Fundamentals of Python

(Reference: Fundamentals of Python, K.A Lambert and B.L Juneja)

Introduction

- **Python** is an interpreted language, and you can run simple python expressions and statement in an interactive programming environment, called the **Shell.**
- Whether you are running Python code as a script or interactively in a shell, the Python interpreter does a great deal of work to carry out the instructions in your program.
- The interpreter reads Python expression or statement called the source code and verifies that it is well formed.
- In this step, the interpreter behaves like a strict English language teacher.

Introduction(cont.)

- As soon as the interpreter encounters such an error, it halts translation with an error message.
- If the python expression is well formed, the interpreter then translates it to an equivalent form in a lower-level language called **byte code**.
- This byte code is next sent to another software component, called the **Python Virtual Machine (PVM)** where it is executed.
- If another error occurs during this step, the execution also halts with an error message.

Introduction(cont.)

- **Algorithm**: An algorithm is a sequence of instructions for solving a problem.
- **Python scripts**: Python scripts are programs that are saved in files and run from a terminal command prompt.
- **Syntax**: Syntax is the set of rules for forming correct expressions and statements in a programming language.

print function

- Syntax of the **print** function:

  print(<expression1>, <expression2>,……., <expression$_n$>)

- **print(<expression>, end=“ “)** would prevent new line of the print function

input function

Syntax

<variable identifier> = input(<a string prompt>)

- How does the **input** function know what to use as the prompt?
- The text/string prompt is an argument for the **input** function that tells it what to use for the prompt.
- The **input** function always builds a **string** from the user's keystrokes and returns it to the program.

Data Types and Expressions

- In programming, a **data type** consists of a set of values and a set of operations that can be performed on those values.
- A **literal** is the way a value of a data type looks to a programmer.
- String literals ‘’ and “ “ are empty string and “\n” is a new line character

Type of Data	Type name	Literals
Integers	int	-1, 0, 1, 2,…..
Real numbers	float	-0.55, 0.333, …
Character strings	str	“Hi”, “ “, ‘A’, ‘66’, ‘5’,….

Type Conversion

- In Python there are two type conversion functions, called **int** (for integers), and **float**(for floating point numbers)

Function	What it does
float(<a string of digits>)	Converts a string of digits to a floating point number
int (<a string of digits>)	Converts a string of digits to an integer value
input (<a string>)	Displays the string prompt and waits for a keyboard input. Returns the input string to the user
Print(<exp1>, <exp2>,…<exp$_n$>)	Evaluate the expressions and displays them and the comma will concatenate the strings.
<string1> + <string2>	Glues the two strings together and returns the result.

round function

- The **round()** function rounds a float to the nearest int value.

```
int1 = float(input("Enter first float: "))
int2 = float(input("Enter second float: "))
sum = int1 + int2
print("The sum of numbers without round is ", sum)
print("The sum of numbers with round is ", round(sum))
```

```
Enter first float: 8.5677
Enter second float: 1.234
The sum of numbers is  10
```

Variable

- A variable associates a name with a value
- **<u>Syntax</u>**

  <variable_name> = <expression>

<u>Example:</u>

```
sum = 20
name = "Anil"
```

- Where sum and name are variables, and 20 and “Anil” are expressions

Exercise

- Write a line of code that prompts the user for his/her name and saves the user’s input in a variable called *name*.
- Get two floating point numbers from keyboard and print their sum.

Escape sequence

\b is the backspace
\n is the newline
\t is the horizontal tab
\\ is the \ character
\’ is the single quotation
\” is the double quotation

String Concatenation

- **Use concatenation operator +**

```
Print("Hello" + "How are you?")
```

* operator

- In python, the * operator allows you to build a string by repeating another string a given number of times.
- For example, if you want the string “python” to be proceeded by 30 spaces;

print(" " * + 30 "Python")

This will print “Python” after 30 space characters

ord and **chr** function

- Python's **ord** and **chr** functions convert charters to their numeric ASCII codes and back again respectively.
- Example of **ord** function:

```
val = 'a'
print(ord(val))
```
#convert a character into its ASCII code

OUTPUT is 97

- Example of **chr** function:

```
val = 97
Print(chr(val))
```
#converts ASCII code into character

Expressions

Operator	Meaning	Syntax
–	Negation	–a
**	Exponentiation	a ** b
*	Multiplication	a * b
/	Division	a / b
//	Quotient	a // b
%	Remainder/Modulus	a % b
+	Addition	a + b
-	Subtraction	a - b

Precedence Rule

- Exponentiation has the highest precedence.
- Negation is evaluated next.
- Multiplication, division, remainder are evaluated before addition and subtraction
- Addition and subtraction are evaluated before assignment.

Precedence Rule

- **Precedence rule – Example:**
  1. 5 + 3 * 2 = 5 + 6 = 11
  2. (5 + 3) * 2 = 8 * 2 = 16
  3. 6 % 2 = 0
  4. 2 * 3 ** 2 = 2 * 9 = 18
  5. 3 ** 2 = 3^2 = 9
  6. 2 ** 3 ** 2 = 2 ** 9 = 2^9 = 512
  7. (2 ** 3) ** 2 = 2^3 ** 2 = 8^2 = 64
  8. 45 / 2 = 22.5 (returns a float result)
  9. 45 // 2 = 22 (returns an integer result)
  10. 45 / 0 = error

Type Conversion

No.	Type(<expression>)	Example
1	int(<a floating point number>)	int(3.77) = 3
2	int(<string>)	Int(“33”) = 33
3	float(<an integer number>)	float(22) = 22.0
4	float(<string>)	float(“22”) = 22
5	Str(<any value>)	Str(99) = “99”

Augmented Assignment

- The assignment symbol can be combined with the arithmetic and concatenation operators to provide **augmented assignment** operations.
- Syntax: **<variable><operator>= <expression>**

No.	Augmented Assignment	Meaning
1	a += 3	a = a + 3
2	a -= 3	a = a - 3
3	a *= 3	a = a * 3
4	a /= 3	a = a / 3
5	a %= 3	a = a % 3
6	a += “Hello”	a = a + “Hello”

The **math** Module

- The **math module** includes several functions that perform basic mathematical operations.
- To use a resource from a module, you write the name of a module as a qualifier, followed by dot (‘.’) and the name of the resource.
- **Example:**

```
import math
print(math.pi)          # 3.14563777288942
print(math.pow(8,2))  #64.0
print(math.pow(5,4))  #625.0
```

Get **Help** for a math Function

- The following example shows how to get help for a **cosine** function:

```
print(help(math.cos))
```

- If you are going to use only a couple of module's resources frequently, you can avoid the use of the qualifier with each reference by **importing** the individual resources as follows:

```
from math import pi, sqrt
print(pi, sqrt(2))
```

- This way you can avoid the usage of the "math." before any math function.

```
from math import*
```

- would import all of the math module's resources

Printing math values in defined precisions

- For example, check the result of **pi** from the math module:

```
import math
print(math.pi)
```

output: 3.141592653589793

- The following syntax is implemented to print the **pi** value in a defined precision and space:

“%<field width>.<precision>f” % float variable/value

- For example, print the two decimal point value of pi by:

```
print(“pi is %0.2f” % math.pi)
```
Output: pi is 3.14

Income Tax Calculator

- The customer requests a program that computes a person's income tax.
- Let us assume the following tax laws:
  1. All taxpayers are charged a flat tax rate of 20%.
  2. All taxpayers are allowed a 10,000$ standard deduction.
  3. For each dependent, a taxpayer is allowed an additional 3,000$ deduction.
  4. Gross income must be entered.
  5. The income tax is expressed as a decimal number.
- **Formule:**

  Taxable_income = Gross income – 10,000 – (3,000 * no. of dependents)

  Income_tax = Taxable_income * tax_rate

```
TAX_TATE = 0.20 # 20%
STANDARD_DEDUCTION = 10000.0
DEPENDENT_DEDUCTION = 3000.0

#Request the gross income

grossIncome = float(input(" Enter the gross income <minimum
  10,000>:" ))
numDependents = int(input(" Enter the number of dependents: "))
#Compute the inmcome tax
taxableIncome = grossIncome - STANDARD_DEDUCTION - \
  (DEPENDENT_DEDUCTION * numDependents)
incomeTax = taxableIncome * TAX_TATE
print("The income tax is $" + str(incomeTax))
```

Output:

Enter the gross income <minimum 10,000>: 20000

Enter the number of dependents: 2

The income tax is $800.0

Practice Questions

- Write a program that takes the radius of a sphere as input and outputs the following:

  - sphere’s diameter

  - circumference

  -surface area

  - volume

- Write a program that calculates and prints the number of minutes in a year.
- Light travels at 3×10^8 meters per second. A light year is the distance a light beam travels in one year. Write a program that calculates and displays the value of a light year.

Control Statements - **Loops**

- **Iteration**: Each repetition of the action is known as a pass or iteration.
- **Loops**: A Loop is a programming structure for iteration.
- There are two types of loops:
    1. Those that repeat an action a predefined number of times, called **definite** loops (or definite iteration).
    2. Those that perform the action until the program determines that it needs to stop, called **indefinite** loops (or indefinite iteration).

The **for** Loop

- Syntax of the **for** loop:

for <variable> in range (<an integer expression>):

<loop body statements>

- Note that the statements in the loop body must be indented and aligned in the same column.
- **Exampe1**: Print "Hello" 5 times with the for loop

```
for x in range (5):  #prints "Hello" 5 times with a newline
    print("Hello")
for x in range (5):  #prints "Hello" 5 times without a newline
    print("Hello", end=" ")
```

The **for** Loop (cont.)

- **Exampe2**: Print 0-5 with the for loop

```
for count in range (5): #prints each digit with a newline
      print("count = ", count, "and", range(5))
```

- Output of the above for loop program is shown below:

```
count =  0 and range(0, 5)
count =  1 and range(0, 5)
count =  2 and range(0, 5)
count =  3 and range(0, 5)
count =  4 and range(0, 5)
```

- **What did you understand from this Output?**

The **for** Loop (cont.)

- It means that, **the range function has two arguments, the first arguments is zero, and the latter one is an integer (a non zero value)**.
- Hence we can re-write the for loop program as below:

```
for count in range (0,5): #prints from 0 to 4 (5-1) in a newline
      print("count = ", count)
```

- The output of the above for loop program can be given as:

```
count =  0
count =  1
count =  2
count =  3
count =  4
```

for loop with Two Variables **in range** Function

- When two arguments are supplied to **range** function of the **for loop**, the count ranges from the first argument to the **second argument minus 1.**
- Syntax **for <variable> in range (<lower bound>, <upper bound>):**

  <loop body>

- Example1:

```
for count in range (1,5): # prints from 1 to 4 (5-1) with newline
    print(count)
```

```
count =  1
count =  2
count =  3
count =  4
```

for loop with Two Variables **in range** Function

- **Example2**: Get the lower and upper values and shows the sum of values from lower to upper.

```
lower = int(input(" Enter the lower bound: "))
upper = int(input(" Enter the upper bound: "))
sum = 0
for count in range (lower, upper + 1):
    print(count)
    sum += count
print(" The sum from ", lower, "to", upper, "is", sum)
```

Class Exercises

- Write a program that can find the factorial of a positive integer number.
- Get a string from user and display its characters ASCII values.

Analyzing the **range** function using the **list** function

- The **list** function can be used to analyze the meaning of the **range** function in a **for loop** by converting its elements as a list, [].

```
for count in range (5):  # prints digits from 0 to 4 (5-1) with newline
    print(count)
```

- The `range(5)` can be analyzed with the list function as shown below:

```
    print(list(range(5)))  # would output: [0, 1, 2, 3, 4] => a list
    print(list(range(1,5)))  # would output: [1, 2, 3, 4] => a list
```

- It means that the **`range(<expression>)`** of a **for loop** is just a **list** of elements, and it can be represented as **[1,2,3,……,n].**

Analyzing the **range** function using the **list** function

- Consider the following **for loop** program:

```
for number in range(1,6):
    print("number = ", number)
```

```
number =  1
number =  2
number =  3
number =  4
number =  5
```

- The `range (1,6)` is equivalent to `[1,2,3,4,5]` as shown below:

```
for number in [1, 2, 3, 4, 5]:
    print("number =", number)
```

```
number =  1
number =  2
number =  3
number =  4
number =  5
```

The **range** Function with a **Third Argument**

- The **range** function expects a **third argument** that allows you to skip some numbers from the loop result.
- The **third argument** specifies a **step value** or the **interval** between the number used in the range, as shown below:

```
for count in range(2, 11, 2):
    print("count = ", count)
```

```
count =  2
count =  4
count =  6
count =  8
count =  10
```

The **range** Function with a **Third Argument**

- A **for loop** that **counts down** with a three-argument **range** function.
- The following program would count from 10 to 1:

```
for count in range(10, 0, -1):
    print("count = ", count)
```

```
count =  10
count =  9
count =  8
count =  7
count =  6
count =  5
count =  4
count =  3
count =  2
count =  1
```

- **Write a program that can print from 10 to 0.**

The **range** Function with a **Third Argument**

- Consider the following for loop program:

```
for count in range(10, 0, -1):
    print(count, end="")
```

Output is not a list: 10 9 8 7 6 5 4 3 2 1

- which can be re-write into the following with the **list function** to get a result in the **list** form:

```
print(list(range(10, 0, -1)))
```

Output is a list: [10,9,8,7,6,5,4,3,2,1]

Exercise

- Write the outputs of the following loops:
  - ```
    for count in range(5):
            print(count + 1, end=" ")
    ```
  - ```
    for count in range(1, 4):
            print(count, end=" ")
    ```
  - ```
    for count in range(1, 6, 2):
            print(count, end=" ")
    ```
  - ```
    for count in range(6, 1, -1):
            print(count, end=" ")
    ```

Formatting Text for Output

- Many data-processing applications require output that has a **tabular format**.
- In this format, numbers and other information are aligned in columns that can be either **left-justified** or **right-justified**.
- The total number of data characters and additional spaces for a given datum in a formatted string is called its **field width**.

Formatting Text for Output

- The example, which displays the exponents 7 through 10 and the values of 10^7 through 10^{10} shows the format of two columns produced by the **print** function:

```
for x in range (7, 11):
    print(x, 10 ** x)
```

```
7 10000000
8 100000000
9 1000000000
10 10000000000
```

Formatting Text for Output

- The following code would show how to **right-justify** the output of the previous exponent program:

```
for x in range (1, 11):
    print("%4d%15d" % (x, 10 ** x))
```

- The following code would show how to **left-justify** the output of the previous exponent program:

```
for x in range (1, 11):
    print("%-4d%-18d" % (x, 10 ** x))
```

Case study: An Investment Report

- The input:
  1. Starting investment amount (float)
  2. Number of years (int)
  3. Interest rate (int)
- The report is displayed in tabular form with a header.
- The computations and outputs:
  - For each year, compute the interest and add it to the investment and print a formatted row of results for that year.
- The ending investment and interest earned are also displayed.

```
#Accept the inputs
startBalance = float(input("Enter the investment amount: "))
years = int(input("Enter the number of years: "))
rate = int(input("Enter the yearly rate in %: "))
#Convert the rate into a decimal
rate /= 100
#Initialize the total interest variable
totalInterest = 0.0
#Create the display header for the table
print("\n%4s%18s%10s%16s"% ("Year", "Starting balance", "Interest", "Ending
  balance"))
#Compute and display the result for each year
for year in range (1, years + 1):
    interest = startBalance * rate
    endbalance = startBalance + interest
    print("%4d%18.2f%10.2f%16.2f" % (year, startBalance, interest, endbalance))
    startBalance = endbalance
    totalInterest += interest
#Display the totals for the given period
print("Ending balance: $%0.2f" % endbalance)
print("Total interest earned: $%0.2f" % totalInterest)
```

The Boolean Type Comparison and Expressions

- The **Boolean** data type consists of only two data values:
  - **True**
  - **False**
- The python’s comparison operators, that cause **Boolean values** are listed below:

Comparison Operator	Meaning
==	Equals
!=	Not equal
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal

The **Boolean Type** Comparison and Expressions

- The following shows the examples of comparisons:

```
print(4 == 4)
print(4 == 5)
print(4 != 4)
print(4 != 5)
print(4 < 5)
print(4 < 3)
print(4 <= 4)
print(4 <= 5)
print(4 <= 3)
print(4 > 3)
print(4 > 5)
print(4 >= 4)
print(4 >= 5)
print(4 >= 3)
```

```
True
False
False
True
True
False
True
True
False
True
False
True
False
True
```

Selection: **if** and **if-else** Statements

- In **if/if else** statement, the computer must pause to examine or test a **condition**, which express a hypothesis about the state of its world at that point of time:
  - If the **condition** is **True**, the computer executes the first alternative action and skips the second alternative.
  - If the **condition** is **False**, the computer skips the first alternative, and executes the second alternative.

if, the one-way Selection Statement

- The simplest for of **selection** is the **if** statement. This type of control statement is called a **one-way selection** statement, because it consists of a condition and just a single sequence of statements.
  - If the **condition** is **True**, the sequence of statements is run.
  - Otherwise, control proceeds to the next statement following the entire selection statement.
- **Syntax** for the **if** statement:

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

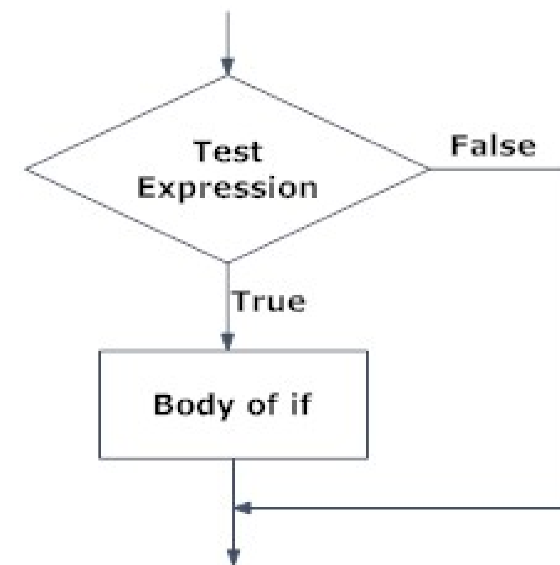


Fig: Operation of if statement

if, the one-way Selection Statement

- The following code would confirm your "A" grade:

```
mark = int(input(" Enter your final mark (out of 100): "))
if mark >= 90:
    print(" Your grade is ", "A")
```

```
Enter your final mark (out of 100): 90
 Your grade is  A
```

if, the one-way Selection Statement

- Get an integer number from user and if it is less than or equal to 10, then prints the range of numbers from 0 up to the number with their exponent with 10 in a right aligned format.

```
number = int(input(" Enter an integer number: "))
if number <= 10:
    for x in range(number + 1):
        print("%4d%10d" % (x, 10 ** x))
```

```
Enter an integer number: 7
   0         1
   1        10
   2       100
   3      1000
   4     10000
   5    100000
   6   1000000
   7  10000000
```

If - else, the two-way Selection Statement

- The **if-else statement** (also called a **two-way selection**) is the most common type of selection statement, because it directs the computer to make a choice between two alternative courses of action.
- Here is the Python **syntax** for the **if-else** statement:

if<condition/test expression>:

<body of if: sequence of statement>

else:

<body of else: sequence of statement>

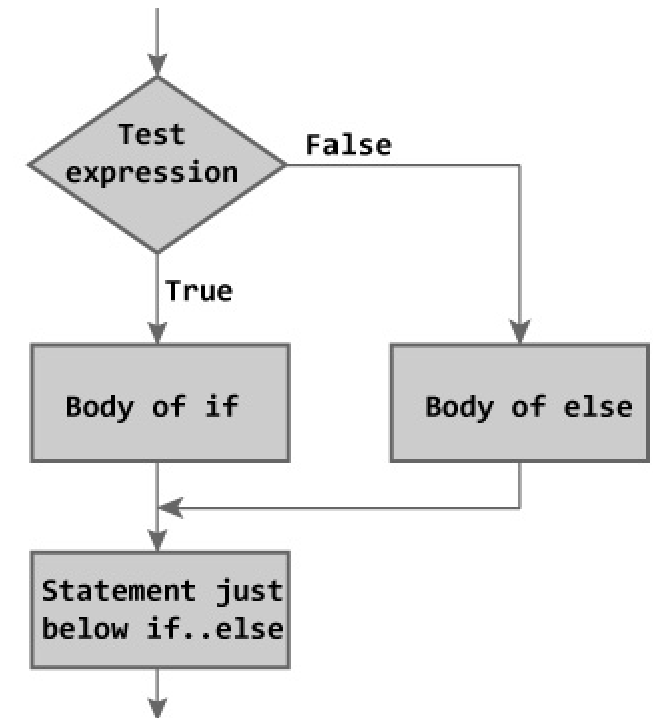


Figure: Flowchart of if...else Statement

If - else, the two-way Selection Statement

- Example1:

```
mark = int(input("Enter your final mark <out of 100>: "))
if mark >= 90:
    print("You have 'A' grade!")
else:
    print("Your grade is not A!")
```

If - else, the two-way Selection Statement

- Example2:

```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
if first >= second:
    maximum = first
    minimum = second
else:
    maximum = second
    minimum = first
print(" The maximum is ", maximum)
print(" The minimum is ", minimum)
```

```
Enter the first number: 23
Enter the second number: 45
 The maximum is  45
 The minimum is  23
```

If - else, the two-way Selection Statement

- Example 3:

```
import math
area = float(input("Enter the area of the circle: "))
if area > 0:
    radius = math.sqrt(area / math.pi)
    print("The radius of the circle is %0.2f" % radius)
else:
    print("Error, the area must be a positive number!")
```

```
Enter the area of the circle: 4536.89
The radius of the circle is 38.00
```

Multi-way **if** Statements

- The **multi-way if statement** is useful when a program is faced with testing several conditions that entail more than two alternative courses of action.
- The **multi-way if statement** considers each condition until one evaluates to **True** or they all evaluate to **False**. The Python **syntax** is the following:

```
if <condition1>:
        <sequence of statement1>
elif <conditionn>:
        <sequence of statementn>
else:
        <default sequence of statements>
```

Multi-way **if** Statements

- Example1: Consider the problem of converting marks to letter grads, based on the following information:
  - Grade “A” = all marks above 89
  - Grade “B” = all marks above 79 and below 90
  - Grade “C” = all marks above 69 and below 80
  - Grade “F” = all marks below 70

```
mark = int(input(" Enter your final mark <out of 100>: "))
if mark > 89:
    grade = "A"
elif mark > 79:
    grade = "B"
elif mark > 69:
    grade = "C"
else:
    grade = "F"
print(" Your garde is ", grade)
```

Multi-way **if** Statements

- Often a course of action must be taken if either of two conditions is true.
- For example, valid inputs to a program often lie within a given range of values.
- Any input above this range should be rejected with an error message, and any input below this range should be dealt with in a similar fashion.

Multi-way **if** Statements

```
mark = int(input(" Enter your final mark <out of 100>: "))
if mark > 100:
    print("Error! The mark must be between 0 and 100.")
elif mark < 0:
    print("Error! The mark must be between 0 and 100.")
else:
    if mark > 89:
        grade = "A"
    elif mark > 79:
        grade = "B"
    elif mark > 69:
        grade = "C"
    else:
        grade = "F"
    print(" Your garde is ", grade)
```

```
 Enter your final mark <out of 100>: 120
Error! The mark must be between 0 and 100.

 Enter your final mark <out of 100>: -30
Error! The mark must be between 0 and 100.

 Enter your final mark <out of 100>: 78
 Your garde is  C
```

Logical Operators and Compound Boolean Expressions

- Note that the first two conditions (from the previous multi-way if program) are associated with identical actions.
- The two conditions can be combined in a **Boolean expression** that uses the logical operator **or**.
- The resulting compound **Boolean expression** is given as:

```
mark = int(input(" Enter your final mark <out of 100>: "))
if mark > 100 or mark < 0:
    print("Error! The mark must be between 0 and 100.")
else:
     # the code to compute grade here
```

Logical Operators and Compound Boolean Expressions

- Yet another way to describe this situation is to use the Boolean logical operator **and:**

```
mark = int(input(" Enter your final mark <out of 100>: "))
if mark >= 0 and mark <= 100:
    if mark > 89:
        grade = "A"
    elif mark > 79:
        grade = "B"
    elif mark > 69:
        grade = "C"
    else:
        grade = "F"
    print(" Your garde is ", grade)
else:
    print("Error! The mark must be between 0 and 100.")
```

Logical Operators and Compound **Boolean Expressions**

- Python includes three **Boolean logical** operators, **and**, **or**, and **not**.
- Both the **and**, and **or** operators expect two operands.
  - The **and** operator returns **True** if and only if both of its operands are **true**, and returns **False** otherwise.
  - The **or** operator returns **False** if and only if both of its operands are **false**, and return **True** otherwise.
  - The **not** operator expects a single operand and returns its **logical negation**; **True** if it's **false**, and **False** if it's **true**.

```
A = True
B = False
print(A and B)
print(A or B)
print(not A)
```

```
False
True
False
```

Operator Precedence from Highest to Lowest

Operator	Symbol
Exponentiation	**
Arithmetic negation	–
Multiplication, division, remainder	*, /, %
Addition, subtraction	+, –
Comparison	==, !=, <, >, <=, >=
Logical negation	not
Logical conjunction and disjunction	and , or
Assignment	=

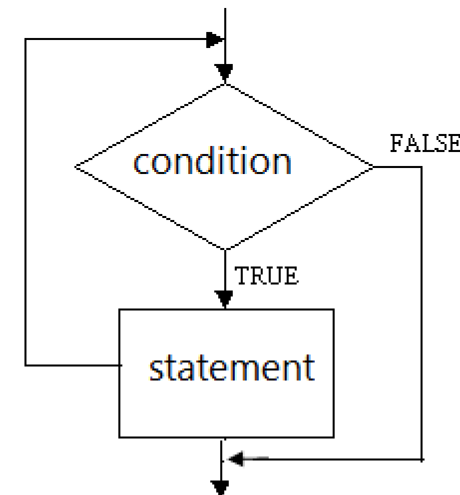
Conditional Iteration: The **while** loop

- Earlier we examined the **for loop**, which executes a set of statements a definite number of times specified by the programmer.
- In many situations, the number of iterations in a loop is unpredictable.
- The loop eventually completes its work, but only when a condition changes.
  - For example, the user might be asked for a set of input values. The program's input loop accepts these values until the user enters a special value or **sentinel** that terminates the loop.
  - This type of process is called **conditional iteration**.
- This section explores the **while loop** to describe conditional iteration.

The Structure and Behavior of a **while** Loop

- **Conditional iteration** requires that a condition be tested within the loop to determine whether the loop should continue.
- Such a condition is called the loop's **continuation condition**:
  - If the **continuation condition** is **false**, the loop ends.
  - If the **continuation condition** is **true**, the statements within the loop body are executed again.
- Syntax for the **while loop**:

```
while<condition>:
        <statements in the loop body>
```



while loop: Examples

- Get a set of numbers from the user until the user press the enter key (return key) and prints their sum. The program recognize this value (enter key value) as the empty string.
- **Pseudocode algorithm**:

```
set the sum to 0.0
input a string
while the string is not the empty string
     convert the string to a float
     add the float to the sum
     input a string
print the sum
```

while loop: Examples

- Here is the Python code:

```
sum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    sum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is ", sum)
```

```
Enter a number or just enter to quit: 1
Enter a number or just enter to quit: 2
Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit:
The sum is  10.0
```

Count Control with a **while** Loop

- You can also use a **while** loop for a **count-controlled loop** as a **for** loop. For example see a summation code with a for loop and a while loop below:

```
sum = 0.0
for x in range(1, 1001):
    sum += x
print("Sum of numbers from 1 to 1000 is ", sum)
```

- The same program with a while loop:

```
sum = 0.0
lcv = 1
while lcv <= 1000:
    sum += lcv
    lcv += 1
print("The sum of numbers from 1 to 1000 is ", sum)
```

Count Control with a **while** Loop

- By contrast, a **for** loop specifies the control information concisely in the header and automates its manipulation behind the scenes.
- The next example shows how the for loop and while loop are supporting in a count down application:

```
for x in range(10, 0, -1):
    print(x)
```

- Count down with while loop:

```
LCV = 10
while LCV >= 1:
    print(LCV)
    LCV -= 1
```

The true **while** loop with **break** Statement

- Python includes a **break** statement that will allow us to break a true while loop (an infinite loop) with **if – else** statement:

```
sum = 0.0
while True:
    number = input("Enter a number or just enter to quit: ")
    if number != "":
        sum += float(number)
    else:
        break
print(sum)
```

The true **while** loop with **break** Statement

- The previous true **while loop** script with a **break** statement can be modified with an **if** statement:

```
sum = 0.0
while True:
    number = input("Enter a number or just enter to quit: ")
    if number == "":
        break
    sum += float(number)
print(sum)
```

The true **while** loop with **break** Statement

- The next example modifies the input section of the grade-conversion program to continue taking input numbers from the user until the user enters an acceptable value:

```
while True:
    mark = int(input("Enter your total mark <0-100>: "))
    if mark >= 0 and mark <= 100:
        break
    else:
        print("Error! The mark must be between 0 and 100.")
if mark > 89:
    print("Your grade is A.")
elif mark > 79:
    print("Your grade is B.")
elif mark > 69:
    print("Your garde is C.")
else:
    print("Your grade is F.")
```

```
Enter your total mark <0-100>: 345
Error! The mark must be between 0 and 100.
```

```
Enter your total mark <0-100>: 97
Your grade is A.
```

Exercises

- Translate the following **for** loops to equivalent **while** loops:

1.
```
for count in range(100):
    print(count)
```

2.
```
for count in range(1, 101):
        print(count)
```

3.
```
for count in range(100, 0, -1):
        print(count)
```

Exercises

- Write a **while** loop that computes the factorial of a given integer *N*.
- The $\log_2$ of a given number *N* is given by *M* in the equation $N = 2^M$. The value of *M* is approximately equal to the number of times *N* can be evenly divided by 2 until it becomes 0. Write a loop that computes this approximation of the log2 of a given number *N*.

Random Numbers

- Python's **random** module supports the random value generation. The function **randint** (in **random** module) returns a random number from among the numbers between the two arguments and including those numbers.
- **Syntax:**

```
import random
rand_value = random.randint(start_integer, final_integer)
```

- For example, see the results from rolling a die 10 times:

```
import random
for x in range(0, 10):
    value = random.randint(1, 6)   #print random values
    print(value)                   # including 1 and 6
```

Random Numbers

- Write a guessing program that allows the user to enter a smaller number and a larger number and guess the **randint** function generated value from the smaller and the larger numbers.

Exercises

- 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.
- 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 a right triangle (from the Pythagorean theorem that in a right triangle, the square of one side equals the sum of the squares of the other two sides.
- Write a program that receives a series of numbers from the user and allows the user to press the enter key to indicate that he/she is finished providing inputs. After the user presses the enter key, the program should print the **sum** of the numbers and their **average**. Finally, the program should check whether the integer value of the average is an **even** or **odd** or a **prime** value.

Strings and Text Files

- In this section, we explore strings and text files, which are useful data structures for organizing and processing text.
- Much about computation is concerned with manipulating text.
- After understanding this section, you will be able to:
  - Access individual characters in a string, Retrieve a substring from a string, Search for a substring in a string, Convert a string representation of a number from one base to another base, and Use string methods to manipulate strings.
  - Open a text file for output and write strings or numbers to the file, and Open a text file for input and read strings or numbers from the file.
  - Use library functions to access and navigates a file system.

The Structure of Strings: **len** function

- A string is a **data structure**. A data structure is a compound unit that consists of several smaller pieces of data.
- A **string** is a sequence of zero or more characters.
- A string’s length is the number of characters it contains. Python’s **len** function returns **length value** (no. of characters) when it is passed a string.
- Usage of **len** function: **len(string)**

```
length = len("Hello")
print("Length of \“Hello\" is", length)
```

```
Length of “Hello" is 5
```

The Structure of Strings: **len** function

- The position of a string’s characters are numbered from 0, on the left, to the length of the string minus 1.
- See the position of characters in the string “**Hi there!**”:

H	I		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

- The string is an **immutable data structure**. This means that its internal data elements, the characters can be accessed, but the structure itself cannot be modified.

The Subscript Operator

- The form of a subscript operator is the following:

  <a string>[<an integer expression>]

- For example:

```
name = "Alan Turing"
print("The first character in \"" + name + "\"is", name[0])
```

```
The first character in "Alan Turing"is A
```

- Get a string from the user and print its last character.

The Subscript Operator

- The following code shows how a count-controlled loop displays the characters and their positions of a string:

```
name = "Alan Turing"
for char in range(len(name)):
    print(char, name[char])
```

```
0 A
1 l
2 a
3 n
4
5 T
6 u
7 r
8 i
9 n
10 g
```

Slicing for Substrings

- Here are some examples that show how slicing is used:

```
name = "Alan Turing"
print(name[-1])
print(name[-2])
print(name[-3])
print(name[0:])
print(name[0:1])
print(name[0:2])
print(name[-3:])
print(name[:len(name)])
```

```
g
n
i
Alan Turing
A
Al
ing
Alan Turing
```

Testing for a Substring with the **in** Operator

- Suppose you want to separate filenames with a **.txt** extension. A slice would work for this application, by using Python's **in** operator.
- The operator **in** returns **True** if the target string is somewhere in the search string, or **False** otherwise.
- The following sample code shows how to separate filenames with **.txt** from a list of various filenames:

```
fileList = ["anil.exe", "data.txt", "function.exe",
 "name.txt", "class.txt"]
for file in fileList:
    if ".txt" in file:
        print(file)
```

```
data.txt
name.txt
class.txt
```

Exercises

1. Assume that the variable **data** refers to the string **“myprogram.exe”**. Write the values of the following expressions:
   1. data[2]
   2. data[-2]
   3. len(data)
   4. data[0:8]
2. Assume that the variable **myString** refers to a string. Write a code segment that uses a loop to print that characters of the string in reverse order.
3. Assume that the variable **myString** refers to a string and the variable **reversedString** refers to an empty string. Write a loop that adds the characters from **myString** to **reversedString** in a reverse order.

Strings and Number System

- **Converting Binary to Decimal**
  - We can code an algorithm for the conversion of a binary number to the equivalent decimal number as a Python script.
  - The input to the script is a string of bits, and its output is the integer that the string represents.
    - The algorithm uses a loop that accumulates the sum of a set of integers.
    - The sum is initially 0.
    - The exponent that corresponds to the position of the string's leftmost bit is the length of the bit string minus 1.
    - The loop visits the digits in the string from the first to the last (left to right), also counting from the largest exponent of 2 down to 0 as it goes.
    - Each digit is converted to its integer value (1 or 0), multiplied by its positional value, and the result is added to the ongoing total.
    - A positional value is computed by using ** operator.

Strings and Number System

- **Converting Binary to Decimal script:**

```
binary = input("Enter a binary string: ")
copy = binary
decimal = 0
exponent = len(binary) - 1
for digit in binary:
    decimal += int(digit) * (2 ** exponent)
    exponent -= 1
print("The decimal equivalent of" + binary + “is", decimal)
```

```
Enter a binary string: 11111111
The decimal equivalent of 11111111 is 255
```

Strings and Number System

- **Converting Decimal to Binary**:
  - This algorithm repeatedly divides the decimal number by 2.
  - After each division, the remainder (either a 0 or a 1) is placed at the beginning of a string of bits.
  - The quotient becomes the next dividend in the process.
  - The string of bits is initially empty, and the process continues while the decimal number is greater than 0.
  - The script expects a non-negative decimal integer as an input and prints the equivalent bit string.

Strings and Number System

- **Converting Decimal to Binary Script:**

```
decimal = int(input("Enter a positive integer: "))
copy = decimal
if decimal == 0:
    print("The decimal equivalent of " + str(copy) + " is", 0)
else:
    binary = "" # empty string
    while decimal > 0:
        remainder = decimal % 2
        decimal //= 2
        binary = str(remainder) + binary
    print("The decimal equivalent of " + str(copy) + " is " + binary)
```

```
Enter a positive integer: 255
The decimal equivalent of 255 is 11111111
```

String Methods

- Let’s start with counting words in a single sentence and finding the average word length.
- This task requires locating the words in a string.
- For supporting this types of applications, Python includes a set of string operations called **methods**.

String Methods: **split**

- Syntax of **split** method:

  <string_object>.<split>(<argument$_1$, argument$_2$,…., argument$_n$>)

- We use the string method **split** to obtain a list of the words contained in an input string:

```
myString = "Have a nice day my dear!"
wordList = myString.split()
print("wordList = ", wordList)
print("There are ", len(wordList), “words in the list.")
```

```
wordList =  ['Have', 'a', 'nice', 'day', 'my', 'dear!']
There are  6 words in the list.
```

String Methods: **split**

- We use the string method **split** to obtain a list of the words contained in an input string. Then we print the length of the list, which equals the number of words, and computes the average of the length of the words in the list.