3 Control Statements

Learning Outcome

Programming Elements and Constructs Apply the fundamental programming constructs to control the flow of program execution: Sequence, Selection, Iteration

3.1 Selection: if and if-else Statements

Selection statements allow a computer to make choices based on a **condition.**

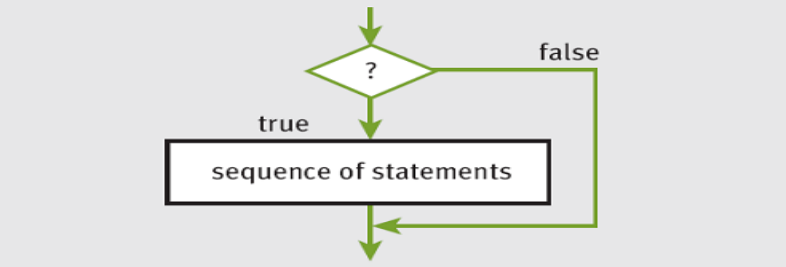
3.1.1 The Boolean Type, Comparisons, and Boolean Expressions

Boolean data type consists of two values: true and false (typically through standard **True/False**)

COMPARISON OPERATOR	MEANING
==	Equals
!=	Not equals
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

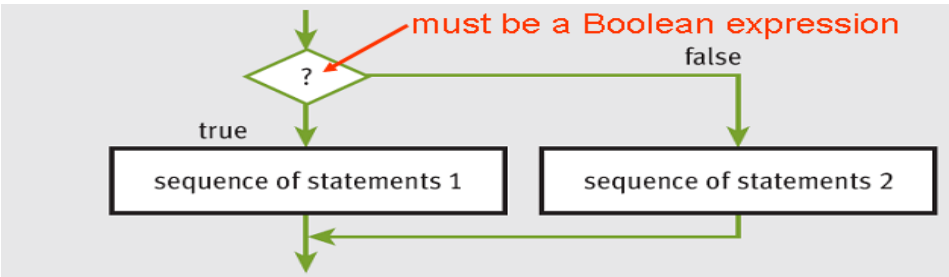
For example, 4 != 4 evaluates to False.

3.1.2 One-Way Selection Statement (if statement)




```
if x < 0:
     x = - x
```

3.1.3 Two-Way Selection Statements (if-else statements)



Syntax:

```
if <condition>:
    <sequence of statements-1>
else:
    <sequence of statements-2>
```

If-else statement can be used to check inputs for errors:

```
import math

area = float(input("Enter the area: "))
if area > 0:
    radius = math.sqrt(area / math.pi)
    print("The radius is", radius)
else:
    print("Error: the area must be a positive number")
```

3.1.4 Multi-Way if Statements

Testing conditions that entail more than two alternative courses of **action**.

Syntax:

```
if <condition-1>:
    <sequence of statements-1>

elif <condition-n>:
    <sequence of statements-n>
else:
    <default sequence of statements>
```

Example: the grading scheme below can be described in code by a multi-way if statement.

LETTER GRADE	RANGE OF NUMERIC GRADES
A	All grades above 89
B	All grades above 79 and below 90
C	All grades above 69 and below 80
F	All grades below 70

```
number = int(input("Enter the numeric grade: "))
if number > 89:
    letter = 'A'
elif number > 79:
    letter = 'B'
elif number > 69:
    letter = 'C'
else:
    letter = 'F'
print("The letter grade is", letter)
```

Another way to describe the grading scheme in pseudocode is to make use of CASE statements.

```
BEGIN
    INPUT number
    CASE OF number
        > 89: letter ← 'A'
        > 79: letter ← 'B'
        > 69: letter ← 'C'
        OTHERWISE: letter ← 'F'
    ENDCASE
END
```

Note that the case clauses are tested in sequence. When a case that applies is found, its statement is executed and the CASE statement is complete. Any remaining cases are not tested.

3.1.5 Logical Operators and Compound Boolean Expressions

When there are multiple conditions to check, we can use logical operators and compound Boolean expressions to simplify the code.

Logical Operators: **and, or, not**

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

A	not A
True	False
False	True

Examples:

```
>>> A = True
>>> B = False
>>> A and B
False
>>> A or B
True
>>> not A
False
```

Compare the two approaches below to check inputs for errors:

```
number = int(input("Enter the numeric grade: "))
if number > 100:
    print("Error: grade must be between 100 and 0")
elif number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```

```
number = int(input("Enter the numeric grade: "))
if number > 100 or number < 0:
    print("Error: grade must be between 100 and 0")
else:
    # The code to compute and print the result goes here
```

- The logical operators are evaluated after comparisons but before the assignment operator
- **not** has higher precedence than **and** and **or**

TYPE OF OPERATOR	OPERATOR SYMBOL
Exponentiation	**
Arithmetic negation	-
Multiplication, division, quotient, remainder	*, /, //, %
Addition, subtraction	+, -
Comparison	==, !=, <, >, <=, >=
Logical negation	not
Logical conjunction and disjunction	and, or
Assignment	=

3.1.6 Nested if Statements

An auto insurance agency assigns rates based on sex and age. Males below age 25 pay the highest premium, $1000. Males 25 or older pay $400. Females below age 21 pay $800, whereas those 21 or older pay $500.

```
if gender == "m":
     if age < 25:
          rate = 1000
     else:
          rate = 400
else:
     if age< 21:
          rate = 800
     else:
          rate = 500
```

Some nested if statements can be described as an else-if statements.

Nested *if* statements	***elif* statements**

`score = int(input("Enter score: "))`
`print ("Grade of student is:", end = " ")`
 	
`if score>=90:`
`  print ("A")`
`else:`
`  if score >=80:`
`  print ("B")`
`  else:`
`  if (score >= 70):`
`  print ("C")`
`  else:`
`  print ("F")`
	`score = int(input("Enter score: "))`
	`print ("Grade of student is:", end = " ")`
	
	`if score>=90:`
	`  print ("A")`
	`elif score >=80:`
	`  print ("B")`
	`elif score >= 70:`
	`  print ("C")`
	`else:`
	`  print("F")`

Tutorial 3A

1. Determine the value of each of the following Python expressions:

   a. not ((4.5 < 12.9) and (6 * 2 >= 13))
   b. not ((4.5 < 12.9) or (6 * 2 >= 13))
   c. (0 == 1) or (2 < 3) and (7 < 6)
   d. (2 < 3) or (0 == 1) and (7 < 6)
   e. (50 < 100) and ('100' < '1034')

2. What's the difference?

(a)	(b)

`score = int(input ("Enter score: "))`
`eligible = input("Enter eligibility: ")`
 	
`if score > 90:`
`  if eligible == "Y":`
`  print ("Hire")`
`else:`
`  print ("Reject")`
	`score = int(input ("Enter score: "))`
	`eligible = input("Enter eligibility: ")`
	
	`if score > 90:`
	`  if eligible == "Y":`
	`  print ("Hire")`
	`  else:`
	`  print ("Reject")`

3. Write code to determine someone's risk of heart disease using the following rules based on age and body mass index (BMI).

		Age	
		< 45	>= 45
BMI	< 22.0	Low	Medium
	>= 22.0	Medium	High

4. Suppose that a student is assigned a band 1, 2 or 3 as follows:

- Band 1 for a score of at least 85
- Band 2 for a score of under 85 but at least 70
- Band 3 for a score under 70

Neither of the program segments below is well-written -- one is correct but inefficient; the other is incorrect. Explain what is wrong with each.

```
if score >= 70:
    band =  2
elif score >=85:
    band = 1
else:
    band = 3
```

```
if score >= 85:
   band = 1
if score >= 70 and  score < 85:
   band = 2
If score < 70:
   band = 3
```

5. Assume that the variables **x** and **y** refers to strings. Write a code segment that prints these strings in alphabetical order. You should assume that they are not equal.

6. [Referring to the practice question in Tutorial 1A] John is requesting a program that computes his income tax. The rule for calculation is

- Taxpayers must enter their gross income.
- Taxpayers are allowed $10,000 standard deduction.
- For each dependent, taxpayer is allowed additional $2000 deduction.
- Income tax is charged at a flat rate of 20% based on taxable income after all deductions.

Write a program following your design in Tutorial 1A.

7. In a two-person game of Rock-Scissors-Paper, each player selects either 'R', 'S' or 'P'. The winner is determined as follows:

Rock breaks Scissors.
Paper covers Rock.
Scissors cuts Paper.

The game is a tie if both players select the same choice.
Write a program to allow players make their input and output the winner.

3.2 Definite Iteration: The for Loop

– Repetition statements (or **loops**) repeat an action
– Each repetition of action is known as **pass** or **iteration**

3.2.1 Range Function

Range function generates a list of integers to help to iterate the loop. It expects up to three arguments.

– range (<upper bound + 1>)

```
>>> list(range(4))    #one argument
[0, 1, 2, 3]
```

With one argument only, the list starts from 0 and includes all numbers smaller than the argument.

– range (<lower bound>, <upper bound + 1>)

```
>>> list(range(1,5))   #two arguments
[1, 2, 3, 4]
```

With two arguments, the list starts from the first argument and includes all numbers smaller than the second argument.

– range (<lower bound>, <upper bound + 1>, <step>)

With three arguments, it allows you specify a step value.

```
>>> list(range(1, 6, 1))    # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2))    # Use every other number
[1, 3, 5]
>>> list(range(1, 6, 3))    # Use every third number
[1, 4]
>>>
```

When the third parameter is a negative number, it becomes a list that counts down.

```
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

3.2.2 Executing a Statement a Given Number of Times

Python's for loop is control statement that most easily supports definite iteration.

The form of this type of loop is:

```
for <variable> in range(<an integer expression>):   ← loop header
    <statement-1>
    ...                                             } loop body
    <statement-n>
```

← statements in body must be indented and aligned in the same column

Example: Loop to compute an exponentiation for a non-negative exponent.

```
>>> number = 2
>>> exponent = 3
>>> product = 1
>>> for eachPass in range(exponent):
        product = product * number
        print(product, end = " ")

2 4 8
>>> product
8
```

If the exponent were 0, the loop body would not execute and value of product would remain as 1

Example: Loop to print the same scripts several times.

```
>>> for eachPass in range(4):
        print("It's alive!", end=" ")

It's alive! It's alive! It's alive! It's alive!
```

3.2.3 Control Variable

Incrementing the control variable by step of 1

```
FOR i ← 1 to 4
        OUTPUT i, " squared is ", i * i
ENDFOR
```

The output will be:

1 squared is 1 <-- printed when i = 1
2 squared is 4 <-- printed when i = 2
3 squared is 9 <-- printed when i = 3
4 squared is 16 <-- printed when i = 4

Decrementing the control variable by step of 1

```
FOR i ← 4 TO 1 STEP −1
        OUTPUT i, " squared is ", i * i
ENDFOR
```

The output will be:

4 squared is 16 <-- printed when i = 4
3 squared is 9 <-- printed when i = 3
2 squared is 4 <-- printed when i = 2
1 squared is 1 <-- printed when i = 1

Do Not Alter the Control Variable
A statement in the **for** loop body should never assign a value to the control variable.

```
FOR i ← 1 to 10
        OUTPUT i
        i ← i + 2        // don't do this
ENDFOR
```

3.2.4 Indentation

Clear indentation and indication of loop body is important. When you want the loop body to be a block of statements, the block must be contained within the ENDFOR markers, for readability.

```
BEGIN
        FOR i ← 6 to 8
                square ← i * i
                OUTPUT "this time i equals ", i
                OUTPUT "its square is ", square
        ENDFOR
        OUTPUT "so long"
 END
```

The output will be:

this time i equals 6
its square is 36
this time i equals 7
its square is 49
this time i equals 8
its square is 64
so long

Remark

1. Note the indentation, and **ENDFOR** to enclose the compound statements which makes up the loop body. This also helps to separate the loop from the statement **OUTPUT "so long"**.
2. The indentation of the loop body to make it stand out to the human eye. Note that the statement **OUTPUT "so long"** is not indented but is aligned with the **FOR** statement, since it comes *after* the loop.

3.2.5 Traversing the Contents of a Data Sequence

Strings are also sequences of characters and values in a sequence can be visited with a **for** loop:

```
for <variable> in <sequence>:
    <do something with variable>
```

```
>>> for character in "Hi there!":
        print(character, end = " ")

H i   t h e r e !
```

3.2.6 Processing Input Groups of Data

General Form in Top Down Design

Before: initialize any variables that need initializing
print any headings

During: FOR i ← ___ to ___
output-input combination(s) to get the information on one person or item
process that person's or item's information
ENDFOR

After: print any final tallies or results

Example The following program uses a **for** loop to process the scores of four students, which are input by the user. It then counts and displays total number of students scored at least 90.

For example, if the user were to input scores as follows,

enter score of student 1: 92
enter score of student 2: 85
enter score of student 3: 95
enter score of student 4: 84
enter score of student 5: 80

The output will be:

2 student (s) scored at least 90

Here is the program, in pseudocode and Python code.

```
# program counting
# find the number of students scored at least 90

BEGIN
    count ← 0

    FOR student ← 1 TO 5
        # read in students' score from user
        OUTPUT "enter score of student ", student, ": "
        INPUT score

        IF score >= 90
            count ← count + 1
        ENDIF
    ENDFOR

    OUTPUT count, " student(s) scored at least 90"
END
```

```
count = 0

for student in range(1, 5+1):
     print ("enter score of student", student, ":",end = " ")
     score = int(input())

     if score >= 90:
          count = count + 1

print (count, "student(s) scored at least 90")
```

Initializing

A variable that is given a starting value before a loop is said to be initialized. For example, in the previous program, the statement

```
count = 0
```

initialized count to 0. In the subsequent for loop, whenever there is a score at least 90, count is increased by 1.

[CAUTION!] If you did not initialize count to 0, its starting value would be unreliable. It might be a value left over from the previous run program. Even in pseudocode, it should be clear that readers need not make their own assumptions of the initial value.

Example The next program will use the variable **sum** to compute the sum of four numbers input by the user. For example, it the user were to input numbers as follows,

enter number: 17
enter number: 13
enter number: 20
enter number: 8

The output will be:

The sum is 58

Here is the program, in pseudocode and Python code.

```
# program summing, finds the sum of 4 input integers

BEGIN
    sum ← 0

    FOR i ← 1 to 4
        OUTPUT "enter number "
        INPUT number

        sum ← sum + number
    ENDFOR

    OUTPUT "The sum is ", sum
END
```

```
sum = 0

for i in range(1, 4+1):
     number = int(input("enter number:"))
     sum = sum + number

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

Example Suppose we wish to find the largest, or maximum, number from a list of five positive integers that are entered by the user. An algorithm can be developed using the following ideas.

Use a variable called **maxSoFar**. Initialize **maxSoFar** with the first number, then give each of the remaining numbers a chance to displace the current value of **maxSoFar**. The final value of **maxSoFar** will be the maximum.

Before: initialize **maxSoFar** with first number

During: **for i = 2 to 5**
read a number
test whether it is larger than **maxSoFar**,
if so change the value of **maxSoFar**

After: print the value of **maxSoFar**

Here is the detailed pseudocode and Python code.

```
# program maxOf5
# finds the maximum of 5 input positive integers

BEGIN
    OUTPUT “enter first number: “
    INPUT maxSoFar

    FOR i ← 2 TO 5
        OUTPUT “enter next number: “
        INPUT numb

        IF numb > maxSoFar
            maxSoFar ← numb
        ENDIF
    ENDFOR

    OUTPUT “the maximum is: “, maxSoFar
END
```

```
maxSoFar = int(input("enter first number: "))

for i in range(2,5+1):
     numb = int(input("enter next number: "))

     if numb > maxSoFar:
               maxSoFar = numb

print ("the maximum is:", maxSoFar)
```

3.2.7 Variable Limits

You can increase the flexibility of a program by using variable(s) as the upper or lower limit in the **for** header. A common application allows the user to specify how many groups of data will be entered.

Example In the following program, in pseudocode, the user is asked to input the number of employees **(n)** to be processed. Note the use of the variable **n** as the upper limit in the **for** header.

FOR employee ← 1 to n

Using **n** gives the program added flexibility -- the user does not have to change any lines in the program to run it for different numbers of employees.

```
# program payroll
# finds each employee's wage and total payroll

BEGIN
        # initialize the total payroll
        sum ← 0
        OUTPUT “enter number of employees: “
        INPUT n

        FOR employee ← 1 to n
                # get one employee's data
                OUTPUT “enter hours and rate of employee “, employee
                INPUT hours, rate

                # process that data and print the wage for one employee
                wage ← hours * rate
                sum ← sum + wage
                OUTPUT “Wage of employee”, employee, “is $”, wage
        ENDFOR

        # print final result
        OUTPUT “total payroll $”, sum
END
```

```
sum = 0
n = int(input("enter number of employees:"))

for employee in range(1, n+1):
     print("\nEnter hours and rate for employee ",employee)
     hours = float(input("Hours: "))
     rate = float(input("Rate: "))

     wage = hours * rate
     sum = sum + wage
     print ("Wage of employee", employee, "is $%0.2f" % wage)
print ("total payroll $%.2f", %sum)
```

3.2.8 Nested for Loops

In nested **for** loops, the entire outer loop body is executed for each of the values of the outer loop control variable. Thus, for each value of the outer loop control variable, the inner **for** loop will run through *all* of its values.

```
FOR n ← 2 to 3
      FOR i ← 6 to 7
            OUTPUT  n, " ", i
            OUTPUT  "hello"
      ENDFOR
ENDFOR
```

The output will be

2 6
hello
2 7
hello

} printed when n = 2

3 6
hello
3 7
hello

} printed when n = 3

3.2.9 Case Study: Row-by-Row Processing

An important application of nested loops is in using an outer loop to process a number of rows, where the processing for each row also requires a loop. Write a program segment to generate a right triangle with **numRows** rows of asterisks, such that the first row has one asterisk, the second row has two asterisks, the third row has three asterisks, and so on, and where the rightmost asterisks from each row are vertically aligned. For example, when **numRows**=4, the segment should generate the output

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

Design: assign a value to **numRows**

FOR n ← 1 to numRows
    indent the appropriate number of spaces (loop needed)
    print **n** asterisks (loop needed)
    drop the cursor to the next line
ENDFOR

Note that the number of spaces you indent keeps decreasing by 1, as **n** increases by 1. Thus, you might realize that in the **n**th row, you must indent **(numRows - row)** spaces, which can be done with a **for** loop that prints **(numRows - row)** blanks.

```
BEGIN
    INPUT numRows
    FOR n ← 1 to numRows
        FOR space ← numRows – n TO 1 STEP −1
            OUTPUT " "
        ENDFOR

        FOR star ← 1 TO n
            OUTPUT "*"
        ENDFOR

        OUTPUT newline
    ENDFOR
END
```

```
numRows = int(input("Number of rows: "))
for row in range(1, numRows+1):
     for space in range(numRows-row, 0, -1):
             print (" ", end='')
     for star in range(1, row+1):
             print ("*", end='')
     print ()
```

If we don't use nested **for** loop, it can be coded as follows:

```
numRows = int(input("Number of rows: "))
for row in range(1, numRows+1):
         print (" "*(numRows-row), end='')
         print ("*"*row)
```

Tutorial 3B

1. Write the outputs of the following loops:

(a)
```
for count in range(5):
    print (count+1, end=" ")
```
(b)
```
for count in range(1, 4):
    print  (count, end=" ")
```
(c)
```
for count in range(1, 6, 2):
    print  (count, end=" ")
```
(d)
```
for count in range(6, 1, -1):
    print  (count, end=" ")
```

2. Write a loop that prints your name 50 times. Each output should begin on a new line.

3. Write a loop that that prints the first 128 ASCII values followed by the corresponding characters.

4. Assume that the variable **testString** refers to a string. Write a loop that prints each character in this string, followed by its ASCII value.

5. Write a loop that counts the number of space characters in a string. Recall that the space character is represented as `" "`.

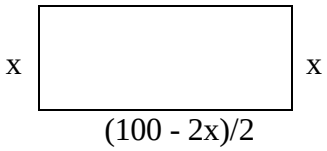
6. The German mathematician Gottfried Leihniz developed the following method to approximate the value of π:

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \ldots\ldots$$

Write a program that allows the user to specify the number of iterations used in this approximation and that displays the resulting value.

7. Write a program that receives as input the noon temperature for each of the days in a week. It will find the average noon temperature for just those days on which the noon temperature was above freezing (0 degree Celsius).

8. Suppose a gardener has 100 feet of fencing and wishes to enclose a rectangular garden alongside her house. Drawing a diagram, we find that the area of the garden equals (x)((100 - 2x)/2)



Write a program that will produce the following table of values and the maximum area.

Length	**Area**
10	**400**
11	**429**
.	**.**
.	**.**
44	**264**
45	**225**

Maximum area is 625

9. Write a program to print a multiplication table based on the number entered. A sample run (with input 5) is shown below:

Number of Columns: 5

	1	**2**	**3**	**4**	**5**
1	**1**				
2	**2**	**4**			
3	**3**	**6**	**9**		
4	**4**	**8**	**12**	**16**	
5	**5**	**10**	**15**	**20**	**25**

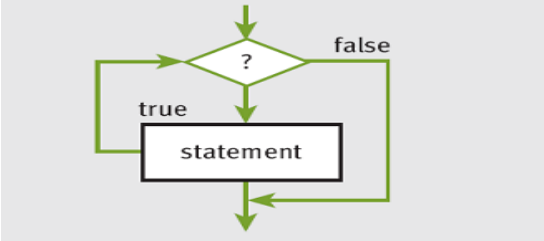
3.3 Conditional Iteration: The while Loop

– Conditional iteration requires that condition be tested within loop to determine if it should continue, it is called continuation condition.
– The statements within loop can execute zero or more times.

3.3.1 The Structure and Behavior of a while Loop

```
while <condition>:
    <sequence of statements>
```

Also called the entry-control loop, the while loop starts by testing the while condition. If the condition is true, the entire loop body is executed. Then control is returned to the top to retest the **while** condition. This process is repeated as long as the **while** condition is true.



[CAUTION!] Improper use may lead to infinite loop

[Pseudocode]

```
WHILE ( test condition )
    # body of loop;
ENDWHILE
```

Example

```
n ← 7
WHILE (n >= 0)
    OUTPUT n
    n ← n - 5
    OUTPUT 'Hi ', n
ENDWHILE
```

The output will be

7
Hi 2
} printed during 1st execution of loop body

2
Hi −3
} printed during 2nd execution of loop body

Post-control loop **REPEAT UNTIL** also behave like a while loop.

```
n ← 7
REPEAT
        OUTPUT n
        n ← n - 5
        OUTPUT 'Hi ', n
UNTIL n < 0
```

The statements in the loop are executed at least once. The condition is tested after the statements are executed and if it evaluates to `TRUE` the loop terminates, otherwise the statements are executed again.

3.3.2 Loop Logic, Errors, and Testing

Errors to rule out during testing while loop:

– Incorrectly initialized loop control variable
– Failure to update this variable correctly within loop
– Failure to test it correctly in continuation condition

To halt loop that appears to hang during testing, type Ctrl+c in terminal window or IDLE shell.

3.3.3 Fixed Step-Controlled While Loops

Problem	Codes using **for** loop	Codes using **while** loop

Sum of all integers from 1 to 100000	`sum = 0`
	`for count in range(1, 100001):`
	`  sum += count`
	`print(sum)`
		`sum = 0`
		`count = 1`
		`while count <= 100000:`
		`  sum += count`
		`  count += 1`
		`print(sum)`

Countdown from 10 to 1	`for count in range(10, 0, -1):`
	`  print(count, end=" ")`
		`count = 10`
		`while count >= 1:`
		`  print(count, end=" ")`
		`  count -= 1`

All odd integers from 1 to 10	`for i in range(1,10,2):`
	`  print(i)`
		`i = 1`
		`while i <= 9:`
		`  print (i)`
		`  i = i + 2`

Count controlled, or fixed step controlled **while** loops are similar to **for** loops in that there is a control variable that increases or decreases by a fixed step. However, when it is a fixed number of iterations, **for** loop is always preferred.

3.3.4 Data Sentinel-Controlled While Loops

In a data sentinel-controlled loops, the user can terminate data entry when he or she chooses by entering an appropriate signal known as sentinel. There are two types of sentinel-controlled loops, which we call types A, and B.

Type A: Using a y or n Question

Suppose that on this run the user wishes to find the sum

15 + 47 + 53 + 64

In the following program, in pseudocode, the user notifies the computer that there is no further data by entering an **n** in response to the prompt **type y to continue, n to stop**.

```
# program FindSum;
# finds the sum of any number of input integers

BEGIN
        sum ← 0
        ans ← 'y'
        WHILE ans = 'y'
                OUTPUT "enter number "
                INPUT numb
                sum ← sum + numb
                OUTPUT "  type y to continue, n to stop: "
                INPUT ans
        ENDWHILE
        OUTPUT "The sum is ", sum
END
```

Here is what would appear on the screen.

enter number 15
type y to continue, n to stop: y
enter number 47
type y to continue, n to stop: y
enter number 43
type y to continue, n to stop: y
enter number 64
type y to continue, n to stop: n
The sum is 169

```
sum = 0
ans = "y"
while (ans == "y") or (ans == "Y") :
     numb = int(input("enter number"))
     sum = sum + numb
     ans = input("type y to continue, n to stop:")
print ("The sum is", sum)
```

Type B: Using a Phony Value

The other type of sentinel is phony value −1. The user signals an end to data entry by typing the phony value −1.

```
BEGIN
    sum ← 0
    OUTPUT "enter number or −1 to stop:"
    INPUT numb

    WHILE numb <> −1
        sum ← sum + numb
        OUTPUT "enter number or −1 to stop:"
        INPUT numb
    ENDWHILE
    OUTPUT "The sum is ", sum
END
```

Note the two occurrences of the same **output-input** combination, once before the loop and once at the bottom of the loop body.

If we use the same data entry of Type A, here is what would appear on the screen.

enter number or -1 to stop: 15
enter number or -1 to stop: 47
enter number or -1 to stop: 43
enter number or -1 to stop: 64
enter number or -1 to stop: −1
The sum is 169

Recall in Section 3.2.7, if we can get the total number of data entry in advance, it will become a fixed number of iteration and for loop is better.

3.3.5 Task-Controlled While Loop

In some situations when the condition for the completion of the task is not a counter or a data sentinel, it is useful to think of a loop in terms of its task as follows:

```
WHILE  task not completed
        # loop body;
ENDWHILE
```

Example The sum of the squares 12 + 22 + 32 + ... eventually goes over 1000.
Write a program, in pseudocode, to find the integer whose square first puts the sum over 1000. The output should be of the form

Sum first goes over 1000 when you add ____ squared
Sum is _____

```
# program over1000;
# finds first square to put sum over 1000

BEGIN
        sum ← 0
        n ← 0
        WHILE sum <= 1000
                n ← n + 1
                sum ← sum + n * n
        ENDWHILE
        OUTPUT “Sum first goes over 1000 when you add “, n, “ squared”
        OUTPUT “Sum is “, sum
END
```

3.3.6 Multiple Task-Controlled While Loops

In cases when there are two possible reasons for terminating a loop, it will be necessary to use a compound condition in the exit test. We have learnt to use compound Boolean expressions in Section 3.1.5.

Usually, in such situations, it will also be necessary to include an **if-else** test after the loop to determine which condition caused the exit.

Example (Bounded Number of Tries)
Write a program, in pseudocode, in which the user is given a maximum of three tries to give the capital of Indonesia. If the user fails to give the correct answer by the third try, he or she is then informed what the correct answer is. Format the program so that it could produce the following two sample runs:

Give capital of Indonesia Bravo
Give capital of Indonesia Manila
Give capital of Indonesia Tokyo
You did not get it in 3 tries
The correct answer is Jakarta
Give capital of Indonesia Bravo
Give capital of Indonesia Jakarta
Nice work. You got it on try 2

```
# program stateCapital

BEGIN
        STATE ← “Indonesia”
        CAPITAL ← “Jakarta”

        tries ← 1
        OUTPUT "Give captial of", STATE, ": "
        INPUT guess
```

```
    WHILE (guess <> CAPITAL) AND (tries < 3)
        OUTPUT "Give captial of", STATE, ": "
        INPUT guess
        tries ← tries + 1
    ENDWHILE

    IF guess = CAPITAL
        OUTPUT "Nice work. You got it on try ", tries
    ELSE
        OUTPUT "You did not get it in 3 tries"
        OUTPUT "The correct answer is ", CAPITAL
    ENDIF
END
```

3.3.7 Error Trapping and Robustness

There are techniques the programmer can use to gain protection against the entry of incorrect data. One technique is to include clear prompts so that the user knows the precise form of the inputs. A second technique is to include program code to detect and trap mistakes. A program that contains such safeguards is said to be **robust**.

The programme below determines whether an input capital letter is in the 1st half (A-M) or the 2nd half (N-Z) of the alphabet.

```
BEGIN
    OUTPUT “enter capital letter “
    INPUT letter
    IF (letter >= 'A') AND (letter <= 'M')
        OUTPUT letter, “ in 1st half of alphabet”
    ELSE
        OUTPUT letter, “ in 2nd half of alphabet”
    ENDIF
END
```

Here is a run of the fragment in which the user failed to heed the prompt to enter a capital letter.

enter capital letter d
d in 2nd half of alphabet

The conditions, letter >= 'A' and letter <= 'M', are evaluated using **ASCII values** of letter. In this case, ASCII values of d is higher than that of ‘A’ and ‘M’.

In the following robust program fragment, note the use of a while loop to force the user to enter a capital letter. Execution does not get beyond the loop until the user enters an uppercase letter.

```
BEGIN
        OUTPUT “enter capital letter “
        INPUT letter

        WHILE (letter < 'A') OR (letter > 'Z')
                OUTPUT “*** Not a Capital Letter ***”
                OUTPUT “enter capital letter “
                INPUT letter
        ENDWHILE

        IF (letter >= 'A') AND (letter <= 'M')
                OUTPUT letter, “ in 1st half of alphabet”
        ELSE
                OUTPUT letter, “ in 2nd half of alphabet”
        ENDIF
END
```

Here is a typical run of the above.

enter capital letter <u>d</u>
***** Not a Capital Letter *****
enter capital letter <u>D</u>
D in 1st half of alphabet

3.3.8 Nested While Loops

Consider the following program, in pseudocode, which will compute separate point totals for males and females. The user will keep inputting data pairs such as **<u>24 m</u>** and **<u>47 f</u>**, where the first item of the data pair is the point total and the second is the sex of the player who achieved it.

```
# program summingPoints

BEGIN
        maleSum ← 0
        femSum ← 0
        ans ← ‘y’

        WHILE (ans = 'y') OR (ans = ‘Y’)
                OUTPUT "enter number of points"
                INPUT points
                OUTPUT “enter sex (m or f):"
                INPUT sex
```

```
        WHILE (sex <> 'm') AND (sex <> 'f')      # Error Trapping
            OUTPUT "enter sex (m or f):"
            INPUT sex
        ENDWHILE

        IF (sex = 'm')
            maleSum ← maleSum + points
        ELSE
            femSum ← femSum + points
        ENDIF

        OUTPUT "Type y to continue, n to stop: "
        INPUT ans
    ENDWHILE

    print "Total points for the male team: ", maleSum
    print "Total points for the female team: ", femSum
END
```

Tutorial 3C

1. As input for **program receipt**, a cashier enters in the data for each of the customer's purchases. Each data group consists of an item name, the price of the item, and the quantity of that item being purchased. The data sentinel **xyz** is used instead of an item name. Here is the screen display for a typical run that would also produce a printed sales receipt.

```
enter item name or xyz to stop: hammer
  enter price per item: 19.50
  enter quantity: 1
1 hammer       $ 19.50

enter item name or xyz to stop: umbrella
  enter price per item: 14.50
  enter quantity: 2
2 umbrella     $ 29.00

enter item name or xyz to stop: lamp
  enter price per item:  3.50
  enter quantity:  6
6 lamp         $ 21.00

enter item name or xyz to stop: xyz

Total bill     $ 69.50
```

2. The greatest common divisor of two positive integers, A and B, is the largest number that can be evenly divided into both of them. Euclid's algorithm can be used to find the greatest common divisor (GCD) of two positive integers. You can use this algorithm in the following manner:

   (a) Compute the remainder of dividing the larger number by the smaller number.
   (b) Replace the larger number with the smaller number and the smaller number with the remainder.
   (c) Repeat this process until the smaller number is zero.

   The larger number at this point is the GCD of A and B. Write a program that lets the user enter two integers and then prints each step in the process of using the Euclidean algorithm to find their GCD.

Assignment 3

1. The factorial of an integer N is the product of all of the integers between 1 and N, inclusive. Write a program that computes the factorial of a given integer N.

2. Write a program that accepts the lengths of three sides of a triangle as inputs.
   The program output should indicate whether or not the triangle is an equilateral triangle.

3. Write a program that reads a word and print the number of letters in the word, the number of vowels in the word, and the percentage of vowels.

   Enter a word:sequoia
   Letters: 7
   Vowels: 5
   Percentage of vowels: 71.43

4. Write a program that inputs a positive integer n and then prints a rectangle of asterisks n lines high and 2n columns wide. For example, if the input is 5 then the output should be

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

5. The sum of the divisors of the number 15 is 24 (1 + 3 + 5 + 15). Write a program that will print the sum of the divisors for each of the integers from 100 to 110. The final line of the printout should state which integer has the largest sum. Format the output as follows:

100: 1 2 4 5 10 20 25 50 100
sum of divisors 217
.
.
.
110: 1 2 5 10 11 22 55 110
sum of divisors _____
_____ has maximum sum of divisors.

6. In the game of Lucky Sevens, the player rolls a pair of dice. If the dots add up to 7, the player wins $4; otherwise, the player loses $1. Suppose that a casino tells player that there are lots of ways to win: (1, 6), (2, 5), etc. Write a program that takes as input the amount of money that the player wants to put into the pot, and play the game until the pot is empty. At that point, the program should print the number of rolls it took to break the player, as well as maximum amount of money in the pot.

Hint: you may use random.randint to generate random number from among numbers between two arguments, included.

```
>>> import random
>>> for roll in range(10):
        print(random.randint(1, 6), end=" ")

2 4 6 4 3 2 3 6 2 2
>>>
```

7. **Guess My Number Game (Design the program for both versions)**

Version A: the computer picks a random number between 1 and 100 that the player has to guess. For each guess, the computer must tell the player if the number is too small or too large for the actual number. When the player gets the number, the program will display the number of tries that the player used. A sample output is shown below.

```
Enter your guess: 50
Too small
Enter your guess: 75
Too large
Enter your guess: 63
Too small
Enter your guess: 69
Too large
Enter your guess: 66
Too large
Enter your guess: 65
You've got it in 6 tries!
```

Version B: The player and the computer trade places. That is, the player picks a random number between 1 and 100 that the computer has to guess. Before you start, think about how the computer will guess.