Project 2: Introduction to Numpy

John Wesley Mathis

Dr. Anthony Choi

June 02, 2024

ECE/SSE 591, Summer 2024

Table of Contents:

Table of Figures:

Deliverable Table

The purpose of this table is to provide a complete view of the concepts covered in chapter 2 of *"Python Data Science Handbook"* (VanderPlas, 2016) and provide a general page location and/or project name for where the topic was demonstrated. This is not an exhaustive list. Some areas may be covered by multiple projects. This is meant only as a means to show specific areas where the topic is demonstrated and to ensure that every area was covered.

Deliverables	Location
Understanding Data Types in Python	Array Calculator
The Basics of NumPy Arrays	Array Calculator
Computation of NumPy Arrays: Universal Functions	Array Calculator
Aggregations: Min, Max, and Everything In Between	SIR Model Analysis
Computation on Arrays: Broadcasting	SIR Model Analysis
Comparisons, Masks, and Boolean Logic	SIR Model Analysis
Fancy Indexing	SIR Model Analysis
Sorting Arrays	SIR Model Analysis
Structured Data: NumPy's Structured Arrays	SIR Model Analysis

Additionally, here is a GitHub link to download the Jupyter Notebook files to test. Please note, in order to properly run, recent versions of Python, NumPy must be installed. Please refer to online documentations on how to install these dependencies.

GitHub Link: https://github.com/jwmathis/SSE591_Project2.git

1. Introduction

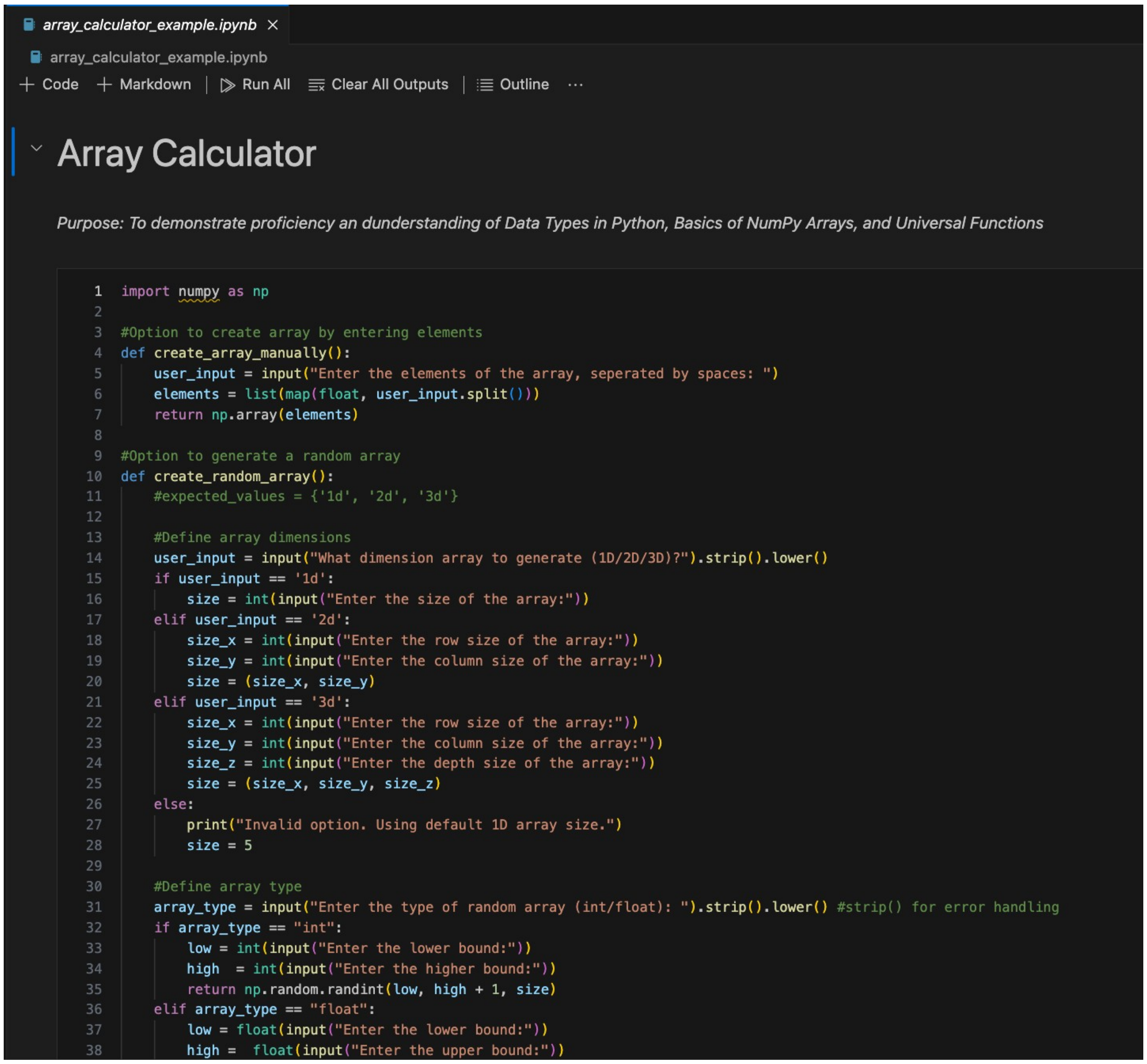
Python is a versatile language with numerous libraries. Because of its easy to understand syntax, it is very popular among the data science community as a useful tool to load, store, and manipulate data. Much of this data is typically managed by converting it into arrays of numbers. However, as useful and easy as it is to use Python programming, managing data efficiently has its many drawbacks. Fortunately, there are specialized tools that have been created to improve Python's ability to handle such numerical data. One such package that has been created is the Numerical Python package or NumPy as it's generally known.

NumPy provides a manner to more efficiently handle data arrays. NumPy's arrays are similar to Python's built-in arrays but where it shines is how is handles storage as the data grows larger. As a result, NumPy has become the foundation for many scientific tools that are in use.

This report aims to demonstrate my proficiency in Python fundamentals as well as NumPy fundamentals that were covered in chapter 2 of the "*Python Data Science Handbook"* written by Jake VanderPlas (2016). Each exercise and mini-project is used to illustrate the concepts outlined in the deliverable table from the previous section. The code presented throughout this report was written using Visual Studio Code with Jupyter Notebook extensions. In the following sections an analysis is done on the two mini-projects. The remaining sections cover various exercises, and explanations are detailed to explore the structure and functionality of coding using NumPy packages.

2. Array CalculatorAnalysis

The first practical example I coded was an array calculator. This calculator covers several fundamental NumPy concepts to demonstrate proficiency in NumPy arrays, Universal Functions, and NumPy's aggregation functions. The program is able to perform various operations on a provided array. The program builds off of many basic python concepts while incorporating new NumPy concepts. Figure 1-3 below provides the complete code for the program. To streamline the code and make it re-usable, 5 functions were created. The first function named `display_menu`, prints out the various options offered by the program for array operations. The second function is named `create_array`, and prints out a menu and provides options for the user to choose how to input an array; whether by manually entering a 1D array, or having an array generated randomly based on the information they provide. The control flow `if` statement is used to select the proper array function creation. The third function is called `create_array_manually` and string manipulation, `map`, and `list`, functions to convert the user input into a list of floats. The list of floats is then converted to a NumPy array using `np.array`. The fourth function is named `create_random_array` and allows the user to customize what type of random array is generated by inputing if they what size of an array they would like along with what type (int/float) of array they would like. Based on the input, a random NumPy array is returned. The fifth function is named `perform_operations` and receives the user's array and menu choice. The function utilizes control flow `if` statements to return the correct operation. The operations performed on the array use NumPy's universal functions such as `np.square`, `np.mean`' and `np.exp`. A `while True` loop runs the code allowing the user play around in the program manipulating arrays until they choose the exit option.

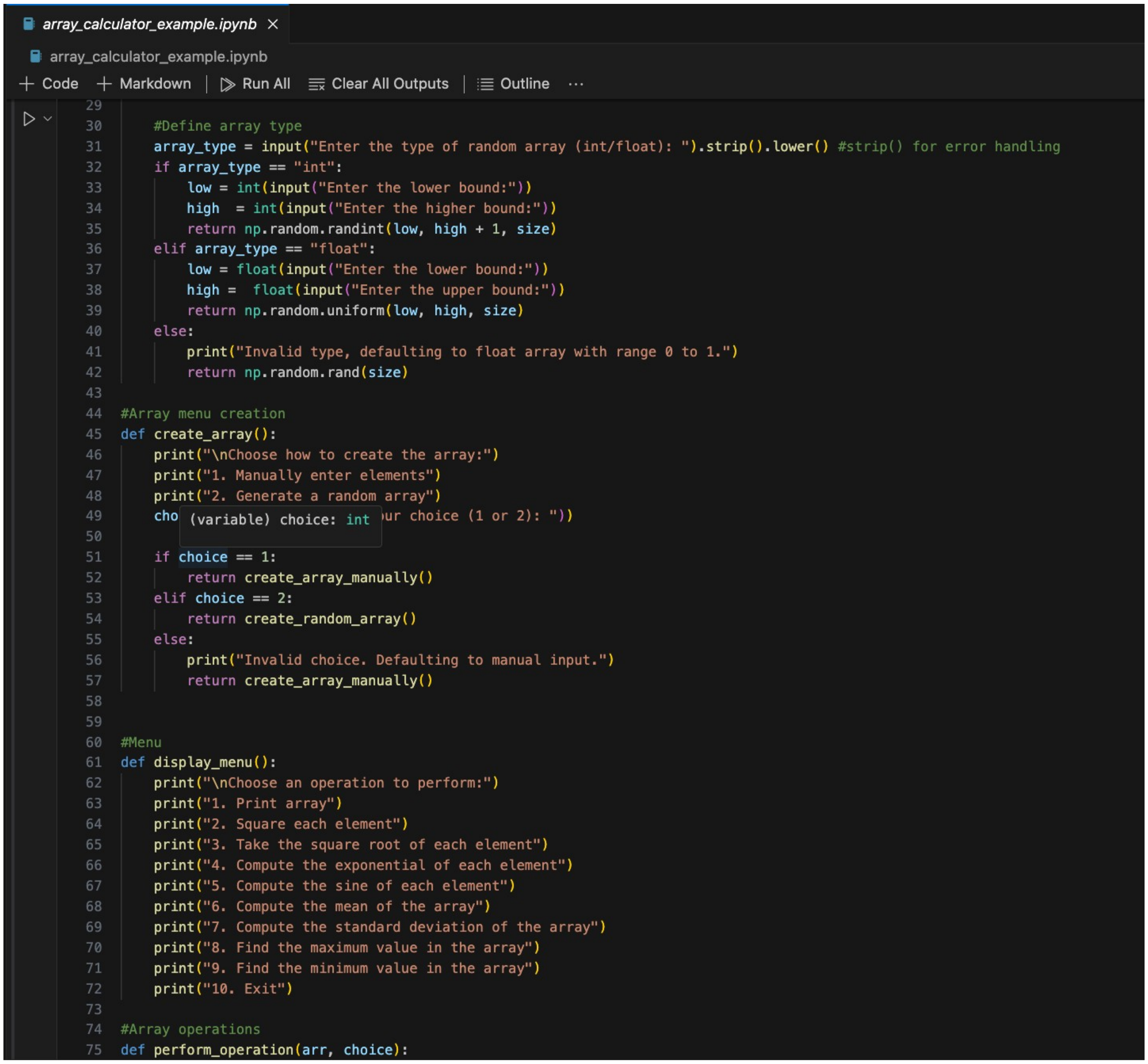

array_calculator_example.ipynb
array_calculator_example.ipynb
Code
Markdown
Run All
Clear All Outputs
Outline

Array Calculator

Purpose: To demonstrate proficiency an dunderstanding of Data Types in Python, Basics of NumPy Arrays, and Universal Functions

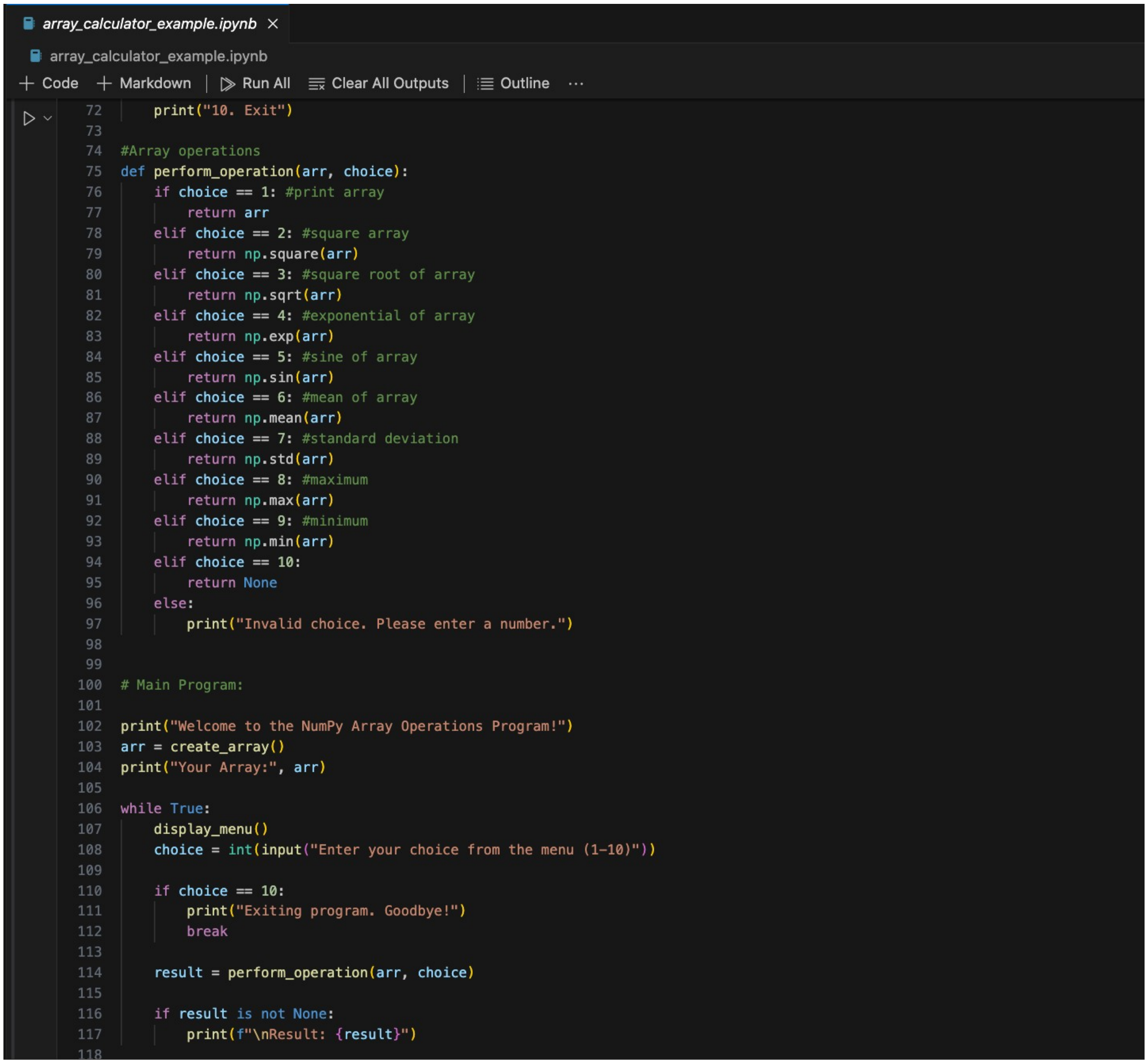
```python
import numpy as np

#Option to create array by entering elements
def create_array_manually():
    user_input = input("Enter the elements of the array, seperated by spaces: ")
    elements = list(map(float, user_input.split()))
    return np.array(elements)

#Option to generate a random array
def create_random_array():
    #expected_values = {'1d', '2d', '3d'}

    #Define array dimensions
    user_input = input("What dimension array to generate (1D/2D/3D)?").strip().lower()
    if user_input == '1d':
        size = int(input("Enter the size of the array:"))
    elif user_input == '2d':
        size_x = int(input("Enter the row size of the array:"))
        size_y = int(input("Enter the column size of the array:"))
        size = (size_x, size_y)
    elif user_input == '3d':
        size_x = int(input("Enter the row size of the array:"))
        size_y = int(input("Enter the column size of the array:"))
        size_z = int(input("Enter the depth size of the array:"))
        size = (size_x, size_y, size_z)
    else:
        print("Invalid option. Using default 1D array size.")
        size = 5

    #Define array type
    array_type = input("Enter the type of random array (int/float): ").strip().lower() #strip() for error handling
    if array_type == "int":
        low = int(input("Enter the lower bound:"))
        high  = int(input("Enter the higher bound:"))
        return np.random.randint(low, high + 1, size)
    elif array_type == "float":
        low = float(input("Enter the lower bound:"))
        high =  float(input("Enter the upper bound:"))
```

Figure 1: Array Calculator: Code


array_calculator_example.ipynb
array_calculator_example.ipynb
Code
Markdown
Run All
Clear All Outputs
Outline

```python

    #Define array type
    array_type = input("Enter the type of random array (int/float): ").strip().lower() #strip() for error handling
    if array_type == "int":
        low = int(input("Enter the lower bound:"))
        high  = int(input("Enter the higher bound:"))
        return np.random.randint(low, high + 1, size)
    elif array_type == "float":
        low = float(input("Enter the lower bound:"))
        high =  float(input("Enter the upper bound:"))
        return np.random.uniform(low, high, size)
    else:
        print("Invalid type, defaulting to float array with range 0 to 1.")
        return np.random.rand(size)

#Array menu creation
def create_array():
    print("\nChoose how to create the array:")
    print("1. Manually enter elements")
    print("2. Generate a random array")
    cho (variable) choice: int ur choice (1 or 2): "))

    if choice == 1:
        return create_array_manually()
    elif choice == 2:
        return create_random_array()
    else:
        print("Invalid choice. Defaulting to manual input.")
        return create_array_manually()


#Menu
def display_menu():
    print("\nChoose an operation to perform:")
    print("1. Print array")
    print("2. Square each element")
    print("3. Take the square root of each element")
    print("4. Compute the exponential of each element")
    print("5. Compute the sine of each element")
    print("6. Compute the mean of the array")
    print("7. Compute the standard deviation of the array")
    print("8. Find the maximum value in the array")
    print("9. Find the minimum value in the array")
    print("10. Exit")

#Array operations
def perform_operation(arr, choice):
```

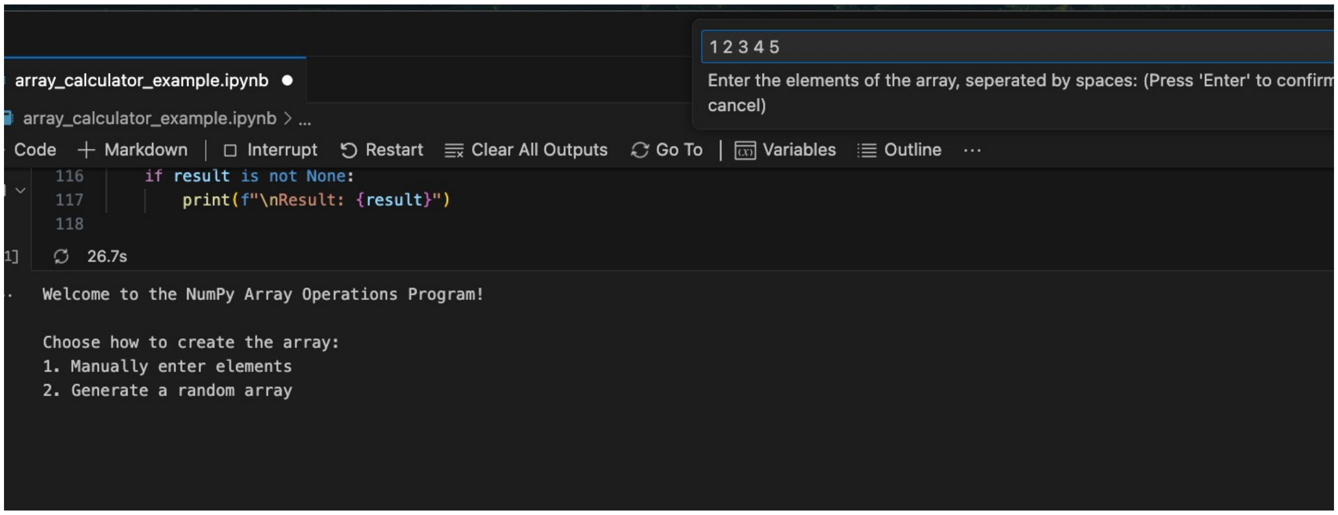
Figure 2: Array Calculator: Code (continued)


array_calculator_example.ipynb
array_calculator_example.ipynb
+ Code
+ Markdown
Run All
Clear All Outputs
Outline

```python
    print("10. Exit")

#Array operations
def perform_operation(arr, choice):
    if choice == 1: #print array
        return arr
    elif choice == 2: #square array
        return np.square(arr)
    elif choice == 3: #square root of array
        return np.sqrt(arr)
    elif choice == 4: #exponential of array
        return np.exp(arr)
    elif choice == 5: #sine of array
        return np.sin(arr)
    elif choice == 6: #mean of array
        return np.mean(arr)
    elif choice == 7: #standard deviation
        return np.std(arr)
    elif choice == 8: #maximum
        return np.max(arr)
    elif choice == 9: #minimum
        return np.min(arr)
    elif choice == 10:
        return None
    else:
        print("Invalid choice. Please enter a number.")


# Main Program:

print("Welcome to the NumPy Array Operations Program!")
arr = create_array()
print("Your Array:", arr)

while True:
    display_menu()
    choice = int(input("Enter your choice from the menu (1-10)"))

    if choice == 10:
        print("Exiting program. Goodbye!")
        break

    result = perform_operation(arr, choice)

    if result is not None:
        print(f"\nResult: {result}")
```

Figure 3: Array Calculator: Code (final)

Figure 4 below shows the program running for the first time. The user is greeted with a welcome message and menu to choose how to create an array. For this example, the user chose option 1 to manually enter the elements.

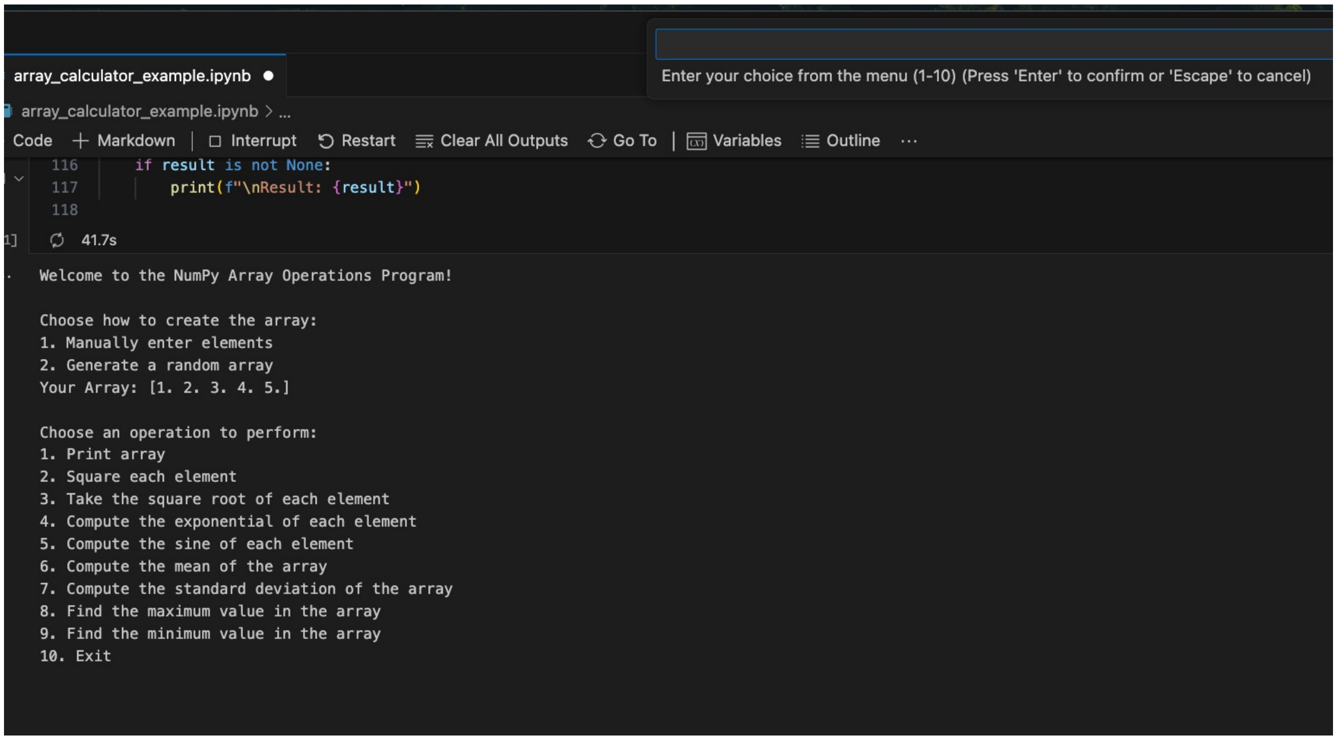


```
12345
Enter the elements of the array, seperated by spaces: (Press 'Enter' to confirm
cancel)
```

```
        if result is not None:
            print(f"\nResult: {result}")

Welcome to the NumPy Array Operations Program!

Choose how to create the array:
1. Manually enter elements
2. Generate a random array
```

Figure 4: Array Calculator: Manually entering elements for an array

After the user has entered the elements, the array is printed to the screen. A new menu of options is displayed allowing the user to choose a number from the menu to perform operations on the array. Figure 5 below shows the output after a user has created an array.

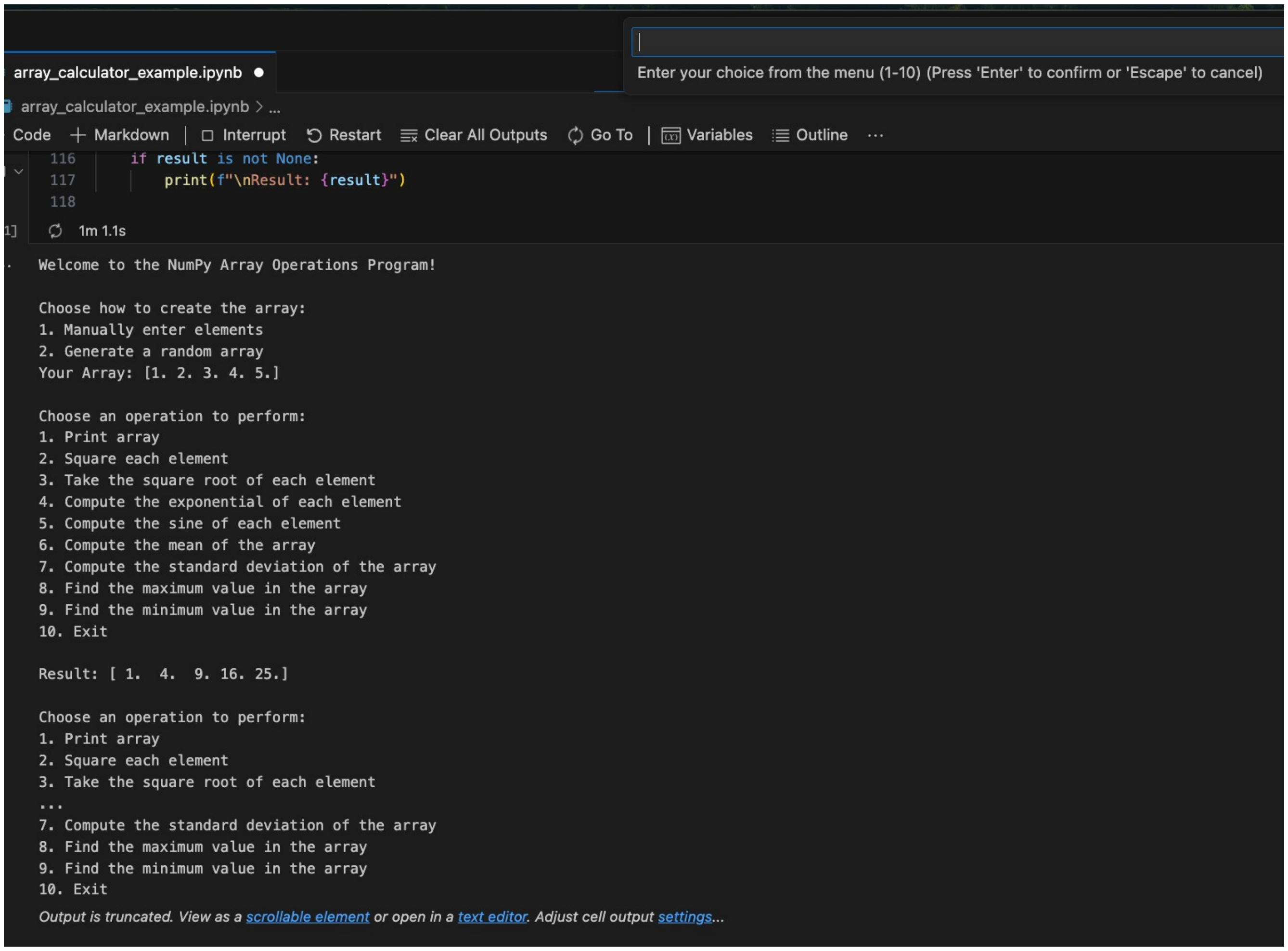


```
Enter your choice from the menu (1-10) (Press 'Enter' to confirm or 'Escape' to cancel)
```

```
        if result is not None:
            print(f"\nResult: {result}")

Welcome to the NumPy Array Operations Program!

Choose how to create the array:
1. Manually enter elements
2. Generate a random array
Your Array: [1. 2. 3. 4. 5.]

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 5: Array Calculator: User defined array output and array operations menu

Figure 6-13 below shows the output of each option. As described previously, the operations are performed on the array by using the NumPy universal functions. These functions are accessed by the control flow *`if`* statements and by the user choice.

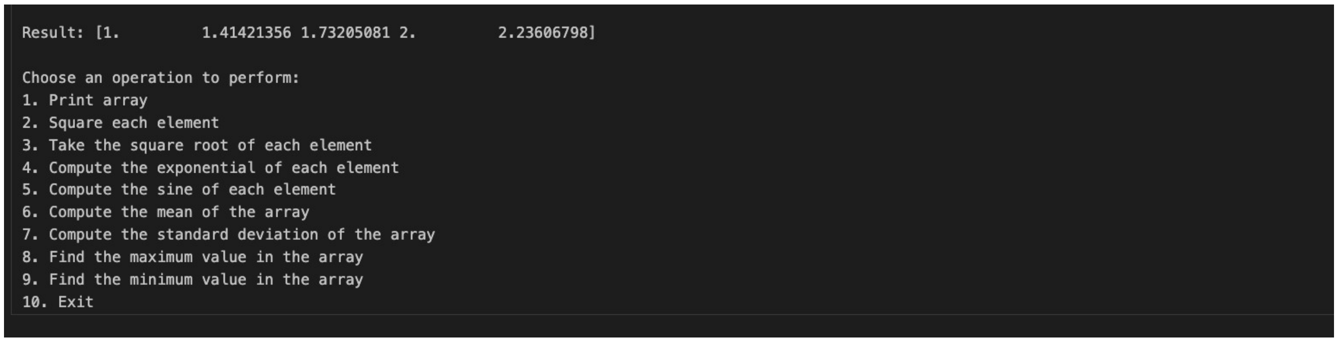

array_calculator_example.ipynb
Enter your choice from the menu (1-10) (Press 'Enter' to confirm or 'Escape' to cancel)
array_calculator_example.ipynb > ...
Code
Markdown
Interrupt
Restart
Clear All Outputs
Go To
Variables
Outline
1m 1.1s

```
    if result is not None:
        print(f"\nResult: {result}")
```

```
Welcome to the NumPy Array Operations Program!

Choose how to create the array:
1. Manually enter elements
2. Generate a random array
Your Array: [1. 2. 3. 4. 5.]

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit

Result: [ 1.  4.  9. 16. 25.]

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
...
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

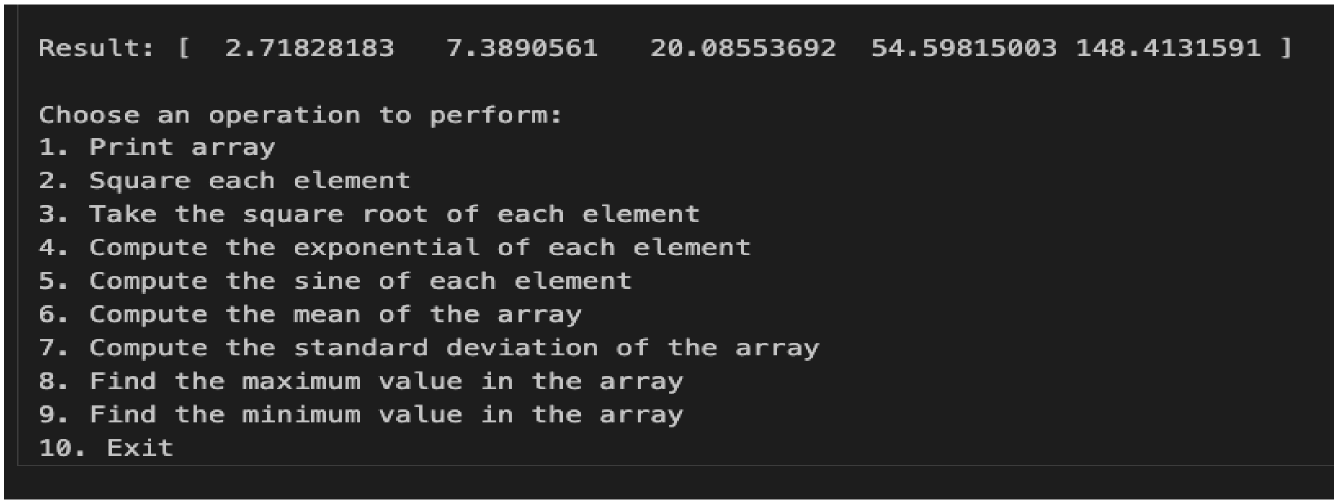
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Figure 6: Array Calculator: Squaring elements output



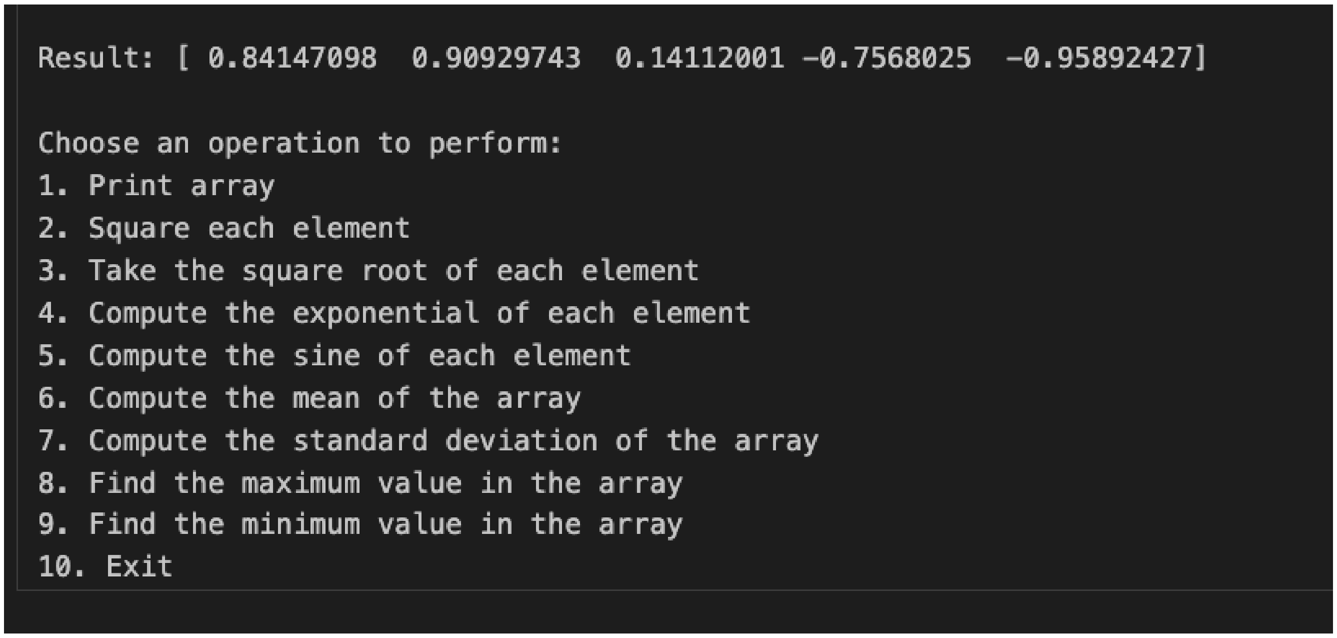
```
Result: [1.         1.41421356 1.73205081 2.         2.23606798]

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 7: Array Calculator: Square root option output



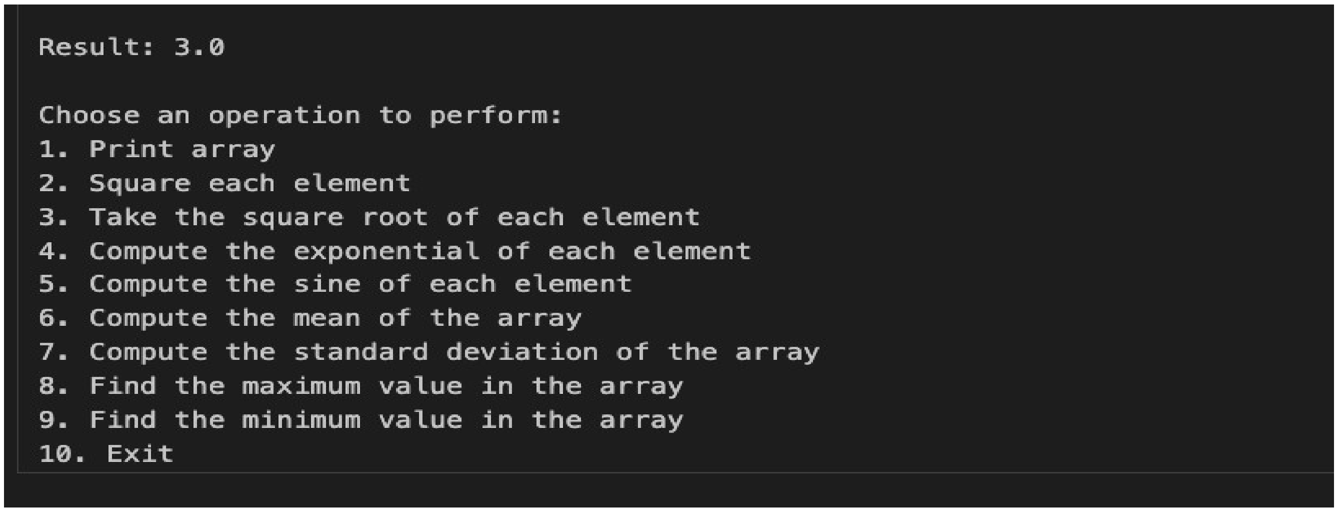
```
Result: [  2.71828183   7.3890561   20.08553692  54.59815003 148.4131591 ]

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 8: Array Calculator: Exponential option output



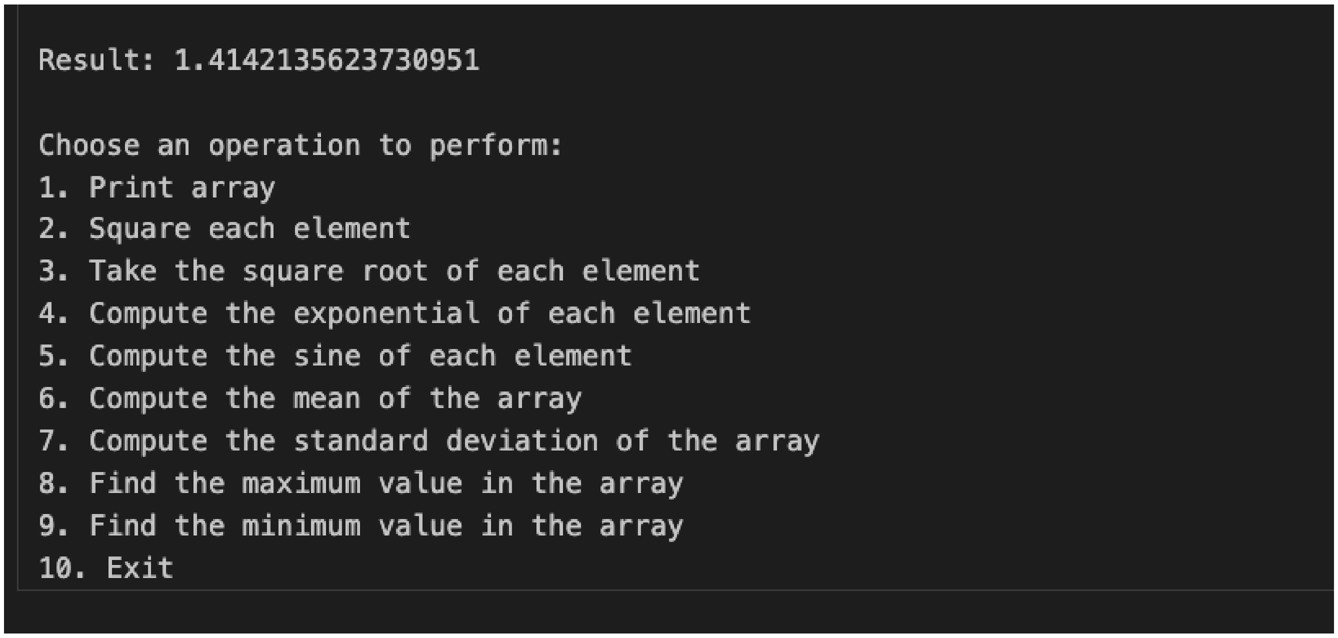
```
Result: [ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427]

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 9: Array Calculator: Sine option output



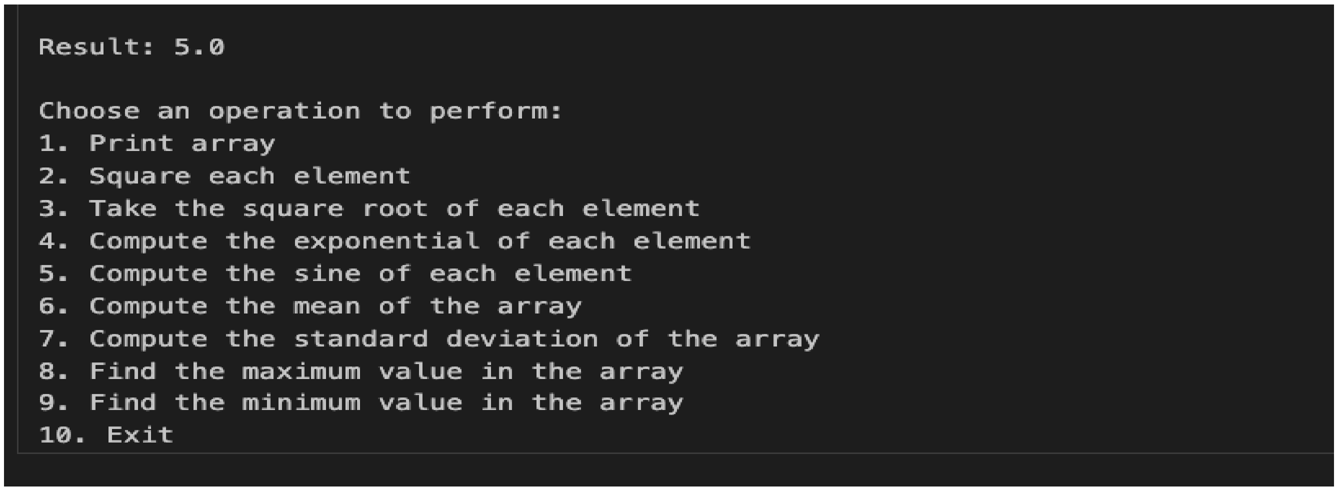
```
Result: 3.0

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 10: Array Calculator: Mean option output



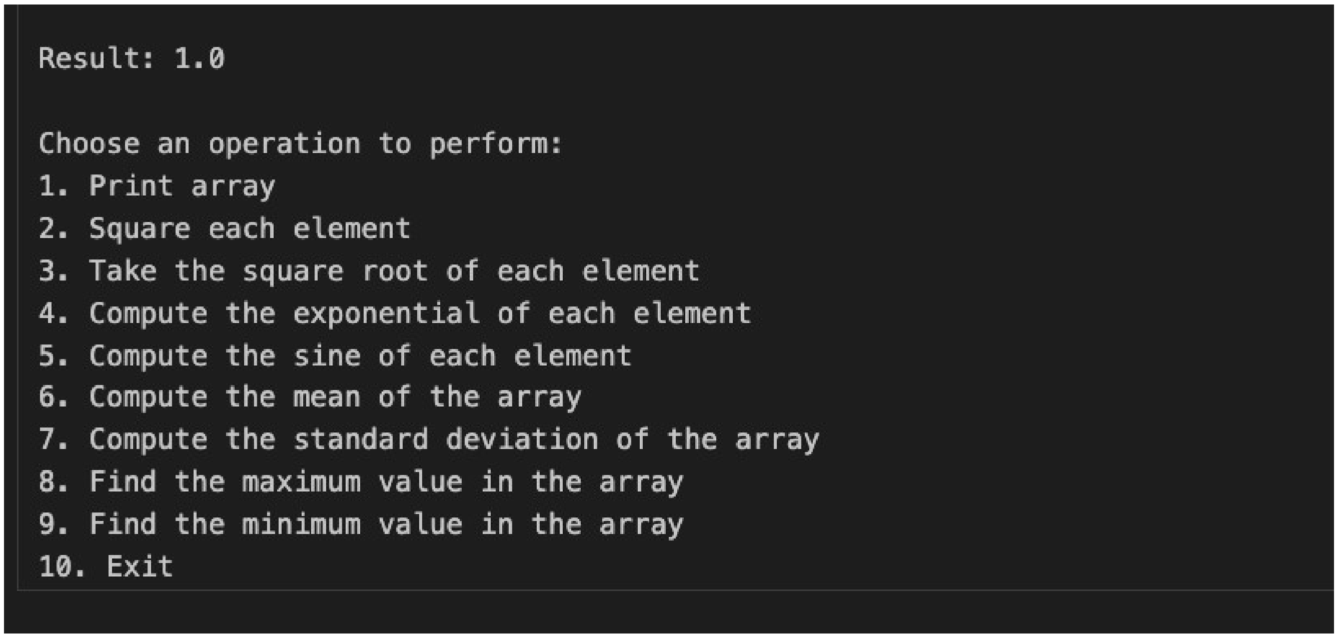
```
Result: 1.4142135623730951

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 11: Array Calculator: Standard deviation option output



```
Result: 5.0

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

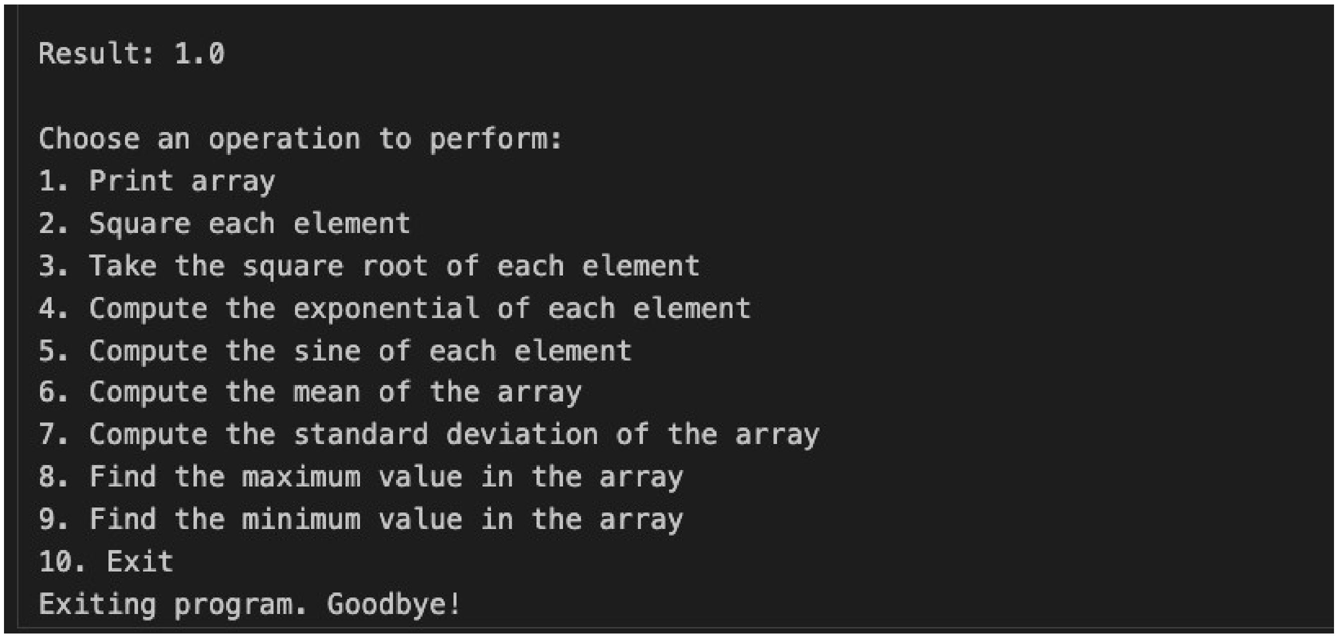
Figure 12: Array Calculator: Max value option output



```
Result: 1.0

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 13: Array Calculator: Min value option output

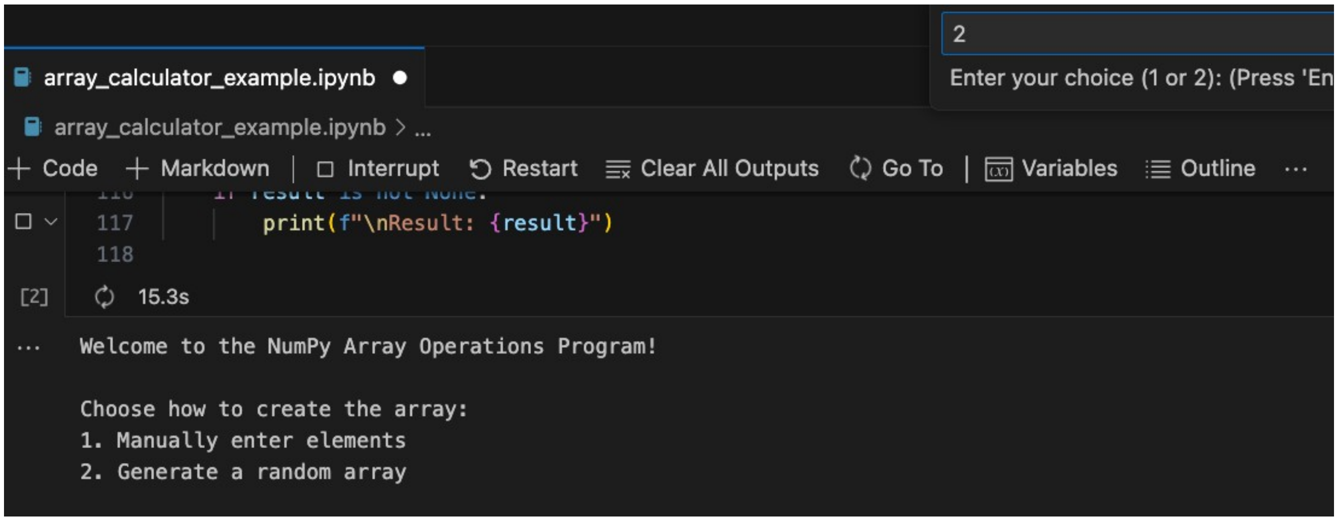
Figure 14 below shows the output if the user selects to exit the program. This functionality was implemented by simply using an *`if`* statement to check if the user as inputted the string '10' to exit. If so, the program prints out an exiting statement and breaks out of the *`while`* loop.



```
Result: 1.0

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
Exiting program. Goodbye!
```

Figure 14: Array Calculator: Exiting the program

To make the program a little more unique, the user is given an option to randomly generate an array based on user defined conditions. To run this part of the program the user must select option 2 when asked how they would like to define the array. Figure 15 shows an example.



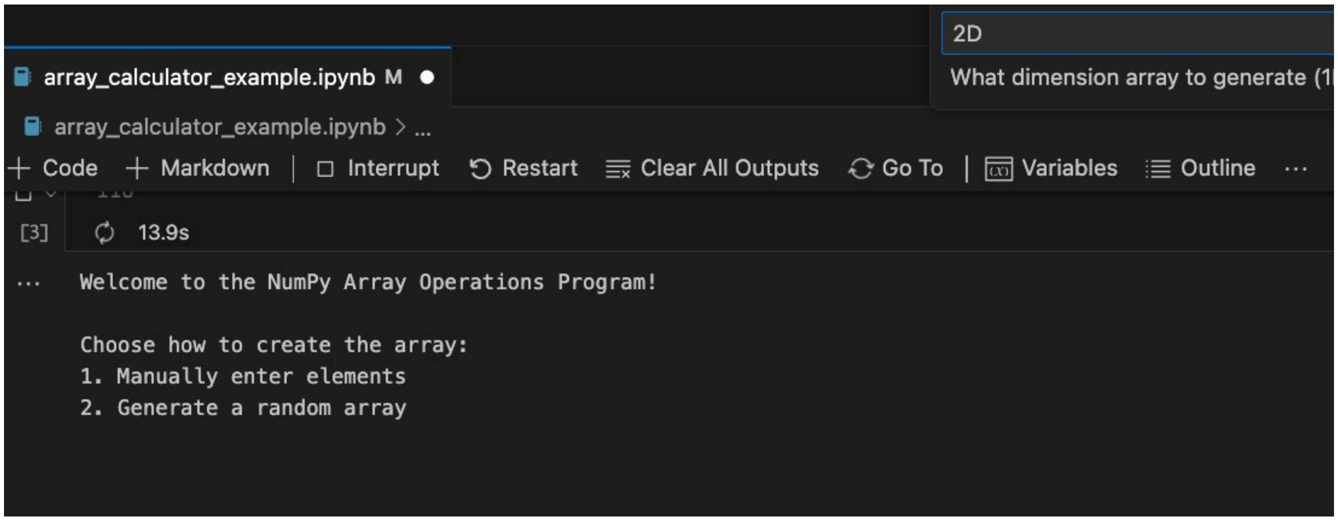
```
2
Enter your choice (1 or 2): (Press 'En
```

```python
        print(f"\nResult: {result}")
```

```
Welcome to the NumPy Array Operations Program!

Choose how to create the array:
1. Manually enter elements
2. Generate a random array
```

Figure 15: Array Calculator: Generating a random array option

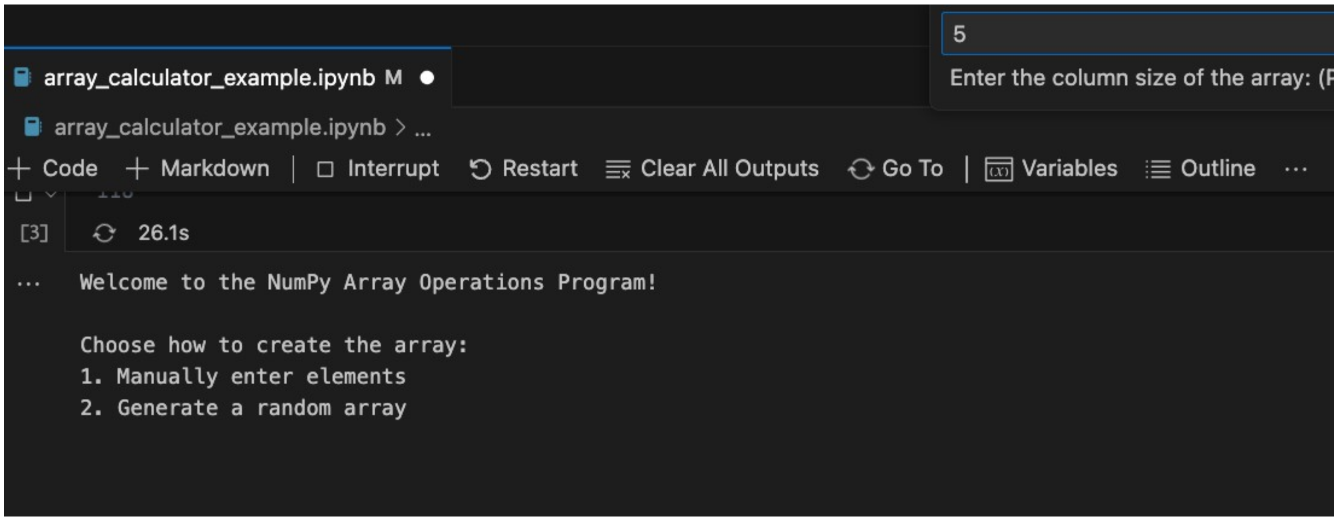
After the user has selected this option, the program asks the user for input to define the array dimension, the row size, the column size, and the data type for the array. Additionally, the user is asked for a range of number to use for random selection. Figures 16-22 show an example of the user entering the details and the array output for a 2-dimensional array. Similar to if the user defines their own elements, the same operations can be performed on the higher dimension arrays.



2D
What dimension array to generate (1

```
Welcome to the NumPy Array Operations Program!

Choose how to create the array:
1. Manually enter elements
2. Generate a random array
```

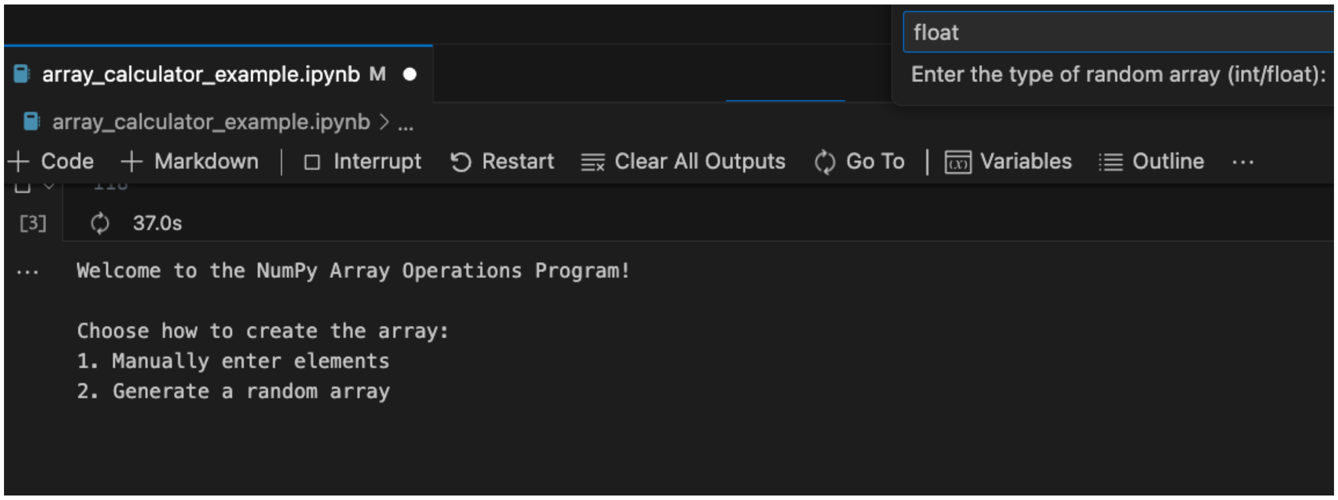
Figure 16: Array Calculator: Selecting array dimension


array_calculator_example.ipynb M
array_calculator_example.ipynb > ...
Code
Markdown
Interrupt
Restart
Clear All Outputs
Go To
Variables
Outline

```
5
Enter the column size of the array: (P
```

```
[3]    26.1s

...    Welcome to the NumPy Array Operations Program!

       Choose how to create the array:
       1. Manually enter elements
       2. Generate a random array
```

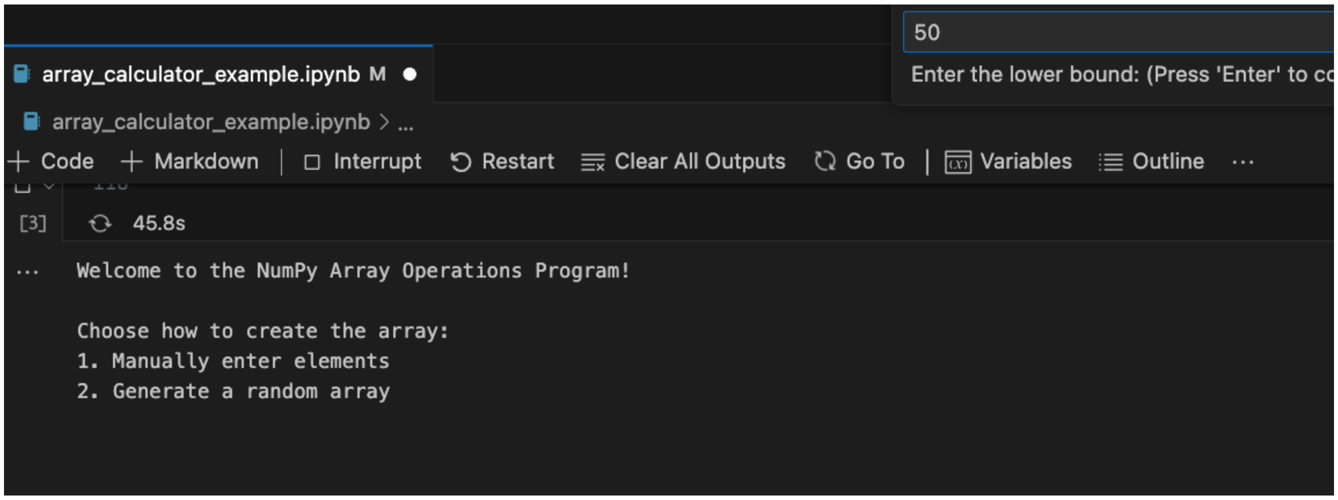
Figure 17: Array Calculator: Defining column size


array_calculator_example.ipynb M
array_calculator_example.ipynb > ...
Code
Markdown
Interrupt
Restart
Clear All Outputs
Go To
Variables
Outline

```
float
Enter the type of random array (int/float):
```

```
[3]    37.0s

...    Welcome to the NumPy Array Operations Program!

       Choose how to create the array:
       1. Manually enter elements
       2. Generate a random array
```

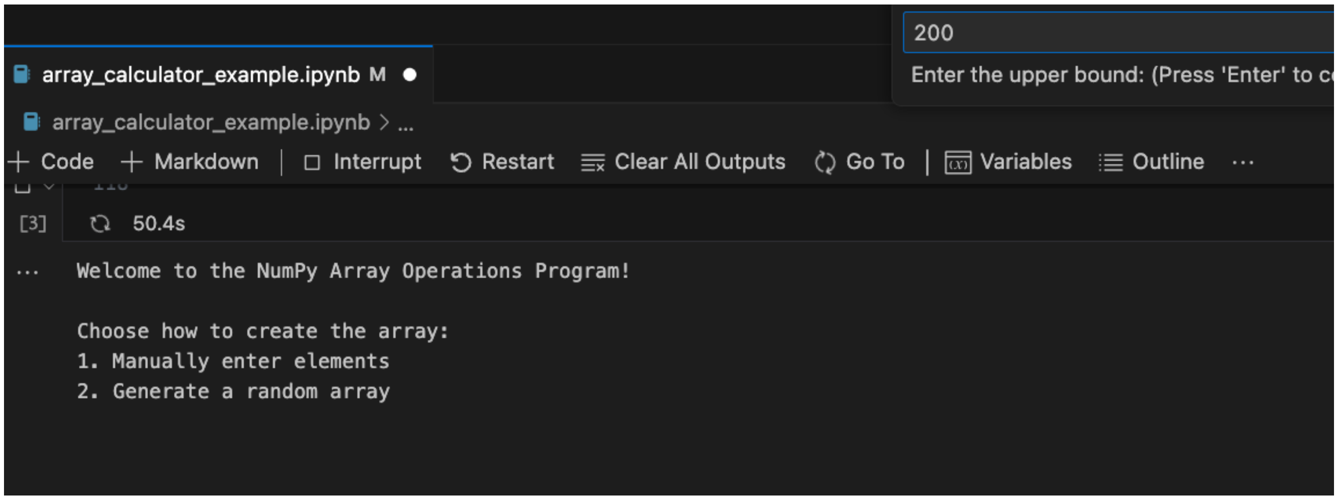
Figure 18: Array Calculator: Defining data type


array_calculator_example.ipynb M
array_calculator_example.ipynb > ...
+ Code + Markdown | Interrupt Restart Clear All Outputs Go To | Variables Outline ...

```
50
Enter the lower bound: (Press 'Enter' to co
```

```
[3]   45.8s
...  Welcome to the NumPy Array Operations Program!

     Choose how to create the array:
     1. Manually enter elements
     2. Generate a random array
```

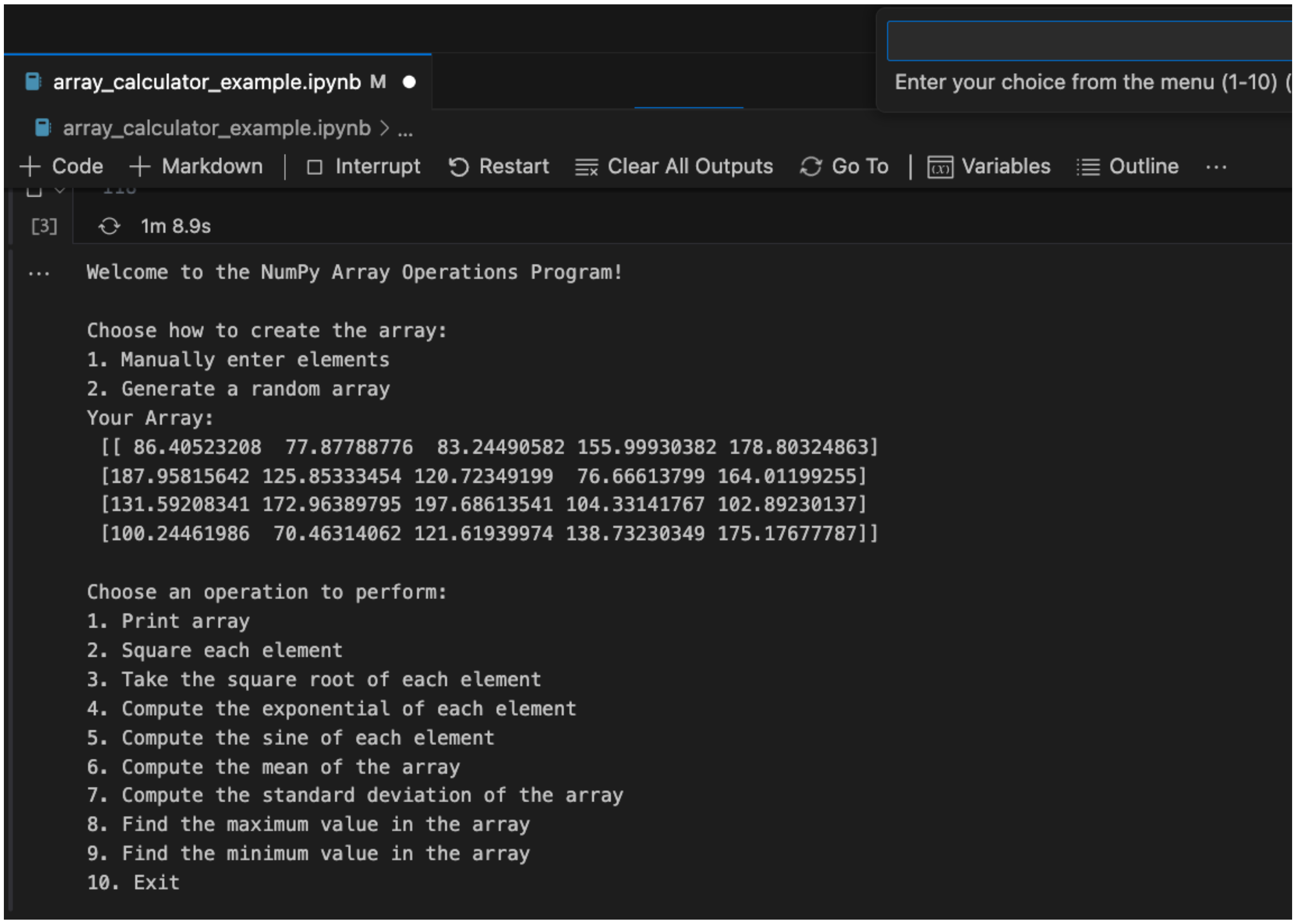
Figure 19: Array Calculator: Defining lower bound


array_calculator_example.ipynb M
array_calculator_example.ipynb > ...
+ Code + Markdown | Interrupt Restart Clear All Outputs Go To | Variables Outline ...

```
200
Enter the upper bound: (Press 'Enter' to co
```

```
[3]   50.4s
...  Welcome to the NumPy Array Operations Program!

     Choose how to create the array:
     1. Manually enter elements
     2. Generate a random array
```

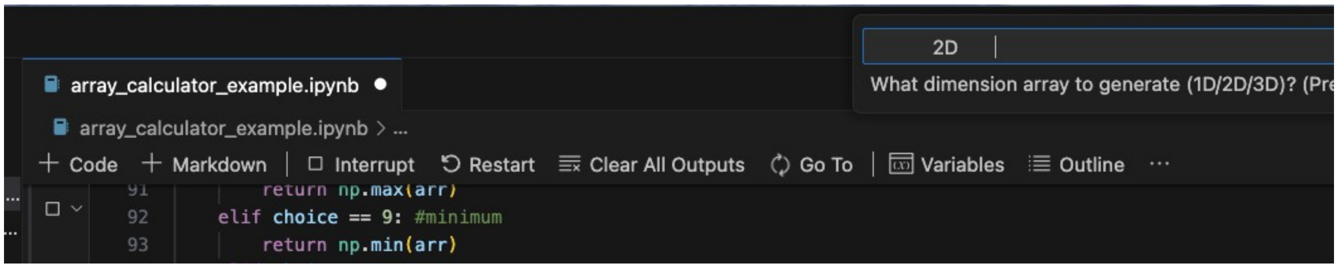
Figure 20: Array Calculator: Defining upper bound


array_calculator_example.ipynb M
array_calculator_example.ipynb > ...
Code
Markdown
Interrupt
Restart
Clear All Outputs
Go To
Variables
Outline
Enter your choice from the menu (1-10) (
[3]
1m 8.9s

```
Welcome to the NumPy Array Operations Program!

Choose how to create the array:
1. Manually enter elements
2. Generate a random array
Your Array:
 [[ 86.40523208  77.87788776  83.24490582 155.99930382 178.80324863]
 [187.95815642 125.85333454 120.72349199  76.66613799 164.01199255]
 [131.59208341 172.96389795 197.68613541 104.33141767 102.89230137]
 [100.24461986  70.46314062 121.61939974 138.73230349 175.17677787]]

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 21: Array Calculator: Random array output

One of the problems encountered while creating this code was handling string input from the user. To ensure proper input was provided, string modulation was used. The *`strip()`* and *`lower()`* methods were incorporated for error handling. This made it easier to ensure that the user did not have to enter the exact phrase correctly to select an option. Figure 22 below show that even entering '   2D' as a response will still permit the user to generate an array.

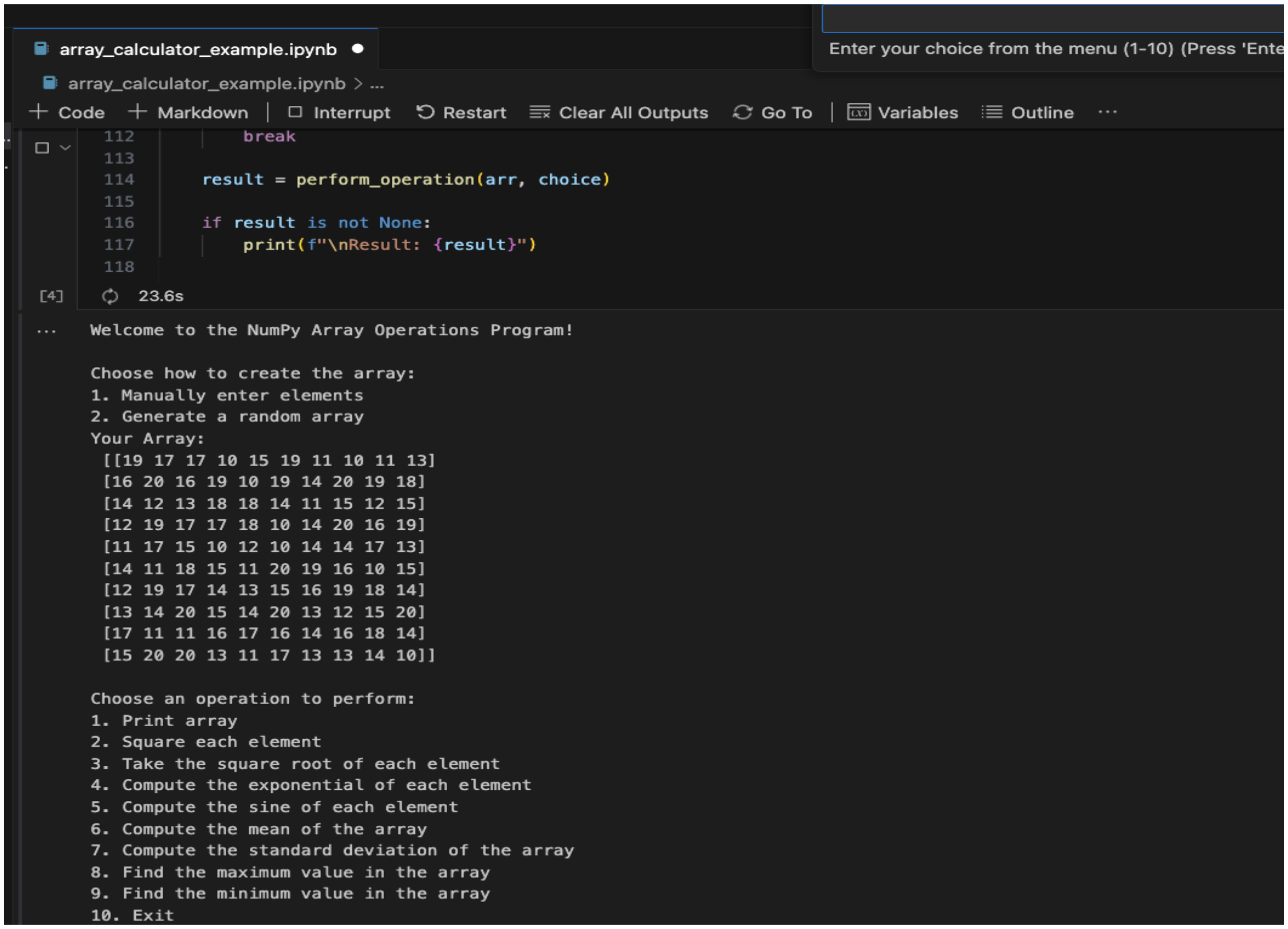

array_calculator_example.ipynb
array_calculator_example.ipynb > ...
Code
Markdown
Interrupt
Restart
Clear All Outputs
Go To
Variables
Outline

```
2D
What dimension array to generate (1D/2D/3D)? (Pre
```

```python
            return np.max(arr)
        elif choice == 9: #minimum
            return np.min(arr)
```

Figure 22a: Array Calculator: Showing input handling

3. SIR Model Analysis



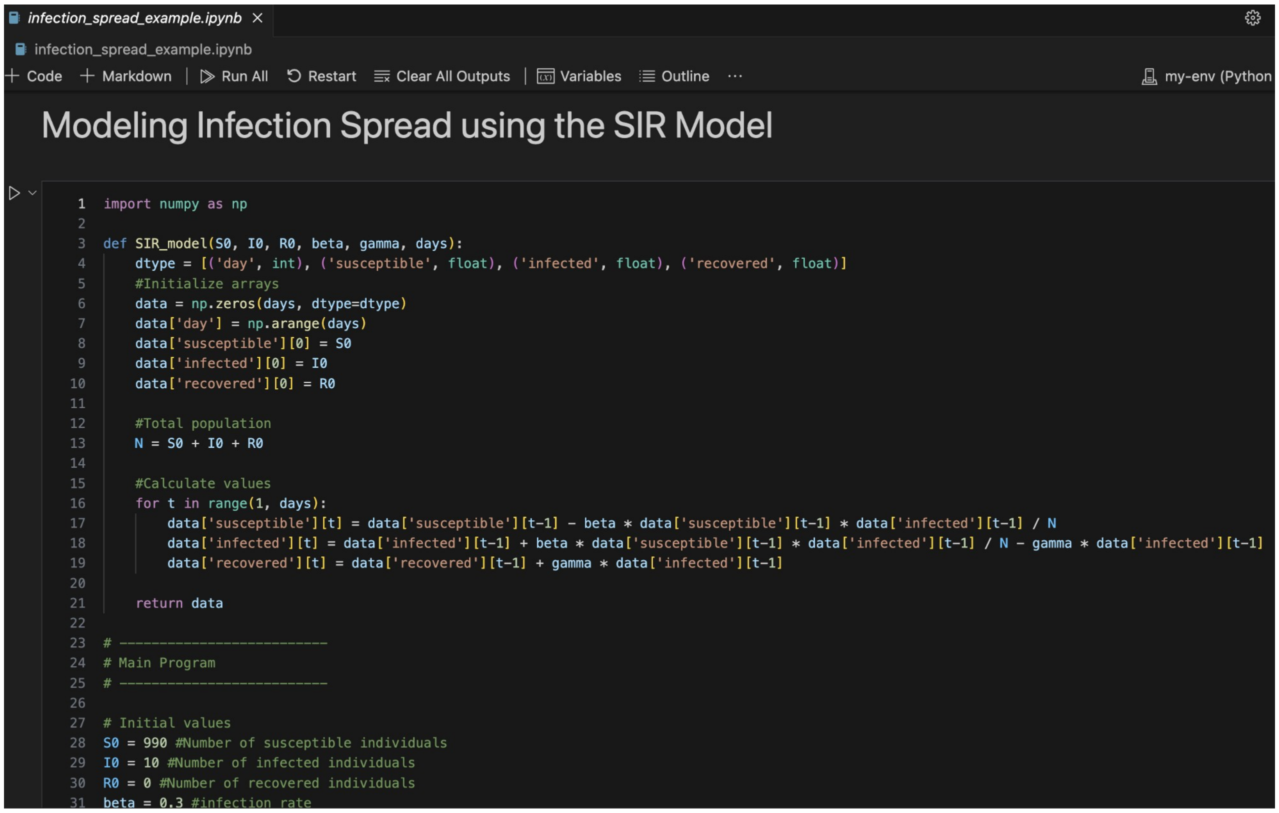
```
Enter your choice from the menu (1-10) (Press 'Ente
```

```python
            break

    result = perform_operation(arr, choice)

    if result is not None:
        print(f"\nResult: {result}")

```

```
Welcome to the NumPy Array Operations Program!

Choose how to create the array:
1. Manually enter elements
2. Generate a random array
Your Array:
 [[19 17 17 10 15 19 11 10 11 13]
 [16 20 16 19 10 19 14 20 19 18]
 [14 12 13 18 18 14 11 15 12 15]
 [12 19 17 17 18 10 14 20 16 19]
 [11 17 15 10 12 10 14 14 17 13]
 [14 11 18 15 11 20 19 16 10 15]
 [12 19 17 14 13 15 16 19 18 14]
 [13 14 20 15 14 20 13 12 15 20]
 [17 11 11 16 17 16 14 16 18 14]
 [15 20 20 13 11 17 13 13 14 10]]

Choose an operation to perform:
1. Print array
2. Square each element
3. Take the square root of each element
4. Compute the exponential of each element
5. Compute the sine of each element
6. Compute the mean of the array
7. Compute the standard deviation of the array
8. Find the maximum value in the array
9. Find the minimum value in the array
10. Exit
```

Figure 22b: Array Calculator: Showing input handling

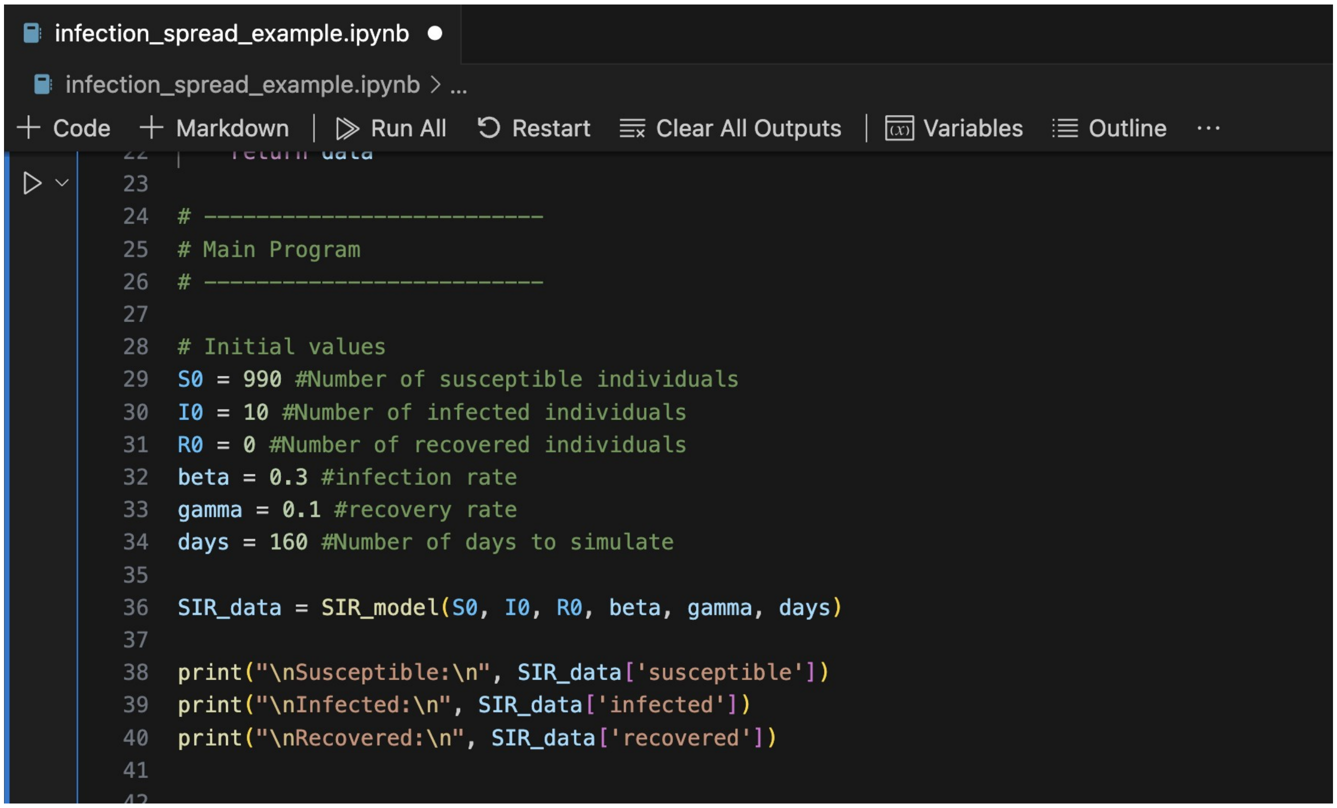
The next project imagines a scenario where there is an outbreak of an infectious disease, such as a zombie apocalypse (Lowe, S., Mathis, J., & Wall, N. (2019)), and creates a Python script to model the spread of the zombie disease. The differential equations used to model the spread of the disease is the SIR model that was developed by Kermack and McKendrick. SIR stands for susceptible, infected, and recovered. For this project I considered two populations: the humans (susceptible group) and the zombies (infected group). The disease can be transmitted via scratching or biting. To simplify the project and modeling, I made the assumptions that the disease spreads rapidly where the change in populations can only be attributed to the infection; and that there are no births, so that the populations stayed constant. I used a structured NumPy array to store and access data in a more organized manner. Four data types were defined: *`day`* as type *int, 'susceptible`, infected`* and *`recovered`* as type *float*. The structured array *`data`* was initialized with pre-defined initial conditions described above. A *`for`* loop was used to update the data for each data type by using the general equations for a SIR model starting with day 1 and ending with the simulations defined variable of *`days`*. The population is calculated by adding together the initial conditions *`S0`, `I0`,* and *`R0`*. This is all stored in the function *`SIR_model(S0, I0, R0, beta, gamma, days)`*. The variable *`beta`* defines the infection rate and the variable *`gamma`* defines the recovery rate. The function returns the structured array *`data`* after using Numpy broadcasting to calculate the values for *`susceptible`, `infected`, and `recovered`* data types for the *`data`* array. Figure 23 and 24 show the final code for the SIR model.



Modeling Infection Spread using the SIR Model

```python
import numpy as np

def SIR_model(S0, I0, R0, beta, gamma, days):
    dtype = [('day', int), ('susceptible', float), ('infected', float), ('recovered', float)]
    #Initialize arrays
    data = np.zeros(days, dtype=dtype)
    data['day'] = np.arange(days)
    data['susceptible'][0] = S0
    data['infected'][0] = I0
    data['recovered'][0] = R0

    #Total population
    N = S0 + I0 + R0

    #Calculate values
    for t in range(1, days):
        data['susceptible'][t] = data['susceptible'][t-1] - beta * data['susceptible'][t-1] * data['infected'][t-1] / N
        data['infected'][t] = data['infected'][t-1] + beta * data['susceptible'][t-1] * data['infected'][t-1] / N - gamma * data['infected'][t-1]
        data['recovered'][t] = data['recovered'][t-1] + gamma * data['infected'][t-1]

    return data

# --------------------------
# Main Program
# --------------------------

# Initial values
S0 = 990 #Number of susceptible individuals
I0 = 10 #Number of infected individuals
R0 = 0 #Number of recovered individuals
beta = 0.3 #infection rate
```

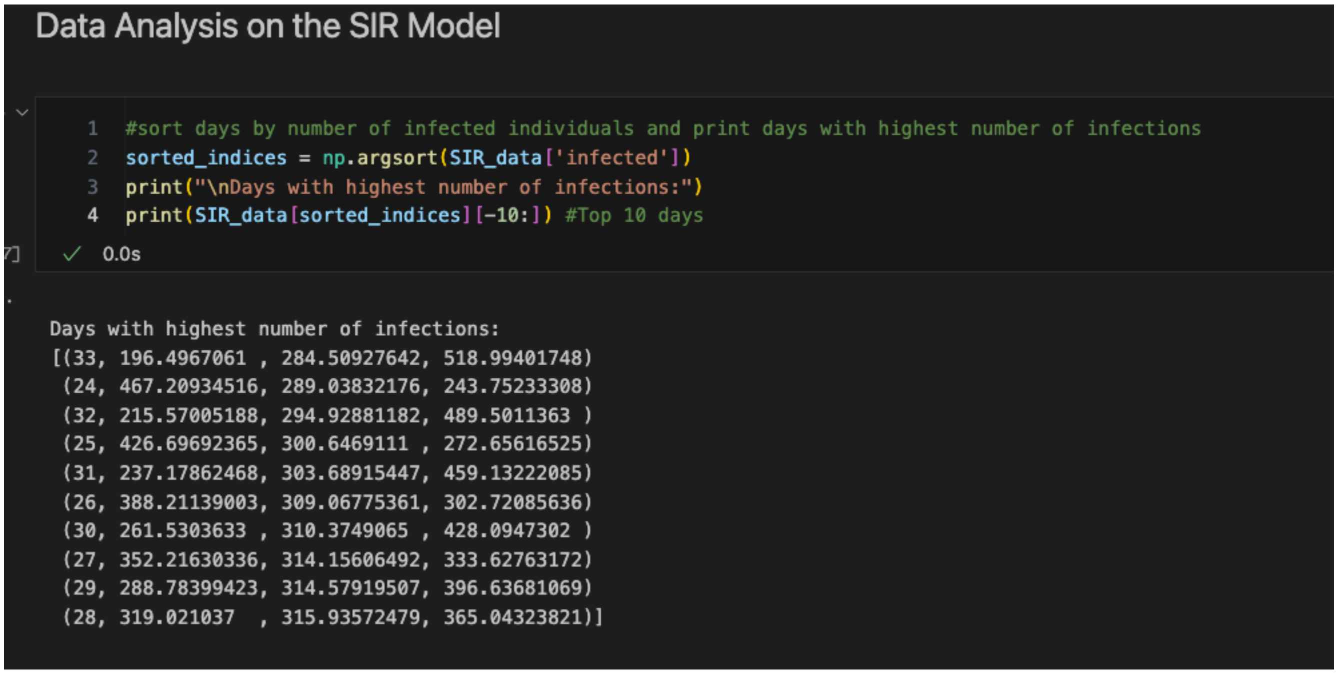
Figure 23: SIR Model Code


infection_spread_example.ipynb
infection_spread_example.ipynb > ...
Code
Markdown
Run All
Restart
Clear All Outputs
Variables
Outline

```python
    return data

# --------------------------
# Main Program
# --------------------------

# Initial values
S0 = 990 #Number of susceptible individuals
I0 = 10 #Number of infected individuals
R0 = 0 #Number of recovered individuals
beta = 0.3 #infection rate
gamma = 0.1 #recovery rate
days = 160 #Number of days to simulate

SIR_data = SIR_model(S0, I0, R0, beta, gamma, days)

print("\nSusceptible:\n", SIR_data['susceptible'])
print("\nInfected:\n", SIR_data['infected'])
print("\nRecovered:\n", SIR_data['recovered'])
```

Figure 24: SIR Model Code (final)

To show the benefit of using Python for modeling disease infection, various data analyses were performed. The first analysis was using `*np.argsort*` to sort the days by the number of infected individuals. With this information stored in a variable, using array indexing, the program prints to the screen the days with the highest number of infections. Figure 25 shows the code and output.


Data Analysis on the SIR Model

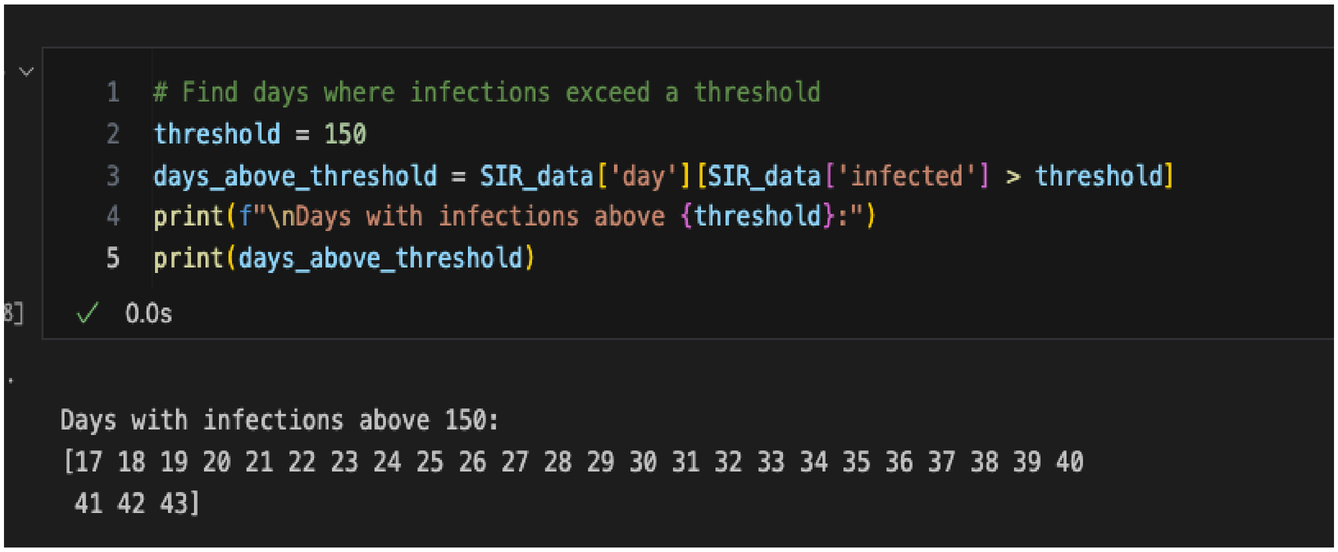
```
#sort days by number of infected individuals and print days with highest number of infections
sorted_indices = np.argsort(SIR_data['infected'])
print("\nDays with highest number of infections:")
print(SIR_data[sorted_indices][-10:]) #Top 10 days
```

✓ 0.0s

```
Days with highest number of infections:
[(33, 196.4967061 , 284.50927642, 518.99401748)
 (24, 467.20934516, 289.03832176, 243.75233308)
 (32, 215.57005188, 294.92881182, 489.5011363 )
 (25, 426.69692365, 300.6469111 , 272.65616525)
 (31, 237.17862468, 303.68915447, 459.13222085)
 (26, 388.21139003, 309.06775361, 302.72085636)
 (30, 261.5303633 , 310.3749065 , 428.0947302 )
 (27, 352.21630336, 314.15606492, 333.62763172)
 (29, 288.78399423, 314.57919507, 396.63681069)
 (28, 319.021037  , 315.93572479, 365.04323821)]
```

Figure 25: SIR Model: Sorting number of infected individuals

The next analysis uses fancy indexing and boolean indexing to access array elements to find the days where the number of infected individuals exceeds a certain threshold. Since fancy indexing allows access to elements in an arbitrary order, it is best used here. The boolean indexing is used to set the condition that must be met. Figure 26 shows the code and the output.



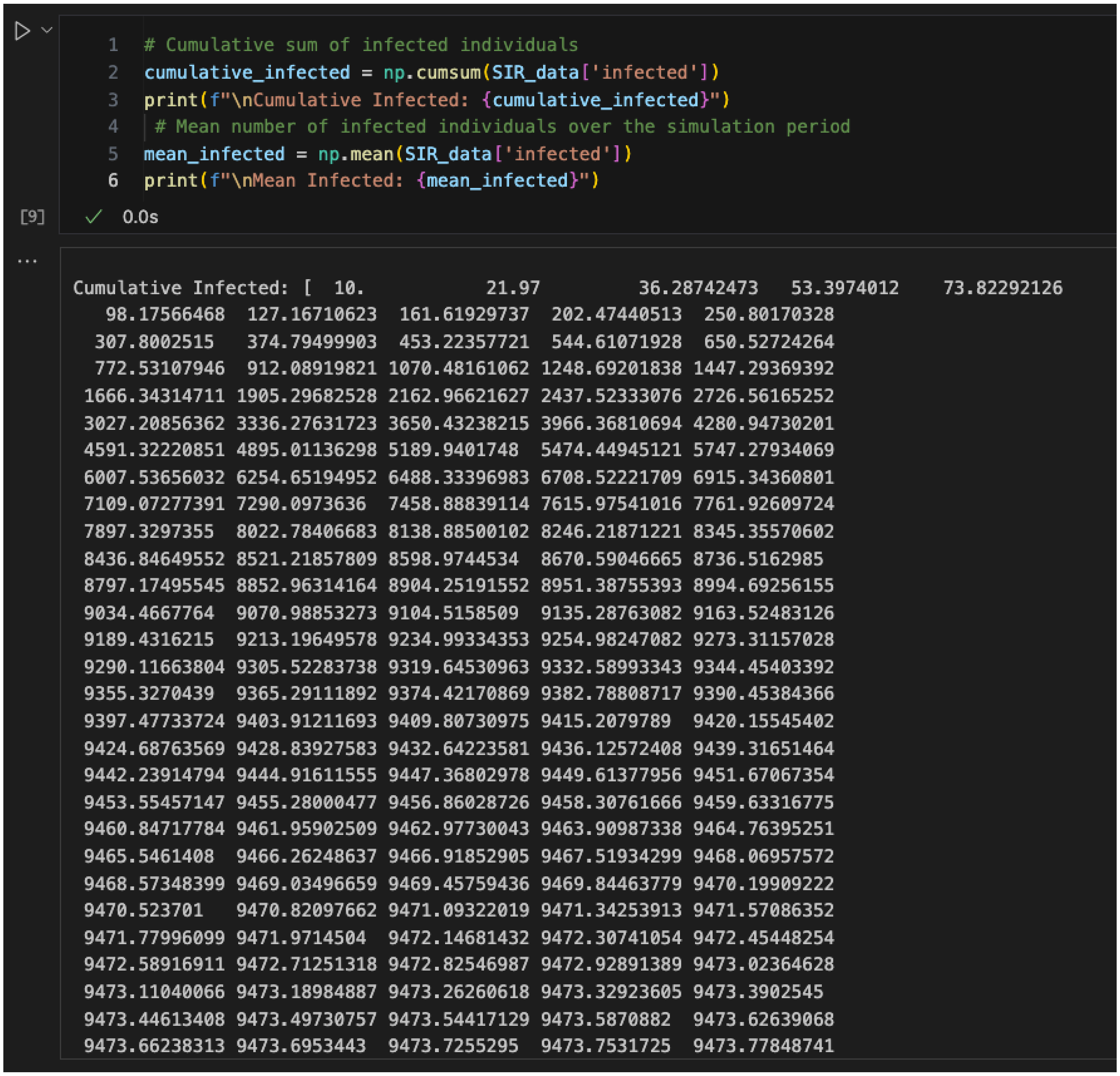
```
# Find days where infections exceed a threshold
threshold = 150
days_above_threshold = SIR_data['day'][SIR_data['infected'] > threshold]
print(f"\nDays with infections above {threshold}:")
print(days_above_threshold)
```

✓ 0.0s

```
Days with infections above 150:
[17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
 41 42 43]
```

Figure 26: SIR Model: Boolean indexing to find where infections exceed a threshold

The next analyses provide statistical information. It uses the built in NumPy cumulative summation and mean methods to calculate the cumulative sum of infected individuals and the mean number of infected individuals over the simulation period, and prints the results to the screen. Figure 27 and 28 shows the code and output. Using *`np.max`* and *`np.argmax`*, the peak number of infections are able to be found and the day this occurred is able to be quickly produced. Figure 29 shows the code and output.

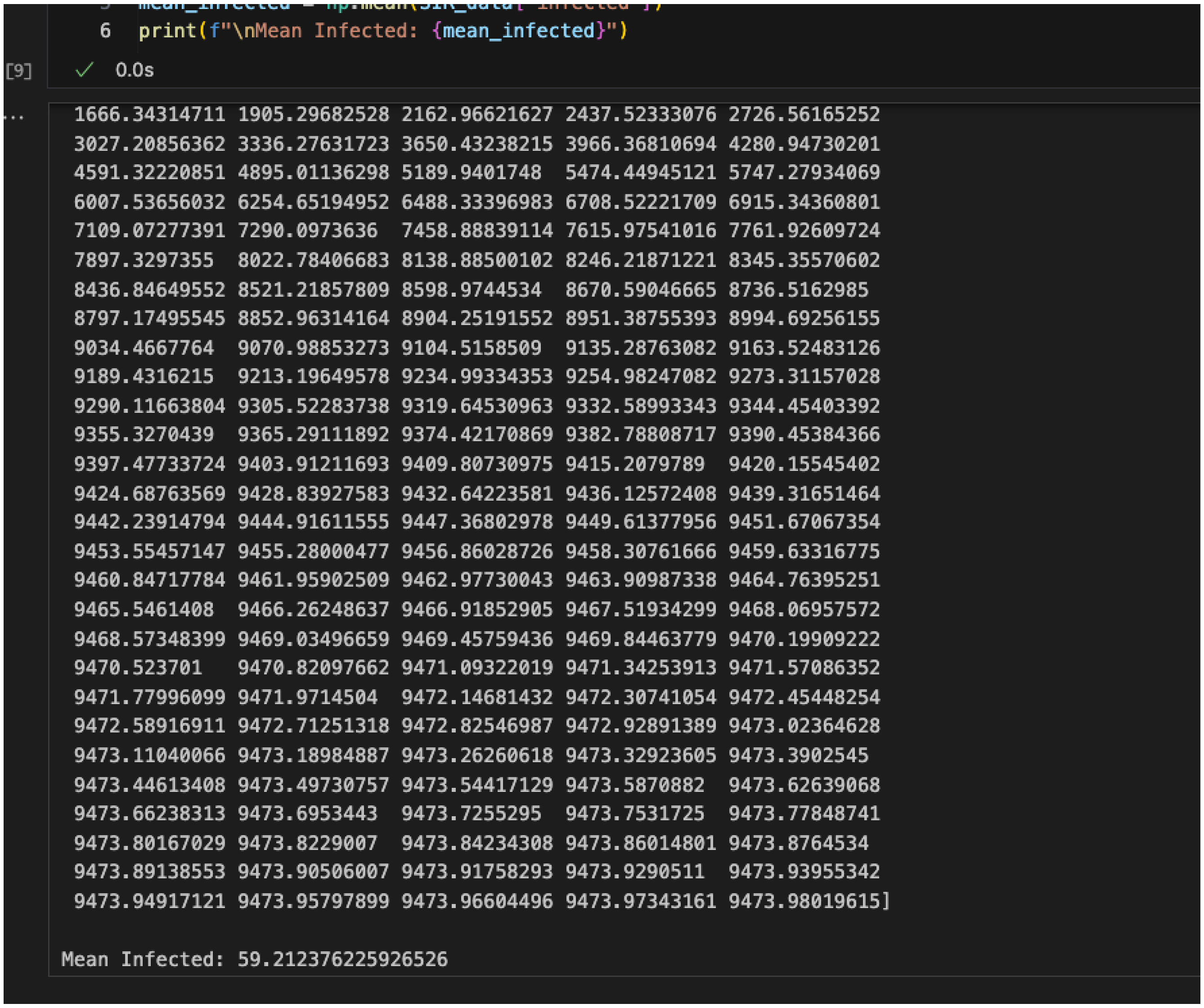


```python
# Cumulative sum of infected individuals
cumulative_infected = np.cumsum(SIR_data['infected'])
print(f"\nCumulative Infected: {cumulative_infected}")
 # Mean number of infected individuals over the simulation period
mean_infected = np.mean(SIR_data['infected'])
print(f"\nMean Infected: {mean_infected}")
```

[9] ✓ 0.0s

```
Cumulative Infected: [  10.           21.97         36.28742473   53.3974012    73.82292126
   98.17566468  127.16710623  161.61929737  202.47440513  250.80170328
  307.8002515   374.79499903  453.22357721  544.61071928  650.52724264
  772.53107946  912.08919821 1070.48161062 1248.69201838 1447.29369392
 1666.34314711 1905.29682528 2162.96621627 2437.52333076 2726.56165252
 3027.20856362 3336.27631723 3650.43238215 3966.36810694 4280.94730201
 4591.32220851 4895.01136298 5189.9401748  5474.44945121 5747.27934069
 6007.53656032 6254.65194952 6488.33396983 6708.52221709 6915.34360801
 7109.07277391 7290.0973636  7458.88839114 7615.97541016 7761.92609724
 7897.3297355  8022.78406683 8138.88500102 8246.21871221 8345.35570602
 8436.84649552 8521.21857809 8598.9744534  8670.59046665 8736.5162985
 8797.17495545 8852.96314164 8904.25191552 8951.38755393 8994.69256155
 9034.4667764  9070.98853273 9104.5158509  9135.28763082 9163.52483126
 9189.4316215  9213.19649578 9234.99334353 9254.98247082 9273.31157028
 9290.11663804 9305.52283738 9319.64530963 9332.58993343 9344.45403392
 9355.3270439  9365.29111892 9374.42170869 9382.78808717 9390.45384366
 9397.47733724 9403.91211693 9409.80730975 9415.2079789  9420.15545402
 9424.68763569 9428.83927583 9432.64223581 9436.12572408 9439.31651464
 9442.23914794 9444.91611555 9447.36802978 9449.61377956 9451.67067354
 9453.55457147 9455.28000477 9456.86028726 9458.30761666 9459.63316775
 9460.84717784 9461.95902509 9462.97730043 9463.90987338 9464.76395251
 9465.5461408  9466.26248637 9466.91852905 9467.51934299 9468.06957572
 9468.57348399 9469.03496659 9469.45759436 9469.84463779 9470.19909222
 9470.523701   9470.82097662 9471.09322019 9471.34253913 9471.57086352
 9471.77996099 9471.9714504  9472.14681432 9472.30741054 9472.45448254
 9472.58916911 9472.71251318 9472.82546987 9472.92891389 9473.02364628
 9473.11040066 9473.18984887 9473.26260618 9473.32923605 9473.3902545
 9473.44613408 9473.49730757 9473.54417129 9473.5870882  9473.62639068
 9473.66238313 9473.6953443  9473.7255295  9473.7531725  9473.77848741
```

Figure 27: SIR Model: Using universal functions `cumsum` and `mean`

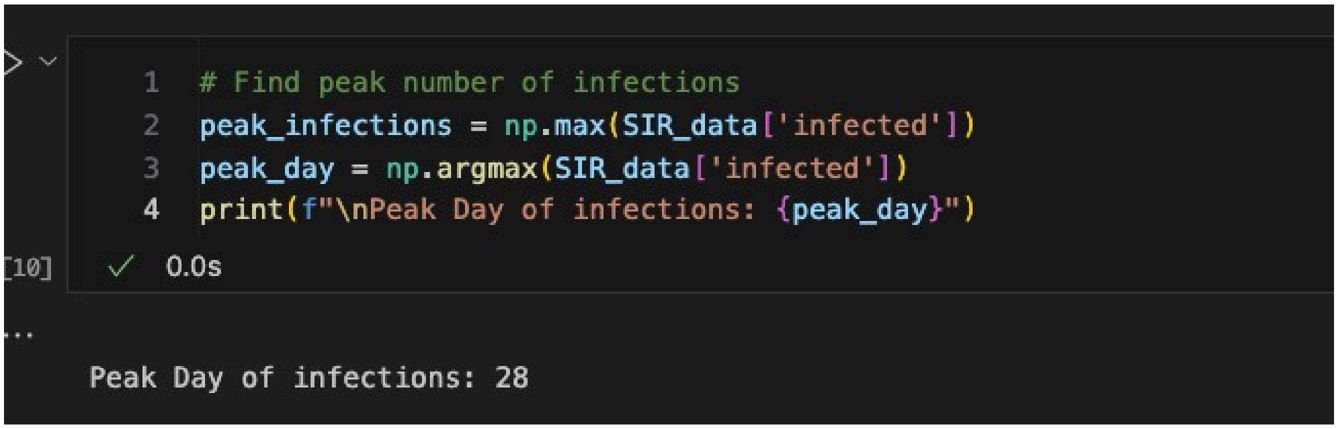


```
print(f"\nMean Infected: {mean_infected}")
```

```
[9]  ✓  0.0s
```

```
 1666.34314711 1905.29682528 2162.96621627 2437.52333076 2726.56165252
 3027.20856362 3336.27631723 3650.43238215 3966.36810694 4280.94730201
 4591.32220851 4895.01136298 5189.9401748  5474.44945121 5747.27934069
 6007.53656032 6254.65194952 6488.33396983 6708.52221709 6915.34360801
 7109.07277391 7290.0973636  7458.88839114 7615.97541016 7761.92609724
 7897.3297355  8022.78406683 8138.88500102 8246.21871221 8345.35570602
 8436.84649552 8521.21857809 8598.9744534  8670.59046665 8736.5162985
 8797.17495545 8852.96314164 8904.25191552 8951.38755393 8994.69256155
 9034.4667764  9070.98853273 9104.5158509  9135.28763082 9163.52483126
 9189.4316215  9213.19649578 9234.99334353 9254.98247082 9273.31157028
 9290.11663804 9305.52283738 9319.64530963 9332.58993343 9344.45403392
 9355.3270439  9365.29111892 9374.42170869 9382.78808717 9390.45384366
 9397.47733724 9403.91211693 9409.80730975 9415.2079789  9420.15545402
 9424.68763569 9428.83927583 9432.64223581 9436.12572408 9439.31651464
 9442.23914794 9444.91611555 9447.36802978 9449.61377956 9451.67067354
 9453.55457147 9455.28000477 9456.86028726 9458.30761666 9459.63316775
 9460.84717784 9461.95902509 9462.97730043 9463.90987338 9464.76395251
 9465.5461408  9466.26248637 9466.91852905 9467.51934299 9468.06957572
 9468.57348399 9469.03496659 9469.45759436 9469.84463779 9470.19909222
 9470.523701   9470.82097662 9471.09322019 9471.34253913 9471.57086352
 9471.77996099 9471.9714504  9472.14681432 9472.30741054 9472.45448254
 9472.58916911 9472.71251318 9472.82546987 9472.92891389 9473.02364628
 9473.11040066 9473.18984887 9473.26260618 9473.32923605 9473.3902545
 9473.44613408 9473.49730757 9473.54417129 9473.5870882  9473.62639068
 9473.66238313 9473.6953443  9473.7255295  9473.7531725  9473.77848741
 9473.80167029 9473.8229007  9473.84234308 9473.86014801 9473.8764534
 9473.89138553 9473.90506007 9473.91758293 9473.9290511  9473.93955342
 9473.94917121 9473.95797899 9473.96604496 9473.97343161 9473.98019615]

Mean Infected: 59.212376225926526
```

Figure 28: SIR Model: Using universal functions `cumsum` and `mean` (continued)

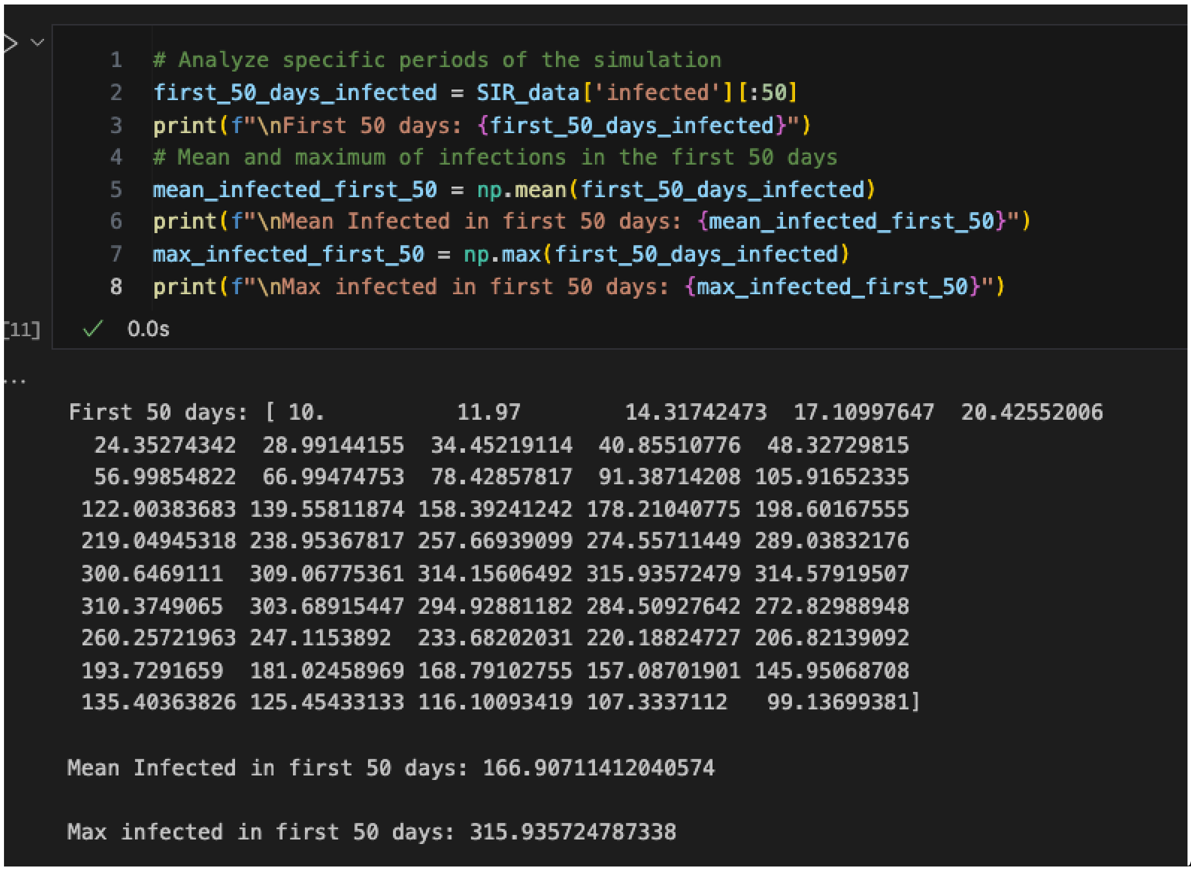


```
# Find peak number of infections
peak_infections = np.max(SIR_data['infected'])
peak_day = np.argmax(SIR_data['infected'])
print(f"\nPeak Day of infections: {peak_day}")
```

```
[10]  ✓  0.0s
...

    Peak Day of infections: 28
```

Figure 29: SIR Model: Using universal functions `max` and `argmax`

To further demonstrate the capabilities of using NumPy, the program was coded to analyze specific periods of the simulation. For this example, the first 50 days were analyzed. Using array slicing and aggregation, the data of the infected for the first 50 days was accessed and stored in the variable *`first_50_days_infected`*. With this information, the aggregation functions, *`mean`* and *`max`* were called and the results printed to the screen. Figure 30 below shows the code and output.

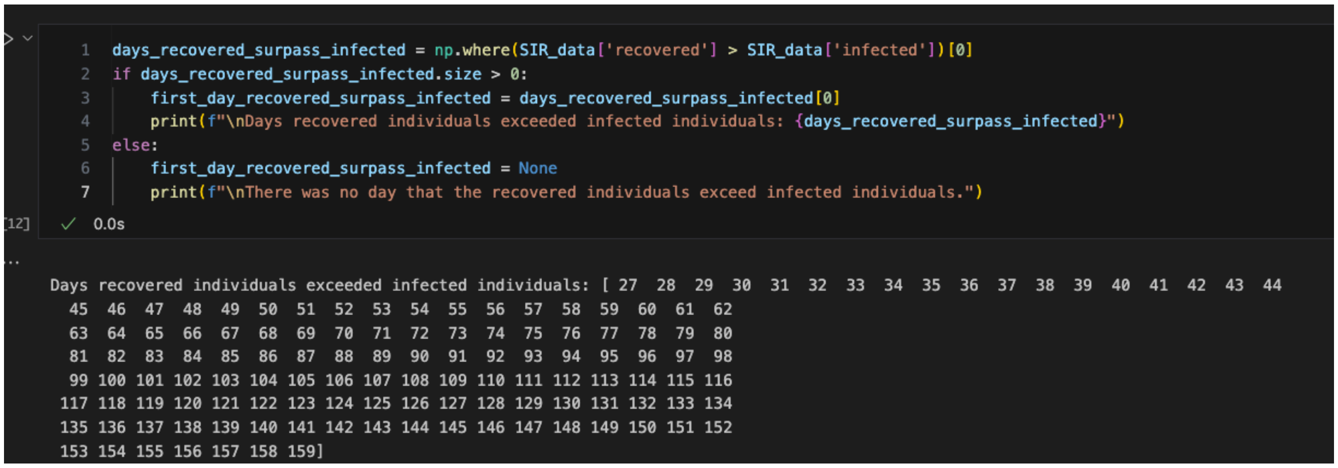


```
# Analyze specific periods of the simulation
first_50_days_infected = SIR_data['infected'][:50]
print(f"\nFirst 50 days: {first_50_days_infected}")
# Mean and maximum of infections in the first 50 days
mean_infected_first_50 = np.mean(first_50_days_infected)
print(f"\nMean Infected in first 50 days: {mean_infected_first_50}")
max_infected_first_50 = np.max(first_50_days_infected)
print(f"\nMax infected in first 50 days: {max_infected_first_50}")
```

```
[11]  ✓  0.0s
...

    First 50 days: [ 10.          11.97        14.31742473  17.10997647  20.42552006
      24.35274342  28.99144155  34.45219114  40.85510776  48.32729815
      56.99854822  66.99474753  78.42857817  91.38714208 105.91652335
     122.00383683 139.55811874 158.39241242 178.21040775 198.60167555
     219.04945318 238.95367817 257.66939099 274.55711449 289.03832176
     300.6469111  309.06775361 314.15606492 315.93572479 314.57919507
     310.3749065  303.68915447 294.92881182 284.50927642 272.82988948
     260.25721963 247.1153892  233.68202031 220.18824727 206.82139092
     193.7291659  181.02458969 168.79102755 157.08701901 145.95068708
     135.40363826 125.45433133 116.10093419 107.3337112   99.13699381]

    Mean Infected in first 50 days: 166.90711412040574

    Max infected in first 50 days: 315.935724787338
```

Figur e 30: SIR Model: First 50 days infected array access, mean, and max values

The next analysis using advanced NumPy indexing and boolean statements. Using `*np.where*`, the program determines the day when the number of recovered individuals surpasses the number of infected individuals. If it occurred, then the days where this occurred are printed to the screen, otherwise a statement alerts the user that this condition never occurred with the provided data. Figure 31 shows the code and output.



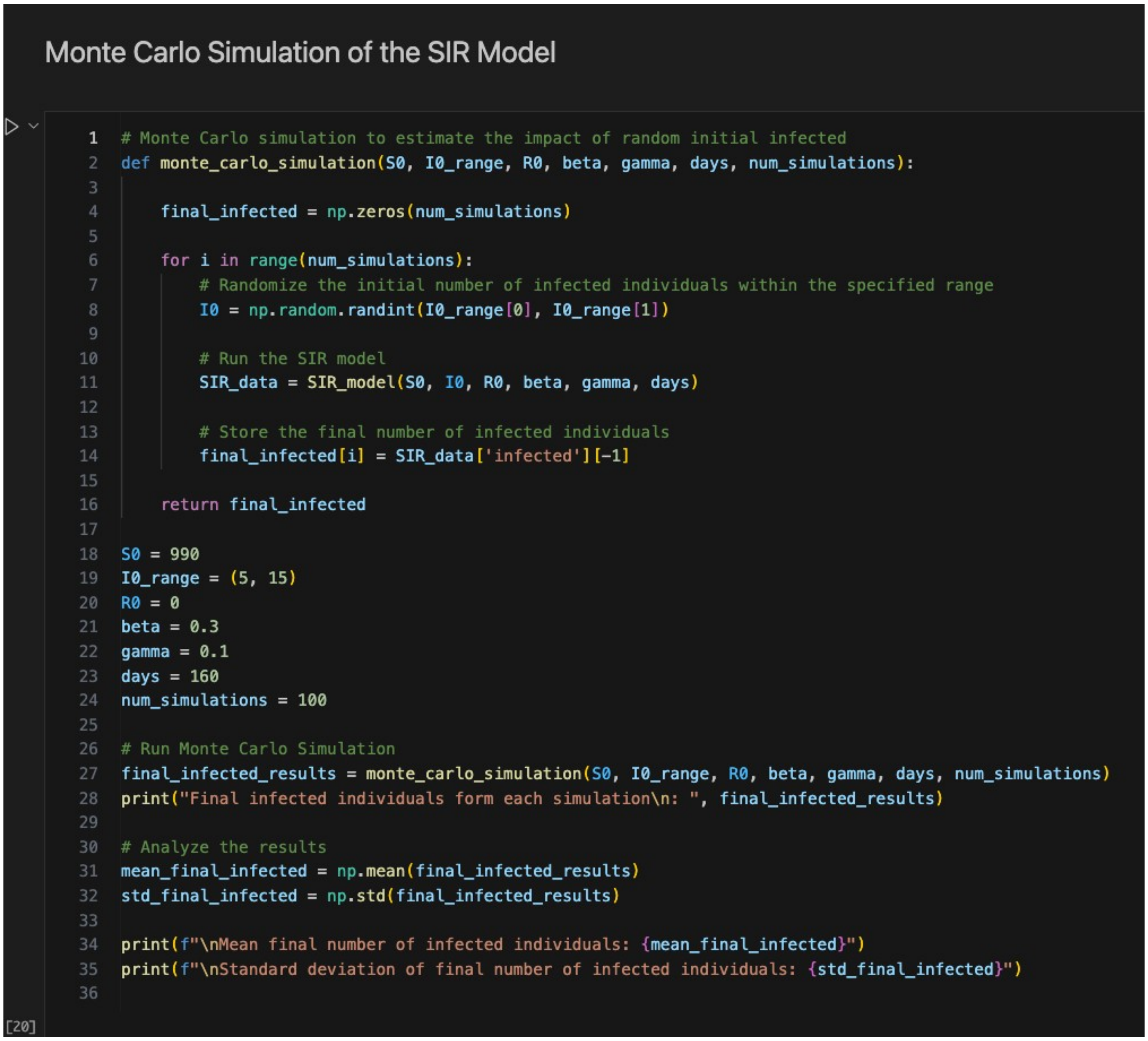
```
days_recovered_surpass_infected = np.where(SIR_data['recovered'] > SIR_data['infected'])[0]
if days_recovered_surpass_infected.size > 0:
    first_day_recovered_surpass_infected = days_recovered_surpass_infected[0]
    print(f"\nDays recovered individuals exceeded infected individuals: {days_recovered_surpass_infected}")
else:
    first_day_recovered_surpass_infected = None
    print(f"\nThere was no day that the recovered individuals exceed infected individuals.")
```

```
[12]  ✓  0.0s
```

```
Days recovered individuals exceeded infected individuals: [ 27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44
  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62
  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98
  99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134
 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152
 153 154 155 156 157 158 159]
```

Figure 31: SIR Model: Days recovered individuals surpassed infected

The last analysis is a Monte Carlo simulation. The Monte Carlo simulation is used to estimate the impact of a random initial condition. For this example, only the initial number of infected, `*I0*`, was randomized. A new function, `*monte_carlo_simulation()*`, was defined. It declares a new array `*final_infected*` using `*np.zeros*`. With the given number of simulations to run through, the code uses a `*for*` loop to initialize the initial condition of `*I0*` and run the `*SIR_model*` function. The final number of infected individuals is accessed and stored in the `*final_infected*` array. Once the results are collected in an array, the `*final_infected_results*` are printed to the screen along with the mean and standard deviation using the appropriate universal functions. Figure 32 and 33 below shows the code and the results.

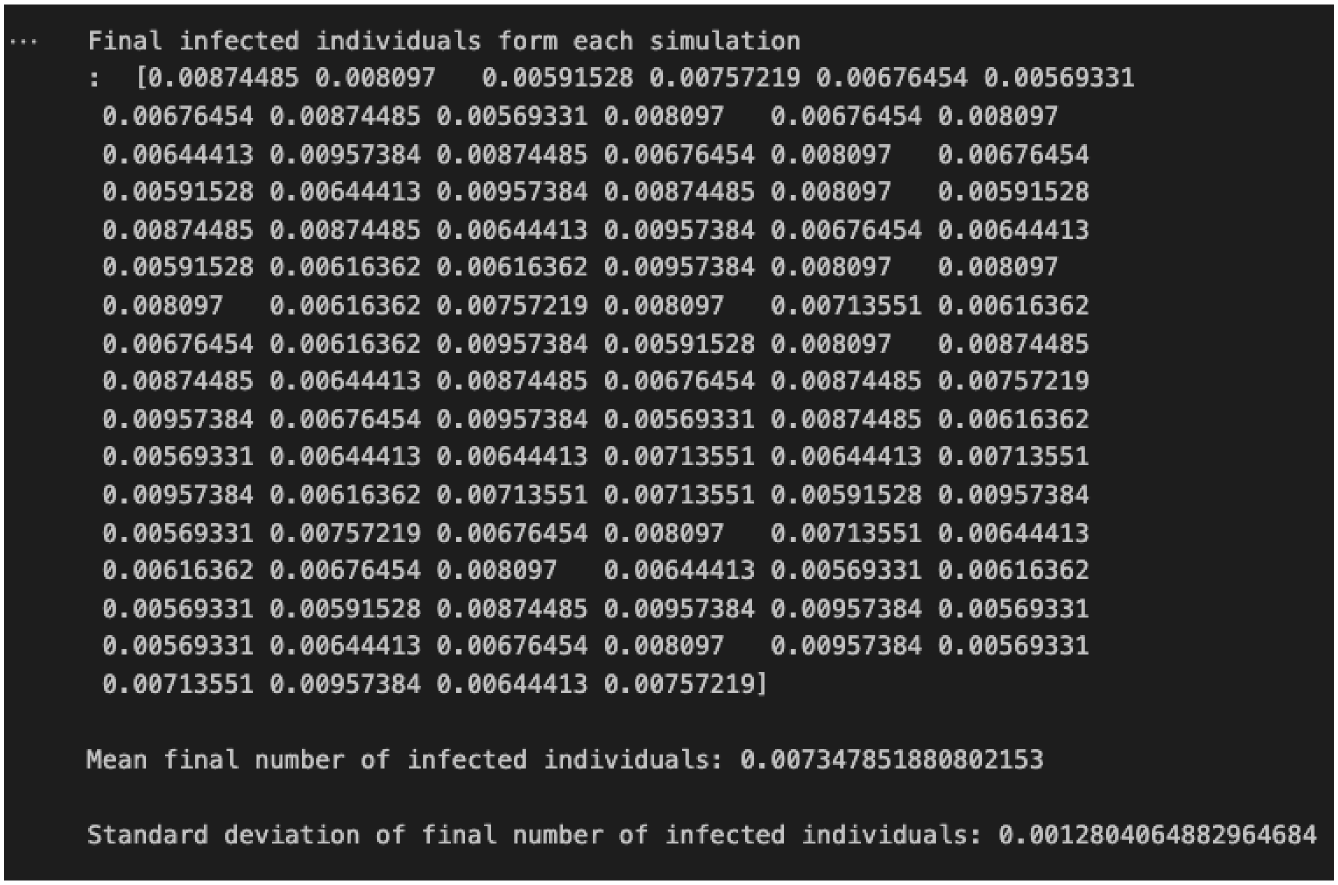


Monte Carlo Simulation of the SIR Model

```python
# Monte Carlo simulation to estimate the impact of random initial infected
def monte_carlo_simulation(S0, I0_range, R0, beta, gamma, days, num_simulations):

    final_infected = np.zeros(num_simulations)

    for i in range(num_simulations):
        # Randomize the initial number of infected individuals within the specified range
        I0 = np.random.randint(I0_range[0], I0_range[1])

        # Run the SIR model
        SIR_data = SIR_model(S0, I0, R0, beta, gamma, days)

        # Store the final number of infected individuals
        final_infected[i] = SIR_data['infected'][-1]

    return final_infected

S0 = 990
I0_range = (5, 15)
R0 = 0
beta = 0.3
gamma = 0.1
days = 160
num_simulations = 100

# Run Monte Carlo Simulation
final_infected_results = monte_carlo_simulation(S0, I0_range, R0, beta, gamma, days, num_simulations)
print("Final infected individuals form each simulation\n: ", final_infected_results)

# Analyze the results
mean_final_infected = np.mean(final_infected_results)
std_final_infected = np.std(final_infected_results)

print(f"\nMean final number of infected individuals: {mean_final_infected}")
print(f"\nStandard deviation of final number of infected individuals: {std_final_infected}")
```

[20]

Figure 32: Monte Carlo Simulation of the SIR Model Code

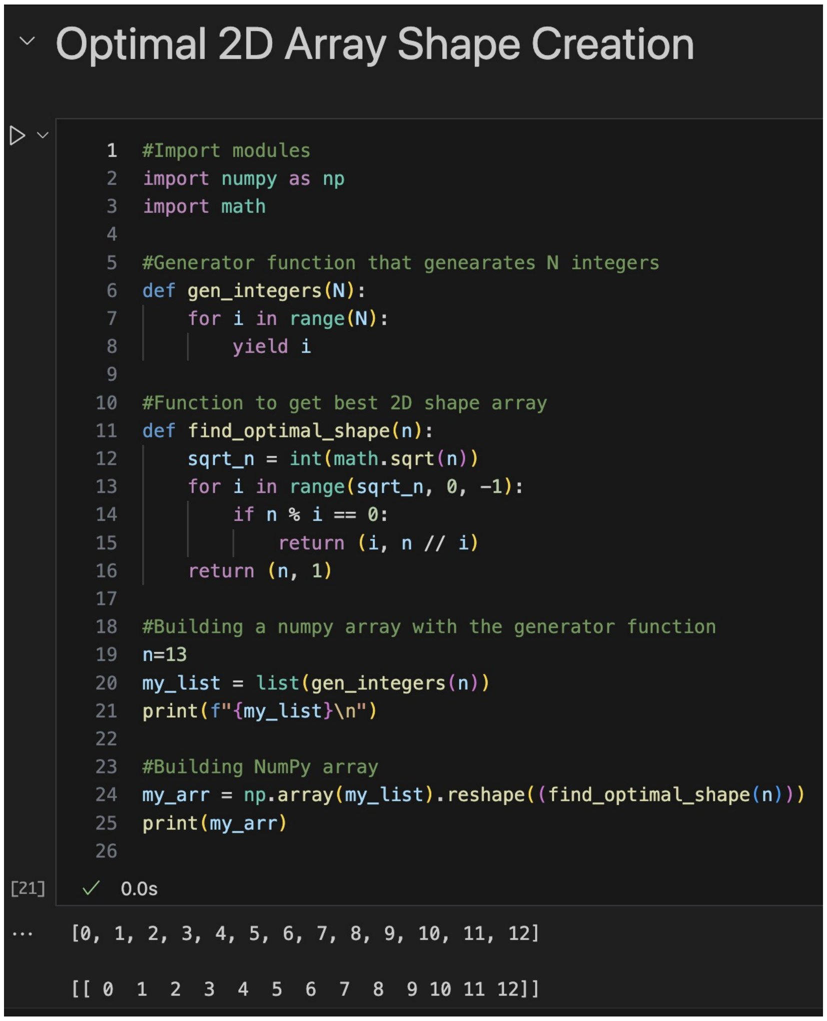


```
...   Final infected individuals form each simulation
      :  [0.00874485 0.008097   0.00591528 0.00757219 0.00676454 0.00569331
       0.00676454 0.00874485 0.00569331 0.008097   0.00676454 0.008097
       0.00644413 0.00957384 0.00874485 0.00676454 0.008097   0.00676454
       0.00591528 0.00644413 0.00957384 0.00874485 0.008097   0.00591528
       0.00874485 0.00874485 0.00644413 0.00957384 0.00676454 0.00644413
       0.00591528 0.00616362 0.00616362 0.00957384 0.008097   0.008097
       0.008097   0.00616362 0.00757219 0.008097   0.00713551 0.00616362
       0.00676454 0.00616362 0.00957384 0.00591528 0.008097   0.00874485
       0.00874485 0.00644413 0.00874485 0.00676454 0.00874485 0.00757219
       0.00957384 0.00676454 0.00957384 0.00569331 0.00874485 0.00616362
       0.00569331 0.00644413 0.00644413 0.00713551 0.00644413 0.00713551
       0.00957384 0.00616362 0.00713551 0.00713551 0.00591528 0.00957384
       0.00569331 0.00757219 0.00676454 0.008097   0.00713551 0.00644413
       0.00616362 0.00676454 0.008097   0.00644413 0.00569331 0.00616362
       0.00569331 0.00591528 0.00874485 0.00957384 0.00957384 0.00569331
       0.00569331 0.00644413 0.00676454 0.008097   0.00957384 0.00569331
       0.00713551 0.00957384 0.00644413 0.00757219]

      Mean final number of infected individuals: 0.007347851880802153

      Standard deviation of final number of infected individuals: 0.0012804064882964684
```

Figure 33: Monte Carlo Simulation Output

5. Optimal Array Shape Creation

The python script shown below in Figure 34 is an exercise demonstrating how to create a NumPy array. This particular example utilizes a generator function named *`gen_integers(N)`* that yields integers from 0 to *`N`*. The generated integers are converted into a list and stored in the list variable named *`my_list`*. To make this exercise a little unique, another function was created named *`find_optimal_shape(n)`*. This function takes in the number of elements generated by *`gen_integers(N)`* and computes the best shape to make the NumPy array. This is accomplished by importing the *`math`* module and taking the square root of the number of elements in the array and converting that number to an integer and storing it in *`sqrt_n`*. Then, using a *`for`* loop, while counting down from *`sqrt_n`* to 0, if the index is a factor of the number of elements, *`n`*, then the shape of the array returned is the index as the rows and the quotient of *`n`* and *`i`*. The list of elements is then converted to a NumPy array and the *`reshape`* method is called with *`find_optimal_shape(n)`* as the argument. Figure 35-37 shows 3 different outputs to the program.

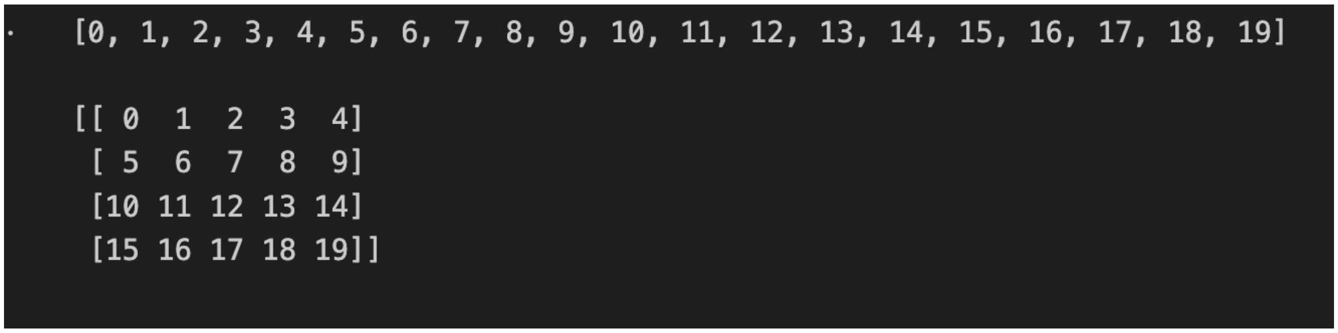

Optimal 2D Array Shape Creation

```python
#Import modules
import numpy as np
import math

#Generator function that genearates N integers
def gen_integers(N):
    for i in range(N):
        yield i

#Function to get best 2D shape array
def find_optimal_shape(n):
    sqrt_n = int(math.sqrt(n))
    for i in range(sqrt_n, 0, -1):
        if n % i == 0:
            return (i, n // i)
    return (n, 1)

#Building a numpy array with the generator function
n=13
my_list = list(gen_integers(n))
print(f"{my_list}\n")

#Building NumPy array
my_arr = np.array(my_list).reshape((find_optimal_shape(n)))
print(my_arr)
```

[21] ✓ 0.0s

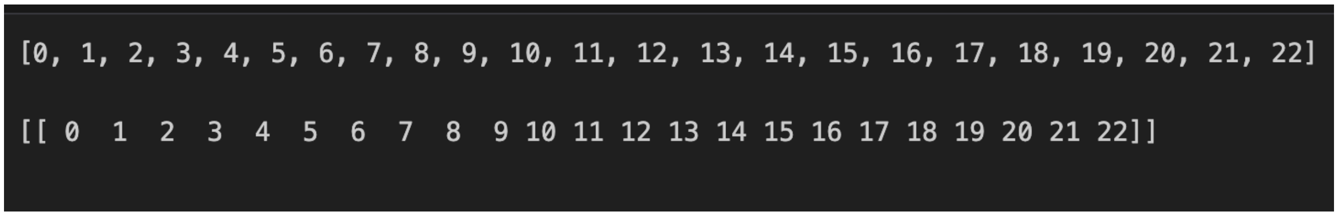
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

[[ 0  1  2  3  4  5  6  7  8  9 10 11 12]]
```

Figure 34: Optimal 2D Array Shape Creation Code and Output



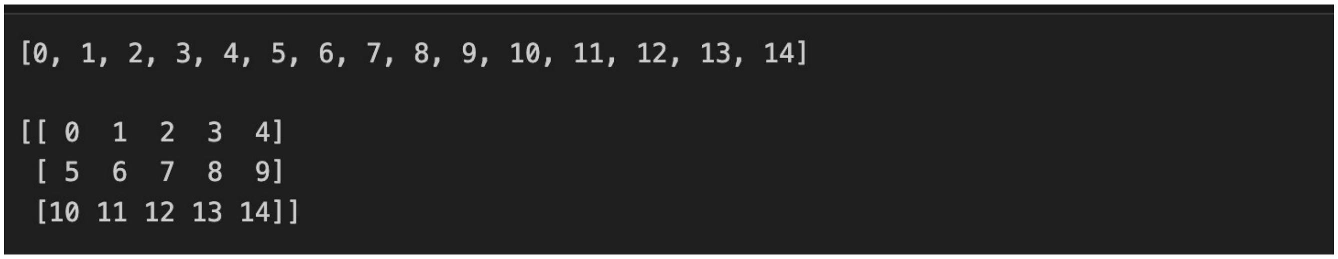
```
·   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

    [[ 0  1  2  3  4]
     [ 5  6  7  8  9]
     [10 11 12 13 14]
     [15 16 17 18 19]]
```

Figure 35: Optimal 2D Array Shape Creation: Output where n = 20



```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]

[[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22]]
```

Figure 36: Figure 35: Optimal 2D Array Shape Creation: Output where n = 23

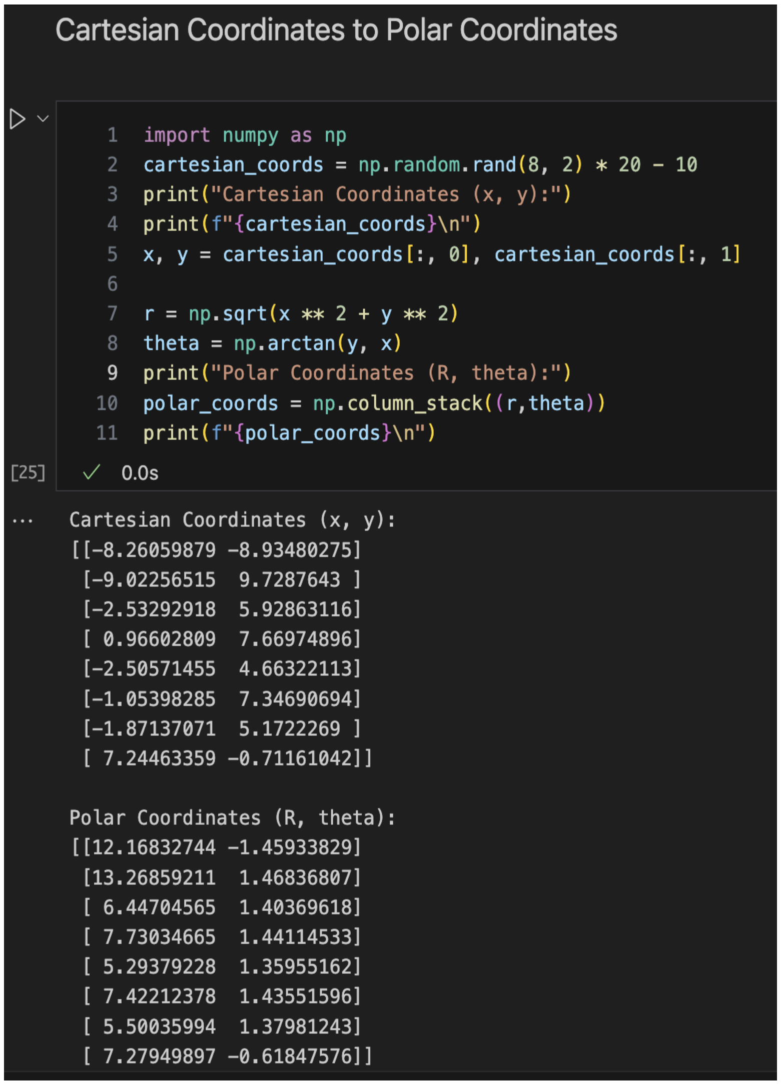


```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

Figure 37: Figure 35: Optimal 2D Array Shape Creation: Output where n = 15

6. Coordinate Conversion

This next exercise further demonstrates NumPy array creation, element-wise operations, slicing, and stacking (Rougier, N.P. (n.d.)). An 8 x 2 array is created with random values drawn from a uniform distribution over [0, 1). Multiplying by 20 and subtracting 10 scales the values to be in a range of [-10, 10). The cartesian coordinates NumPy array are printed to the screen using an f-string to format it. The array is sliced to separate the values that will be the *x* coordinates and *y* coordinates. The Euclidean distance and theta are computed for each coordinate pair using NumPy universal functions. Column stacking is then used to combine the *`r`* and *`theta`* arrays into a single 8 x 2 array. Figure 38 shows the final code and output to the program.


Cartesian Coordinates to Polar Coordinates

```python
import numpy as np
cartesian_coords = np.random.rand(8, 2) * 20 - 10
print("Cartesian Coordinates (x, y):")
print(f"{cartesian_coords}\n")
x, y = cartesian_coords[:, 0], cartesian_coords[:, 1]

r = np.sqrt(x ** 2 + y ** 2)
theta = np.arctan(y, x)
print("Polar Coordinates (R, theta):")
polar_coords = np.column_stack((r,theta))
print(f"{polar_coords}\n")
```

[25] ✓ 0.0s

```
Cartesian Coordinates (x, y):
[[-8.26059879 -8.93480275]
 [-9.02256515  9.7287643 ]
 [-2.53292918  5.92863116]
 [ 0.96602809  7.66974896]
 [-2.50571455  4.66322113]
 [-1.05398285  7.34690694]
 [-1.87137071  5.1722269 ]
 [ 7.24463359 -0.71161042]]

Polar Coordinates (R, theta):
[[12.16832744 -1.45933829]
 [13.26859211  1.46836807]
 [ 6.44704565  1.40369618]
 [ 7.73034665  1.44114533]
 [ 5.29379228  1.35955162]
 [ 7.42212378  1.43551596]
 [ 5.50035994  1.37981243]
 [ 7.27949897 -0.61847576]]
```

Figure 38: Coordinate Conversion Code and Output

7. Conclusion

This report documents my journey in mastering NumPy which involved grasping fundamental concepts such as arrays, computations using universal functions and broadcasting, incorporating comparisons and boolean logic into arrays, indexing and sorting arrays, and learning how to create structured arrays. Through the process of coding, many errors were encountered. The main issues centered around formulating the proper structure for the SIR model to ensure the code was readable but presented in a concise manner. Through testing, the final code was eventually created.

Engaging in mini-projects and working through the exercises has proved to be a valuable source in aiding me to take the theory of Python fundamentals and the NumPy package and put the concepts into practice and think about practical data science applications. This has allowed me to grasp the concepts that make the NumPy package a valuable tool for efficiently managing large sets of data.

References

1. Rougier, N.P. (n.d.). *Numpy 100.* Retrieved from https://github.com/rougier/numpy-100
2. Lowe, S., Mathis, J., & Wall, N. (2019). *Humans vs. Zombies Lab.* Unpublished paper, Mercer University.
3. VanderPlas, J. (*2016*). *Python Data Science Handbook*. O'Reilly Media. Retrieved from https://jakevdp.github.io/PythonDataScienceHandbook/index.html