DAY 12: Scope & Number guessing name

31/01/2025

Namespaces: Local vs. Global Scope

Scope determines where a variable or function can be accessed in your code. Think of it like an apple tree: if it's inside your fenced garden, only you can access it (local scope). If it's by the sidewalk, anyone can access it (global scope).

Local Scope

Variables created inside a function only exist within that function.

```
11
12  #Local Scope
13  def drink_potion():  1 usage
14      potion_strength = 2
15      print(potion_strength)
16
17  drink_potion()
18  print(potion_strenght)  #The output will give a name error,
19                          # because you are calling outside the function
20
```

When you create a new variable or indeed a new function inside another function, it's only accessible, you can only use it when you're inside that function because it has local scope.

Namespace: Anything you name (variables, functions) belongs to a specific scope. If a function is nested inside another, it only exists within that function.

Global Scope

Variables defined outside functions are accessible everywhere.

```
21
22    #Global Scope
23    player_health = 10  # Global variable
24
25    def drink_potion():  1 usage
26        print(player_health)  #  Accessible inside the function
27
28    drink_potion()
29    print(player_health)  #  Still accessible outside
30
```

it's available anywhere within our file because it was defined at the top level of the file.
Now when I say top level, I don't mean physically at the very top of the file. I mean that it's not within another function like potion_strength() here, because this is defined inside a function.

Does Python Have Block Scope?

In Python, **Block Scope** doesn't exist like in languages like C++ or Java. Variables inside `if`, `for`, or `while` blocks are still accessible outside the block, as long as they are in the same function or global scope.

- **No Block Scope:** In Python, variables created inside blocks like `if` or loops are still accessible outside those blocks within the same function or globally.
- **Local Scope in Functions:** Inside functions, variables are local and can't be accessed outside the function.
- **Linter Warnings:** If a variable is created inside a block that might not execute, you may get a warning about it being "referenced before assignment." You can avoid this by initializing variables outside the block.

Example Fix:

```
19
20    new_enemy = None  # Initialize outside
21    if game_level < 5:
22        new_enemy = enemies[0]
23    print(new_enemy)  # No warning
24
```

Prime Number Checker

exercise.py

```python
def prime_number(n):
    for i in range(2,n):
        if n % i == 0:
            return False
    return True
print(prime_number(73))
```

How to Modify a Global Variable

If you want to modify a **global variable** inside a function, you have to explicitly tell Python that you intend to use the global variable, not create a new local one. You can do this using the `global` keyword.

```python
enemies = 1  # Global variable

def modify_enemies():
    global enemies  # Referencing the global variable
    enemies += 1  # Modifies the global variable

modify_enemies()
print(enemies)  # Now it prints 2
```

Without the `global` keyword, Python assumes you are creating a new **local** variable inside the function, and you cannot modify the global variable.

Why Modifying Global Variables is Not Recommended:

While modifying global variables inside a function is possible, it's not a good practice. Here's why:

- **Confusion**: It can be hard to track when and where a global variable is modified, especially in larger programs.
- **Prone to Errors**: If you modify a global variable inside a function, you risk introducing bugs because you don't always know when that global variable might have been modified elsewhere in the code.
- **Readability**: It makes the code less readable and harder to debug because the global variable could have been created or changed at any point in the code.

Python Constants and Global Scope

Global constants are variables that are never modified after they are initially set. These constants represent values that are used throughout your program, but should not change, like the value of **π (pi)** or a **URL**.

- **Example**:
  `PI = 3.14159`
  This value is not expected to change, and every time you need to use it, you just reference `PI`.

Naming Convention for Constants:

To make it clear that a variable is a constant (and therefore not to be changed), Python uses a special naming convention: **uppercase letters** with **underscores** separating words.

- **Examples**:
  - `PI = 3.14159`
  - `MAX_SIZE = 100`
  - `URL = "https://www.example.com"`

Why Use Global Constants?:

Global constants are useful for storing values that are fixed and need to be used across different parts of your program. By defining them globally:

- You avoid repeatedly hardcoding values (e.g., pi) throughout your code.
- It makes your code more maintainable and clear.
- It keeps your code DRY (Don't Repeat Yourself).

Introducing the Final Project: The Number Guessing Game

main.py

```python
import random

print("Welcome to the Number Guessing Game")
print("I'm thinking of a number between 1 to 100")

level_user = input("Choose a level 'Easy' or 'Hard': ").strip().lower()
num = random.randint( a: 1,  b: 100)  # Generate a random number
attempts = 10 if level_user == "easy" else 5  # Set attempts based on difficulty

while attempts > 0:
    guess = int(input(f"You have {attempts} attempts left. Make your guess: "))

    if guess == num:
        print("Congratulations! You won!")
        break  # Exit the loop if guessed correctly
    elif guess > num:
        print("Too high! Try again.")
    else:
        print("Too low! Try again.")

    attempts -= 1  # Reduce attempts after each guess

if attempts == 0:
    print(f"Sorry, you've run out of attempts. The number was {num}.")
```