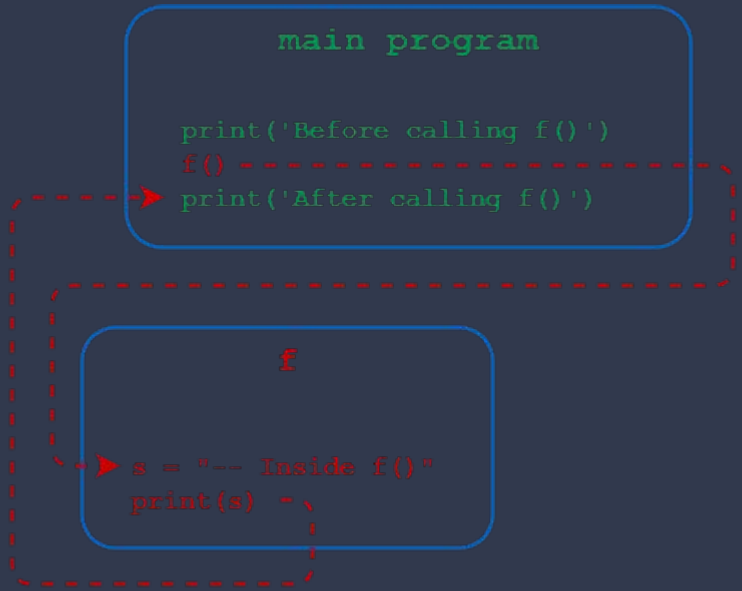
Python Functions

Functions

```
z = f(x,y)
```

A function is a relationship or mapping between one or more inputs and a set of outputs. In mathematics, a function is typically represented like this:



A function is a block of code which only runs when it is called.

A function is a named block of code designed to perform a specific task or calculation.

Benefit:

- Reusability
- Mitigate Redundancy
- Enhance Code organization

- Inbuilt functions
- Custom Functions

Function Parameters/arguments

Return

In built Functions

Built-in functions performs a specific task. The code that accomplishes the task is defined somewhere, but you don't need to know where or even how the code works. All you need to know about is the function's interface:

1. What arguments (if any) it takes
2. What values (if any) it returns

Then you call the function and pass the appropriate arguments. Program execution goes off to the designated body of code and does its useful thing. When the function is finished, execution returns to your code where it left off. The function may or may not return data for your code to use.

https://docs.python.org/3/library/functions.html

Custom Functions

Syntax: def function_name(parameters):

def keyword signifies the start of a function definition. Parameters are placeholders for values passed into the function.

The function body contains the actual instructions or operations that the function will execute. (Indented block of code under the function definition)

Return Statement:

Syntax: return result

Explanation: The return statement specifies the value the function produces as output. If no return statement is present, the function returns None by default.

Let’s Try

- ```
  def my_function():
    print(“I am inside a function”)
  ```
- ```
  def my_function(fname, lname):
    print(fname + " " + lname)
  ```
- ```
  def add_numbers(num1, num2):
          result = num1 + num2
          return result
  ```
  ```

Positional Vs Keyword Argument

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Positional Argument	Keyword Argument

def greet(name, age):
  # Function code
greet("Alice", 25)
	def greet(name, age):
	  # Function code
	greet(age=25, name="Alice")

Positional arguments are passed to a function in the order defined in the function signature. The most straightforward way to pass arguments to a Python function is with positional arguments (also called required arguments). In the function definition, you specify a comma-separated list of parameters inside the parentheses	Keyword arguments are passed with the parameter names explicitly specified. When you're calling a function, you can specify arguments in the form <keyword>=<value>. In that case, each <keyword> must match a parameter in the Python function definition.
• Order matters. • Values are matched by position	• Order doesn't matter. • Values are matched by parameter names.
Use positional arguments when the order of the values is essential.	Use keyword arguments when you want to make the code more readable or when dealing with functions with many parameters.

Best Practices

Best Practices:

- Meaningful Function Names:
  - Choose names that clearly convey the function's purpose.
- Commenting for Clarity:
  - Add comments to explain complex sections or the purpose of the function.
- Proper Use of Parameters:
  - Choose the right type and number of parameters for your function.

Assignment 4

1. Create a function called square that takes a number as an argument and returns its square. Call the function with a sample number and print the result.
2. Define a function calculate_area that takes two parameters, length and width, and returns the area of a rectangle. Call the function with sample values and print the result.
3. Define a function add_numbers that takes two numbers as parameters and returns their sum. Then, define another function calculate_square_sum that takes two numbers, calculates their sum using add_numbers, and returns the square of the sum.
4. Define a function is_even that takes a number as a parameter and returns True if it's even, and False if it's odd.

Recursive Function

Recursion is a programming concept where a function calls itself in its own definition.

How Recursive Functions Work:

- Base Case:
  - Every recursive function has a base case that defines when the function stops calling itself.
  - Without a base case, recursion would continue indefinitely.
- Recursive Step:
  - The function performs its task and calls itself with a modified input, moving closer to the base case.
- Benefit:
  - Solving problems in a more concise and elegant manner.

Recursive Function

Base Case:

- The base case is a critical component of a recursive function.
- It serves as the stopping condition, defining when the function should stop calling itself and start returning values.
- Without a base case, the recursive function would keep calling itself indefinitely, leading to what is known as infinite recursion.
- The base case provides a way to break the recursive chain and return a result, preventing the function from running infinitely.

Recursive Step:

- The recursive step is the part of the function where it performs its task and then calls itself with a modified input.
- After completing a portion of the task, the function invokes itself to work on a smaller or simpler version of the original problem.
- This step is crucial for making progress towards the base case, gradually reducing the problem's complexity.
- Each recursive call should bring the function closer to reaching the base case, ultimately allowing the function to produce a result.

Recursive Function

```
def print_pattern(n):
    if n > 0:
        print_pattern(n-1)  # Recursive call to print the pattern for n-1
        for i in range(1, n+1):
            print(i, end="")
        print()
print_pattern(5)
```

Recursive Functions

```
def sum_of_numbers(n):
    # Base case
    if n == 0:
        return 0
    else:
        # Recursive step
        return n + sum_of_numbers(n-1)
```

- Base Case:
  - If n is 0, the function returns 0.
- Recursive Step:
  - If n is greater than 0, the function calculates the sum of numbers from 1 to n by adding n to the result of calling itself with n-1.

Fibonacci Series

The Fibonacci series is a sequence of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1. In mathematical terms, the Fibonacci sequence is defined by the recurrence relation:

$$ F(n) = F(n-1) + F(n-2) $$

with initial conditions:

$$ F(0) = 0, \quad F(1) = 1 $$

So, the sequence starts: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, and so on. Each subsequent number in the sequence is the sum of the two preceding ones.

```
def fibonacci(n):
    # Base cases
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        # Recursive step
        return fibonacci(n-1) + fibonacci(n-2)
```

- Base Cases:
  - If n is 0, the function returns 0.
  - If n is 1, the function returns 1.
- Recursive Step:
  - If n is greater than 1, the function calculates the nth Fibonacci number by adding the results of calling itself with n-1 and n-2.

Exercised (Also Assignment 4)

1. Write a recursive function called factorial that takes an integer n and returns its factorial (n!). Call the function with a sample value.
2. Create a recursive function to find the sum of all numbers from 1 to a given positive integer n (HINT: Take user input).
3. Write a recursive function that prints a countdown from a given number n to 1.
4. Implement a recursive function to find the nth term in the Fibonacci sequence.
5. Write a recursive function to print the following pattern for a given positive integer k. k will be a user input, between 3 to 10.

   1

   12

   123

   1234

6. Extend the Fibonacci sequence to a Tribonacci sequence, where each term is the sum of the three preceding ones. Write a recursive function to find the nth term in the Tribonacci sequence.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

The base case is when n is 0 or 1, and the recursive step multiplies n by the result of calling factorial with n-1.

Recursive Step is : If n is greater than 1, the function multiplies n by the result of calling itself with n-1.

Recursive Functions

- Pros:
  - Concise and elegant solutions for certain problems.
  - Mimics the mathematical induction approach.
- Cons:
  - Can be less efficient in terms of space complexity due to the function call stack.
  - Requires careful consideration of the base case to avoid infinite recursion.

END