

INE-PYTHON

LECTURE - 05 FUNCTIONS AND MODULARITY

What You Will Learn

- Introduction to Functions
- Designing Program Using Functions
- Passing Arguments to Functions
- Local, Global Variables and Global Constants
- Value-Returning Functions
- Storing Functions in Modules

FUNCTIONS



Introduction to Functions

FUNCTIONS

This program is one long, complex sequence of statements.

In this program the task has been divided into smaller tasks, each of which is performed by a separate function.

```
def function1():
    statement      function
```

```
def function2():
    statement      function
```

```
def function3():
    statement      function
```

```
def function4():
    statement      function
```

FUNCTIONS

แนวคิดสำคัญเกี่ยวกับฟังก์ชัน:

- **การกำหนดฟังก์ชัน:** สร้างฟังก์ชันโดยใช้คำสำคัญ `def` ตามด้วยชื่อฟังก์ชันและพารามิเตอร์
- **พารามิเตอร์:** ตัวแปรที่ใช้รับข้อมูลเข้าในฟังก์ชัน กำหนดไว้ในวงเล็บของหัวฟังก์ชัน
- **ค่าที่ส่งกลับ:** ค่าที่ฟังก์ชันคืนให้ผู้เรียกโดยใช้คำสั่ง `return` ซึ่งจะสิ้นสุดการทำงานของฟังก์ชัน
- **พารามิเตอร์เริ่มต้น:** พารามิเตอร์ที่มีค่าเริ่มต้น เมื่อไม่ได้ส่งค่ามา
- **การเรียกใช้งานฟังก์ชัน:** ใช้ชื่อฟังก์ชันตามด้วยวงเล็บ ซึ่งอาจมีการส่งอาร์กิวเมนต์เข้าไป
- **แนวคิดการแบ่งโมดูล:** แบ่งโปรแกรมออกเป็นฟังก์ชันย่อยเพื่อเพิ่มประสิทธิภาพในการจัดการโค้ด และนำกลับมาใช้ซ้ำ
- **เอกสารประกอบ:** การเพิ่มคำอธิบายหรือ `docstring` เพื่ออธิบายจุดประสงค์ของฟังก์ชัน พารามิเตอร์ และค่าที่ส่งกลับ เพื่อเพิ่มความเข้าใจและง่ายต่อการดูแลโค้ด

FUNCTIONS

เราสามารถกำหนดฟังก์ชันโดยใช้คำสำคัญ `def` ตามด้วยชื่อฟังก์ชันและวงเล็บ `()` ซึ่งภายในวงเล็บอาจมีพารามิเตอร์ หรือไม่มีก็ได้ หัวฟังก์ชันจะตามด้วยเครื่องหมาย `:` และตามด้วยบล็อกของโค้ดที่เยื่องภายใต้ชื่อเป็นคำสั่งของฟังก์ชัน โครงสร้างนี้ใช้เพื่อกำหนดขอบเขตและวัตถุประสงค์ของฟังก์ชัน

ไวยากรณ์:

```
1 def function_name(parameters):  
2     # code block  
3     return value
```

Listing 5.1: Function Definition

FUNCTIONS

```
1 def greet():
2     print("Hello, World!")
3
4 # Calling the function
5 greet()
```

Listing 5.2: Defining a Simple Function

FUNCTIONS

Functions

- **ฟังก์ชัน (Function)** คือ ส่วนของ โค้ดที่สามารถนำกลับมาใช้ซ้ำได้ ซึ่งทำหน้าที่ดำเนินการบางอย่างที่กำหนดไว้
- ฟังก์ชันจะมีชื่อกำกับ และสามารถเรียกใช้งานได้โดยการใช้ชื่อนั้น
- นอกจากนี้ ฟังก์ชันสามารถรับค่าอาร์กิวเมนต์ (arguments) และสามารถส่งค่าผลลัพธ์กลับออกมาได้ (return value) ตามความจำเป็น

FUNCTIONS NAMES

กฎในการตั้งชื่อฟังก์ชันมีดังนี้:

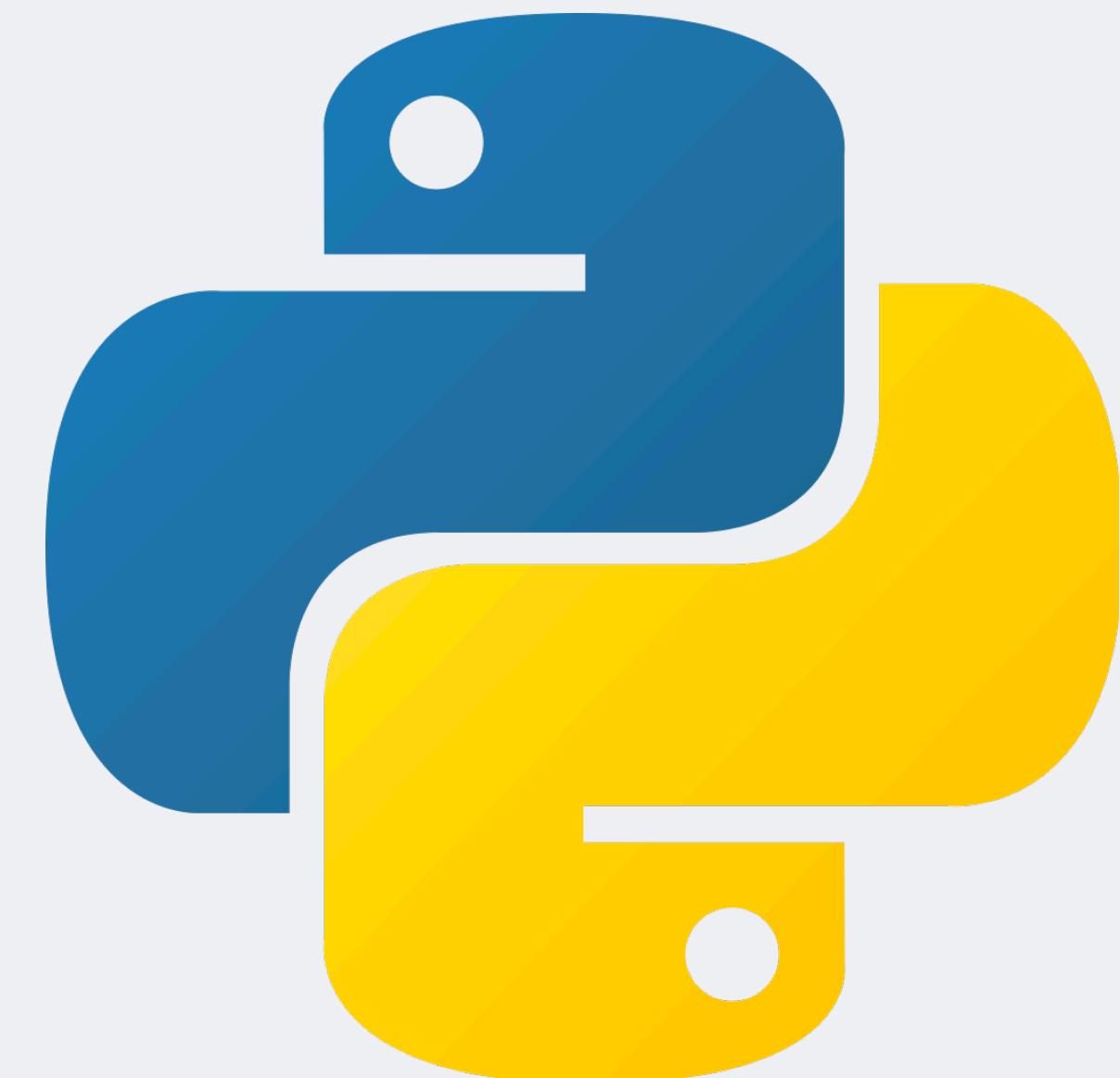
- ห้ามใช้คำส่วน (keywords) ของภาษาเป็นชื่อฟังก์ชัน
- ห้ามนิช่องว่างในชื่อฟังก์ชัน
- อักษรตัวแรกของชื่อฟังก์ชันต้องเป็นตัวอักษรหรือตัวขีดล่าง (_)
- อักษรตัดไปสามารถเป็นตัวอักษร ตัวเลข หรือขีดล่าง (_) ได้
- ตัวอักษรพิมพ์เล็กและพิมพ์ใหญ่ถือว่าแตกต่างกัน (เช่น Function กับ function ถือว่าเป็นคนละชื่อ)

BENEFITS OF MODULARISING A PROGRAM WITH FUNCTIONS

ข้อดีของการใช้ฟังก์ชันในการเขียนโปรแกรม:

- โค้ดที่เรียบง่ายขึ้น: โปรแกรมจะเข้าใจง่ายขึ้นเมื่อแบ่งออกเป็นฟังก์ชันย่อย ๆ
- ใช้ช้าได้: ฟังก์ชันที่เขียนไว้สามารถเรียกใช้ได้หลายครั้งโดยไม่ต้องเขียนซ้ำ
- ทดสอบได้ง่ายขึ้น: สามารถทดสอบแต่ละฟังก์ชันแยกกันได้ ทำให้การตรวจสอบและแก้ไขข้อผิดพลาดง่ายขึ้น
- พัฒนาได้รวดเร็วขึ้น: ฟังก์ชันที่เขียนไว้แล้วสามารถนำไปใช้ช้าในโปรแกรมอื่นได้ ช่วยประหยัดเวลา
- สนับสนุนการทำงานเป็นทีม: โปรแกรมเมอร์แต่ละคนสามารถรับผิดชอบการเขียนฟังก์ชันคนละส่วน ทำให้การทำงานร่วมกันมีประสิทธิภาพมากขึ้น

FUNCTIONS



Defining a Simple Function

FUNCTIONS

```
1 def message():
2     print('I am Arthur')
3     print('King of the Britons')
4
5 # Calling the function
6 print('I have a message for you.')
7 message()
8 print('Goodbye!')
```

Listing 5.3: Defining a Simple Function

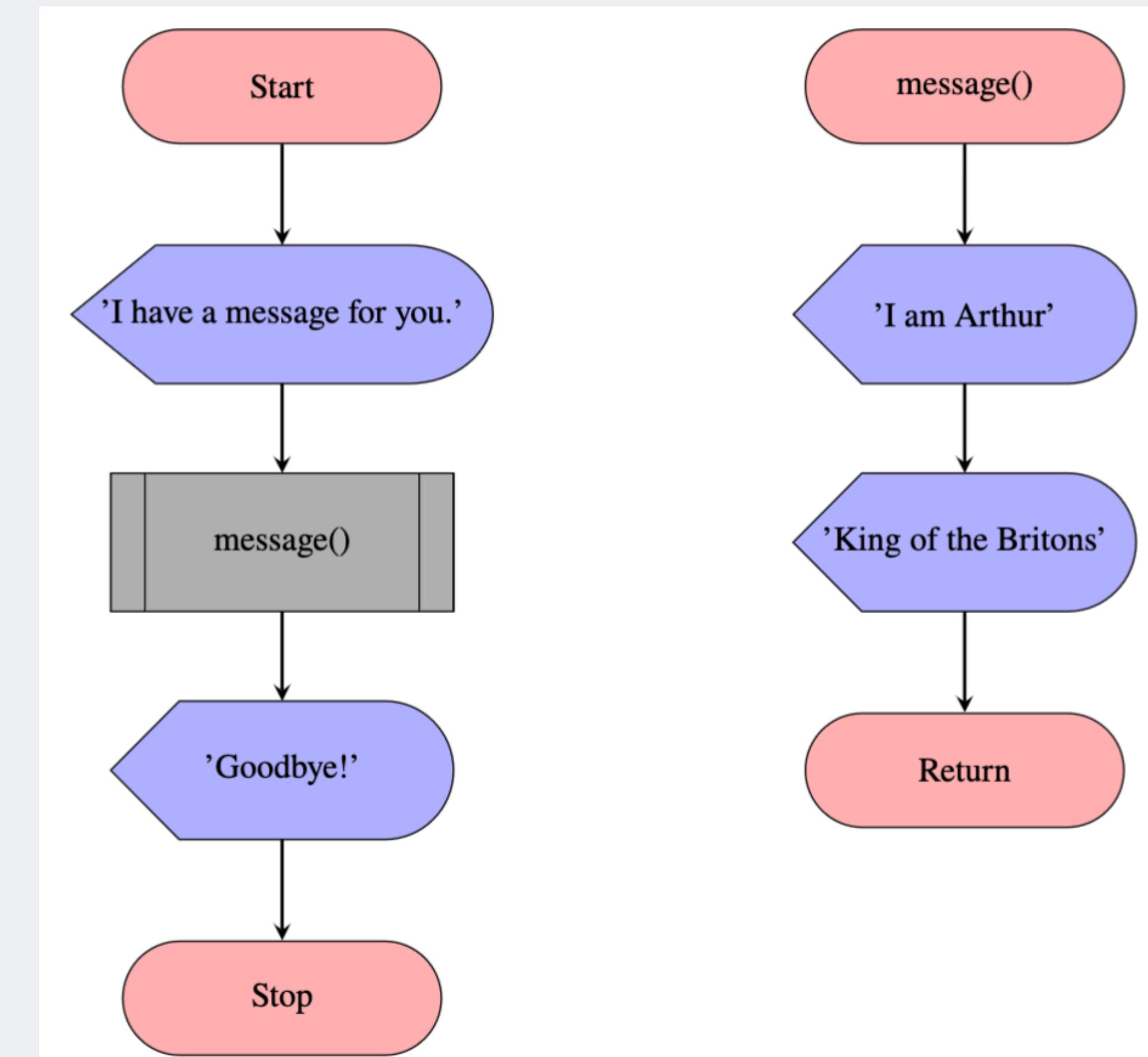
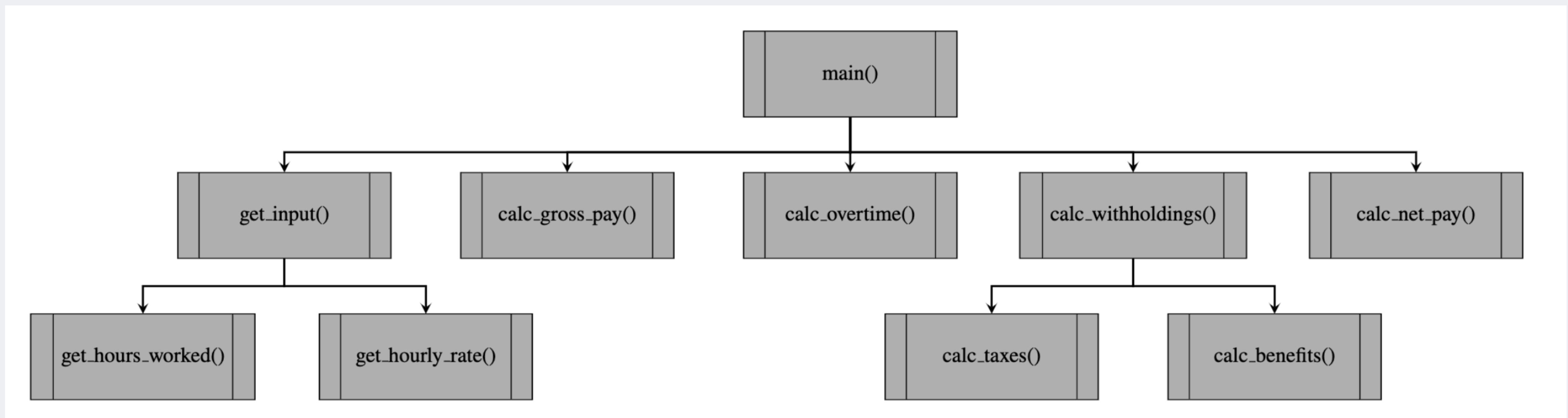


Fig. 5.1: Sample of Flowchart with Function()

FUNCTIONS

```
# This program has two functions.  
# First we define the main function .  
  
def main():  
    print('I have a message for you.')  
    message()  
    print('Goodbye!')  
  
#Next we define the message function  
def message():  
    print('I am Anirach,')  
    print('I Love Python.')  
  
# Call the main function.  
main()
```

FUNCTIONS



FUNCTIONS



Passing Parameter to Functions

FUNCTION PARAMETERS

ฟังก์ชันที่มีพารามิเตอร์สามารถรับค่าข้อมูลนำเข้าเพื่อใช้ในการประมวลผลตามค่าดังกล่าวได้ พารามิเตอร์จะถูกกำหนดไว้ภายในวงเล็บของหัวฟังก์ชัน เมื่อมีการเรียกใช้ฟังก์ชัน ผู้ใช้ต้องส่งอาร์กิวเมนต์ที่ตรงกับพารามิเตอร์นั้นเข้าไป ซึ่งช่วยให้ฟังก์ชันมีความยืดหยุ่นและสามารถทำงานได้หลากหลายตามข้อมูลนำเข้า

ตัวอย่าง:

```
1 def greet(name):
2     print(f"Hello, {name}!")
3
4 # Calling the function with an argument
5 greet("Alice")
```

Listing 5.4: Function with Parameters

FUNCTION WITH PARAMETERS

ฟังก์ชันที่มีพารามิเตอร์หลายตัวสามารถรับค่าข้อมูลนำเข้าหลายค่า ทำให้ฟังก์ชันสามารถทำงานได้อย่างยืดหยุ่นและมีประสิทธิภาพมากขึ้น พารามิเตอร์เหล่านี้จะถูกระบุในวงเล็บของฟังก์ชันโดยคั่นด้วยเครื่องหมายจุลภาค เมื่อเรียกใช้งานฟังก์ชัน ผู้ใช้ต้องส่งอาร์กิวเมนต์ให้ครบถ้วนตามจำนวนพารามิเตอร์ที่กำหนดไว้

ตัวอย่าง:

```
1 def add(a, b):  
2     return a + b  
3  
4 # Calling the function with arguments  
5 result = add(3, 5)  
6 print(result) # Output: 8
```

Listing 5.5: Function with Multiple Parameters

FUNCTION WITH PARAMETERS

ฟังก์ชันที่มีพารามิเตอร์ค่าเริ่มต้นจะกำหนดค่าที่ใช้โดยอัตโนมัติในกรณีที่ไม่มีการส่งอาร์กิวเมนต์สำหรับพารามิเตอร์นั้น ๆ การใช้พารามิเตอร์ค่าเริ่มต้นช่วยให้การเรียกใช้ฟังก์ชันง่ายขึ้นและสามารถปรับเปลี่ยนการทำงานของฟังก์ชันได้ยืดหยุ่นยิ่งขึ้น หมายสำหรับกรณีที่บางพารามิเตอร์ไม่จำเป็นต้องระบุทุกครั้ง

ตัวอย่าง:

```
1 def greet(name="World"):
2     print(f"Hello, {name}!")
3
4 # Calling the function without an argument
5 greet() # Output: Hello, World!
6
7 # Calling the function with an argument
8 greet("Alice") # Output: Hello, Alice!
```

Listing 5.6: Function with Default Parameters

FUNCTION WITH VARIABLE-LENGTH PARAMETERS

ภาษา Python อนุญาตให้คุณกำหนดฟังก์ชันที่สามารถรับจำนวนอาร์กิวเม้นต์ไม่จำกัดได้ โดยใช้อาร์กิวเม้นต์แบบลำดับด้วย `*args` และอาร์กิวเม้นต์แบบระบุชื่อด้วย `**kwargs`

ตัวอย่าง:

```
1 def sum_all(*args):  
2     return sum(args)  
3  
4 print(sum_all(1, 2, 3, 4, 5)) # Output: 15
```

Listing 5.7: Variable-Length Positional Arguments (*args)

FUNCTION DEFAULT PARAMETERS

```
1 def greet(name="World"):
2     print(f"Hello, {name}!")
3
4 # Calling the function without an argument
5 greet() # Output: Hello, World!
6
7 # Calling the function with an argument
8 greet("Alice") # Output: Hello, Alice!
```

Listing 5.5: Function with Default Parameters

FUNCTION WITH VARIABLE-LENGTH PARAMETERS

```
1 def sum_all(*args):  
2     return sum(args)  
3  
4 print(sum_all(1, 2, 3, 4, 5)) # Output: 15
```

Listing 5.7: Variable-Length Positional Arguments (*args)

FUNCTION WITH VARIABLE-LENGTH PARAMETERS

```
1 def find_max(*args):
2     if not args:
3         return None
4     max_value = args[0]
5     for number in args:
6         if number > max_value:
7             max_value = number
8     return max_value
9
10 # Example usage
11 result = find_max(3, 5, 7, 2, 8)
12 print(f"The maximum value is: {result}") # Output: The maximum value is: 8
```

Listing 5.8: Function to find maximum values

FUNCTION WITH VARIABLE-LENGTH PARAMETERS

```
1 def print_all(*args):
2     for index, arg in enumerate(args):
3         print(f"Argument {index + 1}: {arg}")
4
5 # Example usage
6 print_all("Python", 3.8, True, [1, 2, 3], {"key": "value"})
7 # Output:
8 # Argument 1: Python
9 # Argument 2: 3.8
10 # Argument 3: True
11 # Argument 4: [1, 2, 3]
12 # Argument 5: {'key': 'value'}
```

Listing 5.9: Multiple Type Variable-Length Positional Arguments (*args)

FUNCTION WITH VARIABLE-LENGTH PARAMETERS

```
1 def display_info(**kwargs):
2     for key, value in kwargs.items():
3         print(f"{key}: {value}")
4
5 display_info(name="Alice", age=30, city="New York")
6 # Output:
7 # name: Alice
8 # age: 30
9 # city: New York
```

Listing 5.10: Variable-Length Keyword Arguments (**kwargs)

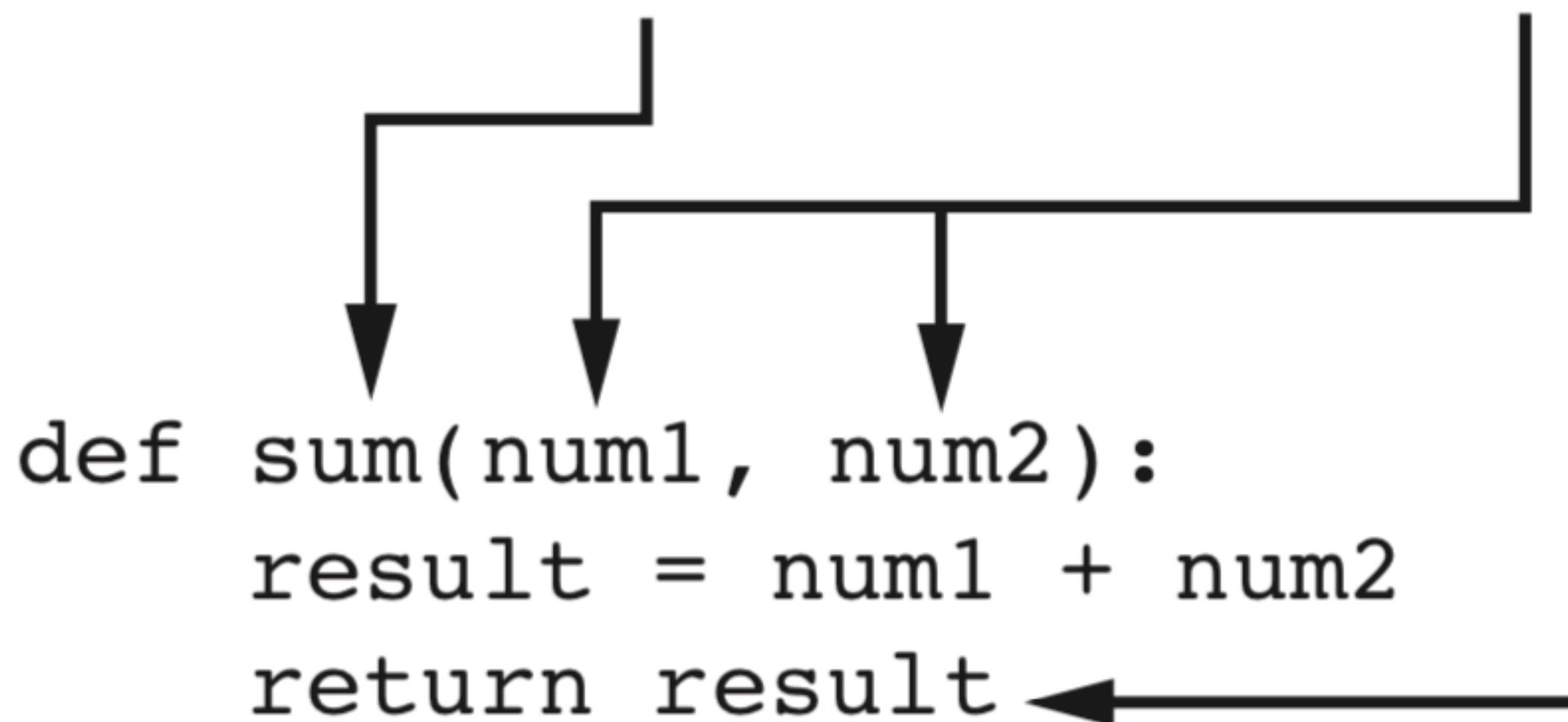
FUNCTIONS



Return Values Functions

VALUE-RETURNING FUNCTIONS

The name of this function is sum. num1 and num2 are parameters.



This function returns the value referenced by the result variable.

FUNCTION WITH RETURN VALUE

ฟังก์ชันที่มีการส่งค่ากลับจะใช้คำสั่ง `return` เพื่อส่งผลลัพธ์กลับไปยังผู้เรียกฟังก์ชัน ซึ่งช่วยให้ฟังก์ชันสามารถส่งค่าผลลัพธ์หลังจากทำงานเสร็จสิ้น และสามารถเก็บค่านั้นไว้ในตัวแปรหรือใช้ในกระบวนการอื่น ๆ ต่อไปได้ การใช้ค่าที่ส่งกลับจะเพิ่มความสามารถในการใช้งานของฟังก์ชันให้มีประสิทธิภาพมากขึ้น

ตัวอย่าง:

```
1 def multiply(a, b):
2     return a * b
3
4 # Calling the function and storing the returned value
5 result = multiply(4, 5)
6 print(result) # Output: 20
```

Listing 5.11: Function with Return Values

IPO AND FUNCTION

```
def discount(price):  
    return price * DISCOUNT_PERCENTAGE
```

IPO Chart for the <code>discount</code> Function		
Input	Processing	Output
An item's regular price	Calculates an item's discount by multiplying the regular price by the global constant <code>DISCOUNT_PERCENTAGE</code>	The item's discount

FUNCTION WITH RETURN MULTIPLE VALUE

ฟังก์ชัน `calculate_stats` ด้านล่างเป็นฟังก์ชันที่ใช้คำนวณและส่งค่ากลับหลายค่า ได้แก่ ผลรวม ค่าเฉลี่ย ค่าสูงสุด และค่าต่ำสุดของการตัวเลข ตัวอย่างการใช้งานแสดงให้เห็นถึงการเรียกใช้ฟังก์ชัน การแตกค่าที่ส่งกลับออกเป็นตัวแปร และการแสดงผลลัพธ์ ซึ่งช่วยให้สามารถคำนวณและเข้าถึงค่าทางสถิติต่าง ๆ ได้อย่างชัดเจนและมีประสิทธิภาพ

ตัวอย่าง:

```
1 def calculate_stats(numbers):
2     total_sum = sum(numbers)
3     average = total_sum / len(numbers)
4     maximum = max(numbers)
5     minimum = min(numbers)
6     return total_sum, average, maximum, minimum
7
8 # Example usage
9 numbers = [5, 10, 15, 20, 25]
10 total, avg, max_num, min_num = calculate_stats(numbers)
11
12 print(f"Total Sum: {total}")
13 print(f"Average: {avg}")
14 print(f"Maximum: {max_num}")
15 print(f"Minimum: {min_num}")
```

Listing 5.12: Function with Return Multiple Values

FUNCTION WITH RETURN VALUE

```
1 def calculate_stats(numbers):
2     total_sum = sum(numbers)
3     average = total_sum / len(numbers)
4     maximum = max(numbers)
5     minimum = min(numbers)
6     return total_sum, average, maximum, minimum
7
8 # Example usage
9 numbers = [5, 10, 15, 20, 25]
10 total, avg, max_num, min_num = calculate_stats(numbers)
11
12 print(f"Total Sum: {total}")
13 print(f"Average: {avg}")
14 print(f"Maximum: {max_num}")
15 print(f"Minimum: {min_num}")
```

Listing 5.7: Function with Return Multiple Values

EXERCISE-I (IS_ARMSTRONG)

```
# Example usage
print(is_armstrong(153)) # Output: True (153 = 1^3 + 5^3 + 3^3)
print(is_armstrong(9474)) # Output: True (9474 = 9^4 + 4^4 + 7^4 + 4^4)
print(is_armstrong(123)) # Output: False (123 is not an Armstrong number)
```

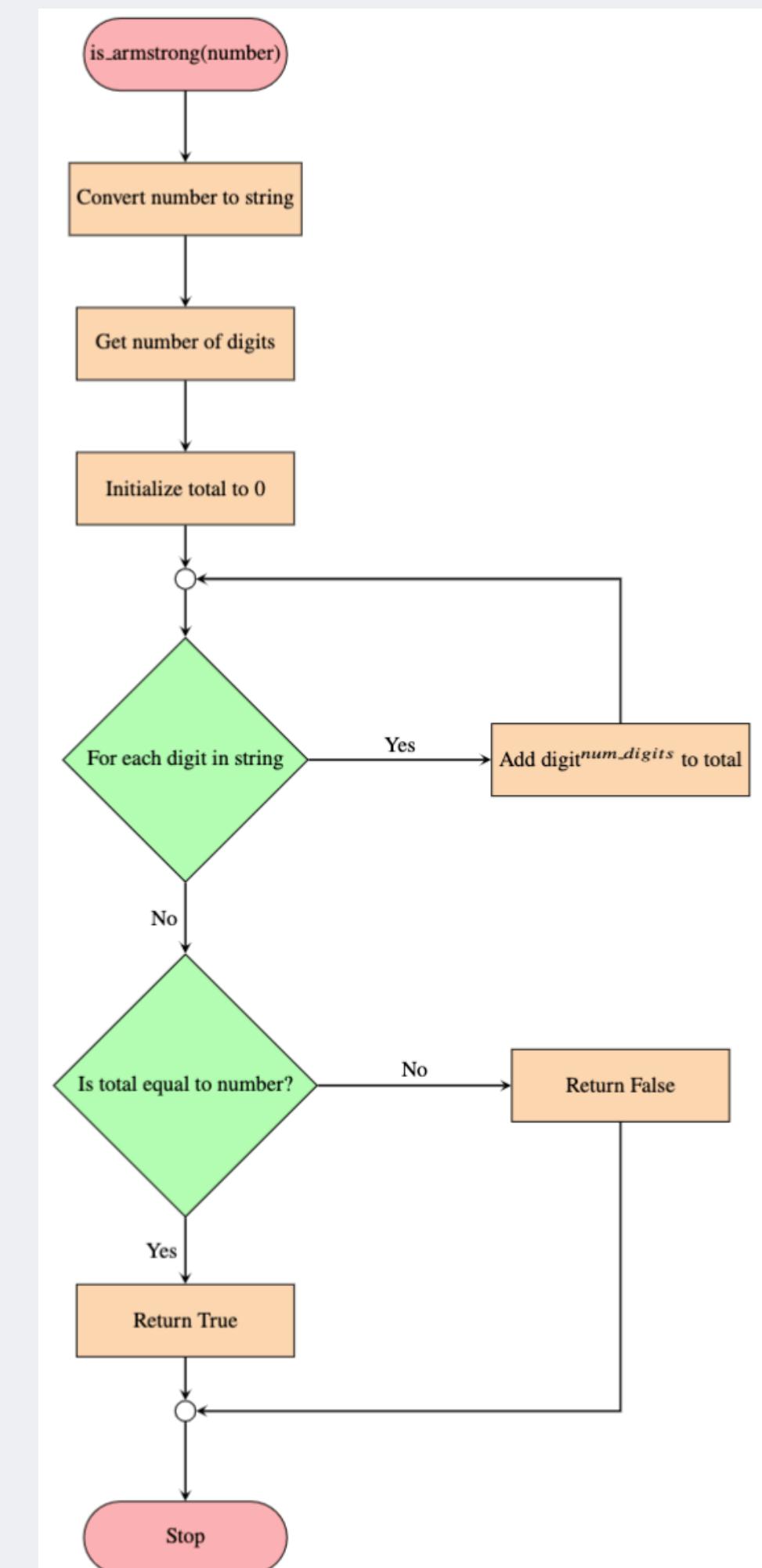


Fig. 5.3: Flowchart for Checking Armstrong Number

FUNCTIONS



Local, Global Variables and Global Constants

FUNCTION SCOPE AND VARIABLES

แนวคิดสำคัญเกี่ยวกับขอบเขตของฟังก์ชัน:

- **ขอบเขต (Scope):** ขอบเขตและช่วงชีวิตของตัวแปร โดยตัวแปรภายในจะถูกจำกัดให้อยู่ภายในฟังก์ชัน ส่วนตัวแปรสากลสามารถเข้าถึงได้จากทุกส่วนของโปรแกรม
- **คำสั่ง global:** คำสั่งที่ใช้ในการเปลี่ยนค่าของตัวแปรสากลจากภายในฟังก์ชัน
- **ตัวแปรภายใน:** ตัวแปรที่ถูกประกาศภายในฟังก์ชันและสามารถใช้งานได้เฉพาะภายในฟังก์ชันนั้นเท่านั้น

LOCAL VARIABLES

ขอบเขตของฟังก์ชันหมายถึงพื้นที่ที่ตัวแปรภายในสามารถมองเห็นและเข้าถึงได้ ตัวแปรที่ประกาศในฟังก์ชันจะใช้งานได้เฉพาะภายในฟังก์ชันนั้น และไม่สามารถเข้าถึงได้จากภายนอก การออกแบบแบบนี้จะช่วยลดความเสี่ยงในการรับกวนค่าของตัวแปรโดยไม่ตั้งใจ และส่งเสริมการเขียนโปรแกรมที่มีระเบียบและปลอดภัยมากยิ่งขึ้น

ตัวอย่าง:

```
1 def my_function():
2     local_variable = "I'm inside the function"
3     print(local_variable)
4
5 # Call Function
6 my_function()
7
8 # Access local_variable from outside, causing an error
9 # print(local_variable) # NameError: name 'local_variable' is not defined
```

Listing 5.13: ตัวอย่างการใช้ขอบเขตของฟังก์ชัน

GLOBAL VARIABLES

ตัวแปรสากลคือ ตัวแปรที่ถูกประกาศไว้นอกฟังก์ชัน และสามารถเข้าถึงได้จากทุกส่วนของโปรแกรม ตัวแปรเหล่านี้ จะคงอยู่ตลอดช่วงการทำงานของโปรแกรม ต่างจากตัวแปรภายในที่หมดอายุเมื่อฟังก์ชันสิ้นสุด ตัวแปรสากลเหมาะสม กับการใช้ร่วมข้อมูลระหว่างฟังก์ชันต่าง ๆ ในโปรแกรม

ตัวอย่าง:

```
1 global_variable = "I'm outside the function"
2
3 def my_function():
4     print(global_variable)
5
6 #
7 my_function() # Output: I'm outside the function
8
9 #
10 print(global_variable) # Output: I'm outside the function
```

Listing 5.14: ตัวอย่างการใช้ตัวแปรสากล

GLOBAL VARIABLES AND CONSTANTS

```
1 # This program simulates 10 tosses of a coin.
2 import random
3
4 # Constants
5 HEADS = 1
6 TAILS = 2
7 TOSSES = 10
8
9 def tosses_coin():
10     for toss in range(TOSSES):
11         # Simulate the coin toss.
12         if random.randint(HEADS, TAILS) == HEADS:
13             print('Heads')
14         else:
15             print('Tails')
16
17 # Call the main function.
18 tosses_coin()
```

Listing 5.12: Example of Global and Constants Variaable

MODIFY GLOBAL VARIABLES

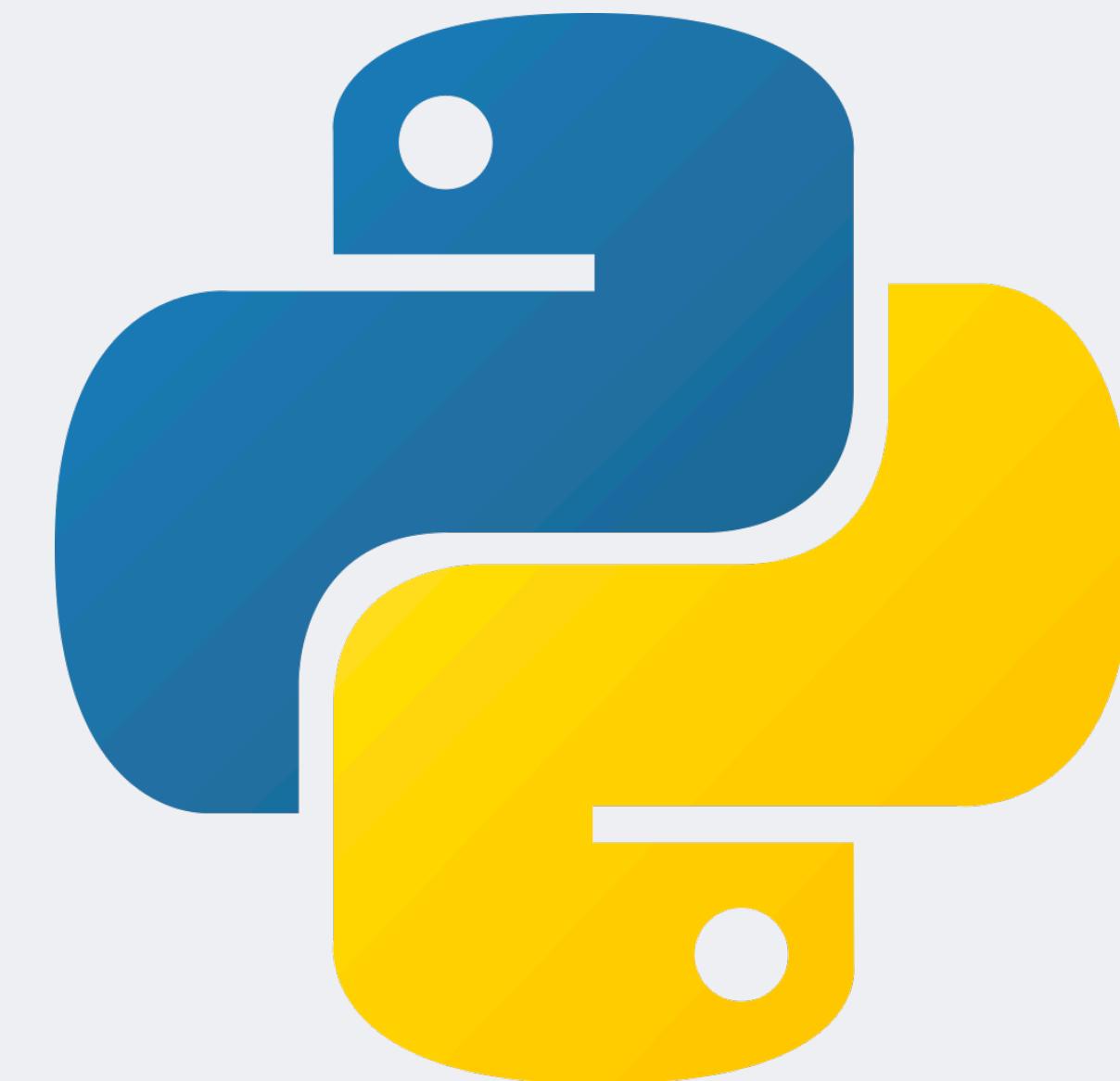
```
1 counter = 0
2
3 def increment():
4     global counter
5     counter += 1
6
7 # Calling the function
8 increment()
9 increment()
10
11 # Accessing the modified global variable
12 print(counter) # Output: 2
```

Listing 5.11: Example of Modify Global Variables

GLOBAL VARIABLES AND CONSTANTS

```
# This program simulates 10 tosses of a coin.  
import random  
  
# Constants  
HEADS = 1  
TAILS = 2  
TOSSES = 10  
  
def main():  
    for toss in range(TOSSES):  
        # Simulate the coin toss.  
        if random.randint(HEADS, TAILS) == HEADS:  
            print('Heads')  
        else:  
            print('Tails')  
  
    # Call the main function.  
main()
```

FUNCTIONS



Storing Functions in Modules

STORING FUNCTIONS IN MODULES

การจัดเก็บฟังก์ชันไว้ในโมดูลเป็นแนวปฏิบัติพื้นฐานที่สำคัญในกระบวนการพัฒนาซอฟต์แวร์ ซึ่งช่วยเพิ่มความเป็นระเบียบ อ่านง่าย ดูแลรักษาได้ง่าย และนำกลับมาใช้ใหม่ได้สะดวก นักพัฒนาสามารถสร้างโครงสร้างโปรแกรมที่เป็นระบบ โดยจัดกลุ่มฟังก์ชันที่เกี่ยวข้องไว้ในโมดูลเดียวกัน ช่วยให้การพัฒนาและดีบักโปรแกรมมีความง่ายและมีประสิทธิภาพมากขึ้น

ในภาษา Python โมดูลคือไฟล์ที่ประกอบด้วยโค้ด Python ซึ่งอาจประกอบด้วยการกำหนดฟังก์ชัน คลาส ตัวแปร และโค้ดที่สามารถเรียกใช้งานได้ ข้อดีหลักของการใช้โมดูลคือช่วยแยกโปรแกรมขนาดใหญ่ออกเป็นส่วนย่อย ๆ ที่จัดการได้ง่ายและเป็นระบบ แนวทางแบบโมดูลาร์นี้สอดคล้องกับหลักการ ”การแยกหน้าที่ (Separation of Concerns)” ซึ่งหมายถึงการจัดการฟังก์ชันเฉพาะอย่างให้ชัดเจนในแต่ละโมดูล

STORING FUNCTIONS IN MODULES

การสร้างโมดูลในภาษา Python ทำได้โดยการบันทึกไฟล์ Python ด้วยนามสกุล .py ตัวอย่างเช่น ไฟล์ที่ชื่อว่า `math_operations.py` ซึ่งประกอบด้วยฟังก์ชันทางคณิตศาสตร์หลายฟังก์ชัน:

ตัวอย่าง:

```
1 # math_operations.py
2
3 def add(a, b):
4     return a + b
5
6 def subtract(a, b):
7     return a - b
8
9 def multiply(a, b):
10    return a * b
11
12 def divide(a, b):
13    if b == 0:
14        return "Error: Division by zero"
15    return a / b
```

Listing 5.17: ตัวอย่างโมดูล `math_operations`

IMPORT AND USING MODULES

```
1 # main.py
2
3 import math_operations
4
5 result_add = math_operations.add(10, 5)
6 result_subtract = math_operations.subtract(10, 5)
7 result_multiply = math_operations.multiply(10, 5)
8 result_divide = math_operations.divide(10, 5)
9
10 print(f"Addition: {result_add}")
11 print(f"Subtraction: {result_subtract}")
12 print(f"Multiplication: {result_multiply}")
13 print(f"Division: {result_divide}")
```

Listing 5.14: Example of Importing and Using Module

IMPORT AND USING MODULES

```
1 from math_operations import add, subtract  
2  
3 result_add = add(10, 5)  
4 result_subtract = subtract(10, 5)
```

Listing 5.19: การนำเข้าเฉพาะฟังก์ชันจากโมดูล

```
1 from math_operations import *  
2  
3 result_add = add(10, 5)  
4 result_subtract = subtract(10, 5)
```

Listing 5.20: การนำเข้าฟังก์ชันทั้งหมดจากโมดูล

```
1 import math_operations as mo  
2  
3 result_add = mo.add(10, 5)
```

Listing 5.21: การใช้ชื่อแฝง (alias) สำหรับโมดูล

BENEFIT OF USING MODULES

การใช้โมดูลในภาษา Python เป็นวิธีที่มีประสิทธิภาพในการจัดระเบียบและจัดการโค้ดให้เป็นระบบ โดยการรวมฟังก์ชันและคลาสที่เกี่ยวข้องไว้ในไฟล์แยกกัน โมดูลช่วยให้โค้ดอ่านง่าย ดูแลรักษาง่าย และสามารถขยายระบบได้ง่ายยิ่งขึ้น แนวทางแบบโมดูลยังช่วยให้สามารถนำกลับมาใช้ซ้ำ และสนับสนุนการทำงานร่วมกันของทีมพัฒนาได้อย่างมีประสิทธิภาพ ด้านล่างนี้คือข้อดีที่สำคัญของการใช้โมดูลใน Python:

- **การจัดระเบียบโค้ด:** การจัดเก็บฟังก์ชันในโมดูลช่วยให้โค้ดมีการจัดโครงสร้างอย่างเป็นระเบียบ ตัวอย่างเช่น ฟังก์ชันที่เกี่ยวกับคณิตศาสตร์สามารถจัดรวมไว้ในโมดูลเดียว ส่วนฟังก์ชันที่เกี่ยวกับการจัดการไฟล์อาจแยกไว้ในอีกโมดูลหนึ่ง
- **การนำกลับมาใช้ซ้ำ:** ฟังก์ชันที่อยู่ในโมดูลสามารถนำกลับมาใช้ในโปรแกรมอื่นได้โดยไม่ต้องเขียนซ้ำ ช่วยลดความซ้ำซ้อนของโค้ด
- **การดูแลรักษา:** เมื่อโค้ดถูกแบ่งเป็นโมดูล จะทำให้การแก้ไขบັນทึกหรืออัปเดตโค้ดทำได้ง่ายและเฉพาะจุด โปรแกรมที่นำเข้าโมดูลนั้นจะได้รับการอัปเดตโดยอัตโนมัติ
- **การจัดการชื่อ (Namespace):** การใช้โมดูลช่วยป้องกันปัญหาการชนกันของชื่อฟังก์ชันหรือค่าตัวแปร เพราะแต่ละโมดูลจะมี namespace ของตนเอง
- **การทำงานร่วมกันของทีม:** ในงานพัฒนาร่วมกัน โมดูลช่วยให้สมาชิกแต่ละคนสามารถทำงานในส่วนของตนเองได้ และสามารถรวมเข้ากับโครงการหลักได้อย่างง่ายดาย
- **ความสามารถในการขยายระบบ:** เมื่อโปรเจกต้มีขนาดใหญ่ขึ้น การใช้โมดูลจะช่วยให้สามารถเพิ่มฟีเจอร์ใหม่ได้ง่ายโดยไม่รบกวนโค้ดเดิม

FUNCTIONS



Recursive: a function that calls itself.

UNDERSTANDING RECURSIVE

ฟังก์ชันแบบเรียกซ้ำ (Recursive Function) คือฟังก์ชันที่เรียกใช้งานตัวเองภายในตัวฟังก์ชันเพื่อลดปัญหาขนาดใหญ่ให้เป็นปัญหาอย่างง่าย แนวคิดของการเรียกซ้ำเป็นพื้นฐานที่สำคัญในวิทยาการคอมพิวเตอร์ โดยช่วยให้การแก้ปัญหาซับซ้อนทำได้ง่ายขึ้น เช่น การคำนวณทางคณิตศาสตร์ การจัดการโครงสร้างข้อมูล และการใช้ในอัลกอริธึม

ฟังก์ชันแบบเรียกซ้ำประกอบด้วยสองส่วนหลัก:

Base Case (กรณีพื้นฐาน): เงื่อนไขที่ใช้สิ้นสุดการเรียกซ้ำ เป็นกรณีที่สามารถให้คำตอบได้โดยไม่ต้องเรียกซ้ำอีกต่อไป

Recursive Case (กรณีเรียกซ้ำ): ส่วนที่ฟังก์ชันเรียกตัวเองด้วยอาร์กิวเม้นต์ที่เปลี่ยนแปลงเพื่อเข้าใกล้ base case ทีละขั้น

FACTORIAL RECURSIVE

$$0! = 1 \text{ (base case)}$$
$$n! = n \times (n - 1)! \text{ for } n > 0 \text{ (recursive case)}$$

Here's how you can implement this in Python:

```
1 def factorial(n):
2     if n == 0:
3         return 1 # Base case
4     else:
5         return n * factorial(n - 1) # Recursive case
6
7 # Example usage
8 print(factorial(5)) # Output: 120
```

Listing 5.18: Factorial Calculation Using Recursion

FIBONACCI RECURSIVE

$F(0) = 0$ (base case)

$F(1) = 1$ (base case)

$F(n) = F(n - 1) + F(n - 2)$ for $n > 1$ (recursive case)

Here's a Python implementation:

```
1 def fibonacci(n):
2     if n == 0:
3         return 0 # Base case
4     elif n == 1:
5         return 1 # Base case
6     else:
7         return fibonacci(n - 1) + fibonacci(n - 2) # Recursive
8             case
9
10 # Example usage
11 print(fibonacci(6)) # Output: 8
```

Listing 5.19: Fibonacci Sequence Using Recursion

ADVANTAGE OF RECURSIVE

- **เข้าใจง่ายและกระชับ:** พัฟก์ชันแบบเรียกซ้ำสามารถแสดงแนวคิดของปัญหาได้ชัดเจน โดยไม่ต้องใช้ลูปที่ซับซ้อน
- **หมายกับโครงสร้างข้อมูล:** หมายกับการประมวลผลข้อมูลเชิงโครงสร้าง เช่น ต้นไม้ (tree), กราฟ (graph), และอัลกอริธึมค้นหา
- **โมดูลาร์:** ทำให้โค้ดมีความเป็นโมดูลและเข้าใจได้ง่าย โดยแบ่งปัญหาเป็นส่วนย่อย ๆ

DISADVANTAGE OF RECURSIVE

- **ประสิทธิภาพ:** มี overhead จากการเรียกฟังก์ชันหลายครั้ง ทำให้ช้ากว่าการใช้ลูปในบางกรณี
- **การใช้หน่วยความจำ:** การเรียกซ้ำหลายชั้นอาจทำให้หน่วยความจำเต็ม (stack overflow)
- **ความซับซ้อนในการดีบัก:** หากไม่มี base case ที่ชัดเจน อาจทำให้เกิดการวนซ้ำไม่รู้จบ

ทางเลือก: ใช้ Loop แทน

```
1 def factorial_iter(n):
2     result = 1
3     for i in range(2, n + 1):
4         result *= i
5     return result
```

Listing 5.24: Iterative Version

FUNCTIONS



Python common functions

COMMON FUNCTIONS

`abs()`

returns the absolute value of a numeric value (e.g. integer or float).

Obviously it can't be a string. It has to be a numeric value.

Example: `abs(-4/3)`

In [5]: `abs(-4/3)`

Out[5]: 1.3333333333333333

COMMON FUNCTIONS

round()

returns the rounded value of a numeric value.

Example: round(-4/3)

In [6]: round(-4/3)

Out[6]: -1

COMMON FUNCTIONS

`min()`

returns the smallest item of a list or of the typed-in arguments. It can even be a string.

Example 1: `min(3,2,5)`

Example 2: `min('c','a','b')`

```
In [7]: min(3,2,5)
```

```
Out[7]: 2
```

```
In [8]: min('c','a','b')
```

`max()`

```
Out[8]: 'a'
```

Guess, what! It's the opposite of `min()`. 😊

COMMON FUNCTIONS

`sorted()`

It sorts a list into ascending order. The list can contain strings or numbers.

Example:

```
a = [3, 2, 1]
```

```
sorted(a)
```

```
In [21]: a = [3, 2, 1]
sorted(a)
```

```
Out[21]: [1, 2, 3]
```

COMMON FUNCTIONS

sum()

It sums a list. The list can have all types of numeric values, although it handles floats... well, not smartly.

```
In [22]: a = [3, 2, 1]
          sum(a)
```

```
Out[22]: 6
```

```
In [23]: b = [4/3, 2/3, 1/3, 1/3, 1/3]
          sum(b)
```

```
Out[23]: 3.000000000000004
```

COMMON FUNCTIONS

<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

EXERCISE-II

```
# Example usage
print(generate_primes(10))    # Output: "2, 3, 5, 7"
print(generate_primes(20))    # Output: "2, 3, 5, 7, 11, 13, 17, 19"
print(generate_primes(1))     # Output: ""
print(generate_primes(2))     # Output: "2"
```

HW-01

```
# Example usage  
result = format_strings("Hello", "world", "this", "is", "a", "test")  
print(result) # Output: "HELLOWORLDTHISISATEST"  
  
result = format_strings("Python", "is", "fun")  
print(result) # Output: "PYTHONISFUN"  
  
result = format_strings("Concatenate", "these", "strings", "please")  
print(result) # Output: "CONCATENATETHESESTRINGSPLEASE"
```

HW-01

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS EXPAND UNRESOLVED COMMENTS

```
@Anirach → /workspaces/hw01-dynamicstringformatter-Anirach (main) $ python formatter.py  
HELLOWORLDTHISISATEST
```

```
PYTHONISFUN  
HELLO-WORLD
```

```
● @Anirach → /workspaces/hw01-dynamicstringformatter-Anirach (main) $ python -m unittest test_formatter.py  
.....
```

```
Ran 6 tests in 0.001s
```

```
OK
```

```
○ @Anirach → /workspaces/hw01-dynamicstringformatter-Anirach (main) $ █
```

HW-01

AnirachClass / hw01-dynamicstringformatter-Anirach

Type ⌘ to search | + ⌄ ⌂ ⌃

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

Actions New workflow

All workflows Showing runs from all workflows

Filter workflow runs

Autograding Tests

Management

Caches Attestations Runners

3 workflow runs Event Status Branch Actor

Workflow	Event	Status	Branch	Actor
Done	now	Success	main	Anirach
Test1	2 hours ago	Success	main	Anirach
add online IDE url; add deadline	2 hours ago	Success	main	github-classroom bot
GitHub Classroom Autograding Workflow	2 hours ago	Success	main	github-classroom bot

ANANYOT SAIWONG Submitted
Submitted

@Namenick-2004 Latest commit 2 hours ago X -O- 1 commit 0/5

Anirach Submitted
Submitted

Link to student Latest commit 1 minute ago ✓ -O- 2 commits 5/5



THANKS

WORD OF THE WISE



Practice does not make perfect.
Only perfect practice makes perfect.

— *Vince Lombardi* —

AZ QUOTES