Think Python

Exercise 5.1.

The time module provides a function, also named time, that returns the current Greenwich Mean Time in “the epoch”, which is an arbitrary time used as a reference point. On UNIX systems, the epoch is 1 January 1970.

>>> import time

>>> time.time()

1437746094.5735958

Write a script that reads the current time and converts it to a time of day in hours, minutes, and seconds, plus the number of days since the epoch.

```
import time

# Get the current time since the epoch
current_time = time.time()

# Convert to days since the epoch
days_since_epoch = int(current_time // (24 * 3600))

# Get the time of day in seconds
seconds_in_day = int(current_time % (24 * 3600))

# Calculate hours, minutes, and seconds
hours = seconds_in_day // 3600
minutes = (seconds_in_day % 3600) // 60
seconds = seconds_in_day % 60

# Display the results
print(f"Current time of day: {hours:02}:{minutes:02}:{seconds:02}")
print(f"Days since the epoch: {days_since_epoch}")
```

```
Current time of day: 23:00:48

Days since the epoch: 20110
```

Exercise 5.2.

Fermat's Last Theorem says that there are no positive integers a, b, and c such that

$$a^n + b^n = c^n$$

for any values of n greater than 2.

1. Write a function named check_fermat that takes four parameters—a, b, c and n—and checks to see if Fermat's theorem holds. If n is greater than 2 and

$$a^n + b^n = c^n$$

the program should print, "Holy smokes, Fermat was wrong!" Otherwise the program should print, "No, that doesn't work."

```
def check_fermat(a, b, c, n):
    """
    Checks Fermat's Last Theorem for given values of a, b, c, and n.
    """
    if n > 2 and (a**n + b**n == c**n):
      print("Holy smokes, Fermat was wrong!")
    else:
      print("No, that doesn't work.")

check_fermat(3, 4, 5, 3)
check_fermat(1, 2, 3, 3)
```

```
No, that doesn't work.
No, that doesn't work.
```

2. Write a function that prompts the user to input values for a, b, c and n, converts them to integers, and uses check_fermat to check whether they violate Fermat's theorem.

```
def prompt_and_check_fermat():
    """
    Prompts the user to input values for a, b, c, and n,
    and checks whether they violate Fermat's Last Theorem.
    """
    # Prompting the user for input
    print("Enter values for a, b, c, and n to check Fermat's theorem.")
    try:
        a = int(input("Enter value for a (positive integer): "))
        b = int(input("Enter value for b (positive integer): "))
        c = int(input("Enter value for c (positive integer): "))
        n = int(input("Enter value for n (integer > 2): "))

        # Use the check_fermat function
        check_fermat(a, b, c, n)
    except ValueError:
        print("Invalid input. Please enter integers only.")

# Example usage
prompt_and_check_fermat()
```

```
Enter values for a, b, c, and n to check Fermat's theorem.
Enter value for a (positive integer): 1
Enter value for b (positive integer): 2
Enter value for c (positive integer): 3
Enter value for n (integer > 2): 2
No, that doesn't work.
```

Exercise 5.3.

If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.)

1. Write a function named is_triangle that takes three integers as arguments, and that prints either "Yes" or "No", depending on whether you can or cannot form a triangle from sticks with the given lengths.

```
def is_triangle(a, b, c):
        """
        Checks if three stick lengths can form a triangle.
        Prints "Yes" if they can form a triangle, otherwise prints "No".
        """
        # Check if any length is greater than or equal to the sum of the other two
        if a + b > c and a + c > b and b + c > a:
                print("Yes")
        else:
                print("No")

is_triangle(3, 4, 5)
is_triangle(12, 1, 1)
```

```
Yes
No
```

2. Write a function that prompts the user to input three stick lengths, converts them to integers, and uses is_triangle to check whether sticks with the given lengths can form a triangle.

```
def prompt_and_check_triangle():
        """
        Prompts the user to input three stick lengths,
        and checks if they can form a triangle.
        """
        print("Enter three stick lengths to check if they can form a triangle.")
        try:
                # Get input from the user
                a = int(input("Enter length of the first stick: "))
                b = int(input("Enter length of the second stick: "))
                c = int(input("Enter length of the third stick: "))

                # Use the is_triangle function
                is_triangle(a, b, c)
        except ValueError:
                print("Invalid input. Please enter positive integers only.")

prompt_and_check_triangle()
```

```
Enter three stick lengths to check if they can form a triangle.
Enter length of the first stick: 5
Enter length of the second stick: 6
Enter length of the third stick: 7
Yes
```

Exercise 5.4. What is the output of the following program? Draw a stack diagram that shows the state of the program when it prints the result.

```
def recurse(n, s):
        if n == 0:
                print(s)
        else:
                recurse(n-1, n+s)
                recurse(3, 0)
```

1. What would happen if you called this function like this: recurse(-1, 0)?

If recurse(-1, 0) is called:

The condition if n == 0 will not be met because n = -1.

The program will attempt to call recurse indefinitely as there is no base case to stop the recursion.

2. Write a docstring that explains everything someone would need to know in order to use this function (and nothing else).

```
def recurse(n, s):
    """
    Prints the sum of integers from n down to 0, calculated recursively.

    Parameters:
        n (int): A non-negative integer. The recursion continues until n reaches 0.
        s (int): An accumulator that stores the running total.

    Returns:
        None. Prints the accumulated total (s) when n reaches 0.

    Usage:
        - Call recurse(n, s) with n >= 0. For example, recurse(3, 0).
        - If n < 0, the function will result in infinite recursion (RecursionError).
    """
```

Exercise 5.5.

Read the following function and see if you can figure out what it does (see the examples in Chapter 4). Then run it and see if you got it right.

```
def draw(t, length, n):
        if n == 0:
                return
        angle = 50
        t.fd(length*n)
        t.lt(angle)
        draw(t, length, n-1)
        t.rt(2*angle)
        draw(t, length, n-1)
        t.lt(angle)
        t.bk(length*n)
```

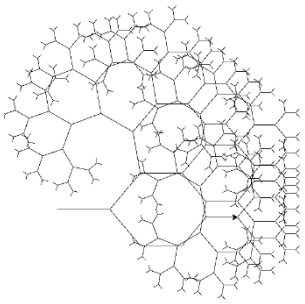
The function creates a recursive pattern where the turtle moves forward, turns left, recursively draws, turns right and then draws again before backing up to the starting point.

```
import turtle

def draw(t, length, n):
        if n == 0:
                return
        angle = 50
        t.fd(length*n)
        t.lt(angle)
        draw(t, length, n-1)
        t.rt(2*angle)
        draw(t, length, n-1)
        t.lt(angle)
        t.bk(length*n)

my_turtle = turtle.Turtle()
my_turtle.speed(0)

draw(my_turtle, 10, 10)
```



Exercise 5.6.

The Koch curve is a fractal that looks something like Figure 5.2. To draw a Koch curve with length x, all you have to do is

1. Draw a Koch curve with length x/3.
2. Turn left 60 degrees.
3. Draw a Koch curve with length x/3.
4. Turn right 120 degrees.
5. Draw a Koch curve with length x/3.
6. Turn left 60 degrees.
7. Draw a Koch curve with length x/3.

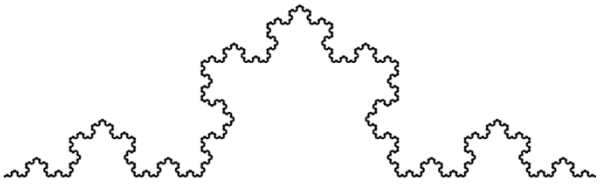


Figure 5.2: A Koch curve.

The exception is if x is less than 3: in that case, you can just draw a straight line with length x.

1. Write a function called koch that takes a turtle and a length as parameters, and that uses the turtle to draw a Koch curve with the given length.

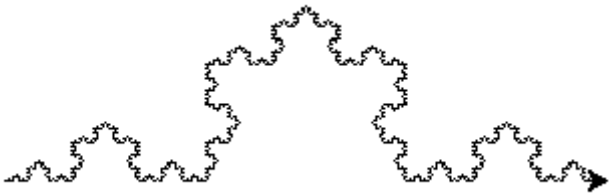
```
import turtle

def koch(t, length):
   # Base case: If length is less than 3, just draw a straight line
   if length < 3:
       t.fd(length)
   else:
       koch(t, length / 3)
       t.lt(60)
       koch(t, length / 3)
       t.rt(120)
       koch(t, length / 3)
       t.lt(60)
       koch(t, length / 3)

# Set up the turtle
my_turtle = turtle.Turtle()
my_turtle.speed(0)

# Call the koch function to draw the Koch curve
koch(my_turtle, 300)

# Finish drawing and keep the window open
turtle.done()
```



2. Write a function called snowflake that draws three Koch curves to make the outline of a snowflake.

```
def snowflake(t, length):
    # Draw three Koch curves to form the snowflake
    for _ in range(3):
       koch(t, length)  # Draw one Koch curve
       t.rt(120)  # Turn 120 degrees
```