Data Science & AI Fellowship

Instructor/Mentor: Sir Noorullah

Assignment # 02

Submitted by: Nadia Mustafa

Problem # 01 – Debugging

1. Error Version (Code)

```
print(Day 1 - String Manipulation)

print("String Concatenation is done with the "+" sign.")

print('e.g. print("Hello " + world)')
```

2. Debugging Explanation

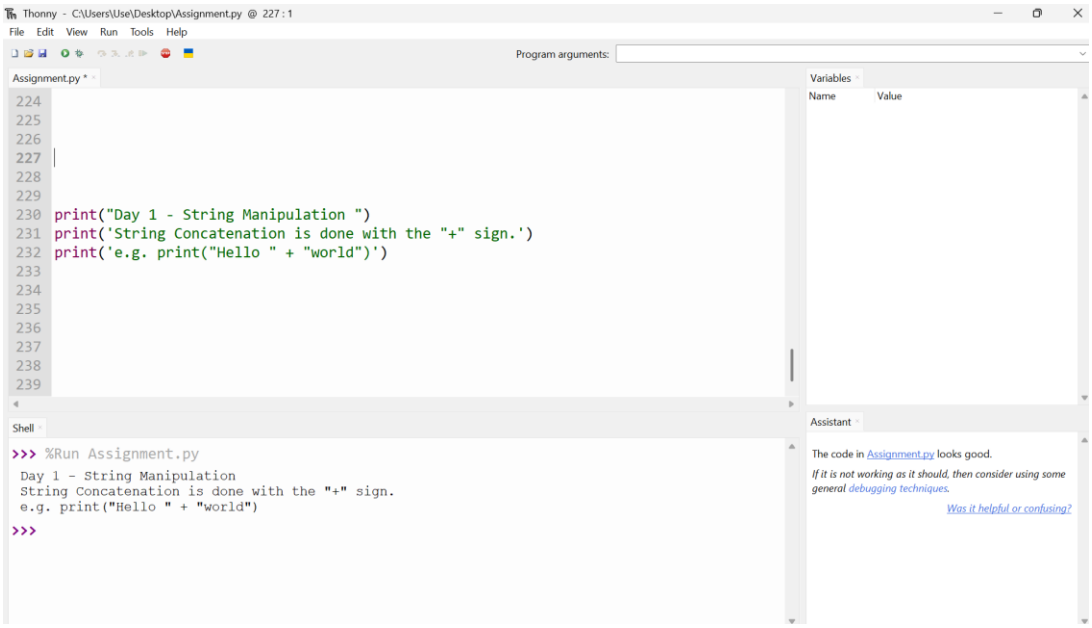
- **Line 1 Error:**
  The text Day 1 - String Manipulation is **not inside quotation marks**. Python thinks it's code, not a string.
- **Line 2 Error:**
  The **string is broken** because of the "+" sign. part. The quotes are not placed correctly.
- **Line 3 Error:**
  The word world is **missing quotation marks**. Python assumes it is a variable, but it should be a string.

3. Corrected Code

```
print("Day 1 - String Manipulation ")

print('String Concatenation is done with the "+" sign. ')

print('e.g. print("Hello " + "world")')
```

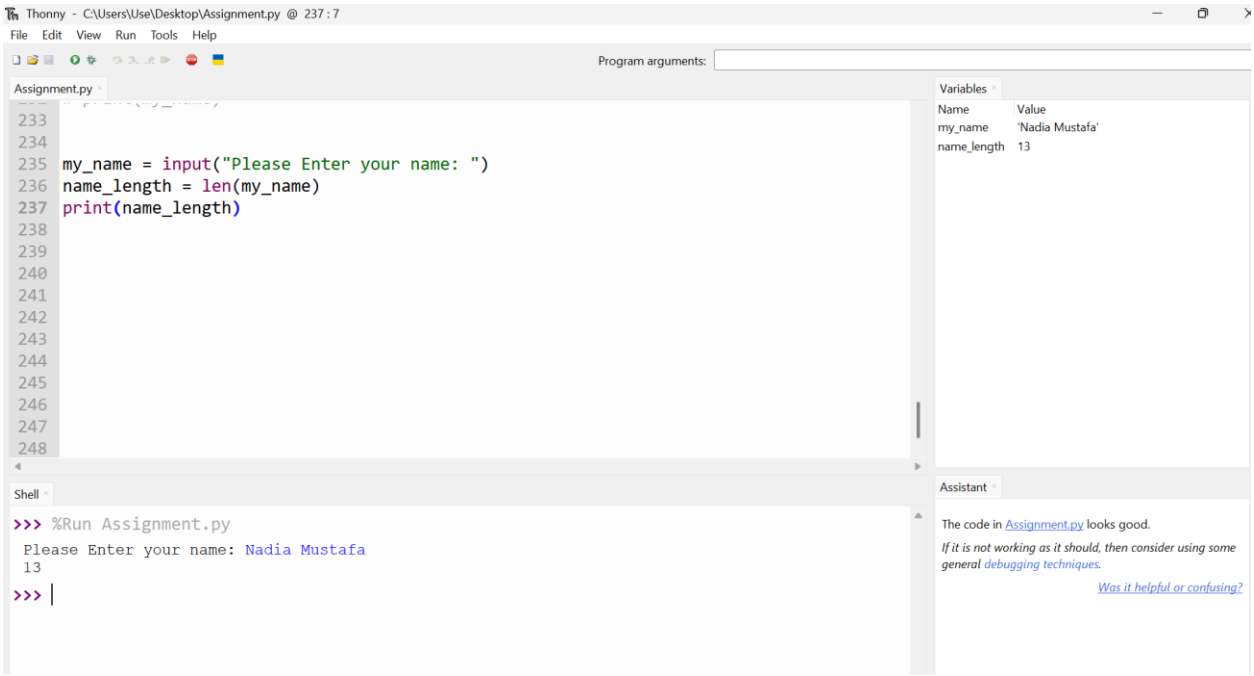


Problem # 02 - Inputs

my_name = input("Please Enter your name: ")

name_length = len(my_name)

print(name_length)



Problem # 03 - Variables

```
a = input("enter any value:  ")

b = input("What is temperature outside: ")


temp = a

a = b

b = temp


print("a: " + a)

print("b: " + b)
```

Example Dry Run

A *dry run* means **we trace the program line by line and write down what's inside each variable (a, b, temp) at every step.**

👉 Suppose the user types:

- a = 10
- b = 25

Step	Code Line	a	b	temp	What happens
1	a = input("enter any value")	10	–	–	User enters 10, stored in a
2	b = input("What is temperature outside")	10	25	–	User enters 25, stored in b
3	temp = a	10	25	10	Save a's value into temp
4	a = b	25	25	10	Put b's value into a
5	b = temp	25	10	10	Put temp (old a) into b
6	Print results	25	10	10	Shows swapped values

Final Output

a: 25

b: 10

✅ That's the **swap in action**: the **two values exchanged places**.

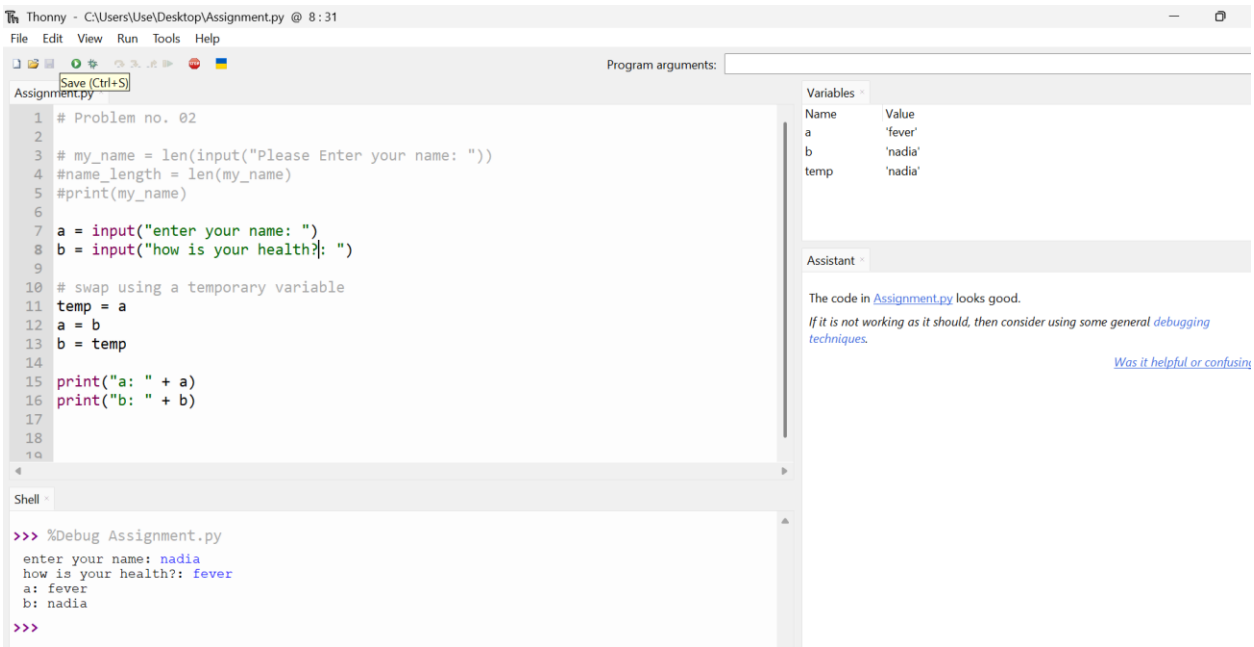
Printing the swapped values

print("a: " + a)

print("b: " + b)

- **"a: "** and **"b: "** are **string labels**.
- The **+** operator **concatenates** (joins) these labels with the **values stored in a and b**.
- Since both **input values** are **stored as strings**, they join correctly.
- Example: **"a: " + "5"** → **"a: 5"**

Example # 02 – with string:



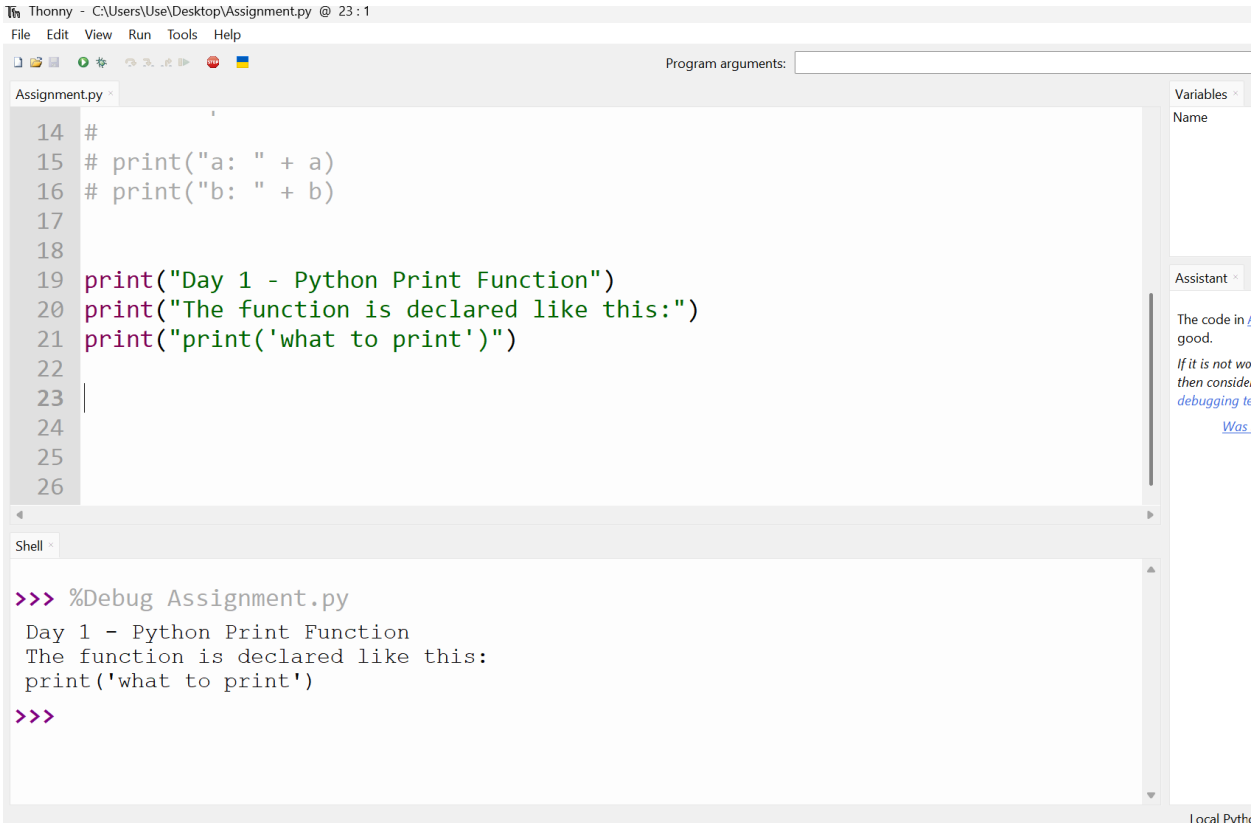
Problem # 04 - Printing to the Console

program in main.py

print("Day 1 - Python Print Function")

print("The function is declared like this:")

print("print('what to print')")



Problem # 05 – Data Types

code on lines 1–3 = Input() & Variable

input("Type a two-digit number: "**)** -> **string value/data type**

two_digit_number = input("Type a two-digit number: ") -> **Variable**

print(**type(**two_digit_number**)**) -> **Type function**

<class 'str'>

What is Subscripting (substring)?

Subscripting = **picking out characters from a string** using **square brackets** [].

Example:

word = "hello"

print(word[0]) # h

print(word[1]) # e

print(word[4]) # o

👉 **Indexing in Python** starts from **0**:

- word**[0]** = **first character**
- word**[1]** = **second character**
- and so on.

So **"39"** → is just like "hello" a **sequence of characters**.

- "39"[0] = "3"
- "39"[1] = "9"

first_digit = int(two_digit_number[0]) # "3" → 3 → **int() Function**

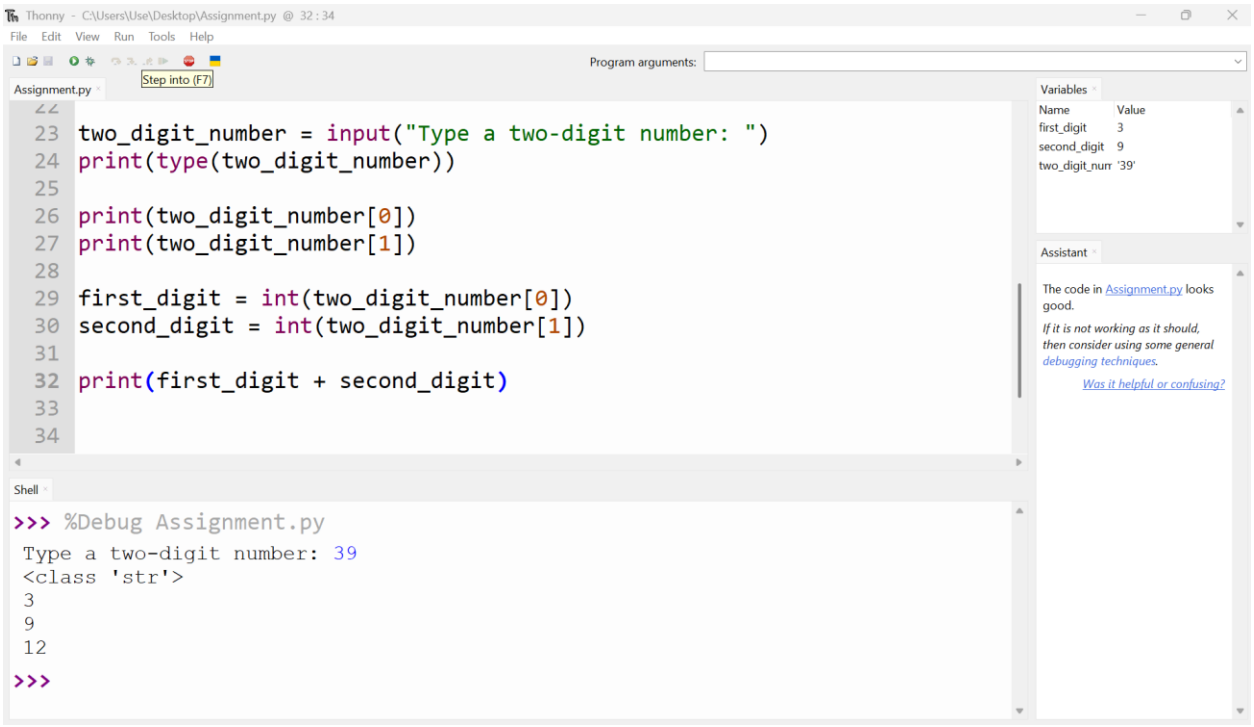
second_digit = int(two_digit_number[1]) # "9" → 9 → **int() Function**

print(first_digit + second_digit**)** # 3 + 9 = 12

✅ **Key Takeaway**

- "3" + "9" → **"39" (string concatenation)**.
- int("3") + int("9") → **12 (integer addition).**

That's why we **must use int()** when we want **digits to be added.**



Problem # 06 - Your Life in Weeks

Your Life in Weeks Program

 # Step 1: Take input

user_age = input("What is your current age?: ")

age = int(user_age)

 # Step 2: Define lifespan

max_age = 90

 # Step 3: Calculate remaining years

years_left = max_age - age

 # Step 4: Convert into days, weeks, and months

days = years_left * 365

weeks = years_left * 52

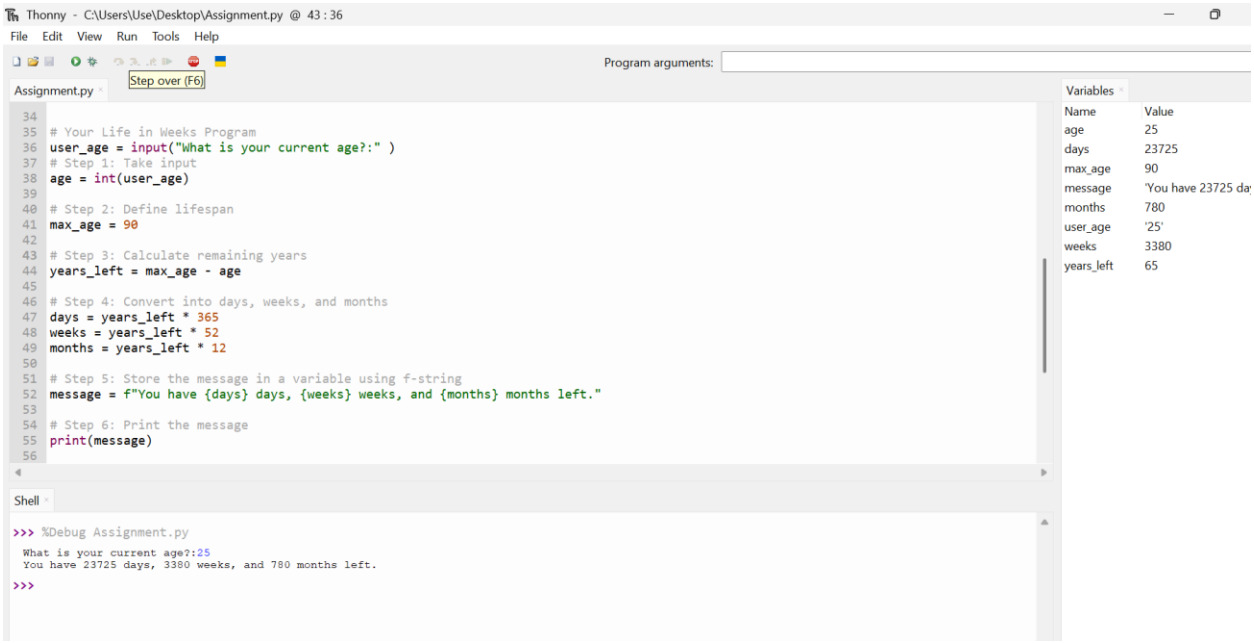
months = years_left * 12

 # Step 5: Store the message in a variable using f-string

message = f"You have {days} days, {weeks} weeks, and {months} months left.**"**

 # Step 6: Print the message

print(message)



1. What is an f-string?

- f stands for **formatted string**.
- An **f-string** is a string with the letter **f (or F) before the quotation marks.**
- It allows us to insert **variables, numbers, or calculations** inside a string easily

2. How f-strings work

- **Inside the f-string**, we use **curly brackets { }.**
- Whatever we **put inside** { } will be **evaluated** (calculated or replaced by its value).
- Anything **outside { }** is treated as **normal text** (**letters, words, symbols**).

---

3. Variables inside { }

age = 25

print(f"I am {age} years old.")

👉 Output:

I am 25 years old.

- **{age}** is replaced by the **value 25**.

---

4. Calculations/Expressions inside { }

x = 10

y = 5

print(f"{x} **+** {y} **=** {**x + y**}")

👉 Output:

10 + 5 = 15

- {x} → 10
- {y} → 5
- {**x + y**} → 15 → **Expression**
- **+** and **=** **outside { }** remain as **plain text.**

---

5. Strings inside { }

first = "Hello"

second = "World"

print(f"{**first + second**}")

👉 Output:

HelloWorld

If you want a **space**:

print(f"**{first} {second}**")

👉 Output:

Hello World

✔ Inside { } → **+** joins strings (concatenation).
✔ With numbers → + adds values.

---

6. F-strings are not only for print()

We can **store them in a variable**, **use them in files**, etc.

name = "Nadia"

marks = 95

message = f"Student {name} scored {marks} marks."

print(**message**)

👉 Output:

Student Nadia scored 95 marks.

Why f-strings are useful

- Automatically convert numbers into strings **(no need for str()).**
- **Cleaner and easier** than using + for concatenation.
- Work with **variables, numbers, calculations, and even functions**.
- Can be used anywhere a string is needed (printing, storing, writing to files).

✅ **Final Takeaway**:

An f-string in Python is a powerful way to mix text with variables or calculations. **Curly brackets { } act as placeholders** where Python inserts the actual values. **Words and symbols** are written normally outside the brackets, while numbers, variables, or expressions go inside.

📌 **Why f-strings useful here**

- Normally, **Python doesn't let you directly join text (string) with numbers (int).**

Example (❌ wrong):

age = 25

print("I am " + age + " years old.") **# This gives an error**

- You would need to convert the **number manually** → str(age).

- But **with f-strings**, Python does the conversion **automatically** inside { }.

F-strings allow us to combine letters (strings) and digits (numbers) in one sentence easily. The **number inside { } is automatically converted into text**, so **we do not need to use str() or the + operator.**

Problem # 07 - Odd or Even

number = int(input("Enter a number: "))

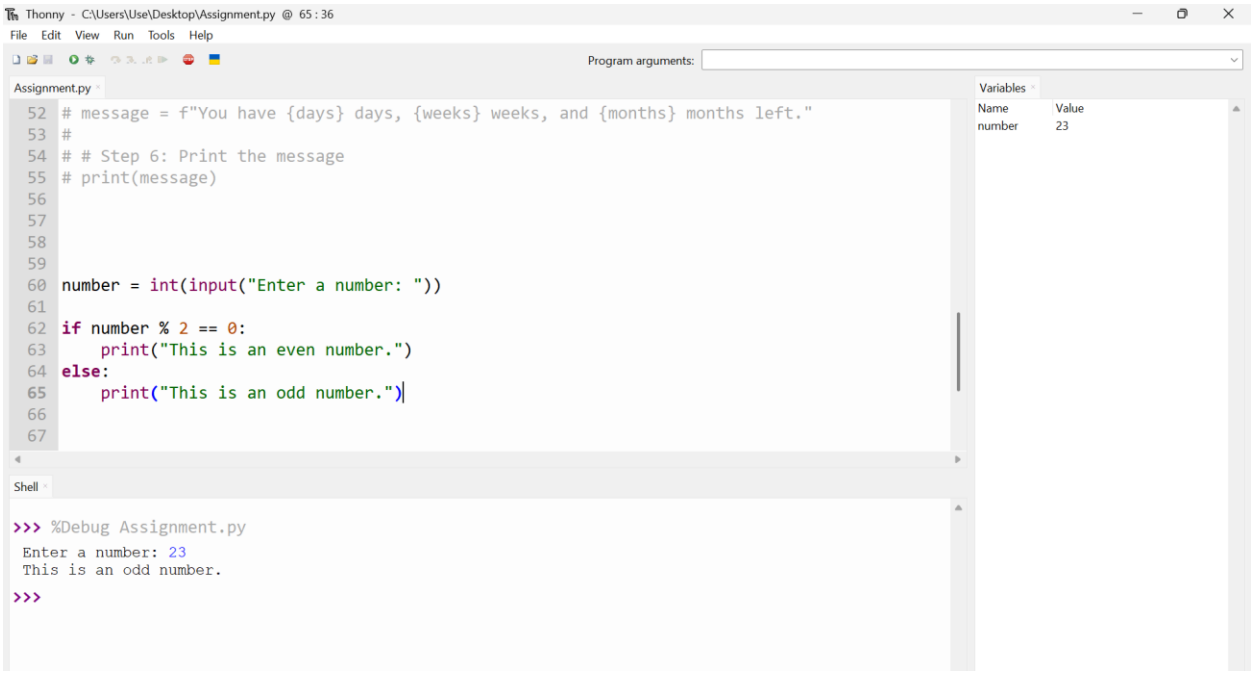
If user types 43, then:

- **input()** → "43" (string)
- **int("43")** → 43 (integer)
- Now we can do math (43 % 2).

```
if number % 2 == 0:
    print("This is an even number.")
else:
    print("This is an odd number.")
```

Examples:

```
6 % 2        # result = 0 → no remainder → even
5 % 2        # result = 1 → remainder → odd
14 % 4       # result = 2 → remainder is 2
```



Problem # 08 - BMI Calculator 2.0

weight = float(**input**("Enter your weight in kg: "))

height = float(**input** ("Enter your height in m: "))

BMI = round(weight / height**2)** **# round() -> to get int**

if BMI <= 18.5: **# Under 18.5**

print(f"Your BMI is {BMI}, you are **\033[1m**underweight**\033[0m**.")

elif BMI > 18.5 and BMI < 25: **# Over 18.5 but below 25**

print(f"Your BMI is {BMI}, you have a **\033[1m**normal weight**\033[0m**.")

elif BMI > 25 and BMI < 30: **# Over 25 but below 30**

print(f"Your BMI is {BMI}, you are **\033[1m**slightly overweight**\033[0m**.")

elif BMI > 30 and BMI < 35: **# Over 30 but below 35**

print(f"Your BMI is {BMI}, you are **\033[1m**obese**\033[0m**.")

elif BMI > 35: **# Above 35**

print(f"Your BMI is {BMI}, you are **\033[1m**clinically obese**\033[0m**.")

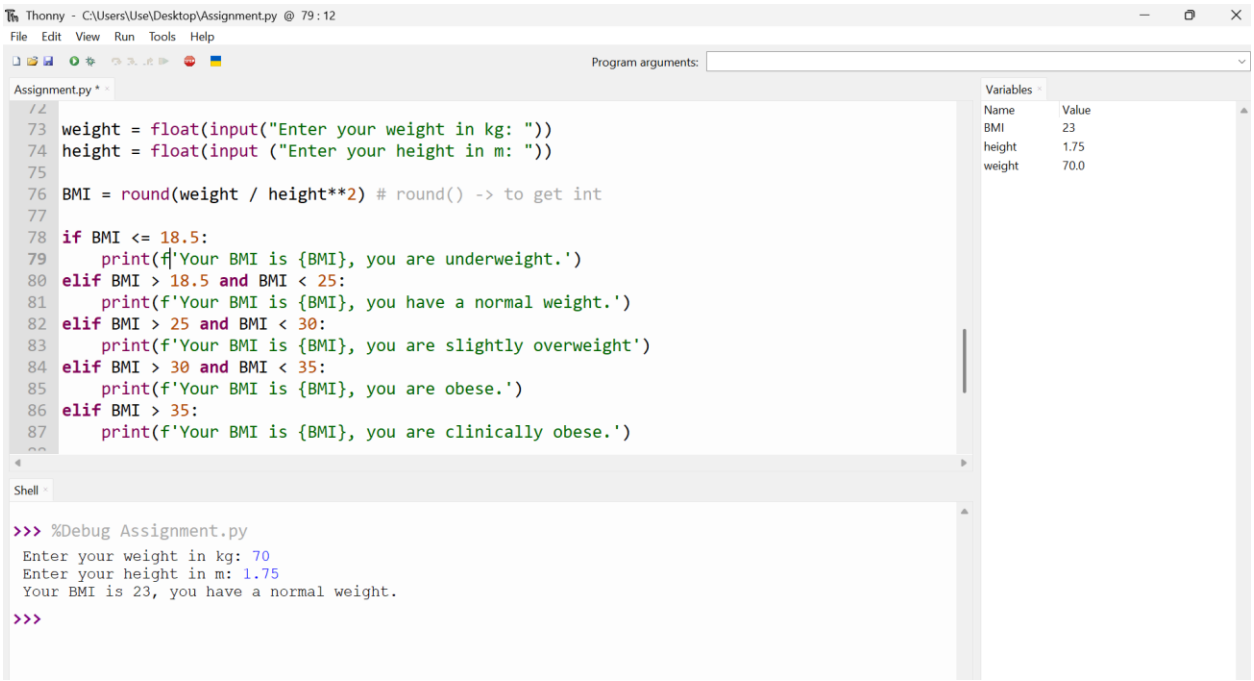
Meaning of \033[1m...\033[0m

This is called an **ANSI escape sequence**. It's a code that tells the terminal **how to format text.**

Breaking it down:

- \033 → This is the **escape character** (written in Python as \033).
- [1m → Means **turn bold ON**.
- ... → Your actual text goes here.
- \033[0m → Means **reset to normal text** (turn bold OFF).

Without Bold Interpretation:



With Bold Interpretation:

Problem # 09 - Leap Year

Leap Year Program – Deduced code structure – From Flowchart

year = 2015

if year % 4 == 0: → Is it divisible by 4?

  if year % 100 == 0: Yes → check divisibility by 100.

    if year % 400 == 0: Yes → check divisibility by 400.

      print('Leap year') Yes → leap.

    else:

      print('Not leap year') No → leap.

  else:

    print('Leap year') Yes → leap.

else:

  print('Not leap year.') No → not leap.

2nd code Structure - Leap Year Program

year = int(input('Enter year to check leap year: '))

if year % 4 == 0 **and** (year % 100 != 0 **or** year % 400 == 0):

print("Leap year.")

else:

print("Not leap year.")

Problem # 10 - Pizza Order Program

- Ask the user to choose a pizza size

.upper() ensures input is always in **uppercase (S, M, or L)**

size = input("Choose a size (S, M, or L): ").upper()

- Ask if the user wants pepperoni (Y for Yes, N for No)

add_pepperoni = input("Do you want pepperoni? (Y or N): ").upper()

- Ask if the user wants extra cheese (Y for Yes, N for No)

extra_cheese = input("Do you want extra cheese? (Y or N): ").upper()

Step 1: Start bill at 0

```
bill = 0                                # this variable will store the total price
```

Step 2: Add price based on pizza size

```
if size == "S":                         # if user chose Small
   bill += 15                           # add 15 dollars to bill
elif size == "M":                       # if user chose Medium
   bill += 20                           # add 20 dollars to bill
elif size == "L":                       # if user chose Large
   bill += 25                           # add 25 dollars to bill
else:
   print("Invalid size chosen!")        # safety check if user typed wrong
   bill = 0                             # reset bill in case of invalid input
```

Step 3: Add cost for pepperoni

```
if add_pepperoni == "Y":                # check if user wants pepperoni
   if size == "S":                      # pepperoni price depends on size
      bill += 2                         # add 2 for small pizza
   else:
      bill += 3                         # add 3 for medium or large pizza
```

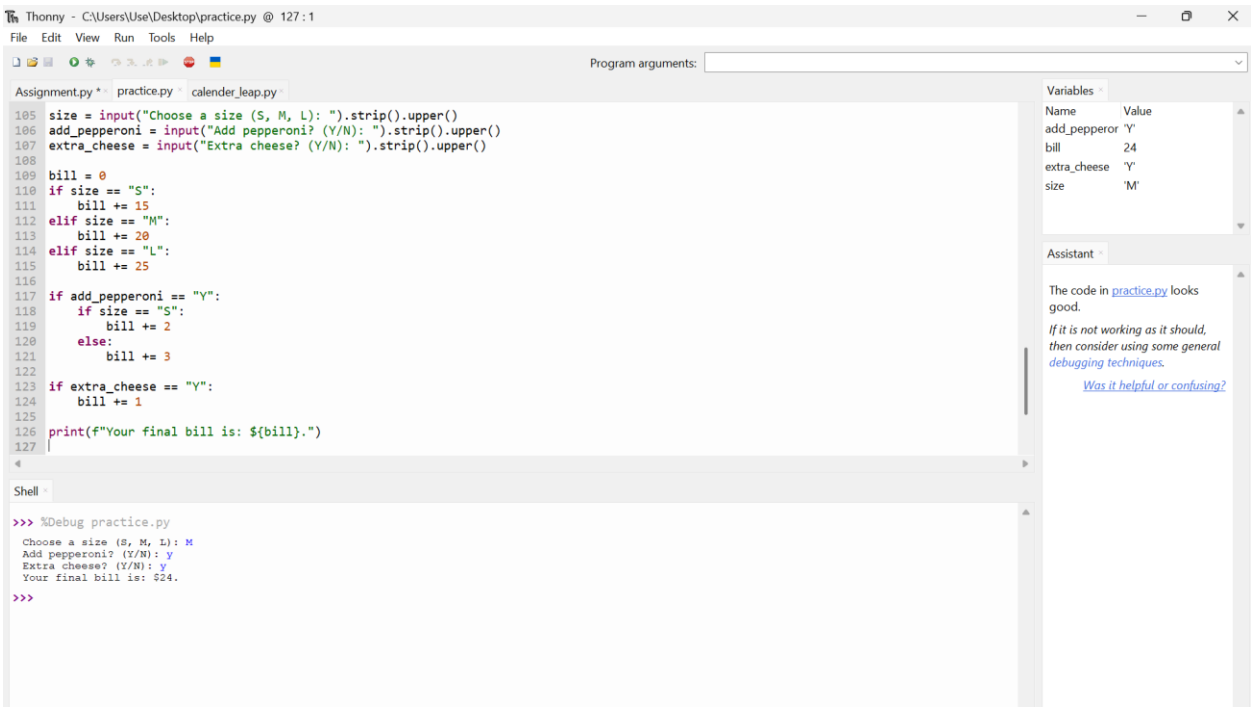
Step 4: Add cost for extra cheese

if extra_cheese == "Y": # check if user wants extra cheese

bill += 1 # always add 1 regardless of size

Step 5: Show the final bill

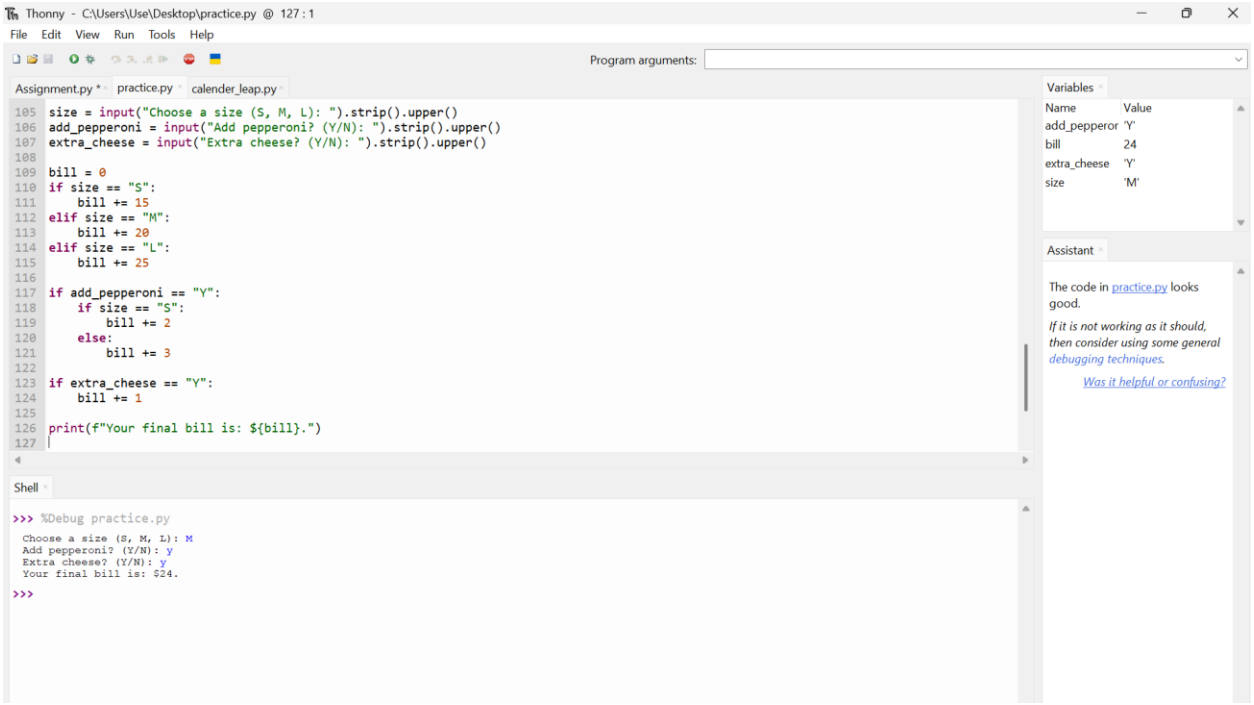
print(f"Your final bill is: ${bill}.")



size = input("Choose a size (S, M, L): ").strip().upper()

add_pepperoni = input("Add pepperoni? (Y/N): ").strip().upper()

extra_cheese = input("Extra cheese? (Y/N): ").strip().upper()



.strip() is a **string method** that removes **unwanted spaces (and invisible characters)** from the beginning and end of a string.

👉 It removes the extra spaces before and after the word, but not the ones **inside**.

Complete dictionary-based solution:

size = input("Choose a size (S, M, L): ").strip().upper()

add_pepperoni = input("Add pepperoni? (Y/N): ").strip().upper()

extra_cheese = input("Extra cheese? (Y/N): ").strip().upper()

Dictionaries for prices

base_prices = {"S": 15, "M": 20, "L": 25}

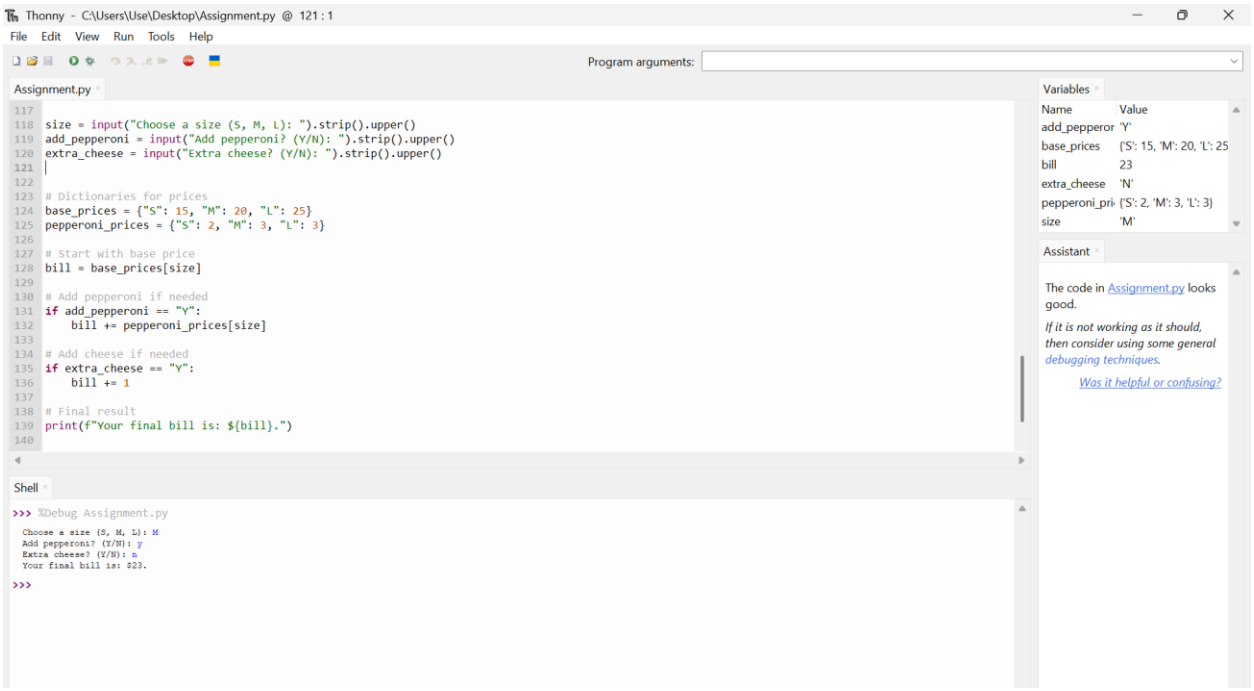
pepperoni_prices = {"S": 2, "M": 3, "L": 3}

bill = base_prices[size] **# Start with base price**

if add_pepperoni == "Y": **# Add pepperoni if needed**

bill += pepperoni_prices[size]

```
if extra_cheese == "Y":                    # Add cheese if needed
    bill += 1
print(f"Your final bill is: ${bill}.")     # Final result
```



The **square brackets []** in Python are called the **subscript operator**. They are used to **pick or access data** from **collections like lists, strings, or dictionaries**.

In this program, we use **[] with a dictionary** to get the correct base price of the pizza:

base_prices = {"S": 15, "M": 20, "L": 25}

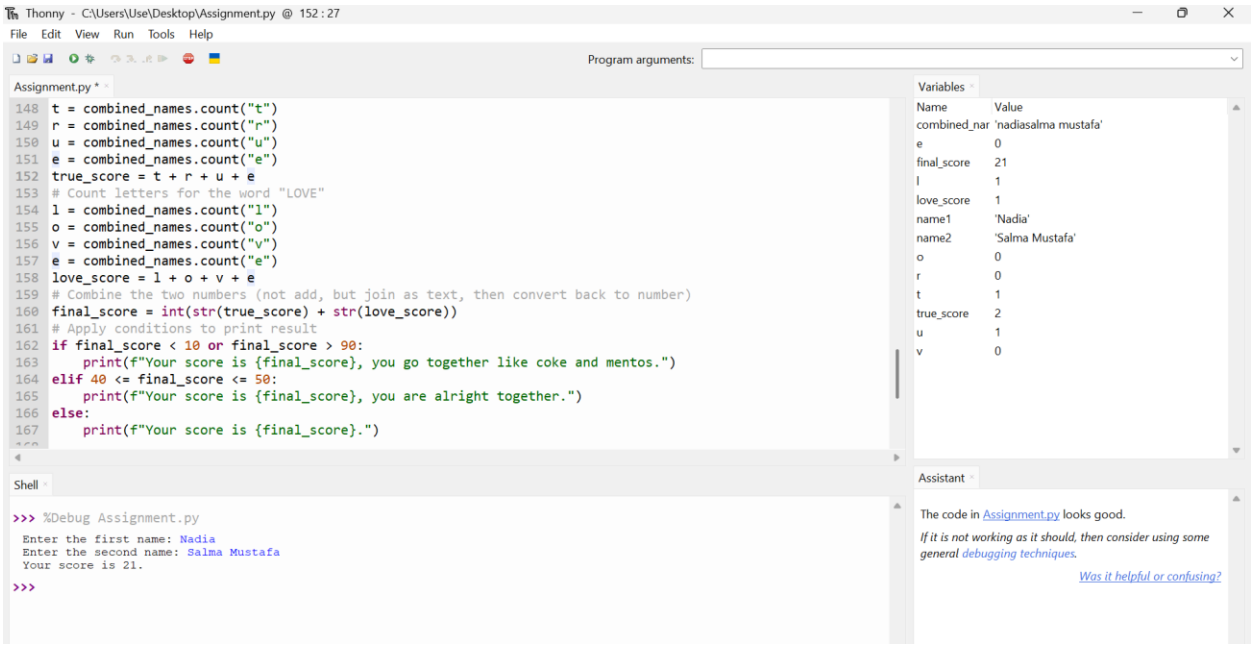
bill = base_prices[size]

If the **user chooses "L",** then base_prices[size] becomes base_prices["L"] and gives the value 25.

👉 Therefore, in this program we use [] to **look up the pizza price by its size (key)** instead of writing multiple if statements.

Problem # 11 - Love Calculator

```
name1 = input("Enter the first name: ")
name2 = input("Enter the second name: ")
combined_names = (name1 + name2).lower()
Count letters for the word "TRUE"
t = combined_names.count("t")
r = combined_names.count("r")
u = combined_names.count("u")
e = combined_names.count("e")
true_score = t + r + u + e
Count letters for the word "LOVE"
l = combined_names.count("l")
o = combined_names.count("o")
v = combined_names.count("v")
e = combined_names.count("e")
love_score = l + o + v + e
Combine the two numbers (not add, but join as text, then convert back to number)
final_score = int(str(true_score) + str(love_score))
Apply conditions to print result
if final_score < 10 or final_score > 90:
    print(f"Your score is {final_score}, you go together like coke and mentos.")
elif 40 <= final_score <= 50:
    print(f"Your score is {final_score}, you are alright together.")
else:
    print(f"Your score is {final_score}.")
```



The **count() function** tells us **how many times a certain letter** appears in a string. We use it because instead of checking letters manually one by one, count() does the work for us **automatically.**

Why Concatenate Instead of Add?

👉 **"Combine"** here means **putting digits side by side**, not adding them.

Example to See the Difference

Take names: "Kim alish" + "Jack Bauer"

- TRUE score = **3**
- LOVE score = **2**

If we add:

3 + 2 = 5

Love Score = 5 ❌ (just **one digit**, doesn't match rules)

If we concatenate:

"3" + "2" = "32"

Convert back to number → 32 ✅ (a proper **two-digit score**)

How Concatenation Works

final_score = int(str(true_score) + str(love_score))

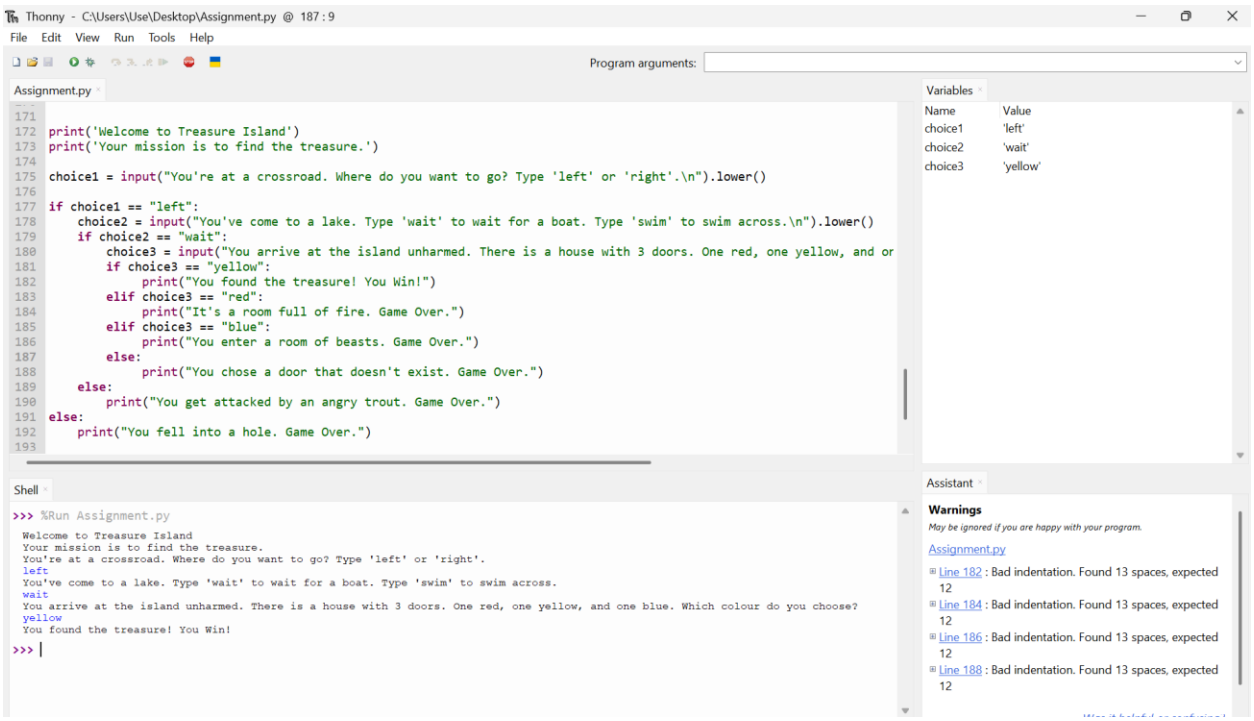
- true_score = 3 → str(true_score) = "3"
- love_score = 2 → str(love_score) = "2"
- "3" + "2" = "32" (string concatenation)
- int("32") = 32 (final number)

✅ **That's why concatenation is required**: it creates a "two-digit love score" **instead of just a small sum**.

Problem # 12 - Treasure Island

```
print("Welcome to Treasure Island.")
print("Your mission is to find the treasure.")          # Introduction
# Step 1: Crossroad
choice1 = input("You're at a crossroad. Where do you want to go? Type 'left' or 'right'.\n").lower()
if choice1 == "left":
  # Step 2: Lake
  choice2 = input("You've come to a lake. Type 'wait' to wait for a boat. Type 'swim' to swim across.\n").lower()
  if choice2 == "wait":
   # Step 3: Doors
    choice3 = input("You arrive at the island unharmed. There is a house with 3 doors: red, yellow, and blue. Which colour do you choose?\n").lower()
    if choice3 == "yellow":
      print("You found the treasure! 🏆 You Win!")
    elif choice3 == "red":
      print("It's a room full of fire 🔥. Game Over.")
```

```
    elif choice3 == "blue":
        print("You enter a room of beasts 🐲. Game Over.")
    else:
        print("You chose a door that doesn't exist. 🚪 Game Over.")
else:
    print("You get attacked by an angry trout 🐟. Game Over.")
else:
    print("You fell into a hole 🕳. Game Over.")
```



What \n Means

- **\n** is called a **newline character** in Python (and many programming languages).
- It tells Python: **“move the cursor to the next line”** when printing text for better readability.

- In this game, it makes the prompts clear, so the **player's answer doesn't mix with the question.**