Lab 3

In this lab we will become familiar with distributions, histograms, and functional programming. Do not use numpy or any other library for this lab.

Before that, lets get setup homework submission and submit your previous lab.

Working on the Command-line.

It is important for you to learn to work on the command line and to be familiar with the Unix environment (e.g. Linux, Mac OS, or Windows Linux Subsystem). We'll go over working on the command-line in detail later in the course.

You are required to submit your work in this course via GitHub. Today in class, you will setup everything on the command-line.

Command-line basics

There is plenty of material online that will help you figure out how to do various tasks on the command line. Commands you may need to know today:

- `ls` : lists the contents of the current directory.
- `pwd` : prints the path of the current directory.
- `cd <directory>` : changes your current directory to the specified directory.
- `cd ..` : changes current directory to the previous directory. Basically steps out of the current directory to the directory containing the current directory.
- `mkdir <directory>` : create a new directory with the specified name.
- `rmdir <directory>` : removes the specified directory. Note it has to be empty.
- `rm <filename>` : deletes the specified file.
- `mv <filename 1> <filename 1>` : Moves or renames a file.
- `cp <filename 1> <filename 2>` : copies an file. If you just provide a path to a directory, it copies the file into that directory with the same filename. If you specifiy a new filename, the copy has a new name. For example `cp File.1.txt File.2.txt` creates a copy of `File.1.txt` with the name `File.2.txt`. Meanwhile `cp File.1.txt my_directory`, where `my_directory` is a directory, creates a copy of `File.1.txt` in directory `my_directory` with the name `File.1.txt`.

For reference, here are some example resources I found by googling:

- Paths and Wildcards: https://www.warp.dev/terminus/linux-wildcards
- Basic commands like copy: https://kb.iu.edu/d/afsk
- General introduction to shell: https://github-pages.ucl.ac.uk/RCPSTrainingMaterials/HPCandHTCusingLegion/2_intro_to_shell.html
- Manual pages: https://www.geeksforgeeks.org/linux-man-page-entries-different-types/?ref=ml_lbp
- Chaining commands: https://www.geeksforgeeks.org/chaining-commands-in-linux/?ref=ml_lbp
- Piping: https://www.geeksforgeeks.org/piping-in-unix-or-linux/
- Using sed: https://www.geeksforgeeks.org/sed-command-linux-set-2/?ref=ml_lbp
- Various Unix commands: https://www.geeksforgeeks.org/linux-commands/?ref=lbp
- Cheat sheets:
  - https://www.stationx.net/unix-commands-cheat-sheet/
  - https://cheatography.com/davechild/cheat-sheets/linux-command-line/
  - https://www.theknowledgeacademy.com/blog/unix-commands-cheat-sheet/

These aren't necessarily the best resources. Feel free to search for better ones. Also, don't forget that Unix has built-in manual pages for all of its commands. Just type `man <command>` at the command prompt. Use the space-bar to scroll through the documentation and "q" to exit.

Setup and Submission

Our course repository is public. The instructions here aim to have you setup a fork of the course repository. Unfortunately because you are forking a public repo, your fork will have to be public also.

You should be familiar with git from the first semester of this course. I assume that you all have github accounts and have setup things to be able to push to github using ssh. The instuctions here lead you to:

We'll overview what you will do before going through step by step instructions.

1. Setup:
   1. Fork the class repository. Some directions in fork-a-repo.
   2. Create a directory on your personal system where you will keep all course materials.

   3. In that directory, clone your fork of the repository.
   4. Using `git remote`, set the upstream to be the class repo, so you can pull from the class and push to your fork.
2. Submission:
   1. Copy your solutions into the appropriate directory (e.g. into `Labs/Lab.2/`) and with appropriate filename `Lab.2.solution.ipynb'.
   2. Commit / push your solutions.
   3. Grant access to course instructors.

Below are step by step instructions with examples (including example directory naming convention). Feel free to modify things as you see fit.

Setup

You should only need to follow this instructions once. Here are some useful git commands:

- Git help: `git help`
- Git remote help: `git help remote`
- Check remote status: `git remote -v`
- Add a remote: `git remote add <stream name> <repo URL>`
- Add a remove: `git remote remove <stream name>`

Steps:

1. In a browser, log into GitHub and navigate to the course repository.
2. On the top right of the page, press the fork button to create a new fork into your own GitHub account.
3. After successful fork, you should find the browser showing your fork of the course repository. Use the green "Code" button to copy path to the repo into your the clipboard of your computer.
4. Open a shell on your personal computer.
5. If you have not done so already, create a new directory/folder where you will keep all course material to navigate to it. For example: `mkdir Data-3402` and `cd Data-3402`.
6. Clone your fork of the repository using `git clone` followed by the path you copied into your clipboard. (copy/paste)
7. Paste the URL to your fork in the worksheet for the TAs and instructors.
8. Now go into the directory of your clone (`cd DATA3402.Fall.2024`).
9. Type `git remote -v` to see the current setup for fetch and pull.
10. Note the URL you see. This should be the same as what you used for your clone for both push and fetch.
11. Delete the origin remote using `git remote remove origin`.
12. Add the course repo as your remote using `git remote add origin https://github.com/UTA-DataScience/DATA3402.Fall.2024.git`.
13. Change the push to point to your fork. This means you will need the URL to your clone we copied earlier and confirmed as the original origin. The command will look something like: `git remote set-url --push origin https://github.com/XXXXXX/DATA3402.Fall.2024.git`, where XXXXX is your username on GitHub.
14. Note that if you setup everything correctly, you now should be able to do `git pull` to get updates from the course repo, and do `git push` to push your commits into your own fork.

Submission

These instructions outline how you submit files. Some useful commands:

- To add a file to local repository: `git add <file>`.
- To commit all changed files into local repository: `git -a -m "A message"`. You need to provide some comment when you commit.
- To push the commited files from the local repository to GitHub: `git push`.
- To get updates from GitHub: `git pull`.

Steps:

1. To submit your labs, navigate to your clone of your fork of the course repository.
2. Use `git pull` to make sure you have the latest updates.
3. Make sure your copy of the lab your are working on is in the appropriate place in this clone. That means if you have the file elsewhere, copy it to the same directory in your clone of your fork.
4. Note that in order to avoid future conflicts, you should always name your solution differently than the original file in the class repo. For example if your file is still named `Lab.2.ipynb` you should rename it using the `mv` command: `mv Lab.2.ipynb Lab.2.solution.ipynb`.
5. Add and files you wish to submit into the repo. For example: `git add Labs/Lab.2/Lab.2.solution.ipynb`
6. Commit any changes: `git commit -a -m "Lab 2 updates"`
7. Push your changes: `git push`
8. Check on github website that your solutions have been properly submitted.

Before you leave the session today, make sure your GitHub Repo is setup. If you need to work further on your lab, navigate jupyter to the copy of the lab you just submitted and work there. Once done, repeat the commit and push commands to submit your updated solution. Note that lab 2 is due by midnight Monday 9/8/2024.

Uniform Distribution

Lets start with generating some fake random data. You can get a random number between 0 and 1 using the python random module as follow:

```
import random
x=random.random()
print("The Value of x is", x)
```

Everytime you call random, you will get a new number.

Exercise 1: Using random, write a function `generate_uniform(N, mymin, mymax)`, that returns a python list containing N random numbers between specified minimum and maximum value. Note that you may want to quickly work out on paper how to turn numbers between 0 and 1 to between other values.

```
# Skeleton
def generate_uniform(N,x_min,x_max):
    out = []
    ### BEGIN SOLUTION
import random

def generate_uniform(N, mymin, mymax):
    out = []
    for _ in range(N):
        rand_num = random.random(0.5)  # Generate a random number between 0 and 1
        scaled_num = mymin + (rand_num * (mymax - mymin))  # Scale to the desired range
        out.append(scaled_num)  # Append the scaled number to the list
    return out
    # Fill in your solution here

    ### END SOLUTION
    return out
```

```
import random

def generate_uniform(N, mymin, mymax):
    out = []
    for _ in range(N):
        rand_num = random.random()  # Generate a random number between 0 and 1
        scaled_num = mymin + (rand_num * (mymax - mymin))  # Scale to the desired range
        out.append(scaled_num)  # Append the scaled number to the list
    return out

# Call the function and assign the returned value to data
data = generate_uniform(1000, -10, 10)

# Print characteristics of the generated data
print("Data Type:", type(data))  # Should show <class 'list'>
print("Data Length:", len(data))  # Should be 1000
if len(data) > 0:
    print("Type of Data Contents:", type(data[0]))  # Should show <class 'float'>
    print("Data Minimum:", min(data))  # Should be >= -10
    print("Data Maximum:", max(data))  # Should be <= 10
```

```
Data Type: <class 'list'>
Data Length: 1000
Type of Data Contents: <class 'float'>
Data Minimum: -9.992773431027342
Data Maximum: 9.951823069938502
```

Exercise 2a: Write a function that computes the mean of values in a list. Recall the equation for the mean of a random variable $\mathbf{x}$ computed on a data set of n values $\{x_i\} = \{x_1, x_2, \ldots, x_n\}$ is $\bar{\mathbf{x}} = \frac{1}{n}\sum_i^n x_i$.

```
# Skeleton
def mean(Data):
    m=0.
    ### BEGIN SOLUTION

    # Fill in your solution here
    def compute_mean(data):
     if len(data) == 0:
        return 0  # Return 0 or handle empty list as needed
    total_sum = sum(data)  # Sum all elements in the list
    mean = total_sum / len(data)  # Calculate the mean
    return mean

    ### END SOLUTION

    return m


def compute_mean(data):
    if len(data) == 0:
        return 0  # Return 0 or handle empty list as needed
    total_sum = sum(data)  # Sum all elements in the list
    mean = total_sum / len(data)  # Calculate the mean
    return mean

# Test the function
data = [1, 2, 3, 4, 5]  # Example data
mean_value = compute_mean(data)
print("Mean of the data:", mean_value)
```

```
Mean of the data: 3.0
```

Exercise 2b: Write a function that computes the variance of values in a list. Recall the equation for the variance of a random variable $\mathbf{x}$ computed on a data set of n values $\{x_i\} = \{x_1, x_2, \ldots, x_n\}$ is $\langle \mathbf{x} \rangle = \frac{1}{n} \sum_i^n (x_i - \bar{\mathbf{x}})$.

```
# Skeleton
def variance(Data):
    m=0.

    ### BEGIN SOLUTION
def compute_variance(data):
    if len(data) == 0:
        return 0  # Return 0 or handle empty list as needed

    mean = compute_mean(data)  # First, calculate the mean
    squared_diffs = [(x - mean) ** 2 for x in data]  # Calculate squared differences
    variance = sum(squared_diffs) / len(data)  # Compute the variance
    return variance

def compute_mean(data):
    if len(data) == 0:
        return 0  # Handle empty list
    total_sum = sum(data)
    mean = total_sum / len(data)
    return mean


    ### END SOLUTION

    return m


# Test your solution here
def compute_variance(data):
    if len(data) == 0:
        return 0  # Return 0 or handle empty list as needed

    mean = compute_mean(data)  # First, calculate the mean
    squared_diffs = [(x - mean) ** 2 for x in data]  # Calculate squared differences
    variance = sum(squared_diffs) / len(data)  # Compute the variance
    return variance

def compute_mean(data):
    if len(data) == 0:
        return 0  # Handle empty list
    total_sum = sum(data)
    mean = total_sum / len(data)
    return mean
```

```
# Test the function
data = [1, 2, 3, 4, 5]  # Example data
variance_value = compute_variance(data)
print ("Variance of Data:", variance(data))
```

```
Variance of Data: 3.0
```

Histogramming

Exercise 3: Write a function that bins the data so that you can create a histogram. An example of how to implement histogramming is the following logic:

- User inputs a list of values `x` and optionally `n_bins` which defaults to 10.
- If not supplied, find the minimum and maximum (`x_min`,`x_max`) of the values in x.
- Determine the bin size (`bin_size`) by dividing the range of the function by the number of bins.
- Create an empty list of zeros of size `n_bins`, call it `hist`.
- Loop over the values in `x`
    - Loop over the values in `hist` with index `i`:
        - If x is between `x_min+i*bin_size` and `x_min+(i+1)*bin_size`, increment `hist[i]`.
        - For efficiency, try to use continue to goto the next bin and data point.
- Return `hist` and the list corresponding of the bin edges (i.e. of `x_min+i*bin_size`).

```
def histogram(x, n_bins=10):
    # If n_bins is not supplied, find the min and max
    x_min = min(x)
    x_max = max(x)

    # Determine the bin size
    bin_size = (x_max - x_min) / n_bins

    # Create a histogram list initialized to zeros
    hist = [0] * n_bins
    bin_edges = [x_min + i * bin_size for i in range(n_bins + 1)]

    # Loop over the values in x
    for value in x:
        for i in range(n_bins):
            # Check if the value falls within the current bin
            if x_min + i * bin_size <= value < x_min + (i + 1) * bin_size:
                hist[i] += 1  # Increment the count for the current bin
                break  # Move to the next value in x

    return hist, bin_edges
```

```
# Test your solution here
data = [1, 2, 2.5, 3, 4, 5, 6, 6.5, 7, 8, 9, 10]
hist_values, edges = histogram(data, n_bins=5)

print("Histogram Values:", hist_values)  # Counts in each bin
print("Bin Edges:", edges)  # Corresponding bin edges
h,b=histogram(data,100)
print(h)
```

```
Histogram Values: [3, 2, 2, 3, 1]
Bin Edges: [1.0, 2.8, 4.6, 6.4, 8.2, 10.0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
```

Exercise 4: Write a function that uses the histogram function in the previous exercise to create a text-based "graph". For example the output could look like the following:

```
[  0,  1] : ######
[  1,  2] : #####
[  2,  3] : ######
[  3,  4] : ####
[  4,  5] : ####
[  5,  6] : ######
[  6,  7] : #####
[  7,  8] : ######
[  8,  9] : ####
[  9, 10] : #####
```

Where each line corresponds to a bin and the number of `#`'s are proportional to the value of the data in the bin.

```
def print_histogram_graph(x, n_bins=10):
    hist_values, bin_edges = histogram(x, n_bins)

    # Find the maximum value in the histogram for scaling
    max_value = max(hist_values)

    # Print the histogram graph
    for i in range(n_bins):
        # Create a proportional bar length based on the maximum value
        if max_value > 0:  # Prevent division by zero
            bar_length = int((hist_values[i] / max_value) * 50)  # Scale to 50 characters wide
        else:
            bar_length = 0

        # Format the bin range for display
        bin_range = f"[{bin_edges[i]:>2.1f}, {bin_edges[i + 1]:>2.1f}]"
        # Create the bar
        bar = '#' * bar_length
         # Print the bin range and corresponding bar
        print(f"{bin_range} : {bar}")


ata = [1, 2, 2.5, 3, 4, 5, 6, 6.5, 7, 8, 9, 10]
h,b=histogram(data,20)
```

Functional Programming

Exercise 5: Write a function the applies a booling function (that returns true/false) to every element in data, and return a list of indices of elements where the result was true. Use this function to find the indices of entries greater than 0.5.

```
def find_indices(data, boolean_func):
    indices = []
    for i, value in enumerate(data):
        if boolean_func(value):
            indices.append(i)
    return indices

# Define a boolean function to check if a value is greater than 0.5
def is_greater_than_0_5(x):
    return x > 0.5


data = [0.1, 0.4, 0.6, 0.8, 0.3, 0.9, 0.2]
indices = find_indices(data, is_greater_than_0_5)
```

Exercise 6: The `inrange(mymin,mymax)` function below returns a function that tests if it's input is between the specified values. Write corresponding functions that test:

- Even
- Odd
- Greater than
- Less than
- Equal
- Divisible by

```
def in_range(mymin,mymax):
    def testrange(x):
        return x<mymax and x>=mymin
    return testrange
```

```
# Examples:
F1=inrange(0,10)
F2=inrange(10,20)

# Test of in_range
print (F1(0), F1(1), F1(10), F1(15), F1(20))
print (F2(0), F2(1), F2(10), F2(15), F2(20))

print ("Number of Entries passing F1:", len(where(data,F1)))
print ("Number of Entries passing F2:", len(where(data,F2)))


### BEGIN SOLUTION

def inrange(mymin, mymax):
    """Returns a function that checks if a number is within a specified range."""
    def check_range(x):
        return mymin <= x <= mymax
    return check_range

def is_even(x):
    """Checks if a number is even."""
    return x % 2 == 0

def is_odd(x):
    """Checks if a number is odd."""
    return x % 2 != 0

def is_greater_than(threshold):
    """Returns a function that checks if a number is greater than the specified threshold."""
    def check_greater(x):
        return x > threshold
    return check_greater

def is_less_than(threshold):
    """Returns a function that checks if a number is less than the specified threshold."""
    def check_less(x):
        return x < threshold
    return check_less

def is_equal(value):
    """Returns a function that checks if a number is equal to the specified value."""
    def check_equal(x):
        return x == value
    return check_equal

def is_divisible_by(divisor):
    """Returns a function that checks if a number is divisible by the specified divisor."""
    def check_divisible(x):
        return x % divisor == 0
    return check_divisible

### END SOLUTION


data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Check even numbers
even_indices = [i for i, value in enumerate(data) if is_even(value)]
print("Indices of even numbers:", even_indices)

# Check odd numbers
odd_indices = [i for i, value in enumerate(data) if is_odd(value)]
print("Indices of odd numbers:", odd_indices)

# Check numbers greater than 5
greater_than_5_indices = [i for i, value in enumerate(data) if is_greater_than(5)(value)]
print("Indices of numbers greater than 5:", greater_than_5_indices)

# Check numbers less than 5
less_than_5_indices = [i for i, value in enumerate(data) if is_less_than(5)(value)]
print("Indices of numbers less than 5:", less_than_5_indices)

# Check numbers equal to 5
equal_to_5_indices = [i for i, value in enumerate(data) if is_equal(5)(value)]
print("Indices of numbers equal to 5:", equal_to_5_indices)
```

```
# Check numbers divisible by 2
divisible_by_2_indices = [i for i, value in enumerate(data) if is_divisible_by(2)(value)]
print("Indices of numbers divisible by 2:", divisible_by_2_indices)
```

```
Indices of even numbers: [1, 3, 5, 7, 9]
Indices of odd numbers: [0, 2, 4, 6, 8]
Indices of numbers greater than 5: [5, 6, 7, 8, 9]
Indices of numbers less than 5: [0, 1, 2, 3]
Indices of numbers equal to 5: [4]
Indices of numbers divisible by 2: [1, 3, 5, 7, 9]
```

Exercise 7: Repeat the previous exercise using `lambda` and the built-in python functions sum and map instead of your solution above.

```
### BEGIN SOLUTION
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Count of even numbers
even_count = sum(map(lambda x: x % 2 == 0, data))
print("Count of even numbers:", even_count)

# Count of odd numbers
odd_count = sum(map(lambda x: x % 2 != 0, data))
print("Count of odd numbers:", odd_count)

# Count of numbers greater than 5
greater_than_5_count = sum(map(lambda x: x > 5, data))
print("Count of numbers greater than 5:", greater_than_5_count)

# Count of numbers less than 5
less_than_5_count = sum(map(lambda x: x < 5, data))
print("Count of numbers less than 5:", less_than_5_count)

# Count of numbers equal to 5
equal_to_5_count = sum(map(lambda x: x == 5, data))
print("Count of numbers equal to 5:", equal_to_5_count)

# Count of numbers divisible by 2
divisible_by_2_count = sum(map(lambda x: x % 2 == 0, data))
print("Count of numbers divisible by 2:", divisible_by_2_count)

### END SOLUTION
```

```
Count of even numbers: 5
Count of odd numbers: 5
Count of numbers greater than 5: 5
Count of numbers less than 5: 4
Count of numbers equal to 5: 1
Count of numbers divisible by 2: 5
```

Monte Carlo

Exercise 7: Write a "generator" function called `generate_function(func,x_min,x_max,N)`, that instead of generating a flat distribution, generates a distribution with functional form coded in `func`. Note that `func` will always be > 0.

Use the test function below and your histogramming functions above to demonstrate that your generator is working properly.

Hint: A simple, but slow, solution is to a draw random number `test_x` within the specified range and another number `p` between the `min` and `max` of the function (which you will have to determine). If `p<=function(test_x)`, then place `test_x` on the output. If not, repeat the process, drawing two new numbers. Repeat until you have the specified number of generated numbers, `N`. For this problem, it's OK to determine the `min` and `max` by numerically sampling the function.

```
import random

def generate_function(func, x_min, x_max, N):
    """Generator function that generates numbers based on the provided functional form."""
    # Sample the function to find the minimum and maximum values
    sample_points = 1000
    x_samples = [x_min + (x_max - x_min) * i / sample_points for i in range(sample_points + 1)]
    y_samples = [func(x) for x in x_samples]
    y_min = min(y_samples)
    y_max = max(y_samples)

    while N > 0:
        # Draw a random x in the range
        test_x = random.uniform(x_min, x_max)
        # Draw a random y in the range of the function's output
        test_y = random.uniform(y_min, y_max)

        # Check if the sample is accepted
        if test_y <= func(test_x):
            yield test_x
            N -= 1



def test_function(x):
    return x ** 2  # Example: parabola (always > 0 for x > 0)

# Using the histogram function to visualize the results
def main():
    # Generate samples
    N = 1000
    x_min = 0
    x_max = 10

    samples = list(generate_function(test_function, x_min, x_max, N))

    # Create histogram of the generated samples
    hist_values, bin_edges = histogram(samples, n_bins=20)

    # Print histogram
    print("Histogram Values:", hist_values)
    print("Bin Edges:", bin_edges)

    # Print the histogram graph
    print_histogram_graph(samples, n_bins=20)

# Run the main function
if __name__ == "__main__":
    main()
```

```
Histogram Values: [4, 5, 2, 8, 12, 16, 16, 31, 41, 38, 23, 56, 68, 74, 81, 100, 94, 100, 114, 116]
Bin Edges: [0.7305070740756969, 1.1939070048069538, 1.6573069355382104, 2.120706866269467, 2.584106797000724, 3.0475067277319807, 3.5109
[0.7, 1.2] : #
[1.2, 1.7] : ##
[1.7, 2.1] :
[2.1, 2.6] : ###
[2.6, 3.0] : #####
[3.0, 3.5] : ######
[3.5, 4.0] : ######
[4.0, 4.4] : #############
[4.4, 4.9] : #################
[4.9, 5.4] : ################
[5.4, 5.8] : #########
[5.8, 6.3] : ########################
[6.3, 6.8] : #############################
[6.8, 7.2] : ###############################
[7.2, 7.7] : ##################################
[7.7, 8.1] : ###########################################
[8.1, 8.6] : ########################################
[8.6, 9.1] : ###########################################
[9.1, 9.5] : #################################################
[9.5, 10.0] : ##################################################
```

Exercise 8: Use your function to generate 1000 numbers that are normal distributed, using the `gaussian` function below. Confirm the mean and variance of the data is close to the mean and variance you specify when building the Gaussian. Histogram the data.

```
import random
import math
import numpy as np

def gaussian(x, mu, sigma):
    """Gaussian (normal) distribution function."""
    return (1 / (sigma * math.sqrt(2 * math.pi))) * math.exp(-((x - mu) ** 2) / (2 * sigma ** 2))

def generate_gaussian(mu, sigma, N):
    """Generator function for normally distributed samples."""
    # Sample to find the maximum of the Gaussian function
    x_min = mu - 4 * sigma  # 4 standard deviations below the mean
    x_max = mu + 4 * sigma  # 4 standard deviations above the mean
    x_max = mu + 4 * sigma  # 4 standard deviations above the mean
    sample_points = 1000
    x_samples = [x_min + (x_max - x_min) * i / sample_points for i in range(sample_points + 1)]
    y_samples = [gaussian(x, mu, sigma) for x in x_samples]
    y_min = min(y_samples)
    y_max = max(y_samples)

    # Generate samples
    samples = []
    while len(samples) < N:
        test_x = random.uniform(x_min, x_max)
        test_y = random.uniform(y_min, y_max)

        if test_y <= gaussian(test_x, mu, sigma):
            samples.append(test_x)

    return samples

def main():
    mu = 0    # Mean
    sigma = 1  # Standard deviation
    N = 1000   # Number of samples

    # Generate samples
    samples = generate_gaussian(mu, sigma, N)

    # Calculate mean and variance of the generated samples
    calculated_mean = np.mean(samples)
    calculated_variance = np.var(samples)

    # Output mean and variance
    print(f"Calculated Mean: {calculated_mean:.2f}")
    print(f"Calculated Variance: {calculated_variance:.2f}")

    # Create histogram of the generated samples
    hist_values, bin_edges = histogram(samples, n_bins=20)

    # Print histogram graph
    print("Histogram Values:", hist_values)
    print("Bin Edges:", bin_edges)
    print_histogram_graph(samples, n_bins=20)

# Run the main function
if __name__ == "__main__":
    main()
```

```
Calculated Mean: 0.02
Calculated Variance: 1.00
Histogram Values: [2, 7, 12, 16, 37, 42, 62, 80, 97, 85, 110, 119, 82, 80, 58, 42, 36, 15, 11, 6]
Bin Edges: [-2.8447632187834966, -2.5705895250405573, -2.296415831297618, -2.022242137554678, -1.7480684438117389, -1.4738947500687996,
[-2.8, -2.6] : 
[-2.6, -2.3] : ##
[-2.3, -2.0] : #####
[-2.0, -1.7] : ######
[-1.7, -1.5] : ###############
[-1.5, -1.2] : #################
[-1.2, -0.9] : ##########################
[-0.9, -0.7] : #################################
[-0.7, -0.4] : ########################################
[-0.4, -0.1] : ###################################
[-0.1, 0.2] : ##############################################
[0.2, 0.4] : ##################################################
[0.4, 0.7] : ##################################
[0.7, 1.0] : #################################
[1.0, 1.3] : ########################
[1.3, 1.5] : #################
[1.5, 1.8] : ###############
[1.8, 2.1] : ######
[2.1, 2.4] : ####
[2.4, 2.6] : ##
```

Exercise 9: Combine your `generate_function`, `where`, and `in_range` functions above to create an integrate function. Use your integrate function to show that approximately 68% of Normal distribution is within one variance.

```
import random
import math
import numpy as np
import numpy as np

def gaussian(x, mu, sigma):
    """Gaussian (normal) distribution function."""
    return (1 / (sigma * math.sqrt(2 * math.pi))) * math.exp(-((x - mu) ** 2) / (2 * sigma ** 2))

def generate_function(func, x_min, x_max, N):
    """Generator function that generates numbers based on the provided functional form."""
    sample_points = 1000
    x_samples = [x_min + (x_max - x_min) * i / sample_points for i in range(sample_points + 1)]
    y_samples = [func(x) for x in x_samples]
    y_min = min(y_samples)
    y_max = max(y_samples)

    while N > 0:
        test_x = random.uniform(x_min, x_max)
        test_y = random.uniform(y_min, y_max)

        if test_y <= func(test_x):
            yield test_x
            N -= 1

def in_range(mymin, mymax):
    """Returns a function that checks if a number is within a specified range."""
    return lambda x: mymin <= x <= mymax

def integrate(func, x_min, x_max, N):
    """Estimate the integral of a function using Monte Carlo integration."""
    samples = list(generate_function(func, x_min, x_max, N))
    return samples

def main():
    mu = 0       # Mean of the normal distribution
    sigma = 1    # Standard deviation of the normal distribution
    N = 10000    # Number of samples for integration

    # Generate samples for the Gaussian function
    samples = integrate(lambda x: gaussian(x, mu, sigma), mu - 4 * sigma, mu + 4 * sigma, N)

    # Count how many samples fall within one standard deviation
    lower_bound = mu - sigma
    upper_bound = mu + sigma
    within_one_std_dev = len(list(filter(in_range(lower_bound, upper_bound), samples)))

    # Calculate the proportion
    proportion = within_one_std_dev / N * 100

    print(f"Proportion of samples within one standard deviation ({lower_bound:.2f}, {upper_bound:.2f}): {proportion:.2f}%")

# Run the main function
if __name__ == "__main__":
    main()
```

```
Proportion of samples within one standard deviation (-1.00, 1.00): 67.59%
```

Start coding or generate with AI.