Lists / True - Arrays

Data Structures

- Continue to learn about lists and now add true - arrays
- Continue to think and learn by solving problems with code

- We are doing to solve a problem using lists
- First finding the smallest neighborhood in a collection of villages

- #1 - Village Neighborhood
- Write a program that finds the size of the smallest neighborhood of a collection of villages
- We want to store neighborhood sizes
- We might have as many as 100 villages
- Using a single variable would be inefficient
- Lists will allow us to aggregate values instead of using single values
- Learn about Python's powerful list operations

- The Challenge
- There are n villages located at distinct points on a straight road.
- Each village is represented by an integer that indicates its position on the road
- A village’s left neighbor is the village with the next smallest position; a village’s right neighbor is the village with the next biggest position
- The neighborhood consists of half the space between that village and its left neighbor + half the space between that village and its right neighbor.

- Example:
- Village at position 10 - it's left neighbor at position 6 and its right neighbor at position 15
- Then this village's neighborhood starts from position 8 (halfway between 6 and 10) and ends at position 12.5 (halfway between 10 and 15)
- The leftmost and rightmost villages have only one neighbor
- So, the definition of a neighborhood doesn't make sense for them
- We'll ignore neighborhoods of those 2 villages in this problem

- The size of a neighborhood is calculated as the neighborhood's rightmost position minus the neighborhood's leftmost position. For example, the neighborhood that goes from 8 to 12.5 has size 12.5 – 8 = 4.5.

- Determine the size of the smallest neighborhood.

- Input
- The input consists of the following lines:

- A line containing integer n, the number of villages. n is between 3 and 100.
- n lines, each of which gives the position of a village.
- Each position is an integer between –1,000,000,000 and 1,000,000,000.
- The positions need not come in order from left to right; the neighbor of a village could be anywhere in these lines.

- Output
- Output the size of the smallest neighborhood.
- Include exactly one digit after the decimal point.

- Why Lists?
- As part of reading the input, we'll need to read n integers (the integers that represent the positions of the villages).

- We will need to use a range for loop to loop exactly n times.

- In the Village Neighborhood, it's not enough to see each integer just once.
- A village's neighborhood depends on its left and right neighbors.
- Without access to those neighbors, we can't calculate the size of the village's neighborhood.
- We need to store all the village positions for later use.

- For an example of why we need to store all the village positions, consider this test case:

```
6
20
50
4
19
15
1
```

- There are six villages here.
- To find the size of a village's neighborhood, we need that village's left and right neighbors.

- The first village in the input is at position 20.
- What's the size of that village's neighborhood?
- To answer that, we need access to all the village positions so that we can find its left and right neighbors.
- Scanning through the positions, you can identify that the left neighbor is at position 19 and the right neighbor is at position 50.
- The size of this village's neighborhood is therefore (20 – 19)/2 + (50 – 20)/2 = 15.5.

- The second village in the input is at position 50.
- What's the size of that village's neighborhood?
- Again, we need to look through the positions to figure it out.
- This village happens to be the rightmost one, so we ignore this village's neighborhood.

- The third village in the input is at position 4.
- The left neighbor is at position 1, and the right neighbor is at position 15, so the size of this village's neighborhood is (4 – 1)/2 + (15 – 4)/2 = 7.

- The fourth village in the input is at position 19.
- The left neighbor is at position 15, and the right neighbor is at position 20, so the size of this village's neighborhood is (19 – 15)/2 + (20 – 19)/2 = 2.5.

- The only remaining village that we need to consider is at position 15.
- If you calculate its neighborhood size, you should get an answer of 7.5.

- Comparing all the neighborhood sizes that we calculated, we see that the minimum—and the correct answer for this test case—is 2.5.

- We need a way to store all the village positions so that we can find the neighbors of each village.
- A string won't help, because strings store characters, not integers. Python lists to the rescue!

- Lists
- A list is a Python type that stores a sequence of values. (You’ll sometimes see list values referred to as elements.)
- We use opening and closing square brackets to delimit the list.

- We can store only characters in strings, but we can store any type of value in lists.
- This list of integers holds the village positions from the prior section.

- >>> [20, 50, 4, 19, 15, 1]
- [20, 50, 4, 19, 15, 1]

- Here’s a list of strings:
- >>> ['one', 'two', 'hello']
- ['one', 'two', 'hello']

- We can even create a list whose values are of different types:

- >>> ['hello', 50, 365.25]
- ['hello', 50, 365.25]

- Lists support the + operator for concatenation and the * operator for replication:

- >>> [1, 2, 3] + [4, 5, 6]
- [1, 2, 3, 4, 5, 6]
- >>> [1, 2, 3] * 4
- [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]

- We even have the in operator, which tells us whether a value is in a list or not:

- >>> 'one' in ['one', 'two', 'hello']
- True
- >>> 'n' in ['one', 'two', 'three']
- False

- And we have the len function to give us the length of a list:

- >>> len(['one', 'two', 'hello'])
- 3

- A list is a sequence, and we can use a for loop to loop through its values:

- >>> for value in [20, 50, 4, 19, 15, 1]:
- ...   print(value)
- ...
- 20
- 50
- 4
- 19
- 15
- 1

- We can make variables refer to lists, just as we make them refer to strings, integers, and floats.
- Let's make two variables refer to lists and then concatenate them to produce a new list.

- >>> lst1 = [1, 2, 3]
- >>> lst2 = [4, 5, 6]
- >>> lst1 + lst2
- [1, 2, 3, 4, 5, 6]

- While we displayed the concatenated list, we did not store it, as we can see by looking at the lists again:

- >>> lst1
- [1, 2, 3]
- >>> lst2
- [4, 5, 6]

- To make a variable refer to the concatenated list, we use assignment:

- >>> lst3 = lst1 + lst2
- >>> lst3
- [1, 2, 3, 4, 5, 6]

- Names like lst, lst1, and lst2 can be used when there’s no need to be more specific about what a list contains.

- But don’t use list itself as a variable name.
- It’s already a name that we can use to convert a sequence to a list:

- >>> list('abcde')
- ['a', 'b', 'c', 'd', 'e']

- If you make a variable named list, you’ll lose this valuable behavior, and you’ll confuse readers who will expect list not to be tampered with.

- Finally, lists support indexing and slicing.
- Indexing returns a single value, and slicing returns a list of values:

- >>> lst = [50, 30, 81, 40]
- >>> lst[1]
- 30
- >>> lst[-2]
- 81
- >>> lst[1:3]
- [30, 81]

- If we have a list of strings, we can access one of its string’s characters by indexing twice, first to select a string and then to select a character:

- >>> lst = ['one', 'two', 'hello']
- >>> lst[2]
- 'hello'
- >>> lst[2][1]
- 'e'

- What does the following code store in the total variable?

```
lst = [a list of numbers]
total = 0
i = 1

while i <= len(lst):
    total = total + i
    i = i + 1
```

- List Mutability
- Strings are immutable, which means they cannot be modified.
- When it looks like we’re changing a string (for example, using string concatenation), we’re really creating a new string, not modifying one that already exists.

- Lists, on the other hand, are mutable, which means they can be modified.

- We can observe this difference by using indexing. If we try to change a character of a string, we get an error:

- >>> s = 'hello'
- >>> s[0] = 'j'
- Traceback (most recent call last):
-   File "<stdin>", line 1, in <module>
- TypeError: 'str' object does not support item assignment

- The error message says that strings don’t support item assignment, which just means that we can’t change their characters.

- But because lists are mutable, we can change their values:

- >>> lst = ['h', 'e', 'l', 'l', 'o']
- >>> lst
- ['h', 'e', 'l', 'l', 'o']
- >>> lst[0] = 'j'
- >>> lst
- ['j', 'e', 'l', 'l', 'o']
- >>> lst[2] = 'x'
- >>> lst
- ['j', 'e', 'x', 'l', 'o']

- What is the output of the following code?

- lst = ['abc', 'def', 'ghi']
- lst[1] = 'wxyz'

- print(len(lst))

- 3, 9, 10, 4 or Error

- Useful strings and lists methods
- dir (‘’) - get a list of methods for a particular type

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__',
<more stuff with underscores>
'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

- Back to our Village Neighborhood
- We might use a couple of operations on a list that might help
- Adding to the list
- Sorting the list
- For example, here are our sample villages in the order that we read them:
- 20 50 4 19 15 1
- That’s a mess! On a real street, they’d come in order of position, like this:
- 1 4 15 19 20 50

- Want the neighbors of the village at position 4?
- Just look immediately to the left and immediately to the right: 1 and 15.
- The neighbors of the village at 15?
- Boom, they’re right there—4 and 19.
- No more searching all over the place.
- We’ll sort the list of village positions to simplify our code.

- We can add to a list using the append method and sort a list using the sort method.
- We’ll learn these two methods, and a few others that you’ll likely find useful as you continue working with lists, and then we’ll come back to solve Village Neighborhood.

- he append method appends to a list, which means that it adds a value to the end of the values already there.
- Here's append adding three village positions to an initially empty list:

- >>> positions = []
- >>> positions.append(20)
- >>> positions
- [20]
- >>> positions.append(50)
- >>> positions
- [20, 50]
- >>> positions.append(4)
- >>> positions
- [20, 50, 4]

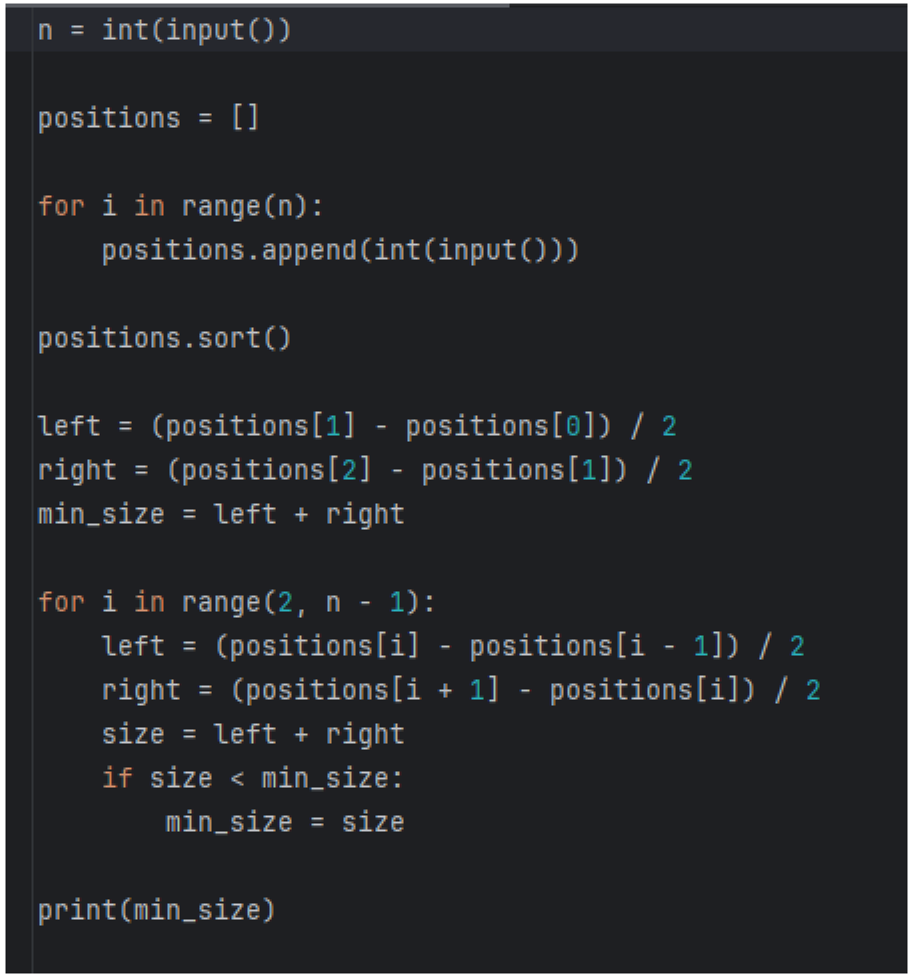
- Notice that we're using append without using an assignment statement.
- The append method doesn't return a list; it modifies an existing list.

- The extend method is related to append.
- You use extend whenever you'd like to concatenate a list (not a single value) to the end of an existing list.
- Here's an example:

- >>> lst1 = [1, 2, 3]
- >>> lst2 = [4, 5, 6]
- >>> lst1.extend(lst2)
- >>> lst1
- [1, 2, 3, 4, 5, 6]
- >>> lst2
- [4, 5, 6]

- There are a lot more, but you can do some research and discover that is needed to manipulate your lists

- Let’s look at the first iteration solution of our code:



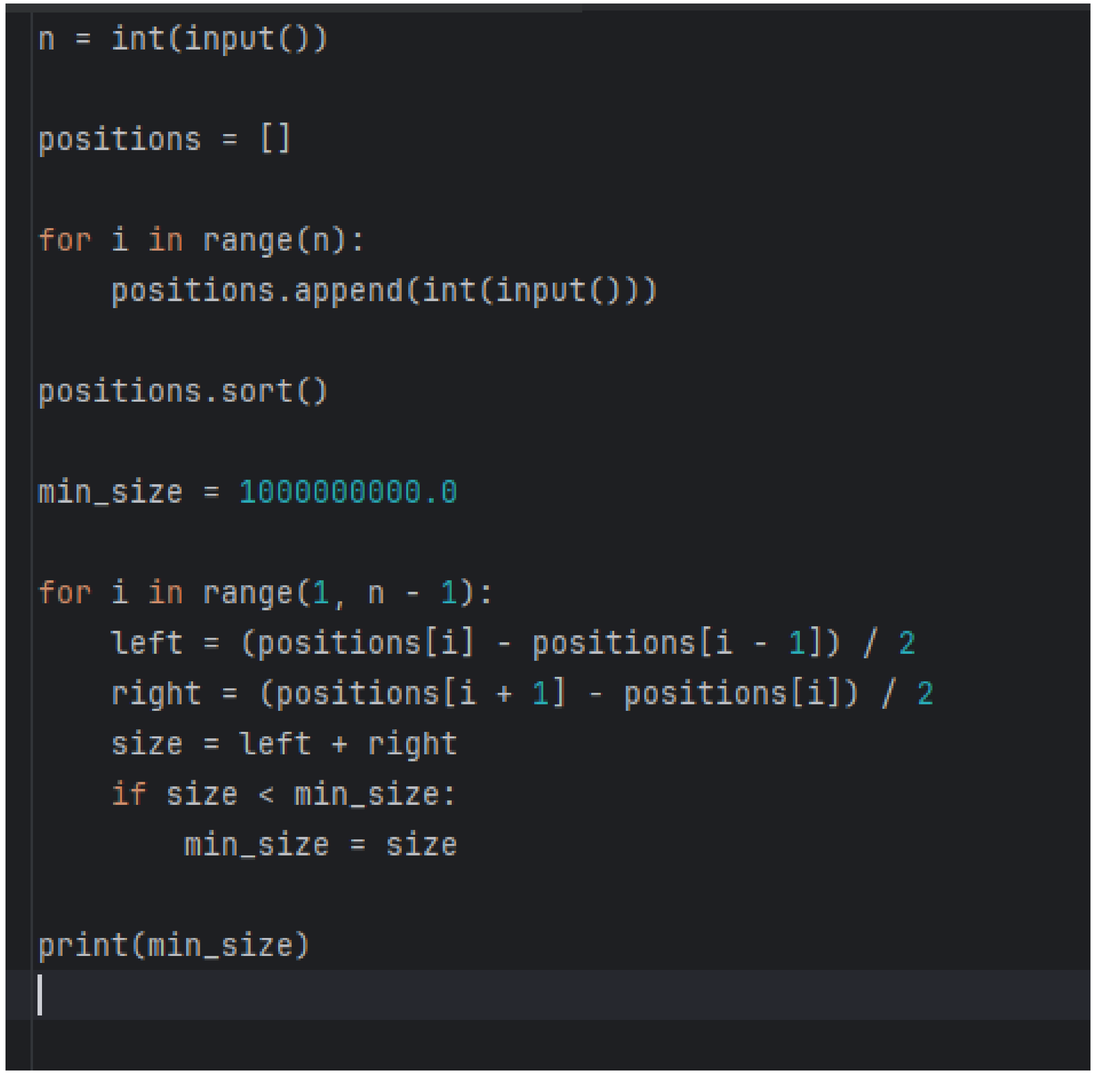
```
n = int(input())

positions = []

for i in range(n):
    positions.append(int(input()))

positions.sort()

left = (positions[1] - positions[0]) / 2
right = (positions[2] - positions[1]) / 2
min_size = left + right

for i in range(2, n - 1):
    left = (positions[i] - positions[i - 1]) / 2
    right = (positions[i + 1] - positions[i]) / 2
    size = left + right
    if size < min_size:
        min_size = size

print(min_size)
```

- Avoiding code duplication
- Means we should be able to improve our code
- Want to avoid repeating code because it adds to the amount of code that we maintain, and it makes it harder to fix problems
- The only reason we're calculating the size of a village's neighborhood before the loop is so that the loop has something to compare the other neighborhood sizes against.
- If we entered the loop without a value for min_size, we'd get an error when the code tries to compare it to the size of the current village.

- If we set min_size to 0.0 before the loop, then the loop will never find a smaller size, and we'll incorrectly output 0.0 no matter the test case.
- Using 0.0 would be a bug!

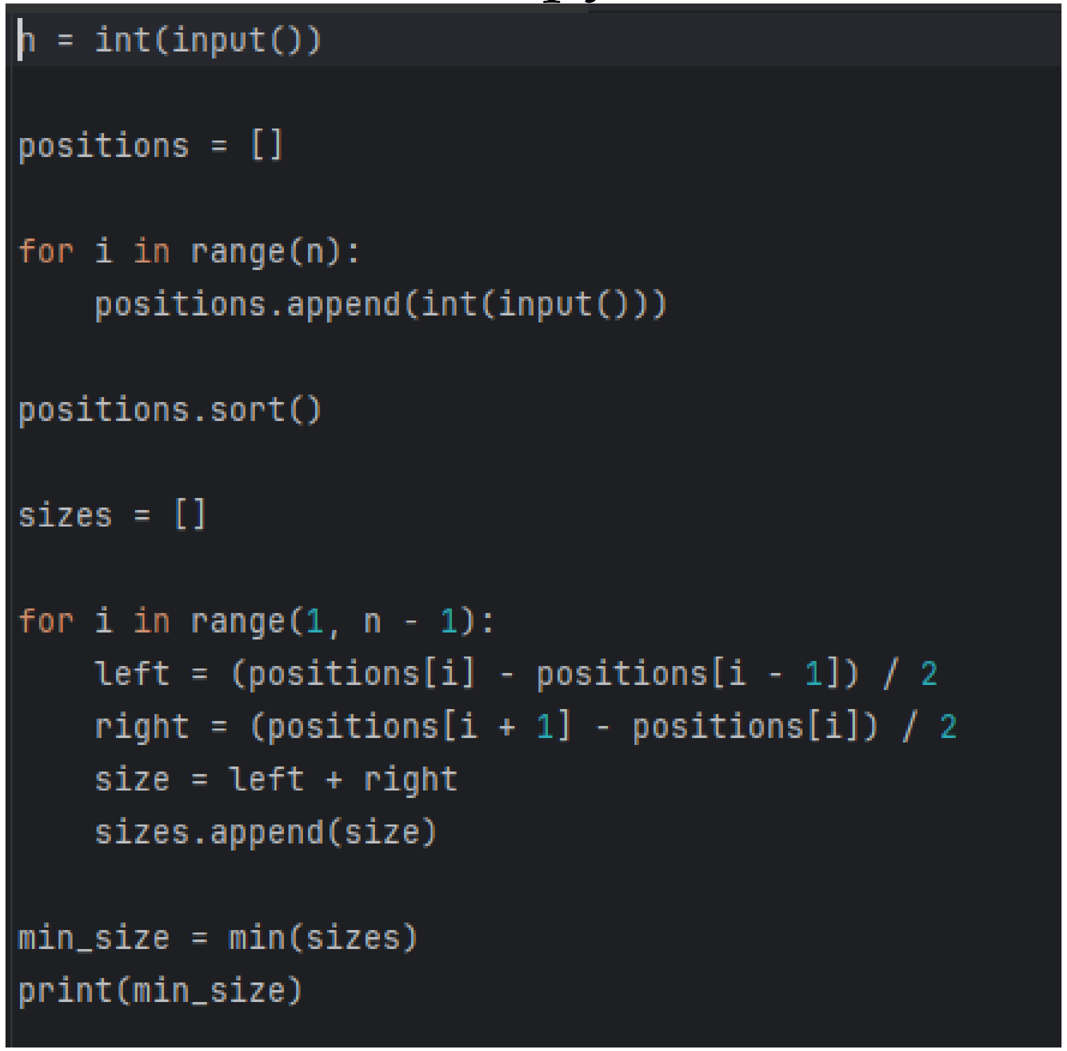
- But a huge value, one at least as big as every possible neighborhood size, will work.
- We just need to make it so huge that the first iteration of the loop is guaranteed to find a size that's no bigger, ensuring that our fake huge size never gets output.

- Second iteration:



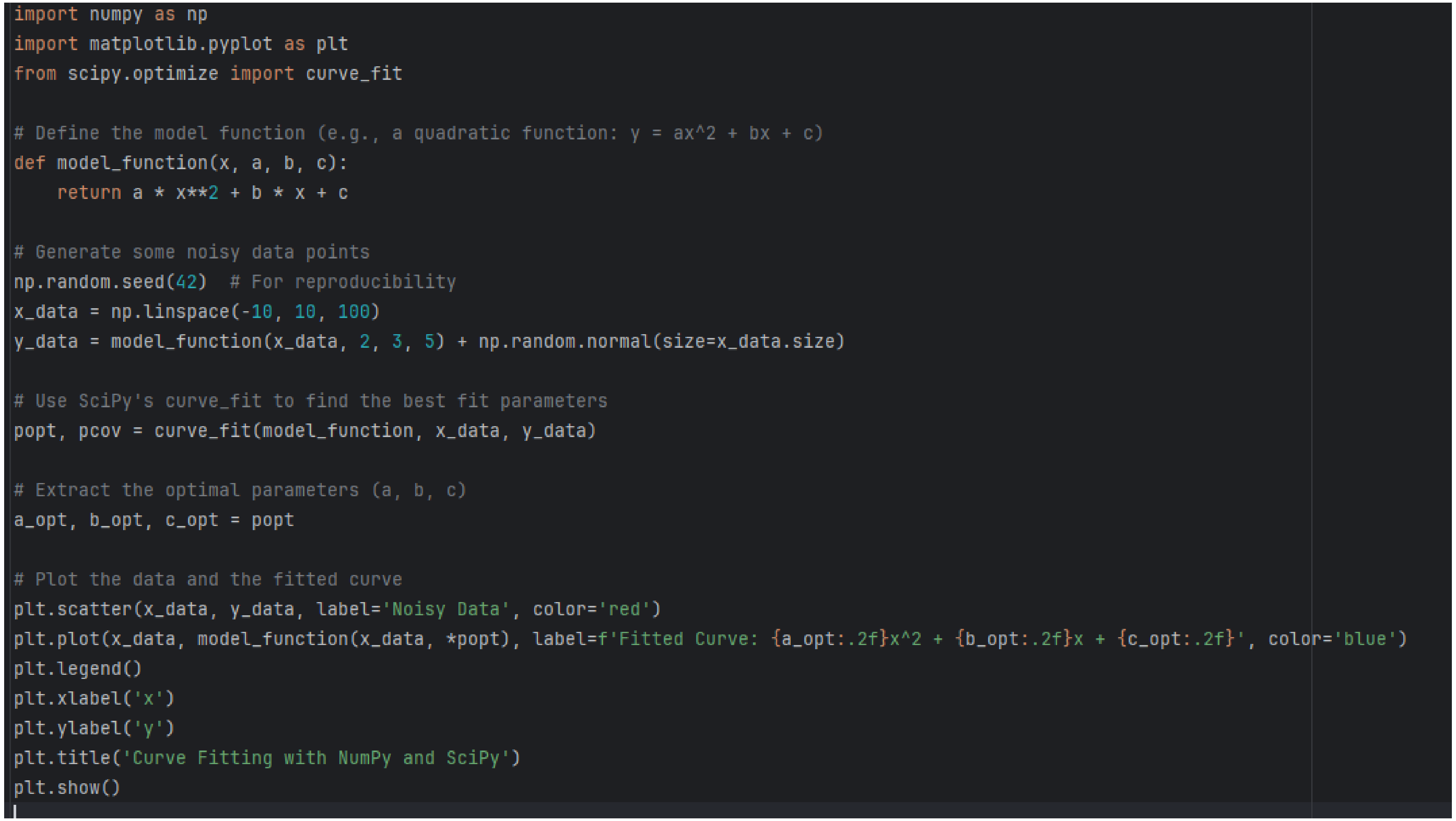
```
n = int(input())

positions = []

for i in range(n):
    positions.append(int(input()))

positions.sort()

min_size = 1000000000.0

for i in range(1, n - 1):
    left = (positions[i] - positions[i - 1]) / 2
    right = (positions[i + 1] - positions[i]) / 2
    size = left + right
    if size < min_size:
        min_size = size

print(min_size)
```

- Basically, another way to avoid code duplication is to store each neighborhood size in a list of sizes - python has a built-in min function
- So, iteration #3:



```
n = int(input())

positions = []

for i in range(n):
    positions.append(int(input()))

positions.sort()

sizes = []

for i in range(1, n - 1):
    left = (positions[i] - positions[i - 1]) / 2
    right = (positions[i + 1] - positions[i]) / 2
    size = left + right
    sizes.append(size)

min_size = min(sizes)
print(min_size)
```

- Real arrays can be added but need to import numpy and or scipy libraries:



```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Define the model function (e.g., a quadratic function: y = ax^2 + bx + c)
def model_function(x, a, b, c):
    return a * x**2 + b * x + c

# Generate some noisy data points
np.random.seed(42)  # For reproducibility
x_data = np.linspace(-10, 10, 100)
y_data = model_function(x_data, 2, 3, 5) + np.random.normal(size=x_data.size)

# Use SciPy's curve_fit to find the best fit parameters
popt, pcov = curve_fit(model_function, x_data, y_data)

# Extract the optimal parameters (a, b, c)
a_opt, b_opt, c_opt = popt

# Plot the data and the fitted curve
plt.scatter(x_data, y_data, label='Noisy Data', color='red')
plt.plot(x_data, model_function(x_data, *popt), label=f'Fitted Curve: {a_opt:.2f}x^2 + {b_opt:.2f}x + {c_opt:.2f}', color='blue')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
plt.title('Curve Fitting with NumPy and SciPy')
plt.show()
```

- Grab the code in canvas
- Let’s see how to install modules