# Object Oriented Programming II

Dr. Burak Kaleci

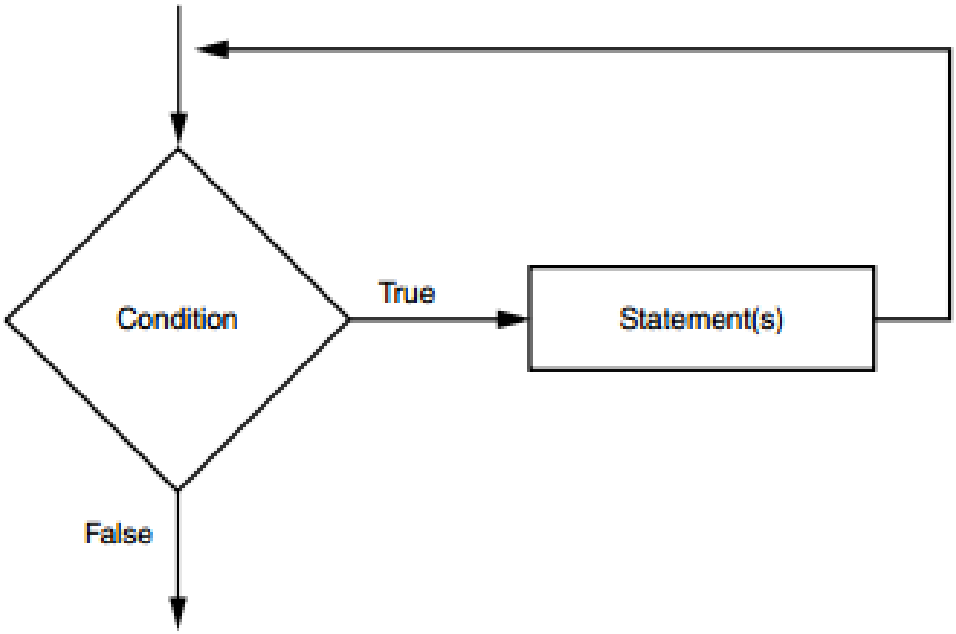
February 15, 2019

# Content

# Introduction

- Programmers commonly have to write code that performs the same task over and over.
- For example, suppose you have been asked to write a program that calculates a 10 percent sales commission for several sales people.
- A **condition-controlled** loop uses a true/false condition to control the number of times that it repeats.
- A **count-controlled** loop repeats a specific number of times.
- In Python, you use the **while** statement to write a **condition-controlled** loop, and you use the **for** statement to write a **count-controlled** loop.

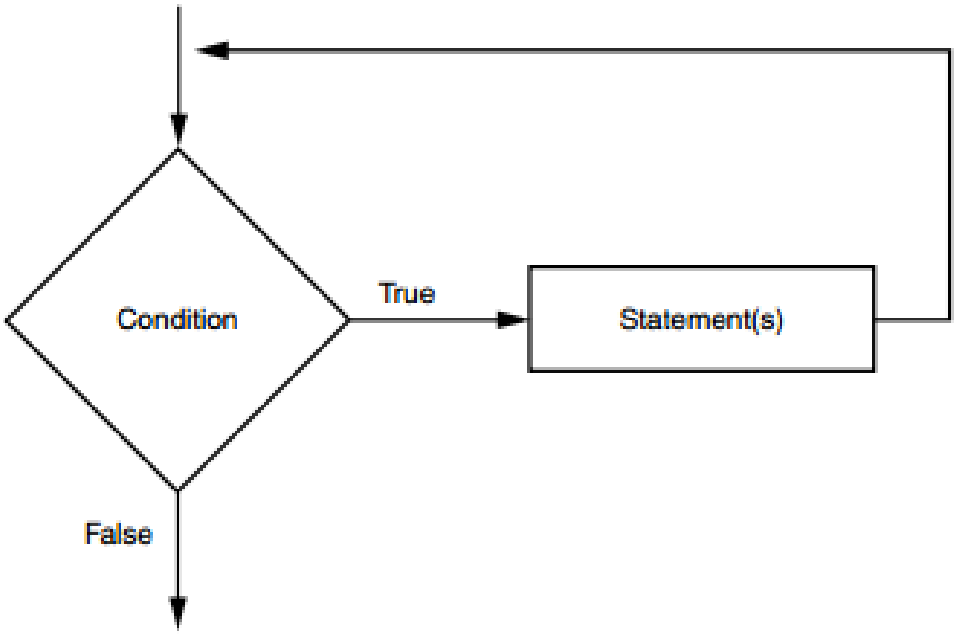
# Introduction




- The **while** loop gets its name from the way it works: while a condition is true, do some task.

- The loop has two parts:
  - a condition that is tested for a true or false value
  - a statement or set of statements that is repeated as long as the condition is true.
- Figure shows the logic of a while loop.
- The diamond symbol represents the condition that is tested.

# Introduction




- In a flowchart, you will always recognize a loop when you see a flow line going back to a previous part of the flowchart.
- Notice what happens if the condition is true: one or more statements are executed, and the program's execution flows back to the point just above the diamond symbol.
- The condition is tested again, and if it is true, the process repeats.
- If the condition is false, the program exits the loop.

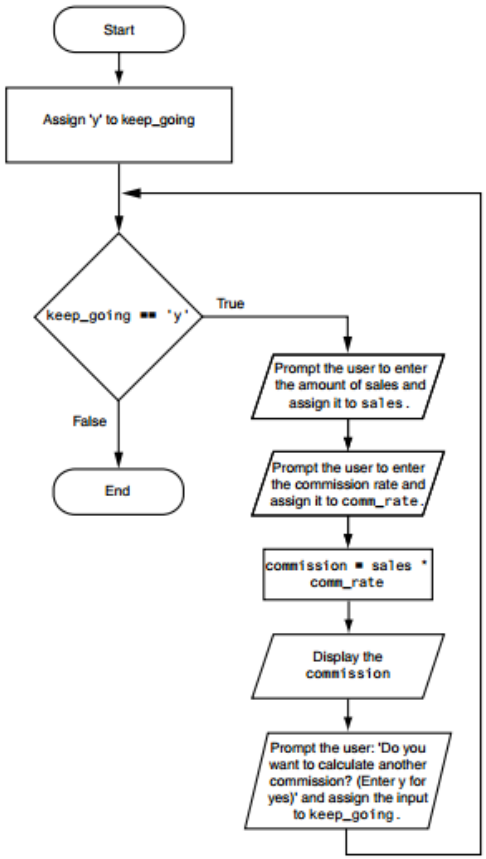
# Introduction

- Here is the general format of the while loop:

```
while condition:
    statement
    statement
    statement
```

- The while statement begins with the word while, followed by a condition, which is an expression that will be evaluated as either true or false.
- A colon appears after the condition.
- Beginning at the next line is a block of statements.
- **All of the statements in a block must be consistently indented.**
- **This indentation is required because the Python interpreter uses it to tell where the block begins and ends.**

# An example




**Problem**:Write a Python program to calculate sales commission for several sales people.

- Prompt a salesperson's sales and commission rate.
- Calculate the commission.
- Display the commission.
- Repeat first three steps until the user wants to exit from the program.

# An example

```
1  # This program calculates sales commissions.
2
3  # Create a variable to control the loop.
4  keep_going = 'y'
5
6  # Calculate a series of commissions.
7  while keep_going == 'y':
8      # Get a salesperson's sales and commission rate.
9      sales = float(input('Enter the amount of sales: '))
10     comm_rate = float(input('Enter the commission rate: '))
11
12     # Calculate the commission.
13     commission = sales * comm_rate
14
15     # Display the commission.
16     print('The commission is $',
17           format(commission, ',.2f'), sep='')
18
19     # See if the user wants to do another one.
20     keep_going = input('Do you want to calculate another ' +
21                        'commission (Enter y for yes): ')
```

- In line 4, we use an assignment statement to create a variable named **keep_going**. Notice the variable is assigned the value 'y'.
- Line 7 is the beginning of a while loop, which starts like this:

  while keep_going == 'y':
- Notice the condition that is being tested:

  keep_going=='y'.

# An example

```
1  # This program calculates sales commissions.
2
3  # Create a variable to control the loop.
4  keep_going = 'y'
5
6  # Calculate a series of commissions.
7  while keep_going == 'y':
8      # Get a salesperson's sales and commission rate.
9      sales = float(input('Enter the amount of sales: '))
10     comm_rate = float(input('Enter the commission rate: '))
11
12     # Calculate the commission.
13     commission = sales * comm_rate
14
15     # Display the commission.
16     print('The commission is $',
17           format(commission, ',.2f'), sep='')
18
19     # See if the user wants to do another one.
20     keep_going = input('Do you want to calculate another ' +
21                        'commission (Enter y for yes): ')
```

- The loop tests this condition, and if it is true, the statements in lines 8 through 21 are executed.
- Then, the loop starts over at line 7. It tests the expression keep_going=='y' and if it is true, the statements in lines 8 through 21 are executed again.
- This cycle repeats until the expression keep_going=='y' is tested in line 7 and found to be false.

# An example

- In order for this loop to stop executing, something has to happen inside the loop to make the expression keep_going== 'y' false.
- The statement in lines 20 through 21 take care of this.
- This statement displays the prompt "Do you want to calculate another commission (Enter y for yes).
- The value that is read from the keyboard is assigned to the keep_going variable.
- If the user enters y (and it must be a lowercase y), then the expression keep_going== 'y' will be true when the loop starts over.
- This will cause the statements in the body of the loop to execute again.
- But if the user enters anything other than lowercase y, the expression will be false when the loop starts over, and the program will exit the loop.

# The while Loop Is a Pretest Loop

- The while loop is known as a pretest loop, which means it tests its condition before performing an iteration.
- Because the test is done at the beginning of the loop, you usually have to perform some steps prior to the loop to make sure that the loop executes at least once.
- For example, the loop in Program starts like this:

  while keep_going == 'y':
- The loop will perform an iteration only if the expression keep_going=='y' is true.

# The while Loop Is a Pretest Loop

- This means that
  - the keep_going variable has to exist,
  - It has to reference the value 'y'.
- To make sure the expression is true the first time that the loop executes, we assigned the value 'y' to the keep_going variable in line 4 as follows:

  keep_going = 'y'
- By performing this step we know that the condition keep_going=='y' will be true the first time the loop executes.
- This is an important characteristic of the while loop: it will never execute if its condition is false to start with.

# Infinite Loops

- In all but rare cases, loops must contain within themselves a way to terminate.
- This means that something inside the loop must eventually make the test condition false.
- The loop in Program stops when the expression keep_going== 'y' is false.
- **If a loop does not have a way of stopping, it is called an infinite loop.**
- An infinite loop continues to repeat until the program is interrupted.
- Infinite loops usually occur when the programmer forgets to write code inside the loop that makes the test condition false.
- In most circumstances, you should avoid writing infinite loops.

# An example

```
1  # This program demonstrates an infinite loop.
2  # Create a variable to control the loop.
3  keep_going = 'y'
4
5  # Warning! Infinite loop!
6  while keep_going == 'y':
7      # Get a salesperson's sales and commission rate.
8      sales = float(input('Enter the amount of sales: '))
9      comm_rate = float(input('Enter the commission rate: '))
10
11     # Calculate the commission.
12     commission = sales * comm_rate
13
14     # Display the commission.
15     print('The commission is $',
16           format(commission, ',.2f'), sep='')
```

- Program demonstrates an infinite loop.
- In this version, we have removed the code that modifies the keep_going variable in the body of the loop.
- Each time the expression keep_going== 'y' is tested in line 6, keep_going will reference the string 'y'.
- As a consequence, the loop has no way of stopping.

# Introduction

- A count-controlled loop iterates a specific number of times.
- Count-controlled loops are commonly used in programs.
- For example, suppose a business is open six days per week, and you are going to write a program that calculates the total sales for a week.
- You will need a loop that iterates exactly six times.
- Each time the loop iterates, it will prompt the user to enter the sales for one day.
- You use the **for** statement to write a count-controlled loop.
- In Python, the for statement is designed to work with a sequence of data items.
- When the statement executes, it iterates once for each item in the sequence.

# Introduction

- Here is the general format of the for loop:

```
for variable in [value1, value2, etc.]:
    statement
    statement
    statement
```

- In the for statement, variable is the name of a variable. Inside the brackets a sequence of values appears, with a comma separating each value.
- A colon appears after the condition.
- Beginning at the next line is a block of statements.
- The for statement executes in the following manner: The variable is assigned the first value in the list, then the statements that appear in the block are executed.
- Then, variable is assigned the next value in the list, and the statements in the block are executed again.
- This continues until variable has been assigned the last value in the list.

# Example 1

```
1  # This program demonstrates a simple for loop
2  # that uses a list of numbers.
3
4  print('I will display the numbers 1 through 5.')
5  for num in [1, 2, 3, 4, 5]:
6      print(num)
```

**Program Output**

```
I will display the numbers 1 through 5.
1
2
3
4
5
```

- The first time the for loop iterates, the num variable is assigned the value 1 and then the statement in line 6 executes (displaying the value 1).
- The next time the loop iterates, num is assigned the value 2, and the statement in line 6 executes (displaying the value 2).
- This process continues until num has been assigned the last value in the list.
- Because the list contains five values, the loop will iterate five times.

# Example 2

**The values that appear in the list do not have to be a consecutively ordered series of numbers.**

```
1  # This program also demonstrates a simple for
2  # loop that uses a list of numbers.
3
4  print('I will display the odd numbers 1 through 9.')
5  for num in [1, 3, 5, 7, 9]:
6      print(num)
```

**Program Output**

```
I will display the odd numbers 1 through 9.
1
3
5
7
9
```

```
1  # This program also demonstrates a simple for
2  # loop that uses a list of strings.
3
4  for name in ['Winken', 'Blinken', 'Nod']:
5      print(name)
```

**Program Output**

```
Winken
Blinken
Nod
```

# Using the range Function with the for Loop

- Python provides a built-in function named **range** that simplifies the process of writing a count-controlled for loop.
- The **range** function creates a type of object known as an iterable.
- An iterable is an object that is similar to a list. It contains a sequence of values that can be iterated over with something like a loop.
- Here is an example of a for loop that uses the range function:

```
for num in range(5):
    print(num)
```

- Notice instead of using a list of values, we call to the range function passing 5 as an argument.
- In this statement, the range function will generate an iterable sequence of integers **in the range of 0 up to (but not including) 5.** This code works the same as the following:

```
for num in [0, 1, 2, 3, 4]:
    print(num)
```

# Using the range Function with the for Loop

- If you pass **one argument** to the range function, that argument is used as the **ending limit** of the sequence of numbers.
- If you pass **two arguments** to the range function, the **first argument** is used as the **starting value** of the sequence, and the **second argument** is used as the **ending limit**.
- Here is an example:

```
for num in range(1, 5):
    print(num)
```

- This code will display the following:

```
1
2
3
4
```

# Using the range Function with the for Loop

- **By default**, the range function produces a sequence of numbers that **increase by 1** for each successive number in the list.
- If you pass a **third argument** to the range function, that argument is used as **step value**.
- Instead of increasing by 1, each successive number in the sequence will increase by the step value.
- Here is an example:

```
for num in range(1, 10,2):
    print(num)
```

# Using the range Function with the for Loop

- In this for statement, three arguments are passed to the range function:
  - The first argument, 1, is the starting value for the sequence.
  - The second argument, 10, is the ending limit of the list. This means that the last number in the sequence will be 9.
  - The third argument, 2, is the step value. This means that 2 will be added to each successive number in the sequence.
- This code will display the following:

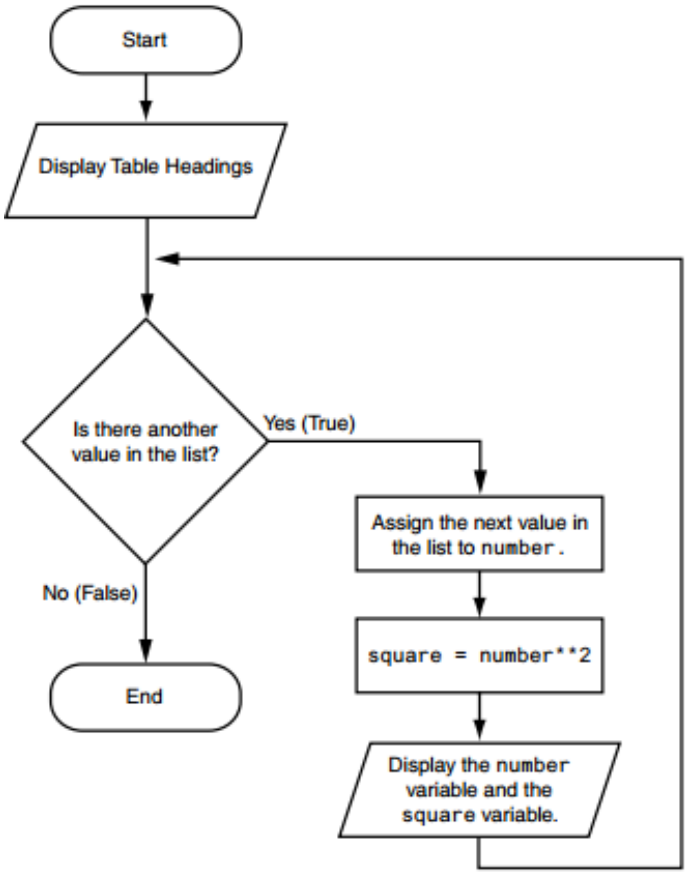
  1
  3
  5
  7
  9

# Using the Control Variable Inside the Loop

- In a for loop, the purpose of the control variable is to reference each item in a sequence of items as the loop iterates.
- In many situations it is helpful to use the control variable in a calculation or other task within the body of the loop.
- For example, suppose you need to write a program that displays the numbers 1 through 10 and their respective squares, in a table similar to the following:

| Number | Square |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |
| 4 | 16 |
| 5 | 25 |
| 6 | 36 |
| 7 | 49 |
| 8 | 64 |
| 9 | 81 |
| 10 | 100 |

# Using the Control Variable Inside the Loop




- Display the table headings.
- Write a for loop that iterates over the values 1 through 10.
- During the first iteration, the control variable will be assigned the value 1, during the second iteration it will be assigned the value 2, and so forth.
- Because the control variable will reference the values 1 through 10 during the loop's execution, you can use it in the calculation inside the loop.
- In each iteration, calculate and display number and its square.

# Using the Control Variable Inside the Loop

```
1  # This program uses a loop to display a
2  # table showing the numbers 1 through 10
3  # and their squares.
4
5  # Print the table headings.
6  print('Number\tSquare')
7  print('--------------')
8
9  # Print the numbers 1 through 10
10 # and their squares.
11 for number in range(1, 11):
12     square = number**2
13     print(number, '\t', square)
```

**Program Output**

```
Number  Square
--------------
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
```

- First, take a closer look at line 6, which displays the table headings:
  print('Number\tSquare')
- Notice inside the string literal, the \t escape sequence between the words Number and Square.
- The for loop that begins in line 11 uses the range function to produce a sequence containing the numbers 1 through 10.
- Inside the loop, the statement in line 12 raises number to the power of 2 and assigns the result to the square variable.
- The statement in line 13 prints the value referenced by number, tabs over, then prints the value referenced by square.

# An Example

```
1  # This program converts the speeds 60 kph
2  # through 130 kph (in 10 kph increments)
3  # to mph.
4
5  START_SPEED = 60              # Starting speed
6  END_SPEED = 131               # Ending speed
7  INCREMENT = 10                # Speed increment
8  CONVERSION_FACTOR = 0.6214    # Conversion factor
9
10 # Print the table headings.
11 print('KPH\tMPH')
12 print('---------------')
13
14 # Print the speeds.
15 for kph in range(START_SPEED, END_SPEED, INCREMENT)
16     mph = kph * CONVERSION_FACTOR
17     print(kph, '\t', format(mph, '.1f'))
```

**Program Output**

```
KPH         MPH
---------------
60          37.3
70          43.5
80          49.7
90          55.9
100         62.1
110         68.4
120         74.6
130         80.8
```

# Another Example

```
1  # This program uses a loop to display a
2  # table of numbers and their squares.
3
4  # Get the starting value.
5  print('This program displays a list of numbers')
6  print('and their squares.')
7  start = int(input('Enter the starting number: '))
8
9  # Get the ending limit.
10 end = int(input('How high should I go? '))
11
12 # Print the table headings.
13 print()
14 print('Number\tSquare')
15 print('--------------')
16
17 # Print the numbers and their squares.
18 for number in range(start, end + 1):
19     square = number**2
20     print(number, '\t', square)
```

**Program Output** (with input shown in bold)

```
This program displays a list of numbers and their squ
Enter the starting number: 5 Enter
How high should I go? 10 Enter

Number  Square
--------------
5       25
6       36
7       49
8       64
9       81
10      100
```

# Generating an Iterable Sequence that Ranges from Highest to Lowest

- In the examples you have seen so far, the range function was used to generate a sequence with numbers that go from lowest to highest.
- Alternatively, you can use the range function to generate sequences of numbers that go from highest to lowest.
- Here is an example:

  range(10, 0, -1)

- In this function call, the starting value is 10, the sequence's ending limit is 0, and the step value is -1.
- This expression will produce the following sequence:

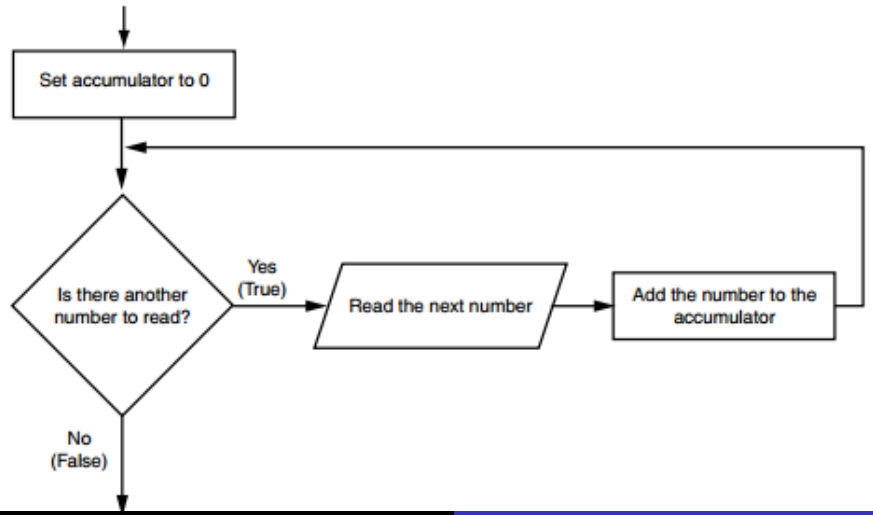
  10, 9, 8, 7, 6, 5, 4, 3, 2, 1

# Calculating a Running Total

- Many programming tasks require you to calculate the total of a series of numbers.
- For example, suppose you are writing a program that calculates a business's total sales for a week.
- The program would read the sales for each day as input and calculate the total of those numbers.
- Programs that calculate the total of a series of numbers typically use two elements
  - A loop that reads each number in the series
  - A variable that accumulates the total of the numbers as they are read
- The variable that is used to accumulate the total of the numbers is called an **accumulator**.

# Calculating a Running Total

- When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop.
- **Notice the first step in the flowchart is to set the accumulator variable to 0.** This is a critical step.
- Each time the loop reads a number, it adds it to the accumulator.
- If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.



# Calculating a Running Total

```
1  # This program calculates the sum of a series
2  # of numbers entered by the user.
3
4  MAX = 5 # The maximum number
5
6  # Initialize an accumulator variable.
7  total = 0.0
8
9  # Explain what we are doing.
10 print('This program calculates the sum of')
11 print(MAX, 'numbers you will enter.')
12
13 # Get the numbers and accumulate them.
14 for counter in range(MAX):
15     number = int(input('Enter a number: '))
16     total = total + number
17
18 # Display the total of the numbers.
19 print('The total is', total)
```

**Program Output** (with input shown in bold)

```
This program calculates the sum of
5 numbers you will enter.
Enter a number: 1 Enter
Enter a number: 2 Enter
Enter a number: 3 Enter
Enter a number: 4 Enter
Enter a number: 5 Enter
The total is 15.0
```

- The total variable, created by the assignment statement in line 7, is the accumulator.
- Notice it is initialized with the value 0.0.
- The for loop, in lines 14 through 16, does the work of getting the numbers from the user and calculating their total.
- Line 15 prompts the user to enter a number then assigns the input to the number variable.

# Calculating a Running Total

```
1  # This program calculates the sum of a series
2  # of numbers entered by the user.
3
4  MAX = 5 # The maximum number
5
6  # Initialize an accumulator variable.
7  total = 0.0
8
9  # Explain what we are doing.
10 print('This program calculates the sum of')
11 print(MAX, 'numbers you will enter.')
12
13 # Get the numbers and accumulate them.
14 for counter in range(MAX):
15     number = int(input('Enter a number: '))
16     total = total + number
17
18 # Display the total of the numbers.
19 print('The total is', total)
```

**Program Output** (with input shown in bold)

```
This program calculates the sum of
5 numbers you will enter.
Enter a number: 1 Enter
Enter a number: 2 Enter
Enter a number: 3 Enter
Enter a number: 4 Enter
Enter a number: 5 Enter
The total is 15.0
```

- Then, the following statement in line 16 adds number to total:

$$\text{total} = \text{total} + \text{number}$$

- After this statement executes, the value referenced by the number variable will be added to the value in the total variable.
- When the loop finishes, the total variable will hold the sum of all the numbers that were added to it. This value is displayed in line 19.

# The Augmented Assignment Operators

- Quite often, programs have assignment statements in which the variable that is on the left side of the = operator also appears on the right side of the = operator.
- Here is an example:

$$x = x + 1$$

- On the right side of the assignment operator, 1 is added to x.
- The result is then assigned to x, replacing the value that x previously referenced.
- Effectively, this statement adds 1 to x.

# The Augmented Assignment Operators

- These types of operations are common in programming. For convenience, Python offers a special set of operators designed specifically for these jobs.
- As you can see, the augmented assignment operators do not require the programmer to type the variable name twice.

| Operator | Example Usage | Equivalent To |
|---|---|---|
| += | x += 5 | x = x + 5 |
| -= | y -= 2 | y = y - 2 |
| *= | z *= 10 | z = z * 10 |
| /= | a /= b | a = a / b |
| %= | c %= 3 | c = c % 3 |

# Introduction

- Consider the following scenario: You are designing a program that will use a loop to process a long sequence of values.
- At the time you are designing the program, **you do not know the number** of values that will be in the sequence.
- **In fact, the number of values in the sequence could be different each time the program is executed.**
- What is the best way to design such a loop?

# Introduction

- Simply ask the user, at the end of each loop iteration, if there is another value to process.
- If the sequence of values is long, however, asking this question at the end of each loop iteration might make the program cumbersome for the user.
- Ask the user at the beginning of the program how many items are in the sequence.
- This might also inconvenience the user, however. If the sequence is very long, and the user does not know the number of items it contains, it will require the user to count them.

# Introduction

- When processing a long sequence of values with a loop, perhaps a better technique is to use a **sentinel**.
- **A sentinel is a special value that marks the end of a sequence of items.** When a program reads the sentinel value, it knows it has reached the end of the sequence, so the loop terminates.
- For example, suppose a doctor wants a program to calculate the average weight of all her patients.
- The program might work like this: A loop prompts the user to enter either a patient's weight, or 0 if there are no more weights.

# Introduction

- When the program reads 0 as a weight, it interprets this as a signal that there are no more weights. The loop ends and the program displays the average weight.
- **A sentinel value must be distinctive enough that it will not be mistaken as a regular value in the sequence.**
- In the example cited above, the doctor enters 0 to signal the end of the sequence of weights. Because no patient's weight will be 0, this is a good value to use as a sentinel.

# Introduction

```
1  # This program displays gross pay.
2  # Get the number of hours worked.
3  hours = int(input('Enter the hours worked this week: '))
4
5  # Get the hourly pay rate.
6  pay_rate = float(input('Enter the hourly pay rate: '))
7
8  # Calculate the gross pay.
9  gross_pay = hours * pay_rate
10
11 # Display the gross pay.
12 print('Gross pay: $', format(gross_pay, ',.2f'))
```

**Program Output** (with input shown in bold)

```
Enter the hours worked this week: 400 Enter
Enter the hourly pay rate: 20 Enter
The gross pay is $8,000.00
```

- If a user provides bad data as input to a program, the program will process that bad data and, as a result, will produce bad data as output.
- For example, look at the payroll program and notice what happens in the sample run when the user gives bad data as input.
- The person receiving the pay check will be pleasantly surprised, because in the sample run the payroll clerk entered 400 as the number of hours worked.
- The clerk probably meant to enter 40, because there are not 400 hours in a week.

# Introduction

```
1  # This program displays gross pay.
2  # Get the number of hours worked.
3  hours = int(input('Enter the hours worked this week: '))
4
5  # Get the hourly pay rate.
6  pay_rate = float(input('Enter the hourly pay rate: '))
7
8  # Calculate the gross pay.
9  gross_pay = hours * pay_rate
10
11 # Display the gross pay.
12 print('Gross pay: $', format(gross_pay, ',.2f'))
```

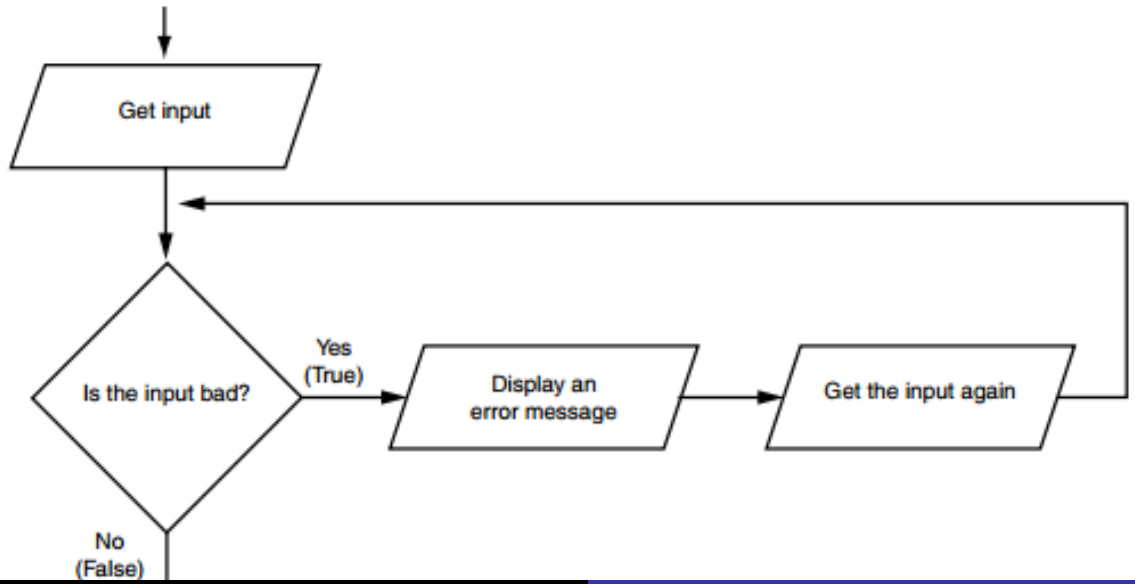
**Program Output** (with input shown in bold)

```
Enter the hours worked this week: 400 Enter
Enter the hourly pay rate: 20 Enter
The gross pay is $8,000.00
```

- The computer, however, is unaware of this fact, and the program processed the bad data just as if it were good data. hours in a week.
- The integrity of a program's output is only as good as the integrity of its input.
- For this reason, you should design your programs in such a way that bad input is never accepted.
- When input is given to a program, it should be inspected before it is processed. If the input is invalid, the program should discard it and prompt the user to enter the correct data.
- **This process is known as input validation.**

# Input Validation

- Figure shows a common technique for validating an item of input. In this technique, the input is read, then a loop is executed.
- If the input data is bad, the loop executes its block of statements.
- The loop displays an error message so the user will know that the input was invalid, and then it reads the new input. The loop repeats as long as the input is bad.



# Input Validation

- Notice the flowchart reads input in two places: first just before the loop, and then inside the loop.
- The first input operation -just before the loop- is called a **priming read**,and its purpose is to get the first input value that will be tested by the validation loop.
- If that value is invalid, the loop will perform subsequent input operations.
- Let's consider an example. Suppose you are designing a program that reads a test score and you want to make sure the user does not enter a value less than 0.

```
8  # Get a test score.
9  score = int(input('Enter a test score: '))
10
11 # Make sure it is not less than 0.
12 while score < 0:
13     print('ERROR: The score cannot be negative.')
14     score = int(input('Enter the correct score: '))
```

# An Example

```
1  # This program calculates retail prices.
2
3  MARK_UP = 2.5 # The markup percentage
4  another = 'y' # Variable to control the loop.
5
6  # Process one or more items.
7  while another == 'y' or another == 'Y':
8      # Get the item's wholesale cost.
9      wholesale = float(input("Enter the item's " +
10                             "wholesale cost: "))
11
12     # Validate the wholesale cost.
13     while wholesale < 0:
14         print('ERROR: the cost cannot be negative.')
15         wholesale = float(input('Enter the correct' +
16                                 'wholesale cost: '))
17
18     # Calculate the retail price.
19     retail = wholesale * MARK_UP
20
21     # Display the retail price.
22     print('Retail price: $', format(retail, ',.2f'), sep='')
23
24
25     # Do this again?
26     another = input('Do you have another item? ' +
27                     '(Enter y for yes): ')
```

**Program Output (with input shown in bold)**

```
Enter the item's wholesale cost: -.50 Enter
ERROR: the cost cannot be negative.
Enter the correct wholesale cost: 0.50 Enter
Retail price: $1.25.
Do you have another item? (Enter y for yes): n Enter
```

# An Example

```
1  # This program averages test scores. It asks the user for the
2  # number of students and the number of test scores per student.
3
4  # Get the number of students.
5  num_students = int(input('How many students do you have? '))
6
7  # Get the number of test scores per student.
8  num_test_scores = int(input('How many test scores per student? '))
9
10 # Determine each student's average test score.
11 for student in range(num_students):
12     # Initialize an accumulator for test scores.
13     total = 0.0
14     # Get a student's test scores.
15     print('Student number', student + 1)
16     print('-----------------')
17     for test_num in range(num_test_scores):
18         print('Test number', test_num + 1, end='')
19         score = float(input(': '))
20         # Add the score to the accumulator.
21         total += score
22
23     # Calculate the average test score for this student.
24     average = total / num_test_scores
25
26     # Display the average.
27     print('The average for student number', student + 1,
28           'is:', average)
29     print()
```

# Example 1

**Problem**: Reverse a number.

**Example**: Assume that the number is 54321.

Reverse of the number is 12345

```
 8 # Get a test score.
 9 num = int(input('Enter a number: '))
10 rvs=0
11
12 # Make sure it is not less than 0.
13 while num!= 0:
14     remainder=num%10
15     rvs=rvs*10+remainder
16     num=num//10
17
18 print("Reverse is ", rvs)
```

# Example 2

In number theory, a **perfect number** is a **positive integer** that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself.

**Example**: The first perfect number is 6.

Its proper divisors are 1, 2, and 3, and $1 + 2 + 3 = 6$.

The next perfect number is $28 = 1 + 2 + 4 + 7 + 14$.

```
 9 num = int(input('Enter a number: '))
10 sum=0
11 i=1
12
13 # Make sure it is not less than 0.
14 while i<num:
15     if num%i==0:
16         sum=sum+i
17     i=i+1
18 if sum==num:
19     print(num, "is perfect number")
20 else:
21     print(num, "is not perfect number")
```