Python Chapter 5 Functions Activities

Contents

Python Chapter 5 Functions Activities 1
Recursion 1
Stack 2
Tutorial 5.1: Stack Example 2
Docstrings 4
Tutorial 5.2: Factorial Recursion 6
Input Validation with Boolean Functions 9
Assignment Submission 10

Time required: 90 minutes

Recursion

Please watch this video: Comparing Iterative and Recursive Factorial Functions

Recursion is a programming technique in which a function calls itself to solve a problem. It's a powerful and elegant way to solve certain types of problems, particularly those that can be broken down into smaller, similar subproblems. When using recursion, it's essential to understand how the call stack works, as it plays a crucial role in managing the recursive calls.

Recursion is a problem-solving technique that involves solving a problem by breaking it down into smaller, more manageable instances of the same problem. In a recursive function, the function calls itself with a modified input, working towards a base case where the recursion stops. Each recursive call contributes to solving a part of the problem, and the results of these calls are combined to obtain the final solution.

1. **Base Case:** A recursive function must have a base case or termination condition. This is the condition under which the recursion stops, preventing infinite recursion.
2. **Recursive Case:** In the recursive case, the function calls itself with a smaller or modified version of the problem. This step typically reduces the problem toward the base case.

3. **Inductive Step:** The recursive function combines the results of recursive calls to solve the overall problem.

Stack

Please watch the following video: [Stack vs Heap Memory – Simple Explanation]

When a function is called in Python (or most other programming languages), a portion of memory called the "call stack" is used to keep track of the function calls. Here's how the stack works in the context of recursion:

1. **Function Calls:** When a function is called, a new frame is created on the call stack. This frame contains information about the function's execution context, including its local variables and the return address (where the program should continue after the function call).
2. **Recursive Calls:** In a recursive function, when a new recursive call is made, a new frame is added to the stack. Each frame corresponds to a particular instance of the function call, with its own set of local variables.
3. **Stacking:** As recursive calls continue; new frames are stacked on top of previous ones. The stack grows vertically with each recursive call.
4. **Base Case:** When the base case is reached, the recursion starts to unwind. The frames are popped off the stack one by one, and the results are combined.
5. **Result Propagation:** The results of each recursive call are propagated upward through the call stack as the recursion unwinds, allowing the function to compute its final result.
6. **Completion:** Eventually, the entire stack is cleared, and the program returns the final result to the caller.

Tutorial 5.1: Stack Example

Complete the following example of a stack works.

```python
# Initialize an empty list to use as the stack
stack = []


Codiumate: Options | Test this function
def push(val):
    """Push a value onto the stack"""
    # Add the value to the end of the stack
    stack.append(val)


Codiumate: Options | Test this function
def pop():
    """Pop a value from the stack"""
    # Get the last value from the stack
    val = stack[-1]

    # Remove the last value from the stack
    del stack[-1]

    # Return the popped value
    return val


# Push values onto the stack
push(3)  # Push 3 onto the stack
push(2)  # Push 2 onto the stack
push(1)  # Push 1 onto the stack

print('Initial stack')
print(stack)

print('\nElements popped from stack:')
# Pop and print values from the stack
print(pop())  # Print the last value (1)
print(pop())  # Print the next last value (2)
print(pop())  # Print the next last value (3)

print('\nStack after elements are popped:')
print(stack)
```

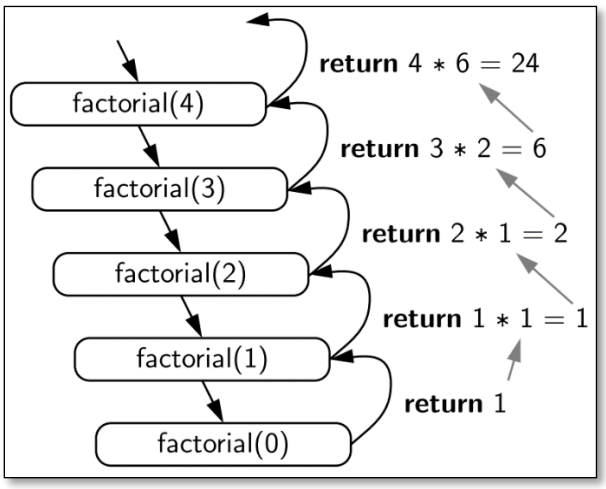
Example run:

```
Initial stack
[3, 2, 1]

Elements popped from stack:
1
2
3

Stack after elements are popped:
[]
```

Understanding the call stack is essential for managing recursion effectively. If you don't define a proper base case or termination condition, or if the recursion depth is too deep, you may encounter a stack overflow error, which occurs when the call stack becomes too large to handle.



Docstrings

A docstring in Python is a string literal that serves as documentation for a module, function, class, or method. It is placed immediately after the definition and is enclosed in triple quotes (either single or double). Docstrings are used to provide information about what the code does, how it should be used, and any relevant details.

1. **Documentation Purpose**: Docstrings are primarily used for documenting code to make it more understandable for developers, including yourself and others who might use or maintain the code.
2. **Location:** Docstrings are placed immediately after the definition. This makes them easily accessible when someone wants to learn about the code's purpose and usage.

3. **Triple Quotes:** Python allows docstrings to be enclosed in either triple single quotes (''') or triple double quotes (""""). Triple quotes are used to allow multi-line docstrings.
4. **Content:** Docstrings typically include information such as a brief description of the code's purpose, explanations of parameters and return values, usage examples, and any special considerations or notes.
5. **Access:** Docstrings can be accessed directly through many IDE's such as Visual Studio Code. Hover your mouse over the function call, the function definition and docstring appear. Many Python libraries use docstrings to in a similar fashion to help the programmer quickly understand how to use the function.

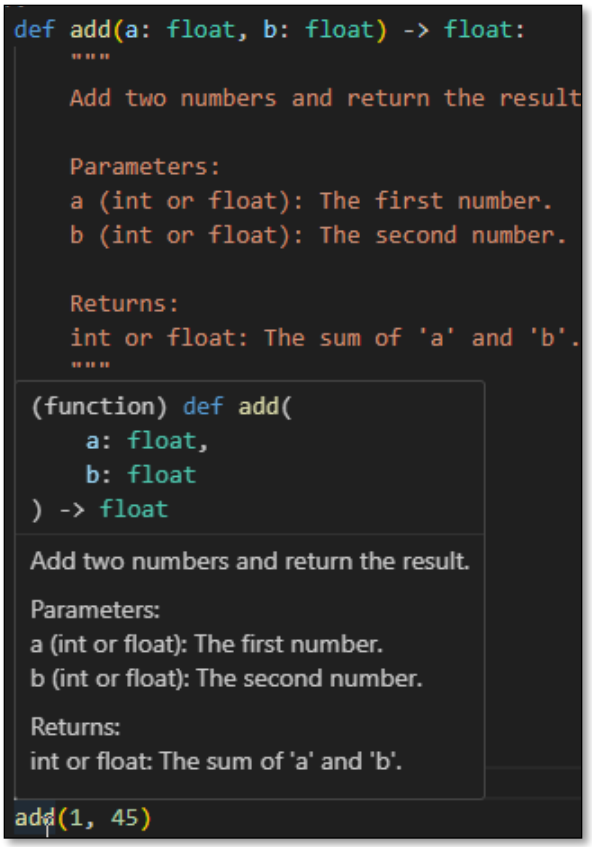
Docstring example:

```
def add(a, b):
    """
    Add two numbers and return the result.

    Parameters:
    a (int or float): The first number.
    b (int or float): The second number.

    Returns:
    int or float: The sum of 'a' and 'b'.
    """
    return a + b
add(1, 45)
```

In VSCode.



Tutorial 5.2: Factorial Recursion

Create a Python program named: **factorial_recursion.py**

This tutorial program includes a very comprehensive docstring.

```python
"""
 * Name: factorial_recursion.py
 * Written by:
 * Written on:
 * Purpose:
"""


def factorial(n: int, step=1) -> int:
    """Calculate the factorial of a non-negative integer 'n' using recursion.

    This function computes the factorial of 'n' by recursively multiplying
    'n' with the factorial of (n-1) until 'n' reaches 0, at which point it
    returns 1 as the base case.

    Parameters:
    - n (int): The non-negative integer for which to calculate the factorial.
    - step (int, optional): A variable to keep track of the iteration step
      (default is 1).

    Returns:
    - int: The factorial of 'n'.

    Example:
    >>> factorial(5)
    120
    >>> factorial(0)
    1
    >>> factorial(1)
    1
    >>> factorial(10)
    3628800
    """
```

```
    # Base case: When n reaches 0, return 1.
    if n == 0:
        return 1
    # Recursive case: When n is not 0, calculate the factorial by
    # multiplying n with the factorial of (n-1).
    else:
        # Print a message to indicate the calculation being performed.
        print(f"Calculating factorial({n}) * factorial({n-1})")

        # Make a recursive call to calculate the factorial
        # of (n-1) and increment the step counter.
        # Continue adding recursion to the stack until n == 0
        print(f"Add to call stack")
        result = n * factorial(n-1, step + 1)

        # After the function is done recursing
        # The function unwinds and pops each function call off the stack
        # Print the result of the steps
        print(f"Pop off call stack")
        print(f"Step {step}: factorial({n}) = {result}")

        # Return the result of the factorial calculation for the current n.
        return result


# Example usage:
number = int(input("Enter number for factorial: "))
result = factorial(number)
print(f"Final Result: {result}")
```

Example run:

```
Enter number for factorial: 5
Calculating factorial(5) * factorial(4)
Add to call stack
Calculating factorial(4) * factorial(3)
Add to call stack
Calculating factorial(3) * factorial(2)
Add to call stack
Calculating factorial(2) * factorial(1)
Add to call stack
Calculating factorial(1) * factorial(0)
Add to call stack
Pop off call stack
Step 5: factorial(1) = 1
Pop off call stack
Step 4: factorial(2) = 2
Pop off call stack
Step 3: factorial(3) = 6
Pop off call stack
Step 2: factorial(4) = 24
Pop off call stack
Step 1: factorial(5) = 120
Final Result: 120
```

Input Validation with Boolean Functions

The following code provides a Boolean return value. Instead of catching program exceptions, this function evaluates whether the input fits the range of the data. This function can be combined with the last input function.

```
def main():
    number = int(input("Please enter a positive number: "))
    # while the is_invalid function returns true
    # the loop keeps asking for valid input
    while is_invalid(number):
        print("Try again.")
        number = int(input("Please enter a positive number: "))

    print(f"Success! {number} is a positive number. ")

def is_invalid( number ):
    """While this function returns true, the data is invalid"""
    if number < 1:
        status = True
    else:
        status = False

    return status

main()
```

Example run:

```
Please enter a positive number: -1
Try again.
Please enter a positive number: 2
Success! 2 is a postive number.
```

Assignment Submission

1. Use pseudocode or TODO.
2. Comment your code to show evidence of understanding.
3. Attach the program files.
4. Attach screenshots showing the successful operation of the program.
5. Submit in Blackboard.