MODULE 1

Python Basics: Entering Expressions into the Interactive Shell, The Integer, Floating-Point, and String Data Types, String Concatenation and Replication, Storing Values in Variables, Your First Program, Dissecting Your Program, **Flow control:** Boolean Values, Comparison Operators, Boolean Operators, Mixing Boolean and Comparison Operators, Elements of Flow Control, Program Execution, Flow Control Statements, Importing Modules, Ending a Program Early with sys.exit(), **Functions:** def Statements with Parameters, Return Values and return Statements, The None Value, Keyword Arguments and print(), Local and Global Scope, The global Statement, Exception Handling, A Short Program: Guess the Number

Textbook 1: Chapters 1 – 3

What Is Programming?

Programming is the act of entering instructions for the computer to perform some task.

What Is Python?

Python is a high- programming language (close to natural language like English) with syntax rules for writing what is considered valid Python code and the Python interpreter software that reads source code (written in the Python language) and convert it into machine code to performs the instructions.

Why the name Python?

Founder of python – Guido Van Rossum in 1991 read scripts of ***"Monty Python's flying circus"***, a BBC comedy series and thought that he needed a short, unique name – so named the programming language as python. Python - not from snakes. Python programmers are affectionately called Pythonistas.

Some facts (anxieties) about programming :-

1) Programmers Don't Need to Know Much Math

Most programming doesn't require math beyond basic arithmetic. In fact, being good at programming isn't that different from being good at solving Sudoku puzzles. Just because Sudoku involves numbers doesn't mean you have to be good at math to figure out the solution. The same is true of programming. And like all skills, the more you program, the better we'll become.

2) You Are Not Too Old to Learn Programming

The second most common anxiety about learning to program is that people think they're too old to learn it. It's important to have a "growth mindset" about programming—in other words, understand that people develop programming skills through practice. They aren't just born as programmers, and being unskilled at programming now is not an indication that we can never become an expert.

3) Programming Is a Creative Activity

Programming is a creative task, like painting, writing, knitting, or constructing LEGO castles.

Downloading and Installing Python

We can download Python for Windows, macOS, and Ubuntu for free at **https://python.org/downloads/.**

While the Python interpreter is the software that runs Python programs, the Mu editor software is where we'll enter our programs, much the way we type in a word processor.

Mu – editor for writing python code.

IDLE - IDLE is another editor for writing Python code. The Integrated Development and Learning Environment (IDLE) software installs along with Python

The Interactive Shell

On opening the IDLE app, a shell or a window opens just like the Terminal in macOS or Command Prompt on Windows. This window is called interactive shell that lets us enter instructions.

Once IDLE app is opened, we can see a **>>>.**

in MU and >>> are called prompts.

>>> [cursor will blink] –awaits for the user to give input.

>>> print('Hello, world!')

builtins module in python is imported by default, so no need to import.
Ex – print, pow, min, max methods (functions) present in builtins module.

After we type that line and press ENTER, the interactive shell should display this in response:

Hello, world!

IDLE opens in 2 modes –

1) Interactive shell mode – also called REPL (Read-Evaluate-Print-Loop) : opens interactive shell that runs python instructions one at a time and instantly shows us the result. Prompt >>> will be shown.
2) Script mode or file editor mode : opens file editor window that lets us type many instructions, save the file and run (F5) the program. Prompt >>> will not be shown.

CHAPTER 1: PYTHON BASICS

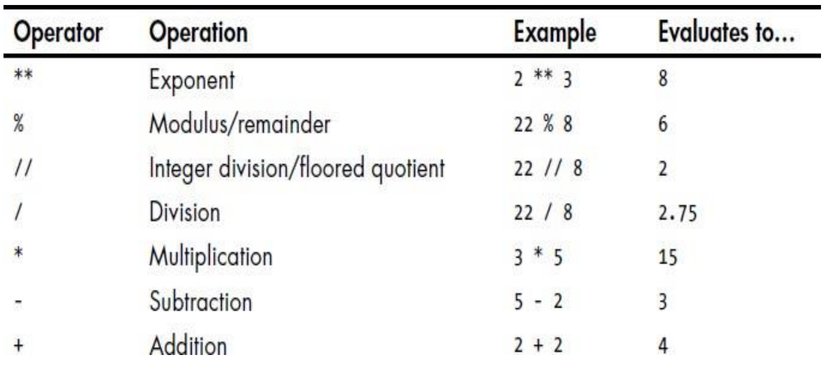
1. Entering expressions into the interactive shell
2. The integer, floating-point and String Data Types
3. String concatenation and replication
4. Storing values in variables
5. Your first program
6. Dissecting your program

Python Programming language has a wide range of syntactical constructions, standard library functions and Interactive development environment features.

1.1. Entering expressions into the interactive shell

- On launching the IDLE app, we can see a window with the >>> prompt should appear; that's the interactive shell.

```
>>> 2 + 2
4
```



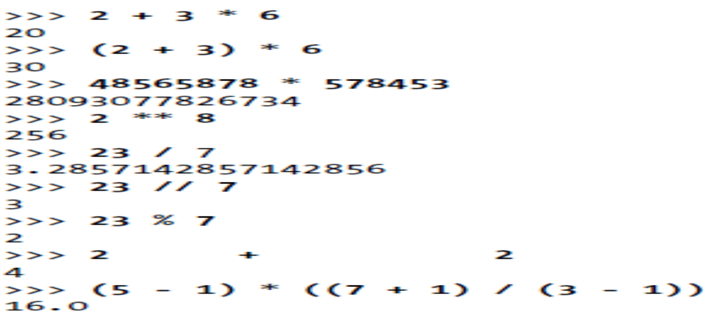
Operator	Operation	Example	Evaluates to...
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

- In Python, 2 + 2 is called an *expression*, which is the most basic kind of programming instruction in the language.
- Expressions consist of *values* (such as 2) and *operators* (such as +), and they can always *evaluate* (that is, reduce) down to a single value.
- That means we can use expressions anywhere in the python code that we could also use a value.
- A single value with no operators isalso considered an expression.
- The other operators which can be used are:

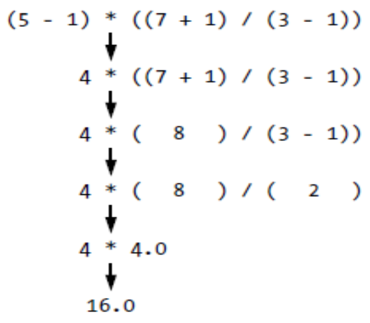
- The order of operations (also called precedence) of Python math operators is similar to that of mathematics.
- The ** operator is evaluated first, associativity is from right to left; the *, /, //, and % operators are evaluated next, associativity is from left to right; and the + and - operators are evaluated last (also from left to right associativity).

  Precedence – order of evaluation if the expression has different operators.

  Associativity - order of evaluation if the expression has operators of same precedence.
- We can use parentheses to override the usual precedence if we need to.



```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2       +          2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```



```
(5 - 1) * ((7 + 1) / (3 - 1))
      ↓
      4 * ((7 + 1) / (3 - 1))
      ↓
      4 * (   8   ) / (3 - 1))
      ↓
      4 * (   8   ) / (  2  )
      ↓
      4 * 4.0
      ↓
      16.0
```

Figure 1-1: Evaluating an expression reduces it to a single value.

Due to wrong instructions errors occurs as shown below:

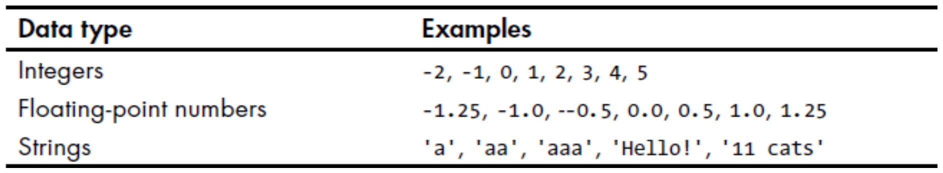
>>> 2+
SyntaxError:

Expressions are values combined with operators and they always evaluate down to a single value.

1.2 The integer, floating-point and String Data Types

- A data type is a category for values, and every value belongs to exactly one data type.

Table 1-2: Common Data Types



Data type	Examples
Integers	-2, -1, 0, 1, 2, 3, 4, 5
Floating-point numbers	-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25
Strings	'a', 'aa', 'aaa', 'Hello!', '11 cats'

- The integer (or int) data type indicates values that are whole numbers both positive and negative.
- Numbers with a decimal point, such as 3.14, are called floating-point numbers (or float).

- 42 is an integer, the value 42.0 would be a floating-point number.
- Anything (any text) enclosed in single (‘ ’) or double quote (“ “) is a string (or str).
- The string with no characters, ' ', called a blank or empty string.

>>> ‘Hello world!
SyntaxError:

1.3 String concatenation and replication

- + is the addition operator when it operates on two integers or floating-point values.

```
>>> 2 + 2
4
```

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

- However, when + is used on two string values, it joins the strings as the string concatenation operator.
- If we try to use the + operator on a string and an integer value, Python will not know how to handle this,and it will display an error message.

>>> ‘Alice’ + 42
TypeError:

Solution to this problem - We have to convert into string because python cannot do this automatically. >>> ‘Alice’ + str(42)

- The * operator is used for multiplication when it operates on two integer or floating-point values.
- But, when the * operator is used on one string value and one integer value, it becomes the stringreplication operator.

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The * operator can be used with only two numeric values (for multiplication) or one string value and oneinteger value (for string replication). Otherwise, Python will just display an error message.

>>> ‘Alice’ * ‘Bob’
TypeError:
>>> ‘Alice’ * 5.0
TypeError:

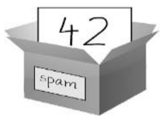
1.4 Storing Values in Variables

- A variable is like a box in the computer’s memory where you can store a single value.
- If we need to use variables later, then the result must be stored in variable. Ex – x, spam

Assignment Statements

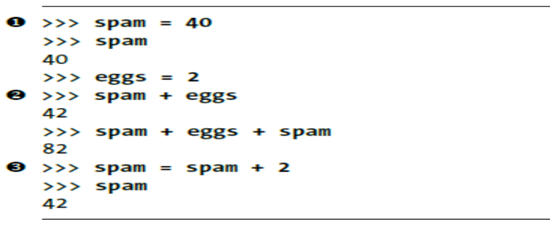
- We will store values in variables with an assignment statement.
- An assignment statement consists of a variable name, an equal sign (called the assignment operator), andthe value to be stored.
- Ex: spam = 42, then the variable name called spam will have integer value 42 stored in it.

spam



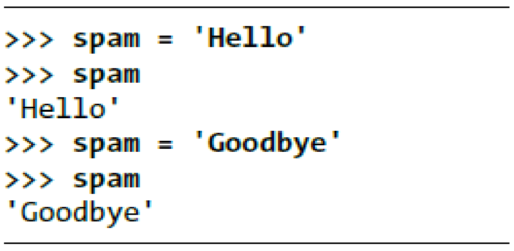
Variable is a labelled box that a value is placed in.

- A variable is initialized (or created) the first time a value is stored in it.
- After that, we can use it in expressions with other variables and values.
- When a variable is assigned a new value, the old value is forgotten, which is why spam evaluatedto 42 instead of 40 at the end of the example.

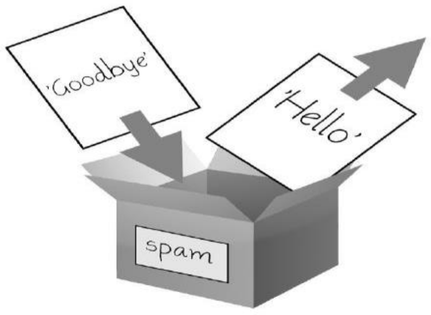


```
❶ >>> spam = 40
  >>> spam
  40
  >>> eggs = 2
❷ >>> spam + eggs
  42
  >>> spam + eggs + spam
  82
❸ >>> spam = spam + 2
  >>> spam
  42
```

One more example



```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```



Variable names

- A good variable name describes the data it contains.

  (book examples – spam, eggs are the variable names – Monty Python Spam Sketch)
- A descriptive name will help make our code readable.
- We can name a variable anything as long as it obeys the following three rules:

1. It can be only one word.

2. It can use only letters, numbers, and the underscore (_) character.
3. It can’t begin with a number.

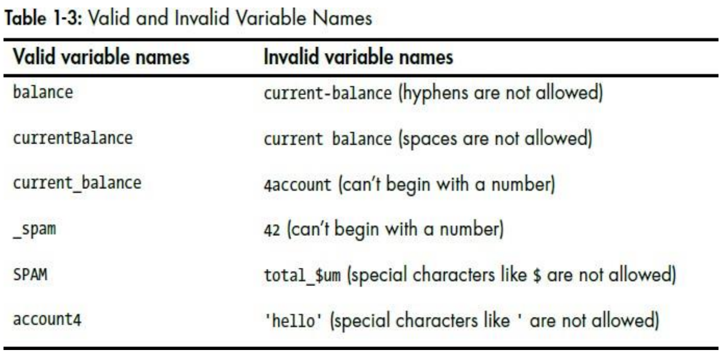


Table 1-3: Valid and Invalid Variable Names

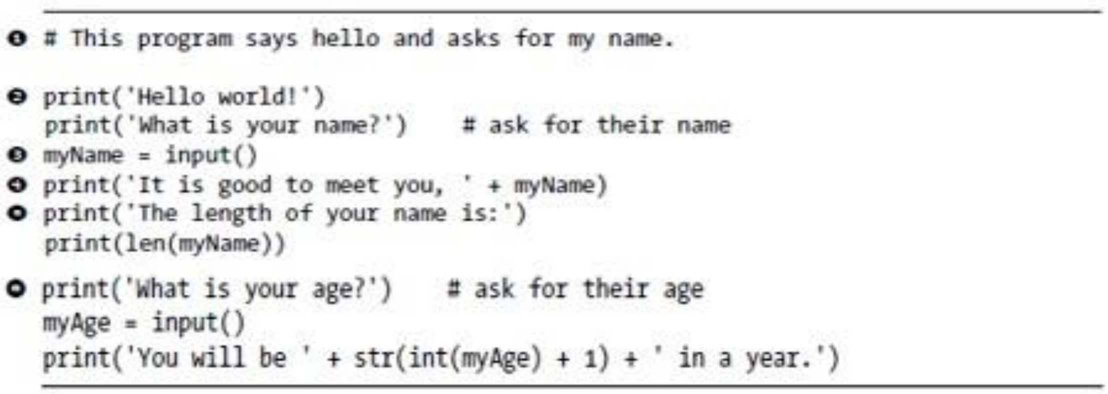
Valid variable names	Invalid variable names
balance	current-balance (hyphens are not allowed)
currentBalance	current balance (spaces are not allowed)
current_balance	4account (can’t begin with a number)
_spam	42 (can’t begin with a number)
SPAM	total_$um (special characters like $ are not allowed)
account4	'hello' (special characters like ' are not allowed)

- ➢ Variable names are case-sensitive, meaning that spam, SPAM, Spam, and sPaM are four differentvariables.
- ➢ This book uses camelcase for variable names instead of underscores; that is,variables lookLikeThis instead of looking_like_this.

1.5 Your First Program

Open a file editor window and enter the following program, click the save button and save it as hello.py – once saved, let us run the program by clicking run button or pressing F5 key. The program runs and shows the output in the interactive shell window.

Example program:


```
❶ # This program says hello and asks for my name.

❷ print('Hello world!')
  print('What is your name?')    # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
  print(len(myName))
❻ print('What is your age?')    # ask for their age
  myAge = input()
  print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

- ➢ The output looks like:

```
>>>
Hello, world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2
What is your age?
4
You will be 5 in a year.
>>>
```

1.6 Dissecting Your Program

Comments *(description of our program)*

```
❶ # This program says hello and asks for my name.
```

- ➢ Python ignores comments, and we can use them to write notes or remind us what the code is trying to do.
- ➢ Any text for the rest of the line following a hash mark (#) is part of a comment.
- ➢ # can be put in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code.
- ➢ The line with # can be removed later when we are ready to put the line back in.

(We can add as many blank lines in our program – to make our code easier to read like paragraphs in book)

The print() Function

- ➢ The print() function displays the string value inside the parentheses on the screen.

```
❷ print('Hello world!')
  print('What is your name?') # ask for their name
```

```
Hello World!
What is Your name?
```

- ➢ The line print('Hello world!') means "Print out the text in the string 'Hello world!'."
- ➢ When Python executes this line, we say that Python is *calling* the print() function and the string value isbeing *passed* to the function.
- ➢ A value that is passed to a function call is an *argument.*
- ➢ When we write a function name, the opening and closing parentheses at the end identify it as the name of the function.

We can also use this function to put a blank line on the screen; just call print() with nothing in between the parentheses.

The Input Function

- The input() function waits for the user to type some text on the keyboard and press ENTER.

```
❸ myName = input()
```

- This function call evaluates to a string equal to the user's text, and assigns the myName variable to this string value.
- If the user entered 'Sonia', then the expression would evaluate to myName = 'Sonia'.

Printing the User's Name

```
❹ print('It is good to meet you, ' + myName)
```

- If 'Sonia' is the value stored in myName, then this expression evaluates to 'It is good to meet you, Sonia'.

The len() Function

- We can pass the len() function a string value (or a variable containing a string), and the function evaluatesto the integer value of the number of characters in that string.
- len(myName) evaluates to an integer. It is then passed to print() to be displayed on the screen.

```
❺ print('The length of your name is:')
   print(len(myName))
```

```
>>> print('I am ' + 29 + ' years old.')
```

```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

```
TypeError:
```

Python gives an error because we can use the + operator only to add two integers together or concatenatetwo strings. We can't add an integer to a string because this is ungrammatical in Python.

The str(), int() and float() Functions

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

- Casting – Converting from one datatype of a value to another type is called casting.

- The str(), int(), and float() functions will evaluate to the string, integer, and floating-point forms of the value we pass, respectively.

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
```

```
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

```
>>> str(3)
‘3’
```

- The str() function is handy when we have an integer or float that we want to concatenate to a string.
- The int() function is also helpful if we have a number as a string value that we want to use in somemathematics.
- The int() function is useful to round a floating point number.

```
>>> spam = input()
101
>>> spam
'101'
```

- The value stored inside spam isn’t the integer 101 but the string '101'.
- If we want to do math using the value in spam, use the int() function to get the integer form of spam andthen store this as the new value in spam.

```
>>> spam = int(spam)
>>> spam
101
```

```
>>> spam * 10 / 5
202.0
```

- Now we should be able to treat the spam variable as an integer instead of a string.
- If we pass a value to int() that it cannot evaluate as an integer, Python will display an error message.

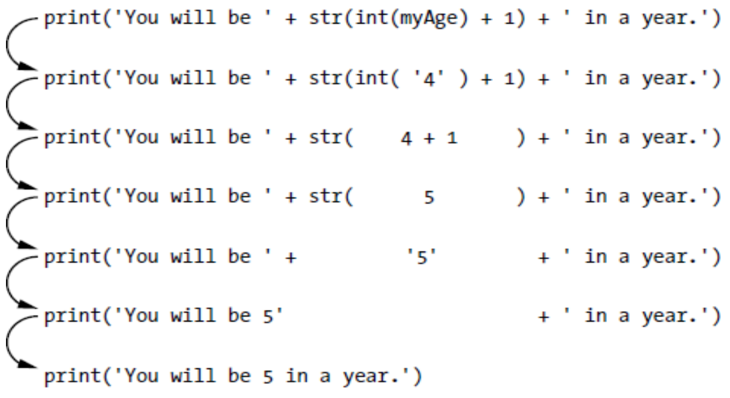
```
>>> int(‘twelve’)
ValueError:
>>> int(’99.99’)
ValueError:
>>> float(‘twelve’)
ValueError:
>>> float(’99.99’)
99.99
```

- The int() function is also useful if we need to round a floating-point number down. If we want to round afloating-point number up, just add 1 to it afterward.

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

```
❻ print('What is your age?') # ask for their age
  myAge = input()
  print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

- In our program, we used the int() and str() functions to get a value of the appropriate data type for the code.
- The myAge variable contains the value returned from input().
- Because the input() function always returns a string (even if the user typed in a number), we can usethe int(myAge) code to return an integer value of the string in myAge.



```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
print('You will be ' + str(int( '4' ) + 1) + ' in a year.')
print('You will be ' + str(    4 + 1     ) + ' in a year.')
print('You will be ' + str(      5       ) + ' in a year.')
print('You will be ' +          '5'        + ' in a year.')
print('You will be 5'                      + ' in a year.')
print('You will be 5 in a year.')
```

Figure 1-4: The evaluation steps, if 4 was stored in myAge

Text and Number Equivalence

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

Python makes this distinction because strings are text, while integers and floats are both numbers.

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

CHAPTER 2: FLOW CONTROL

1. Boolean Values
2. Comparison Operators
3. Boolean Operators
4. Mixing Boolean and Comparison Operators
5. Elements of Flow Control
6. Program Execution
7. Flow Control Statements
8. Importing Modules
9. Ending a Program Early with sys.exit()

Introduction

- ➢ We almost never want our programs to start from the first line of code and simply execute every line, straight to the end. In fact, based on how expressions evaluate, a program can decide to skip instructions, repeat them, or choose one of several instructions to run.
- ➢ Flow control statements can decide which python instructions to execute under which conditions.
- ➢ These flow control statements directly correspond to the symbols in a flowchart.
- ➢ In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program.

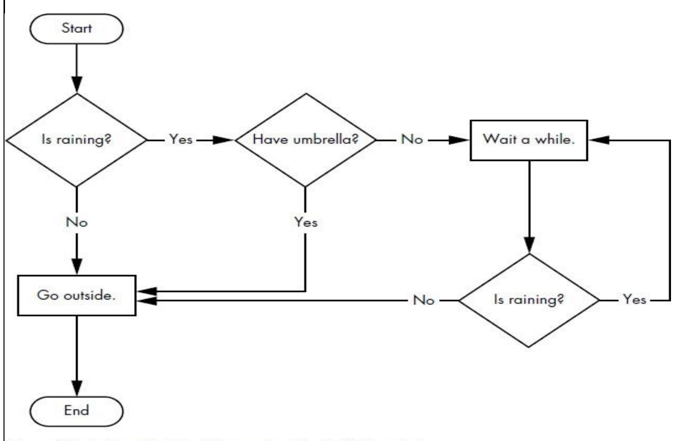


Figure 2-1: A flowchart to tell you what to do if it is raining

Symbol Name	Symbol	function
Oval	⬭	Used to represent start and end of flowchart
Parallelogram	▱	Used for input and output operation
Rectangle	▭	Processing: Used for arithmetic operations and data-manipulations
Diamond	◇	Decision making.
Arrows	← ↑ ↓ →	Flow line Used to indicate the flow of logic by connecting symbols
Circle	○	Page Connector

2.8 Boolean Values

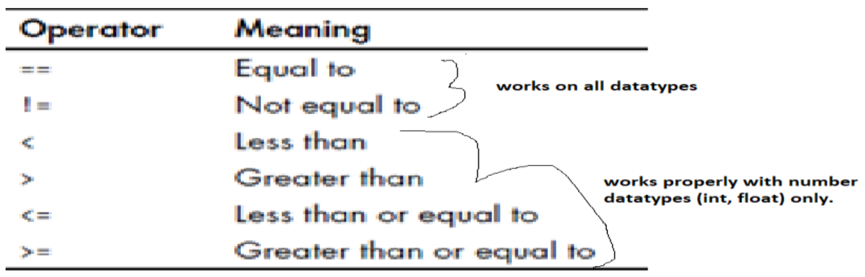
- The integer, floating-point and string data types have an unlimited number of possible values.
- The Boolean data type has only two values: True and False.
- When typed as Python code, the Boolean values True and False lack the quotes we place aroundstrings, and they always start with a capital T or F, with the rest of the word in lowercase.
- Examples:

```
❶ >>> spam = True
   >>> spam
   True
❷ >>> true
   Traceback (most recent call last):
     File "<pyshell#2>", line 1, in <module>
       true
   NameError: name 'true' is not defined
❸ >>> True = 2 + 2
   SyntaxError: assignment to keyword
```

- Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If wedon't use the proper case ❷ or we try to use True and False for variable names ❸, Python will give us an error message.

2.9 Comparison Operators

- Comparison operators compare two values and evaluate down to a single Boolean value.



Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

works on all datatypes
works properly with number datatypes (int, float) only.

- The above table lists the comparison operators.
- These operators evaluate to True or False depending on the values we give them.
- The == and != operators can actually work with values of any data type.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

```
  >>> 'hello' == 'hello'
  True
  >>> 'hello' == 'Hello'
  False
  >>> 'dog' != 'cat'
  True
  >>> True == True
  True
  >>> True != False
  True
  >>> 42 == 42.0
  True
❶ >>> 42 == '42'
  False
```

- An integer or floating-point value will always be unequal to a string value. The expression 42== '42' ❶ evaluates to False because Python considers the integer 42 to be different from thestring '42'.
- The <, >, <=, and >= operators work properly only with integer and floating-pointvalues.

```
  >>> 42 < 100
  True
  >>> 42 > 100
  False
  >>> 42 < 42
  False
  >>> eggCount = 42
❶ >>> eggCount <= 42
  True
  >>> myAge = 29
❷ >>> myAge >= 10
  True
```

The Difference Between the == and = Operators

- The == operator (equal to) asks whether two values are the same as each other.
- The = operator (assignment) puts the value on the right into the variable on the left.

2.10 Boolean Operators

- The three Boolean operators (and, or, and not) are used to compare Boolean values.
- Like comparison operators, they evaluate these expressions down to a Boolean value.

Binary Boolean Operators

- The **and** and the **or** operators always take two Boolean values (or expressions), so they're considered binary Operators.

and operator: The and operator evaluates an expression to True if both Boolean values are True; otherwise, it evaluates to False.

Table 2-2: The and Operator's Truth Table

Expression	Evaluates to...
True and True	True
True and False	False
False and True	False
False and False	False

```
>>> True and True
True
>>> True and False
False
```

Truth table shows every possible result of Boolean operator.

or operator: The or operator valuates an expression to True if either of the two Boolean values is True. If bothare False, it evaluates to False.

Table 2-3: The or Operator's Truth Table

Expression	Evaluates to...
True or True	True
True or False	True
False or True	True
False or False	False

```
>>> False or True
True
>>> False or False
False
```

not operator: The not operator operates on only one Boolean value (or expression). The not operator simplyevaluates to the opposite Boolean value.

Much like using double negatives in speech and writing, we can nest not operators ❶, though there's never not no reason to do this in real programs.

Table 2-4: The not Operator's Truth Table

Expression	Evaluates to...
not True	False
not False	True

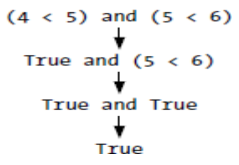
```
  >>> not True
  False
❶ >>> not not not not True
  True
```

2.11 Mixing Boolean and Comparison Operators

Since the comparison operators evaluate to Boolean values, we can use them in expressions with the Boolean operators. Ex:

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

- The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. **Evaluation process for (4 < 5) and (5 < 6)** as shown in Figure below:



```
(4 < 5) and (5 < 6)
        ↓
  True and (5 < 6)
        ↓
    True and True
        ↓
        True
```

- We can also use multiple Boolean operators in an expression, along with the comparison operators.

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

- The Boolean operators have an order of operations just like the math operators do. After any math andcomparison operators evaluate, Python evaluates the **not** operators first, then the **and** operators, and then the **or** operators.

Precedence (Order of evaluation arranged in the higher to lower priority)

1. math (arithmetic operations)
2. comparison
3. not
4. and
5. or

2.12 Elements of Flow Control

- Flow control statements often start with a part called the **condition**, and all are followed by a **block of code called the clause.**

1) Condition
2) Block of code , the clause

Conditions:

- The Boolean expressions are nothing but conditions, which are the same thing as expressions; condition is just a more specific name in the context of flow control statements.
- Conditions always evaluate down to a Boolean value, True or False.
- A flow control statement decides what to do based on whether its condition is True or False, and almostevery flow control statement uses a condition.

Blocks of Code:

➢ Lines of Python code can be grouped together in blocks. There are three rules for blocks.

1. Blocks begin when the indentation increases.
2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block's indentation.

```
name = 'Mary'
password = 'swordfish'
if name == 'Mary':
 ❶ print('Hello, Mary')
    if password == 'swordfish':
     ❷ print('Access granted.')
    else:
     ❸ print('Wrong password.')
```

In Java, C, C++ - We use
```
{
        // block of code
}
```

A block begins with indentation zero, this is called the main block.

➢ The first block of code ❶ starts at the line print('Hello Mary') and contains all the lines after it. Inside thisblock is another block ❷, which has only a single line in it: print('Access Granted.'). The third block ❸ is also one line long: print('Wrong password.').

<u>2.13</u> <u>Program Execution</u>

➢ The program execution (or simply, execution) is a term for the current instruction being executed.

➢ In a program with flow control statements, we will find ourselves jumping around source code based on conditions, and we will probably skip entire clauses.

<u>2.14</u> <u>Flow Control Statements</u>

➢ Flow Control statements are the actual decisions (diamond in flow-chart) that programs will make.

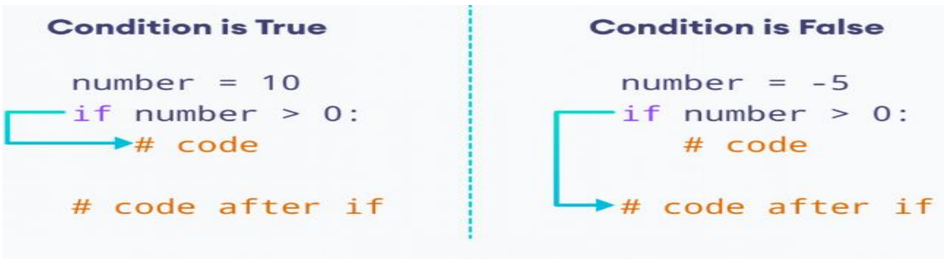
i. if statements
ii. else statements
iii. elif statements
iv. while loop statements
v. break statements
vi. continue statements
vii. for

1. <u>if Statements:</u>

➢ The most common type of flow control statement is the if statement.

➢ An if statement's clause (that is, the block following the if statement) will execute if the statement'scondition is True. The clause is skipped if the condition is False.

- In plain English, an if statement could be read as, "If this condition is true, execute the code in theclause." In Python, an if statement consists of the following:
  5. The if keyword
  6. A condition (that is, an expression that evaluates to True or False)
  7. A colon
  8. Starting on the next line, an indented block of code (called the if clause)



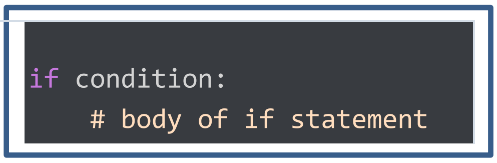
Condition is True

```
number = 10
if number > 0:
    # code

# code after if
```

Condition is False

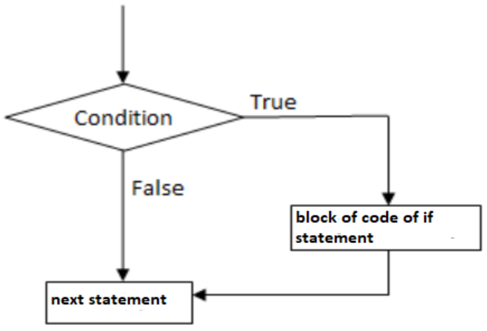
```
number = -5
if number > 0:
    # code

# code after if
```

- Syntax:



```
if condition:
    # body of if statement
```

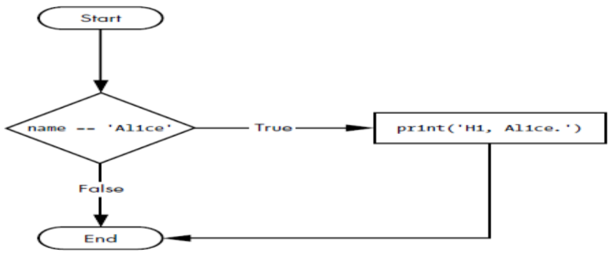
Flowchart:



- Example:

```
if name == 'Alice':
    print('Hi, Alice.')
```

Flowchart:



Input two numbers and display the larger / smaller number using if statements.

# Python Program to find Largest of two Numbers using if statements

a = int(input('Enter the first number: '))

b = int(input('Enter the second number: '))

if a>=b:

```
  print(a,'is greater')
if a<b:
  print(b,'is greater')
```

Output

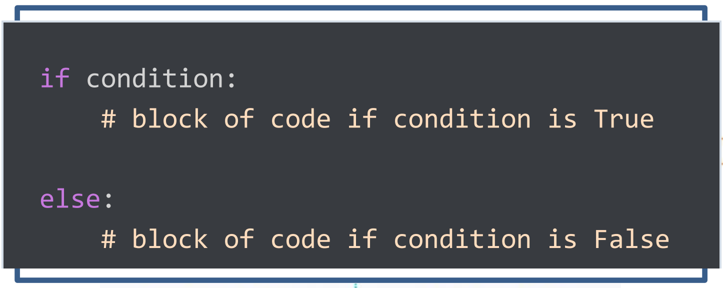
```
Enter the first number: 5
Enter the second number: 6
6 is greater
Enter the first number: 6
Enter the second number: 5
6 is greater
```

2. ***else Statements:***

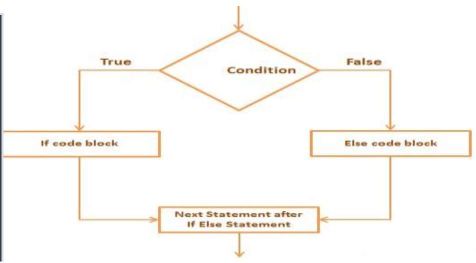
- An if clause can optionally be followed by an else statement.
- The else clause is executed only when the if statement's condition is False.
- In plain English, an else statement could be read as, "If this condition is true, execute this code. Orelse, execute that code."
- An else statement doesn't have a condition, and in code, an else statement always consists of thefollowing:

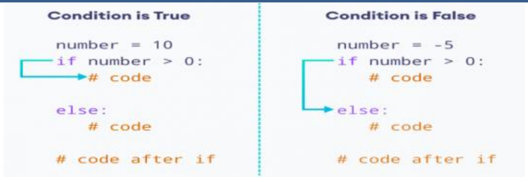
5. The else keyword
6. A colon
7. Starting on the next line, an indented block of code (called the else clause)

Syntax: Flowchart:



```
if condition:
    # block of code if condition is True

else:
    # block of code if condition is False
```





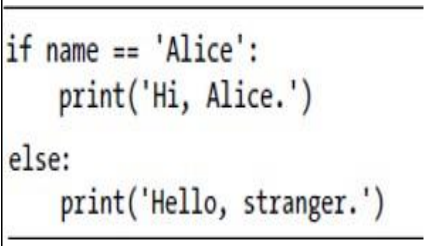
Condition is True

```
number = 10
if number > 0:
    # code

else:
    # code

# code after if
```

Condition is False

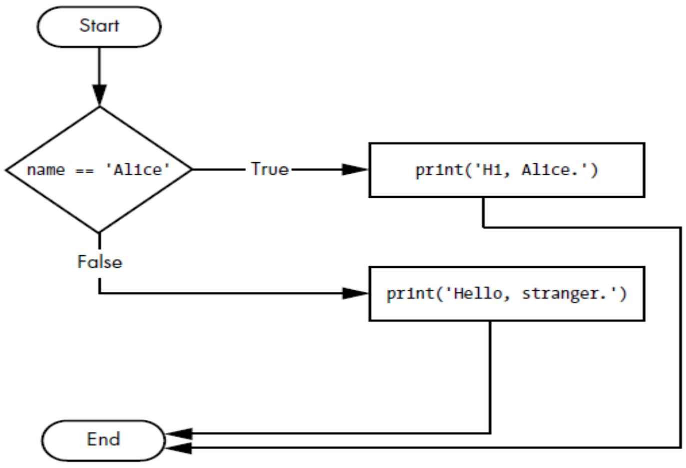
```
number = -5
if number > 0:
    # code

else:
    # code

# code after if
```

Example:



```
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

Flowchart:



Input two numbers and display the larger / smaller number.

```
# Python Program to find Largest of two Numbers using if-else statements
a = int(input('Enter the first number: '))
b = int(input('Enter the second number: '))
if a>b:
  print(a,'is greater')
else:
  print(b,'is greater')
```

Output

```
Enter the first number: 5
Enter the second number: 6
6 is greater
```

WAP to print if the input number is odd or even.

```
# Python Program to check a given number is odd or even
n = int(input('Enter the number: '))
if n%2==0:
  print(n,'is even')
else:
  print(n,'is odd')
```

Output

```
Enter the number: 5
5 is odd
Enter the number: 6
6 is even
```

WAP to say if the person is eligible to vote or not

```
# Python Program to check if a person is eligible to vote or not
name = input('Enter the name of a person: ')
age = int(input('Enter the age of the person: '))
if age>=18:
  print(name,'is eligible to vote')
else:
  print(name,'is not eligible to vote')
```

Output

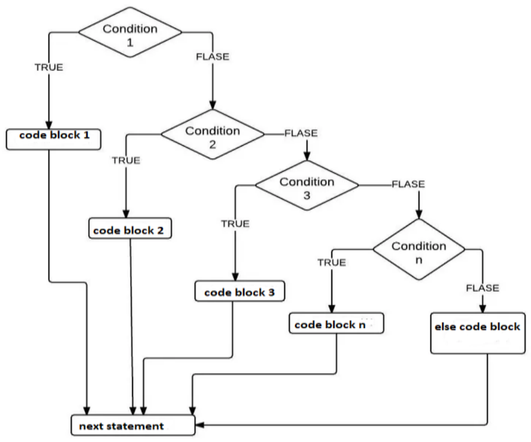
```
Enter the name of a person: sonia
Enter the age of the person: 33
sonia is eligible to vote
Enter the name of a person: xyz
Enter the age of the person: 16
xyz is not eligible to vote
```

3. elif Statements:

- While only one of the if or else clauses will execute, we may have a case where we want oneof many possible clauses to execute.
- The elif statement is an "else if" statement that always follows an if or another elif statement.
- It provides another condition that is checked only if all of the previous conditions were False.
- In code, an elif statement always consists of the following:

5. The elif keyword
6. A condition (that is, an expression that evaluates to True or False)
7. A colon
8. Starting on the next line, an indented block of code (called the elif clause)

```
if condition-1:
    # code block 1
elif condition-2:
    # code block 2
elif condition-3:
    # code block 3
 .
 .
 .
elif condition-n:
    # code block n
else:
    # else code block
```



Flowchart:

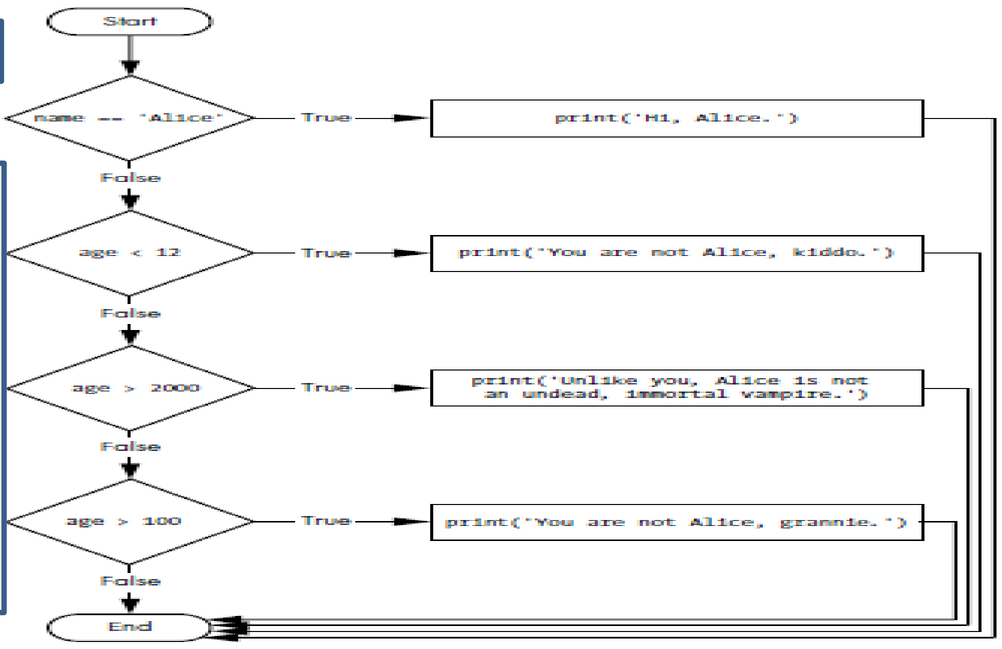
When there is a chain of elif statements, only one or none of the clauses will be executed.

Example

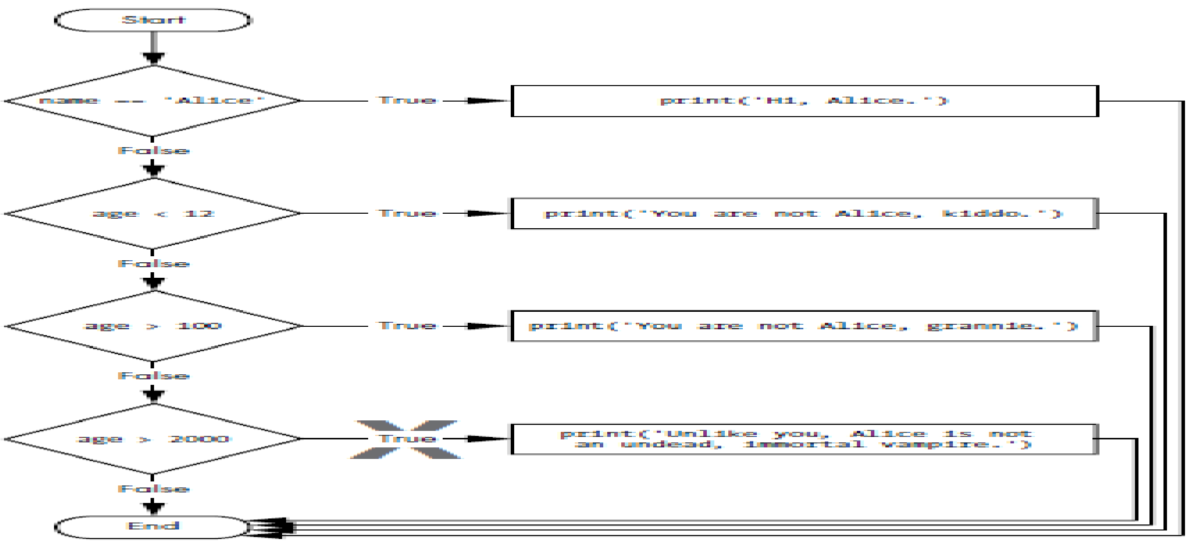
```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

Flow Chart

Assume these values before this code is executed - name = 'Carol' age = 3000


Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
age > 100
True
print('You are not Alice, grannie.')
False
End

- The **order of the elif statements does matter,** however. Let's see by rearranging the previous code.
- We might expect the code to print the string 'Unlike you, Alice is not an undead, immortal vampire.'.
- However, because the age > 100 condition is True (after all, 3000 is greater than 100) ❶, the string 'You are not Alice, grannie.' is printed, and the rest of the elif statements are automatically skipped.
- Remember, at most only one of the clauses will be executed, and for elif statements, the order matters!


Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 100
True
print('You are not Alice, grannie.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
End

- **Optionally, we can have an else statement after the last elif statement.**
- In that case, it is guaranteed that at least one (and only one) of the clauses will be executed.
- If the conditions in every if and elif statement are False, then the else clause is executed.
- In plain English, this type of flow control structure would be, "If the first condition is true, do this. Else,if the second condition is true, do that. Otherwise, do something else."

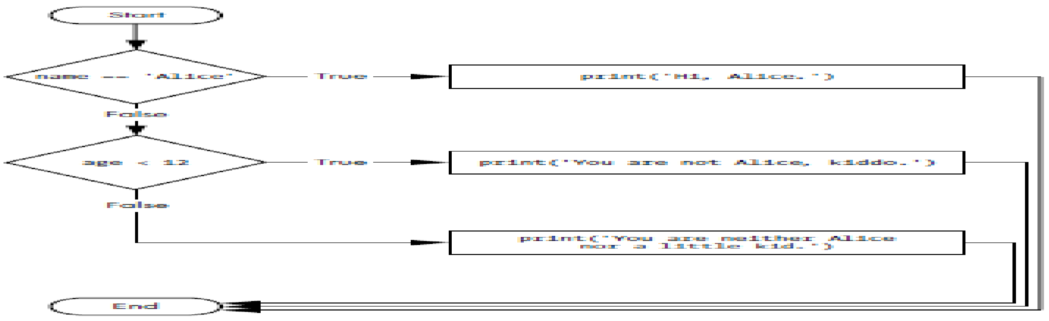
```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

Assume these values before this code is executed -

name = 'Carol'

age = 3000

- Example:


Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
print('You are neither Alice nor a little kid.')
End

First, there is always exactly one if statement. Any elif statements should follow the if statement.

Second, if we want to be sure that at least one clause is executed, close the structure with an else statement.

Input three numbers and display the larger / smaller number.

```
# Python Program to find Largest of three Numbers using elif statements
n1 = int(input('Enter the first number: '))
n2 = int(input('Enter the second number: '))
n3 = int(input('Enter the third number: '))
if n1<n2 and n1<n3:
  print(n1,'is smaller')
elif n2<n3:
  print(n2,'is smaller')
else:
  print(n3,'is smaller')
```

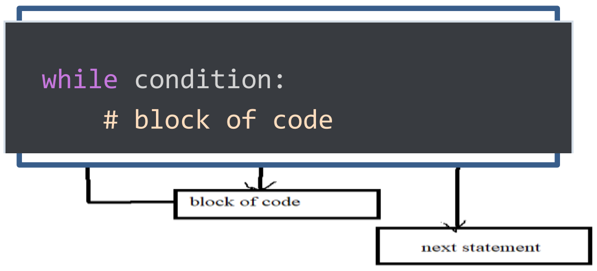
Output

```
Enter the first number: 2
Enter the second number: 7
Enter the third number: 1
1 is smaller
```

4. ***while loop Statements:***

- We can make a block of code execute over and over again with a while statement.
- The code in a while clause will be executed as long as the while statement’s condition is True.
- In code, a while statement always consists of the following:
  5. The while keyword
  6. A condition (that is, an expression that evaluates to True or False.
  7. A colon
  8. Starting on the next line, an indented block of code (called the while clause)
- We can see that a while statement looks similar to an if statement. The difference is in how they behave. At the end of an if clause, the program execution continues after the if statement.
- The while clause is often called the while loop or just the loop.

- In the while loop, the condition is always checked at the start of each iteration (that is, each time the loopis executed).
- If the condition is True, then the clause is executed, and afterward, the condition is checked again.
- The first time the condition is found to be False, the while clause is skipped.

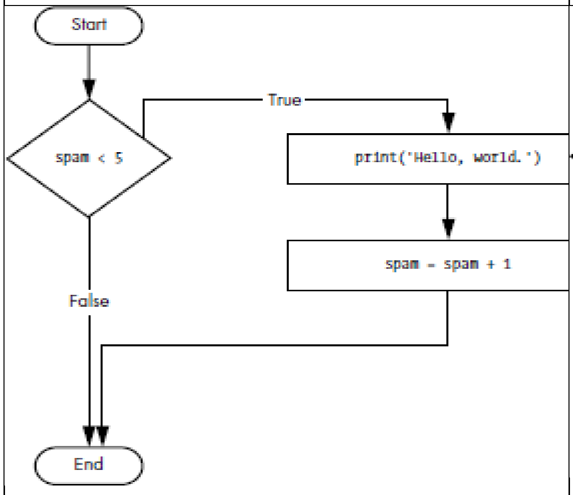
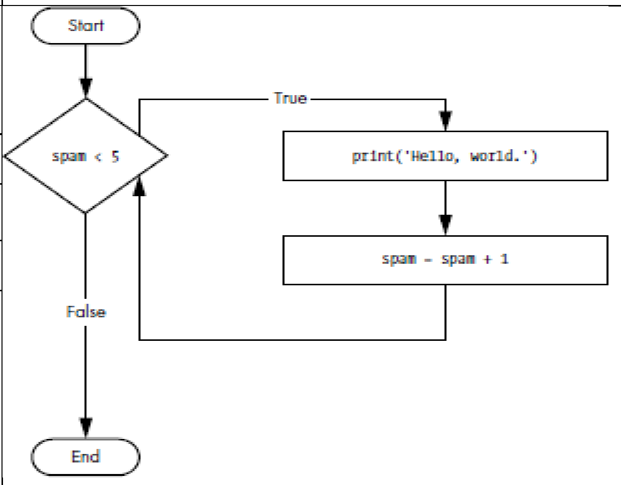


```
while condition:
    # block of code
```

Flowchart:

Using if statement	Using while statement

`spam = 0`
`if spam < 5:`
`    print('Hello, world.')`
`    spam = spam + 1`
	`spam = 0`
	`while spam < 5:`
	`    print('Hello, world.')`
	`    spam = spam + 1`

The statements within the if statement is executed only once if the condition is True.	The statements within the while statement is executed as long as the while statement's condition is True.
if condition: # block of code of if	while condition: # block of code of while
 Start spam < 5 True print('Hello, world.') spam = spam + 1 False End	 Start spam < 5 True print('Hello, world.') spam = spam + 1 False End
Output Hello, world	Output Hello, world Hello, world

	Hello, world Hello, world Hello, world
The output is simply Hello, world printed only once	The output is simply Hello, world repeated five times.

Any Loop should contain:

- Loop variable initialization
- Condition that changes
- Increment/Decrement in the loop variable (changes in the loop variable)
- Example:

WAP to print 1 to 10 and trace it.

```
# Python Program to print 1 to 10
i=1
while i<=10:
    print(i)
    i=i+1
```

Output

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

Tracing

i=1
1<=10 (T)
i=1+1=2
2<=10 (T)
i=2+1=3
3<=10 (T)
i=3+1=4
4<=10 (T)
i=4+1=5
5<=10 (T)
i=5+1=6
6<=10 (T)
i=6+1=7
7<=10 (T)
i=7+1=8
8<=10 (T)
i=8+1=9
9<=10 (T)
i=9+1=10
10<=10 (T)
i=10+1=11
11<=10 (F)
While loop stops as the condition is false now.

WAP to print the even numbers between 1 to 10 and trace it.

```
# Program to print even numbers between 1 to 10
i=1
while i<=10:
        if i%2==0:
                print(i)
        i=i+1
```

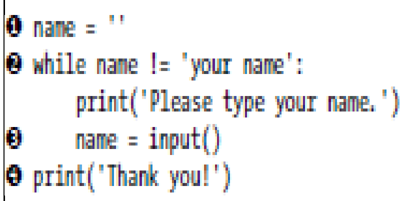
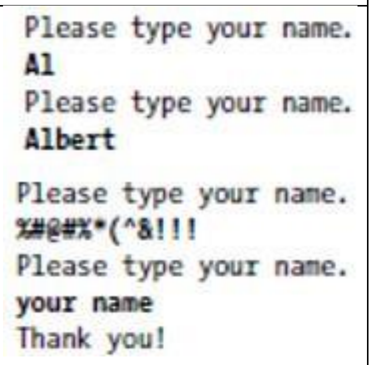
```
Tracing
i=1
1<=10 (T)
          1%2==0 (F)
          i=1+1=2
2<=10 (T)
          2%2==0 (T)
                    print i
          i=2+1=3
3<=10 (T)
          3%2==0 (F)
          i=3+1=4
4<=10 (T)
          4%2==0 (T)
                    print i
          i=4+1=5
5<=10 (T)
          5%2==0 (F)
          i=5+1=6
6<=10 (T)
          6%2==0 (T)
                    print i
          i=6+1=7
7<=10 (T)

          7%2==0 (F)
          i=7+1=8
8<=10 (T)
          8%2==0 (T)
                    print i
          i=8+1=9
9<=10 (T)
          9%2==0 (F)
          i=9+1=10
10<=10 (T)
          10%2==0 (T)
                    print i
          i=10+1=11
11<=10 (F) while loop stops
now as it is false.
```

Output

```
    2
    4
    6
    8
    10
```

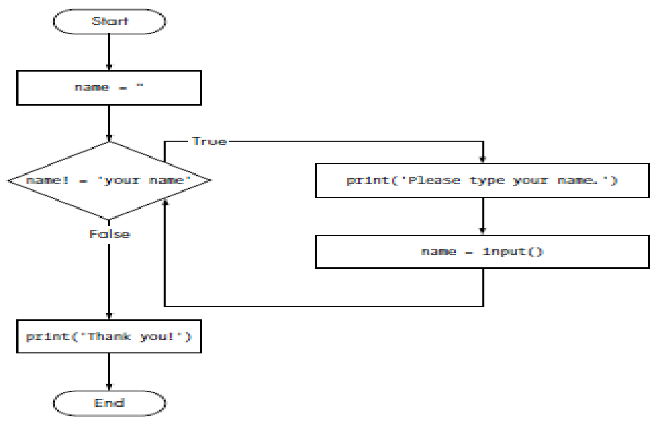
An annoying while loop:

➢ Here's a small example program that will keep asking to type, literally, your name.

Example Program	Output
	

```
❶ name = ''
❷ while name != 'your name':
      print('Please type your name.')
❸     name = input()
❹ print('Thank you!')
```

```
Please type your name.
Al
Please type your name.
Albert

Please type your name.
%#@#%*(^&!!!
Please type your name.
your name
Thank you!
```



Flow Chart

- First, the program sets the name variable ❶ to an empty string.
- This is so that the name != 'your name' condition will evaluate to True and the program execution will enter the while loop's clause ❷.
- The code inside this clause asks the user to type their name, which is assigned to the name variable ❸.
- Since this is the last line of the block, the execution moves back to the start of the while loop and reevaluates the condition.
- If the value in name is not equal to the string 'your name', then the condition is True, and the execution enters the while clause again.
- But once the user types your name, the condition of the while loop will be 'your name' != 'your name',which evaluates to False.

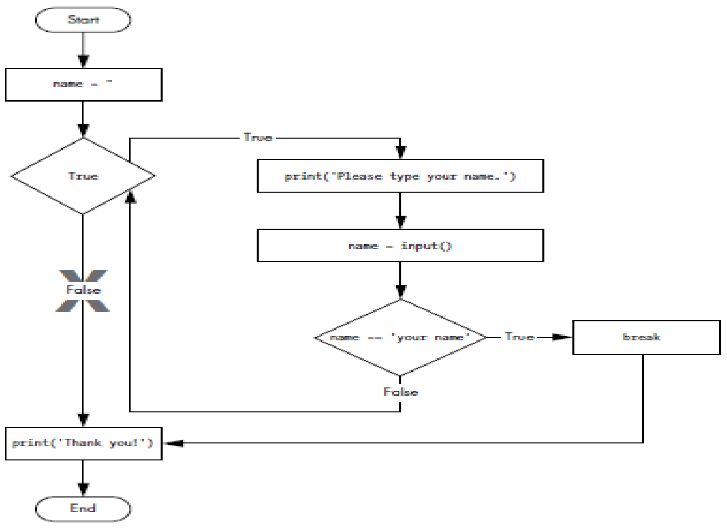
If you never enter **your name**, then the while loop's condition will never be false, and the program will just keep asking forever.

9. ***break Statements:***

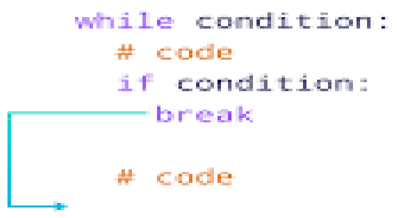
- There is a shortcut to getting the program execution to break out of a while loop's clause early.
- If the execution reaches a break statement, it immediately exits the while loop's clause.
- In code, a break statement simply contains the break keyword.
- Example:

```
❶ while True:
      print('Please type your name.')
❷     name = input()
❸     if name == 'your name':
❹         break
❺ print('Thank you!')
```

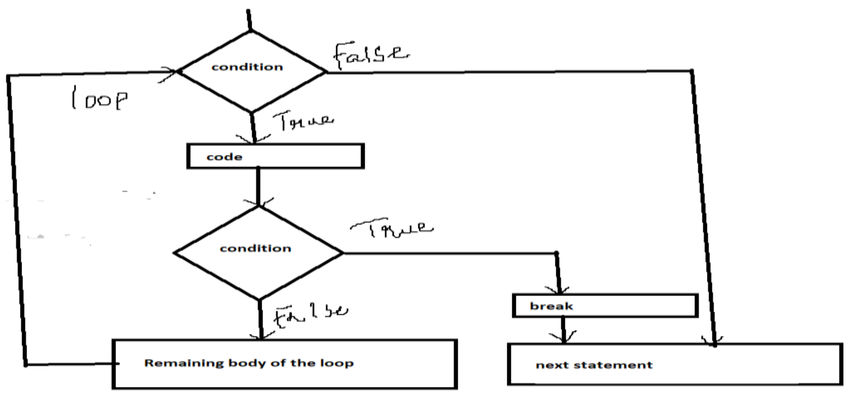
- The first line ❶ creates an infinite loop; it is a while loop whose condition is always True. (Theexpression True, after all, always evaluates down to the value True.)
- The program execution will always enter the loop and will exit it only when a break statement isexecuted. (An infinite loop that never exits is a common programming bug.)
- Just like before, this program asks the user to type your name ❷.
- Now, however, while the execution is still inside the while loop, an if statement gets executed ❸ to check whether name is equal to your name.
- If this condition is True, the break statement is run ❹, and the execution moves out of the loopto print('Thank you!') ❺.
- Otherwise, the if statement's clause with the break statement is skipped, which puts the execution at the end of the while loop.
- At this point, the program execution jumps back to the start of the while statement ❶ to recheck the condition. Since this condition is merely the True Boolean value, the execution enters the loop to ask the user to type your name again.
- Flowchart:



<u>Syntax -</u> break



```
while condition:
   # code
   if condition:
      break

   # code
```



WAP to print only the first 7 numbers between 1 to 20 using break statement.

```
# Program to print only the first 7 numbers between 1 to 20 using break statement.
i = 1
while i<=20:
    print(i)
    if i>=7:
        break
    i = i + 1
print('End of program')
```

Output

```
1
2
3
4
5
6
7
End of program
```

WAP to find first 5 multiples of 6 using break statement.

```
# Program to find first 5 multiples of 6
i = 1
while  i<=10:
    print('6 * ',(i), '=',6 * i)
    if i >= 5:
        break
    i = i + 1
```

Output

```
6 *  1 = 6
```

6 * 2 = 12

6 * 3 = 18

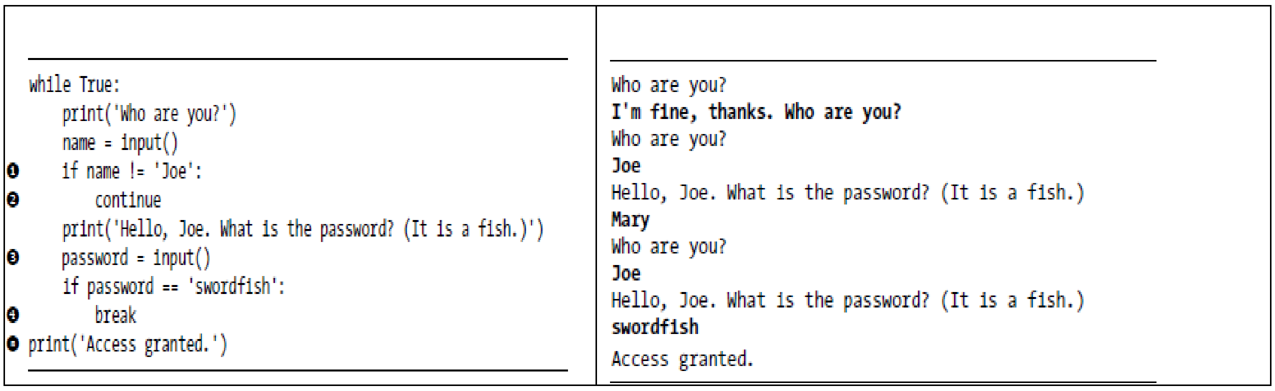
6 * 4 = 24

6 * 5 = 30

10. <u>Continue statements:</u>

- ➢ Like break statements, continue statements are used inside loops.
- ➢ When the program execution reaches a continue statement, the program execution immediately jumpsback to the start of the loop and reevaluates the loop's condition.

- ➢ Example and Output:

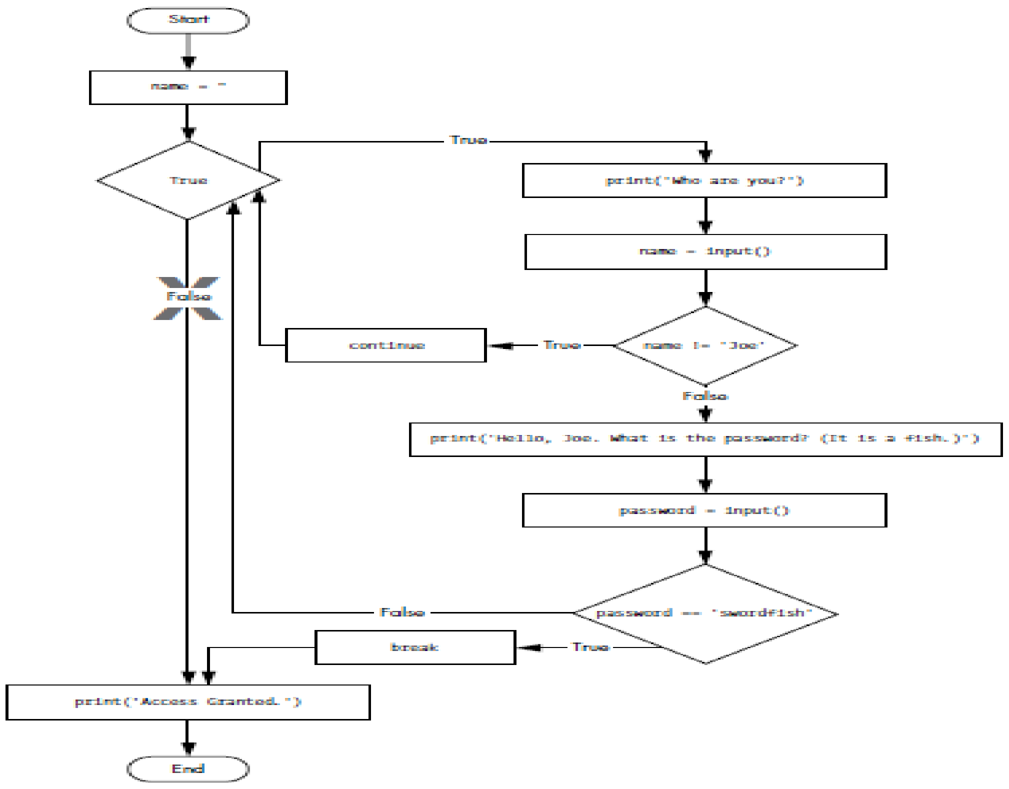


```
  while True:
      print('Who are you?')
      name = input()
❶     if name != 'Joe':
❷         continue
      print('Hello, Joe. What is the password? (It is a fish.)')
❸     password = input()
      if password == 'swordfish':
❹         break
❺ print('Access granted.')
```

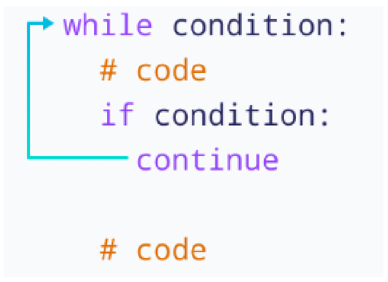
```
Who are you?
I'm fine, thanks. Who are you?
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.
```

- ➢ If the user enters any name besides Joe ❶, the continue statement ❷ causes the program execution tojump back to the start of the loop.
- ➢ When it reevaluates the condition, the execution will always enter the loop, since the condition is simplythe value True. Once they make it past that if statement, the user is asked for a password ❸.
- ➢ If the password entered is swordfish, then the break statement ❹ is run, and the execution jumps out ofthe while loop to print Access granted ❺.
- ➢ Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start ofthe loop.

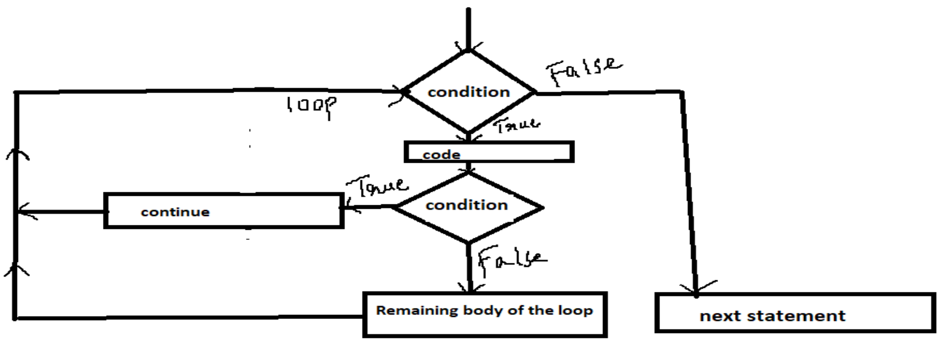
➢ Flowchart:



➢ <u>Syntax</u> - continue



```
while condition:
    # code
    if condition:
        continue

    # code
```



WAP to print 1 to 10 except 4 and 6 using continue statement.

# WAP to print 1 to 10 except 4 and 6 using continue statement.

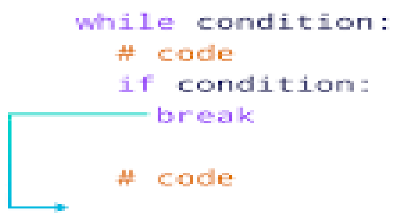
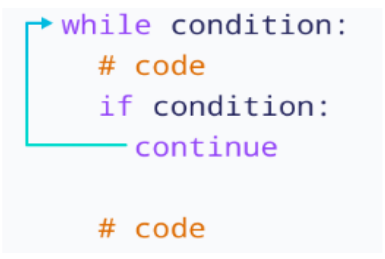
```
i = 1
while i<=10:
    if i==4 or i==6:
        i=i+1
        continue
    print(i)
    i=i+1
print('end of program')
```

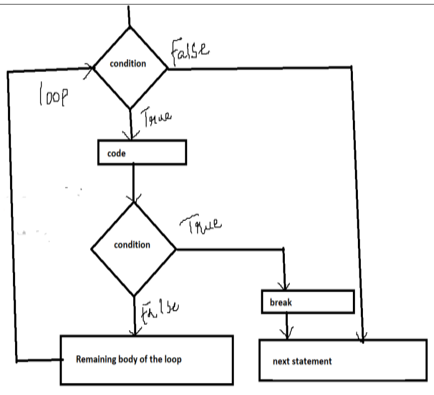
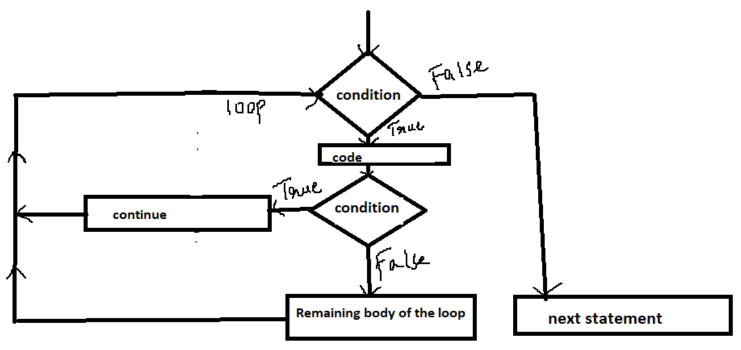
Output

```
1
2
3
5
7
8
9
10
end of program
```

Difference between break and continue.

break	continue
Keyword is break	Keyword is continue
On encountering break statement, control moves out of the loop.	On encountering continue statement, control moves to the next iteration of the loop by skipping the remaining while clause below continue.

Syntax – break  `while condition:`
`    # code`
`    if condition:`
`        break`
`    # code`
	Syntax - continue  `while condition:`
	`    # code`
	`    if condition:`
	`        continue`
	`    # code`

 condition False loop True code condition True False break Remaining body of the loop next statement	 condition False loop True code True continue condition False Remaining body of the loop next statement

```
i = 1
while i<=20:
    print(i)
    if i>=7:
        break
    i = i + 1
print('End of program')
```

Output

```
1
2
3
4
5
6
7
End of program
```

```
i = 1
while i<=10:
    if i==4 or i==6:
        i=i+1
        continue
    print(i)
    i=i+1
print('end of program')
```

Output

```
1
2
3
5
7
8
9
10
end of program
```

TRUTHY AND FALSEY VALUES

- Conditions will consider some values in other data types equivalent to True and False.
- When used in conditions, 0, 0.0, and ' ' (the empty string) are considered False, while all other values are considered True.

Example:

```
i=int(input('Enter a number'))
if i:
    print('iam True')
else:
    print('iam False')
```

Output

```
Enter a number4
iam True
Enter a number0
iam False
```

NOTE: If we ever run a program that has a bug causing it to get stuck in an infinite loop, press CTRL-C.This will send a KeyboardInterrupt error to our program and cause it to stop immediately.

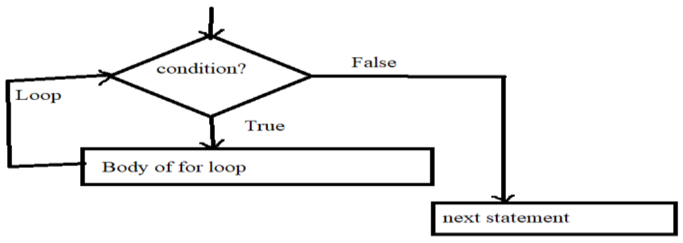
```
while True:
        print('Hello, world')
```

11. for loops and the range() function:

- The while loop keeps looping while its condition is true but if we want to execute a block of code only a certain **number of times** then we can do this with a for loop statement and the range() function.
- In code, a for statement looks something like for i in range(5): and always includes the

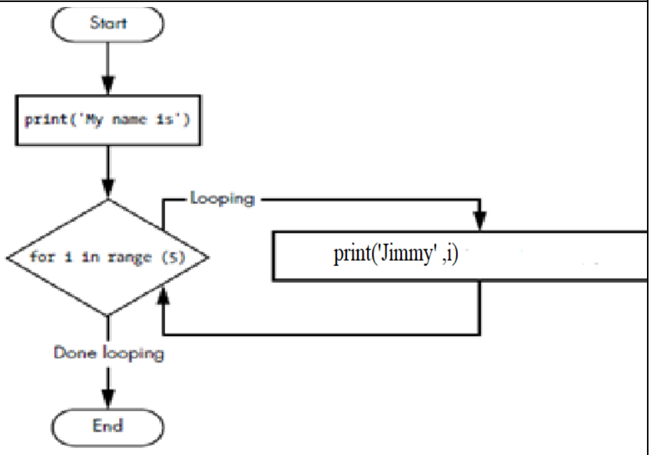
following:

1. The for keyword
2. A variable name
3. The in keyword
4. A call to the range() method with up to three integers passed to it
5. A colon
6. Starting on the next line, an indented block of code (called the for clause)



```
for i in range(n):
    #code block
```

Example	Output	Flow Chart

print('My name is')
for i in range(5):
    print('Jimmy',i)
	My name is
	Jimmy 0
	Jimmy 1
	Jimmy 2
	Jimmy 3
	Jimmy 4
		 Start
		print('My name is')
		Looping
		for i in range (5)
		print('Jimmy' ,i)
		Done looping
		End

- ➢ The code in the for loop’s clause is run five times.
- ➢ The first time it is run, the variable i is set to 0.
- ➢ The print() call in the clause will print Jimmy Five Times.
- ➢ After Python finishes an iteration through all the code inside the for loop’s clause, the execution goesback to the top of the loop, and the for statement increments i by one.
- ➢ This is why range(5) results in five iterations through the clause, with i being set to 0, then 1, then 2,then 3, and then 4.
- ➢ The variable i will go up to, but will not include, the integer passed to range().

An equivalent while loop:

```
print('My name is')
i=0
while i<5:
    print('Jimmy' ,i)
    i=i+1
```

- Example 2:

```
❶ total = 0
❷ for num in range(101):
❸     total = total + num
❹ print(total)
```

- The result should be 5,050. When the program first starts, the total variable is set to 0 ❶.
- The for loop ❷ then executes total = total + num ❸ 100 times.
- By the time the loop has finished all of its 100 iterations, every integer from 0 to 100 will have been addedto total. At this point, total is printed to the screen ❹.

The Starting, Stopping, and Stepping Arguments to range()

- Some functions can be called with multiple arguments separated by a comma, and range() is one ofthem.
- This lets us change the integer passed to range() to follow any sequence of integers, including startingat a number other than zero.
- The first argument will be where the for loop's variable starts, and the second argument will be up to,but not including, the number to stop at.

```
for i in range(12, 16):
    print(i)
```

```
12
13
14
15
```

- The range() function can also be called with three arguments. The first two arguments will be the startand stop values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0, 10, 2):
    print(i)
```

- So calling range(0, 10, 2) will count from zero to eight by intervals of two.

```
0
2
4
6
8
```

- The range() function is flexible in the sequence of numbers it produces for for loops. We can even usea negative number for the step argument to make the for loop count down instead of up.

```
for i in range(5, -1, -1):
    print(i)
```

```
5
4
3
2
1
0
```

- Running a for loop to print i with range(5, -1, -1) should print from five down to zero.

2.15 Importing Modules

- All Python programs can call a basic set of functions called built-in functions, includingthe print(), input(), and len() functions.
- Python also comes with a set of modules called the standard library.
- Each module is a Python program that contains a related group of functions that can be embedded inour programs.
- For example, the math module has mathematics-related functions, the random module has randomnumber–related functions, and so on.
- Before we can use the functions in a module, we must import the module with an import statement. Incode, an import statement consists of the following:
  1. The import keyword
  2. The name of the module

3. Optionally, more module names, as long as they are separated by commas

- Once we import a module, we can use all the functions of that module.
- Example with output:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

```
4
1
8
4
1
```

- The random.randint() function call evaluates to a random integer value between the two integers that we pass it.
- Since randint() is in the random module, we must first type **random.** in front of the function name to tell Python to look for this function inside the random module.
- Here's an example of an import statement that imports four different modules:

```
import random, sys, os, math
```

- Now we can use any of the functions in these four modules.

```
import random,math
n=random.randint(1, 10)
print('The random number is',n)
print('The square of the number is',math.pow(n,2))
```

```
Output
The random number is 10
The square of the number is 100.0
```

from import Statements

- An alternative form of the import statement is composed of the from keyword, followed by the modulename, the import keyword, and a star; for example, from random import *.
- With this form of import statement, calls to functions in random will not need the random prefix.
- However, using the full name makes for more readable code, so it is better to use the normal form ofthe import statement.

```
from math import pow,sqrt
from random import *
n=randint(1, 10)
print('The random number is',n)
print('square of the number is',pow(n,2))
print('square root of the number is',sqrt(n))
```

```
Output
The random number is 5
square of the number is 25.0
square root of the number is 2.23606797749979
```

2.16 Ending a Program Early with sys.exit()

- How to terminate the program - It always happens if the program execution reaches the bottom of the instructions.
- However, we can cause the program to terminate, or exit, by calling the sys.exit() function. Since thisfunction is in the sys module, we have to import sys before our program can use it.

```
import sys
while True:
        print(‘Type exit to exit’)
        r=input()
        if r==’exit’:
                sys.exit()
        print(‘you typed ‘,r)
```

Output

```
        Type exit to exit.
        hi
        You typed hi.
        Type exit to exit.
        hello
        You typed hello.
        Type exit to exit.
```

- This program has an infinite loop with no break statement inside. The only way this program will end isif the user enters exit, causing sys.exit() to be called.
- When response is equal to exit, the program ends.
- Since the response variable is set by the input() function, the user must enter exit in order to stop theprogram.

CHAPTER 3: FUNCTIONS

1. def Statements with Parameters
2. Return Values and return Statements
3. The None Value
4. Keyword Arguments and print()
5. Local and Global Scope
6. The global Statement
7. Exception Handling
8. A Short Program: Guess the Number

Introduction

- Python provides several built-in functions like print(), input(), and len() functions, but we can also write your own functions.
- Two types of functions – built-in functions, user-defined functions.
- A function is like a mini-program within a program.

Example:

```
❶def hello():                                        # called function
        ❷print('hi')
          print('hey')
          print('hi all')
❸hello()                                   # call
  hello()
  hello()
```

- The first line is a def statement ❶, which defines a function named hello().
- The code in the block that follows the def statement ❷ is the body of the function. This code is executedwhen the function is called, not when the function is first defined.
- The hello() lines after the function ❸ are function calls.
- In code, a function call is just the function’s name followed by parentheses, possibly with some numberof arguments in between the parentheses.
- When the program execution reaches these calls, it will jump to the top line in the function and beginexecuting the code there.

- When it reaches the end of the function, the execution returns to the line that called the function andcontinues moving through the code as before.
- Since this program calls hello() three times, the code in the hello() function is executed three times. Whenwe run this program, the output looks like this:

```
hi
hey
hi all
hi
hey
hi all
hi
hey
hi all
```

Uses of functions

- A major purpose of functions is to **group code that gets executed multiple times**. Without a function defined, we would have to copy and paste this code each time, and the program would look like this:

```
print('hi')
print('hey')
print('hi all')
print('hi')
print('hey')
print('hi all')
print('hi')
print('hey')
print('hi all')
```

- We must generally deduplicate the code, which means getting rid of duplicated or copy-and-pasted code. Deduplication makes our **programs shorter, easier to read, and easier to update**.

3.1 def Statements with Parameters

- When we call the print() or len() function, we pass in values, called arguments, by typing them between the parentheses.
- We can also define our own functions that accept arguments.

```
❶ def hello(name):
❷     print('Hello ' + name)

❸ hello('Alice')
  hello('Bob')
```

```
Hello Alice
Hello Bob
```

- Example with output:
- The definition of the hello() function in this program has a parameter called name ❶.
- A parameter is a variable that an argument is stored in when a function is called.
- The first time the hello() function is called, it’s with the argument 'Alice' ❸.
- The program execution enters the function, and the variable name is automatically set to 'Alice', which iswhat gets printed by the print() statement ❷.
- One special thing to note about parameters is that the value stored in a parameter is forgotten when thefunction returns.

Define, Call, Pass, Argument, Parameter

```
❶ def hello(name):
      print('Hello, ' + name)
❷ hello('Alice')
```

To **define** a function is to create it, just like an assignment statement like spam = 42 creates the spam variable. The def statement defines the hello() function ❶.

The hello('Alice') line ❷ **calls** the now-created function, sending the execution to the top of the function’s code. This function call is also known as **passing** the string value 'Alice' to the function. A value being passed to a function in a function call is an **argument**. The argument 'Alice' is assigned to a local variable named name. Variables that have arguments assigned to them are **parameters**.

<u>3.2 Return Values and Return Statements</u>

- The value that a function call evaluates to is called the return value of the function.
- Ex: len(‘Hello’) → return value is 5
- When creating a function using the def statement, we can specify what the return value should be witha return statement.
- A return statement consists of the following:
  1. The return keyword
  2. The value or expression that the function should return.

```
def add(a,b):
        return a+b
sum=add(5,4)
print('sum is',sum)
```

<u>Output</u>

```
sum is 9
```

- When an expression is used with a return statement, the return value is what this expression evaluates to.
- For example, here 5+4 is evaluated and 9 is returned.

3.3 The None Value

- In Python there is a value called None, which represents the absence of a value. None must be typed with capital N.
- None is the only value of the NoneType data type.
- This value-without-a-value can be helpful when we need to store something that won't be confused for a real value in a variable.
- One place where None is used is as the return value of print().
- The print() function displays text on the screen, but it doesn't need to return anything in the same way len() or input() does. But **since all function calls** need to evaluate to a return value, print() returns None.

```
s=print('hello')
print(s)
```

OUTPUT

```
hello
None
```

- Behind the scenes, Python adds return None to the end of any function definition with no return statement.

3.4 Keyword Arguments and print()

- Most arguments are identified by their position in the function call.
- For example, random.randint(1, 10) is different from random.randint(10, 1).
- The function call random.randint(1, 10) will return a random integer between 1 and 10, because the first argument is the low end of the range and the second argument is the high end while random.randint(10, 1) causes an error.
- Rather than through their position, **keyword arguments are identified by the keyword put before them** in the function call.
- Keyword arguments are often used for **optional parameters**.

- For example, the print() function has the optional parameters **end and sep** to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

```
print('Hello')
print('World')
```

```
Hello
World
```

- The two strings appear on separate lines because the print() function automatically adds a newline character to the end of the string it is passed.
- However, we can set the **end keyword argument** to change this to a different string.
- For example, if the program were this:

```
print('Hello', end=' ')
print('world')
```

```
Hello world
```

- The output is printed on a single line because there is no longer a new-line printed after 'Hello'. Instead, the blank string is printed. This is useful if we need to disable the newline that gets added to the end of every print() function call.
- Similarly, when we pass multiple string values to print(), the function will automatically separate them with a single space.

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

- But we could replace the default separating string by passing the **sep keyword argument**.

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

3.5 Local and Global Scope

- Parameters and variables that are assigned in a called function are said to exist in that function's ***local scope.***
- Variables that are assigned outside all functions are said to exist in the ***global scope***.
- A variable that exists in a local scope is called a ***local variable***, while a variable that exists in the global scope is called a ***global variable.***
- A variable must be one or the other; it cannot be both local and global.
- When a scope is destroyed, all the values stored in the scope's variables are forgotten.
- There is only one global scope, and it is created when our program begins. When our

program terminates, the global scope is destroyed, and all its variables are forgotten.

- A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten.
- Scopes matter for several reasons:

1. Code in the global scope cannot use any local variables.
2. However, a local scope can access global variables.
3. Code in a function's local scope cannot use variables in any other local scope.
4. We can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

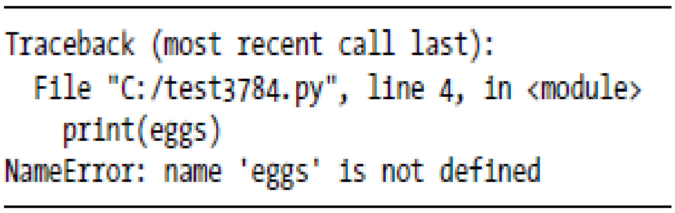
Local Variables Cannot Be Used in the Global Scope

- Consider this program, which will cause an error when we run it:

Program

```
def spam():
    eggs = 99
spam()
print(eggs)
```

Output → Error



```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

Output

NameError:

- The error happens because the eggs variable exists only in the local scope created when spam() is called.
- Once the program execution returns from spam, that local scope is destroyed, and there is no longer avariable named eggs.

Local Scopes Cannot Use Variables in Other Local Scopes

- A new local scope is created whenever a function is called, including when a function is called from another function.
- Consider this program:

```
def spam():
   ❶eggs = 99
   ❷bacon()
   ❸print(eggs)
def bacon():
   ❹ham = 101
    eggs = 0
spam()
```

OUTPUT

```
99
```

- When the program starts, the spam() function is called, and a local scope is created.
- The local variable eggs ❶ is set to 99.
- Then the bacon() function is called ❷, and a second local scope is created.
- Multiple local scopes can exist at the same time.
- In this new local scope, the local variable ham is set to 101, and a local variable eggs—which is differentfrom the one in spam()'s local scope—is also created ❹ and set to 0.
- When bacon() returns, the local scope for that call is destroyed. The program execution continues inthe spam() function to print the value of eggs ❸, and since the local scope for the call to spam() still existshere, the eggs variable is set to 99.

Global Variables can be read from a Local Scope

- Consider the following program:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

OUTPUT

```
42
42
```

- Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs is used in spam(), Python considers it a reference to the global variable eggs. This is why 42 is printed when the previous program is run.

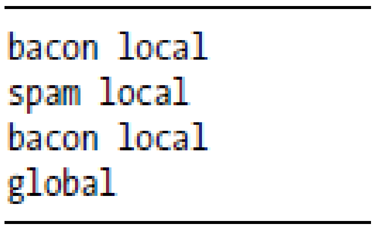
Local and Global Variables with the Same Name

- To simplify, avoid using local variables that have the same name as a global variable or another localvariable.

➢ But technically, it’s perfectly legal to do so.

Example Output

```
def spam():
❶       eggs = 'spam local'
        print(eggs) # prints 'spam local'
def bacon():
❷       eggs = 'bacon local'
        print(eggs) # prints 'bacon local'
        spam()
        print(eggs) # prints 'bacon local'
❸eggs = 'global'
  bacon()
  print(eggs) # prints 'global'
```



```
bacon local
spam local
bacon local
global
```

➢ There are actually three different variables in this program, but confusingly they are all named eggs. Thevariables are as follows:

❶ A variable named eggs that exists in a local scope when spam() is called.

❷ A variable named eggs that exists in a local scope when bacon() is called.

❸ A variable named eggs that exists in the global scope.

➢ Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time.

➢ This is why we should avoid using the same variable name in different scopes.

3.6 <u>The Global Statement</u>

➢ If we need to modify a global variable from within a function, use the global statement.

➢ If we have a line such as global eggs at the top of a function, it tells Python, “In this function, eggs refers to the global variable, so don’t create a local variable with this name.”

➢ Because eggs is declared global at the top of spam() ❶, when eggs is set to 'spam' ❷, this assignment is done to the globally scoped eggs. No local eggs variable is created.

- For example:

Program

```
def spam():
 ❶     global eggs
 ❷     eggs = 'spam'
eggs = 'global'
spam()
print(eggs)
```

Output

```
spam
```

- There are four rules to tell whether a variable is in a local scope or global scope:
  1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
  2. If there is a global statement for that variable in a function, it is a global variable.
  3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
  4. But if the variable is not used in an assignment statement, it is a global variable.

In the spam() function, eggs is the global eggs variable, because there's a global statement for eggs at the beginning of the function ❶.

In bacon(), eggs is a local variable, because there's an assignment statement for it in that function ❷.

In ham() ❸, eggs is the global variable, because there is no assignment statement or global statement for it in that function.

In a function, a variable will either always be global or always be local. There's no way that the code in a function can use a local variable named eggs and then later in that same function use the global eggs variable.

- Example:

Program

```
def spam():
❶     global eggs
      eggs = 'spam' # this is the global
def bacon():
❷     eggs = 'bacon' # this is a local
      print(eggs)
def ham():
❸     print(eggs) # this is the global
eggs = 42 # this is the global
spam()
print(eggs)
bacon()
ham()
```

Output

```
spam
bacon
spam
```

Note

- **If we ever want to modify the value stored in a global variable from in a function, we must use a global statement on that variable.**

  If we try to use a local variable in a function before we assign a value to it, as in the following program, Python will give you an error.

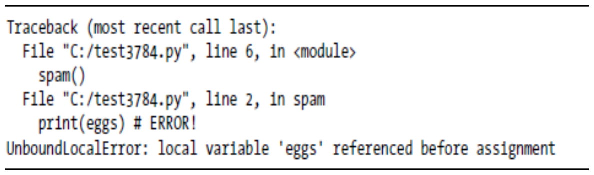
Program

```
def spam():
    print(eggs) # ERROR!
   eggs = 'spam local'
eggs = 'global'
spam()
```

Output

OUTPUT

UnboundLocalError



```
Traceback (most recent call last):
  File "C:/test3784.py", line 6, in <module>
    spam()
  File "C:/test3784.py", line 2, in spam
    print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

- This error happens because Python sees that there is an assignment statement for eggs in the spam() function and therefore considers eggs to be local.

- But because print(eggs) is executed before eggs is assigned anything, the local variable eggs doesn't exist. Python will not fall back to using the global eggs variable.

3.7 Exception Handling

- If we don't want to crash the program due to errors instead we want the program to **detect errors, handle them, and then continue to run** –then we can use Exception Handling.
- For example,

```
def spam(d):
    return 42 / d
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output

```
21.0
3.5
Traceback (most recent call last):
  File "C:\Users\sonim\AppData\Local\Programs\Python\Python37\kg.py", line 5, in
<module>
    print(spam(0))
  File "C:\Users\sonim\AppData\Local\Programs\Python\Python37\kg.py", line 2, in spam
    return 42 / d
ZeroDivisionError: division by zero
```

OUTPUT

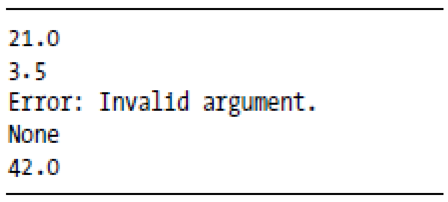
```
21.0
3.5
ZeroDivisionError:
```

- A ZeroDivisionError happens whenever we try to divide a number by zero.
- From the line number givenin the error message, we know that the return statement in spam() is causing an error.
- **Errors can be handled with try and except statements.**
- The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens.

Program

```
def spam(d):
    try:
        return 42 / d
    except ZeroDivisionError:
        print('Error: Invalid argument.')
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Output



```
21.0
3.5
Error: Invalid argument.
None
42.0
```

- We can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.
- Note that any errors that occur in function calls in a try block will also be caught. Consider the followingprogram, which instead has the spam() calls in the try block:

Program

```
def spam(d):
    return 42 / d
try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

Output

```
21.0
3.5
Error: Invalid argument.
```

- The reason print(spam(1)) is never executed is because once the execution jumps to the code inthe except clause, it does not return to the try clause. Instead, it just continues moving down as normal.

3.8 A Short program: Guess the Number

This is a simple “guess the number” game. When we run this program, the output will look something like this:

```
I am thinking of a number between 1 and 20
Take a guess 12
Your guess is too high.
Take a guess 10
Your guess is too high.
Take a guess 8
Your guess is too high.
Take a guess 4
Your guess is too high.
Take a guess 2
Good job! You guessed my number in  5 guesses!
```

- Code for the above program is:

```
# This is a guess the number game.
import random
secret=random.randint(1, 20)
print('I am thinking of a number between 1 and 20')
for i in range(1, 7):
    guess = int(input('Take a guess '))
    if guess < secret:
        print('Your guess is too low.')
    elif guess > secret:
        print('Your guess is too high.')
    else:
        break
if guess == secret:
    print('Good job! You guessed my number in '+ str(i) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secret))
```

- Let’s look at this code line by line, starting at the top.

```
# This is a guess the number game.
import random
secret=random.randint(1, 20)
```

- First, a comment at the top of the code explains what the program does.
- Then, the program imports the random module so that it can use the random.randint() function to generate a number for the user to guess.
- The return value, a random integer between 1 and 20, is stored in the variable secret.

```
print('I am thinking of a number between 1 and 20')
for i in range(1, 7):
    guess = int(input('Take a guess '))
```

- The program tells the player that it has come up with a secret number and will give the player six chancesto guess it.

- The code that lets the player enter a guess and checks that guess is in a for loop that will loop at most six times.
- The first thing that happens in the loop is that the player types in a guess.
- Since input() returns a string, its return value is passed straight into int(), which translates the string into an integer value. This gets stored in a variable named guess.

```
if guess < secret:
    print('Your guess is too low.')
elif guess > secret:
    print('Your guess is too high.')
```

- These few lines of code check to see whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen.

```
else:
    break
```

- If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number,in which case we want the program execution to break out of the for loop.

```
if guess == secret:
  print('Good job! You guessed my number in ', i , 'guesses!')
else:
  print('Nope. The number I was thinking of was ' , secret)
```

- After the for loop, the previous if...else statement checks whether the player has correctly guessed the number and prints an appropriate message to the screen.
- In both cases, the program displays a variable that contains an integer value(i and secret).
- Since it must concatenate these integer values to strings, it passes these variables to the str() function,which returns the string value form of these integers.
- Now these strings can be concatenated with the + operators before finally being passed to the print() function call.