Lab 5 - Name - Binding - Scope - Report

Students:

1. Nguyen Huynh Thao My - ITCSIU21204
2. Pham Duc Dat - ITITIU20184

1 Overview

Lab 5 focuses on the exploration of scope, binding, and name resolution in Python, emphasizing the differences between static and dynamic scoping. The lab includes a series of exercises designed to deepen understanding of these concepts through practical implementation.

In Exercise 1, the program demonstrates static and dynamic binding using inheritance, showcasing how method resolution changes depending on whether the binding occurs at compile-time or runtime. Exercise 2 explores variable scope and the lookup process in nested functions, illustrating the contrast between static (lexical) and dynamic scoping. Exercise 3 delves into scope and name resolution within nested blocks, highlighting the importance of local and global variables and how they interact in different contexts. Finally, Exercise 4 applies these concepts to a real-world scenario, demonstrating the impact of scope and visibility on the behavior of a system that integrates global and local configurations.

Through these exercises, Lab 5 provides hands-on experience with essential programming concepts, equipping students with the knowledge to effectively manage scope and binding in complex Python applications.

2 Results

In this section, you:

- Exercise 1, the differences between static and dynamic binding were clearly illustrated, showing how method calls are resolved differently at compile-time versus runtime.

- Exercise 2 highlighted the importance of scope in nested functions, particularly how variable lookup can vary between static and dynamic scoping models. The results reinforced the predictability of static scoping and the flexibility of dynamic scoping.

- In Exercise 3, the program explored scope and name resolution within nested blocks, emphasizing how variables defined in inner blocks are not accessible in outer blocks, thereby illustrating the principles of local and global scope.

- Exercise 4 applied these concepts in a practical scenario, demonstrating how global and local configurations interact in an integrated monitoring and user settings system. The results provided clear insights into how scope and visibility affect the behavior and reliability of Python programs.

2.1 Running the code

2.1.1 Running Exercise 1

To run Exercise 1, use the following command in your terminal:

```
python3 Exercise1.py
```

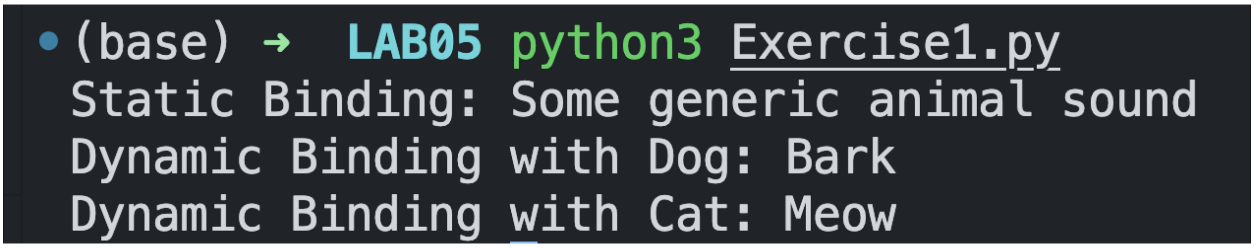


Figure 1: Output of Exercise 1

2.1.2 Running Exercise 2

To run Exercise 2, use the following command in your terminal:

```
python3 Exercise2.py
```

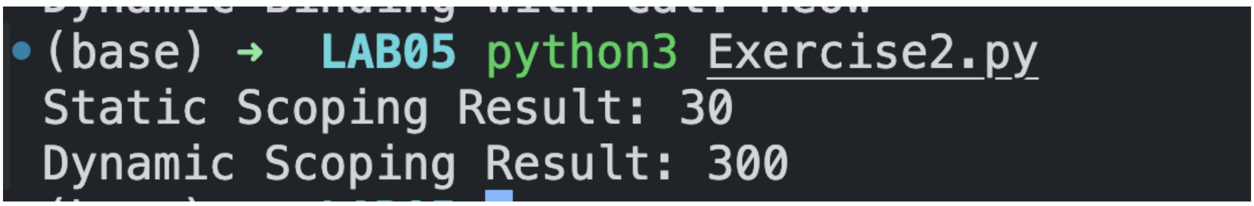


Figure 2: Output of Exercise 2

2.1.3 Running Exercise 3

To run Exercise 3, use the following command in your terminal:

```
python3 Exercise3.py
```

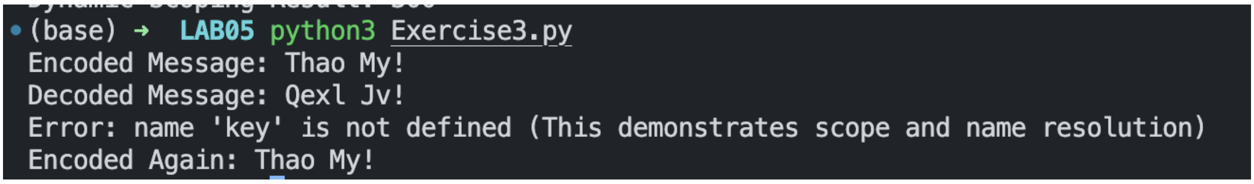


Figure 3: Output of Exercise 3

2.1.4 Running Exercise 4

To run Exercise 4, use the following command in your terminal:

```
python3 Exercise4.py
```

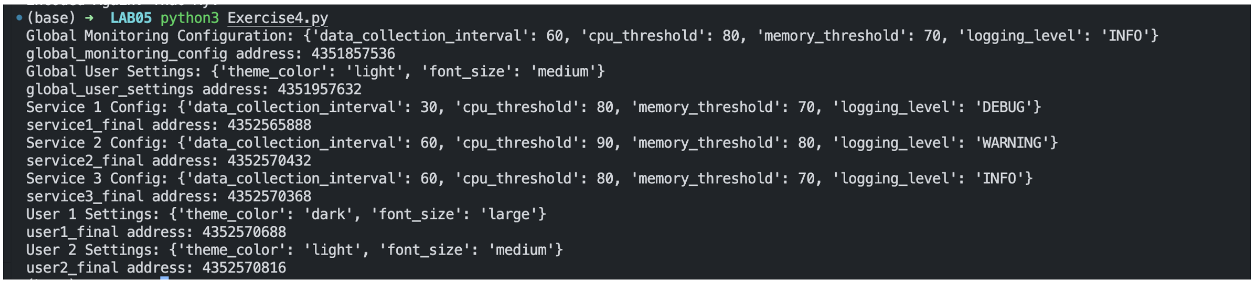


Figure 4: Output of Exercise 4

2.2 Exercise 1: Write a program to demonstrate the difference between static and dynamic binding

Code Image:

Output Image:

Explanation:

The provided code defines a base class `Animal` with a method `sound()` that returns a generic animal sound. Two derived classes, `Dog` and `Cat`, inherit from `Animal` and override the `sound()` method to return specific sounds ("Bark" and "Meow", respectively).

```python
class Animal:
    def sound(self):
        return "Some generic animal sound"

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

def demonstrate_static_binding():
    """This function demonstrates static binding (early binding)."""
    animal = Animal()
    sound = animal.sound()  # The method to be called is determined at compile-time
    return f"Static Binding: {sound}"

def demonstrate_dynamic_binding():
    """This function demonstrates dynamic binding (late binding)."""
    dog = Dog()
    cat = Cat()

    # The method to be called is determined at runtime based on the object
    dog_sound = dog.sound()
    cat_sound = cat.sound()

    return f"Dynamic Binding with Dog: {dog_sound}\nDynamic Binding with Cat: {cat_sound}"

def main():
    print(demonstrate_static_binding())
    print(demonstrate_dynamic_binding())

if __name__ == "__main__":
    main()
```

Figure 5: Code demonstrating static and dynamic binding

```
(base) → LAB05 python3 Exercise1.py
Static Binding: Some generic animal sound
Dynamic Binding with Dog: Bark
Dynamic Binding with Cat: Meow
```

Figure 6: Program Output

- **Static Binding:** The function `demonstrate_static_binding()` demonstrates static binding by creating an instance of the `Animal` class and calling its `sound()` method. In this case, the method to be called is determined at compile-time, resulting in the output "Some generic animal sound".

- **Dynamic Binding:** The function `demonstrate_dynamic_binding()` demonstrates dynamic binding by creating instances of the `Dog` and `Cat` classes. The actual `sound()` method that gets called is determined at runtime based on the type of the object, resulting in the outputs "Bark" and "Meow".

- **Main Function:** The `main()` function calls both `demonstrate_static_binding()`

and `demonstrate_dynamic_binding()` to illustrate the difference in behavior between static and dynamic binding.

- **Number of Blocks:** The program consists of three blocks: the class definitions, the function definitions, and the main block where the functions are called.
- **Number of Variables:** The program has three main variables: `animal`, `dog`, and `cat`, which hold instances of the `Animal`, `Dog`, and `Cat` classes, respectively.
- **Binding Process:**
  - In the static binding example, the method `sound()` is bound to the `Animal` class at compile-time.
  - In the dynamic binding example, the method `sound()` is determined at runtime based on the actual object type (either `Dog` or `Cat`).
- **Scope of Variables:**
  - The variable `animal` has a local scope within `demonstrate_static_binding()`.
  - The variables `dog` and `cat` have local scopes within `demonstrate_dynamic_binding()`.
- **Table of Blocks and Variables:**

Block	Visible Variables	Hidden Variables	Comments
Global	None	None	Global scope
demonstrate_static_binding	animal	None	Local variable `animal`
demonstrate_dynamic_binding	dog, cat	None	Local variables `dog`, `cat`

Table 1: Blocks and Variable Visibility

Explanation:

This program clearly demonstrates the difference between static and dynamic binding in object-oriented programming. Static binding, also known as early binding, occurs at compile-time and is shown by the `Animal` class's `sound()` method. Dynamic binding, or late binding, occurs at runtime and is demonstrated through the derived `Dog` and `Cat` classes, where the actual method invoked depends on the object's type.

The output from the static binding is determined by the class the method belongs to at compile-time, while the output from the dynamic binding is determined by the actual object instance at runtime.

2.3 Exercise 2: Write functions where inner functions use variables from outer functions. Show how the lookup of variables differs in static and dynamic scoping. Analyze the differences and discuss the advantages of each scoping mechanism.

Code Image:

Exercise1.py   Exercise2.py

Exercise2.py > ...

```python
def calculate_static_sum():
    static_number1 = 10
    static_number2 = 20

    def add_static_numbers():
        return static_number1 + static_number2

    return add_static_numbers()

def calculate_dynamic_sum():
    dynamic_number1 = 10
    dynamic_number2 = 20

    def add_dynamic_numbers():
        return dynamic_lookup('dynamic_number1') + dynamic_lookup('dynamic_number2')

    return add_dynamic_numbers()

def dynamic_lookup(var_name):
    import inspect
    for frame in inspect.stack():
        if var_name in frame[0].f_globals or var_name in frame[0].f_locals:
            return frame[0].f_globals.get(var_name, frame[0].f_locals.get(var_name))
    raise NameError(f"{var_name} not found")

def demonstrate_scoping():
    global dynamic_number1, dynamic_number2
    dynamic_number1 = 100
    dynamic_number2 = 200

    static_result = calculate_static_sum()
    print(f"Static Scoping Result: {static_result}")

    dynamic_result = calculate_dynamic_sum()
    print(f"Dynamic Scoping Result: {dynamic_result}")

if __name__ == "__main__":
    demonstrate_scoping()
```

Figure 7: Code demonstrating static and dynamic scoping

Output Image:

Explanation:

The code defines two functions: `calculate_static_sum()` and `calculate_dynamic_sum()`.

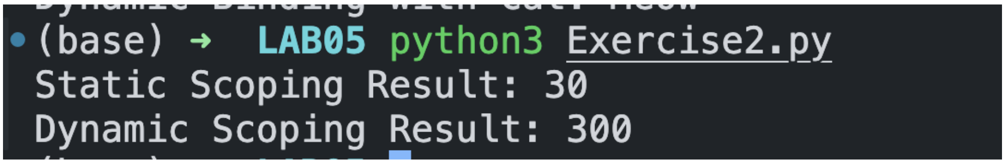


Figure 8: Program Output

Each function demonstrates a different scoping mechanism:

- **calculate_static_sum Function (Static Scoping):**
  - This function defines two local variables, `static_number1` and `static_number2`.
  - An inner function `add_static_numbers` adds these two numbers.
  - Since Python uses static (lexical) scoping by default, the inner function accesses the variables from its enclosing scope (the outer function).
- **calculate_dynamic_sum Function (Simulated Dynamic Scoping):**
  - This function also defines two local variables, `dynamic_number1` and `dynamic_number2`.
  - An inner function `add_dynamic_numbers` uses `dynamic_lookup` to find these variables dynamically at runtime.
  - The global variables `dynamic_number1` and `dynamic_number2` are used instead of the local ones, simulating dynamic scoping.
- **demonstrate_scoping Function:**
  - This function first demonstrates static scoping by calling `calculate_static_sum()` and printing the result.
  - Then, it demonstrates dynamic scoping by calling `calculate_dynamic_sum()` and printing the result.
- **Number of Blocks:**

  - The program has a total of four main blocks: the global scope, the `calculate_static_sum()` function, the `calculate_dynamic_sum()` function, and the `demonstrate_scoping()` function.

- **Number of Variables:**
  - The program has five variables: `static_number1`, `static_number2`, `dynamic_number1`, `dynamic_number2`, and `add_static_numbers`.

- **Binding Process:**
  - The variables `static_number1` and `static_number2` are bound at compile-time in the context of static scoping.
  - The variables `dynamic_number1` and `dynamic_number2` are looked up dynamically at runtime.

- **Scope of Variables:**
  - `static_number1` and `static_number2` have local scope within `calculate_static_sum()`.
  - `dynamic_number1` and `dynamic_number2` have local scope within `calculate_dynamic_sum()`, but are overridden by global variables during dynamic lookup.

- **Table of Blocks and Variables:**

Summary:
This program illustrates how the same operation—adding two numbers—can produce different results depending on the scoping mechanism used. Static scoping (lexical scoping) relies on the structure of the code to determine variable references, making it predictable and reliable. Dynamic scoping, simulated here, allows variables to be looked up dynamically at runtime, leading to results that depend on the current state of the environment. While static scoping provides more certainty, dynamic scoping offers flexibility that can be advantageous in certain scenarios.

Block	Visible Variables	Hidden Variables	Comments
Global	dynamic_number1, dynamic_number2	None	Global scope
calculate_static_sum	static_number1, static_number2	None	Local variables
calculate_dynamic_sum	dynamic_number1, dynamic_number2	Overridden by globals	Local variables overridden during dynamic lookup
demonstrate_scoping	static_result, dynamic_result	None	Calls and displays results from both functions

Table 2: Blocks and Variable Visibility

2.4 Exercise 3: Write a program to demonstrate scope and name resolution in nested blocks.

Code Image:

Output Image:

Explanation:

The code defines two functions: `decode_message()` and `demonstrate_scope_with_decode()`. These functions are designed to show how scope and name resolution work in Python:

- **Decoding a Message:**
  - The `decode_message` function takes an encoded message as input and uses an inner function `decode_inner` to perform the decoding.
  - The `decode_inner` function uses a Caesar cipher with a key of 3 to shift characters back by three positions in the alphabet.
  - The inner function `decode_inner` has its own scope where `key` is defined, and it is used only within that function.
- **Demonstrate Scope with Decode:**

```python
def decode_message(encoded_message):
    def decode_inner(encoded):
        key = 3
        decoded_chars = []

        for char in encoded:
            if char.isalpha():
                shifted = ord(char) - key
                if char.islower():
                    if shifted < ord('a'):
                        shifted += 26
                elif char.isupper():
                    if shifted < ord('A'):
                        shifted += 26
                decoded_chars.append(chr(shifted))
            else:
                decoded_chars.append(char)

        return ''.join(decoded_chars)

    decoded_message = decode_inner(encoded_message)
    return decoded_message

def demonstrate_scope_with_decode():
    encoded_message = "Thao My!"
    print(f"Encoded Message: {encoded_message}")

    decoded = decode_message(encoded_message)
    print(f"Decoded Message: {decoded}")

    try:
        print(f"Key used for decoding: {key}")
    except NameError as e:
        print(f"Error: {e} (This demonstrates scope and name resolution)")

    def encode_message(decoded_message):
        def encode_inner(decoded):
            key = 3
            encoded_chars = []

            for char in decoded:
                if char.isalpha():
                    shifted = ord(char) + key
                    if char.islower():
                        if shifted > ord('z'):
                            shifted -= 26
                    elif char.isupper():
                        if shifted > ord('Z'):
                            shifted -= 26
                    encoded_chars.append(chr(shifted))
                else:
                    encoded_chars.append(char)

            return ''.join(encoded_chars)

        encoded_message = encode_inner(decoded_message)
        return encoded_message

    encoded_again = encode_message(decoded)
    print(f"Encoded Again: {encoded_again}")

if __name__ == "__main__":
    demonstrate_scope_with_decode()
```

Figure 9: Code demonstrating scope and name resolution in nested blocks

```
(base) → LAB05 python3 Exercise3.py
Encoded Message: Thao My!
Decoded Message: Qexl Jv!
Error: name 'key' is not defined (This demonstrates scope and name resolution)
Encoded Again: Thao My!
```

Figure 10: Program Output

– The `demonstrate_scope_with_decode` function calls `decode_message` to decode an encoded message.

  - After decoding, it tries to access the `key` used in `decode_inner`, which raises a `NameError` because `key` is not in the scope of `demonstrate_scope_with_decode`.
  - The program also demonstrates how to reverse the process by encoding the decoded message again using an outer function `encode_message` and an inner function `encode_inner`.

- **Name Resolution:**
  - The `key` used in the decoding process is local to `decode_inner` and cannot be accessed outside of it, demonstrating the concept of scope and name resolution.
  - Similarly, the encoding process uses its own inner scope with a `key` variable, independent of the decoding process.

- **Number of Blocks:**
  The program contains a total of four main blocks: the global scope, the `decode_message()` function, the `demonstrate_scope_with_decode()` function, and the `encode_message()` function.

- **Number of Variables:**
  The program uses six main variables: `encoded_message`, `decoded_message`, `key`, `decoded`, `encoded_again`, and `encoded_chars`.

- **Binding Process:**
  Binding for each variable occurs when the respective function is called, and variables within that function are bound to their values. For instance, `key` is bound within the `decode_inner` and `encode_inner` functions at runtime when they are called.

- **Scope of Variables:**
  The `encoded_message`, `decoded_message`, `decoded`, and `encoded_again` variables have a local scope within their respective functions. The `key` variable is local to the `decode_inner` and `encode_inner` functions.

- **Table of Blocks and Variables:**

Summary:

Block	**Visible Variables**	**Hidden Variables**	**Comments**
Global	encoded_message, decoded, encoded_again	None	Global scope
decode_message	encoded_message, key, decoded_chars	None	Local variables within the `decode_message` function
decode_inner	key, decoded_chars	None	Local variables within the `decode_inner` function
demonstrate_scope_with_decode	encoded_message, decoded, key, encoded_again	None	Demonstrates scope and name resolution with nested functions
encode_message	decoded_message, key, encoded_chars	None	Local variables within the `encode_message` function
encode_inner	key, encoded_chars	None	Local variables within the `encode_inner` function

Table 3: Blocks and Variable Visibility

This example offers a detailed exploration of scope and name resolution in Python through the use of nested functions. By demonstrating how an encoded message is decoded and then re-encoded, the program highlights the concept of variable scope, particularly the fact that variables defined within an inner function, such as `key` in `decode_inner` and `encode_inner`, remain confined to that function. This encapsulation is a fundamental principle in Python, ensuring that inner variables do not interfere with variables in the outer scope, thus maintaining the integrity and independence of different code blocks.

The exercise also underscores the importance of understanding name resolution, where Python resolves variable references by searching from the innermost to the outermost scope. The inability to access the `key` variable outside of its defining function illustrates how Python's scoping rules prevent unintended interactions between different parts of the code. This kind of strict scope control is essential for writing modular and maintainable code, where each function or block operates independently without unintended side effects.

In practice, mastering scope and name resolution in Python is crucial for developers, as it directly impacts the reliability and robustness of their programs. By ensuring that variables are appropriately scoped and resolved, developers can prevent bugs and conflicts that might arise from variable shadowing or accidental reuse of variable names. This exercise serves as a reminder of the importance of these concepts and their role in creating well-structured and error-free code.

2.5 Exercise 4: Create a system where user settings (e.g. theme color, font size) can be defined globally, and individual users can override these settings locally.

Code Image:

Output Image:

Explanation

- **Global Monitoring Configuration:**
  The global configuration, `global_monitoring_config`, serves as a default setting for all services. This configuration is set in the global scope, making it accessible throughout the program. Global variables like `global_monitoring_config` and `global_user_settings` are defined at the top level of the script, providing default monitoring settings and user preferences, respectively.

- **Service Configurations:**
  For each service, local configurations (`service1_config`, `service2_config`) override certain global settings. The function `resolve_monitoring_config` is used to merge these local configurations with the global settings, ensuring that each service has the correct monitoring parameters based on both global defaults and specific

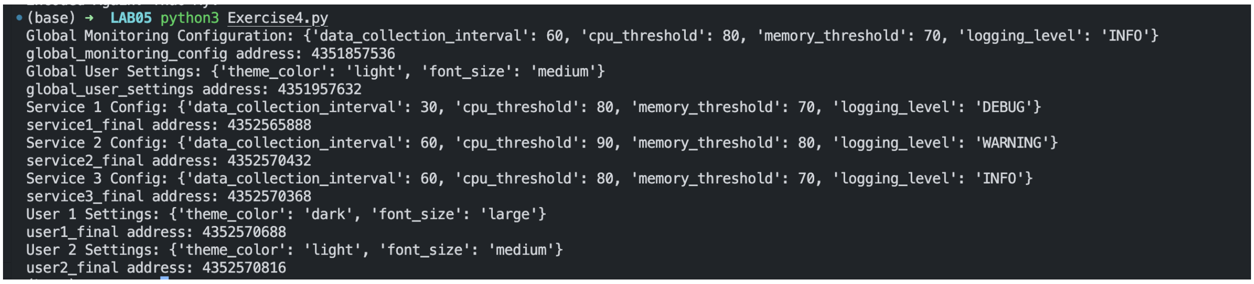
```python
global_monitoring_config = {
    "data_collection_interval": 60,
    "cpu_threshold": 80,
    "memory_threshold": 70,
    "logging_level": "INFO"
}
global_user_settings = {
    "theme_color": "light",
    "font_size": "medium"
}
def print_variable_address(name, var):
    """
    Print the memory address of a variable using Python's built-in id() function.
    """
    print(f"{name} address: {id(var)}")

def resolve_config(global_config, local_config):
    """
    Merge global configuration with local overrides and return the final configuration.
    """
    config = global_config.copy()
    config.update(local_config)
    return config

def demonstrate_scope_and_visibility():
    print("Global Monitoring Configuration:", global_monitoring_config)
    print_variable_address("global_monitoring_config", global_monitoring_config)

    print("Global User Settings:", global_user_settings)
    print_variable_address("global_user_settings", global_user_settings)

    service1_config = {
        "data_collection_interval": 30,
        "logging_level": "DEBUG"
    }
    service1_final = resolve_config(global_monitoring_config, service1_config)
    print("Service 1 Config:", service1_final)
    print_variable_address("service1_final", service1_final)

    service2_config = {
        "cpu_threshold": 90,
        "memory_threshold": 80,
        "logging_level": "WARNING"
    }
    service2_final = resolve_config(global_monitoring_config, service2_config)
    print("Service 2 Config:", service2_final)
    print_variable_address("service2_final", service2_final)

    service3_final = resolve_config(global_monitoring_config, {})
    print("Service 3 Config:", service3_final)
    print_variable_address("service3_final", service3_final)

    user1_settings = {
        "theme_color": "dark",
        "font_size": "large"
    }
    user1_final = resolve_config(global_user_settings, user1_settings)
    print("User 1 Settings:", user1_final)
    print_variable_address("user1_final", user1_final)

    user2_final = resolve_config(global_user_settings, {})
    print("User 2 Settings:", user2_final)
    print_variable_address("user2_final", user2_final)

if __name__ == "__main__":
    demonstrate_scope_and_visibility()
```

Figure 11: Code for system

local overrides. The service configurations are defined within the `demonstrate_scope_and_visibility` function, where they are locally scoped to this function and not accessible outside.

- **Variable Address Printing:**

  To illustrate the scope and visibility of these configurations, the program includes



```
(base) → LAB05 python3 Exercise4.py
Global Monitoring Configuration: {'data_collection_interval': 60, 'cpu_threshold': 80, 'memory_threshold': 70, 'logging_level': 'INFO'}
global_monitoring_config address: 4351857536
Global User Settings: {'theme_color': 'light', 'font_size': 'medium'}
global_user_settings address: 4351957632
Service 1 Config: {'data_collection_interval': 30, 'cpu_threshold': 80, 'memory_threshold': 70, 'logging_level': 'DEBUG'}
service1_final address: 4352565888
Service 2 Config: {'data_collection_interval': 60, 'cpu_threshold': 90, 'memory_threshold': 80, 'logging_level': 'WARNING'}
service2_final address: 4352570432
Service 3 Config: {'data_collection_interval': 60, 'cpu_threshold': 80, 'memory_threshold': 70, 'logging_level': 'INFO'}
service3_final address: 4352570368
User 1 Settings: {'theme_color': 'dark', 'font_size': 'large'}
user1_final address: 4352570688
User 2 Settings: {'theme_color': 'light', 'font_size': 'medium'}
user2_final address: 4352570816
```

Figure 12: Program Output

a function, `print_variable_address`, which prints out the memory addresses of the configuration dictionaries. This helps demonstrate how each service's settings are handled in memory, showing the differences between global and locally overridden settings.

- **Number of Blocks:**
  The program consists of two main blocks: the global configuration block (Block 1) and the local configuration block within the `demonstrate_scope_and_visibility` function (Block 2).

- **Number of Variables:**
  There are six main variables: `global_monitoring_config`, `global_user_settings`, `service1_config`, `service2_config`, `service1_final`, and `service2_final`.

- **Binding Process:**
  Binding for global variables occurs at the start of the program when they are defined. Binding for local variables happens within the `demonstrate_scope_and_visibility` function when it is called.

- **Scope of Variables:**
  `global_monitoring_config` and `global_user_settings` have global scope. `service1_config`, `service2_config`, `service1_final`, and `service2_final` have local scope within the `demonstrate_scope_and_visibility` function.

- **Table of Blocks and Variables:**

Summary

This exercise demonstrates the management of global and local configurations within

Block	**Visible Variables**	**Hidden Variables**	**Comments**
Global	`global_monitoring_config`, `global_user_settings`	None	Global scope
demonstrate_scope_and_visibility	`service1_final`, `service2_final`	`service1_config`, `service2_config`	Local variables within the function
resolve_monitoring_config	`service1_config`, `service2_config`	`global_monitoring_config`	Resolves local and global configurations

Table 4: Blocks and Variable Visibility

a Python program, simulating the setup of a monitoring system for an IT infrastructure. By defining global and local configurations, the program shows how different services can have specific settings that override the global defaults, and how these configurations are handled in memory. The use of variable address printing highlights the differences between global and local scope, ensuring that the right settings are applied in each context. This exercise is a practical example of how scope and visibility are crucial concepts in programming, particularly when dealing with complex systems that require flexible and configurable setups.