 = float(input("Enter the first number: "))
inpNum2 = float(input("Enter the second number: "))

tryExceptFunction(inpNum1,inpNum2)
```

Explanation: - Step 1: The program prompts the user to input two numbers using input(). These are converted to floats using float() for division. - Step 2: The program attempts to divide the two numbers and prints the result. - Step 3: The ZeroDivisionError is handled with an error message if the second number is zero. - Step 4: The ValueError is handled to catch invalid inputs (e.g., if the user enters non-numeric values). - Step 5: A generic Exception is included to catch any other unexpected errors and print the error message.

This solution ensures that the program runs smoothly even when encountering common errors like invalid input or division by zero.

3 Challenge (H/W): Recursive Menu-Based Calculator with Error Handling

Create a Python program that serves as a menu-driven calculator supporting addition, subtraction, multiplication, and division. The calculator should continue to prompt the user until they choose to exit, using recursion to handle multiple calculations. Proper error handling must be included for invalid inputs and exceptions.

Challenge Requirements: Functions & Argument Passing: - Create individual functions for each operation: add, subtract, multiply, and divide. - Each function should take two numbers as arguments. - Implement an additional function called menu to display the calculator menu and guide user choices.

Recursion: - Implement a recursive function named calculator that displays the menu, performs the selected operation, and prompts the user for another calculation. - Use recursion to continue asking for user input until the user decides to exit.

Exception Handling: - Use try and except blocks to manage invalid number inputs, incorrect menu choices, and division by zero. - Provide clear error messages to guide the user to enter valid data.

3.1 Solution:

```
[ ]:
```