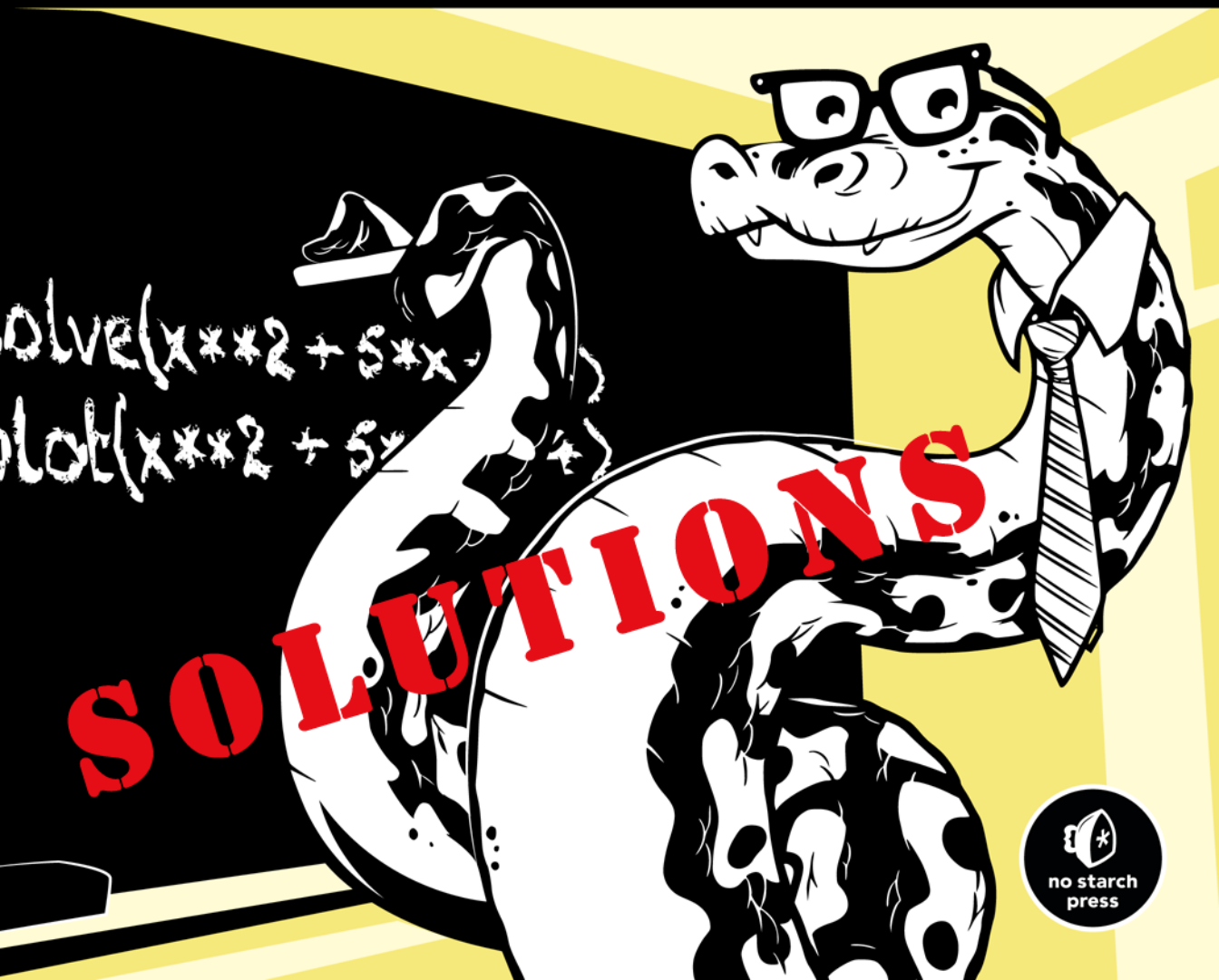
DOING MATH WITH PYTHON

USE PROGRAMMING TO EXPLORE *ALGEBRA*, *STATISTICS*, *CALCULUS*, AND MORE!

AMIT SAHA



no starch press

DOING MATH WITH PYTHON

SOLUTIONS TO PROGRAMMING CHALLENGES

CONTENTS

Chapter 1 Programming Solutions

#1: Even-Odd Vending Machine

To solve this challenge, first convert the input to an integer, then check if it is an even or odd number. If it's odd, print the next nine odd numbers; if it's even, print the next nine even numbers. The following program is one way to achieve that:

```
'''
even_odd_vending.py

Print whether the input is even or odd. If even, print the next 9 even numbers
If odd, print the next 9 odd numbers.
'''
def even_odd_vending(num):

❶     if (num % 2) == 0:
          print('Even')
      else:
          print('Odd')
      count = 1
❷     while count <= 9:
          num += 2
          print(num)
          # increment the count of numbers printed
          count += 1

if __name__ == '__main__':
    try:
        num = float(input('Enter an integer: '))
        if num.is_integer():
            even_odd_vending(int(num))
        else:
            print('Please enter an integer')
    except ValueError:
        print('Please enter a number')
```

The input is converted into a floating point number (instead of an integer) so that we can use the `is_integer()` method to check if the input is an integer. If it is, we use the `int()` function to convert it back to an integer before calling the `even_odd_vending()` function.

The `even_odd_vending()` function accepts an integer as a parameter, uses the modulus operator (`%`) to check whether it is divisible by 2 ❶, and prints `Even` or `Odd` depending on the result.

Whether the number is even or odd, the next number to be printed can be obtained by adding 2 to the previous number. We do this in a `while` loop ❷ that continues until we've printed nine numbers. We use a label, `count`, to keep track of how many numbers we have already printed.

If the user enters an even number when they run the program, it will print `Even` and the next nine even numbers:

```
Enter an integer: 2
Even
4
6
8
10
12
14
16
18
20
```

For an odd number, `Odd` will be printed along with the next nine odd numbers.

#2: Enhanced Multiplication Table Generator

The solution to this challenge is an extension of the multiplication table generator program we wrote earlier in the chapter and is shown here:

```
'''
enhanced_multi_table.py

Multiplication table printer: Enter the number and the number
of multiples to be printed
'''

def multi_table(a, n):
    for i in range(1, n+1):
        print('{0} x {1} = {2}'.format(a, i, a*i))

if __name__ == '__main__':
    try:
        a = float(input('Enter a number: '))
        n = float(input('Enter the number of multiples: '))
        if not n.is_integer() or n < 0:
            print('The number of multiples should be a positive integer')
        else:
            multi_table(a, int(n))
    except ValueError:
        print('You entered an invalid input')
```

Note that we check if the desired number of multiples is an integer using the `is_integer()` method. If the correct inputs have been entered, we call the `multi_table()` function with two parameters: `a` (the number whose multiple we want to print) and `n` (the number of multiples we want to print).

When the program is run, it will ask for the inputs and then print the desired number of multiples. If the user enters an invalid input, it will print an error message and exit.

#3: Enhanced Unit Converter

The solution to this challenge enhances the unit conversion program you wrote earlier so it can convert between kilograms and pounds, and Celsius and Fahrenheit:

```
'''
enhanced_unit_converter.py

Unit converter:

Kilometers and Miles
Kilograms and Pounds
Celsius and Fahrenheit
'''

def print_menu():
    print('1. Kilometers to Miles')
    print('2. Miles to Kilometers')
    print('3. Kilograms to Pounds')
    print('4. Pounds to Kilograms')
    print('5. Celsius to Fahrenheit')
    print('6. Fahrenheit to Celsius')

def km_miles():
    km = float(input('Enter distance in kilometers: '))
    miles = km / 1.609
    print('Distance in miles: {0}'.format(miles))

def miles_km():
    miles = float(input('Enter distance in miles: '))
    km = miles * 1.609
    print('Distance in kilometers: {0}'.format(km))

def kg_pounds():
    kg = float(input('Enter weight in kilograms: '))
    pounds = kg * 2.205
    print('Weight in pounds: {0}'.format(pounds))

def pounds_kg():
    pounds = float(input('Enter weight in pounds: '))
    kg = pounds / 2.205
    print('Weight in kilograms: {0}'.format(kg))

def cel_fahren():
    celsius = float(input('Enter temperature in Celsius: '))
    fahrenheit =  celsius*(9 / 5) + 32
    print('Temperature in fahrenheit: {0}'.format(fahrenheit))

def fahren_cel():
    fahrenheit = float(input('Enter temperature in Fahrenheit: '))
    celsius = (fahrenheit - 32)*(5/9)
    print('Temperature in celsius: {0}'.format(celsius))
```

```
if __name__ == '__main__':
    print_menu()
    choice = input('Which conversion would you like to do? ')

    if choice == '1':
        km_miles()
    if choice == '2':
        miles_km()

    if choice == '3':
        kg_pounds()
    if choice == '4':
        pounds_kg()

    if choice == '5':
        cel_fahren()
    if choice == '6':
        fahren_cel()
```

Four new functions have been added to our earlier program. The `kg_pounds()` function accepts a mass in kilograms as input and returns the corresponding mass in pounds. The reverse conversion is performed by the `pounds_kg()` function. The `cel_fahren()` and `fahren_cel()` functions convert Celsius to Fahrenheit and Fahrenheit to Celsius, respectively.

When the program is run, it asks the user to enter a number from 1 to 6 to choose which conversion they want to perform. Next it asks them to input the relevant quantity, and then it outputs the converted result:

```
1. Kilometers to Miles
2. Miles to Kilometers
3. Kilograms to Pounds
4. Pounds to Kilograms
5. Celsius to Fahrenheit
6. Fahrenheit to Celsius
Which conversion would you like to do? 5
Enter temperature in Celsius: 37
Temperature in Fahrenheit: 98.60000000000001
```

If the user enters a number that is not one of the numbers recognized by the program, it silently exits. You may want to print a helpful message to indicate this in your solution.

#4: Fraction Calculator

For the solution to this challenge, we add functions for the different mathematical operations—`add()`, `subtract()`, `divide()`, and `multiply()`—and depending on the user input, we call the relevant function.

```
'''
fractions_operations.py

Fraction operations
'''
```

```
from fractions import Fraction
def add(a, b):
    print('Result of adding {0} and {1} is {2} '.format(a, b, a+b))

def subtract(a, b):
    print('Result of subtracting {1} from {0} is {2}'.format(a, b, a-b))

def divide(a, b):
    print('Result of dividing {0} by {1} is {2}'.format(a, b, a/b))

def multiply(a, b):
    print('Result of multiplying {0} and {1} is {2}'.format(a, b, a*b))

if __name__ == '__main__':
    try:
        a = Fraction(input('Enter first fraction: '))
        b = Fraction(input('Enter second fraction: '))
        op = input('Operation to perform - Add, Subtract, Divide, Multiply: ')
        if op == 'Add':
            add(a, b)
        if op == 'Subtract':
            subtract(a, b)
        if op == 'Divide':
            divide(a, b)
        if op == 'Multiply':
            multiply(a, b)
    except ValueError:
        print('Invalid fraction entered')
```

When the program is run, it will ask for two fractions and the operation to be carried out. Then it will display the result of that operation. Here is an example with the `Subtract` operation:

```
Enter first fraction: 1/3
Enter second fraction: 2/3
Operation to perform - Add, Subtract, Divide, Multiply: Subtract
Result of subtracting 2/3 from 1/3 is -1/3
```

If the user enters an invalid input, such as 1, the program will print an error message.

#5: Give Exit Power to the User

This challenge was kept open ended to give you an opportunity to try improving the programs so that they continued executing until the user quit them. Here is the fraction calculator program that asks the user whether they want to exit after every calculation:

```
'''
fractions_operations_exit_power.py

Fraction operations: Do not exit until asked to
'''
```

```
from fractions import Fraction
def add(a, b):
    print('Result of adding {0} and {1} is {2} '.format(a, b, a+b))

def subtract(a, b):
    print('Result of subtracting {1} from {0} is {2}'.format(a, b, a-b))

def divide(a, b):
    print('Result of dividing {0} by {1} is {2}'.format(a, b, a/b))

def multiply(a, b):
    print('Result of multiplying {0} and {1} is {2}'.format(a, b, a*b))

if __name__ == '__main__':

    while True:

        try:
            a = Fraction(input('Enter first fraction: '))
            b = Fraction(input('Enter second fraction: '))
            op = input('Operation to perform - Add, Subtract, Divide, Multiply: ')
            if op == 'Add':
                add(a, b)
            if op == 'Subtract':
                subtract(a, b)
            if op == 'Divide':
                divide(a, b)
            if op == 'Multiply':
                multiply(a, b)
        except ValueError:
            print('Invalid fraction entered')
        answer = input('Do you want to exit? (y) for yes ')
        if answer == 'y':
            break
```

Sample output from this program is shown here:

```
Enter first fraction: 1/3
Enter second fraction: 4/6
Operation to perform - Add, Subtract, Divide, Multiply: Divide
Result of dividing 1/3 by 2/3 is 1/2
Do you want to exit? (y) for yes n
Enter first fraction: 2/3
Enter second fraction: 4/5
Operation to perform - Add, Subtract, Divide, Multiply: Multiply
Result of multiplying 2/3 and 4/5 is 8/15
Do you want to exit? (y) for yes y
```

You can find another example, the modified unit conversion program, in the file *enhanced_unit_converter_exit_power.py*, along with other solutions.

Chapter 2 Programming Solutions

#1: How Does the Temperature Vary During the Day?

For this challenge, you needed to search for the weather of a city in Google's search engine and re-create a plot of the day's temperature forecast. Figure 1 shows an example of what you would usually see when you search for "New York weather."

Figure 1: Sample result when "New York weather" is entered in Google's search engine.

Once you have the plot, make two lists: one to store the time of day, and the other to store the corresponding temperature forecast at that time. Then call the `plot()` function to create a graph. The time of day is shown as strings. However, if you attempt to pass in a list of strings to the `plot()` function, your program will fail and give you an error. After all, what sense does it make to state that you want a point to be plotted at ("7 AM", 70)? Instead, assign numbers to each time of day that you want to plot—that is, 1 for 4 AM, 2 for 7 AM, and so on. Then, use the `xticks()` function to change the labels of these numbers to match the time of day they correspond to. Here's the solution to the challenge:

```
'''
nyc_forecast_basic.py

Create a graph showing a city's temperature forecast for the day
'''
```

```
import matplotlib.pyplot as plt

def plot_forecast():

    time_of_day = ['4 AM', '7 AM', '10 AM', '1 PM', '4 PM', '7PM', '10 PM']
    forecast_temp = [71, 70, 74, 80, 82, 81, 76]
❶   time_interval = range(1, len(time_of_day) + 1)

    plt.plot(time_interval, forecast_temp, 'o-')
❷   plt.xticks(time_interval, time_of_day)
    plt.show()

if __name__ == '__main__':
    plot_forecast()
```

We create the numbers used to represent the different times of day at ❶. Then we use the `xticks()` function at ❷ to remap the numbers back to the strings, which makes the time (7 AM, etc.) show up in the graph instead of the numbers. The first argument to the function is the list of numbers, and the second is the list of labels you want to assign to each number.

When you run the program, you will see a graph showing the day's temperature forecast, which will closely match what you saw in the search engine result.

#2: Exploring a Quadratic Function Visually

To solve this challenge, first use the `range()` function to generate 10 integers between –100 and 100 in a list ❶. This will generate the integers –100, –80, . . . 80. Then, calculate the value of the function $x^2 + 2x + 1$ at each of these integers and store each value in `y_values`. Finally, use the `plot()` function to plot the numbers in `x_values` and `y_values`.

```
'''
quad_function_plot.py

Plot a quadratic function
'''

import matplotlib.pyplot as plt

def draw_graph(x, y):
    plt.plot(x, y)
    plt.show()

if __name__ == '__main__':
    # assume values of x
❶   x_values = range(-100, 100, 20)
    y_values = []
    for x in x_values:
        # calculate the value of the quadratic
        # function
        y_values.append(x**2 + 2*x + 1)
    draw_graph(x_values, y_values)
```

When you run the program, the graph of the function will look something like Figure 2.

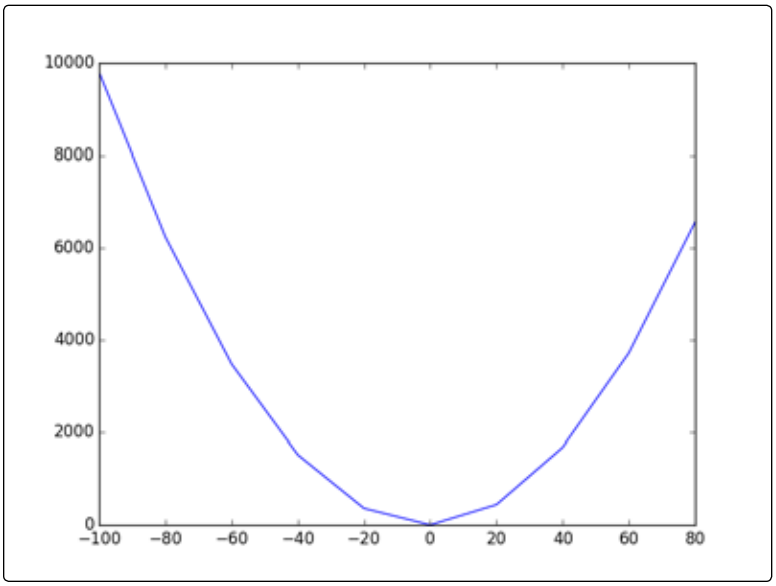


Figure 2: Graph of the quadratic function $x^2 + 2x + 1$. Increasing the number of points between –10 and 10 will make the graph appear smoother.

Because this is a quadratic function, the variation of the function with respect to the variable *x* is nonlinear.

#3: Enhanced Projectile Trajectory Comparison Program

Here's the solution to this challenge:

```
"""
projectile_comparison_gen.py

Compare the projectile motion of a body thrown with various combinations of initial
velocity and angle of projection
"""

import matplotlib.pyplot as plt
import math

g = 9.8

def draw_graph(x, y):
    plt.plot(x, y)
    plt.xlabel('x-coordinate')
    plt.ylabel('y-coordinate')
    plt.title('Projectile motion at different initial velocities and angles')
```

```
def frange(start, final, interval):

    numbers = []
    while start < final:
        numbers.append(start)
        start = start + interval

    return numbers

def draw_trajectory(u, theta, t_flight):
    # List of x and y coordinates
    x = []
    y = []
    intervals = frange(0, t_flight, 0.001)
    for t in intervals:
        x.append(u*math.cos(theta)*t)
        y.append(u*math.sin(theta)*t - 0.5*g*t*t)

    # Create the graph
    draw_graph(x, y)

if __name__ == '__main__':

    num_trajectories = int(input('How many trajectories? '))

    velocities = []
    angles = []
    for i in range(1, num_trajectories+1):
        v = input('Enter the initial velocity for trajectory {0} (m/s): '.format(i))
        theta = input('Enter the angle of projection for trajectory {0} (degrees): '.format(i))
        velocities.append(float(v))
        angles.append(math.radians(float(theta)))

    for i in range(num_trajectories):
        # Calculate time of flight, maximum horizontal distance and
        # maximum vertical distance
        t_flight = 2*velocities[i]*math.sin(angles[i])/g
        S_x = velocities[i]*math.cos(angles[i])*t_flight
        S_y = velocities[i]*math.sin(angles[i])*(t_flight/2) - (1/2)*g*(t_flight/2)**2
❶       print('Initial velocity: {0} Angle of Projection: {1}'.format(velocities[i],
              math.degrees(angles[i])))
        print('T: {0} S_x: {1} S_y: {2}'.format(t_flight, S_x, S_y))
        print()
❷       draw_trajectory(velocities[i], angles[i], t_flight)


    # Add a legend and show the graph
    legends = []
    for i in range(0, num_trajectories):
        legends.append('{0} - {1}'.format(velocities[i], math.degrees(angles[i])))
❸   plt.legend(legends)
    plt.show()
```

When executed, the program first asks for the number of trajectories that the user wants to compare. Then it asks for the initial velocity and angle of projection for each trajectory. For each of these trajectories, it prints the time of flight, range, and maximum vertical height reached ❶ and calls the `draw_trajectory()` function ❷, which creates a plot for each trajectory.

To identify which trajectory corresponds to which initial velocity and angle, we add a legend to the graph ❸. Finally, we call the `show()` function to show the graph.

Here's a sample run:

```
How many trajectories? 3
Enter the initial velocity for trajectory 1 (m/s): 25
Enter the angle of projection for trajectory 1 (degrees): 60
Enter the initial velocity for trajectory 2 (m/s): 50
Enter the angle of projection for trajectory 2 (degrees): 50
Enter the initial velocity for trajectory 3 (m/s): 30
Enter the angle of projection for trajectory 3 (degrees): 60
Initial velocity: 25.0 Angle of Projection: 59.99999999999999
T: 4.41849695808387 S_x: 55.23121197604839 S_y: 23.91581632653061

Initial velocity: 50.0 Angle of Projection: 50.0
T: 7.8167800318263065 S_x: 251.22646760515516 S_y: 74.85001133079913

Initial velocity: 30.0 Angle of Projection: 59.99999999999999
T: 5.302196349700644 S_x: 79.53294524550968 S_y: 34.438775510204074
```

Figure 3 shows the graph created by the program. You can see the three different trajectories with a legend showing which trajectory belongs to which combination of initial velocity and angle of projection.

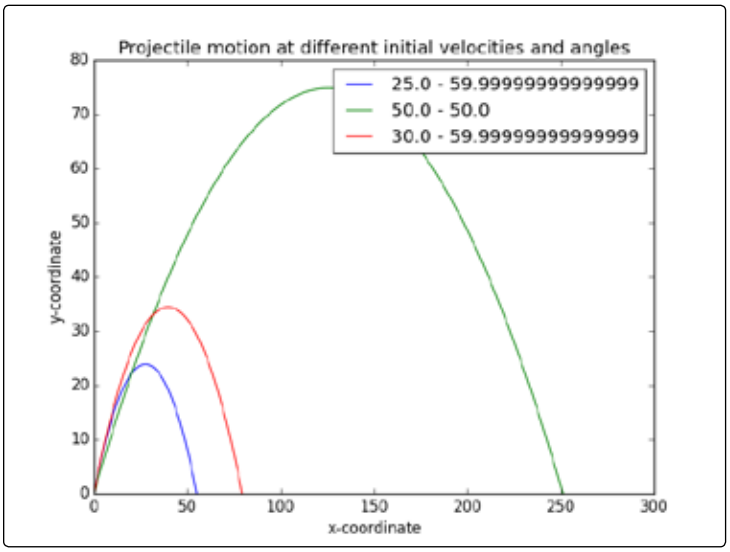


Figure 3: Trajectories of three bodies in projectile motion launched with a different combinations of initial velocity and angle of projection

#4: Visualizing Your Expenses

The solution for this challenge requires you to ask the user for a number of categories and what each of those categories is. Then you ask for the expenditure in that category for that week and create a bar chart using the `barh()` function. Here's the complete program:

```
'''
expenditures_barchart.py

Visualizing weekly expenditure using a bar chart
'''

import matplotlib.pyplot as plt

def create_bar_chart(data, labels):
    # number of bars
    num_bars = len(data)
    # This list is the point on the y-axis where each
    # bar is centered. Here it will be [1, 2, 3..]
    positions = range(1, num_bars+1)
❶   plt.barh(positions, data, align='center')
    # Set the label of each bar
    plt.yticks(positions, labels)
    plt.xlabel('Amount')
    plt.ylabel('Categories')
    plt.title('Weekly expenditures')
    # Turns on the grid which may assist in visual estimation
    plt.grid()
❷   plt.show()

if __name__ == '__main__':
    n = int(input('Enter the number of categories: '))
    labels = []
    expenditures = []
    for i in range(n):
        category = input('Enter category: ')
        expenditure = float(input('Expenditure: '))

❸       labels.append(category)
❹       expenditures.append(expenditure)
❺   create_bar_chart(expenditures, labels)
```

The input categories are stored in the list `labels` ❸, which will also be used for the labels in the bar chart. The corresponding expenditure for each category is stored in the list `expenditures` ❹. We then call the `create_bar_chart()` function ❺ with the expenditures as the first argument and labels as the second argument. This function then calls the `barh()` function ❶ with the appropriate arguments, setting the axes labels and other details, and finally calls the `show()` function ❷, which displays the bar chart showing the expenditures for each category.

Here's a sample execution of the program:

```
Enter the number of categories: 4
Enter category: Food
Expenditure: 70
Enter category: Transportation
Expenditure: 35
Enter category: Entertainment
Expenditure: 30
Enter category: Phone/Internet
Expenditure: 30
```

Figure 4 shows the bar chart created for this input.

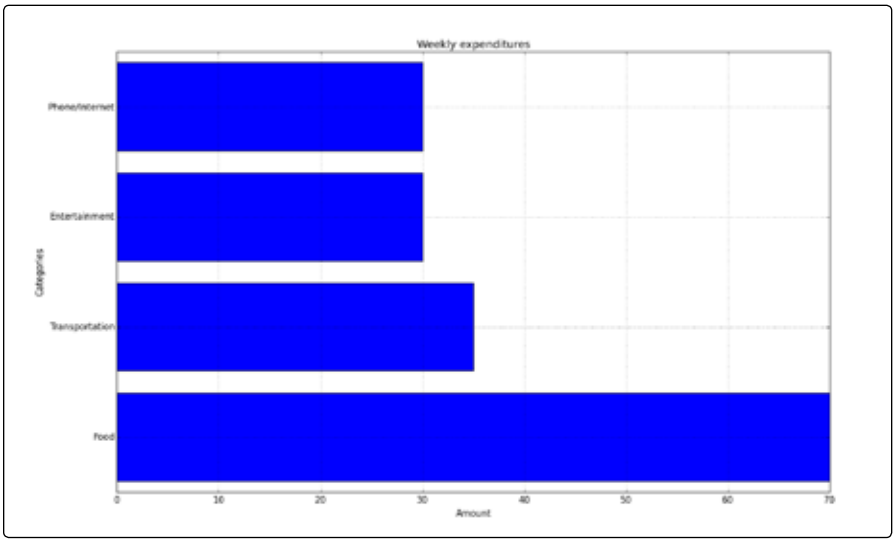


Figure 4: A bar chart showing the weekly expenditure in each category

#5: Exploring the Relationship Between the Fibonacci Sequence and the Golden Ratio

For the solution to this challenge, we'll calculate the first 100 numbers of the Fibonacci sequence using the `fibo()` function mentioned in the challenge description. Then we will calculate the consecutive difference between these numbers, store those differences in a list, and use the `plot()` function to create a graph that shows these differences. Here's the solution:

```
'''
fibonacci_goldenration.py

Relationship between Fibonacci sequence and golden ratio
'''
```

```
import matplotlib.pyplot as plt

def fibo(n):
    if n == 1:
        return [1]
    if n == 2:
        return [1, 1]
    # n > 2
    a = 1
    b = 1
    # first two members of the series
    series = [a, b]
    for i in range(n):
        c = a + b
        series.append(c)
        a = b
        b = c

    return series

def plot_ratio(series):
    ratios = []
❶   for i in range(len(series)-1):
        ratios.append(series[i+1]/series[i])
    plt.plot(ratios)
    plt.title('Ratio between Fibonacci numbers & golden ratio')
    plt.ylabel('Ratio')
    plt.xlabel('No.')
    plt.show()

if __name__ == '__main__':
    # Number of fibonacci numbers
    num = 100
    series = fibo(num)
    plot_ratio(series)
```

The `for` loop at ❶ is the key step in the `plot_ratio()` function. We loop through the numbers of the Fibonacci sequence and store the difference between the second term and the first term, the third term and the second term, and so on, in the list `ratios`. We then call the `plot()` function, add a title and axes labels, and finally call the `show()` function to show the graph.

Chapter 3 Programming Solutions

#1: Better Correlation Coefficient–Finding Program

We can make a simple change to the `find_corr_x_y()` function so that it checks the length of the two lists being passed to it. If they are not equal in length, we return `None`. Here's the function, along with some other changes:

```
'''
linear_correlation_enhanced.py

Linear correlation program

'''
def find_corr_x_y(x,y):

❶     if len(x) != len(y):
          print('The two sets of numbers are of unequal size')
          return None

      n = len(x)

      # find the sum of the products
❷     prod = [xi*yi for xi, yi in zip(x, y)]
      sum_prod_x_y = sum(prod)

      # sum of the numbers in x
      sum_x = sum(x)
      # sum of the numbers in y
      sum_y = sum(y)

      # square of the sum of the numbers in x
      squared_sum_x = sum_x**2
      # square of the sum of the numbers in y
      squared_sum_y = sum_y**2

      # find the squares of numbers in x and the
      # sum of the squares
❸     x_square = [xi**2 for xi in x]
      x_square_sum = sum(x_square)

      # find the squares of numbers in y and the
      # sum of the squares
      y_square = [yi**2 for yi in y]
      y_square_sum = sum(y_square)

      # numerator
      numerator = n*sum_prod_x_y - sum_x*sum_y
      denominator_term1 = n*x_square_sum - squared_sum_x
      denominator_term2 = n*y_square_sum - squared_sum_y
      denominator = (denominator_term1*denominator_term2)**0.5
```

```
    correlation = numerator/denominator

    return correlation
```

Besides adding the check for the length of the two lists right at the beginning ❶, you can see that I've also used list comprehensions (as explained in Appendix B) to make some parts of the program more compact (for example, at ❷ and ❸).

Now, if you call this function with two sets of data, each containing the same number of items, it will return the correlation coefficient as it did before. If the two sets of data have an unequal number of items, it returns `None`. When you use this function in your programs, you have to check the return value and take appropriate action depending on the result.

```
corr = find_corr_x_y(x,y)
if not corr:
    print('Correlation correlation could not be calculated')
else:
    print('The correlation coefficient between x and y is {0}'.format(corr))
```

For example, if the return value is `None`, we print a message saying that that the correlation coefficient could not be calculated.

#2: Statistics Calculator

The program that reads the numbers from the file *mydata.txt* and calculates the various statistical measures is as follows:

```
'''
statistics_calculator.py

Read numbers from a file, calculate and print statistical measures:
mean, median, mode, variance, standard deviation
'''


❶ from stats import mean, median, mode, variance_sd

  def read_data(filename):
      numbers = []
      with open(filename) as f:
          for line in f:
              numbers.append(float(line))

      return numbers

  if __name__=='__main__':
❷     data = read_data('mydata.txt')
      m = mean(data)
      median = median(data)
      mode = mode(data)
      variance, sd = variance_sd(data)
```

```
    print('Mean: {0:.5f}'.format(m))
    print('Median: {0:.5f}'.format(median))
    print('Mode: {0:.5f}'.format(mode))
    print('Variance: {0:.5f}'.format(variance))
    print('Standard deviation: {0:.5f}'.format(sd))
```

I've created a module, *stats.py* (in the same directory as *statistics_calculator.py*) that contains the functions for calculating the mean, median, mode, variance, and standard deviation. These functions are imported at ❶. These functions are the same functions we wrote in the chapter except they are named slightly differently. The other difference is that the function `variance_sd()` returns both the variance and standard deviation as tuples. Moving these functions into a separate module means that we can avoid defining the functions every time we want to use them.

We call the `read_data()` function, which reads the numbers from the file and returns them in a list, `data` ❷. Once we have the list of numbers, we can call the different functions to calculate the corresponding statistical measures and finally print them.

#3: Experiment with Other CSV Data

The CSV file, *USA_SP_POP_TOTL.csv*, contains two columns of data: a date and the total population from 1960-12-31 to 2012-12-31. We will read the population and the year from the file, calculate the statistical measures, and create the graphs:

```
'''
us_population_stats.py

Read the US population data from a CSV file, calculate the growth in
population in consecutive years, and compute various statistical measures

Also creates two graphs - one showing the total population over the years and
the other showing the change between consecutive years
'''

import matplotlib.pyplot as plt
import csv
from stats import mean, median, variance_sd

def read_csv(filename):

    years = []
    population = []

    with open(filename) as f:
        reader = csv.reader(f)
        next(reader)

        summer = []
        highest_correlated = []
```

```
            for row in reader:
                # Extract only the year from
                # date
❶               year = row[0].split('-')[0]
                years.append(year)
                population.append(float(row[1]))
        # Reverse the lists because the original data lists the
        # most recent years first
        population.reverse()
        years.reverse()

        return population, years

    def plot_population(population, years):
❷       plt.figure(1)
        xaxis_positions = range(0, len(years))
        plt.plot(population, 'r-')
        plt.title('Total population in US')
        plt.xlabel('Year')
        plt.ylabel('Population')
❸       plt.xticks(xaxis_positions, years, rotation=45)

    def calculate_stats(population):

        # find the growth in population in consecutive years
        growth = []
        for i in range(0, len(population)-1):
            growth.append(population[i+1] - population[i])
        print('Mean growth: {0:.5f}'.format(mean(growth)))
        print('Median growth: {0:.5f}'.format(median(growth)))
        print('Variance/Sd growth: {0:.5f}, {1:.5f}'.format(*variance_sd(growth)))
        return growth

    def plot_population_diff(growth, years):

        xaxis_positions = range(0, len(years)-1)
❹       xaxis_labels = ['{0}-{1}'.format(years[i], years[i+1])
                        for i in range(len(years)-1)]
        plt.figure(2)
        plt.plot(growth, 'r-')
        plt.title('Population Growth in consecutive years')
        plt.ylabel('Population Growth')
        plt.xticks(xaxis_positions, xaxis_labels, rotation=45)

    if __name__ == '__main__':
        population, years = read_csv('USA_SP_POP_TOTL.csv')
        plot_population(population, years)
        growth = calculate_stats(population)
        plot_population_diff(growth, years)
        plt.show()
```

The `read_csv()` function reads the data from the file to create two lists: `years` and `population`, which contain the year and the corresponding population of each year, respectively. As described earlier, the file contains a date, but we want to extract just the year (so that the text fits in our graphs), which is what we do at ❶. The `split()` method returns a list of words from a string and splits the string at the specified delimiter. For example:

```
>>> d = '2012-09-10'
>>> d.split('-')
['2012', '09', '10']
```

Thus, if we wanted the year 2012, we would grab the first item in the returned list:

```
>>> words = d.split('-')
>>> words[0]
'2012'
```

Since we read the file from the beginning and the latest dates are listed first, we reverse both lists before returning them with the `reverse()` method. The `reverse()` method reverses a list in place. For example, say that `list = [1, 2, 3]`. Calling `list.reverse()` would change `list` to `[3, 2, 1]`. Once we have the two lists of the years and the population figures, we create the first graph that shows the population over the years in the function `plot_population()`. Since we plan to create two separate graphs, we create a figure explicitly by calling the `figure()` function and assigning it a number, `1`, at ❷. We plot the population and add the years as the *x*-axis labels using the `xticks()` function ❸. The `rotation` keyword argument lets us display the labels at an angle so that the consecutive years do not overlap. Here, we specify that the label be at a 45-degree angle.

In the `calculate_stats()` function, we calculate and print the statistical measures of the population growth using the functions that are now part of the `stats` module. This function also returns the list `growth`, which contains the growth in population over consecutive years.

The `plot_population_diff()` function creates a second figure and plots the population growth over the years. We use list comprehension to create a list of labels for the *x*-axis of the form 1960–1961, 1961–1962 and so on to indicate the years ❹. We also specify that we want the labels to be at an angle of 45 degrees using of the `rotation` keyword argument.

When you run the program, you should see two graphs side by side as shown in Figure 5.

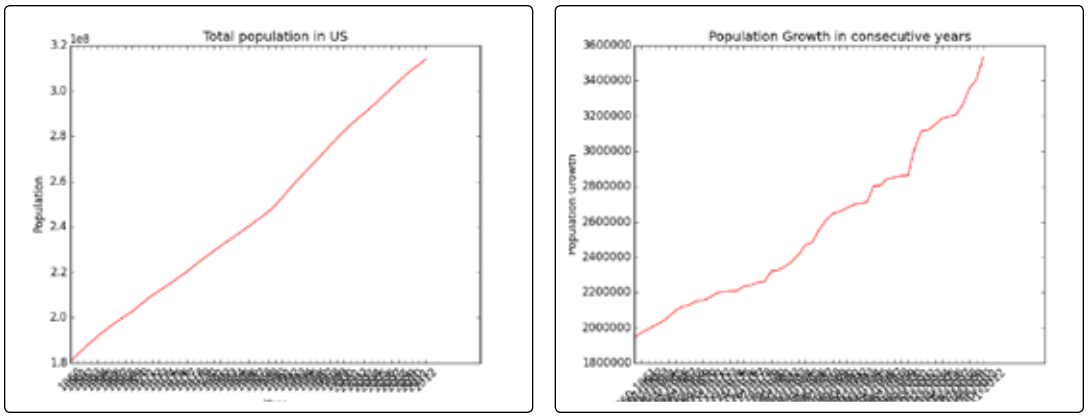


Figure 5: Two population graphs

One window will be titled *Figure 1* and the other *Figure 2*, and the statistical measures will be printed as shown here:

```
Mean growth: 2562366.15385
Median growth: 2476370.00000
Variance/Sd growth: 188985554755.28406, 434724.68846
```

You will need to maximize the graph windows to see the *x*-axis labels clearly.

#4: Finding the Percentile

The program for calculating the score at a certain percentile using the algorithm discussed in the challenge description is as follows:

```
'''
percentile_score.py

Calculate the number from a list of numbers that corresponds
to a specific percentile

This implements the method described at
http://web.stanford.edu/class/archive/anthsci/anthsci192/anthsci192.1064/handouts/
calculating%20percentiles.pdf
'''

def find_percentile_score(data, percentile):
    if percentile < 0 or percentile > 100:
        return None
❶   data.sort()
    if percentile == 0:
        return data[0]
    if percentile == 100:
        return data[-1]
```

```
        n = len(data)
❷       i = ((n*percentile)/100) + 0.5

        if i.is_integer():
            real_idx = int(i-1)
❸           return data[real_idx]
        else:
❹           k = int(i)
❺           f = i - k
            real_idx_1 = k - 1
            real_idx_2 = k
❻           return (1-f)*data[real_idx_1] + f*data[real_idx_2]

    def read_data(filename):
        numbers = []
        with open(filename) as f:
            for line in f:
                numbers.append(float(line))
        return numbers

    if __name__ == '__main__':
        percentile = float(input('Enter the percentile score you want to calculate: '))
        data = read_data('marks.txt')
        percentile_score = find_percentile_score(data, percentile)
        if percentile_score:
            print('The score at {0} percentile: {1}'.format(percentile, percentile_score))
        else:
            print('Could not find the score corresponding to {0} percentile'.format(percentile))
```

The `find_percentile_score()` function implements the algorithm. It first sorts the list of numbers passed to it using the `sort()` method ❶. Then, it checks for two percentile scores: 0 and 100. If the specified percentile score to be found is 0, it returns the first member of the sorted list and if the specified percentile score is 100, it returns the last element. At ❷, we calculate the value of i corresponding to step 2 of the algorithm. If it's an integer, we return the ith element from the list of numbers ❸. Since a list in Python starts at 0, the element that we want will correspond to the element at $(i - 1)$ in the list. If the value is not an integer, we extract the integer and the fractional parts of the number at ❹ and ❺, and refer to them with the labels `k` and `f`, respectively. We then return the number `(1-f)*data[k] + f*data[k+1]` ❻, which corresponds to the specified percentile score.

You will find a *marks.txt* file in the source directory that contains a list of scores. If you run this program, when you enter a specified percentile, it will print the corresponding marks:

```
Enter the percentile score you want to calculate: 88
The score at 88.0 percentile: 19.5
```

However, this algorithm doesn't work for percentiles that are greater than 98 with this set of data. An alternative algorithm is implemented

by Microsoft Excel, which you can read about at *https://en.wikipedia.org/wiki/Percentile#Microsoft_Excel_method*. The following code implements this method:

```
'''
percentile_score_microsoft_excel.py

Calculate the number from a list of numbers that corresponds
to a specific percentile

This implements the "Microsoft Excel Method":
https://en.wikipedia.org/wiki/Percentile#Microsoft_Excel_method

'''

def find_percentile_score(data, percentile):
    if percentile < 0 or percentile > 100:
        return None
    data.sort()
    if percentile == 0:
        return data[0]
    if percentile == 100:
        return data[-1]
    n = len(data)
❶   rank = (percentile/100)*(n-1) + 1
❷   k = int(rank)
❸   d = rank - k

    real_idx_1 = k-1
    real_idx_2 = k

❹   return data[real_idx_1] + d*(data[real_idx_2]-data[real_idx_1])

def read_data(filename):
    numbers = []
    with open(filename) as f:
        for line in f:
            numbers.append(float(line))
    return numbers

if __name__ == '__main__':
    percentile = float(input('Enter the percentile score you want to calculate: '))
    data = read_data('marks.txt')
    percentile_score = find_percentile_score(data, percentile)
    if percentile_score:
        print('The score at {0} percentile: {1}'.format(percentile, percentile_score))
    else:
        print('Could not find the score corresponding to {0} percentile'.format(percentile))
```

The key step in this program is at ❶, where we calculate the *rank*. Then we extract the integer and the fractional part of the rank at ❷ and ❸ and return the number corresponding to the specified percentile at ❹.

Here is a sample run of the program:

```
Enter the percentile score you want to calculate: 88
The score at 88.0 percentile: 19.5
```

As expected, both programs return the same 88 percentile score. Let's try finding the score corresponding to the percentile 99.6:

```
Enter the percentile score you want to calculate: 99.6
The score at 99.6 percentile: 20.0
```

#5: Creating a Grouped Frequency Table

The following program creates a grouped frequency table representing the data in the *marks.txt* file:

```
'''
grouped_frequency.py

Create a grouped frequency table from a list of numbers
'''

def create_classes(numbers, n):
    low = min(numbers)
    high = max(numbers)

    # width of each class
    width = (high - low)/n
    classes = []
    a = low
    b = low + width
    classes = []
    while a < (high-width):
        classes.append((a, b))
        a = b
        b = a + width
    # The last class may be of size
    # less than width
    classes.append((a, high+1))
    return classes

def classify(numbers, classes):
    # Create a list with the same number of elements
    # as the number of classes
    count = [0]*len(classes)
    for n in numbers:
        for index, c in enumerate(classes):
            if n >= c[0] and n < c[1]:
                count[index] += 1
                break
    return count
```

```
def read_data(filename):
    numbers = []
    with open(filename) as f:
        for line in f:
            numbers.append(float(line))
    return numbers

if __name__ == '__main__':

    num_classes = int(input('Enter the number of classes: '))
    numbers = read_data('marks.txt')

    classes = create_classes(numbers, num_classes)
    count = classify(numbers, classes)
    for c, cnt in zip(classes, count):
        print('{0:.2f} - {1:.2f} \t {2}'.format(c[0], c[1], cnt))
```

When the program runs, it asks the user to input the desired number of classes and reads the data from the `marks.txt` file into a list, `numbers`. We then call the function `create_classes()`, which returns the specified number of classes referred to by the label `classes` and calls the `classify()` function, which returns a list containing the count of the numbers in each class. We finally print the table by going over the two lists using the `zip()` function.

Here's a sample execution of the program:

```
Enter the number of classes: 4
10.50 - 12.88      4
12.88 - 15.25      4
15.25 - 17.62      7
17.62 - 21.00      10
```

You can see that the size of each class is approximately 2.37.

Chapter 4 Programming Solutions

#1: Factor Finder

The following code implements the factor finder program:

```
'''
factorizer.py

Factor an input expression
'''

from sympy import factor, sympify, SympifyError

def factorize(expr):
    return factor(expr)

if __name__ == '__main__':
❶    expr = input('Enter an expression to factorize: ')
    try:
        expr_obj = sympify(expr)
    except SympifyError:
        print('Invalid expression entered as input')
    else:
        print(factorize(expr_obj))
```

We ask the user to input an expression to factorize at ❶, convert it to a SymPy object using `sympify()` function, and then call the `factorize()` function, in which SymPy's `factor()` function is used find the factors. Here is a sample run of the program:

```
Enter an expression to factorize: x**2 + 5*x + 6
(x + 2)*(x + 3)
```

If an invalid expression is input, an error message will be printed.

#2: Graphical Equation Solver

The following program takes in two equations expressed in the form `ax + by - k = 0` as input, attempts to find the solution to the two equations, and creates a graph showing the two lines:

```
'''
graphical_eq_solve.py

Graphical equation solver
'''

from sympy import Symbol, sympify, solve, SympifyError
from sympy.plotting import plot
```

```
def solve_plot_equations(eq1, eq2, x, y):
    # Solve
    solution = solve((eq1, eq2), dict=True)
    if solution:
        print('x: {0} y: {1}'.format(solution[0][x], solution[0][y]))
    else:
        print('No solution found')
    # Plot
    eq1_y = solve(eq1,'y')[0]
    eq2_y = solve(eq2, 'y')[0]
    plot(eq1_y, eq2_y, legend=True)

if __name__=='__main__':

    eq1 = input('Enter your first equation : ')
    eq2 = input('Enter your second equation: ')

    try:
        eq1 = sympify(eq1)
        eq2 = sympify(eq2)
    except SympifyError:
        print('Invalid input')
    else:
        x = Symbol('x')
        y = Symbol('y')
        # check if the expressions consist of only two variables
❶       eq1_symbols = eq1.atoms(Symbol)
❷       eq2_symbols = eq2.atoms(Symbol)

        if len(eq1_symbols)> 2 or len(eq2_symbols) > 2:
            print('The equations must have only two variables - x and y')
        elif x not in eq1_symbols or y not in eq1_symbols:
            print('First equation must have only x and y variables')
        elif x not in eq2_symbols or y not in eq2_symbols:
            print('Second equation must have only x and y variables')
        else:
            solve_plot_equations(eq1, eq2, x, y)
```

After we validate the input equations, we find the symbols in each equation using the `atoms()` method ❶ and ❷. When we call the `atoms()` method using the `Symbol` class as a parameter, the method returns a list of the symbols in that expression. Then, we check that the input expressions contain only *x* and *y* as the symbols. If that's the case, we call the `solve_plot_equations()` function. There, we call the `solve()` function to attempt to find the solution to the two equations. If no solution is found, a message stating so is printed. Then, we express the two equations in terms of *x*, and call the `plot()` function to create the graph.

Here is an example run of the program:

```
Enter your first equation : 3*x + 2*y - 3
Enter your second equation: 2*x + 3*y - 2
x: 1 y: 0
```

Figure 6 shows a graph of the two input equations that also demonstrates the solution of the two equations (1, 0).

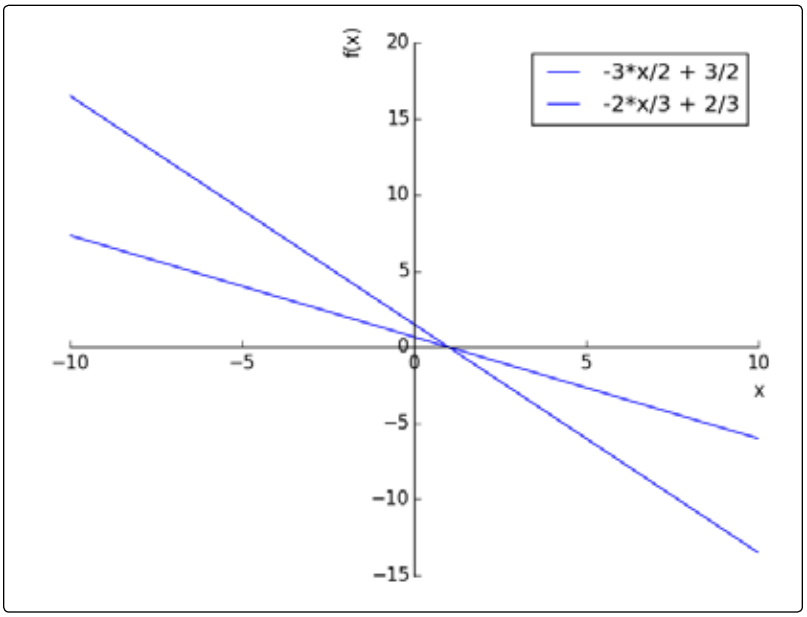


Figure 6: Graph showing the two input equations

#3: Summing a Series

The following program shows how to use the `summation()` function to implement this solution:

```
'''
series_summation.py

Sum an arbitrary series
'''

from sympy import summation, sympify, Symbol, pprint
def find_sum(n_term, num_terms):
    n = Symbol('n')
    s = summation(n_term, (n, 1, num_terms))
    pprint(s)


if __name__ == '__main__':
    n_term = sympify(input('Enter the nth term: '))
    num_terms = int(input('Enter the number of terms: '))

    find_sum(n_term, num_terms)
```

We ask the user to input the *n*th term of the series, and the number of terms they want to sum. We then call the `find_sum()` function, which calls SymPy's `summation()` function, passing the *n*th term as the first argument. The second argument is a tuple consisting of the symbol, `1`, and the number of terms up to which we want to find the sum. We then pretty print the sum using the `pprint()` function. Here is a sample run:

```
Enter the nth term: a+(n-1)*d
Enter the number of terms: 3
3·a + 3·d
```

#4: Solving Single-Variable Inequalities

The following program implements a generic inequality solver:

```
'''
isolve.py

Single variable inequality solver
'''

from sympy import Symbol, sympify, SympifyError
from sympy import solve_poly_inequality, solve_rational_inequalities
from sympy import solve_univariate_inequality, Poly
from sympy.core.relational import Relational, Equality

def isolve(ineq_obj):
    x = Symbol('x')

    expr = ineq_obj.lhs
    rel = ineq_obj.rel_op

    if expr.is_polynomial():
        p = Poly(expr, x)
        return solve_poly_inequality(p, rel)
    elif expr.is_rational_function():
        p1, p2 = expr.as_numer_denom()
        num  = Poly(p1)
        denom = Poly(p2)
        return solve_rational_inequalities([[((num, denom), rel)]])
    else:
        return solve_univariate_inequality(ineq_obj , x, relational=False)

if __name__ == '__main__':
    ineq = input('Enter the inequality to solve: ')
    try:
        ineq_obj = sympify(ineq)
    except SympifyError:
        print('Invalid inequality')
```

```
    else:
        # We check if the input expression is an inequality here
❶       if isinstance(ineq_obj, Relational) and not isinstance(ineq_obj, Equality):
            print(isolve(ineq_obj))
        else:
            print('Invalid inequality')
```

The program asks the user to input an inequality and then converts it into a SymPy object using the `sympify()` function. We then check whether the expression is an inequality and not any arbitrary expression. We do this by using the `isinstance()` Python function ❶ to check whether the object returned by the `sympify()` function is a `Relational` object and whether it is not an `Equality` object. This check is needed because only a `Relational` object has a `rel_op` attribute that refers to the relational operator (`>`, `>=`, `<` and `<=`) that will be used later in the function `isolve()`. Once we have checked whether the input is an inequality, we call the `isolve()` function.

In the `isolve()` function, we use the functions `is_polynomial()` and `is_rational_function()` to check if the inequality is a polynomial or a rational function and call the appropriate inequality-solver function. If it is neither, we call the `solve_univariate_inequality()` function.

Here is a sample execution for a polynomial inequality:

```
Enter the inequality to solve: -x**2 + 4 < 0
[(-oo, -2), (2, oo)]
```

Here is a sample run for a rational inequality:

```
Enter the inequality to solve: ((x-1)/(x+2)) > 0
(-oo, -2) U (1, oo)
```

And here is a sample run for an expression that is neither a polynomial inequality nor a rational inequality:

```
Enter the inequality to solve: sin(x) - 0.6 > 0
(0.643501108793284, 2.49809154479651)
```

Chapter 5 Programming Solutions

#1: Using Venn Diagrams to Visualize Relationships Between Sets

Here's the solution to this challenge:

```
"""
venn_sports.py

Is football the favorite sport in my class too?
Let's find out using a Venn diagram
"""

from sympy import FiniteSet
from matplotlib_venn import venn2
import matplotlib.pyplot as plt
import csv

def read_csv(filename):

    football = []
    others = []

    with open(filename) as f:
        reader = csv.reader(f)
        next(reader)
        for row in reader:
            if row[1] == '1':
                football.append(row[0])
            if row[2] == '1':
                others.append(row[0])

    return football, others


def draw_venn(f, o):
    venn2(subsets=(f, o), set_labels=('Football', 'Others'))
    plt.show()

if __name__ == '__main__':
    football, others = read_csv('sports.csv')
❶   f = FiniteSet(*football)
❷   o = FiniteSet(*others)
    draw_venn(f, o)
```

The key step is creating the two sets—the set of students who like football and the set of students who like another sport—by reading the data from *sports.csv*, which is created by compiling the results from the survey. This file is read by the `read_csv()` function, which creates two lists made up of student IDs (`football` and `others`), and returns them.

We then create two sets corresponding to each category to represent the two groups of students ❶ and ❷ and call the `draw_venn()` function, which draws the Venn diagram.

#2: Law of Large Numbers

For this challenge, we roll a die a specified number of times to see how the average value gets closer to the theoretical expected value of 3.5. Here's the program:

```
'''
law_ln.py

Verify the law of large numbers using a six-sided die roll as an example
'''
import random

def roll(num_trials):
    rolls = []
    for t in range(num_trials):
        rolls.append(random.randint(1, 6))
    return sum(rolls)/num_trials

if __name__ == '__main__':
    expected_value = 3.5
    print('Expected value: {0}'.format(expected_value))
    for trial in [100, 1000, 10000, 100000, 500000]:
        avg = roll(trial)
        print('Trials: {0} Trial average {1}'.format(trial, avg))
```

The `roll()` function simulates rolling a six-sided die a certain number of times as specified by the argument `num_trials` and returns the average value rolled. Call this function with the number of rolls set to 100, 1000, 10000, 100000, and 500000 to see how the average value gets closer to the expected value of 3.5:

```
Expected value: 3.5
Trials: 100 Trial average 3.49
Trials: 1000 Trial average 3.519
Trials: 10000 Trial average 3.4966
Trials: 100000 Trial average 3.50036
Trials: 500000 Trial average 3.501474
```

#3: How Many Tosses Before You Run Out of Money?

The solution to this challenge is as follows:

```
'''
game_tosses.py

A player wins 1$ for every head and loses 1.5$ for every tail.
```

```
    The game is over when the player's balance reaches 0$
    '''
    import random

    def play(start_amount):

        win_amount = 1
        loss_amount = 1.5

        cur_amount = start_amount
        tosses = 0

❶       while cur_amount > 0:
            tosses += 1
❷           toss = random.randint(0, 1)
            if toss == 0:
                cur_amount += win_amount
                print('Heads! Current amount: {0}'.format(cur_amount))
            else:
                cur_amount -= loss_amount
                print('Tails! Current amount: {0}'.format(cur_amount))
        print('Game over :( Current amount: {0}. Coin tosses: {1}'.
            format(cur_amount, tosses))

    if __name__ == '__main__':
        start_amount = float(input('Enter your starting amount: '))
        play(start_amount)
```

When the program runs, it asks the user to enter a starting point and then calls the `play()` function to start the game.

Then, using a `while` loop, it continues tossing a coin until the current amount (`cur_amount`) is less than or equal to 0 ❶. The coin toss is simulated using the `random.randint()` function so that either a 0 or 1 is returned ❷. For every heads (that is, the number 0), we add 1 to the current amount, and for every tails (the number 1), we deduct 1.5. We keep a count of the total number of coin tosses via the label `tosses`.

#4: Shuffling a Deck of Cards

Here's the 52 card shuffle program:

```
'''
shuffle_enhanced.py

Shuffle a deck of 52 cards
'''
import random

class Card:
    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
```

```
def initialize_deck():
    suits = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    ranks = ['Ace', '2', '3','4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']
    cards = []
    for suit in suits:
        for rank in ranks:
❶       card = Card(suit, rank)
            cards.append(card)
    return cards

def shuffle_and_print(cards):
❷   random.shuffle(cards)
❸   for card in cards:
        print('{0} of {1}'.format(card.rank, card.suit))

if __name__ == '__main__':
    cards = initialize_deck()
    shuffle_and_print(cards)
```

Each card in the deck is represented as an object of the `Card` class. We initialize the deck of 52 cards in the `initalize_deck()` function by creating a `Card` object for each combination of rank and suit and store each object in a list `cards` ❶. We call the `shuffle_and_print()` function with this list of cards in which we use the `random.shuffle()` function to shuffle the cards ❷. Finally, we print each of the cards ❸.

#5: Estimating the Area of a Circle

The program that simulates throwing darts to estimate the area of a circle is as follows:

```
'''
estimate_circle_area.py

Estimate the area of a circle
'''
import math
import random

def estimate(radius, total_points):
    center = (radius, radius)

    in_circle = 0
    for i in range(total_points):
        x = random.uniform(0, 2*radius)
        y = random.uniform(0, 2*radius)
        p = (x, y)
        # distance of the point created from circle's center
        d = math.sqrt((p[0]-center[0])**2 + (p[1]-center[1])**2)
        if d <= radius:
            in_circle += 1
    area_of_square = (2*radius)**2
    return (in_circle/total_points)*area_of_square
```

```
if __name__ == '__main__':
    radius = float(input('Radius: '))
    area_of_circle = math.pi*radius**2
    for points in [10**3, 10**5, 10**6]:
        print('Area: {0}, Estimated ({1}): {2}'.
              format(area_of_circle, points, estimate(radius, points)))
```

The `estimate()` function accepts two arguments—the radius of the circle and the number of virtual darts we want to throw at the dartboard. Then, we simulate throwing a dart at a square board of size `2*radius` using the `random.uniform()` function to generate the *x*- and the *y*-coordinates of point `p`, where the dart hits the board. If the distance between this point and the center of the circle is less than or equal to the radius, it means that the dart landed within the circle, and we increase the count of darts that fell within the circle. Finally, we return the fraction of the number of virtual darts that fell within the circle multiplied by the area of the square. This is the estimated area of the circle.

As we increase the number of virtual darts, the estimated area gets closer to the theoretically calculated area of the circle:

```
Radius: 1
Area: 3.141592653589793, Estimated (1000): 3.14
Area: 3.141592653589793, Estimated (100000): 3.14756
Area: 3.141592653589793, Estimated (1000000): 3.143504
```

The program to estimate the value of π has just a slight change from the previous program:

```
'''
estimate_pi.py

Estimate the value of pi
'''
import math
import random

def estimate(total_points):
    radius = 1
    center = (radius, radius)

    in_circle = 0
    for i in range(total_points):
        x = random.uniform(0, 2*radius)
        y = random.uniform(0, 2*radius)
        p = (x, y)
        # distance from circle's center
        d = math.sqrt((p[0]-center[0])**2 + (p[1]-center[1])**2)
        if d <= radius:
            in_circle += 1
    return (in_circle/total_points)*4
```

```
if __name__ == '__main__':
    for points in [10**3, 10**5, 10**6]:
        print('Known value: {0}, Estimated ({1}): {2}'.
              format(math.pi, points, estimate(points)))
```

The `estimate()` function takes a total number of virtual darts that we want to simulate throwing at a dartboard. Similar to the program for estimating the area of a circle, we keep count of the number of virtual darts that fall within the circle and return the fraction of the total darts that fell in the circle multiplied by 4. This is the estimated value of π. The estimate gets better as the number of darts increases.

Chapter 6 Programming Solutions

#1: Packing Circles into a Square

The solution to the program is as shown here:

```
'''
circle_in_square.py

Circles in a square
'''

from matplotlib import pyplot as plt

def draw_square():
    square = plt.Polygon([(1, 1), (5, 1), (5, 5), (1, 5)], closed=True)
    return square

def draw_circle(x, y):
    circle = plt.Circle((x, y), radius=0.5, fc='y')
    return circle

if __name__ == '__main__':

    ax = plt.gca()
    s = draw_square()
    ax.add_patch(s)
    y = 1.5
❶   while y < 5:
        x = 1.5
❷   while x < 5:
❸       c = draw_circle(x, y)
❹       ax.add_patch(c)

            x += 1.0
        y += 1.0

    plt.axis('scaled')
    plt.show()
```

First we add a square (each side of the square has a length of 4) to the figure by creating a `Polygon` object with the `draw_square()` function. Then, using two `while` loops at ❶ and ❷, we add circles, each with a radius of 0.5, so that they all lie inside the square. At ❸ we draw the circle and at ❹ we add the circle to the figure.

#2: Drawing the Sierpiński Triangle

The program to draw the Sierpiński triangle is as follows:

```
'''
sierpinski.py
```

```
Draw the Sierpinski triangle
'''
import random
import matplotlib.pyplot as plt

def transformation_1(p):
    x = p[0]
    y = p[1]
    x1 = 0.5*x
    y1 = 0.5*y
    return x1, y1

def transformation_2(p):
    x = p[0]
    y = p[1]
    x1 = 0.5*x + 0.5
    y1 = 0.5*y + 0.5
    return x1, y1

def transformation_3(p):
    x = p[0]
    y = p[1]
    x1 = 0.5*x + 1
    y1 = 0.5*y
    return x1, y1

def get_index(probability):
    r = random.random()
    c_probability = 0
    sum_probability = []
    for p in probability:
        c_probability += p
        sum_probability.append(c_probability)
    for item, sp in enumerate(sum_probability):
        if r <= sp:
            return item
    return len(probability)-1

def transform(p):
    # list of transformation functions
    transformations = [transformation_1, transformation_2, transformation_3]
    probability = [1/3, 1/3, 1/3]
    # pick a random transformation function and call it
    tindex = get_index(probability)
    t = transformations[tindex]
    x, y = t(p)
    return x, y

def draw_sierpinski(n):
    # We start with (0, 0)
    x = [0]
    y = [0]

    x1, y1 = 0, 0
```

```
    for i in range(n):
       x1, y1 = transform((x1, y1))
       x.append(x1)
       y.append(y1)
    return x, y

if __name__ == '__main__':
    n = int(input('Enter the desired number of points'
                  'in the Sierpinski Triangle: '))
    x, y = draw_sierpinski(n)
    # Plot the points
    plt.plot(x, y, 'o')
    plt.title('Sierpinski with {0} points'.format(n))
    plt.show()
```

Similar to the program we wrote to draw the Barnsley fern, this program has three different functions—`transformation_1()`, `transformation_2()`, and `transformation_3()`—that correspond to the three possible transformations, one of which is selected with an equal probability during each transformation.

When the program is run, it asks the user to enter the desired number of points in the triangle. This also corresponds to the number of iterations or transformations our initial point (0, 0) undergoes.

Experiment with the number of points to see different triangles.

#3: Exploring Hénon's Function

The program that graphs the Hénon function with 20,000 points is as follows:

```
'''
henon.py

Plot 20,000 iterations of the Henon function
'''

import matplotlib.pyplot as plt

def transform(p):
    x,y  = p
    x1 = y + 1.0 - 1.4*x**2
    y1 = 0.3*x

    return x1, y1

if __name__ == '__main__':
    p = (0, 0)
    x = [p[0]]
    y = [p[1]]
    for i in range(20000):
        p = transform(p)
```

```
        x.append(p[0])
        y.append(p[1])
    plt.plot(x, y, 'o')
    plt.show()
```

The initial point (0, 0) undergoes transformation for 20,000 points. The transformation in this case is implemented by the `transform()` function. We store the coordinates of each of the points in two lists, `x` and `y`, and finally plot the data.

The program to animate drawing the Hénon function requires a bit more effort:

```
'''
henon_animation.py

Animating 20000 iterations of the Henon function

'''

import matplotlib.pyplot as plt
from matplotlib import animation

def transform(p):
    x,y  = p
    x1 = y + 1.0 - 1.4*x**2
    y1 = 0.3*x

    return x1, y1

def update_points(i, x, y, plot):
    plot.set_data(x[:i], y[:i])
    return plot,

if __name__ == '__main__':
    p = (0, 0)
    x = [p[0]]
    y = [p[1]]
    for i in range(10000):
        p = transform(p)
        x.append(p[0])
        y.append(p[1])

    fig = plt.gcf()
    ax = plt.axes(xlim = (min(x), max(x)),
                  ylim = (min(y), max(y)))
❶ plot = plt.plot([], [], 'o')[0]
    anim = animation.FuncAnimation(fig, update_points,
                                   fargs=(x, y, plot),
                                   frames = len(x),
                                   interval = 25)
    plt.title('Henon Function Animation')
    plt.show()
```

First we create two lists to store the coordinates of all the points. Then, we create an empty plot by calling the `plot()` function with two empty lists ❶. Recall that matplotlib's `plot()` function returns a list of objects (Chapter 2), and in this case there is only one object, which we retrieve using the index 0. We create a label, `plot`, to refer to that object. Then, we create the `FuncAnimation` object with the relevant arguments. The animation will use the number of points to set the number of frames. Frame 1 will have only the first point, frame 2 will have the first two points, and so on. The `update_points()` function updates the figure each frame. In the `update_points()` function, we use matplotlib's `set_data()` function to add points to `plot`. In frame `i` we specify that we want the first `i` points stored in the `x` and `y` lists.

When you run the program, you should see an animation showing all the points lined along the curves.

#4: Drawing the Mandelbrot Set

The program drawing the Mandelbrot set is shown here:

```
'''
mandelbrot.py

Draw a Mandelbrot set

Using "Escape time algorithm" from:
http://en.wikipedia.org/wiki/Mandelbrot_set#Computer_drawings

Thanks to http://www.vallis.org/salon/summary-10.html for some important
ideas for implementation.

'''
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Subset of the complex plane we are considering
x0, x1 = -2.5, 1
y0, y1 = -1, 1

def initialize_image(x_p, y_p):
    image = []
    for i in range(y_p):
        x_colors = []
        for j in range(x_p):
            x_colors.append(0)
        image.append(x_colors)
    return image

def mandelbrot_set():
    # Number of divisions along each axis
    n = 400
    # Maximum iterations
    max_iteration=1000
```

```
    image = initialize_image(n, n)

    # Generate a set of equally spaced points in the region
    # above
    dx = (x1-x0)/(n-1)
    dy = (y1-y0)/(n-1)
❶ x_coords = [x0 + i*dx for i in range(n)]
❷ y_coords = [y0 + i*dy for i in range(n)]

    for i, x in enumerate(x_coords):
        for k, y in enumerate(y_coords):
            z1 = complex(0, 0)
            iteration = 0
            c = complex(x, y)
❸           while (abs(z1) < 2  and iteration < max_iteration):
                    z1 = z1**2 + c
                    iteration += 1
❹           image[k][i] = iteration
    return image

if __name__ == '__main__':
    image = mandelbrot_set()
❺   plt.imshow(image, origin='lower', extent=(x0, x1, y0,y1),
                cmap=cm.Greys_r, interpolation='nearest')
    plt.show()
```

The function `mandelbrot_set()` draws the Mandelbrot set in the region enclosed by the points (–2.5, –1) and (1, 1) for a total of 1,600 equally distributed points. We use the `initialize_image()` function as discussed in the problem description to initialize a list of lists, which are then populated with the appropriate color.

We generate 400 equally spaced points along each axis at ❶ and ❷. Then, using two `for` loops, we iterate over each of the 1,600 points, assigning a color to each one. To assign a color, we first create a complex number that corresponds to each of these points using the `complex()` function. Then, we use a `while` loop ❸ to implement steps 4 and 5 of the algorithm as described in the problem description. When the `while` loop exits, the iteration value determines the color of the point ❹.

Finally, we return the list of lists, `image`, which is passed as an argument to the `imshow()` function ❺. The `extent` keyword argument specifies the region we considered for drawing the set, and we pass in the *x*-coordinate of the points followed by the *y*-coordinates. When you run the program, you should see the Mandelbrot set (Figure 7).

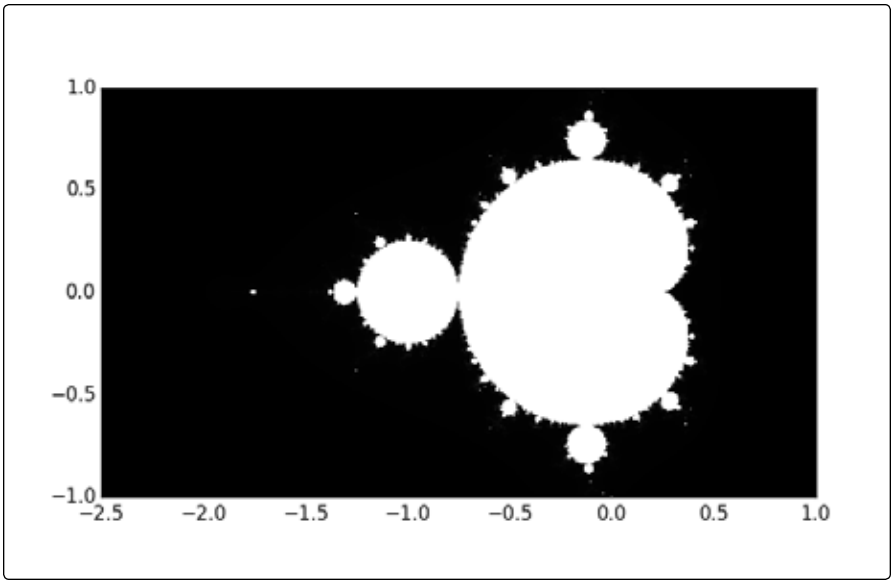


Figure 7: Mandelbrot set drawn by plotting 1,600 points in the region bounded by (–2.5, –1.0) and (1.0, 1.0).

If you change the number of points along each axis and play around with the number of iterations, you can observe the effect of each change on the final figures you see.

Chapter 7 Programming Solutions

#1: Verify the Continuity of a Function at a Point

The solution to this challenge finds the limit of a function at a given point from both the positive and negative directions and evaluates the function at that point. The solution is as follows:

```
'''
verify_continuity.py

Verify the continuity of a function
'''

from sympy import Limit, Symbol, sympify, SympifyError

def check_continuity(f, var, a):
❶    l1 = Limit(f, var, a, dir='+').doit()
❷    l2 = Limit(f, var, a, dir='-').doit()
    f_val = f.subs({var:a})

    if l1 == l2 and f_val == l1:
        print('{0} is continuous at {1}'.format(f, a))
    else:
        print('{0} is not continuous at {1}'.format(f, a))

if __name__ == '__main__':
    f = input('Enter a function in one variable: ')
    var = input('Enter the variable: ')
    a = float(input('Enter the point to check the continuity at: '))
    try:
        f = sympify(f)
    except SympifyError:
        print('Invalid function entered')
    else:
        var = Symbol(var)
        d = check_continuity(f, var, a)
```

The `check_continuity()` function accepts a function, the variable of the function, and the point at which the continuity is to be checked. It then evaluates the limit from both directions using `dir` while creating the `Limit` object at ❶ and ❷, and evaluates the function at the given point using the `subs()` method. Next, it checks if the limits `l1` and `l2` are equal and if so, it checks if they are equal to the value of the function at that point. If both the conditions are true, the function is continuous at that point.

#2: Implement the Gradient Descent

As noted in the problem statement, the only difference between the solution for this challenge and the program implementing gradient ascent is how the next point is created. The following program implements the gradient descent algorithm to find the minimum value of a function:

```
'''
grad_descent.py

Use gradient descent to find the minimum value of a
single variable function. This also checks for the existence
of a solution for the equation f'(x)=0 and plots the intermediate
points traversed.
'''

from sympy import Derivative, Symbol, sympify, solve
import matplotlib.pyplot as plt

def grad_descent(x0, f1x, x):
    # check if f1x=0 has a solution
    if not solve(f1x):
        print('Cannot continue, solution for {0}=0 does not exist'.
format(f1x))
        return None
    epsilon =  1e-6
    step_size = 1e-4
    x_old = x0
❶   x_new = x_old-step_size*f1x.subs({x: x_old}).evalf()

    # list to store the X values traversed
    X_traversed = []
❷   while abs(x_old - x_new) > epsilon:
❸   X_traversed.append(x_new)
        x_old = x_new
        x_new = x_old - step_size*f1x.subs({x:x_old}).evalf()

    return x_new, X_traversed

def frange(start, final, interval):

    numbers = []
    while start < final:
        numbers.append(start)
        start = start + interval

    return numbers
```

```
def create_plot(X_traversed, f, var):
    # First create the graph of the function itself
    x_val = frange(-1, 1, 0.01)
    f_val = [f.subs({var:x}) for x in x_val]
    plt.plot(x_val, f_val, 'bo')
    # calculate the function value at each of the intermediate
    # points traversed
    f_traversed = [f.subs({var:x}) for x in X_traversed]
    plt.plot(X_traversed, f_traversed, 'r.')
    plt.legend(['Function', 'Intermediate points'], loc='best')
    plt.show()


if __name__ == '__main__':

    f = input('Enter a function in one variable: ')
    var = input('Enter the variable to differentiate with respect to: ')
    var0 = float(input('Enter the initial value of the variable: '))
    try:
        f = sympify(f)
    except SympifyError:
        print('Invalid function entered')
    else:
        var = Symbol(var)
        d = Derivative(f, var).doit()
        var_min, X_traversed = grad_descent(var0, d, var)
        if var_min:
            print('{0}: {1}'.format(var.name, var_min))
            print('Minimum value: {0}'.format(f.subs({var:var_min})))
❹       create_plot(X_traversed, f, var)
```

At ❶ we create the point. As in the program for gradient ascent, we continue exploring new points until the new point and the current point differ by a value less than $1e-6$ ❷. We also store all the points in a list, `X_traversed` ❸, so that we can plot all the intermediate points in the `create_plot()` function. A sample execution of the program would look like this:

```
Enter a function in one variable: 3*x**2 + 2*x
Enter the variable to differentiate with respect to: x
Enter the initial value of the variable: 0.1
x: -0.331668643986980
Minimum value: -0.333325019761474
```

Figure 8 shows the graph of this example.

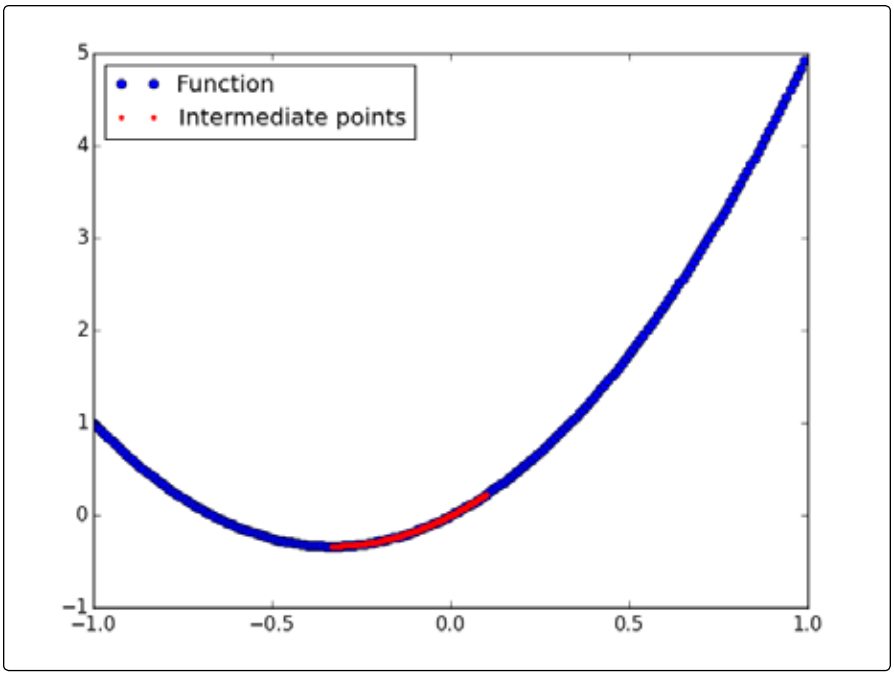


Figure 8: Graph showing the function $3x^2 + 2x$ and how gradient descent finds the minimum of the function

#3: Area Between Two Curves

Here's the solution for this challenge:

```
'''
area_curves.py

Find the area enclosed by two curves between two points
'''

from sympy import Integral, Symbol, SympifyError, sympify

def find_area(f, g, var, a, b):
    a = Integral(f-g, (var, a, b)).doit()
    return a

if __name__ == '__main__':
    f = input('Enter the upper function in one variable: ')
    g = input('Enter the lower function in one variable: ')
    var = input('Enter the variable: ')
    l = float(input('Enter the lower bound of the enclosed region: '))
    u = float(input('Enter the upper bound of the enclosed region: '))

    try:
        f = sympify(f)
        g = sympify(g)
```

```
    except SympifyError:
        print('One of the functions entered is invalid')
    else:
        var = Symbol(var)
        print('Area enclosed by {0} and {1} is: {2} '.
              format(f, g, find_area(f, g, var, l, u)))
```

When the program runs, it asks the user to input two functions (the lower and upper bounds of the variable) and calls the `find_area()` function with these values. The `find_area()` function then calculates the integral

$$\int_a^b \left(f(x) - g(x) \right) dx$$

by creating the `Integral` object, which is the area enclosed by the two curves between `a` and `b`. An example execution of the program is as follows:

```
Enter the upper function in one variable: x+1
Enter the lower upper function in one variable: x*exp(-x**2)
Enter the variable: x
Enter the lower bound of the enclosed region: 0
Enter the upper bound of the enclosed region: 2
Area enclosed by x + 1 and x*exp(-x**2) is: 3.50915781944437
```

I should note that this challenge describes a specific case of finding the area between curves. You can learn about other cases at *http://tutorial.math.lamar.edu/Classes/CalcI/AreaBetweenCurves.aspx* and write programs to solve them. This example demonstrates solving an adaptation of this website's Example 2 using our program.

#4: Finding the Length of a Curve

This solution finds the length of a function between two points:

```
'''
length_curve.py

Find the length of a curve between two points
'''

from sympy import Derivative, Integral, Symbol, sqrt, SympifyError, sympify

def find_length(fx, var, a, b):
    deriv = Derivative(fx, var).doit()
    length = Integral(sqrt(1+deriv**2), (var, a, b)).doit().evalf()
    return length

if __name__ == '__main__':
    f = input('Enter a function in one variable: ')
    var = input('Enter the variable: ')
    l = float(input('Enter the lower limit of the variable: '))
    u = float(input('Enter the upper limit of the variable: '))
```

```
    try:
        f = sympify(f)
    except SympifyError:
        print('Invalid function entered')
    else:
        var = Symbol(var)
        print('Length of {0} between {1} and {2} is: {3} '.
              format(f, l, u, find_length(f, var, l, u)))
```

A sample of the program is shown here using the example in the challenge description as the input:

```
Enter a function in one variable: 2*x**2 + 3*x + 1
Enter the variable: x
Enter the lower limit of the variable: -5
Enter the upper limit of the variable: 10
Length of 2*x**2 + 3*x + 1 between -5.0 and 10.0 is: 268.372650946022
```

Another example, using input from Khan Academy (*https://www.khanacademy.org/math/integral-calculus/solid_revolution_topic/arc-length/v/arc-length-example-2*), is as follows:

```
Enter a function in one variable: x**3/6 + 1/(2*x)
Enter the variable: x
Enter the lower limit of the variable: 1
Enter the upper limit of the variable: 2
Length of x**3/6 + 1/(2*x) between 1.0 and 2.0 is: 1.41666666666667
```