# Chapter 5 Loops

# Motivations

Suppose that you need to print a string (e.g., "Programming is fun!") a hundred times. It would be tedious to have to write the following statement a hundred times:

print("Programming is fun!");

So, how do you solve this problem?

# Opening Problem

Problem:

100 times

```
print("Programming is fun!");
print("Programming is fun!");
print("Programming is fun!");
print("Programming is fun!");
print("Programming is fun!");
print("Programming is fun!");

...

...

...
print("Programming is fun!");
print("Programming is fun!");
print("Programming is fun!");
```

# Introducing while Loops

```
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

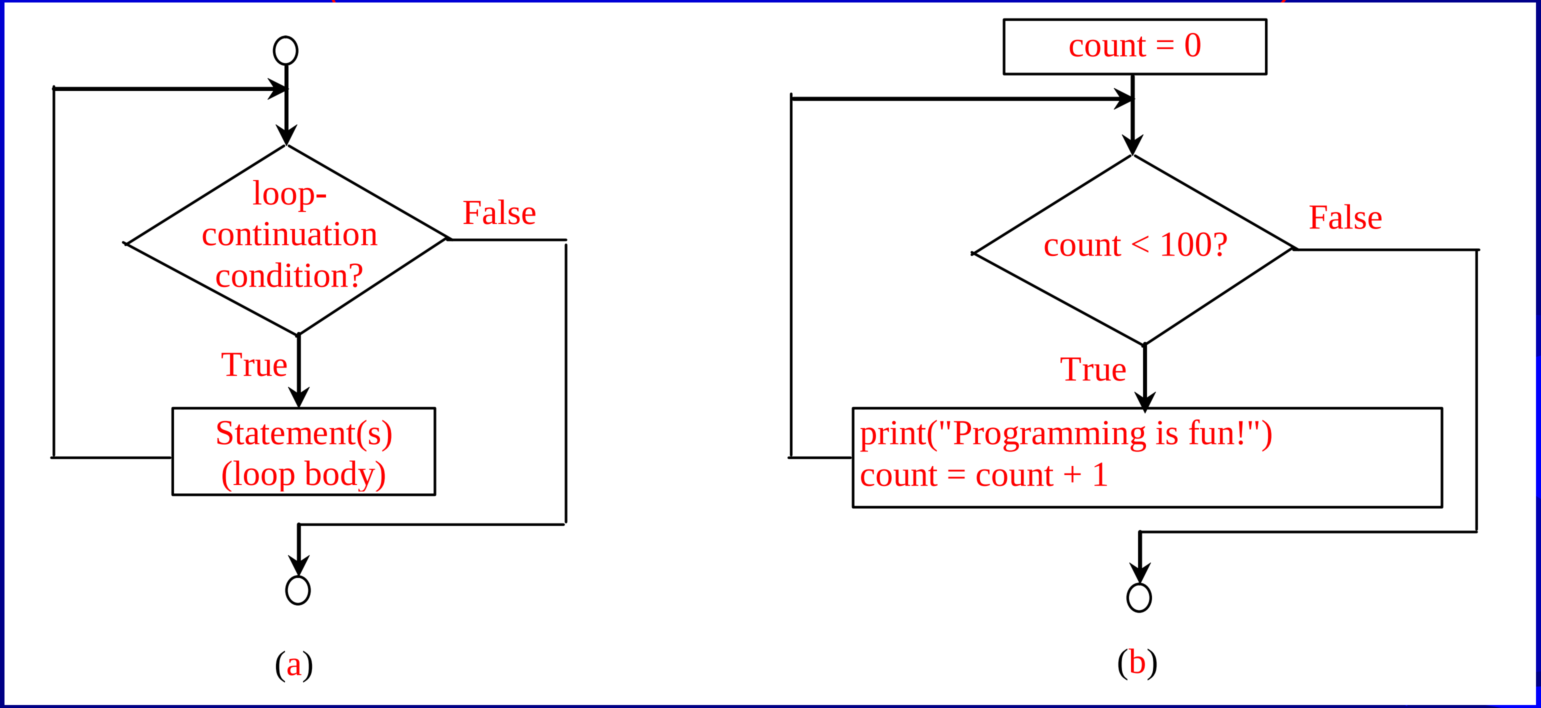
# Objectives

- To write programs for executing statements repeatedly by using a **while** loop (§5.2).
- To develop loops following the loop design strategy (§§5.2.1-5.2.3).
- To control a loop with the user's confirmation (§5.2.4).
- To control a loop with a sentinel value (§5.2.5).
- To obtain a large amount of input from a file by using input redirection instead of typing from the keyboard (§5.2.6).
- To use **for** loops to implement counter-controlled loops (§5.3).
- To write nested loops (§5.4).
- To learn the techniques for minimizing numerical errors (§5.5).
- To learn loops from a variety of examples (**GCD**, **FutureTuition**, **MonteCarloSimulation**, **PrimeNumber**) (§§5.6, 5.8).
- To implement program control with **break** and **continue** (§5.7).
- To use a loop to control and simulate a random walk (§5.9).

# `while` Loop Flow Chart

```
while loop-continuation-condition:
    # Loop body
    Statement(s)
```

```
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```



animation

# Trace while Loop

Initialize count

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

animation

# Trace while Loop, cont.

(count < 2) is true

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

animation

# Trace while Loop, cont.

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

Print Welcome to Python

animation

# Trace while Loop, cont.

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

Increase count by 1
count is 1 now

animation

# Trace while Loop, cont.

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

(count < 2) is still true since count is 1

animation

# Trace while Loop, cont.

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

Print Welcome to Python

animation

# Trace while Loop, cont.

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

Increase count by 1
count is 2 now

animation

# Trace while Loop, cont.

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

(count < 2) is false since count is 2 now

animation

# Trace while Loop

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
```

The loop exits. Execute the next statement after the loop.

# The `while` Loop: a Condition-Controlled Loop

- In order for a loop to stop executing, something has to happen inside the loop to make the condition false
- Iteration: one execution of the body of a loop
- `while` loop is known as a *pretest* loop
  - Tests condition before performing an iteration
    - Will never execute if condition is false to start with
    - Requires performing some steps prior to the loop

# Infinite Loops

- Loops must contain within themselves a way to terminate
  - Something inside a `while` loop must eventually make the condition false
- Infinite loop: loop that does not have a way of stopping
  - Repeats until program is interrupted
  - Occurs when programmer forgets to include stopping code in the loop

```
# Example 1
i = 1
while i>0:
    print(i)
    i = i + 1

# Example 2
j = 1
while True:
    print(j)
    j = j + 1
```

# Problem: An Advanced Math Learning Tool

The Math subtraction learning tool program generates just one question for each run. You can use a loop to generate questions repeatedly. This example gives a program that generates five questions and reports the number of the correct answers after a student answers all five questions.

SubtractionQuizLoop

Run

# Problem: Guessing Numbers

Write a program that randomly generates an integer between 0 and 100, inclusive. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can choose the next input intelligently. Here is a sample run:

GuessNumberOneTime Run

GuessNumber Run

# Ending a Loop with a Sentinel Value

- Often the number of times a loop is executed is not predetermined. You may use an input value to signify the end of the loop. Such a value is known as a *sentinel value*.

- Must be distinctive enough so as not to be mistaken for a regular value in the sequence

- Write a program that reads and calculates the sum of an unspecified number of integers. The input 0 signifies the end of the input.

SentinelValue Run

# Numerical Errors

- Numeric errors involving floating-point numbers are inevitable.
- Floating Point Arithmetic: Issues and Limitations (https://docs.python.org/3/tutorial/floatingpoint.html )
- IEEE-754 Floating Point Converter (https://www.h-schmidt.net/FloatConverter/IEEE754.html )
- https://realpython.com/python-data-types/#floating-point-numbers
- Here is an example that sums a series that starts with 0.01 and ends with 1.0. The numbers in the series will increment by 0.01, as follows: 0.01 + 0.02 + 0.03 and so on.

TestSum

Run

# Caution

Don't use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results. Consider the following code for computing 1 + 0.9 + 0.8 + ... + 0.1:

```
item = 1
sum = 0
while item != 0: # No guarantee item will be 0
    sum += item
    item -= 0.1
print(sum)
```

Variable item starts with 1 and is reduced by 0.1 every time the loop body is executed. The loop should terminate when item becomes 0. However, there is no guarantee that item will be exactly 0, because the floating-point arithmetic is approximated. This loop seems OK on the surface, but it is actually an infinite loop.

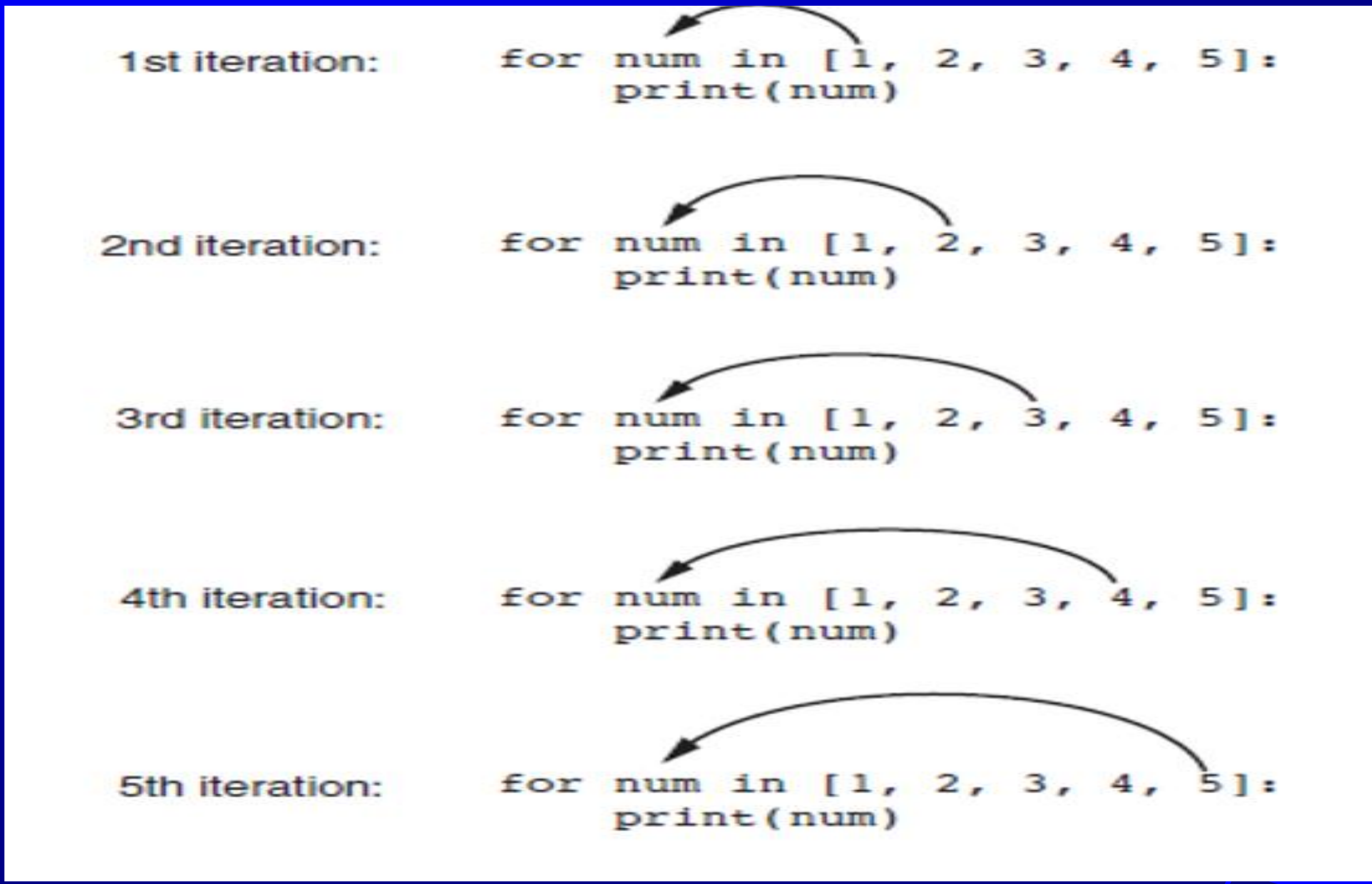
# The `for` Loop: a Count-Controlled Loop

- Count-Controlled loop: iterates a specific number of times
  - Use a `for` statement to write count-controlled loop
    - Designed to work with sequence of data items
      - Iterates once for each item in the sequence
    - General format:

```
for variable in [val1, val2, etc]:
    statements
```

    - Target variable: the variable which is the target of the assignment at the beginning of each iteration

# For Loop with a List



# Using the `range` Function with the `for` Loop

- The `range` function simplifies the process of writing a `for` loop
  - `range` returns an iterable object
    - Iterable: contains a sequence of values that can be iterated over
- `range` characteristics:
  - One argument: used as ending limit
  - Two arguments: starting value and ending limit
  - Three arguments: third argument is step value

# One argument: used as ending limit: **range(endValue)**

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
>>>
```

- i starts from 0

# Two arguments: starting value and ending limit : **range(initialValue, endValue)**

```
for i in range(initialValue, endValue):
    # Loop body
```

```
i = initialValue  # Initialize loop-control variable
while i < endValue:
    # Loop body
    ...
    i++ # Adjust loop-control variable
```

Two arguments: starting value and ending limit :

# range(initialValue, endValue)

```
>>> for v in range(4, 8):
...     print(v)
...
4
5
6
7
>>>
```

# Three arguments: third argument is step value

# **range(initialValue, endValue, step)**

```
>>> for v in range(3, 9, 2):
...     print(v)
...
3
5
7
>>>
```

# Generating an Iterable Sequence that Ranges from Highest to Lowest

- The `range` function can be used to generate a sequence with numbers in descending order
  - Make sure starting number is larger than end limit, and step value is negative
  - Example: `range (5, 1, -1)`

# Three arguments: third argument is step value

# **range(initialValue, endValue, step)**

```
>>> for v in range(5, 1, -1):
...     print(v)
...
5
4
3
2
>>>
```

# Nested Loops

- Nested loop: loop that is contained inside another loop
- Inner loop goes through all of its iterations for each iteration of outer loop
- Inner loops complete their iterations faster than outer loops
- Total number of iterations in nested loop: number_iterations_inner x number_iterations_outer

```
for i in range(1,5):
    for j in range(1,4):
        print("i=",i," j=",j)
```

Output:

```
i= 1   j= 1
i= 1   j= 2
i= 1   j= 3
i= 2   j= 1
i= 2   j= 2
i= 2   j= 3
i= 3   j= 1
i= 3   j= 2
i= 3   j= 3
i= 4   j= 1
i= 4   j= 2
i= 4   j= 3
```

Number of Iterations = 4 x 3 = 12

# Nested Loops

- Nested loop: loop that is contained inside another loop

- Problem: Write a program that uses nested for loops to print a multiplication table.

MultiplicationTable

Run

# Problem:
# Finding the Greatest Common Divisor

Problem: Write a program that prompts the user to enter two positive integers and finds their greatest common divisor.

Solution: Suppose you enter two integers 4 and 2, their greatest common divisor is 2. Suppose you enter two integers 16 and 24, their greatest common divisor is 8. So, how do you find the greatest common divisor? Let the two input integers be n1 and n2. You know number 1 is a common divisor, but it may not be the greatest commons divisor. So you can check whether k (for k = 2, 3, 4, and so on) is a common divisor for n1 and n2, until k is greater than n1 or n2.

GreatestCommonDivisor

Run

# Problem: Predicting the Future Tuition

Problem: Suppose that the tuition for a university is $10,000 this year and tuition increases 7% every year. In how many years will the tuition be doubled?

FutureTuition

Run

# Problem: Predicating the Future Tuition

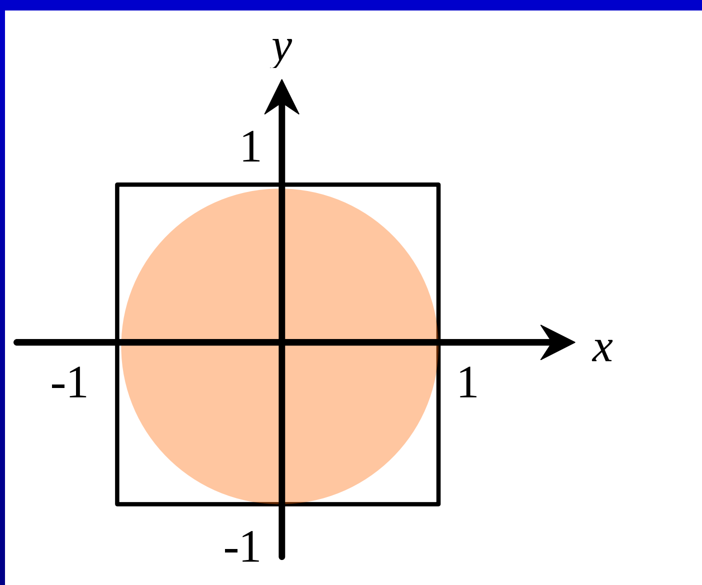
```
year = 0  # Year 0
tuition = 10000
year += 1 # Year 1
tuition = tuition * 1.07
year += 1 # Year 2
tuition = tuition * 1.07
year += 1 # Year 3
tuition = tuition * 1.07
```

[FutureTuition]

Run

# Problem: *Monte Carlo Simulation*

The Monte Carlo simulation refers to a technique that uses random numbers and probability to solve problems. This method has a wide range of applications in computational mathematics, physics, chemistry, and finance. This section gives an example of using the Monto Carlo simulation for estimating π.




circleArea / squareArea = π / 4.

π can be approximated as 4 * numberOfHits / 1000000.

MonteCarloSimulation

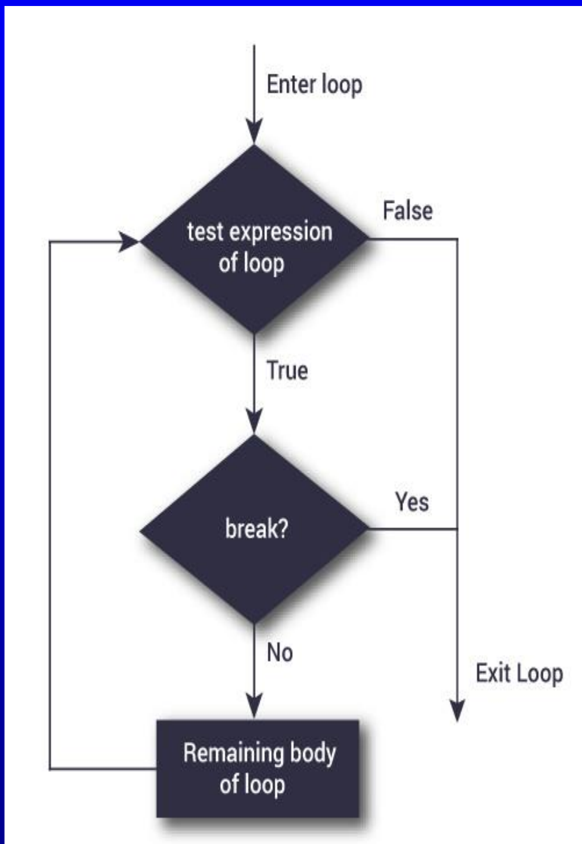
Run

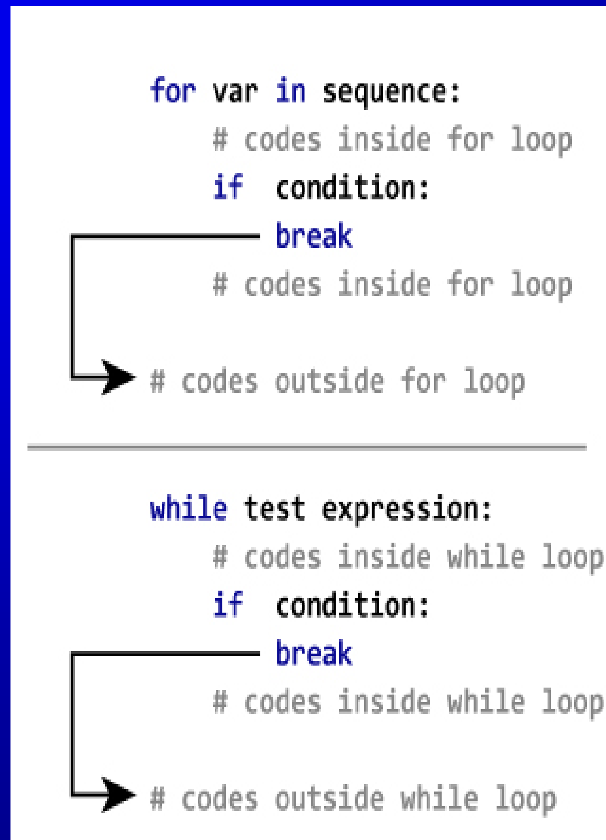
# break Statement in Python

- The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

- If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

# break Statement in Python



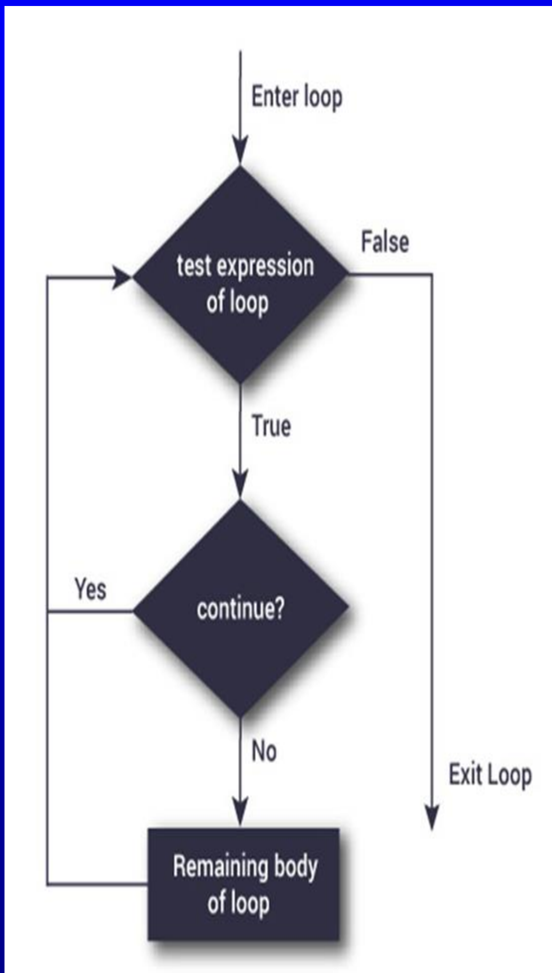

```
# Use of break statement
# inside loop
for val in "string":
    if val == "i":
        break
        print(val)
print("The end")
```

```
s
t
r
The end
```

# Python continue statement

- The continue statement is used to skip the rest of the code inside a loop for the current iteration only.
- Loop does not terminate but continues on with the next iteration.

# Python continue statement




```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

```
# Program to show the use of
# continue statement inside
# loops
for val in "string":
    if val == "i":
        continue
    print(val)
print("The end")
```

```
s
t
r
n
g
The end
```

# Using `break` and `continue`

Examples for using the `break` and `continue` keywords:

- TestBreak.py

  TestBreak    Run

- TestContinue.py

  TestContinue    Run

# break

```
sum = 0
number = 0

while number < 20:
    number += 1
    sum += number
    if sum >= 100:
        break

print("The number is ", number)
print("The sum is ", sum)
```

Break out of the loop

# continue

```
sum = 0
number = 0

while (number < 20):
    number += 1
    if (number == 10 or number == 11):
        continue
    sum += number

print("The sum is ", sum)
```

Jump to the end of the iteration

# Post-test Loop with break

```
// C language post-test (do-while)
// loop
int i = 1;
do{
  printf("%d\n", i);
  i = i + 1;
} while(i <= 3);
```

```
Output:
1
2
3
```

```
# Python
i = 1
while True:
    print(i)
    i = i + 1
    if(i > 3):
        break
```

# Guessing Number Problem Revisited

Here is a program for guessing a number. You can rewrite it using a **break** statement.

GuessNumberUsingBreak

Run

# Problem: Displaying Prime Numbers

Problem: Write a program that displays the first 50 prime numbers in five lines, each of which contains 10 numbers. An integer greater than 1 is *prime* if its only positive divisor is 1 or itself. For example, 2, 3, 5, and 7 are prime numbers, but 4, 6, 8, and 9 are not.

Solution: The problem can be broken into the following tasks:

- For number = 2, 3, 4, 5, 6, ..., test whether the number is prime.
- Determine whether a given number is prime.
- Count the prime numbers.
- Print each prime number, and print 10 numbers per line.

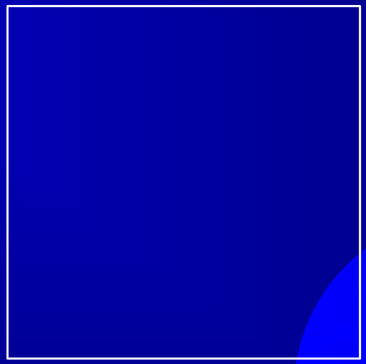
PrimeNumber

Run

# Turtle Graphics: Using Loops to Draw Designs

- You can use loops with the turtle to draw both simple shapes and elaborate designs. For example, the following for loop iterates four times to draw a square that is 100 pixels wide:

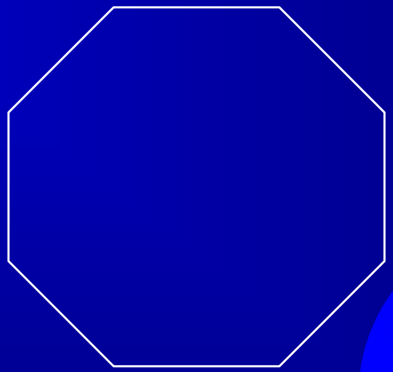
```
import turtle
for x in range(4):
    turtle.forward(100)
    turtle.right(90)
turtle.done()
```



# Turtle Graphics: Using Loops to Draw Designs

- This `for` loop iterates eight times to draw the octagon:

```
import tutle
for x in range(8):
    turtle.forward(100)
    turtle.right(45)
turtle.done()
```

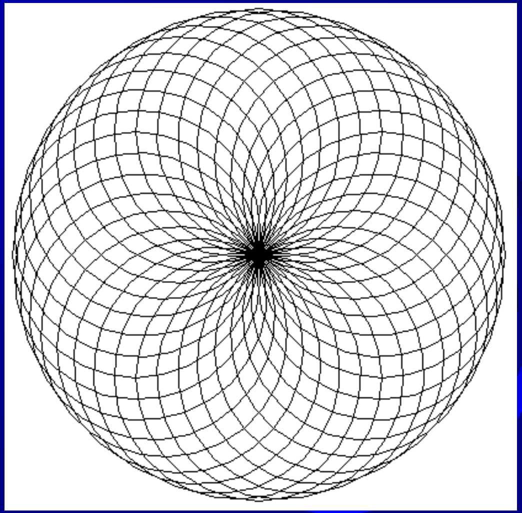


# Turtle Graphics: Using Loops to Draw Designs

- You can create interesting designs by repeatedly drawing a simple shape, with the turtle tilted at a slightly different angle each time it draws the shape.

```
import turtle
NUM_CIRCLES = 36      # Number of circles to draw
RADIUS = 100          # Radius of each circle
ANGLE = 10            # Angle to turn

for x in range(NUM_CIRCLES):
    turtle.circle(RADIUS)
    turtle.left(ANGLE)
turtle.done()
```



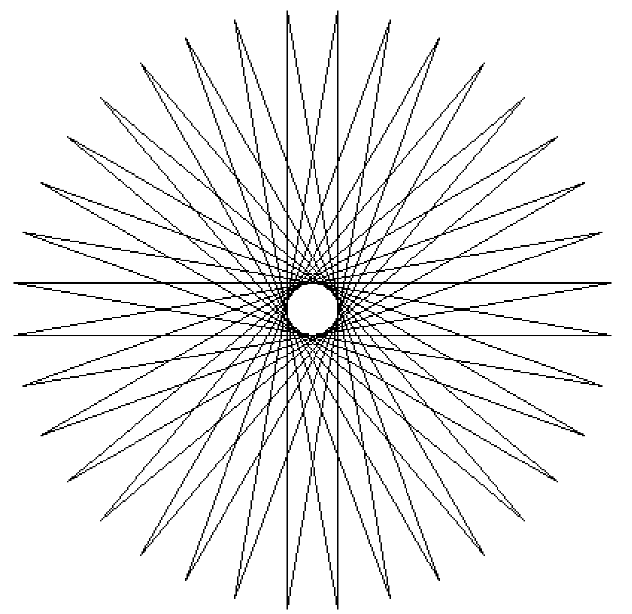
# Turtle Graphics: Using Loops to Draw Designs

- This code draws a sequence of 36 straight lines to make a "starburst" design.

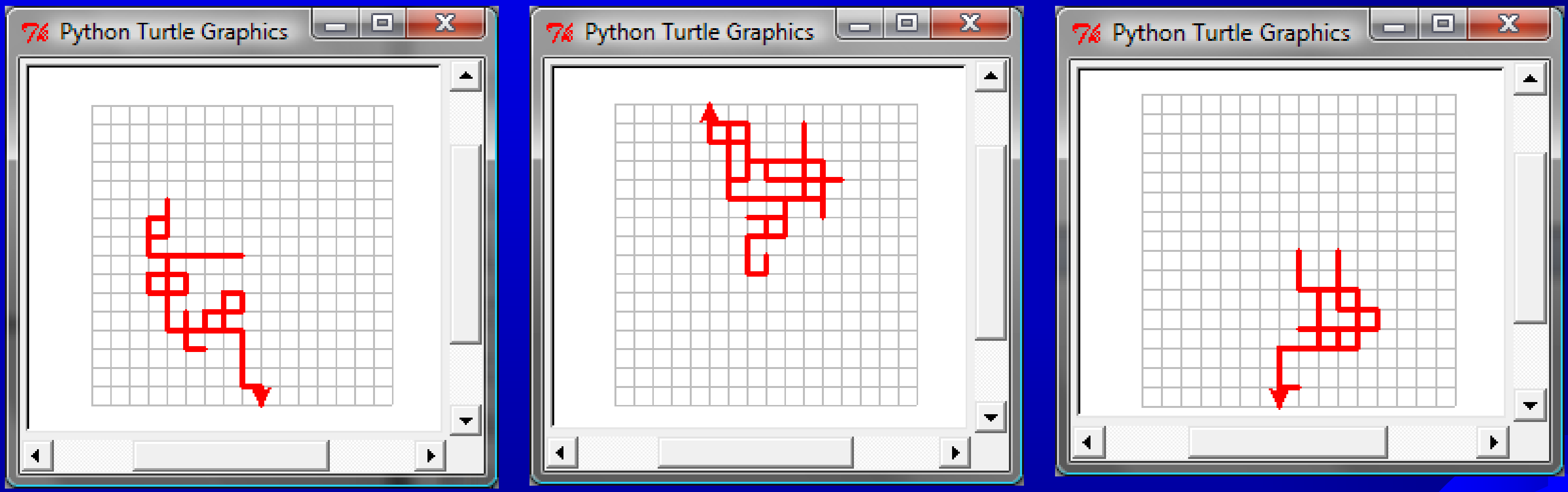
```
import turtle
START_X = -200         # Starting X coordinate
START_Y = 0            # Starting Y coordinate
NUM_LINES = 36         # Number of lines to draw
LINE_LENGTH = 400      # Length of each line
ANGLE = 170            # Angle to turn

turtle.hideturtle()
turtle.penup()
turtle.goto(START_X, START_Y)
turtle.pendown()

for x in range(NUM_LINES):
    turtle.forward(LINE_LENGTH)
    turtle.left(ANGLE)
turtle.done()
```



# Turtle: Random Walk




RandomWalk

Run