Project 2: Introduction to Numpy

John Wesley Mathis

Dr. Anthony Choi

June 02, 2024

ECE/SSE 591, Summer 2024

Table of Contents:

[Deliverable Table 5](#__RefHeading___Toc2569_331549091)

[1. Introduction 6](#__RefHeading___Toc2573_331549091)

[2. Array CalculatorAnalysis 7](#__RefHeading___Toc2575_331549091)

[3. SIR Model Analysis 22](#__RefHeading___Toc2577_331549091)

[5. Optimal Array Shape Creation 32](#__RefHeading___Toc2583_331549091)

[6. Coordinate Conversion 33](#__RefHeading___Toc2585_331549091)

[7. Conclusion 35](#__RefHeading___Toc2609_331549091)

[References 36](#__RefHeading___Toc3692_331549091)

Table of Figures:

[*Figure 1: Array Calculator: Code* 8](#Figure!0|sequence)

[*Figure 2: Array Calculator: Code (continued)* 9](#Figure!1|sequence)

[*Figure 3: Array Calculator: Code (final)* 10](#Figure!2|sequence)

[*Figure 4: Array Calculator: Manually entering elements for an array* 11](#Figure!3|sequence)

[*Figure 5: Array Calculator: User defined array output and array operations menu* 11](#Figure!4|sequence)

[*Figure 6: Array Calculator: Squaring elements output* 12](#Figure!5|sequence)

[*Figure 7: Array Calculator: Square root option output* 12](#Figure!6|sequence)

[*Figure 8: Array Calculator: Exponential option output* 13](#Figure!7|sequence)

[*Figure 9: Array Calculator: Sine option output* 13](#Figure!8|sequence)

[*Figure 10: Array Calculator: Mean option output* 14](#Figure!9|sequence)

[*Figure 11: Array Calculator: Standard deviation option output* 14](#Figure!10|sequence)

[*Figure 12: Array Calculator: Max value option output* 15](#Figure!11|sequence)

[*Figure 13: Array Calculator: Min value option output* 15](#Figure!12|sequence)

[*Figure 14: Array Calculator: Exiting the program* 16](#Figure!13|sequence)

[*Figure 15: Array Calculator: Generating a random array option* 16](#Figure!14|sequence)

[*Figure 16: Array Calculator: Selecting array dimension* 17](#Figure!15|sequence)

[*Figure 17: Array Calculator: Defining column size* 18](#Figure!17|sequence)

[*Figure 18: Array Calculator: Defining data type* 18](#Figure!18|sequence)

[*Figure 19: Array Calculator: Defining lower bound* 19](#Figure!19|sequence)

[*Figure 20: Array Calculator: Defining upper bound* 19](#Figure!20|sequence)

[*Figure 21: Array Calculator: Random array output* 20](#Figure!21|sequence)

[Figure 22a: Array Calculator: Showing input handling 21](#Figure!16|sequence)

[*Figure 23: SIR Model Code* 22](#Figure!22|sequence)

[*Figure 24: SIR Model Code (final)* 23](#Figure!23|sequence)

[*Figure 25: SIR Model: Sorting number of infected individuals* 24](#Figure!24|sequence)

[*Figure 26: SIR Model: Boolean indexing to find where infections exceed a threshold* 24](#Figure!25|sequence)

[*Figure 27: SIR Model: Using universal functions `cumsum` and `mean`* 25](#Figure!26|sequence)

[*Figure 28: SIR Model: Using universal functions `cumsum` and `mean` (continued)* 27](#Figure!27|sequence)

[*Figure 29: SIR Model: Using universal functions `max` and `argmax`* 28](#Figure!28|sequence)

[*Figure 30: SIR Model: First 50 days infected array access, mean, and max values* 28](#Figure!29|sequence)

[*Figure 31: SIR Model: Days recovered individuals surpassed infected* 29](#Figure!30|sequence)

[*Figure 32: Monte Carlo Simulation of the SIR Model Code* 30](#Figure!31|sequence)

[*Figure 33: Monte Carlo Simulation Output* 31](#Figure!32|sequence)

[Figure 34: Optimal 2D Array Shape Creation Code and Output 32](#Figure!33|sequence)

[*Figure 35: Optimal 2D Array Shape Creation: Output where n = 20* 33](#Figure!34|sequence)

[*Figure 36: Figure 35: Optimal 2D Array Shape Creation: Output where n = 23* 33](#Figure!35|sequence)

[*Figure 37: Figure 35: Optimal 2D Array Shape Creation: Output where n = 15* 33](#Figure!36|sequence)

[*Figure 38: Coordinate Conversion Code and Output* 34](#Figure!37|sequence)

# **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

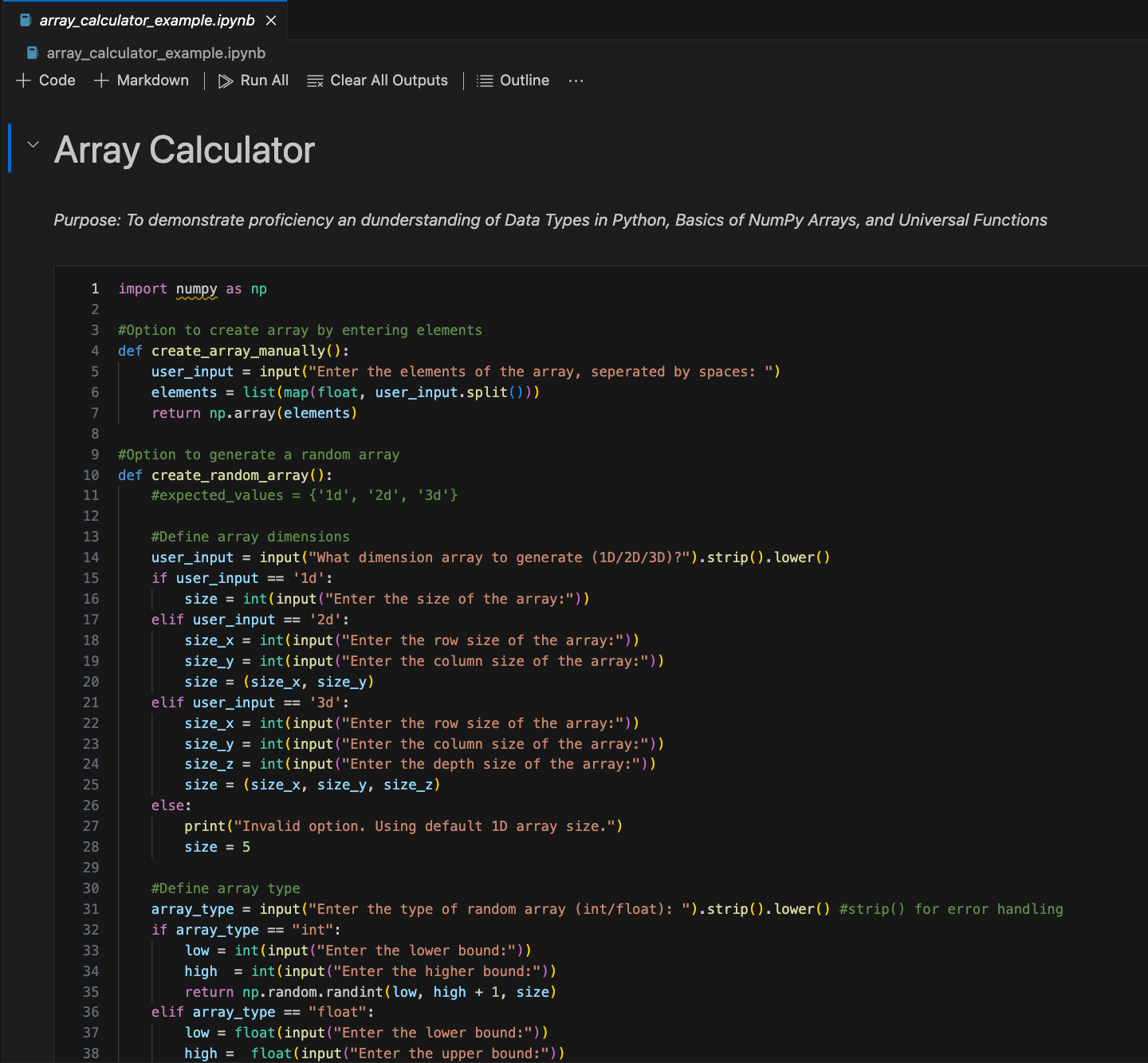
Python is a versatile language with numerous libraries. Because of its easy to understand syntax, it is very popular among the data science community as a useful tool to load, store, and manipulate data. Much of this data is typically managed by converting it into arrays of numbers. However, as useful 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.

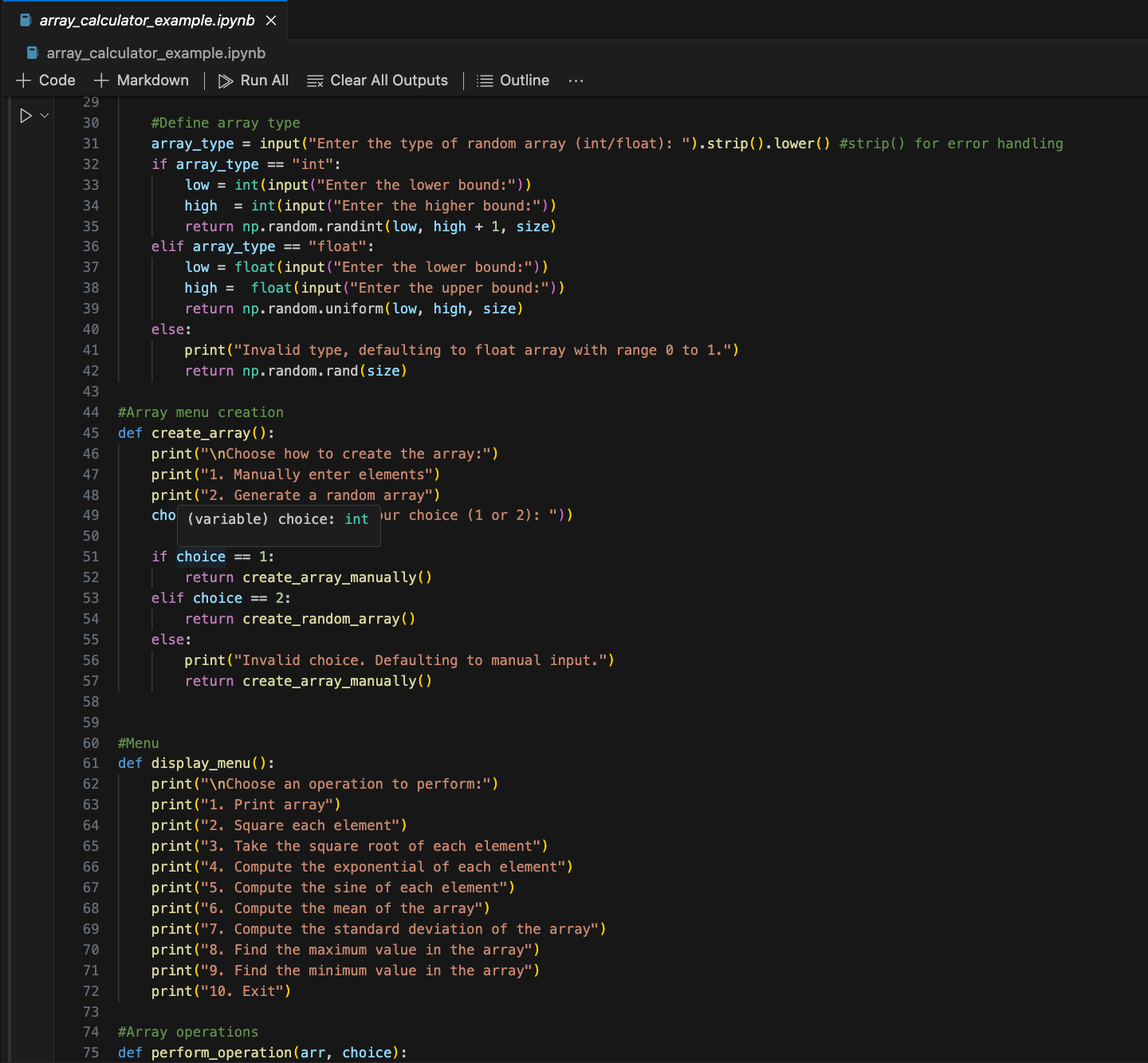
NumPy provides a manner to more efficiently handle data arrays. NumPy’s arrays are similar to Python’s built-in arrays but where it shines is how is handles storage as the data grows larger. As a result, NumPy has become the foundation for many scientific tools that are in use.

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.

*Figure 1: Array Calculator: Code*

*Figure 2: Array Calculator: Code (continued)*

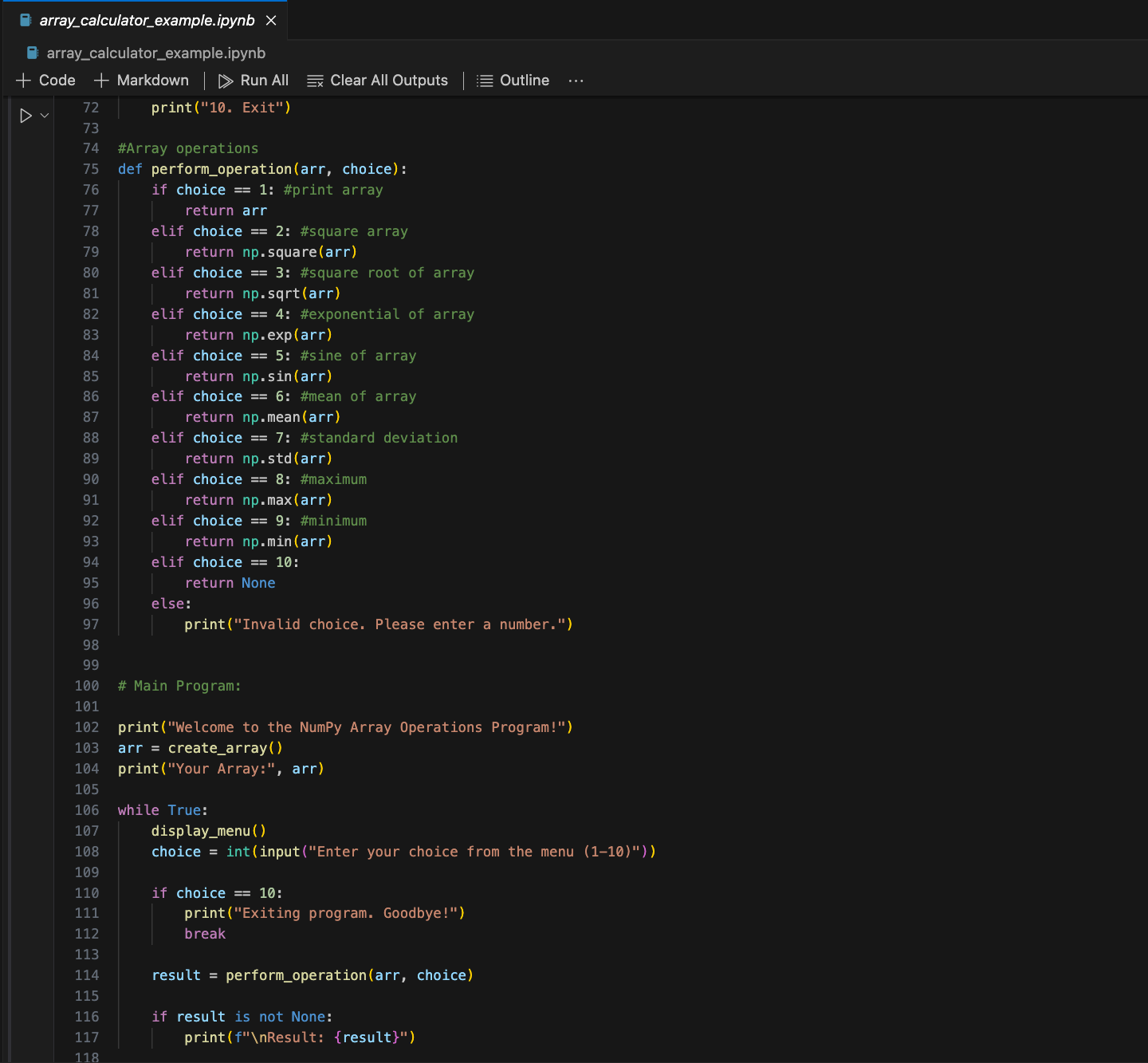
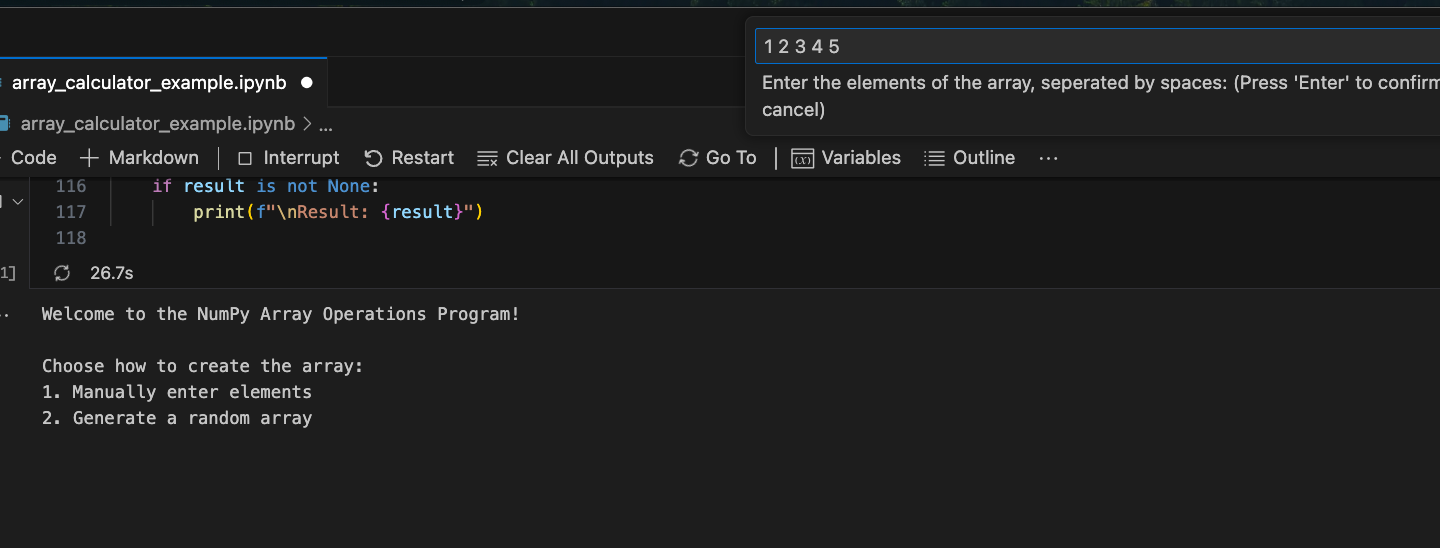
*Figure 3: Array Calculator: Code (final)*

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.

*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.

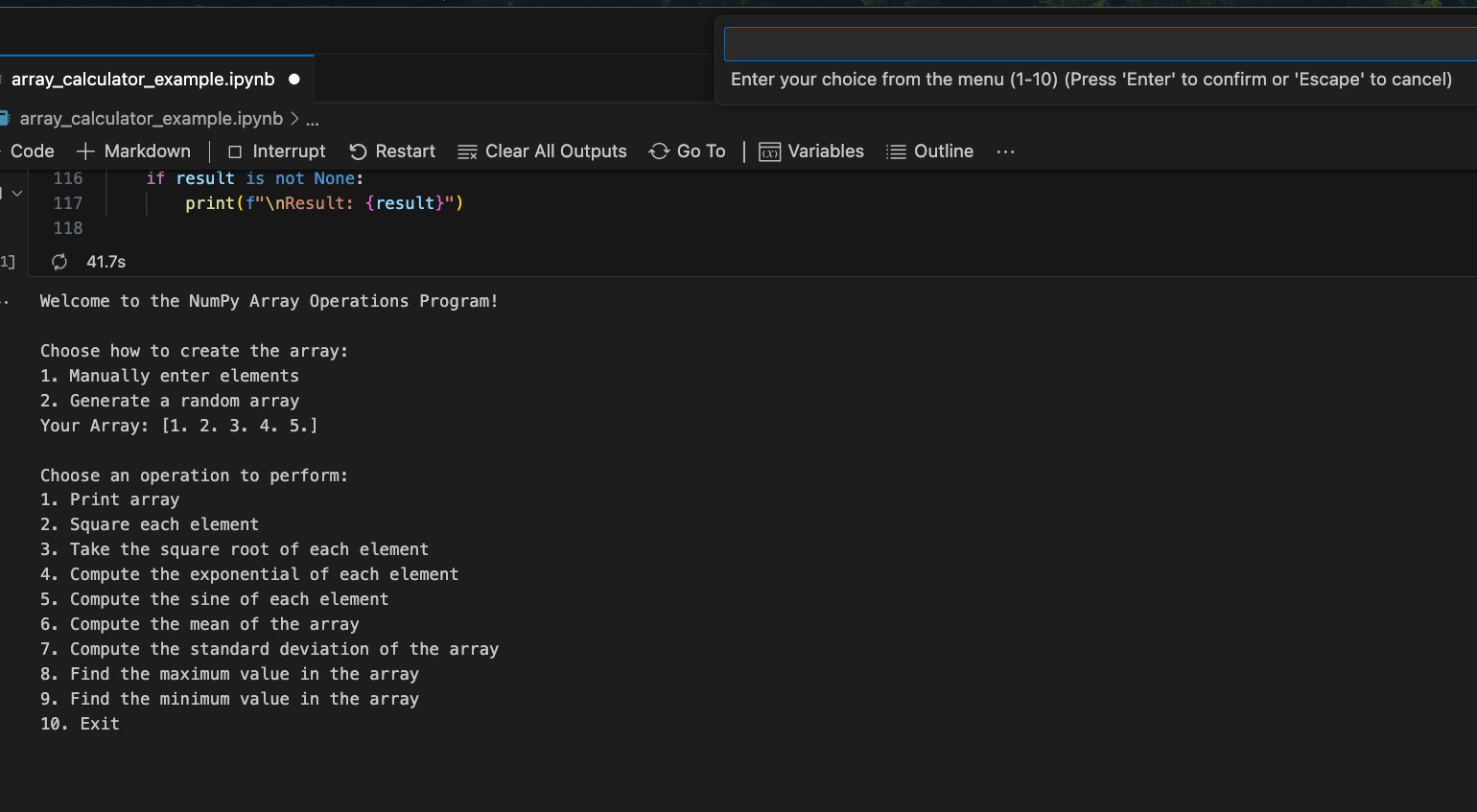
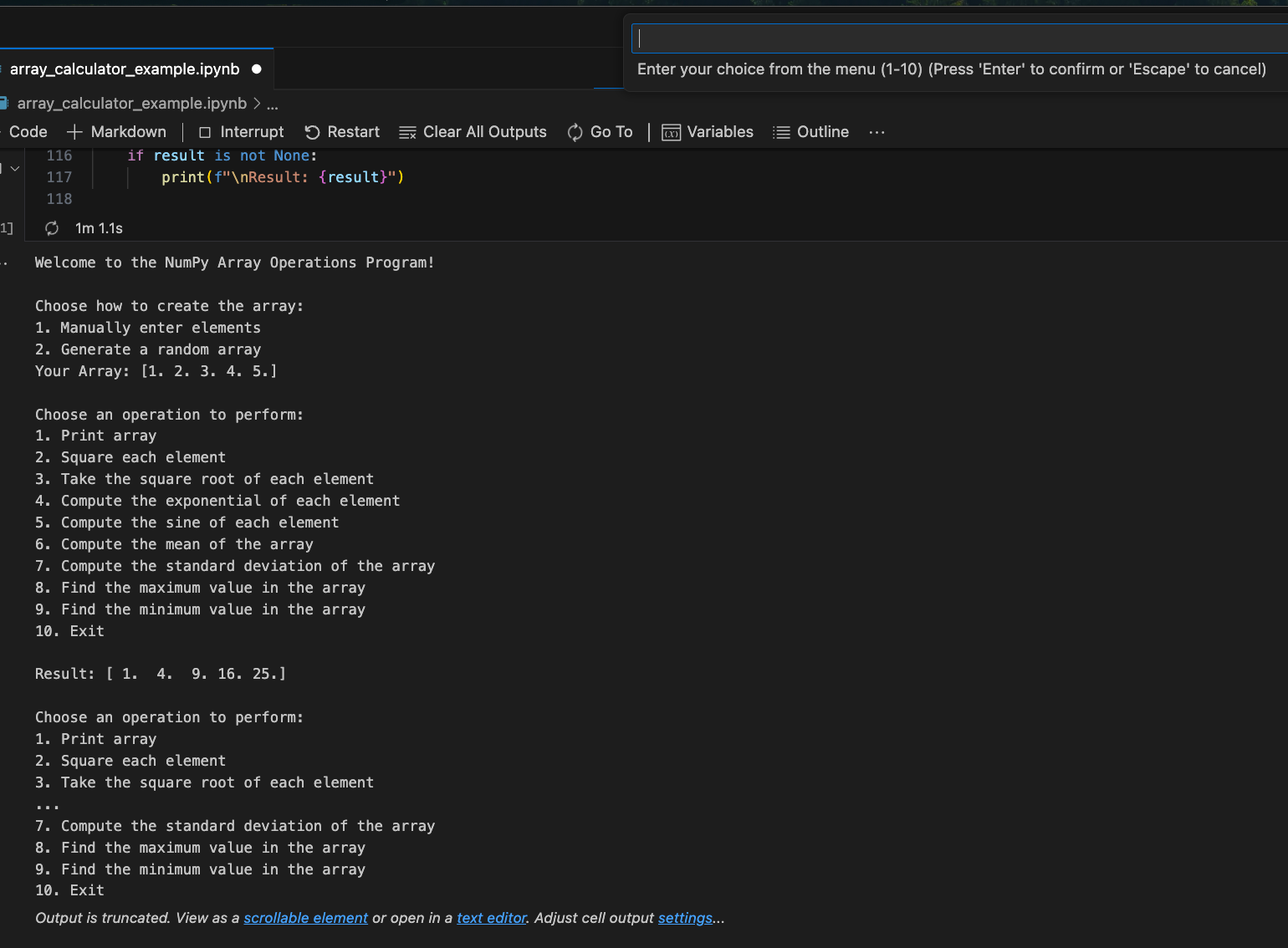
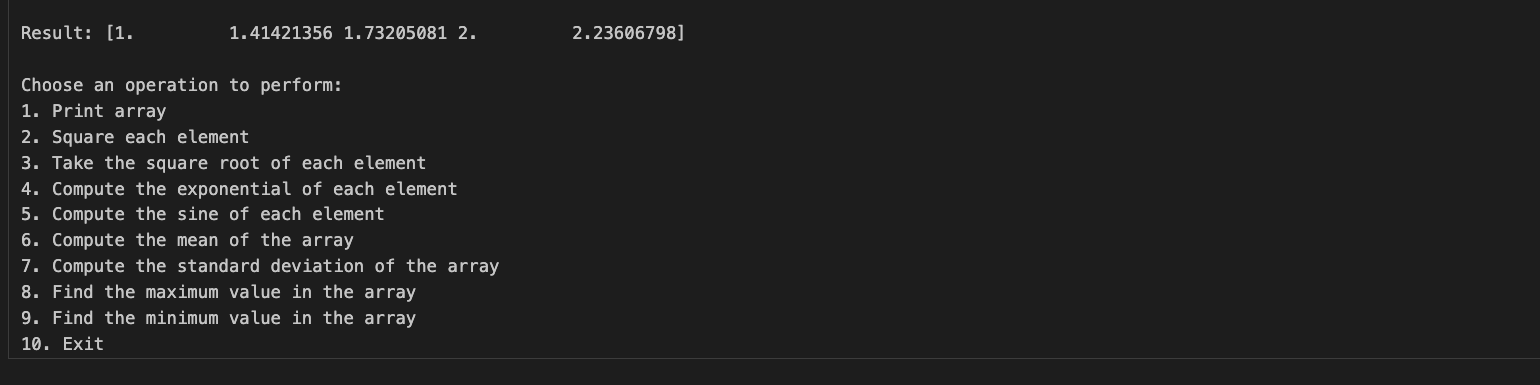
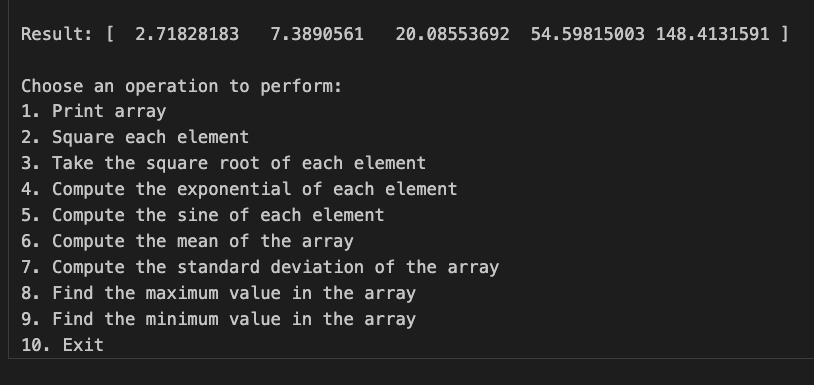
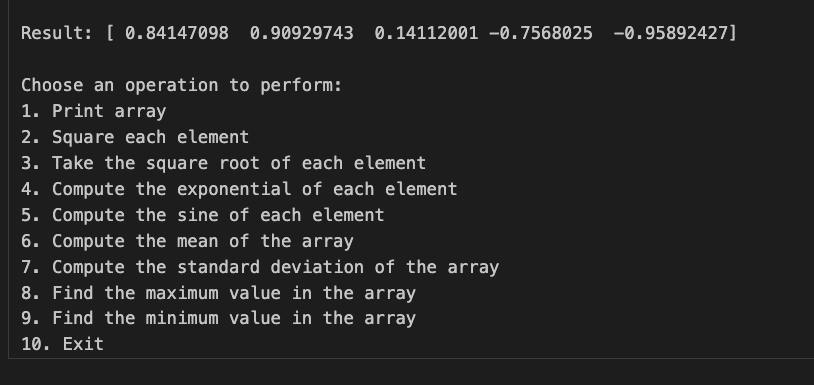
*Figure 5: Array Calculator: User defined array output and array operations menu*

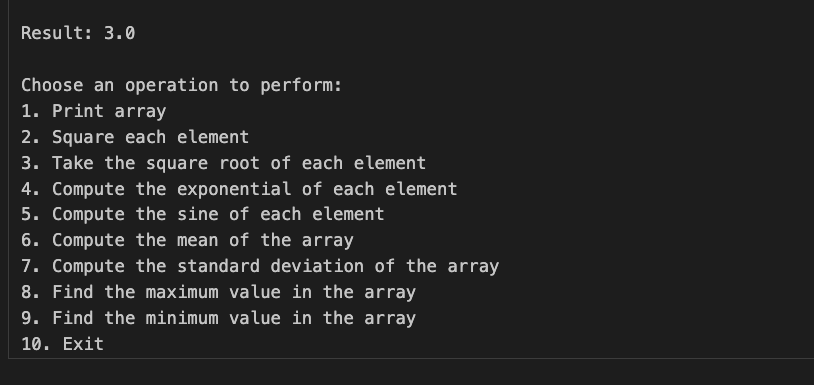
Figure 6-13 below shows the output of each option. As described previously, the operations are performed on the array by using the NumPy universal functions. These functions are accessed by the control flow *`if`* statements and by the user choice.

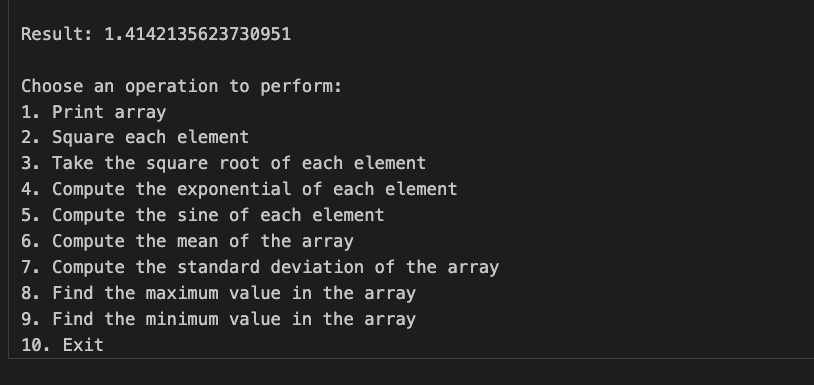
*Figure 6: Array Calculator: Squaring elements output*

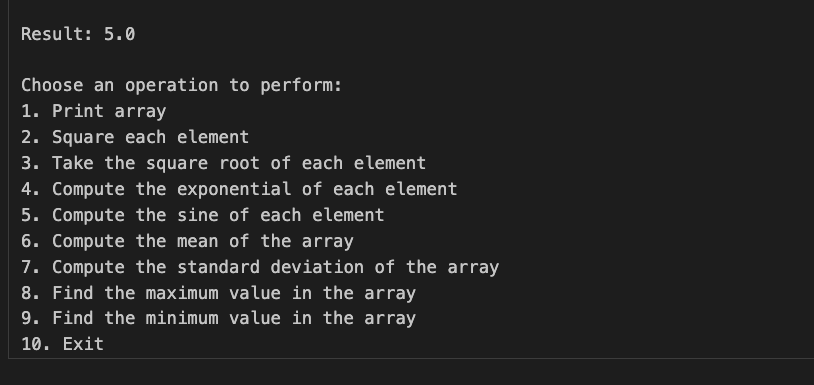
*Figure 7: Array Calculator: Square root option output*

*Figure 8: Array Calculator: Exponential option output*

*Figure 9: Array Calculator: Sine option output*

*Figure 10: Array Calculator: Mean option output*

*Figure 11: Array Calculator: Standard deviation option output*

*Figure 12: Array Calculator: Max value option output*

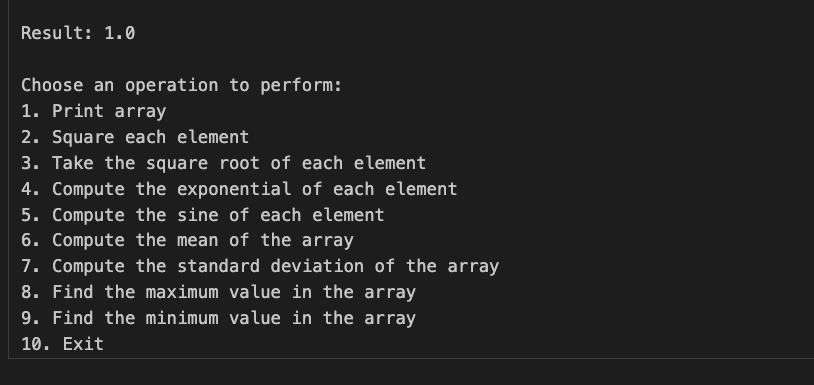
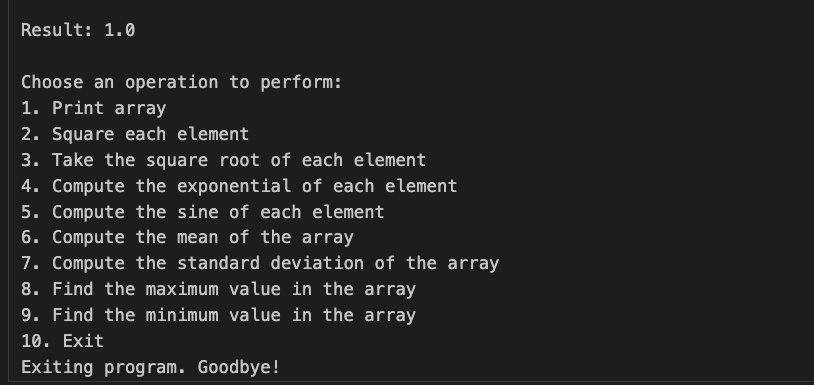
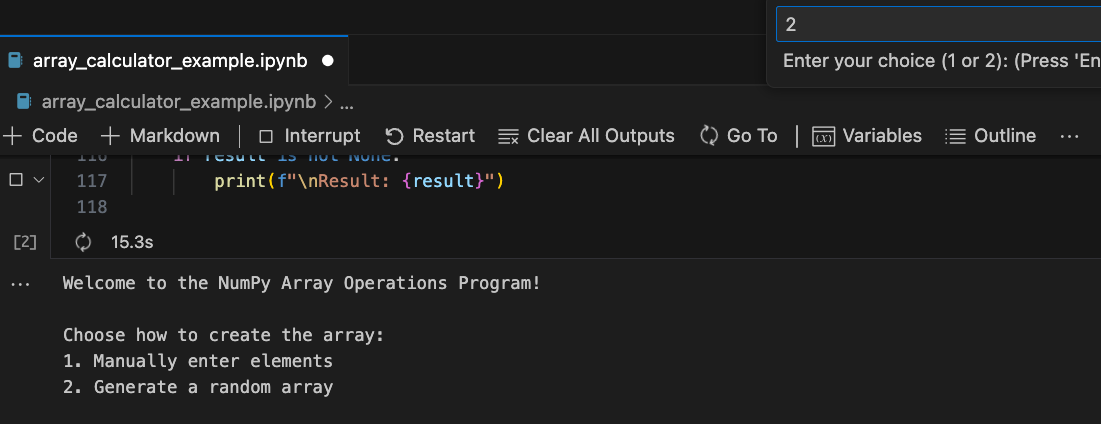
*Figure 13: Array Calculator: Min value option output*

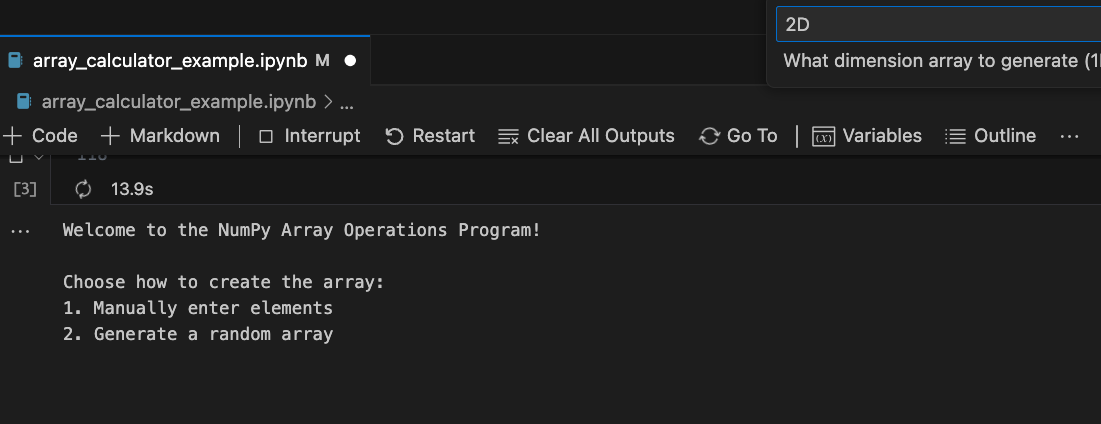
Figure 14 below shows the output if the user selects to exit the program. This functionality was implemented by simply using an *`if`* statement to check if the user as inputted the string ‘10’ to exit. If so, the program prints out an exiting statement and breaks out of the *`while`* loop.

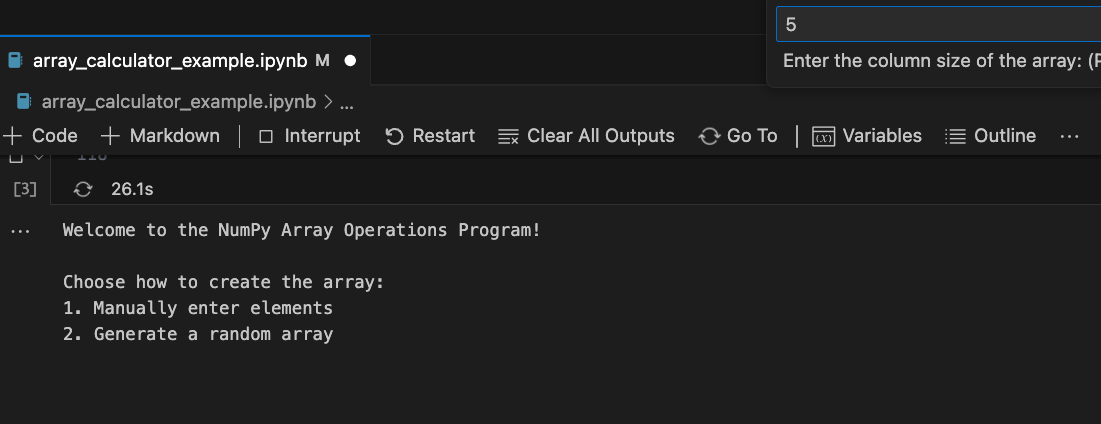
*Figure 14: Array Calculator: Exiting the program*

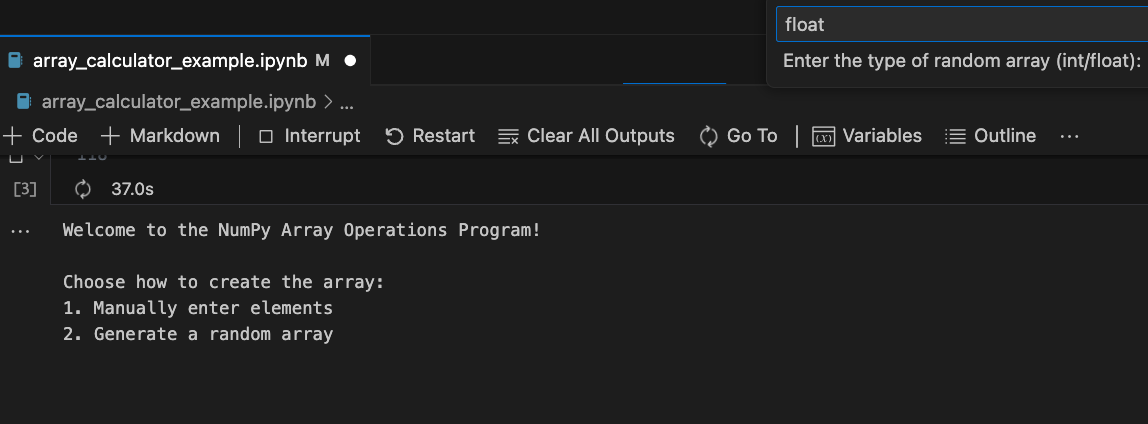
To make the program a little more unique, the user is given an option to randomly generate an array based on user defined conditions. To run this part of the program the user must select option 2 when asked how they would like to define the array. Figure 15 shows an example.

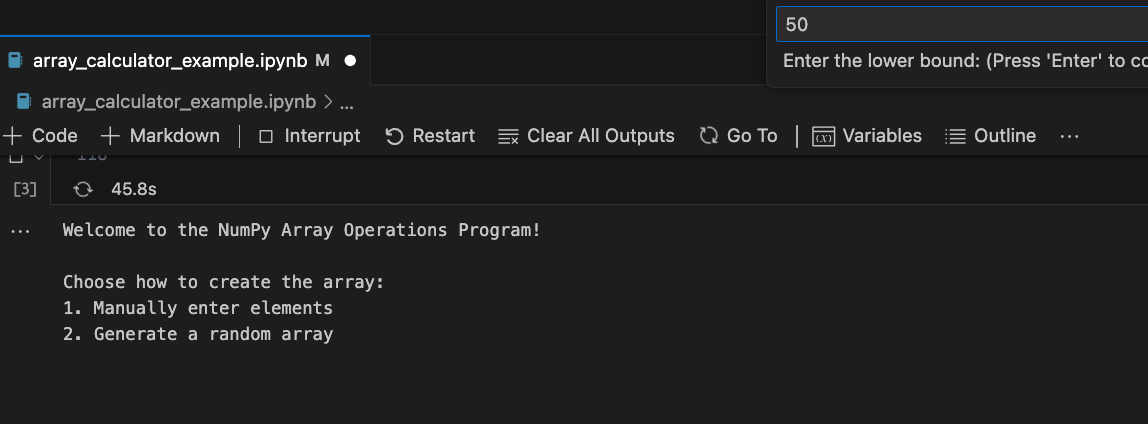
*Figure 15: Array Calculator: Generating a random array option*

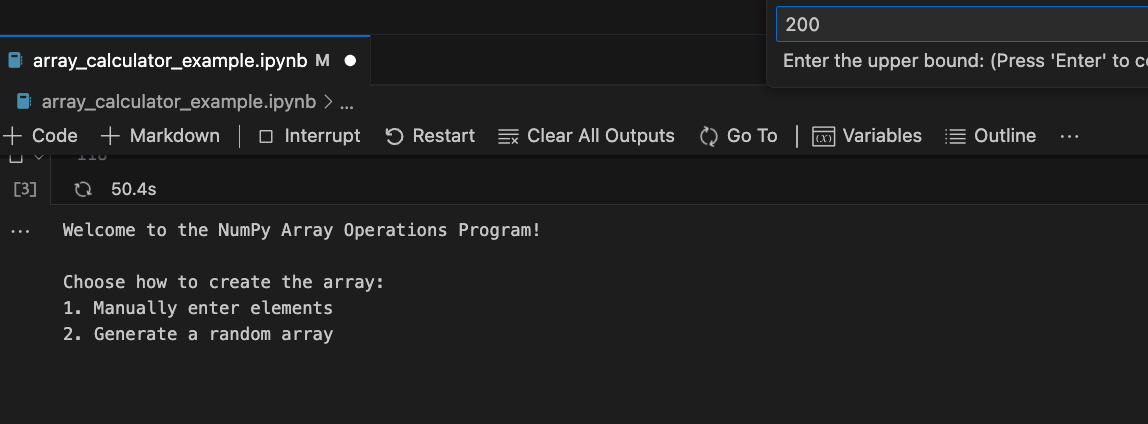
After the user has selected this option, the program asks the user for input to define the array dimension, the row size, the column size, and the data type for the array. Additionally, the user is asked for a range of number to use for random selection. Figures 16-22 show an example of the user entering 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.

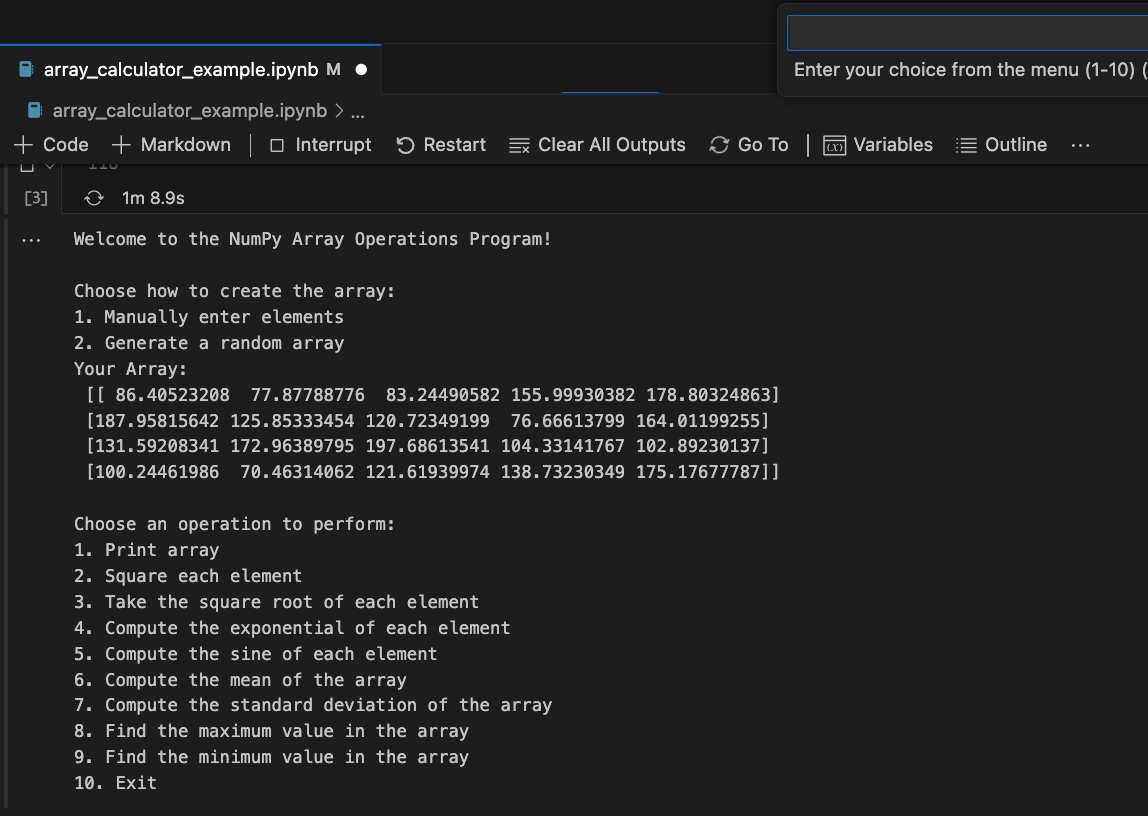
*Figure 16: Array Calculator: Selecting array dimension*

*Figure 17: Array Calculator: Defining column size*

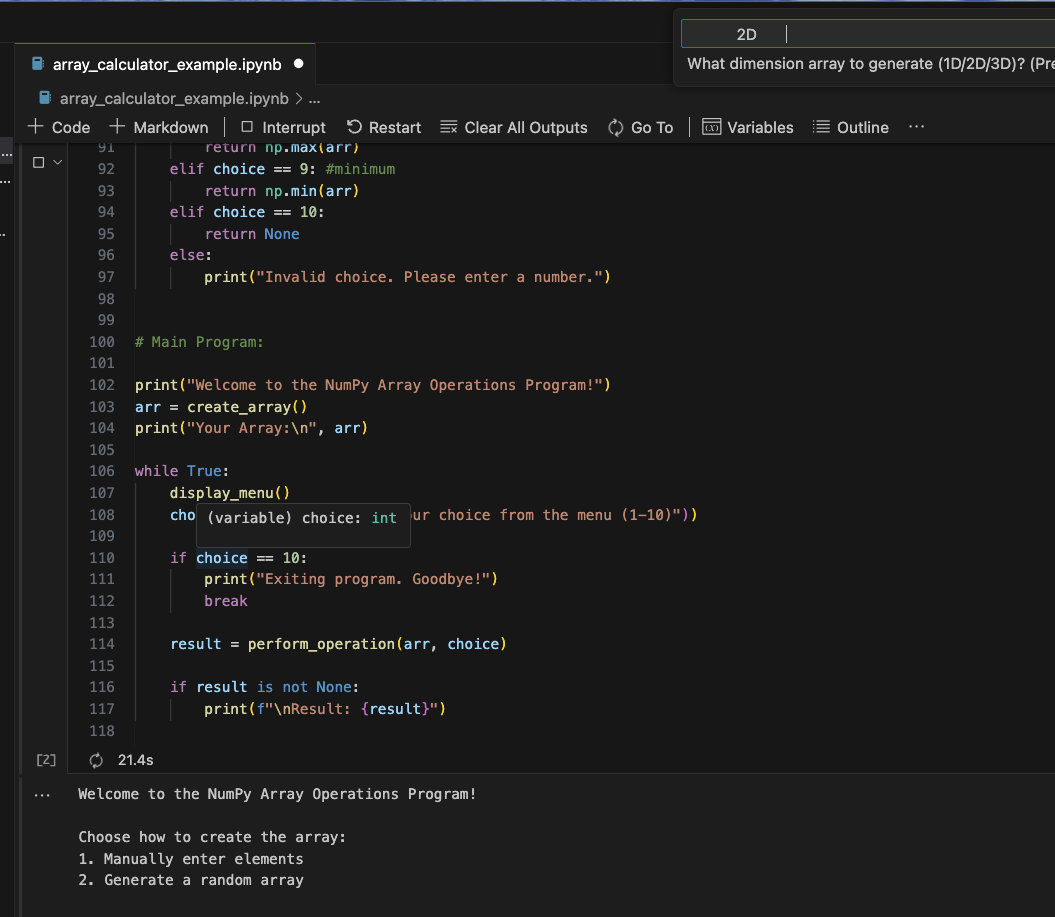
*Figure 18: Array Calculator: Defining data type*

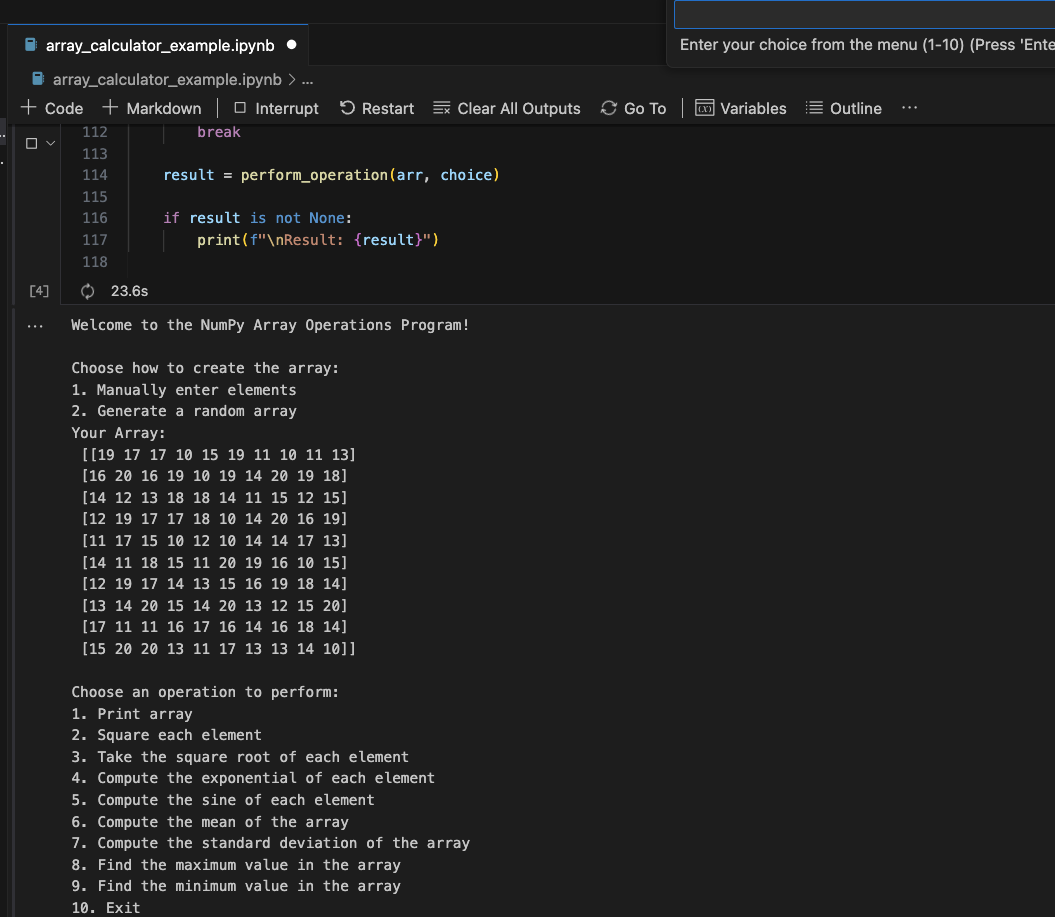
*Figure 19: Array Calculator: Defining lower bound*

*Figure 20: Array Calculator: Defining upper bound*

*Figure 21: Array Calculator: Random array output*

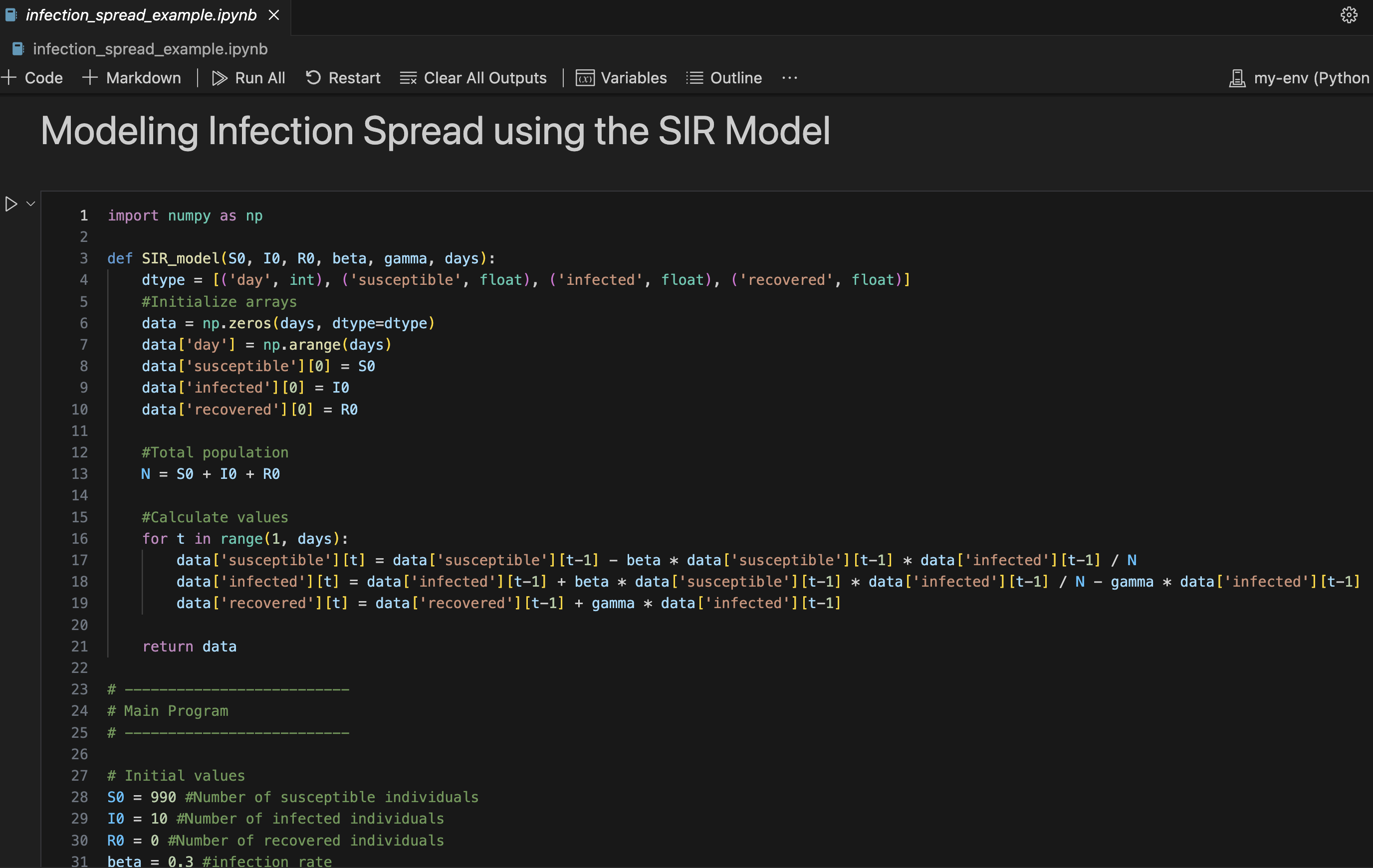
One of the problems encountered while creating this code was handling string input from the user. To ensure proper input was provided, string modulation was used. The *`strip()`* and *`lower()`* methods were incorporated for error handling. This made it easier to ensure that the user did not have to 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.

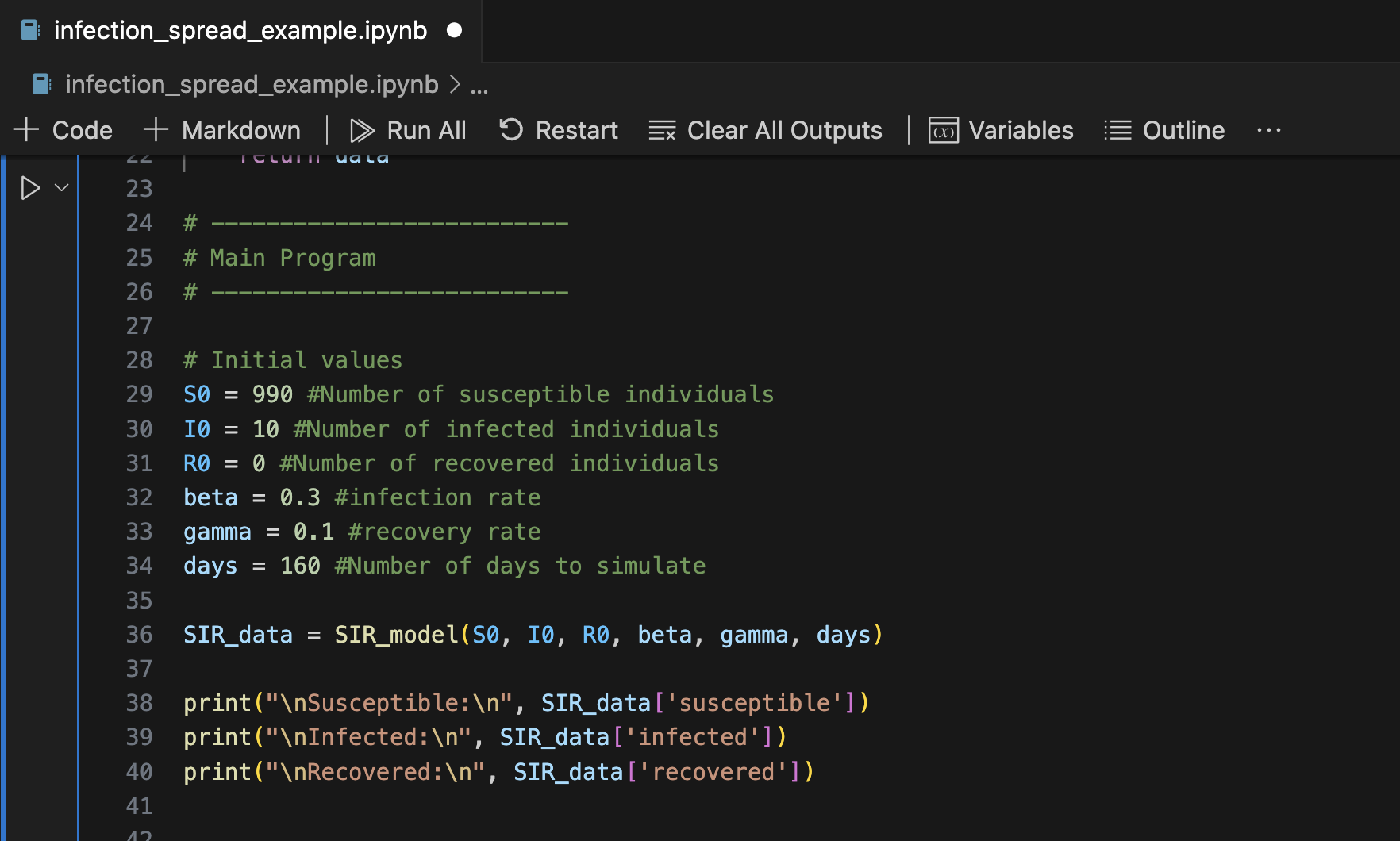
*Figure 22a: Array Calculator: Showing input handling*

*Figure 22b: Array Calculator: Showing input handling*

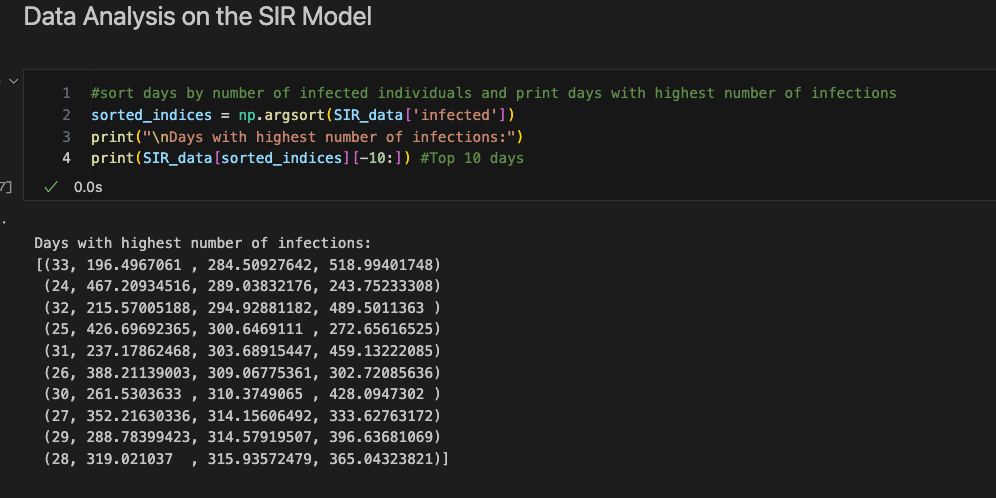
# 3. SIR Model Analysis

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.

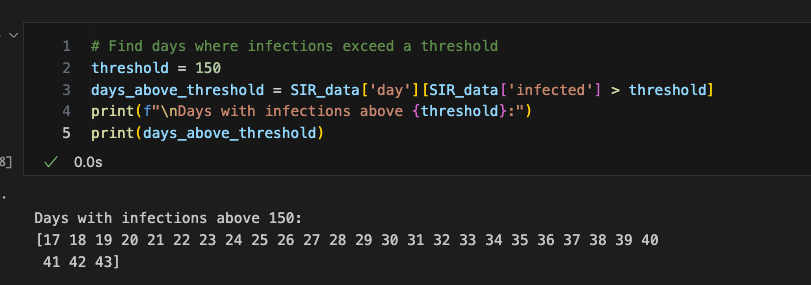
*Figure 23: SIR Model Code*

*Figure 24: SIR Model Code (final)*

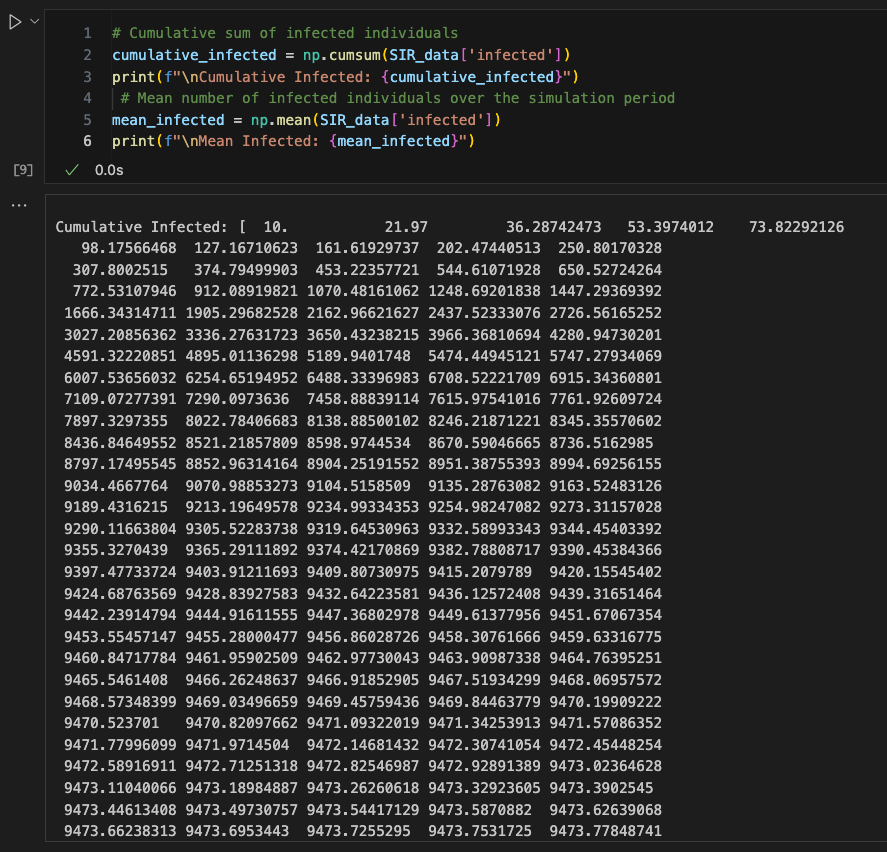
To show the benefit of using Python for modeling disease infection, various data analyses were performed. The first analysis was using *`np.argsort`* to sort the days by the number of infected individuals. With this information stored in a variable, using array indexing, the program prints to the screen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

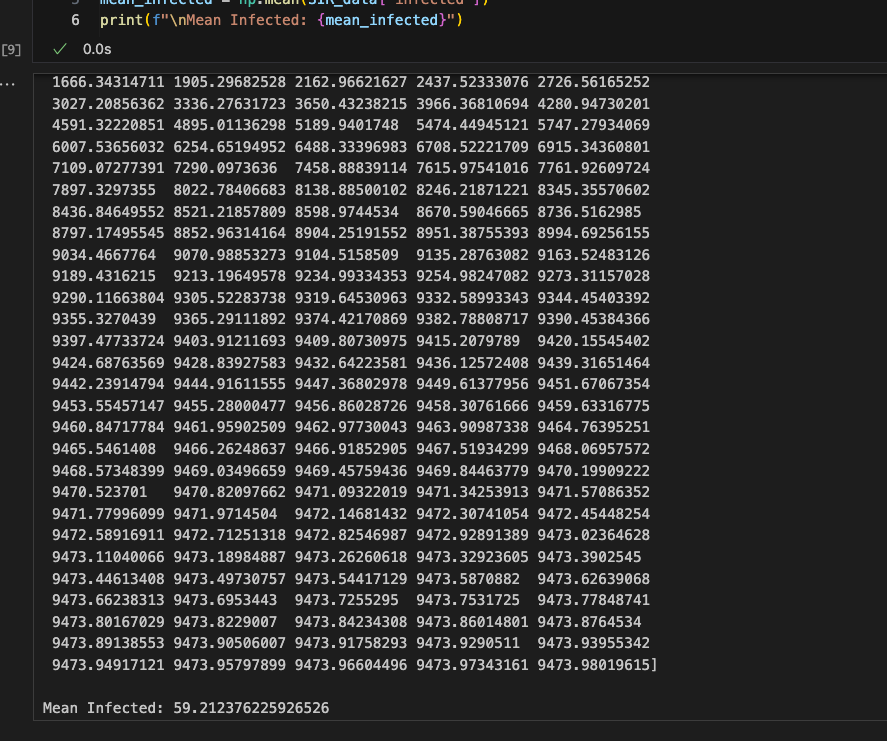
*Figure 25: SIR Model: Sorting number of infected individuals*

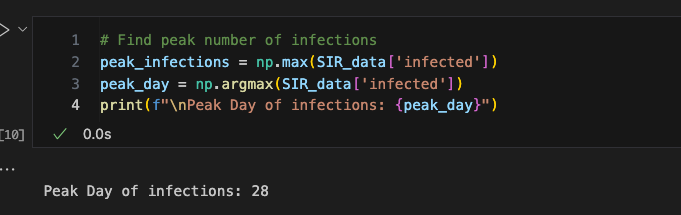
The next analysis uses fancy indexing and boolean indexing to access array elements to find the days where the number of infected individuals exceeds a certain threshold. Since fancy indexing allows access to elements in an arbitrary order, it is best used here. The boolean indexing is used to set the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

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

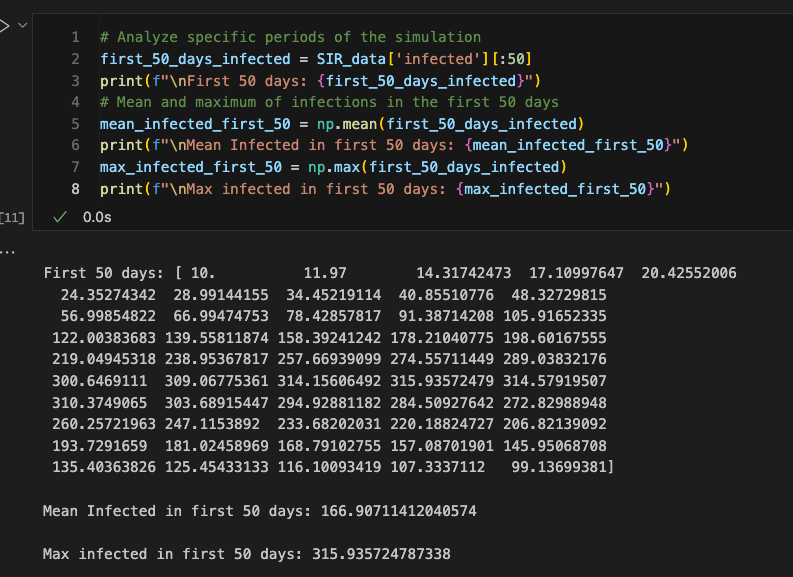
The next analyses provide statistical information. It uses the built in NumPy cumulative summation and mean methods to calculate the cumulative sum of infected individuals and the mean number of infected individuals over the simulation period, and prints the results to the screen. Figure 27 and 28 shows 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.

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

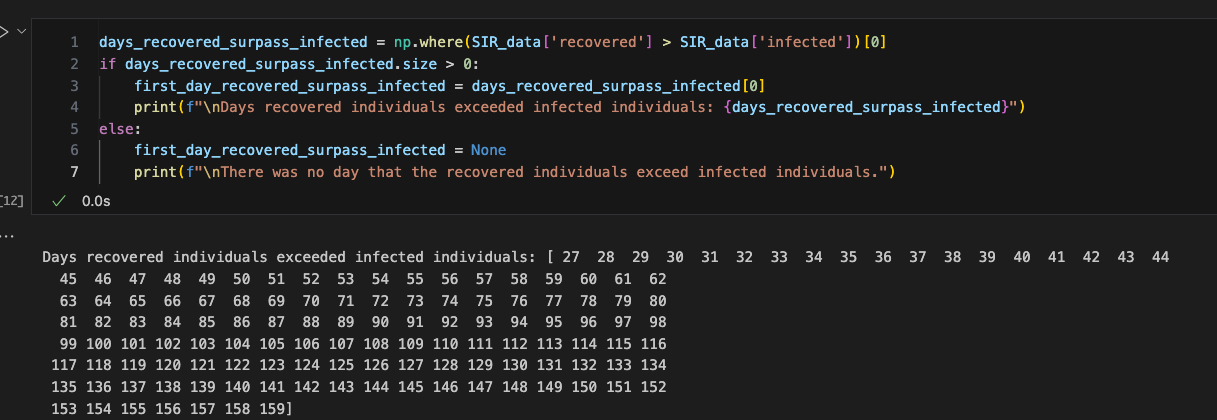
*Figure 28: SIR Model: Using universal functions `cumsum` and `mean` (continued)*

*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.

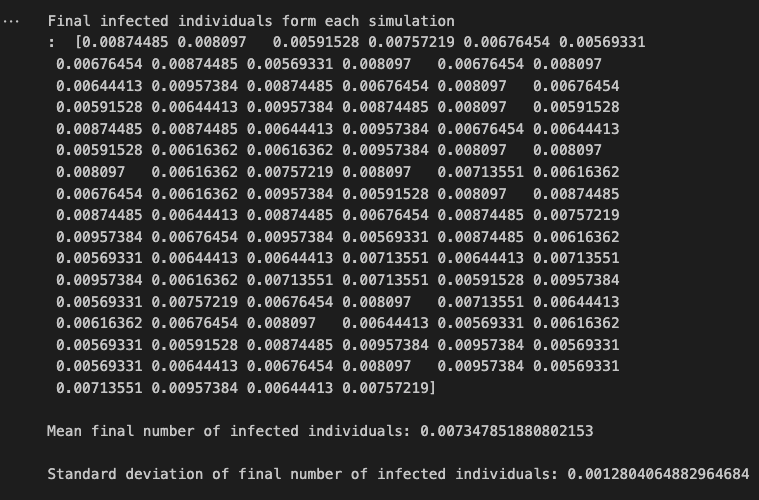
*Figure 30: SIR Model: First 50 days infected array access, mean, and max values*

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.

*Figure 31: SIR Model: Days recovered individuals surpassed infected*

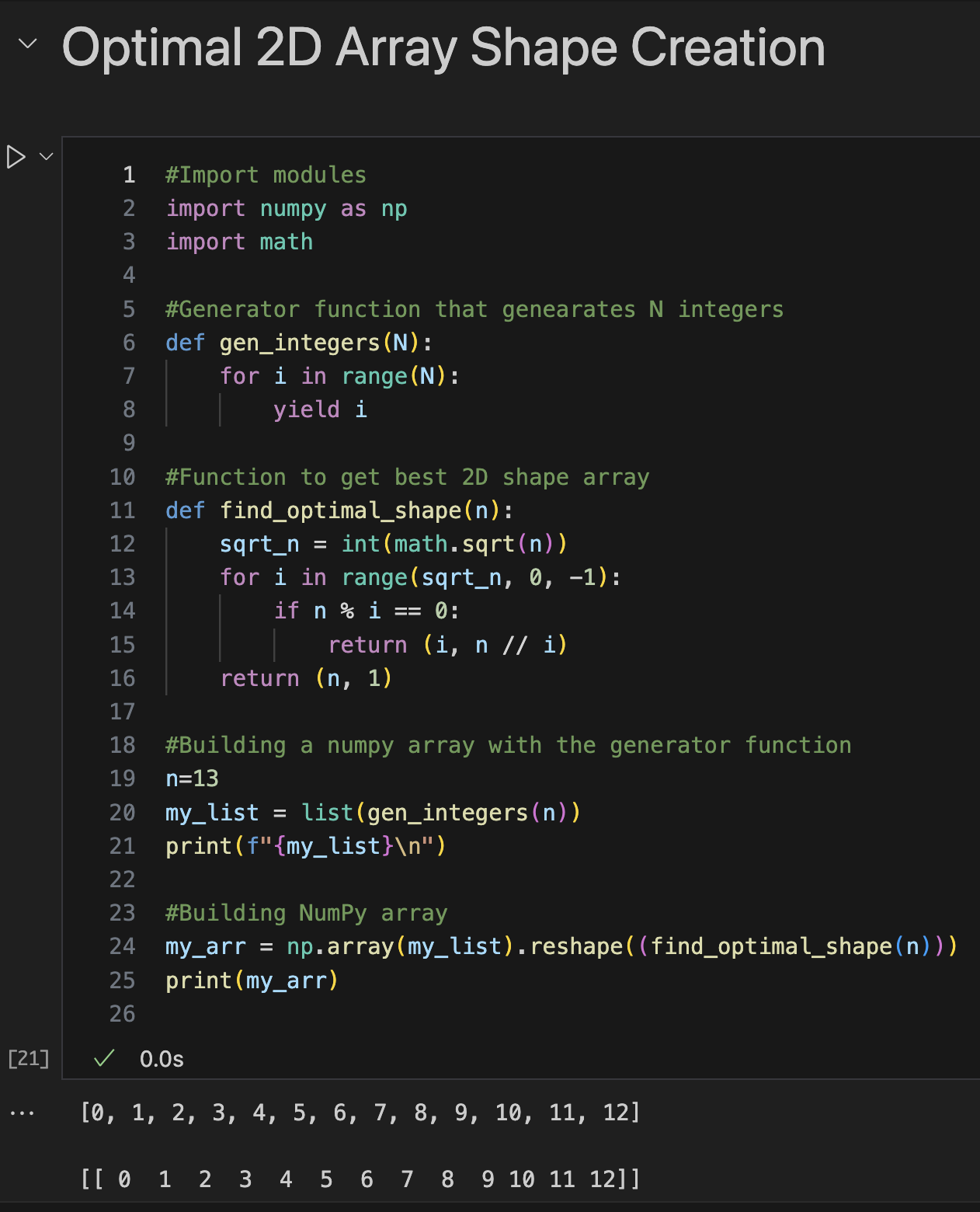
The last analysis is a Monte Carlo simulation. The Monte Carlo simulation is used to estimate the impact of a random initial condition. For this example, only the initial number of infected, *`I0`*, was randomized. A new function, *`monte\_carlo­\_simulation()`*, was defined. It declares a new array *`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.

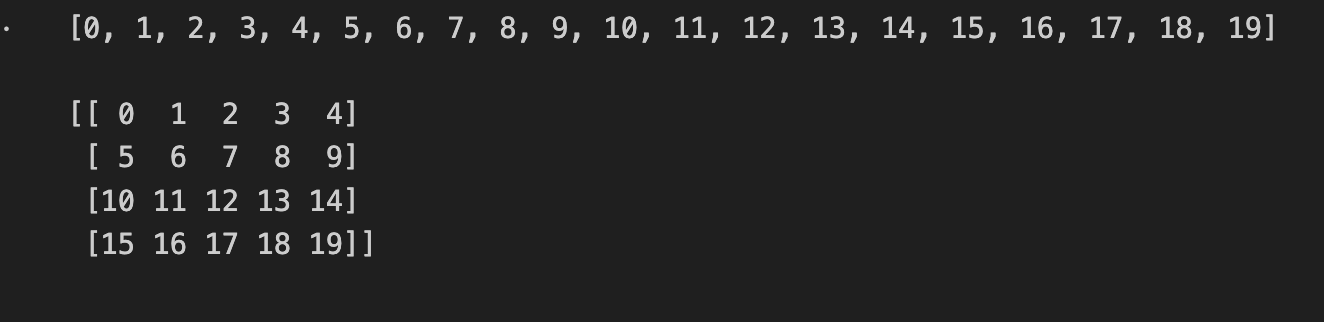
*Figure 32: Monte Carlo Simulation of the SIR Model Code*

*Figure 33: Monte Carlo Simulation Output*

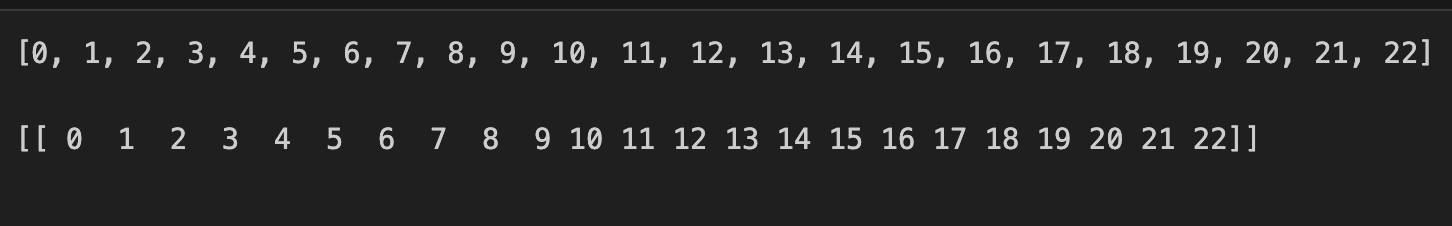
# 5. Optimal Array Shape Creation

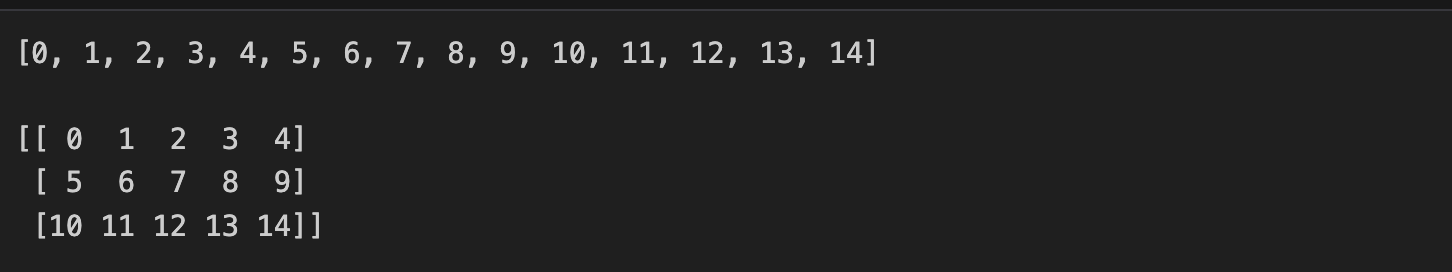
The python script shown below in Figure 34 is an exercise demonstrating how to create a NumPy array. This particular example utilizes a generator function named *`gen\_integers(N)`* that yields integers from 0 to *`N`*. The generated integers are converted into a list and stored in the list variable 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.

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

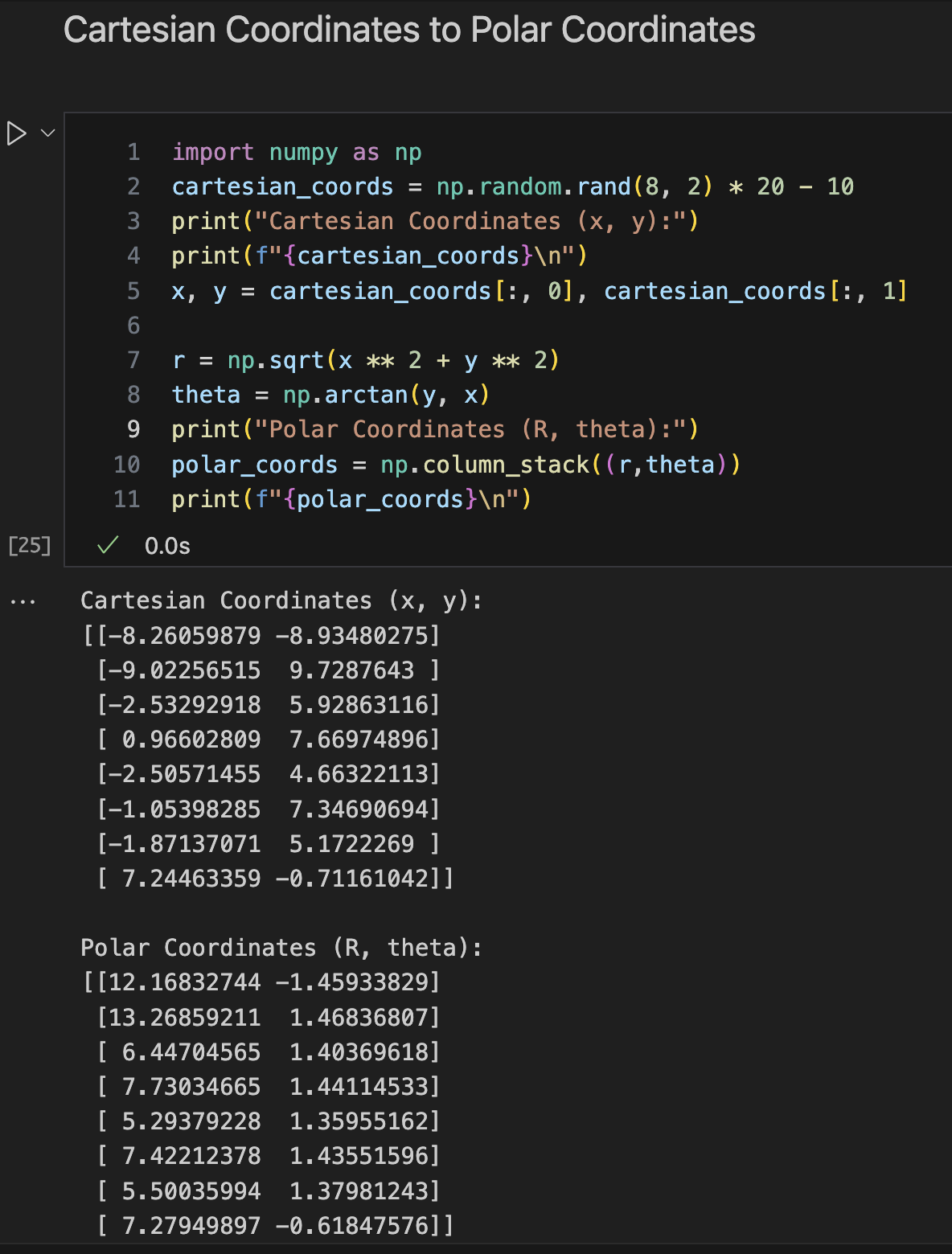
*Figure 35: Optimal 2D Array Shape Creation: Output where n = 20*

# 6. Coordinate Conversion

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

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

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.

*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