```
users = [ {"id":0, "name":"Hero"}, {"id":1,"name":"Dunn"}, {"id":2, "name":"Sue"}, {"id":3,"name":"Chi"}, {"id":4, "name":"Thor"}, {"id":5,"n
friendship_pairs = [(0, 1), (0, 2), (1, 2), (1,3), (2, 3), (3, 4), (4, 5), (5, 6), (5, 7), (6,8), (7, 8), (8, 9)]
friendships = {user["id"]: [] for user in users}
for i, j in friendship_pairs:
 friendships[i].append(j)
 friendships[j].append(i)
def no_of_friends(user):
  user_id = user["id"]
  friend_ids = friendships[user_id]
  return len(friend_ids)
total_connections = sum(no_of_friends(user) for user in users)
avg_connections = total_connections / len(users)
no_friends_by_id = [(user["id"], no_of_friends(user)) for user in users]
no_friends_by_id.sort(key=lambda id_and_friends: id_and_friends[1], reverse=True)
def foaf_ids_bad(user):
 return[foaf_id
 for friend_id in friendships[user["id"]]
 for foaf_id in friendships[friend_id]]
from collections import Counter
def friends_of_friends(user):
 user_id = user["id"]
 return Counter(foaf_id
  for friend_id in friendships[user_id]
  for foaf_id in friendships[friend_id]
  if foaf_id != user_id
  and foaf_id not in friendships[user_id])
print(friends_of_friends(users[3]))
```

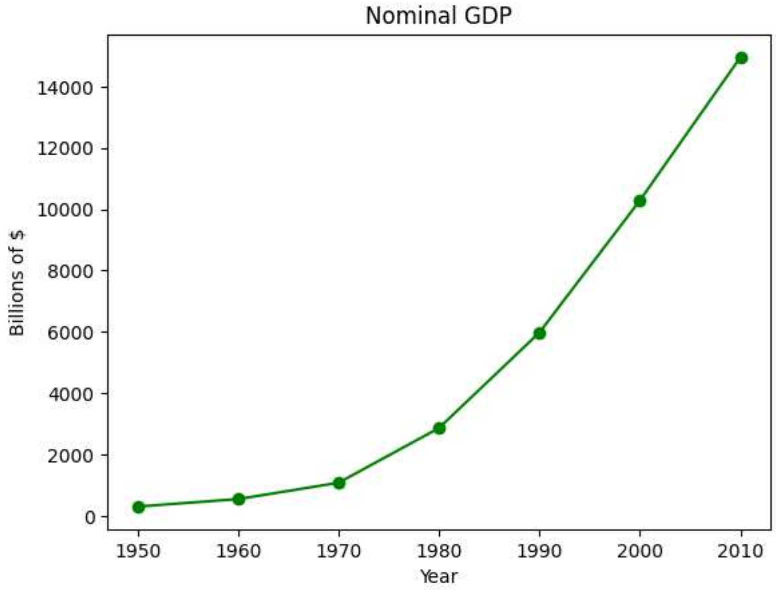
```
Counter({0: 2, 5: 1})
```

```
interests = [(0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0,"Java"), (0, "Spark"), (0, "Storm"), (0, "Cassandra"), (1,"NoSQL"), (1, "MongoD
def data_scientists_who_like(target_interest):
    return [user_id
        for user_id, user_interest in interests
            if user_interest == target_interest]
from collections import defaultdict, Counter

# Keys are interests, values are lists of user_ids with that interest
user_ids_by_interest = defaultdict(list)
for user_id, interest in interests:
    user_ids_by_interest[interest].append(user_id)

# Keys are user_ids, values are lists of interests for that user_id.
interests_by_user_id = defaultdict(list)
for user_id, interest in interests:
    interests_by_user_id[user_id].append(interest)

def most_common_interests_with(user):
    return Counter(
        interested_user_id
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"]
    )
print(most_common_interests_with)
```

```
<function most_common_interests_with at 0x7fe7516f5c60>
```

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# Create a line chart, years on the x-axis, GDP on the y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# Add a title
plt.title("Nominal GDP")

# Add a label to the y-axis
plt.ylabel("Billions of $")
```

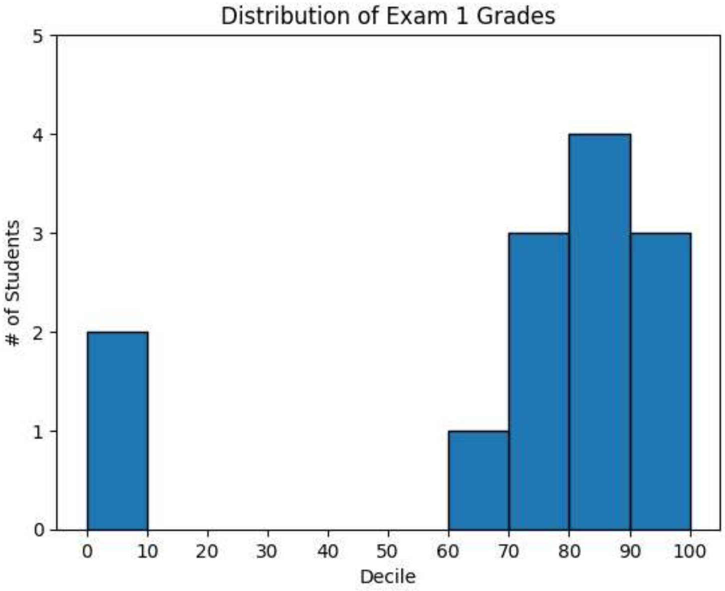
```
# Add a label to the x-axis
plt.xlabel("Year")

# Display the chart
plt.show()

# Save the figure as a PNG file
plt.savefig("gdp_chart.png")
```



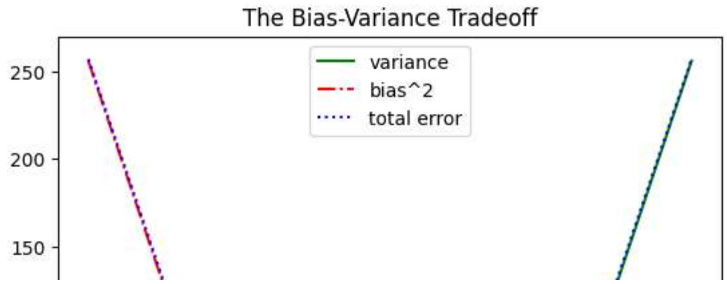
```
<Figure size 640x480 with 0 Axes>
```

```
from matplotlib import pyplot as plt

movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# Plot bars with left x-coordinates [0, 1, 2, 3, 4] and heights [num_oscars]
plt.bar(range(len(movies)), num_oscars)

plt.title("My Favorite Movies")
plt.ylabel("# of Academy Awards")

# Label x-axis with movie names at bar centers
plt.xticks(range(len(movies)), movies)

plt.show()
```

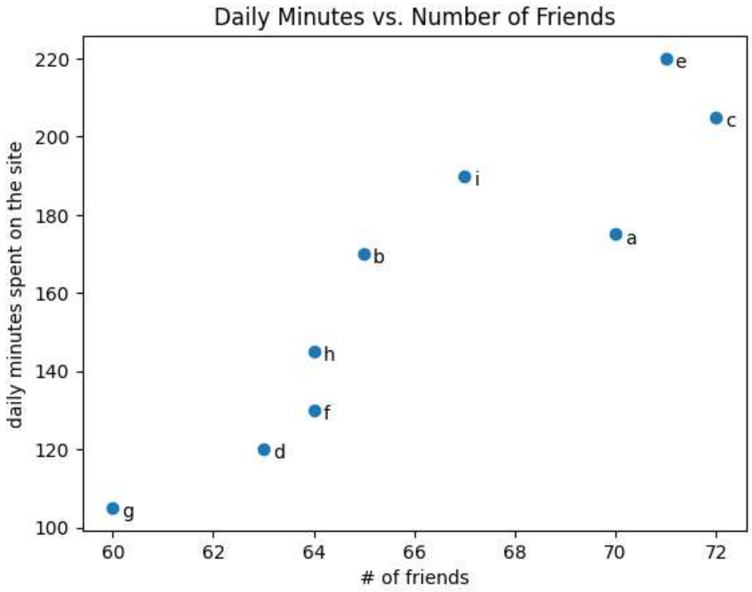
```
from collections import Counter
from matplotlib import pyplot as plt

grades = [83, 95, 91, 87, 70, 0, 85, 82, 100, 67, 73, 77, 0]

# Bucket grades by decile, but put 100 in with the 90s
histogram = Counter(min(grade // 10 * 10, 90) for grade in grades)

plt.bar([x + 5 for x in histogram.keys()], histogram.values(), 10, edgecolor=(0, 0, 0))
plt.axis([-5, 105, 0, 5])
plt.xticks([10 * i for i in range(11)])
plt.xlabel("Decile")
plt.ylabel("# of Students")
plt.title("Distribution of Exam 1 Grades")
plt.show()
```



```
from matplotlib import pyplot as plt

variance = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# We can make multiple calls to plt.plot to show multiple series on the same chart
plt.plot(xs, variance, 'g-', label='variance')  # green solid line
plt.plot(xs, bias_squared, 'r-.', label='bias^2')  # red dot-dashed line
plt.plot(xs, total_error, 'b:', label='total error')  # blue dotted line

# Because we have assigned labels to each series, we can get a legend for free (loc=9 means "top center")
plt.legend(loc=9)
plt.xlabel("model complexity")
plt.xticks([])
plt.title("The Bias-Variance Tradeoff")
plt.show()
```



```
from matplotlib import pyplot as plt

friends = [70, 65, 72, 63, 71, 64, 60, 64, 67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

plt.scatter(friends, minutes)

for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label, xy=(friend_count, minute_count), xytext=(5, -5), textcoords='offset points')

plt.title("Daily Minutes vs. Number of Friends")
plt.xlabel("# of friends")
plt.ylabel("daily minutes spent on the site")
plt.show()
```



```
from typing import List, Tuple, Callable
import math

Vector = List[float]
Matrix = List[Vector]

def add(v: Vector, w: Vector) -> Vector:
    assert len(v) == len(w)
    return [vi + wi for vi, wi in zip(v, w)]

def subtract(v: Vector, w: Vector) -> Vector:
    assert len(v) == len(w)
    return [vi - wi for vi, wi in zip(v, w)]

def vector_sum(vectors: List[Vector]) -> Vector:
    assert vectors
    num_elements = len(vectors[0])
    assert all(len(v) == num_elements for v in vectors)
    return [sum(vector[i] for vector in vectors) for i in range(num_elements)]

def scalar_multiply(c: float, v: Vector) -> Vector:
    return [c * vi for vi in v]
```

```
def vector_mean(vectors: List[Vector]) -> Vector:
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))

def dot(v: Vector, w: Vector) -> float:
    assert len(v) == len(w)
    return sum(vi * wi for vi, wi in zip(v, w))

def sum_of_squares(v: Vector) -> float:
    return dot(v, v)

def magnitude(v: Vector) -> float:
    return math.sqrt(sum_of_squares(v))

def squared_distance(v: Vector, w: Vector) -> float:
    return sum_of_squares(subtract(v, w))

def distance(v: Vector, w: Vector) -> float:
    return math.sqrt(squared_distance(v, w))

def shape(A: Matrix) -> Tuple[int, int]:
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0
    return num_rows, num_cols

def get_row(A: Matrix, i: int) -> Vector:
    return A[i]

def get_column(A: Matrix, j: int) -> Vector:
    return [A_i[j] for A_i in A]

def make_matrix(num_rows: int, num_cols: int, entry_fn: Callable[[int, int], float]) -> Matrix:
    return [[entry_fn(i, j) for j in range(num_cols)] for i in range(num_rows)]

def identity_matrix(n: int) -> Matrix:
    return make_matrix(n, n, lambda i, j: 1 if i == j else 0)

    # Example usage of the functions

# Vector operations
v1 = [1, 2, 3]
v2 = [4, 5, 6]
v_sum = vector_sum([v1, v2])
print("Vector Sum:", v_sum)

v_diff = subtract(v1, v2)
print("Vector Difference:", v_diff)

v_mean = vector_mean([v1, v2])
print("Vector Mean:", v_mean)

dot_product = dot(v1, v2)
print("Dot Product:", dot_product)

squared_sum = sum_of_squares(v1)
print("Sum of Squares:", squared_sum)

v_magnitude = magnitude(v1)
print("Vector Magnitude:", v_magnitude)

# Matrix operations
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
A_shape = shape(A)
print("Matrix Shape:", A_shape)

row = get_row(A, 1)
print("Row:", row)

column = get_column(A, 2)
print("Column:", column)

identity = identity_matrix(3)
print("Identity Matrix:")
for row in identity:
    print(row)
```

```
Vector Sum: [5, 7, 9]
Vector Difference: [-3, -3, -3]
Vector Mean: [2.5, 3.5, 4.5]
Dot Product: 32
Sum of Squares: 14
Vector Magnitude: 3.7416573867739413
Matrix Shape: (3, 3)
Row: [4, 5, 6]
Column: [3, 6, 9]
Identity Matrix:
[1, 0, 0]
[0, 1, 0]
[0, 0, 1]
```

```
from collections import Counter
import matplotlib.pyplot as plt

friend_counts = Counter(num_friends)
xs = range(101)  # largest value is 100
ys = [friend_counts[x] for x in xs]

plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])  # x-axis and y-axis limits
plt.title("Histogram of Friend Counts")
plt.xlabel("# of friends")
plt.ylabel("# of people")
plt.show()

num_points = len(num_friends)
largest_value = max(num_friends)
smallest_value = min(num_friends)
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]
second_smallest_value = sorted_values[1]
second_largest_value = sorted_values[-2]

def mean(xs: List[float]) -> float:
    return sum(xs) / len(xs)

def median_odd(xs: List[float]) -> float:
    return sorted(xs)[len(xs) // 2]

def median_even(xs: List[float]) -> float:
    sorted_xs = sorted(xs)
    hi_midpoint = len(xs) // 2
    return (sorted_xs[hi_midpoint - 1] + sorted_xs[hi_midpoint]) / 2

def median(v: List[float]) -> float:
    return median_even(v) if len(v) % 2 == 0 else median_odd(v)

def quantile(xs: List[float], p: float) -> float:
    p_index = int(p * len(xs))
    return sorted(xs)[p_index]

def mode(x: List[float]) -> List[float]:
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.items() if count == max_count]

def data_range(xs: List[float]) -> float:
    return max(xs) - min(xs)

from scratch.linear_algebra import sum_of_squares

def de_mean(xs: List[float]) -> List[float]:
    x_bar = mean(xs)
    return [x - x_bar for x in xs]

def variance(xs: List[float]) -> float:
    assert len(xs) >= 2
    n = len(xs)
    deviations = de_mean(xs)
    return sum_of_squares(deviations) / (n - 1)
```

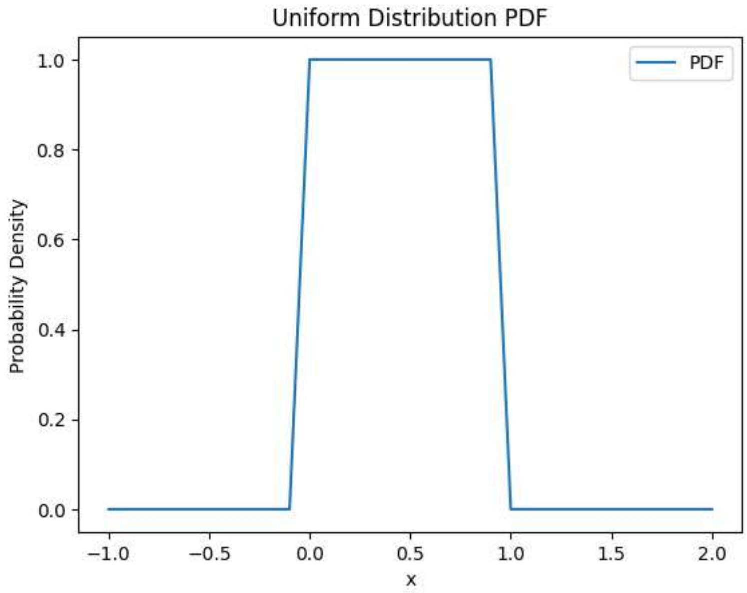
```
import math

def standard_deviation(xs: List[float]) -> float:
    return math.sqrt(variance(xs))

def interquartile_range(xs: List[float]) -> float:
    return quantile(xs, 0.75) - quantile(xs, 0.25)

from scratch.linear_algebra import dot

def covariance(xs: List[float], ys: List[float]) -> float:
    assert len(xs) == len(ys)
    return dot(de_mean(xs), de_mean(ys)) / (len(xs) - 1)

def correlation(xs: List[float], ys: List[float]) -> float:
    stdev_x = standard_deviation(xs)
    stdev_y = standard_deviation(ys)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(xs, ys) / stdev_x / stdev_y
    else:
        return 0
```

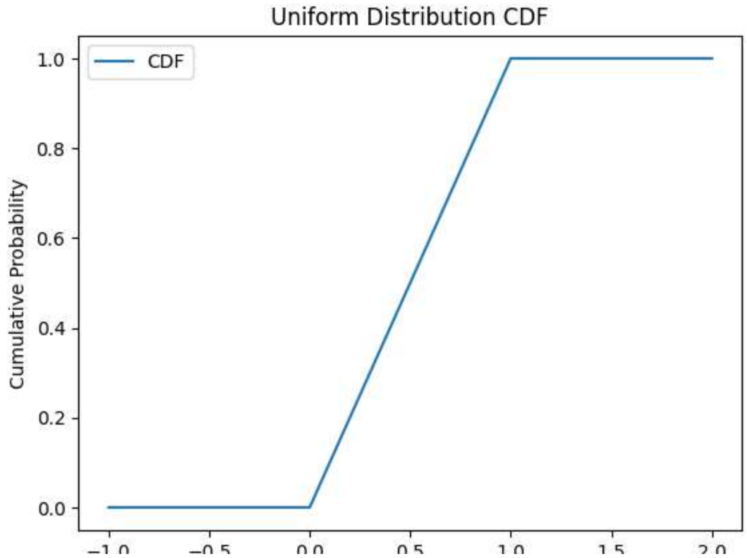
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-56b1ad72e33f> in <cell line: 4>()
      2 import matplotlib.pyplot as plt
      3 
----> 4 friend_counts = Counter(num_friends)
      5 xs = range(101)  # largest value is 100
      6 ys = [friend_counts[x] for x in xs]

NameError: name 'num_friends' is not defined
```

SEARCH STACK OVERFLOW

```
def uniform_pdf(x: float) -> float:
    return 1 if 0 <= x < 1 else 0

def uniform_cdf(x: float) -> float:
    if x < 0:
        return 0
    elif x < 1:
        return x
    else:
        return 1
```

```
import matplotlib.pyplot as plt

# Generate a range of values for x
x_values = [x / 10 for x in range(-10, 21)]

# Calculate the PDF and CDF values for each x
pdf_values = [uniform_pdf(x) for x in x_values]
cdf_values = [uniform_cdf(x) for x in x_values]

# Plot the PDF
plt.plot(x_values, pdf_values, label='PDF')
plt.xlabel('x')
plt.ylabel('Probability Density')
plt.title('Uniform Distribution PDF')
plt.legend()
plt.show()

# Plot the CDF
plt.plot(x_values, cdf_values, label='CDF')
plt.xlabel('x')
plt.ylabel('Cumulative Probability')
plt.title('Uniform Distribution CDF')
plt.legend()
plt.show()
```





```
import matplotlib.pyplot as plt
import random

# Normal Distribution
x_values = [x / 10 for x in range(-50, 51)]
pdf_values = [normal_pdf(x, mu=0, sigma=1) for x in x_values]
cdf_values = [normal_cdf(x, mu=0, sigma=1) for x in x_values]

# Plot the PDF and CDF of the Normal Distribution
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(x_values, pdf_values)
plt.xlabel('x')
plt.ylabel('Probability Density')
plt.title('Normal Distribution PDF')

plt.subplot(1, 2, 2)
plt.plot(x_values, cdf_values)
plt.xlabel('x')
plt.ylabel('Cumulative Probability')
plt.title('Normal Distribution CDF')

plt.tight_layout()
plt.show()

# Binomial Distribution
n = 20
p = 0.5
num_trials = 1000
```

```
# Simulate Binomial trials
binomial_values = [binomial(n, p) for _ in range(num_trials)]

# Count the occurrences of each value
value_counts = [binomial_values.count(k) for k in range(n+1)]

# Plot the Binomial Distribution
plt.bar(range(n+1), value_counts)
plt.xlabel('k')
plt.ylabel('Count')
plt.title('Binomial Distribution')

plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-17-7c541cf282c1> in <cell line: 6>()
      4 # Normal Distribution
      5 x_values = [x / 10 for x in range(-50, 51)]
----> 6 pdf_values = [normal_pdf(x, mu=0, sigma=1) for x in x_values]
      7 cdf_values = [normal_cdf(x, mu=0, sigma=1) for x in x_values]
      8

<ipython-input-17-7c541cf282c1> in <listcomp>(.0)
      4 # Normal Distribution
      5 x_values = [x / 10 for x in range(-50, 51)]
----> 6 pdf_values = [normal_pdf(x, mu=0, sigma=1) for x in x_values]
      7 cdf_values = [normal_cdf(x, mu=0, sigma=1) for x in x_values]
      8

NameError: name 'normal_pdf' is not defined
```

SEARCH STACK OVERFLOW

```
from typing import Tuple
import math
from scratch.probability import normal_cdf
from scratch.probability import inverse_normal_cdf

def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:
    """Returns mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma

# The normal_cdf is the probability the variable is below a threshold
normal_probability_below = normal_cdf

# It's above the threshold if it's not below the threshold
def normal_probability_above(lo: float, mu: float = 0, sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is greater than lo."""
    return 1 - normal_cdf(lo, mu, sigma)

# It's between if it's less than hi, but not less than lo
def normal_probability_between(lo: float, hi: float, mu: float = 0, sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is between lo and hi."""
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# It's outside if it's not between
def normal_probability_outside(lo: float, hi: float, mu: float = 0, sigma: float = 1) -> float:
    """The probability that an N(mu, sigma) is not between lo and hi."""
    return 1 - normal_probability_between(lo, hi, mu, sigma)

def normal_upper_bound(probability: float, mu: float = 0, sigma: float = 1) -> float:
    """Returns the z for which P(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability: float, mu: float = 0, sigma: float = 1) -> float:
    """Returns the z for which P(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability: float, mu: float = 0, sigma: float = 1) -> Tuple[float, float]:
    """Returns the symmetric (about the mean) bounds that contain the specified probability"""
    tail_probability = (1 - probability) / 2
    # upper bound should have tail_probability above it
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)
    # lower bound should have tail_probability below it
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)
    return lower_bound, upper_bound

mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
lower_bound, upper_bound = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# 95% bounds based on assumption p is 0.5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# an actual mu and sigma based on p = 0.55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)

# a type 2 error means we fail to reject the null hypothesis,
# which will happen when X is still in our original interval
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-24-031fa206af84> in <cell line: 3>()
      1 from typing import Tuple
      2 import math
----> 3 from scratch.probability import normal_cdf
      4 from scratch.probability import inverse_normal_cdf
      5

ModuleNotFoundError: No module named 'scratch.probability'

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES SEARCH STACK OVERFLOW

```
from typing import Tuple
import math
from scratch.probability import normal_cdf, normal_probability_between

def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:
    """Returns mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma

def normal_two_sided_bounds(probability: float, mu: float = 0, sigma: float = 1) -> Tuple[float, float]:
    """Returns the symmetric (about the mean) bounds that contain the specified probability"""
    tail_probability = (1 - probability) / 2

    # Upper bound should have tail probability above it
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)

    # Lower bound should have tail probability below it
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)

    return lower_bound, upper_bound

def normal_power(n: int, p: float, lo: float, hi: float, mu_1: float, sigma_1: float) -> float:
    """Calculates the power of a hypothesis test"""
    type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
    power = 1 - type_2_probability
    return power

# Example usage
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)
power = normal_power(1000, 0.5, lo, hi, mu_1, sigma_1)

print("Power:", power)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-22-5939bfe7e167> in <cell line: 3>()
      1 from typing import Tuple
      2 import math
----> 3 from scratch.probability import normal_cdf, normal_probability_between
      4 
      5 def normal_approximation_to_binomial(n: int, p: float) -> Tuple[float, float]:

ModuleNotFoundError: No module named 'scratch'

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES SEARCH STACK OVERFLOW

```
! pip install scratch
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting scratch
  Downloading scratch-1.0.0.tar.gz (4.3 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: scratch
  Building wheel for scratch (setup.py) ... done
  Created wheel for scratch: filename=scratch-1.0.0-py2.py3-none-any.whl size=4890 sha256=34002715c4ddfa8ab0ed8db8dcef7f6cc1fe0e6a7d4de8
  Stored in directory: /root/.cache/pip/wheels/3d/bf/6e/e1ae84c0715e36d2c2c808a0ef17c289866ca1e79c32ad378c
Successfully built scratch
Installing collected packages: scratch
Successfully installed scratch-1.0.0
```

```
import sys
import re

# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]

# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

```
import sys

count = 0
for line in sys.stdin:
    count += 1

# print goes to sys.stdout
print(count)
```

```
0
```

```
import sys
from collections import Counter

# pass in the number of words as the first argument
try:
    num_words = int(sys.argv[1])
except:
    print("usage: most_common_words.py num_words")
    sys.exit(1)  # nonzero exit code indicates error

# strip(): Remove spaces at the beginning and end of the string
```

```
counter = Counter(word.lower() for line in sys.stdin
                 for word in line.strip().split() if word)  # skip empty 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

$ cat the bible . txt | python most common words . py 10

```
# 'r' means read-only, it's assumed if you leave it out
file_for_reading = open('readingfile.txt', 'r')
file_for_reading2 = open('readingfile.txt')

# 'w' is write -- will destroy the file if it already exists!
file_for_writing = open('writingfile.txt', 'w')

# 'a' is append -- for adding to the end of the file
file_for_appending = open('appendingfile.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

CH 10

1 D

```
from collections import Counter
import math
import random
import matplotlib.pyplot as plt
from scratch.probability import inverse_normal_cdf

def bucketize(point, bucket_size):
    """Floor the point to the next lower multiple of bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points, bucket_size):
    """Buckets the points and counts how many in each bucket"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points, bucket_size, title=""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
    plt.title(title)
    plt.show()

random.seed(0)
uniform = [200 * random.random() - 100 for _ in range(10000)]
normal = [57 * inverse_normal_cdf(random.random()) for _ in range(10000)]

plot_histogram(uniform, 10, 'Uniform Histogram')
plot_histogram(normal, 10, 'Normal Histogram')
```

2D

```
from collections import Counter, defaultdict
from functools import partial, reduce
from scratch.statistics import correlation, standard_deviation, mean
from scratch.probability import inverse_normal_cdf
import math
import random
import matplotlib.pyplot as plt
import dateutil.parser

def random_normal():
    """Returns a random draw from a standard normal distribution"""
    return inverse_normal_cdf(random.random())

xs = [random_normal() for _ in range(1000)]
ys1 = [x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]

plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("Very Different Joint Distributions")
plt.show()

print(correlation(xs, ys1))
print(correlation(xs, ys2))
```

Splitting of data

```
training and testing without o/p
```

training and testing without o/p

```
import random
from typing import TypeVar, List, Tuple

X = TypeVar('X')  # generic type to represent a data point

def split_data(data: List[X], prob: float) -> Tuple[List[X], List[X]]:
    """Split data into fractions [prob, 1 - prob]"""
    data = data[:]  # Make a shallow copy
    random.shuffle(data)  # because shuffle modifies the list.
    cut = int(len(data) * prob)  # Use prob to find a cutoff
    return data[:cut], data[cut:]  # and split the shuffled list there.

data = [n for n in range(1000)]
train, test = split_data(data, 0.75)

# The proportions should be correct
assert len(train) == 750
assert len(test) == 250

# And the original data should be preserved (in some order)
assert sorted(train + test) == data

Y = TypeVar('Y')  # generic type to represent output variables

def train_test_split(xs: List[X], ys: List[Y], test_pct: float) -> Tuple[List[X], List[X], List[Y], List[Y]]:
    # Generate the indices and split them
    idxs = [i for i in range(len(xs))]
    train_idxs, test_idxs = split_data(idxs, 1 - test_pct)
    return (
        [xs[i] for i in train_idxs],  # x_train
        [xs[i] for i in test_idxs],  # x_test
        [ys[i] for i in train_idxs],  # y_train
        [ys[i] for i in test_idxs]  # y_test
    )

xs = [x for x in range(1000)]  # xs are 1 ... 1000
ys = [2 * x for x in xs]  # each y_i is twice x_i
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.25)

# Check that the proportions are correct
assert len(x_train) == len(y_train) == 750
assert len(x_test) == len(y_test) == 250

# Check that the corresponding data points are paired correctly
assert all(y == 2 * x for x, y in zip(x_train, y_train))
assert all(y == 2 * x for x, y in zip(x_test, y_test))
```

Training and splitting data with values

```
import random
from typing import TypeVar, List, Tuple

X = TypeVar('X')  # generic type to represent a data point

def split_data(data: List[X], prob: float) -> Tuple[List[X], List[X]]:
    """Split data into fractions [prob, 1 - prob]"""
    data = data[:]  # Make a shallow copy
    random.shuffle(data)  # because shuffle modifies the list.
    cut = int(len(data) * prob)  # Use prob to find a cutoff
    return data[:cut], data[cut:]  # and split the shuffled list there.

data = [n for n in range(1000)]
train, test = split_data(data, 0.75)

# The proportions should be correct
assert len(train) == 750
assert len(test) == 250

# And the original data should be preserved (in some order)
assert sorted(train + test) == data

Y = TypeVar('Y')  # generic type to represent output variables

def train_test_split(xs: List[X], ys: List[Y], test_pct: float) -> Tuple[List[X], List[X], List[Y], List[Y]]:
    # Generate the indices and split them
    idxs = [i for i in range(len(xs))]
    train_idxs, test_idxs = split_data(idxs, 1 - test_pct)
    return (
        [xs[i] for i in train_idxs],  # x_train
        [xs[i] for i in test_idxs],  # x_test
        [ys[i] for i in train_idxs],  # y_train
        [ys[i] for i in test_idxs]  # y_test
    )

xs = [x for x in range(1000)]  # xs are 1 ... 1000
ys = [2 * x for x in xs]  # each y_i is twice x_i
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.25)

# Check that the proportions are correct
assert len(x_train) == len(y_train) == 750
assert len(x_test) == len(y_test) == 250

# Check that the corresponding data points are paired correctly
assert all(y == 2 * x for x, y in zip(x_train, y_train))
assert all(y == 2 * x for x, y in zip(x_test, y_test))

print("x_train:", x_train)
print("y_train:", y_train)
print("x_test:", x_test)
print("y_test:", y_test)
```

```
x_train: [529, 102, 678, 146, 401, 985, 302, 863, 344, 426, 39, 19, 160, 997, 885, 596, 429, 610, 681, 903, 973, 286, 514, 322, 479, 753
y_train: [1058, 204, 1356, 292, 802, 1970, 604, 1726, 688, 852, 78, 38, 320, 1994, 1770, 1192, 858, 1220, 1362, 1806, 1946, 572, 1028, 6
x_test: [170, 92, 679, 76, 754, 311, 436, 381, 860, 993, 989, 995, 493, 63, 309, 747, 394, 861, 274, 735, 41, 868, 572, 737, 604, 128, 4
y_test: [340, 184, 1358, 152, 1508, 622, 872, 762, 1720, 1986, 1978, 1990, 986, 126, 618, 1494, 788, 1722, 548, 1470, 82, 1736, 1144, 14
```

Double-click (or enter) to edit

✓ 0s completed at 11:51 PM ● ×