

Repetition Structures

陳建良



Introduction to Repetition Structures

- 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 salespeople.
- Although it would not be a good design, one approach would be to write the code to calculate one salesperson's commission, and then repeat that code for each salesperson.

```
# Get a salesperson's sales and commission rate.
sales = float(input('Enter the amount of sales: '))
comm_rate = float(input('Enter the commission rate: '))
# Calculate the commission.
commission = sales * comm_rate
# Display the commission.
print('The commission is $', format(commission, ',.2f'), sep='')

# Get another salesperson's sales and commission rate.
sales = float(input('Enter the amount of sales: '))
comm_rate = float(input('Enter the commission rate: '))
# Calculate the commission.
commission = sales * comm_rate
# Display the commission.
print('The commission is $', format(commission, ',.2f'), sep='')

# Get another salesperson's sales and commission rate.
sales = float(input('Enter the amount of sales: '))
comm_rate = float(input('Enter the commission rate: '))
# Calculate the commission.
commission = sales * comm_rate
# Display the commission.
print('The commission is $', format(commission, ',.2f'), sep='')
```

#And this code goes on and on . . .



- As you can see, this code is one long sequence structure containing a lot of duplicated code. There are several disadvantages to this approach, including the following:
 - The duplicated code makes the program large.
 - Writing a long sequence of statements can be time consuming.
 - If part of the duplicated code has to be corrected or changed, then the correction or change has to be done many times.
- This can be done with a *repetition structure*, which is more commonly known as a *loop*.



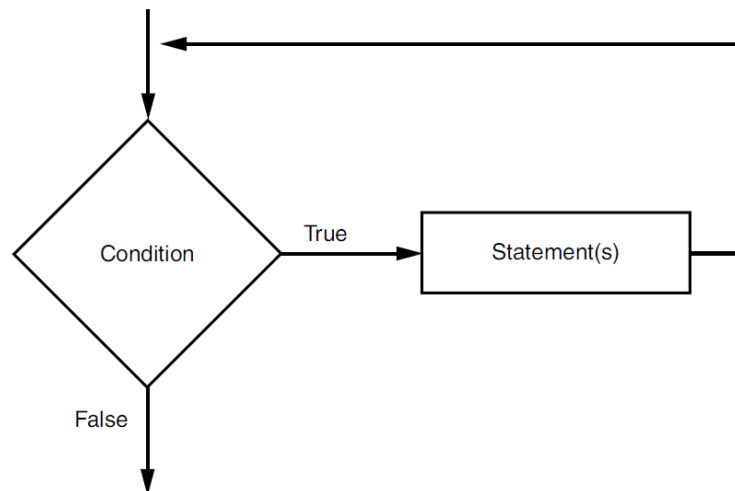
Condition-Controlled and Count-Controlled Loops

- We will look at two broad categories of loops: condition-controlled and count-controlled.
- 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.



The **while** Loops: A Condition-Controlled Loop

- 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.



■ Here is the general format of the while loop in Python:

while condition:

statement

statement

etc.



```
# This program calculates sales commissions.

# Create a variable to control the loop.
keep_going = 'y'

# Calculate a series of commissions.
while keep_going == 'y':
    # Get a salesperson's sales and commission rate.
    sales = float(input('Enter the amount of sales: '))
    comm_rate = float(input('Enter the commission rate: '))
    # Calculate the commission.
    commission = sales * comm_rate
    # Display the commission.
    print('The commission is $',
          format(commission, ',.2f'), sep='')
    # See if the user wants to do another one.
    keep_going = input('Do you want to calculate another ' +
                       'commission (Enter y for yes): ')

```

Program Output (with input shown in bold)

```
Enter the amount of sales: 10000.00 
Enter the commission rate: 0.10 
The commission is $1,000.00
Do you want to calculate another commission (Enter y for yes): y 
Enter the amount of sales: 20000.00 
Enter the commission rate: 0.15 
The commission is $3,000.00
Do you want to calculate another commission (Enter y for yes): y 
Enter the amount of sales: 12000.00 
Enter the commission rate: 0.10 
The commission is $1,200.00
Do you want to calculate another commission (Enter y for yes): n 
```



This condition is tested.

```
while keep_going == 'y':
```

If the condition is true,
these statements are
executed, and then the
loop starts over.

If the condition is false,
these statements are
skipped, and the
program exits the loop.

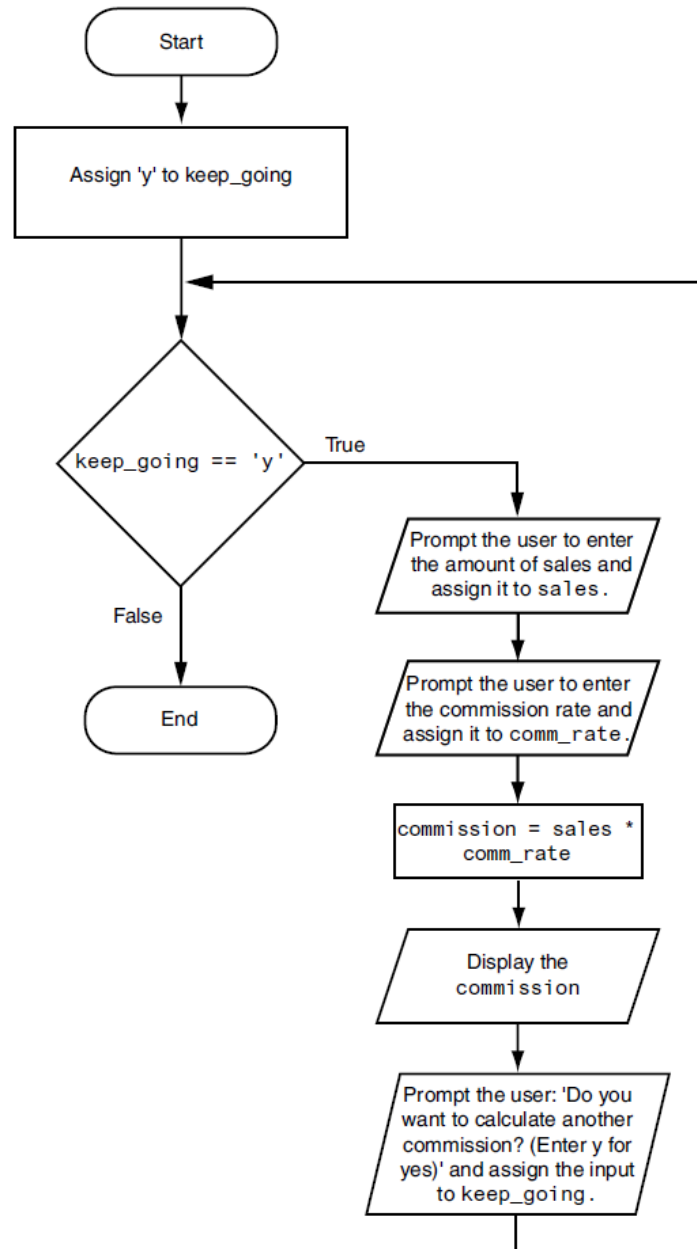
```
# Get a salesperson's sales and commission rate.
sales = float(input('Enter the amount of sales: '))
comm_rate = float(input('Enter the commission rate: '))

# Calculate the commission.
commission = sales * comm_rate

# Display the commission.
print('The commission is $',
      format(commission, ',.2f'), sep='')

# See if the user wants to do another one.
keep_going = input('Do you want to calculate another ' +
                   'commission (Enter y for yes): ')
```





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.



■ A project currently underway at Chemical Labs, Inc. requires that a substance be continually heated in a vat. A technician must check the substance's temperature every 15 minutes. If the substance's temperature does not exceed 102.5 degrees Celsius, then the technician does nothing. However, if the temperature is greater than 102.5 degrees Celsius, the technician must turn down the vat's thermostat, wait 5 minutes, and check the temperature again. The technician repeats these steps until the temperature does not exceed 102.5 degrees Celsius. The director of engineering has asked you to write a program that guides the technician through this process.

■ Here is the algorithm:

1. Get the substance's temperature.
2. Repeat the following steps as long as the temperature is greater than 102.5 degrees Celsius:
 - a. Tell the technician to turn down the thermostat, wait 5 minutes, and check the temperature again.
 - b. Get the substance's temperature.
3. After the loop finishes, tell the technician that the temperature is acceptable and to check it again in 15 minutes.

Program Output (with input shown in bold)

Enter the substance's Celsius temperature: **104.7**

The temperature is too high.

Turn the thermostat down and wait
5 minutes. Take the temperature
again and enter it.

Enter the new Celsius temperature: **103.2**

The temperature is too high.

Turn the thermostat down and wait
5 minutes. Take the temperature
again and enter it.

Enter the new Celsius temperature: **102.1**

The temperature is acceptable.

Check it again in 15 minutes.



Infinite Loops

- loops must contain within themselves a way to terminate. This means that something inside the loop **must eventually make the test condition false**.
- If a loop does not have a way of stopping, it is called an infinite loop.
- Infinite loops usually occur when the programmer forgets to write code inside the loop that makes the test condition false.



```
# This program demonstrates an infinite loop.
# Create a variable to control the loop.
keep_going = 'y'

# Warning! Infinite loop!
while keep_going == 'y':
    # Get a salesperson's sales and commission rate.
    sales = float(input('Enter the amount of sales: '))
    comm_rate = float(input('Enter the commission rate: '))

    # Calculate the commission.
    commission = sales * comm_rate

    # Display the commission.
    print('The commission is $',
          format(commission, ',.2f'), sep='')

```



The **for** Loop: A Count-Controlled Loop

- A count-controlled loop iterates a specific **number of times**. Count-controlled loops are commonly used in programs.
- 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.
- Here is the general format:

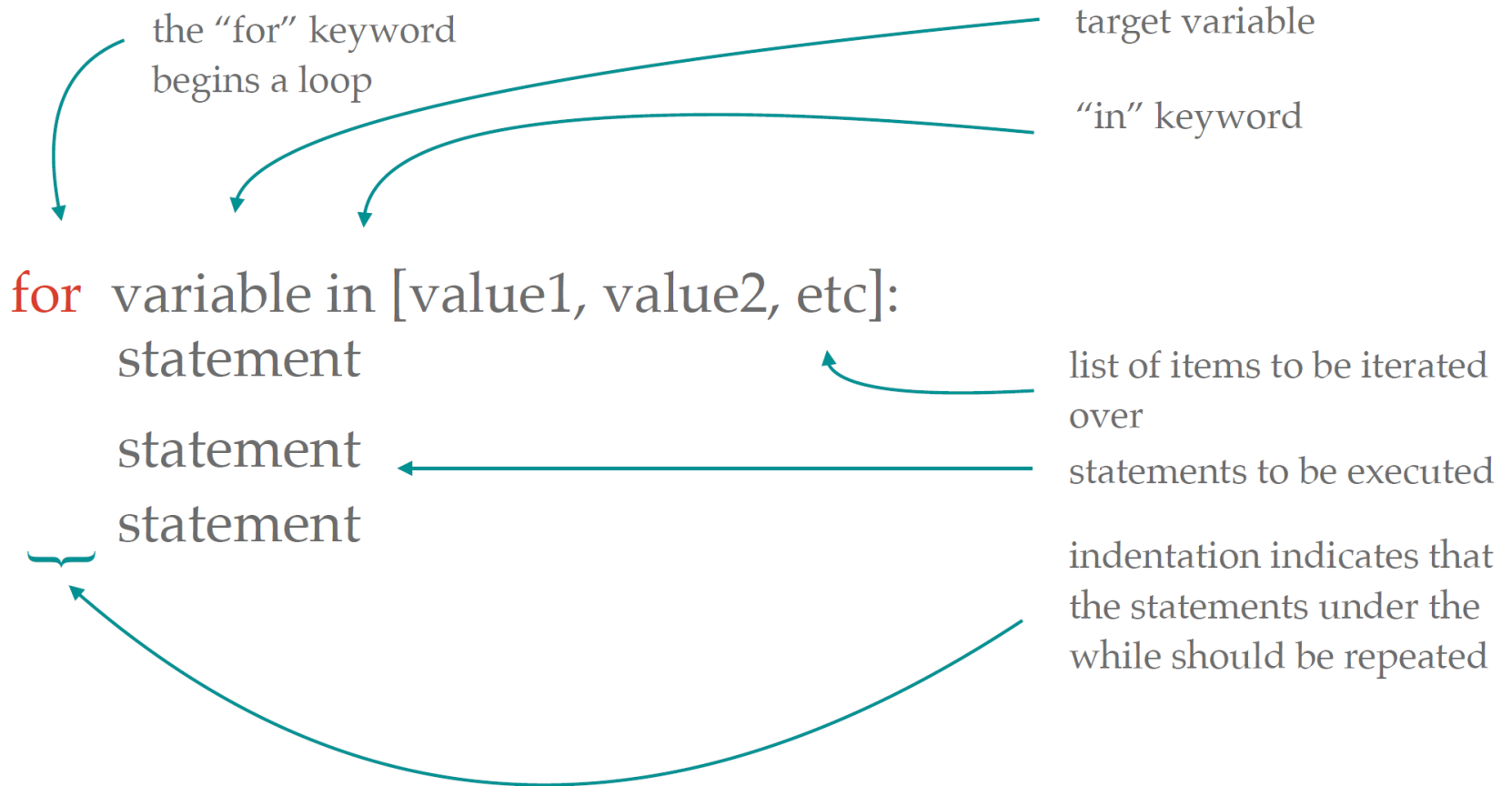
for variable in [value1, value2, etc.]:

statement

statement

etc.






```
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
```




1st iteration: `for num in [1, 2, 3, 4, 5]:`
 `print(num)`




2nd iteration: `for num in [1, 2, 3, 4, 5]:`
 `print(num)`




3rd iteration: `for num in [1, 2, 3, 4, 5]:`
 `print(num)`



4th iteration: `for num in [1, 2, 3, 4, 5]:`
 `print(num)`



5th iteration: `for num in [1, 2, 3, 4, 5]:`
 `print(num)`



```
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)
```

- The above code works the same as the following:

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



```
1 # This program demonstrates how the range
2 # function can be used with a for loop.
3
4 # Print a message five times.
5 for x in range(5):
6     print('Hello world')
```

Program Output

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

- 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)
```

- If you pass a third argument to the range function, that argument is used as *step value*.

- Here is an example:

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



Using the Target Variable Inside the Loop

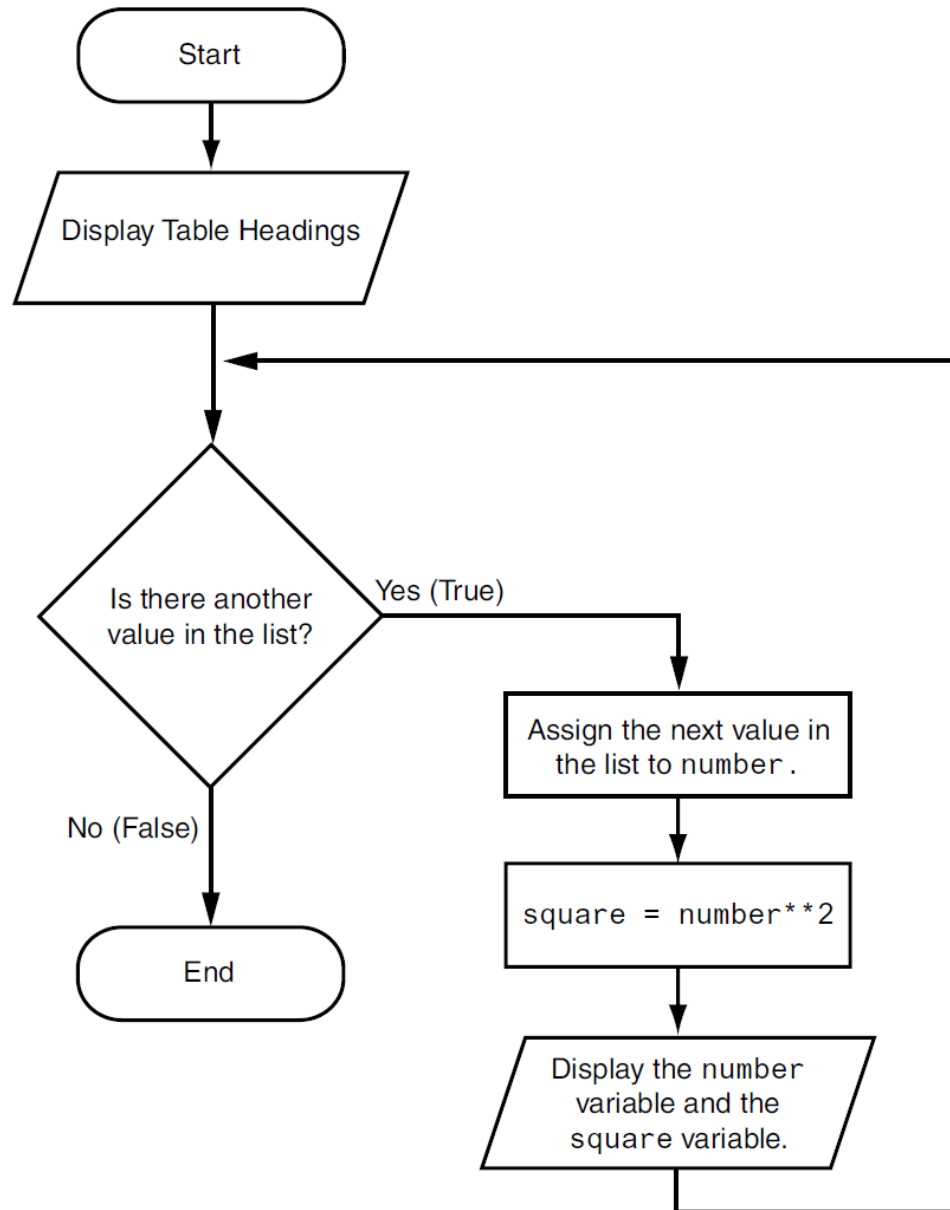
```
1 # This program uses a loop to display a
2 # table showing the numbers 1 through 10
3 # and their squares.
4
5 # Print the table headings.
6 print('Number\tSquare')
7 print('-----')
8
9 # Print the numbers 1 through 10
10 # and their squares.
11 for number in range(1, 11):
12     square = number**2
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





- Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car's speedometer indicates kilometers per hour (KPH). She has asked you to write a program that displays a table of speeds in KPH with their values converted to miles per hour (MPH). The formula for converting KPH to MPH is:

$$MPH = KPH * 0.6214$$

- In the formula, *MPH* is the speed in miles per hour, and *KPH* is the speed in kilometers per hour.
- The table that your program displays should show speeds from 60 KPH through 130 KPH, in increments of 10, along with their values converted to MPH. The table should look something like this:

KPH	MPH
60	37.3
70	43.5
80	49.7
<i>etc. . . .</i>	
130	80.8

- After thinking about this table of values, you decide that you will write a for loop. The list of values that the loop will iterate over will be the kilometer-per-hour speeds. In the loop, you will call the range function like this:

`range(60, 131, 10)`



Letting the User Control the Loop Iterations

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

Program Output (with input shown in bold)

This program displays a list of numbers
(starting at 1) and their squares.
How high should I go? **5**

Number	Square
1	1
2	4
3	9
4	16
5	25



- Please write a program that allows the user to specify both the starting value and the ending limit of the sequence.

Program Output (with input shown in bold)

This program displays a list of numbers and their squares.

Enter the starting number: **5**

How high should I go? **10**

Number	Square
--------	--------

5	25
6	36
7	49
8	64
9	81
10	100



Generating an Iterable Sequence that Ranges from Highest to Lowest

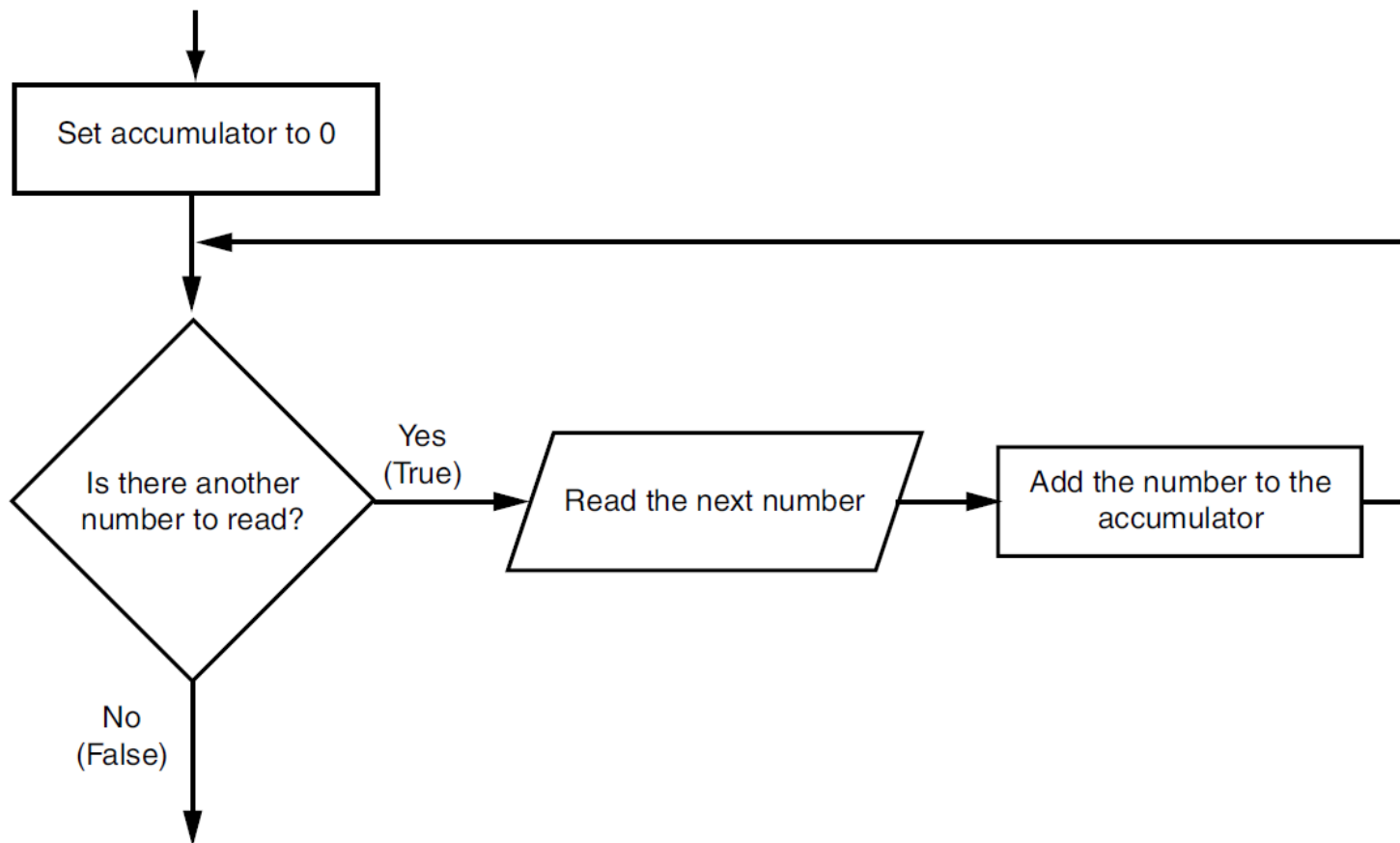
- The range function was used to generate a sequence with numbers that go from lowest to highest.
- You can use the range function to generate sequences of numbers that go from highest to lowest.


`range(10, 0, -1)`



Calculating a Running Total

- Many programming tasks require you to calculate the total of a series of 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**.
- It is often said that the loop keeps a **running total** because it accumulates the total as it reads each number in the series.





```
# This program calculates the sum of a series  
# of numbers entered by the user.
```

```
MAX = 5 # The maximum number
```

```
# Initialize an accumulator variable.  
total = 0.0
```

```
# Explain what we are doing.  
print('This program calculates the sum of')  
print(MAX, 'numbers you will enter.')
```

```
# Get the numbers and accumulate them.  
for counter in range(MAX):  
    number = int(input('Enter a number: '))  
    total = total + number
```

```
# Display the total of the numbers.  
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 a number: **2**

Enter a number: **3**

Enter a number: **4**

Enter a number: **5**

The total is 15.0

The Augmented Assignment Operators

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



Sentinels

- A sentinel is a special value that marks the end of a sequence of values.

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?



- 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.
- 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.



- The county tax office calculates the annual taxes on property using the following formula:

$$\text{property tax} = \text{property value} \times 0.0065$$

- Every day, a clerk in the tax office gets a list of properties and has to calculate the tax for each property on the list. You have been asked to design a program that the clerk can use to perform these calculations.
- In your interview with the tax clerk, you learn that each property is assigned a lot number, and all lot numbers are 1 or greater.
- You decide to write a loop that uses the number 0 as a sentinel value. During each loop iteration, the program will ask the clerk to enter either a property's lot number, or 0 to end.



Program Output (with input shown in bold)

Enter the property lot number
or enter 0 to end.

Lot number: **100**

Enter the property value: **100000.00**

Property tax: \$650.00.

Enter the next lot number or
enter 0 to end.

Lot number: **200**

Enter the property value: **5000.00**

Property tax: \$32.50.

Enter the next lot number or
enter 0 to end.

Lot number: **0**

```
# Get the first lot number.
print('Enter the property lot number')
print('or enter 0 to end.')
lot = int(input('Lot number: '))

# Continue processing as long as the user
# does not enter lot number 0.
while lot != 0:
    # Get the property value.
    value = float(input('Enter the property value: '))

    # Calculate the property's tax.
    tax = value * TAX_FACTOR

    # Display the tax.
    print('Property tax: $', format(tax, ',.2f'), sep='')

    # Get the next lot number.
    print('Enter the next lot number or')
    print('enter 0 to end.')
    lot = int(input('Lot number: '))
```



Input Validation Loops

- 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.
- Input validation is the process of inspecting data that has been input to a program, to make sure it is valid before it is used in a computation.
- Input validation is commonly done with a loop that iterates as long as an input variable references bad data.




```
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)

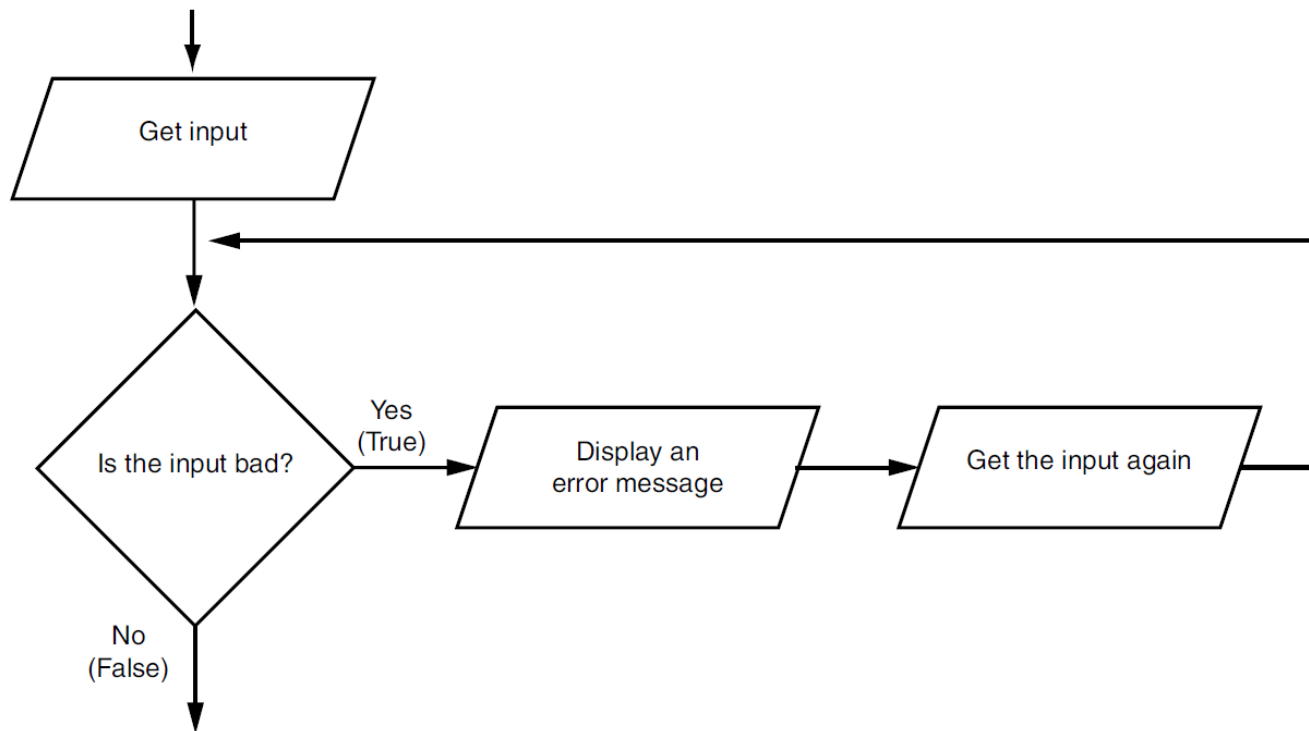
Enter the hours worked this week: **400**

Enter the hourly pay rate: **20**

The gross pay is \$8,000.00



- 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*.



```
# Get a test score.  
score = int(input('Enter a test score: '))  
# Make sure it is not less than 0 or greater than 100.  
while score < 0 or score > 100:  
    print('ERROR: The score cannot be negative')  
    print('or greater than 100.')  
    score = int(input('Enter the correct score: '))
```

- An input validation loop is sometimes called an *error trap* or an *error handler*.



- Samantha owns an import business, and she calculates the retail prices of her products with the following formula:

$$\text{retail price} = \text{wholesale cost} \times 2.5$$

- She currently uses the program to calculate retail prices.

Program Output (with input shown in bold)

Enter the item's wholesale cost: **10.00**

Retail price: \$25.00.

Do you have another item? (Enter y for yes): **y**

Enter the item's wholesale cost: **15.00**

Retail price: \$37.50.

Do you have another item? (Enter y for yes): **y**

Enter the item's wholesale cost: **12.50**

Retail price: \$31.25.

Do you have another item? (Enter y for yes): **n**



- Samantha has encountered a problem when using the program, She has asked you to modify the program so it will not allow a negative number to be entered for the wholesale cost.

Program Output (with input shown in bold)

Enter the item's wholesale cost: **-.50**

ERROR: the cost cannot be negative.

Enter the correct wholesale cost: **0.50**

Retail price: \$1.25.

Do you have another item? (Enter y for yes): **n**



Nested Loops

- A loop that is inside another loop is called a nested loop.
- The innermost loop will iterate through all its iterations for every single iteration of an outer loop.
- Inner loops complete their iterations faster than outer loops.
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.



```
for seconds in range(60):  
    print(seconds)  
for minutes in range(60):  
    for seconds in range(60):  
        print(minutes, ':', seconds)  
for hours in range(24):  
    for minutes in range(60):  
        for seconds in range(60):  
            print(hours, ':', minutes, ':', seconds)
```



Write a program that a teacher might use to get the average of each student's test scores.

Program Output (with input shown in bold)

```
How many students do you have? 3  Enter
How many test scores per student? 3  Enter

Student number 1
-----
Test number 1: 100  Enter
Test number 2: 95  Enter
Test number 3: 90  Enter
The average for student number 1 is: 95.0

Student number 2
-----
Test number 1: 80  Enter
Test number 2: 81  Enter
Test number 3: 82  Enter
The average for student number 2 is: 81.0

Student number 3
-----
Test number 1: 75  Enter
Test number 2: 85  Enter
Test number 3: 80  Enter
The average for student number 3 is: 80.0
```

```
# This program averages test scores. It asks the user for the
# number of students and the number of test scores per student.

# Get the number of students.
num_students = int(input('How many students do you have? '))

# Get the number of test scores per student.
num_test_scores = int(input('How many test scores per student? '))

# Determine each student's average test score.
for student in range(num_students):
    # Initialize an accumulator for test scores.
    total = 0.0
    # Get a student's test scores.
    print('Student number', student + 1)
    print('-----')
    for test_num in range(num_test_scores):
        print('Test number', test_num + 1, end='')
        score = float(input(': '))
        # Add the score to the accumulator.
        total += score

    # Calculate the average test score for this student.
    average = total / num_test_scores

    # Display the average.
    print('The average for student number', student + 1,
          'is:', average)
    print()
```



- Suppose we want to print asterisks on the screen in the righthand side rectangular pattern:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

- The following code can be used to display one row of asterisks:

```
for col in range(6):
    print('*', end="")
```

- We can place the loop inside another loop that iterates eight times

```
for row in range(8):
    for col in range(6):
        print('*', end="")
    print()
```



- Please re-write a program that prompts the user for the number of rows and columns.

Program Output (with input shown in bold)

How many rows? **5**

How many columns? **10**



- Please write a program to print asterisks in a pattern that looks like the following triangle:

```
*  
* *  
* * *  
* * * *  
* * * * *  
* * * * * *  
* * * * * * *  
* * * * * * * *
```

```
# This program displays a triangle pattern.  
BASE_SIZE = 8  
  
for r in range(BASE_SIZE):  
    for c in range(r + 1):  
        print('*', end='')  
    print()
```



- Please write a program to display the following stair-step pattern:

```
#
 #
  #
   #
    #
     #
```

```
# This program displays a stair-step pattern.
NUM_STEPS = 6

for r in range(NUM_STEPS):
    for c in range(r):
        print(' ', end='')
    print('#')
```



Turtle Graphics: Using Loops to Draw Designs

- You can use loops to draw graphics that range in complexity from simple shapes to elaborate designs.

- For example,

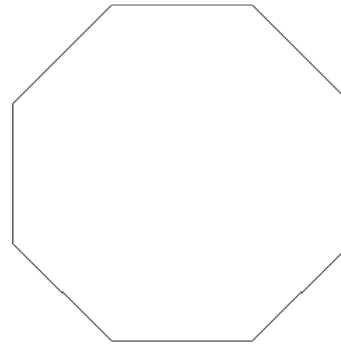
- for x in range(4):

- turtle.forward(100)

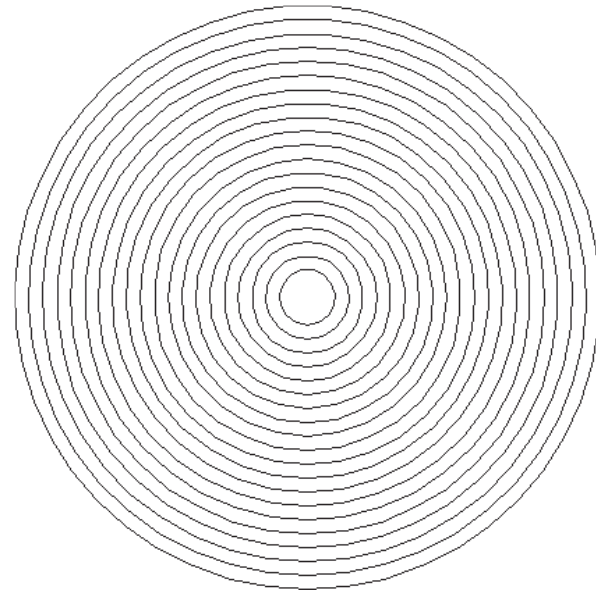
- turtle.right(90)

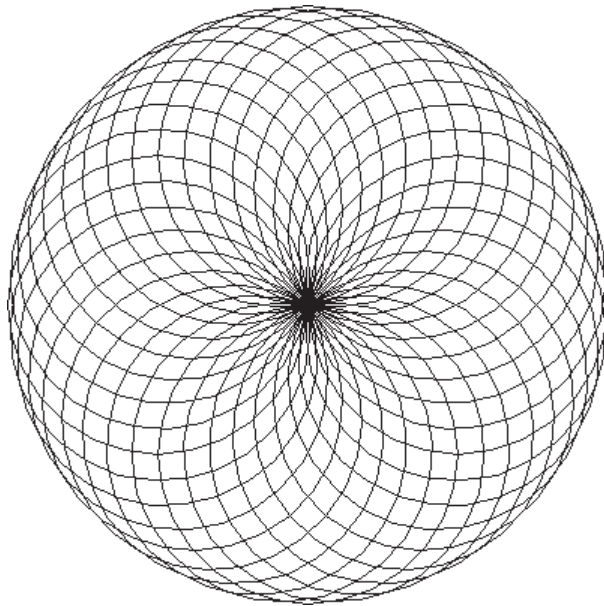


```
for x in range(8):  
    turtle.forward(100)  
    turtle.right(45)
```



```
import turtle
radius = 20
for count in range(20):
    turtle.circle(radius)
    radius = radius + 10
    x = turtle.xcor()
    y = turtle.ycor()
    turtle.penup()
    turtle.goto(x, y-10)
    turtle.pendown()
turtle.done()
turtle.Terminator
```





```
# This program draws a design using repeated circles.  
import turtle
```

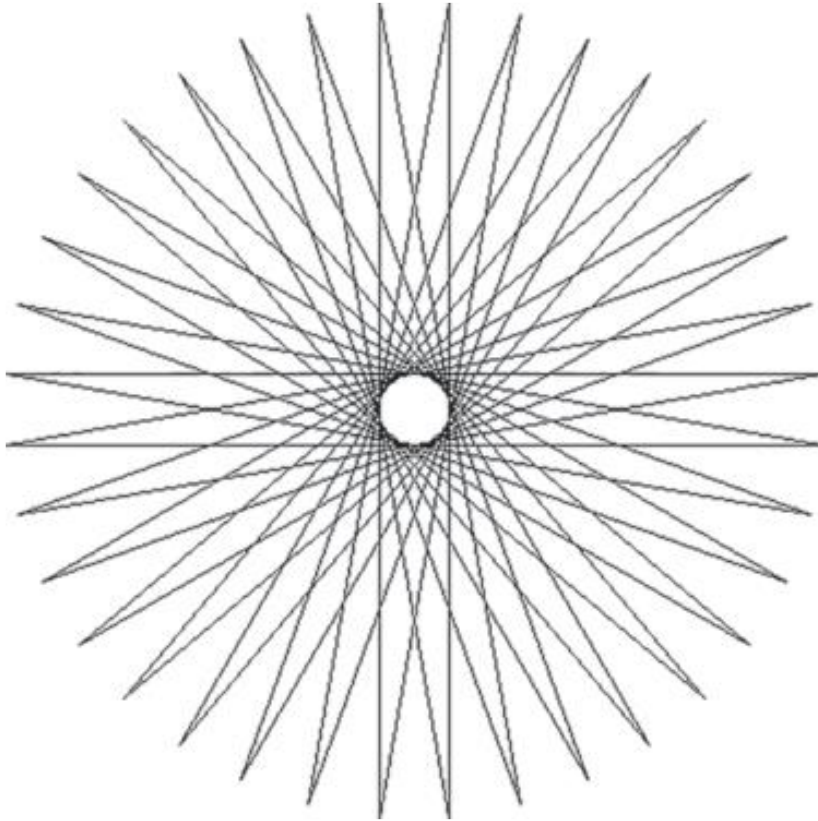
```
# Named constants
```

```
NUM_CIRCLES = 36    # Number of circles to draw  
RADIUS = 100        # Radius of each circle  
ANGLE = 10          # Angle to turn  
ANIMATION_SPEED = 0 # Animation speed
```

```
# Set the animation speed.  
turtle.speed(ANIMATION_SPEED)
```

```
# Draw 36 circles, with the turtle tilted  
# by 10 degrees after each circle is drawn.  
for x in range(NUM_CIRCLES):  
    turtle.circle(RADIUS)  
    turtle.left(ANGLE)
```





```
import turtle
```

```
turtle.hideturtle()  
turtle.penup()  
turtle.goto(-200, 0)  
turtle.pendown()
```

```
for x in range(36):  
    turtle.forward(400)  
    turtle.left(170)
```

```
turtle.done()  
turtle.Terminator
```

