Initial Input:

```
[11/19/24]seed@VM:~/.../a3$ python3 IDS2.py Events.txt Stats.txt 10
---------------------------------------------------------------------
Checking for inconsistencies between Events.txt and Stats.txt...
---------------------------------------------------------------------
```

When Events.txt and Stats.txt is input as argument, both files will be processed by read_event_file and read_stats_file respectively. It reads line by line and appends it to an empty list []

```python
def main():
    if len(sys.argv) != 4:
        print("Usage: python3 IDS.py <event_file_name> <stats_file_name> <days>")
        sys.exit(1)

    eventFile = sys.argv[1]  # Event file name from command line arguments
    statsFile = sys.argv[2]  # Stats file name from command line arguments
    days = int(sys.argv[3])  # Number of days from command line arguments

    # Read event and stats data
    eventData = read_event_file(eventFile)
    statsData = read_stats_file(statsFile)
    newLogsFile, logCount = generateNewLogFileName()
```

```python
# read event file
def read_event_file(filename):
    line = []

    with open(filename, "r") as e_line:
        lines = e_line.readlines()

        for part in lines:
            line.append(part.strip())

    return line

# read stats file
def read_stats_file(filename):
    line = []

    with open(filename, "r") as s_line:
        lines = s_line.readlines()

        for part in lines:
            line.append(part.strip())

    return line
```

The list returned will be used in the function processEvents and processData to further be Split each lines into meaningful components, such as event names, types, and numerical properties. These values are stored as follows:

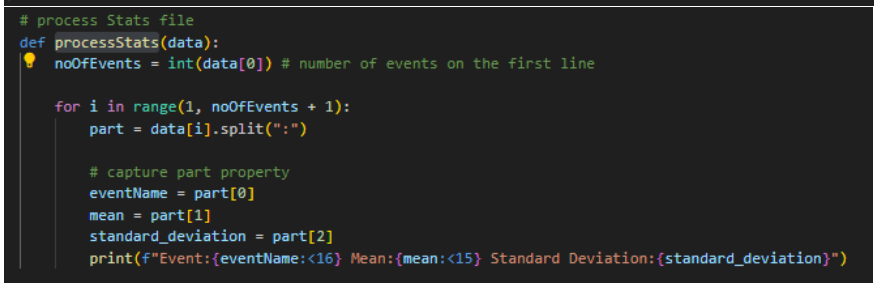
Events Data:

Each line is split by : into individual components (eventName, eventType [CD], minimum, maximum, weight).

Statistics Data:

Similar to events, the lines are split by : to extract (eventName, mean, and standard deviation)

```
# process Event file
def processEvents(data):
    noOfEvents = int(data[0]) # number of events on the first line
    weights = []

    for i in range(1, noOfEvents + 1):
        part = data[i].split(":")

        # capture part property
        eventName = part[0]
        eventType = part[1]
        minimum = part[2]
        maximum = part[3]
        weight = part[4]

        # validations
        # neither Continuous or Discrete
        if eventType != "C" and eventType != "D":
            print("Event type must be either C or D")
            return

        # minimum value
        if minimum == "": # value empty
            print("Minimum values cannot be empty")
            return

        # weight value
        if weight.find(".") > 0: # float value found in weight variable
            print("Weight values must be an integer")
            return

        if weight == "": # value empty
            print("Weight values cannot be empty")

        # maximum value
        if maximum == "":  # Value empty
            maximum = str(random.randint(1, 999))

        # in the case of Discrete events
        if eventType == "D":
            # float values found in minimum or maximum variable
            if minimum.find(".") > 0 or maximum.find(".") > 0:
                print("Float found in a Discrete Event")
                return

        # in the case of Continuous events
        if eventType == "C":
            minimum = "{:.2f}".format(float(minimum)) # 2 decimal places
            maximum = "{:.2f}".format(float(maximum)) # 2 decimal places

        # sum of weight
        weights.append(int(weight))

        print(f"Event:{eventName:<15} Type:{eventType:<15} Min:{minimum:<15} Max:{maximum:<15} Weight:{weight}")
    return weights
```



```
# process Stats file
def processStats(data):
    noOfEvents = int(data[0]) # number of events on the first line

    for i in range(1, noOfEvents + 1):
        part = data[i].split(":")

        # capture part property
        eventName = part[0]
        mean = part[1]
        standard_deviation = part[2]
        print(f"Event:{eventName:<16} Mean:{mean:<15} Standard Deviation:{standard_deviation}")
```

Some validation are made in place during the processing of both files:

Events:

```
1) Event type must be either C or D
------------------------
<<Events.txt>>
Event type must be either C or D


<<Stats.txt>>
Event:Logins           Mean:4              Standard Deviation:1.5
Event:Time online      Mean:150.5          Standard Deviation:25.00
Event:Emails sent      Mean:10             Standard Deviation:3
Event:Emails opened    Mean:12             Standard Deviation:4.5
Event:Emails deleted   Mean:7              Standard Deviation:2.25
------------------------
```

```
2) Minimum values cannot be empty
------------------------
<<Events.txt>>
Minimum values cannot be empty


<<Stats.txt>>
Event:Logins           Mean:4              Standard Deviation:1.5
Event:Time online      Mean:150.5          Standard Deviation:25.00
Event:Emails sent      Mean:10             Standard Deviation:3
Event:Emails opened    Mean:12             Standard Deviation:4.5
Event:Emails deleted   Mean:7              Standard Deviation:2.25
```

```
3) Weight values must be an integer
------------------------
<<Events.txt>>
Weight values must be an integer


<<Stats.txt>>
Event:Logins           Mean:4              Standard Deviation:1.5
Event:Time online      Mean:150.5          Standard Deviation:25.00
Event:Emails sent      Mean:10             Standard Deviation:3
Event:Emails opened    Mean:12             Standard Deviation:4.5
Event:Emails deleted   Mean:7              Standard Deviation:2.25
```

```
4) Weight values cannot be empty
------------------------------------------------------------------------
No inconsistencies found
------------------------
<<Events.txt>>
Weight values cannot be empty
```

5)
```
if maximum == "":  # Value empty
    maximum = str(random.randint(1, 999))
```

If maximum is empty, it will randomly generate a value from 1 – 999

6) To ensure that there are no decimal value for “D”

```
if eventType == "D":
    # float values found in minimum or maximum variable
    if minimum.find(".") > 0 or maximum.find(".") > 0:
        print("Float found in a Discrete Event")
        return
```

Once done it will be run through the check_file_inconsistency function to compare both Event.txt data and Stats.txt data to ensure that there are no inconsistency

1)
```
if noOfEventData != noOfStatsData:
    print("Number of data is inconsistent")
```

```
------------------------------------------------------------------------
Checking for inconsistencies between Events.txt and Stats.txt...
------------------------------------------------------------------------
Number of data is inconsistent
```

```
# check for inconsistencies between Events and Stats file
def check_file_inconsistency(eventData, statsData):
    noOfEventData = int(eventData[0])
    noOfStatsData = int(statsData[0])

    # different number of data
    if noOfEventData != noOfStatsData:
        print("Number of data is inconsistent")
        return False

    # loop through array of data and check the consistency between part event
    for i in range(1, noOfEventData):
        # different event
        if eventData[i].split(":")[0] != statsData[i].split(":")[0]:
            print(f"Inconsistencies found in line {i + 1}")
            return False

    print("No inconsistencies found")
    return True
```

Activity Engine and the Logs:

The generateDataSet() function generate data for multiple events over a given number of days. It iterates through the events defined in the eventData (which contains event names and their properties) and generates event samples using the generateData() function.

```python
# generate data set for part event
def generateDataSet(days, eventData, statsData):

    # get the number of events
    noOfEvents = int(eventData[0])

    # track the set of data to be used to simulate activity for the baseline
    activityData = []

    for i in range(days):
        for j in range(1, noOfEvents + 1):
            # split event data
            eData = eventData[j].split(":")
            eventName = eData[0] # name of event
            eventType = eData[1] # type of event
            minimum = int(eData[2]) # min of event
            if eData[3] == "":
                maximum = random.randint(1,sys.maxsize)  # Generate random max value
            else:
                maximum = int(eData[3])  # max of event

            # split stats data
            sData = statsData[j].split(":")
            mean = float(sData[1]) # mean
            standardDeviation = float(sData[2]) # standard deviation

            # generate set of data as close to mean and stdev
            dataSet = generateData(mean, standardDeviation, days, minimum, maximum, eventType)
            activityData.append(dataSet) # add


    print(f"<<Data set generation for {days} days completed!>>")
    return activityData
```

The generateData() function uses the **Normal Distribution** (s.NormalDist(mean, standardDeviation) object) to generate a set of event samples. The samples array represents the generated data over days, based on the mean and standard deviation values. The sample data is generated and then filtered to ensure it falls within the specified minimum and maximum values, ensuring compliance with the constraints defined in the Stats.txt file. If any values fall outside the valid range, they are discarded (the loop continue will ignore those values and re-sample if necessary).

For discrete events, the generated samples are rounded to integers using round(samples[index]). While for continuous events, the generated samples are rounded to two decimal places using round(samples[index], 2).

The code uses a **tolerance check** to ensure that the generated sample data is consistent with the target mean and standard deviation. A tolerance of 0.05 is applied to verify if the sample data is within the acceptable range for both mean and standard deviation. If the data meets the required statistical properties, it is returned. Otherwise, the process repeats until a suitable sample is generated.

```python
def generateData(mean, standardDeviation, days, minimum, maximum, eventType):
    while True:
        n = s.NormalDist(mean,standardDeviation)
        samples = n.samples(days)

        # Keep values within min/max bounds
        for index in range(len(samples)):
            if eventType == "D":  # Discrete event
                samples[index] = round(samples[index])  # Round value to integer
            elif eventType == "C":  # Continuous event
                samples[index] = round(samples[index], 2)  # Round value to 2 decimal places

            # Check for out-of-bounds values
            if samples[index] < minimum or samples[index] > maximum:
                continue

        # Use a consistent tolerance for mean and standard deviation checks
        tolerance = 0.05  # 5% tolerance

        # Check if the generated data matches the desired mean and standard deviation
        if (mean * (1 - tolerance) <= s.mean(samples) <= mean * (1 + tolerance) and
            standardDeviation * (1 - tolerance) <= s.stdev(samples) <= standardDeviation * (1 + tolerance)):
            return samples  # Return the generated sample data
```

These data will be used in outputData() function to calculate statistical measures (mean, variance, standard deviation) for each event's data, for each day and writes these statistics to a file in a readable format (in a new file called Baseline_Statistic.txt). This will then be served as the baseline data for further analysis for the subsequent statistic files.

```python
    print("------------------------")
    # generate data set for part event for part day
    dataSet = generateDataSet(days, eventData, statsData)
    displayGeneratedData(eventData, dataSet)  # Call the display function

    print("------------------------")
    # simulate activity and write to logs file
    simulateActivity(newLogsFile, days, eventData, dataSet)


    # get data from logs file
    data, eventName = readLogs(newLogsFile)


    # write result of analysis to text file - BASELINE
    mean, stddev = outputData(data, eventName, "Baseline_Statistics.txt")
```

```python
# write statistics to file
def outputData(data, eventName, filename):
    # get mean
    mean = calculateMean(data)

    # get variance
    variance = calculateVariance(data, mean)

    # get standard deviation
    stddev = calculateStddev(variance)

    # write to file
    with open(filename, "a") as file:
        file.write(str(len(eventName)))
        for i in range(len(eventName)): # number of events
            file.write(f"\n{eventName[i]}:{str(mean[i])}:{str(stddev[i])}")

    return mean, stddev
```

For the logging of file, the generated data are stored in following format

```
for i in range(days):
        file.write(f"Day {i + 1}\n")
        file.write(f"{noOfEvents}\n")

        for j in range(noOfEvents):
            # deconstruct eventData
            data = eventData[j + 1].split(":")
            eventName = data[0]
            eventType = data[1]

            file.write(f"{eventName}:{eventType}:{dataSet[j][i]}:\n")

        file.write("\n")
```

```
Day 1
5
Logins:D:6:
Time online:C:142.01:
Emails sent:D:9:
Emails opened:D:8:
Emails deleted:D:10:

Day 2
5
Logins:D:6:
Time online:C:177.72:
Emails sent:D:9:
Emails opened:D:9:
Emails deleted:D:6:

Day 3
5
Logins:D:2:
Time online:C:139.39:
Emails sent:D:14:
Emails opened:D:8:
Emails deleted:D:6:

Day 4
5
Logins:D:2:
Time online:C:163.84:
Emails sent:D:7:
Emails opened:D:15:
```

It is stored in a plain text format to ensure readability, simplicity, and compatibility with various tools and workflows. This format aligns well with the need for human-readable data that will be reused in later parts of the program.

Analysis Engine:

The functions calculateMean, calculateVariance and calculateStddev are used to calculate compute daily totals for each event once the random dataset has been generated over the specified input date. It will then be stored in a list be used to access to calculate the baseline data.

```python
# calculate mean
def calculateMean(data):
    mean = []

    # process data obtained from the logs file - MEAN
    for index, value in enumerate(data): # number of events
        sum = 0
        for i, v in enumerate(value): # number of days
            sum += v

            if i + 1 == len(value): # rparted the last day
                # calculate mean - maintain at 2dp if exceeded
                mean.append(round(sum / (i + 1), 2))

    return mean

# calculate variance
def calculateVariance(data, mean):
    variance = []

    # process data obtained from the logs file - VARIANCE
    for index, value in enumerate(data): # number of events
        sum = 0
        for i, v in enumerate(value): # number of days
            sum += (v - mean[index]) ** 2

            if i + 1 == len(value): # rparted the last day
                # calculate variance - maintain at 2dp if exceeded
                variance.append(round(sum / (i + 1), 2))

    return variance

# calculate standard deviation
def calculateStddev(variance):
    stddev = []

    # calculate the standard deviation based on the variance
    for index, value in enumerate(variance):
        stddev.append(round(value ** 0.5, 2)) # square root variance

    return stddev
```

The generated data set and the daily totals can be seen via the displayGenerateData() function showing the number of values generated for each events across the number of days defined. It also shows the Daily totals of each day.

```python
def displayGeneratedData(eventData, dataSet):
    print("\nGenerated Data:")
    noOfEvents = int(eventData[0])

    # Display the generated data for part event
    for i in range(noOfEvents):
        eventName = eventData[i + 1].split(":")[0]
        print(f"{eventName}: {dataSet[i]}")

    print("\nDaily Totals:")
    # Calculate and display daily totals
    for day in range(len(dataSet[0])):
        daily_total = sum(dataSet[event][day] for event in range(noOfEvents))
        print(f"Day {day + 1}: {daily_total:.2f}")

def generateNewLogFileName(baseName="logs", extension=".txt"):
    logCount = 1
    while os.path.exists(f"{baseName}{logCount}{extension}"):
        logCount += 1
    return f"{baseName}{logCount}{extension}", logCount
```

```
------------------------
<<Data set generation for 10 days completed!>>

Generated Data:
Logins: [4, 6, 6, 3, 5, 6, 3, 2, 3, 3]
Time online: [118.26, 168.1, 176.96, 139.08, 191.53, 138.47, 156.93, 132.74, 183.51, 150.88]
Emails sent: [14, 10, 12, 11, 7, 11, 12, 11, 12, 4]
Emails opened: [14, 10, 11, 12, 15, 10, 12, 21, 5, 7]
Emails deleted: [3, 9, 6, 4, 10, 8, 8, 8, 9, 8]

Daily Totals:
Day 1: 153.26
Day 2: 203.10
Day 3: 211.96
Day 4: 169.08
Day 5: 228.53
Day 6: 173.47
Day 7: 191.93
Day 8: 174.74
Day 9: 212.51
Day 10: 172.88
------------------------
```

Alert Engine:

The alert engine uses the function anomalyCounter by taking day computed daily totals – mean (from Baseline.txt) / stddev(from Baseline.txt) * weight (from the Events.txt). This will compute the anomaly counter for each day.

```
# for part event:
#   [total - mean(from Baseline.txt)] / stddev(from Baseline.txt) * weight(from Events.txt)
def anomalyCounter(filename, weight, mean, stddev):
    print("Currently calculating daily totals")

    # read new logs file and capture its data
    data = readNewLogs(filename) # data = [[], [], ..., []]

    # track the daily totals
    dailyCounter = []

    for index, value in enumerate(data): # loop through the number of days
        counter = 0 # track counter for daily events
        for i, v in enumerate(value): # loop through the number of events
            counter += float(round(((abs((float(v) - mean[i])) / stddev[i]) * weight[i]), 2))

        dailyCounter.append(counter)

    print("Daily totals calculated!\n")
    return dailyCounter
```

Calculating the total threshold by summing up all the weights of each event from the Events.txt

```
# get threshold
def getThreshold(weights):
    sum = 0
    for i in range(len(weights)):
        sum += weights[i]

    return 2 * sum
```

Once the above are done, the flagging() function will loop through the loop of dailycounter list and compare it against the threshold. If the dailycounter > threshold, it will be flagged as an <<ALERT>> and be appended to the list

```python
def flagging(data, threshold):
    print("Currently checking for anomalies\n")

    flagged = []

    for i in range(len(data)):
        alert = False
        if data[i] > threshold:
            flagged.append(i + 1) # day number
            alert = True

        print(f"Day {i + 1} anomaly count = {round(data[i], 2)} {'<<ALERT>>' if alert == True else ''}")

    print("\n")

    # alert
    if len(flagged) != 0:
        print("ALERT! Anomalies detected!")
        print("--------------------------")
        for index, value in enumerate(flagged):
            print(f"Day {value} has been flagged!")
        print("\n")
    else:
        print("No anomalies detected\n")


    return flagged
```

```
Threshold: 18
Currently checking for anomalies

Day 1 anomaly count = 13.41
Day 2 anomaly count = 11.77
Day 3 anomaly count = 10.12
Day 4 anomaly count = 14.7
Day 5 anomaly count = 21.48 <<ALERT>>
Day 6 anomaly count = 10.6
Day 7 anomaly count = 12.74
Day 8 anomaly count = 19.47 <<ALERT>>
Day 9 anomaly count = 8.83
Day 10 anomaly count = 26.56 <<ALERT>>


ALERT! Anomalies detected!
--------------------------
Day 5 has been flagged!
Day 8 has been flagged!
Day 10 has been flagged!
```

A while loop is used to prompt user to load another statistics file and specify a new number of days for analysis. If not, user can input “q” to quit.

```
Enter the new Stats.txt file (or 'q' to quit): temp.txt
Enter the number of days for the new stats file: 10
```