

## Laboratory Activity No. 5

### Functions, Modules, and Packages

**Course Code:** CPE103

**Program:** BSCPE

**Course Title:** Object-Oriented Programming

**Date Performed:** 02/15/25

**Section:** 1-A

**Date Submitted:** 02/15/25

**Name:** Filjohn B. Delinia

**Instructor:** Engr. Maria Rizette Sayo

#### 1. Objective(s):

This activity aims to introduce students to the concept of modules and packages in Python.

#### 2. Intended Learning Outcomes (ILOs):

The students should be able to:

2.1 Use the different Python built-in functions, modules, and packages.

2.2 Create a program with user-defined functions

2.3 Create a program that will import or load a user-created module and import its functions.

2.4 Create a program that will import or load a user-created package and import its modules.

#### 3. Discussion:

As the program gets larger and more complex, it will be unavoidable for a programmer to split related and repeated codes into separate files, this in Python is referred to as Modules and Packages.

A **module** contains variables and functions that can be imported or used again in another program (Python file) without the programmer having to code the same variables and functions again. A python program (.py file) is considered to be a module. A **package** is simply a group of related modules or .py files combined together. A package commonly is considered to be a library which contains an enormous amount of functions and modules.

Python comes with built-in modules and packages such as the **math** module, **statistics** module, **random** module. For scientific computing, the following packages is mentioned here. The **NumPy** package is a general-purpose array-processing package. It provides a high-performance multidimensional array object. The **NumPy** package is used for scientific computing. Another well-known Python package is **Scikit-learn** which is an open source machine learning library that supports supervised and unsupervised learning for implementing Artificial Intelligence related tasks. For Software Development, the **Tkinter** and the **PyQt5** are the most commonly used. For Web Development, **Flask** and **Django** are the most widely used framework(composed of packages) for building websites with Python.

#### Functions

To implement a simple module or package, we need to refamiliarize ourselves with the concept of functions. Recall that a function is composed of a block of codes stored together that when called it executes those codes stored inside it. Functions help programmers organize related code into modular pieces that can be called upon to perform repetitive tasks. The Python interpreter has a number of functions and types built into it that are always available but programmers can add their own custom functions to implement more customized logic. The image below shows an example of a function written in Python.

```
def function_name(parameter1,parameter2):  
    # code 1  
    # code 2  
    # ...  
    # code n  
    return value # optional return
```

The def is a special keyword used to indicate a function definition or creation. The function\_name is the name of the function to be created. Recall that a function is also declared using parenthesis. Inside the parenthesis are inputs or arguments that can be accepted in the function through the declaration of parameters. Parameters are temporary variables or placeholder that can

be used inside the function. In Python, a return value is optional since the interpreter automatically makes the decision if it will be a void datatype function, an int datatype function, string datatype function, boolean datatype function and adjusts memory allocation accordingly.

Functions can be built-in or user-defined, for instance the Python print() is an example of a built-in function. The len() is also a built-in function. Other examples of built-in functions are: int(), str(), float(), bool(), list(), tuple(), dict(), open(). Functions will be used in the activity to create modules and packages. More built-in functions can be found in the official Python documentation.

## Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

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

Source: <https://docs.python.org/3/library/functions.html>

In this activity, you will be exploring the various built-in Python functions, modules, and packages, and how to create functions in Python, put those functions in a module, call functions from a module to another program, and create your own package of modules.

For more information you may also visit the official python documentation:

<https://docs.python.org/3/tutorial/modules.html>

<https://docs.python.org/3/tutorial/modules.html#packages>

### 4. Materials and Equipment:

Desktop Computer with Anaconda Python or Google Colab  
Windows Operating System

## 5. Procedure:

For this activity we will be creating Python programs using the Spyder IDE.

### Functions

#### Exploring built-in functions

1. Create a folder named **builtinfunctions**
2. Create a Python file inside the functions1 folder named **evaluator.py** and copy the code shown below:

```
# Propositional Logic evaluator for discrete math for 2-3 variables
print("Propositional logic evaluator for discrete math")
variables = int(input("How many variables? "))
total_combinations = 2**variables

combinations_list = [] # store all the possible combinations

# generate the combinations
for i in range(total_combinations):
    bin_equivalent = bin(i)[2:]
    while len(bin_equivalent) < variables:
        bin_equivalent = "0" + bin_equivalent
    combinations_list.append(tuple(int(val) for val in bin_equivalent))
    # this will generate a list with values [(0,0),(0,1),(1,0),(1,1)]
    # for two variables

# main program
expression = input("Enter the propositional logic expression: ")
# note: Only the letters A,B, and C are allowed to be used
# example: not(A and B) or (A and C)
if variables == 2:
    print("A B f")
    for A,B in combinations_list:
        evaluated_expression = eval(expression)
        print(A,B, evaluated_expression)
elif variables == 3:
    print("A B C f")
    for A,B,C in combinations_list:
        evaluated_expression = eval(expression)
        print(A,B,C, evaluated_expression)
```

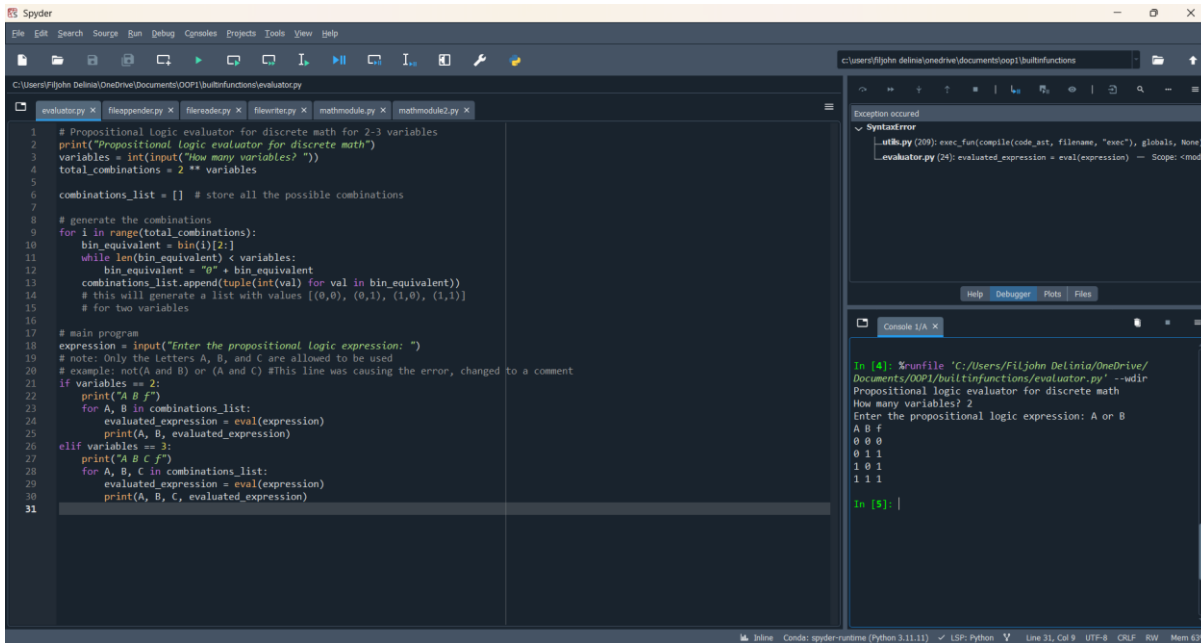
**Note:** and, or, not must be small cases.

3. Run the program and observe the output. Try to analyze the purpose of the built-in functions used in the program (the keywords in color violet).
4. You may modify the code in order to study it as it will be used later in the next sections of the activity.

### OBSERVATION:

The program is a **Propositional Logic Evaluator** that evaluates logical expressions for 2 or 3 variables (A, B, and optionally C). It generates all possible truth value combinations (0 and 1) for the variables, evaluates the user-input expression using `eval()`, and outputs a truth table. Key functions like `int()`, `input()`, `bin()`, and `eval()` are used to handle input, generate combinations, and evaluate expressions. While functional, the program assumes valid input and could be enhanced with input validation, error handling, and support for more variables. The use of `eval()` is efficient but requires caution due to potential security risks. Overall, it effectively demonstrates dynamic evaluation of propositional logic content.

## OUTPUT:



The screenshot shows the Spyder IDE interface. The main editor displays a Python script named `evaluator.py` located at `C:\Users\Filjohn Delinia\OneDrive\Documents\OOP1\builtinfunctions\evaluator.py`. The script is a propositional logic evaluator for discrete math, designed for 2-3 variables. It generates all possible combinations of variable values and evaluates a given logical expression for each combination. The script includes comments and a main program section that prompts the user for an expression and the number of variables. The console output shows the execution of the script with the following steps:

```
In [4]: %runfile 'C:/Users/Filjohn Delinia/OneDrive/
Documents/OOP1/builtinfunctions/evaluator.py' --wdir
Propositional logic evaluator for discrete math
How many variables? 2
Enter the propositional logic expression: A or B
A B f
0 0 0
0 1 1
1 0 1
1 1 1

In [9]: |
```

## Using the open() function for file handling

1. Create a folder named **filehandling** outside of **builtinfunctions** folder
2. Create a Python file inside the **filehandling** folder named **filewriter.py**
3. Open the **filehandler.py** using Spyder IDE and type the code as shown below:

```
name = "Royce Chua"
file = open("newfile1.txt", 'w')
file.write(f"Hello, {name}!\n")
file.write("Isn't this amazing!\n")
file.write("that we can create and write on text files\n")
file.write("using Python.")
file.close()
```

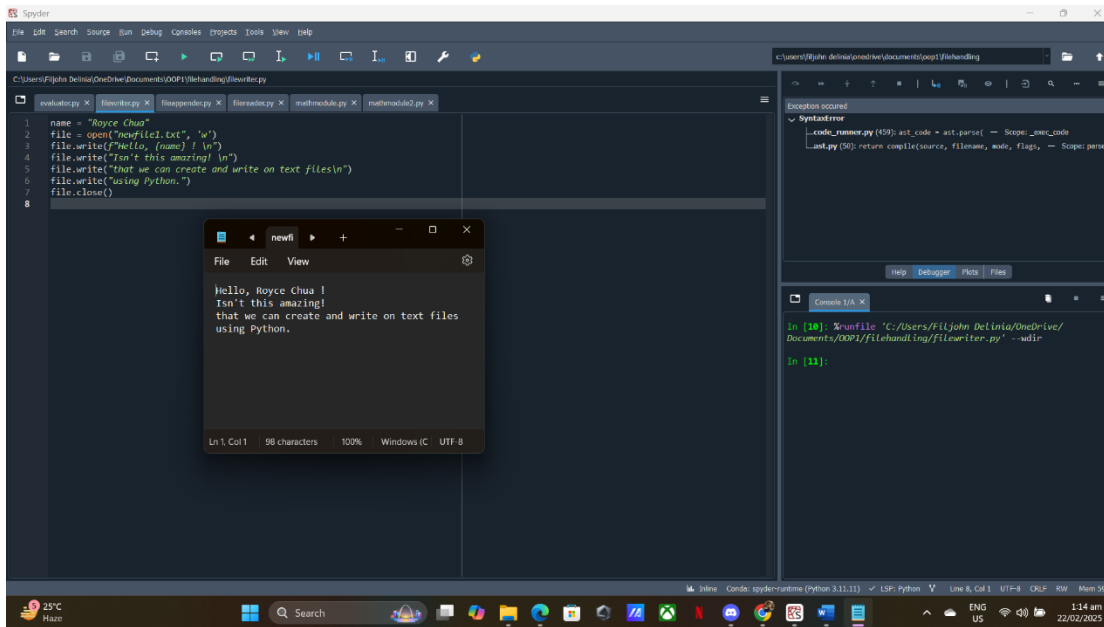
**Note:** You may use `help(open)` either in a Python program or in the shell for more information about the built-in function.

- Run the program and observe the output.

OBSERVATION:

After running this program, a file named `newfile1.txt` will be created in the same directory as `filewriter.py` with the written content.

OUTPUT:



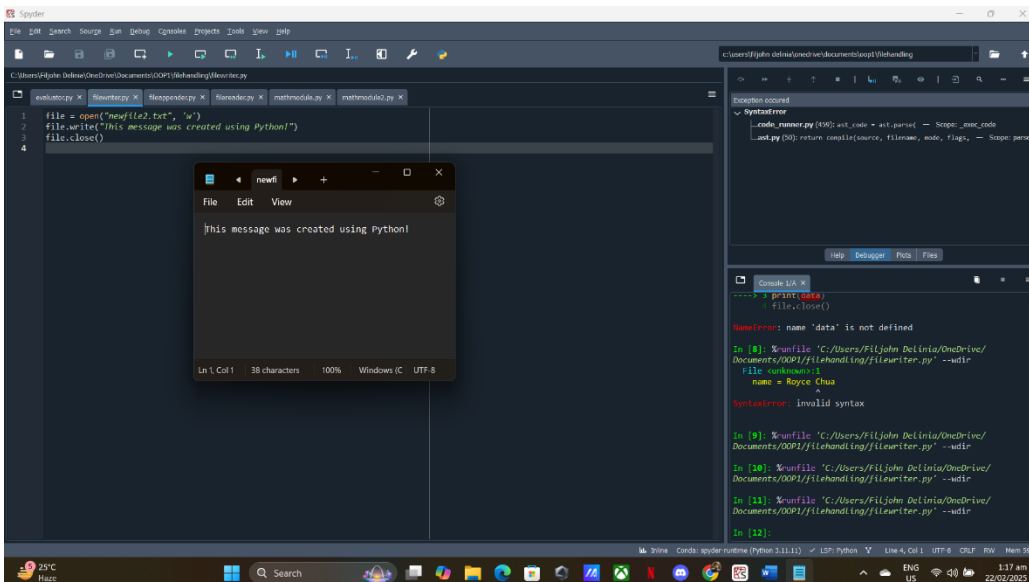
```
1 name = "Royce Chua"
2 file = open("newfile1.txt", "w")
3 file.write("Hello, (name) \n")
4 file.write("Isn't this amazing! \n")
5 file.write("that we can create and write on text files\n")
6 file.write("using Python.")
7 file.close()
```

newfile1.txt

Hello, Royce Chua!  
Isn't this amazing!  
that we can create and write on text files  
using Python.

- Modify the program to create a file called `newfile2.txt` and print the message (excluding the “`”) “This message was created using Python!”`

OUTPUT:



```
1 file = open("newfile2.txt", "w")
2 file.write("This message was created using Python!")
3 file.close()
```

newfile2.txt

This message was created using Python!

- Create another Python file inside the `filehandling` folder named `filereader.py` and type the code as shown below:

```

file = open("new.txt",'r')
data = file.read()
print(data)
file.close()

```

7. It should display an error. Identify and resolve the cause of the error based on the message given by Python. The file that should be read is newfile1.
8. After fixing the error, run the program again and observe the output.
9. Modify the filereader.py program so that the message of **newfile2.txt** is displayed.
10. Modify again the program with the following code below:

```

file = open("newfile2.txt",'r')
data = file.read(12)
print(data)
file.close()

```

11. Create a new Python file still inside the oop1 folder called **fileappender.py** and copy the code shown below:

```

file = open("newfile2.txt",'a')
file.write("and also by the programmer of course. ")
file.close()

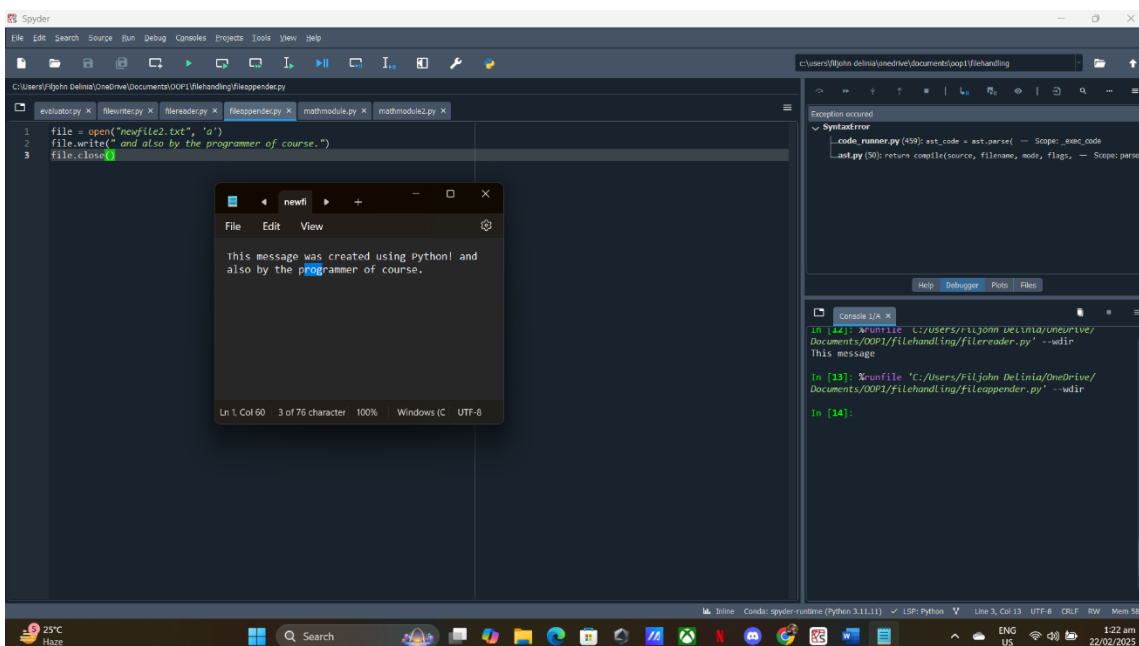
```

12. Run the program and observe the output.

## OBSERVATION:

It appends in the text " and also by the programmer of course." to newfile2.txt.

## OUTPUT:



## User-defined Functions

1. Create a new folder called **userfunctions** outside of **filehandling** folder
2. In the **userfunctions** folder create a program called **truthtablegenerator.py** and copy the code below:

```
def generate_truthtable(number_of_variables):
    total_combinations = 2**number_of_variables
    combinations_list = []
    for i in range(total_combinations):
        bin_equivalent = bin(i)[2:]
        while len(bin_equivalent) < number_of_variables:
            bin_equivalent = "0" + bin_equivalent
        combinations_list.append(tuple(int(val) for val in bin_equivalent))
    return combinations_list

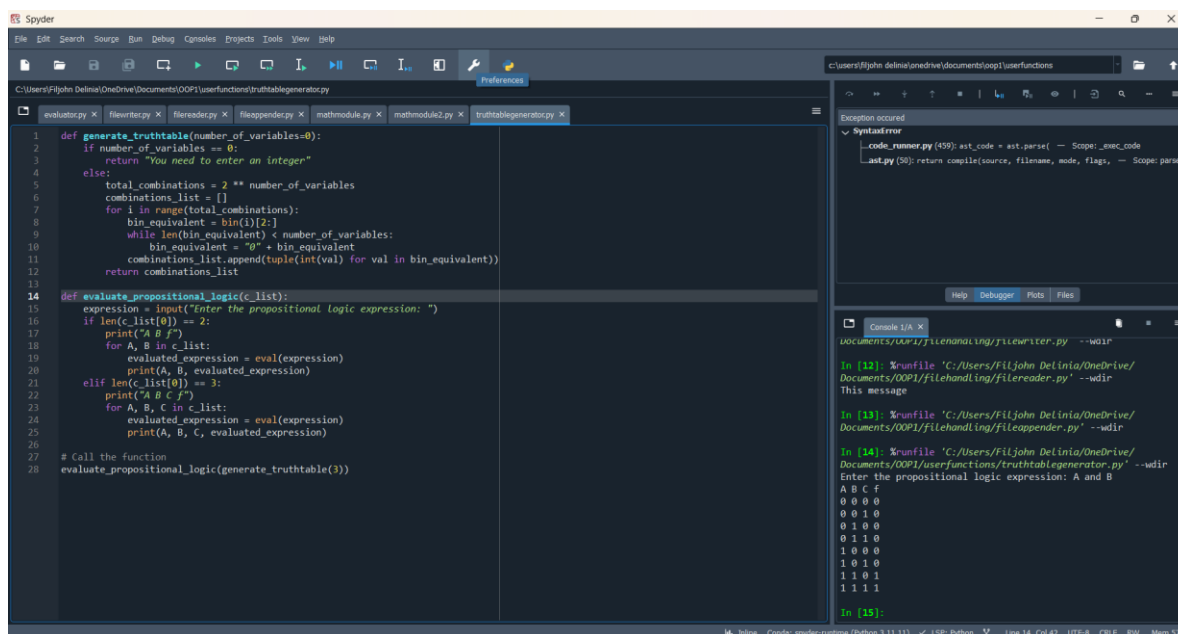
print(generate_truthtable(3))
```

3. Run the program and observe the output.

## OBSERVATION:

The script generates a truth table for a given number of variables and evaluates a user-provided propositional logic expression. It displays all possible input combinations alongside the computed truth values, effectively simulating a truth table.

## OUTPUT:



```
def generate_truthtable(number_of_variables):
    total_combinations = 2**number_of_variables
    combinations_list = []
    for i in range(total_combinations):
        bin_equivalent = bin(i)[2:]
        while len(bin_equivalent) < number_of_variables:
            bin_equivalent = "0" + bin_equivalent
        combinations_list.append(tuple(int(val) for val in bin_equivalent))
    return combinations_list

def evaluate_propositional_logic(c_list):
    expression = input("Enter the propositional logic expression: ")
    if len(c_list[0]) == 2:
        print("A B F")
        for A, B in c_list:
            evaluated_expression = eval(expression)
            print(A, B, evaluated_expression)
    elif len(c_list[0]) == 3:
        print("A B C F")
        for A, B, C in c_list:
            evaluated_expression = eval(expression)
            print(A, B, C, evaluated_expression)
    # Call the function
    evaluate_propositional_logic(generate_truthtable(3))
```

Exception occurred

code\_runner.py (49): est\_code = est.parse( - Scope: parse  
...ast.py (50): return compile(source, filename, mode, flags, - Scope: parse

Help | Debugger | Plots | Files

Console I/O X

Documents\user\filehandling\filewriter.py --wdir

In [12]: %runfile 'C:/Users/Fljohn Delinia/OneDrive/Documents/OOP1/filehandling/filereader.py' --wdir  
This message

In [13]: %runfile 'C:/Users/Fljohn Delinia/OneDrive/Documents/OOP1/filehandling/fileappender.py' --wdir

In [14]: %runfile 'C:/Users/Fljohn Delinia/OneDrive/Documents/OOP1/userfunctions/truthtablegenerator.py' --wdir  
Enter the propositional logic expression: A and B

```
A B C F
0 0 0 0
0 0 1 0
0 1 0 0
0 1 1 0
1 0 0 0
1 0 1 0
1 1 0 1
1 1 1 1
```

In [15]:

4. Modify the program by changing `print(generate_truthtable(3))` to `print(generate_truthtable())` then run the program

5. An error should occur, modify the program according to the code shown below:

```
def generate_truthtable(number_of_variables=0):  
    if number_of_variables == 0:  
        return "You need to enter an integer"  
    else:  
        total_combinations = 2**number_of_variables  
        combinations_list = []  
        for i in range(total_combinations):  
            bin_equivalent = bin(i)[2:]  
            while len(bin_equivalent)<number_of_variables:  
                bin_equivalent="0"+bin_equivalent  
            combinations_list.append(tuple(int(val) for val in bin_equivalent))  
        return combinations_list
```

**Note:** The code modifications are underlined in red.

6. In the same file/program, create a new function with the name **evaluate\_propositional\_logic()**. The parameter is **c\_list**(combinations list) and the code can be found under the **# main program** comment in the first program made earlier.
7. After successfully placing the code in the function, call the function using this code  
**evaluate\_propositional\_logic(generate\_truthtable(3))**
8. Analyze why in the **generate\_truthtable** function we needed to print the function whereas in the **evaluate\_propositional\_logic** function, it prints the values on its own.
9. Compare the program **truthtablegenerator.py** with **evaluator.py**. Identify the advantages of placing code within functions against the sequential code done in the first.

#### COMPARE:

The **generate\_truthtable** function returns a list for flexibility, while **evaluate\_propositional\_logic** prints directly to display results. Using functions in **evaluator.py** improves modularity, reusability, maintainability, and scalability compared to the sequential approach in **truthtablegenerator.py**, making the program more efficient and adaptable.



## Modules

### Built-in Modules

#### math module

1. Create a new folder called **modules1** outside of the **userfunctions**
2. Create a new Python file called **mathmodule.py** and copy the following codes as shown below:

```
import math

def quadratic_formula(a,b,c):
    if b**2-(4*a*c)<0: # involving imaginary numbers
        x1 = (complex(-b,math.floor(math.sqrt(abs(b**2-(4*a*c))))))/2*a
        x2 = (complex(-b,-1*math.floor(math.sqrt(abs(b**2-(4*a*c))))))/2*a
        return x1, x2
    else:
        x1 = (-b+math.sqrt(b**2-(4*a*c)))/(2*a)
        x2 = (-b-math.sqrt(b**2-(4*a*c)))/(2*a)
        return x1, x2

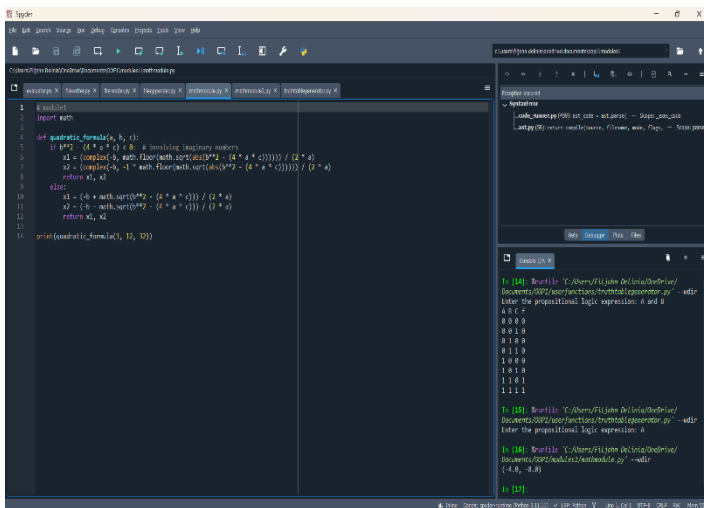
#print(quadratic_formula(1,12,32))
print(quadratic_formula(1,2,3))
```

3. Run the program and observe the output. You may switch between the two sets of values.

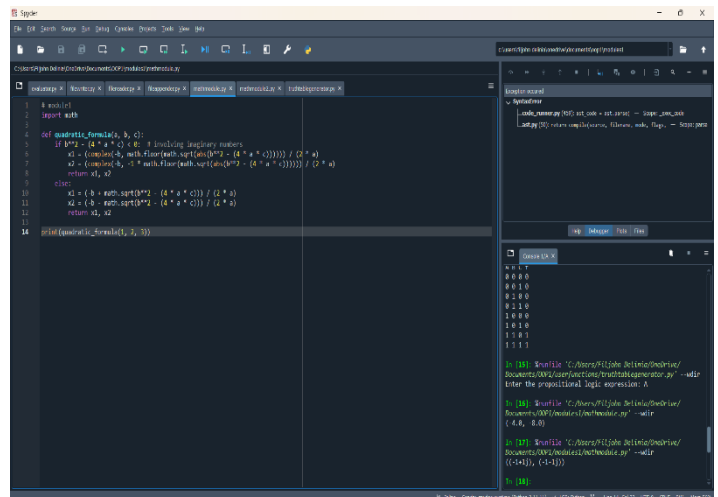
#### OBSERVATION:

Running the program produces complex roots  $(-1+1j)$  and  $(-1-1j)$  for `quadratic_formula(1, 2, 3)`, as the discriminant is negative. For `quadratic_formula(1, 12, 32)`, it outputs real roots  $-4.0$  and  $-8.0$  due to a positive discriminant. The program correctly differentiates between real and complex solutions, though its handling of imaginary numbers could be refined.

#### OUTPUTS:



```
Python Shell
In [10]: print(quadratic_formula(1, 2, 3))
Out[10]: (-1+1j, -1-1j)
```



```
Python Shell
In [10]: print(quadratic_formula(1, 12, 32))
Out[10]: (-4.0, -8.0)
```

4. Create a new Python file called **mathmodule2.py** and copy the following codes as shown below:

```
import math

def angle_demo():
    angle = math.sin(math.pi/2) # the default input is in radians
    # angle sin(90)=1 in degrees == sin(pi/2)=1 in radians
    print(angle)
    # to make it convenient, convert to radians
    angle = math.sin(math.radians(90))
    print(angle)
    # this is also similar for cosine and other trigonometric and
    # hyperbolic functions

angle_demo()
```

5. To view additional functions in the module. Type and run `help(math)` while it is imported.

#### time and datetime module

1. Create a new file in the folder named **dateandtime.py**
2. Copy and run the code as show below:

```
import time

def pause():
    for i in range(10,0,-1):
        print(f"The program will end in {i}..")
        time.sleep(1)

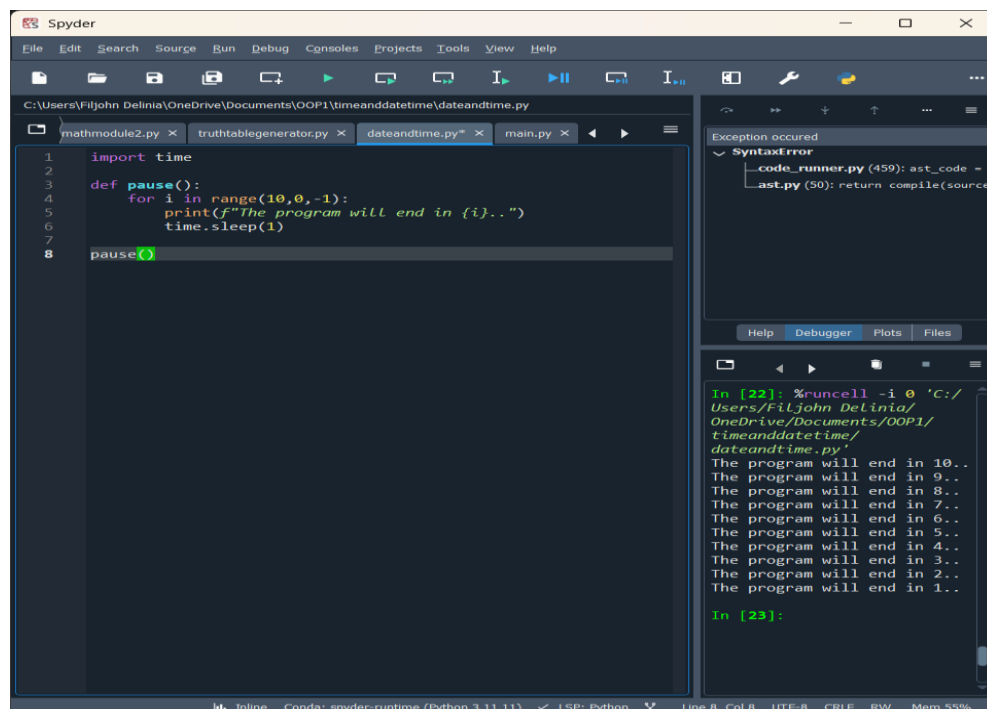
pause()
```

3. Observe the output.

#### OBSERVATION:

The program displays a countdown from 10 to 1, pausing for one second between each number, and then ends after approximately 10 seconds.

#### OUTPUT:



4. In the same file, copy and add the code to the file as shown below:

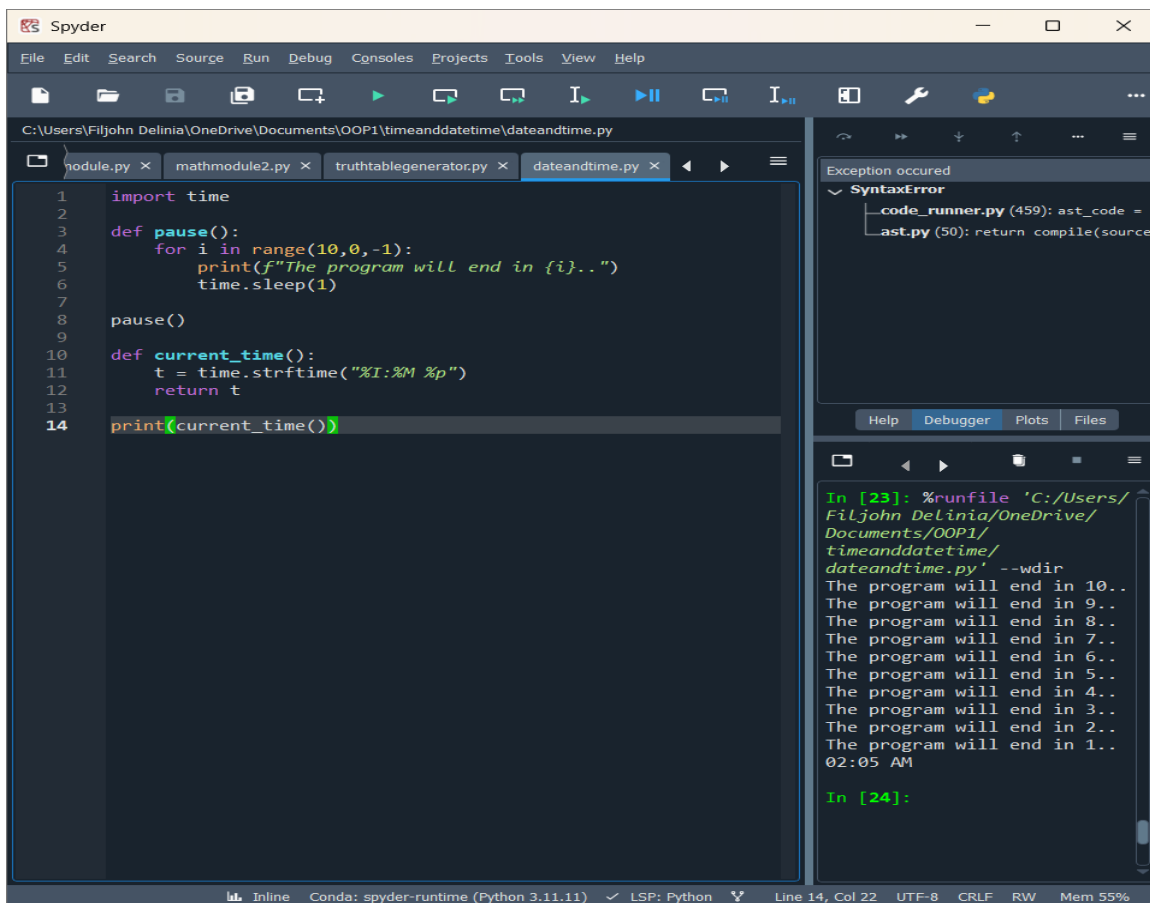
```
def current_time():  
    t = time.strftime("%I:%M %p")  
    return t  
  
print(current_time())
```

5. Run the program, and observe the output.

#### OBSERVATION:

The program first performs a 10-second countdown, displaying messages from 10 to 1 with a one-second pause between each. After the countdown, it prints the current time in a 12-hour format with AM/PM.

#### OUTPUT:



```
1 import time  
2  
3 def pause():  
4     for i in range(10,0,-1):  
5         print(f"The program will end in {i}..")  
6         time.sleep(1)  
7  
8 pause()  
9  
10 def current_time():  
11     t = time.strftime("%I:%M %p")  
12     return t  
13  
14 print(current_time())
```

Exception occurred  
SyntaxError  
code\_runner.py (459): ast\_code = :  
ast.py (50): return compile(source

```
In [23]: %runfile 'C:/Users/  
Filjohn Delinia/OneDrive/  
Documents/OOP1/  
timeanddatetime/  
dateandtime.py' --wdir  
The program will end in 10..  
The program will end in 9..  
The program will end in 8..  
The program will end in 7..  
The program will end in 6..  
The program will end in 5..  
The program will end in 4..  
The program will end in 3..  
The program will end in 2..  
The program will end in 1..  
02:05 AM  
  
In [24]:
```

6. In the same file, copy and add the code to the file as shown below:

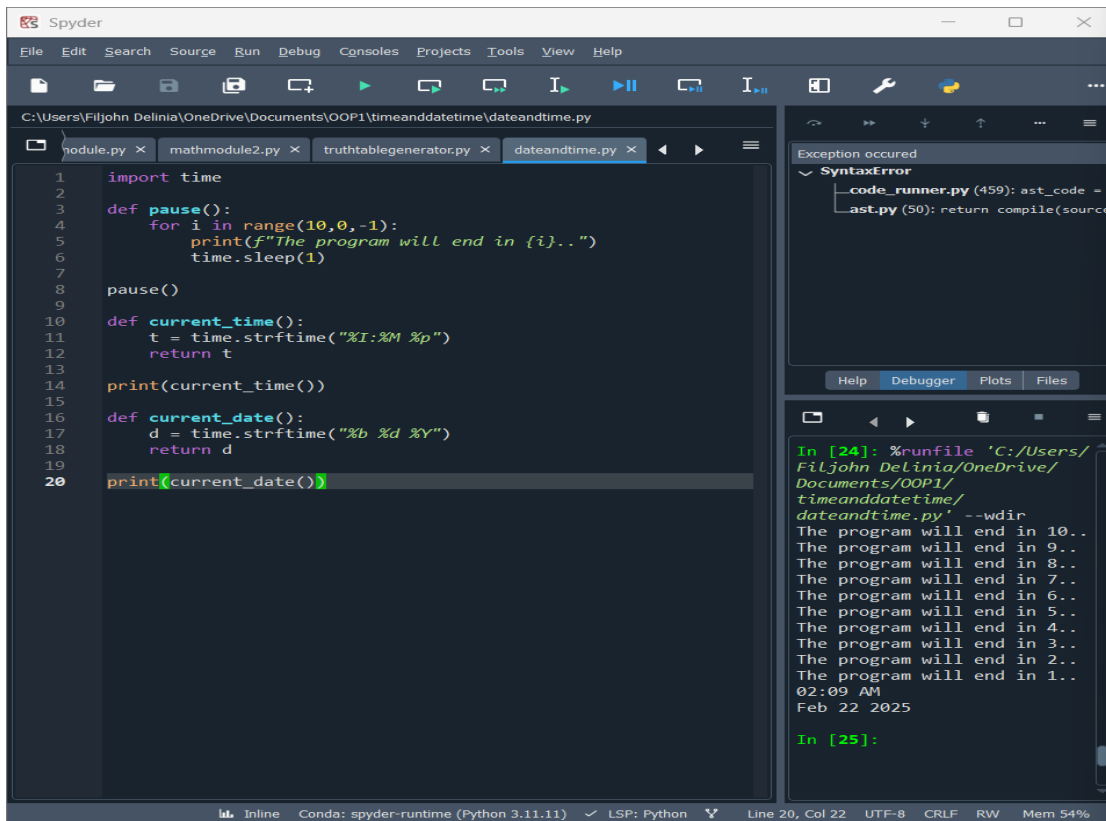
```
def current_date():  
    d = time.strftime("%b %d %Y")  
    return d  
  
print(current_date())
```

7. Run the program, and observe the output.

## OBSERVATION:

The program first runs a 10-second countdown, displaying messages from 10 to 1 with one-second pauses. After the countdown, it prints the current time in a 12-hour AM/PM format, followed by the current date in the Month Day Year format.

## OUTPUT:



The screenshot shows the Spyder Python IDE interface. The main editor window displays a Python script named `dateandtime.py` located at `C:\Users\Filjohn Delinia\OneDrive\Documents\OOP1\timeanddatetime\dateandtime.py`. The script contains the following code:

```
1 import time
2
3 def pause():
4     for i in range(10,0,-1):
5         print(f"The program will end in {i}..")
6         time.sleep(1)
7
8 pause()
9
10 def current_time():
11     t = time.strftime("%I:%M %p")
12     return t
13
14 print(current_time())
15
16 def current_date():
17     d = time.strftime("%b %d %Y")
18     return d
19
20 print(current_date())
```

The right-hand pane shows the IPython console with the following output:

```
In [24]: %runfile 'C:/Users/Filjohn Delinia/OneDrive/
Documents/OOP1/
timeanddatetime/
dateandtime.py' --wdir
The program will end in 10..
The program will end in 9..
The program will end in 8..
The program will end in 7..
The program will end in 6..
The program will end in 5..
The program will end in 4..
The program will end in 3..
The program will end in 2..
The program will end in 1..
02:09 AM
Feb 22 2025

In [25]:
```

The bottom status bar indicates the current position is Line 20, Col 22, using UTF-8 encoding, CRLF line endings, and RW permissions, with 54% memory usage.

### User-defined Modules

1. The previously created dateandtime.py is considered to be a module that you can import.
2. In the same **modules1** folder, create a new file called main.py and copy the following code:

```
import dateandtime
```

```
print("The current time is",dateandtime.current_time())
```

3. The program will not run as expected, and you will need to remove the following codes in the dateandtime.py which are underlined in red.

```

import time

def pause():
    for i in range(10,0,-1):
        print(f"The program will end in {i}..")
        time.sleep(1)

#pause()

def current_time():
    t = time.strftime("%I:%M %p")
    return t

#print(current_time())

def current_date():
    d = time.strftime("%b %d %Y")
    return d

#print(current_date())

```

4. Run the main.py program again and you will now see the correct output which is the current time.
5. Modify the main.py to also display the current date using the current\_date() in dateandtime.py
6. To remove the need to constantly indicate the module name dateandtime. in each function, modify the code as shown below:

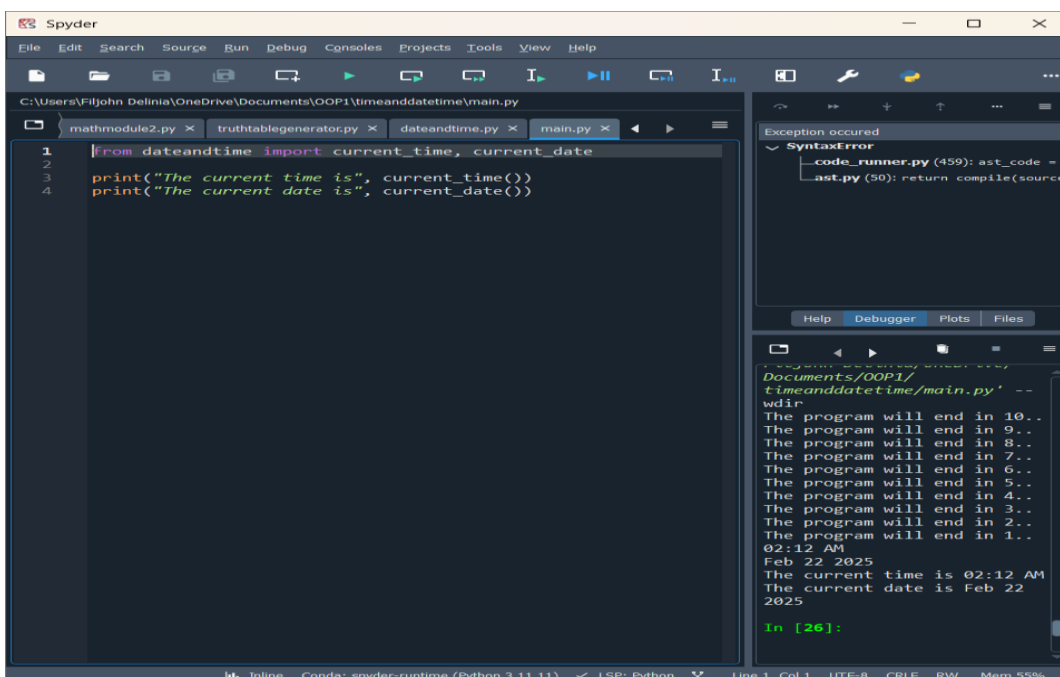
```

from dateandtime import current_time, current_date

print("The current time is",current_time())
print("The current date is",current_date())

```

## OUTPUT:



## 6. Supplementary Activity:

### Tasks

#### Simple Word Filter

1. Create a function that would accept two inputs: a sentence(string), and a list containing bad words that the user would like to censor but not remove. The function should return the newly filtered sentence wherein the bad words are replaced with asterisks equal to the length of the censored word.
2. Given a certain Physics problem create a function(projectilemotion\_solver) that would take in the following inputs below and return the needed information when the function is called. Name the program containing the function projectilemotion.py then create another program main\_program.py and import projectilemotion.py  
“A long jumper leaves the ground at an angle of 20.0° above the horizontal and at a speed of 11.0 m/s. “
  - (a) How far does he jump in the horizontal direction?
  - (b) What is the maximum height reached?

Given a projectile motion problem like this where the angle and speed are given, the range or distance travelled in the horizontal direction can be determined by using the formula:

$$R = \frac{v_i^2 \sin 2\theta_i}{g}$$

The maximum height can be determined using the formula:

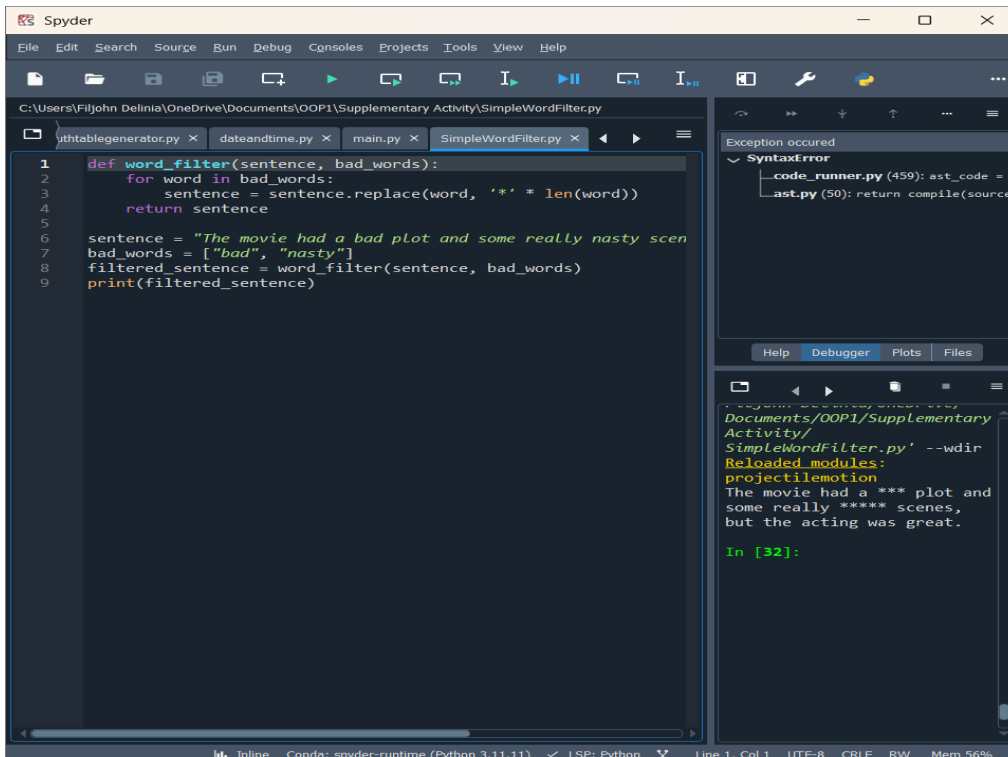
$$h = \frac{v_i^2 \sin^2 \theta_i}{2g}$$

Reference: Serway, Jewet (2019), Physics for Scientists and Engineers 9e

3. Create a quadratic equation solver module that would write the inputs of the user and the corresponding output into text files.

### OUTPUTS:

1.



```
1 def word_filter(sentence, bad_words):
2     for word in bad_words:
3         sentence = sentence.replace(word, '*' * len(word))
4     return sentence
5
6 sentence = "The movie had a bad plot and some really nasty scen
7 bad_words = ["bad", "nasty"]
8 filtered_sentence = word_filter(sentence, bad_words)
9 print(filtered_sentence)
```

```
Exception occurred
SyntaxError
_code_runner.py (459): ast_code = ...
ast.py (50): return compile(source ...

Help Debugger Plots Files

C:\Users\Filjohn Delinia\OneDrive\Documents\OOP1\Supplementary Activity\SimpleWordFilter.py
Reloaded modules:
projectilemotion
The movie had a *** plot and
some really ***** scenes,
but the acting was great.

In [32]:
```

2.

```

Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help

C:\Users\Filjohn Delinia\OneDrive\Documents\OOP1\Supplementary Activity\main_program.py

main.py x SimpleWordFilter.py x projectilemotion.py x main_program.py x

1 from projectilemotion import projectilemotion_solver
2
3 initial_speed = 11.0 # m/s
4 angle_degrees = 20.0 # degrees
5
6 range_distance, max_height = projectilemotion_solver(initial_sp
7
8 print(f'(a) The horizontal distance jumped is: {range_distance:}
9 print(f'(b) The maximum height reached is: {max_height:.2f} met

Exception occurred
SyntaxError
code_runner.py (459): ast_code =
ast.py (50): return compile(source)

Help Debugger Plots Files

In [32]: %runfile 'C:/Users/Filjohn Delinia/OneDrive/
Documents/OOP1/Supplementary
Activity/main_program.py' --
wdir
(a) The horizontal distance
jumped is: 7.93 meters
(b) The maximum height
reached is: 0.72 meters

In [33]:

Inline Conda: spyder-runtime (Python 3.11.11) LSP: Python Line 9, Col 69 UTF-8 CRLF RW Mem 56%

```

3.

```

Spyder
File Edit Search Source Run Debug Consoles Projects Tools View Help

C:\Users\Filjohn Delinia\OneDrive\Documents\OOP1\Supplementary Activity\main_quadratic.py

Filter.py x projectilemotion.py x main_program.py x main_quadratic.py x

1 from quadratic_solver import solve_quadratic
2
3 def write_to_file(a, b, c, roots):
4
5     Writes the inputs and outputs to a text file.
6
7     with open('quadratic_results.txt', 'a') as file:
8         file.write(f'Inputs: a = {a}, b = {b}, c = {c}\n')
9         file.write('Roots:\n')
10        for root in roots:
11            file.write(f' {root}\n')
12        file.write('\n')
13
14    def get_user_input():
15
16        Prompts the user to input coefficients a, b, and c.
17
18        a = float(input('Enter coefficient a: '))
19        b = float(input('Enter coefficient b: '))
20        c = float(input('Enter coefficient c: '))
21        return a, b, c
22
23    def main():
24
25        Main function to solve the quadratic equation and write re
26
27        a, b, c = get_user_input()
28        roots = solve_quadratic(a, b, c)
29        write_to_file(a, b, c, roots)
30        print('Roots:')
31        for root in roots:
32            print(root)
33
34    if __name__ == '__main__':
35        main()

Exception occurred
ModuleNotFoundError
utils.py (209): exec_fun(compile(co
main_quadratic.py (1): from quadra

Help Debugger Plots Files

In [38]: %runfile 'C:/Users/Filjohn Delinia/OneDrive/
Documents/OOP1/Supplementary
Activity/main_quadratic.py' --
wdir
Enter coefficient a: 1
Enter coefficient b: -3
Enter coefficient c: 2
Roots:
2.0
1.0

In [39]:

Inline Conda: spyder-runtime (Python 3.11.11) LSP: Python Line 32, Col 20 UTF-8 CRLF RW Mem 56%

```



## Questions

### 1. Why do built-in functions exist?

Built-in functions are like ready-made tools in Python. They exist to save you time and effort by providing common, pre-written solutions for tasks like printing, calculating lengths, or adding numbers. They're fast, reliable, and tested, so you don't have to reinvent the wheel every time you code.

### 2. What are the advantages/disadvantages of placing code inside functions vs sequential codes.

Functions are like reusable building blocks—they help you avoid repeating code, make your programs easier to understand, and let you fix bugs faster. However, they can add a tiny bit of extra work for the computer. Sequential code is simpler for small tasks but gets messy and repetitive as your program grows, making it harder to manage.

### 3. What is the difference between a function and a module?

A function is a single task, like a recipe for adding two numbers. A module is like a cookbook—it's a file that holds many related recipes (functions, classes, or variables). For example, a `math_operations.py` module might have functions for adding, subtracting, and multiplying.

### 4. What is the difference between a module and a package?

A module is a single file with code, like a single cookbook. A package is a folder full of cookbooks (modules), organized neatly with a special `__init__.py` file to tell Python it's a package. For example, a `my_package` folder could contain `module1.py` and `module2.py`.

## 7. Conclusion:

In Conclusion, built-in functions are like handy tools that save you time and effort by solving common problems for you. Organizing your code into functions makes it reusable, easier to understand, and simpler to fix when something goes wrong. Modules and packages take this a step further by helping you group related code into neat, organized structures. Whether you're working on a small script or a big project, these ideas help you write cleaner, more efficient, and professional code.

## 8. Assessment Rubric: