**My Approach:**

Parameters for the rolling window are passed in using the command line. The parameters are a string of numbers that are unordered and contain duplicates.

Before anything is done with the parameters an object is created for the rolling window. The size method for the window creates a list of None the size of the window. For size of three the method would create [None, None, None]

The parameters are checked to make sure that they can be converted from a string into an int (I am assuming that the number being passed in are only ints and not floats). Then using a variable called position (that starts at 0) delete an element from the list and adds the new elements to the same spot. The position is incremented by one and if the size of the position is larger than the window size position is reset to 0.

There are three methods that are used. One converts the list to a tuple. Tuples are immutable so all the operations are done to the list, but the specifications asked to return a tuple. The two other methods are max that returns the max and getAverage which returns the average.

For getAverage I assumed that the division was based on total spaces in the rolling window and not the numbers in the rolling window. For example if 15 is the only element in the rolling window of three the average is 5.

For max I iterated through the list and found the max value this is a cheaper operation than a sort because this has a time complexity of (O)N versus a sort such as quicksort or mergesort which has a time complexity of (O)NlogN. For getAverage I also iterated through the list and divided that by the size of the rolling window.

**Assumptions:**

When calculating the average of a rolling window the amount divided by is the size of spaces of the rolling window not the amount of numbers in the rolling window.

The numbers coming in from the input stream are ints and not floats.

**Time complexity** (0) N+2M

N: Iterating through all the elements in the input stream

M: Iterating through all the elements in the rolling window

**Space complexity** (O)N + M

N: List for rolling window

M: Tuple for rolling window

**Improvements given more time:**

I would have created a descending ordered singly linked list. Each Node would have two values, the values of the node and the orders that the node was inserted into the linked list. This way the insertion of a node would delete the node with the lowest order number and insert the new node based on its value.

This way the head of the linked list would always be the max giving it a time complexity of (0)1 versus (0)N. Also, a pointer could be added the end of the list to find the min in (O)1 time. A sum would be used to keep track of the value of all elements. Each new node would be added to the sum and the node being deleted would be subtracted from the sum. Getting the average would be dividing the sum by the size of the rolling window. This operation would also be time of (O)1 versus a time of (O)N.

**Changes between size 3 and 20:**

Because they were both small all I changed was the input to the method so calculating for the average size in the rolling window changed. I created an object for both using the same class. I used an OO approach to both so the code could be used for rolling windows of different sizes.

**Changes if the rolling window was larger:**

Operations of time (O)N are fast but can become expensive if the size of the rolling window becomes much larger. This is why I would have used a descending ordered singly linked list. Although the inserting new nodes would still take a time of (O)N max, min, and average would be (O)1.

**How to extend code handle median:**

Create a temp variable based on the list and have the list sorted. Then divide that list by 2. That would get the middle of the list.