



1BEC305-Python

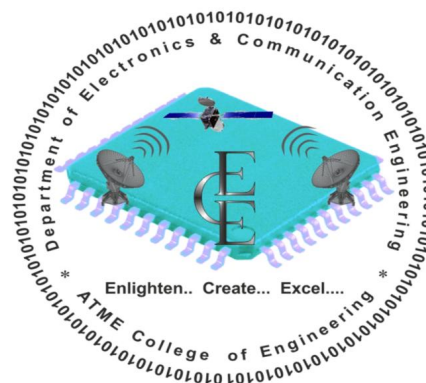
Introduction to Python Programming

1BEC305

*A Simplified Notes As per VTU Syllabus
2025 Scheme*

MODULE 1

Python: Basics



Dept. of E&CE
ATME College of Engineering, Mysuru
Affiliated to Visvesvaraya Technological University
Belagavi - 590018

Chapter 1: Python Basics

- **What is a Computer Program:** A computer program is a sequence of instructions that specifies how to perform a computation.
- **What is Python:** Python is a general-purpose, interpreted, object-oriented, powerful high-level programming language with dynamic semantics.
- **Python language features / Advantages / Characteristics *****
 1. **Simple and Readable Syntax:** Python's syntax is clear and easy to read, making it an ideal language for both beginners and experienced programmers. This simplicity can lead to faster development and reduce the complexity of code.
 2. **Rich Standard Library:** Python has several standard libraries that provide ready-to-use modules and functions for various tasks.
 3. **High-level Language:** It is easier for humans to understand and write. This nature makes Python suitable for rapid development and prototyping.
 4. **Dynamically Typed:** The data types of variables are determined during run-time. We do not need to specify the data type of a variable during writing codes.
 5. **Object-Oriented Programming (OOP) Support:** Python supports OOP paradigms, allowing developers to create reusable and modular code through classes and objects, encapsulation, inheritance, etc.
 6. **Interpreted Language:** Python is an interpreted language, meaning that code is executed line by line without the need for compilation. This enables rapid development cycles and facilitates interactive programming.
 7. **Integrated Language:** We can easily integrate it with other languages such as C, C++, etc.
 8. **Cross-platform Compatibility:** Python can be easily installed on Windows, macOS, and various Linux distributions, allowing developers to create software that runs seamlessly across different operating systems.
 9. **Open source and free:** It is an open-source language which means that anyone can download it, use it, and share it. Moreover, it is free of cost.
 10. **Garbage Collected:** Memory allocation and de-allocation are automatically managed. Programmers do not specifically need to manage the memory.

11. **Supports GUI programming:** Python has support for creating various GUI applications. Furthermore, these applications can work in many system software and libraries. PyQt5 is the most popular for making graphical apps.
 12. **Community Support and Documentation:** Python has an active developer community that provides support through forums, mailing lists, and online resources. Its official documentation is comprehensive and well-maintained, with detailed explanations, tutorials, and examples for all levels of developers.
-

- **Terminology: Interpreter and Compiler :**

1. **Compilers:** A compiler translates code from a high-level programming language into machine code before the program runs. It takes the **entire program as input**, hence memory requirement is greater. Errors are displayed after the entire program is checked. For example C compiler.
 2. **Interpreter:** An interpreter translates code written in a high-level programming language into machine code **line-by-line as the code runs**. Memory requirement is less. Errors are displayed for every instruction that is interpreted. Examples are Python, JavaScript, Ruby etc.
-

1.1. Entering expressions into the interactive shell:

- **Python IDLE:** Python interpreter is the software that runs Python programs in the Integrated Development and Learning Environment (IDLE). IDLE that comes as a default implementation when you install Python. It has two main window types, the *Shell window* and the *Editor window*. An IDE includes
 1. *A source code editor*
 2. *A compiler or interpreter*
 3. *An integrated debugger*
 4. *A graphical user interface (GUI)*
- **Python interpreter can be used in two basic modes:**
 1. *Interactive mode:* This mode is used when a user wants to run one single line or one block of code. Entering the command and expression at any prompt (`>>>`) of the shell. Faster results but I can't save and edit the code.
 2. *Script mode:* Script Mode, is used when the user is working with a block of code. This mode facilitates the development of more extensive, more structured programs. It is suitable for prototype development & can save and edit the code.

Math Operators Used in Python: (July 2024-6M)

There are plenty of other operators you can use in Python expressions, Table 1-1 lists all the math operators in Python.

Table 1-1: Math Operators from Highest to Lowest Precedence

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

- The order of operations (precedence) of Python math operators is similar to that of mathematics.
- The ****** operator is evaluated first; the *****, **/**, **//**, and **%** operators are evaluated next, from left to right; and the **+** and **-** operators are evaluated last (also from left to right).
- The BODMAS rule follows the order, ie B – Brackets, O – Order of powers or roots, D – Division, M – Multiplication A – Addition, and S – Subtraction.
- Mathematical expressions with multiple operators need to be solved from left to right in the order of BODMAS.
- Enter the following expressions into the interactive shell:

```
>>> 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.

1.2 Python Data Types: The Integer, floating-point and String***

- These datatypes are built-in datatypes in Python. In programming, datatype is an important concept.
- A data type is a category for values, and every value belongs to one data type. Some types of data types are as follows with example

▪ Integer type(int):

- It is a numeric data type in Python used to hold numeric values.
- The integer (or int) data type indicates values that are whole numbers.
- It is represented as *int* and it holds signed integers of non-limited length.
- For example *int*: 2, 3, 0, -13, 78, -132, etc.

▪ Float type(float):

- It is also a numeric data type in Python used to hold numeric values.
- The float (or float) data type indicates values that are decimal numbers.
- It is represented as *float* and it holds floating decimal points and it's accurate up to 15 decimal places. Ex Float: 2.345, -0.234, 5.0, 1.23, etc..

▪ String Type(str):

- Python programs can also have text values called strings, or *str*.
- Strings in Python are surrounded by either single quotation marks or double quotation marks. Ex: 'Hello' is the same as "Hello".
- It displays a string literal with the print() function.
- The string with no characters, ' ', is called a blank string.
- Data type can be identified of any object by using the **type()** function

Example	Compiled output
<pre> 1 x='Hi 1st sem students' 2 y= 2024 3 z= 20.345 4 print(type(x)) 5 print(type(y)) 6 print(type(z)) </pre>	<pre> <class 'str'> <class 'int'> <class 'float'> </pre>

- If the error message SyntaxError: EOL while scanning string literal, the final single quote character at the end of the string is probably missing.

```

>>> 'Hello world!
SyntaxError: EOL while scanning string literal

```

1.3 String concatenation and replication: (Dec 2023-6M)

- **String concatenation:**

- *Concatenation* is the process of **joining two or more strings together** and forming one single string.
- Different ways to concatenate strings using the **+** operator, **join()** method, **%** operator, **format()** function, Using **","** (comma) and Using **f-string**.
- For example, **+** is the addition operator when it operates on two integers or floating-point values.
- However, when **+** is used on two string values, it joins the strings as the string concatenation operator.
- **Example :**

```
>>> 'hello' + ' Good morning'
'hello Good morning'
```

- **String replication:**

- Replicating the same string "**n**" **several times** is called string *replication*.
- Use the ***** **operator** on a string to repeat the string a provided number of times.
- The syntax for the operation is: `result = 'string' * integer number`
- **Example:**

```
>>> 'suresh'*3
'sureshsureshsuresh'
```

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

1.4 Storing Values in Variables:

- Variables are boxes in a computer's memory for storing data values.
- If we need to use the result of an evaluated expression later, then the result must be stored in a variable.
- **Assignment Statements:** This statement is used to assign the value to the specified variable. An assignment statement consists of a variable name, an equal sign (called the assignment operator), and the value to be stored.

Ex: `spam = 42`



Figure 1-2: `spam = 42` is like telling the program, "The variable `spam` now has the integer value 42 in it."

- **Overwriting the variable:** It refers to the process of assigning a new value to an existing variable, thereby replacing its previous value.

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

❶ A variable is initialized the first time a value is stored in it.

❷ After that, use it in expressions with other variables and values.

❸ When a variable is assigned a new value, the old value is forgotten, which is why spam is evaluated to 42 instead of 40 at the end.

- **Variable Names ***:** A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). We can name a variable anything as long as it obeys the following rules:

1. It can be only **one word**.
2. It **can't begin** with a **number**.
3. It can use only **letters, numbers**, and the **underscore (_)** character.
4. Only contain **alpha-numeric** characters and underscores (A, 0-9, and _)
5. Variable names are **case-sensitive** (age, Age & AGE are 3 different variables)
6. A variable name cannot be any of the Python **keywords**.

- Following Table 1-3: gives examples of 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)

1.6 Dissecting the Program: By dissecting the any program, we learned some of the following Python Instructions and Functions.

1. Comments
2. Printing the User's Name
3. The `print()` Function
4. The `input()` Function
5. The `len()` Function
5. The `str()`, `int()`, & `float()` Functions

1. Comments: The following line is called a *comment*.

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

- Any text for the rest of the line following a hash mark (#) is part of a comment.
 - A comment is a text note that explains the source code.
 - It increases the readability of the program.
 - Python interpreters ignore these comments and do not execute them.
 - Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out code*.
 - Python also ignores the blank line after the comment.
-

2. Printing the User's Name:

- The following call to `print()` contains the expression `'It is good to meet you, ' + myName` between the parentheses (*see line 6 in the example program*)

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

- Remember that expressions can always evaluate to a single value.
 - This single string value is then passed to `print()`, which prints it on the screen.
-

3. The `print()` Function:*((June 2024-6M, July 2024-6M))**

- The `print()` function displays the string value inside the parentheses on the screen or any other standard output device.

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

- The line `print('Hello world!')` means Print out the text in the string 'Hello world!'. When Python executes this line, you say that Python is calling the `print()` function and the string value is being passed to the function.
 - A value that is passed to a function call is an *"argument"*.
 - The quotes are not printed on the screen. They just mark where the string begins and ends; they are not part of the string value.
 - We can also use this function to put a blank line on the screen; just call `print()` with nothing in between the parentheses.
-

4. The input() Function*:(June 2024-6M,July 2024-6M)**

- The **input()** function allows takes a user input. By default, it returns the user input in the form of a string.
- The **input()** function waits for the user to type some text on the keyboard and press ENTER.

Example: `myName = input()`

- The **input()** function calls as an expression that evaluates to whatever string the user typed in.

5. The len() Function*:(June 2024-6M, July 2024-6M)**

- The **len()** function evaluates to the integer value of the number of characters in that string.
- **Example 1:** Enter the following into the interactive shell.

```
>>> len('Good')
4
>>> len('Nothing is Permanent in this world')
34
>>> len('4most')
5
>>> len('')
0
```

6. The str(), int(), and float() Functions: *****

- **The str() Function:**

- The **str()** function converts the specified value into a string.
- The **str()** function can be passed an integer value and will evaluate to a string value version of it, as follows.

- **Example 1:**

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

- **Example 2:**

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
```

- **The int() Function:**

- The **int()** function converts the specified value into an integer number.
- It converts a number in a given base to a decimal.

Ex 1: `>>> int('42')`
42
`>>> int('-99')`
-99

Ex 2: `>>> int(1.25)`
1
`>>> int(1.99)`
1

▪ The **float()** Function:

- The **float()** function converts the specified value into a floating point number.

```
Ex: >>> float('3.14')
      3.14
      >>> float(10)
      10.0
```

FLOW CONTROL

○ Introduction:

- Control statements direct the flow of a Python program's execution. They introduce logic, decision-making, and looping.
- 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.
- Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles.
- Example

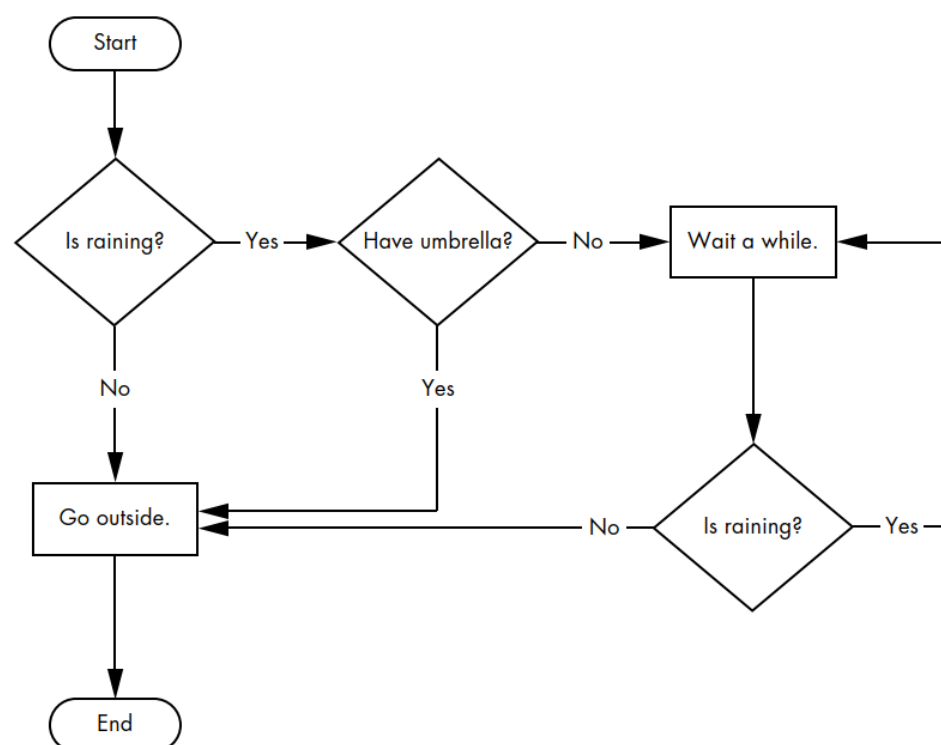


Figure 2-1: A flowchart to tell you what to do if it is raining

2. 1. Boolean Data types ***

- Boolean data types have one of two values: **True** or **False**.
- In programming we often need to know if an expression is **True** or **False**.
- In Python, these Boolean values always start with a capital T or F, with the rest of the word in lowercase. Example

```
>>> 10>9
      True
>>> 10 == 9
      False
>>> 10<9
      False
```

2.2 Comparison Operators*** : (July 2024-4M, Dec 2023-6M)

- The comparison operators are also called **relational operators**.
- Comparison operators **compare two values** and evaluate them down to a **single Boolean value**. The table below lists the comparison operators

Operator	Meaning	Example
<	Less than	a	Greater than	a>b
<=	Less than or equal to	a<=b
>=	Greater than or equal to	a>=b
==	Is equal to	a==b
!=	Is not equal to	a!=b

- These operators evaluate to True or False depending on the values we give them.

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

- The == and != operators can actually work with values of any data type.
- The <, >, <=, and >= operators, on the other hand, work properly only with integer and floating-point values.

▪ The Difference Between the == and = Operators: (July 2024-4M)

- The **= operator** is used for **assignment**, such as when assigning a value to a variable.
- The **== operator** is the **relational operator** for checking the **equality of two values**. If the values are the same, it will return True and will return False otherwise.

○ Example 1: Assign operator

```
>>> num = 15
>>> num
15

>>> num1 = 'EC'
>>> num1
'EC'
```

○ Example 2: Equality operator

```
>>> 3 == 3
True
>>> 5 == 8
False
>>> 'EC' == 'EC'
True
>>> 'CS' == 'EC'
False
```

2.3 Boolean Operators**

- Python has three Boolean operators, or logical operators: **and**, **or**, & **not**.
- These logical operators are used to compare Boolean values, as shown below.

Operator	Meaning	Example
and	True if both the operands are true	X and Y
or	True if either of the operands is true	X or Y
not	True if the operand is false	not X

▪ and operator :

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

▪ or operator :

- The or operator evaluates an expression to True if **either** of the two Boolean values is True. If both are False, it evaluates to False

○ Table: The 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 False
False
>>> False or True
True
>>> True or False
True
>>> True or True
True
```

▪ Not operator:

- The not-operator operates on only one Boolean value (or expression). The not operator simply evaluates to the opposite Boolean value

○ Table: The **not** Operator's Truth Table

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

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

2.4 Mixing Boolean and Comparison Operators:

- Since the comparison operators evaluate 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. The computer's evaluation process for $(4 < 5)$ and $(5 < 6)$ is shown in Figure below:

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

- The multiple Boolean operators in an expression, along with the comparison operators shown below

```
>>> 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 and comparison operators are evaluated, Python evaluates the **not** operators first, then the **and**, and then the **or** operators

2.5. Elements of Flow Control: Flow control includes (Dec 2023-6M)

1. Conditional statements
2. Block of codes

1. Conditional statements:

- Condition is just a more specific name in the context of flow control statements. Eg (if, elif, else), loops (for, while).
- 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 almost every flow control statement uses a condition.

2. Block of codes:

- Lines of Python code can be grouped together in blocks. There are three rules for blocks.
 - a. Blocks begin when the indentation increases.
 - b. Blocks can contain other blocks.
 - c. Blocks end when the indentation decreases to zero or to a containing block's indentation.

○ Example:

```

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

```

▪ Description:

- The first block of code ❶ starts at the line `print('Hello Mary')` and contains all the lines after it.
- Inside this block 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.')`.

2.6 Flow Control Statements:***** Major control statements used in python programming are

1. if Statements
2. else Statements
3. elif Statements
4. while loop Statements
5. break Statements
6. Continue statements
7. for loops and the range() function
8. The Starting, Stopping, and Stepping Arguments to range()

1. if Statements : (July 2024-10M, Dec 2023-6M)

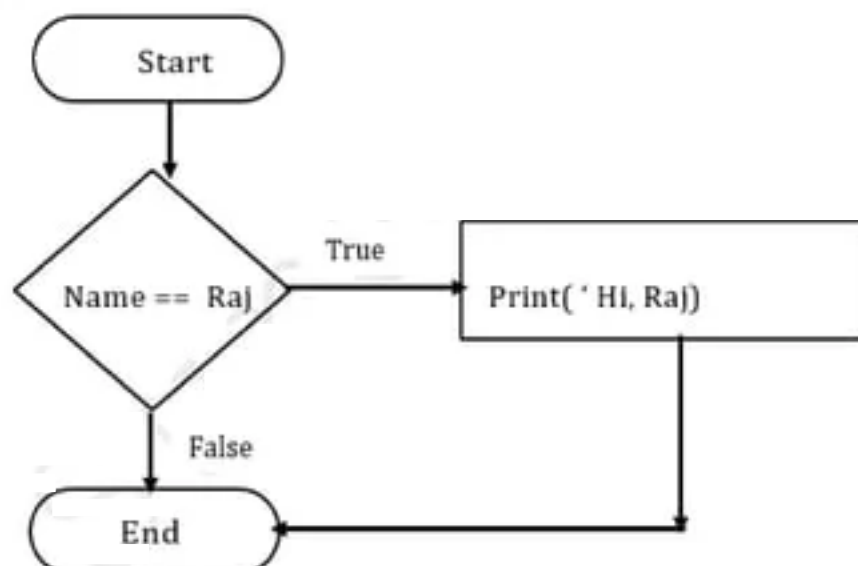
- The most common type of flow control statement is the **if** statement
- An 'if statement executes a block of code only if the **specified condition is met**.
- In Python, an if statement consists of the following:

1. The **if** keyword
2. A condition (that is, an expression that evaluates to True or False)
3. A colon
4. Starting on the next line, an indented block of code (called the if clause)

- **Example:** for if statement

```
>>> if name == Raj:
    Print(' Hi Raj')
```

- **Flowchart:** The flowchart for an if statement



2. **else** Statements : (July 2024-10M, Dec 2023-6M)

- 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**.
- **else** statement could be read as, "If this condition is true, execute this code. Or else, execute that code."
- An **else** statement doesn't have a condition, and in code, an **else** statement always consists of the following:

1. *The else keyword*

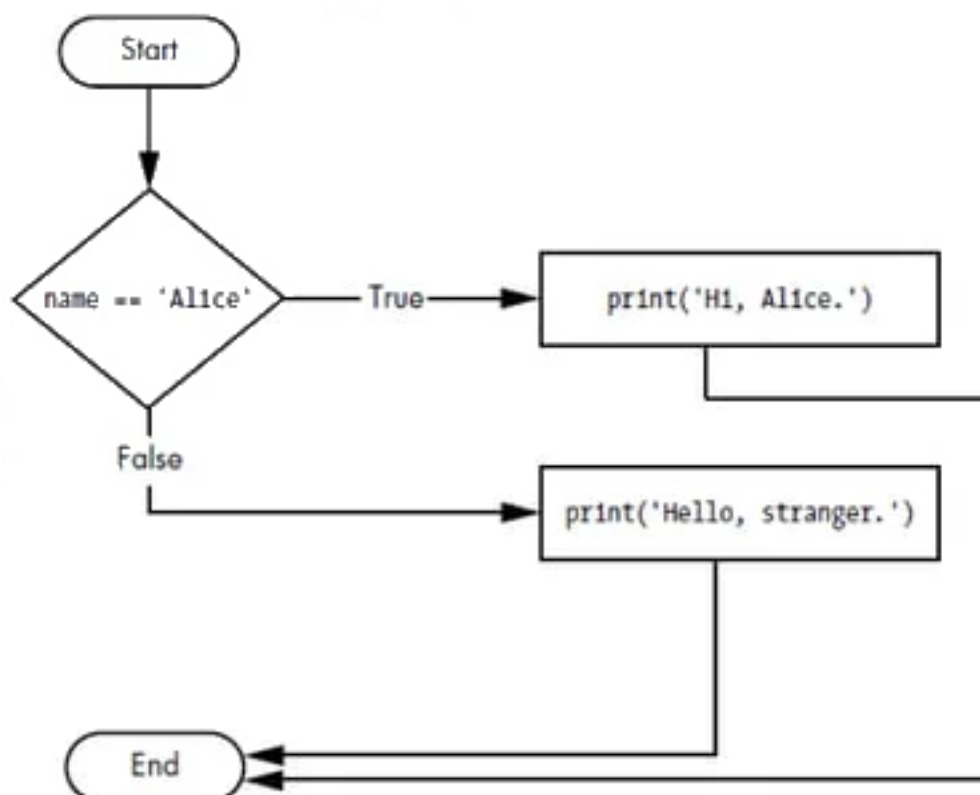
2. *A colon*

3. *Starting on the next line, an indented block of code (called the else clause)*

- **Example:** for else statement

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

- **Flowchart:** The flowchart for an **else** statement



3. elif Statements: (June 2024-6M, July 2024-6M, Dec 2023-6M)

- In Python programming, the "else if" statement, is often called "elif".
- It is a conditional statement that allows you to specify multiple conditions to be evaluated sequentially. It provides another condition that is checked only if all of the previous conditions were False.

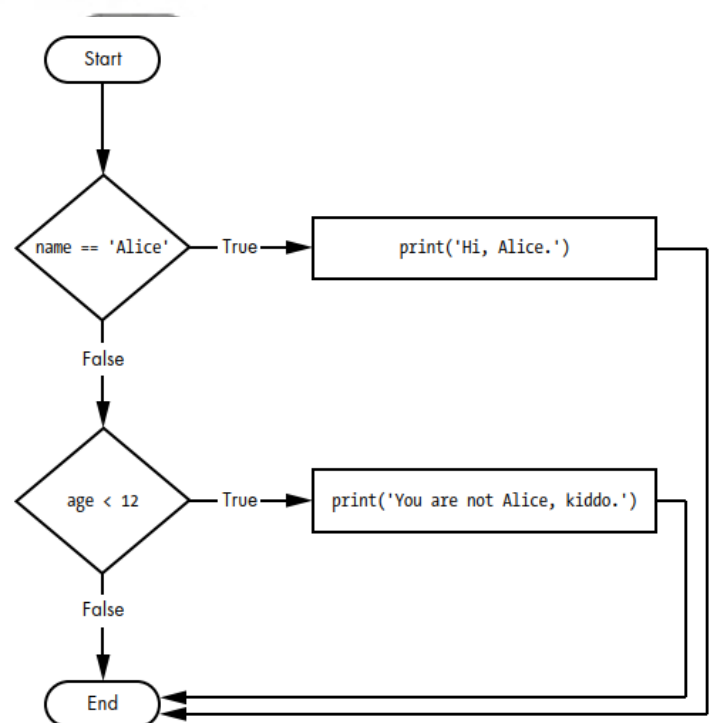
1. The *elif* keyword
2. A condition (that is, an expression that evaluates to True or False)
3. A colon
4. Starting on the next line, an indented block of code (called the *elif* clause)

▪ **Example:** for *elif* statement

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

- **Flowchart:** The flowchart for an *elif* statement

```
# if..else chain statement
letter = "A"
if letter == "B":
    print("letter is B")
else:
    if letter == "C":
        print("letter is C")
    else:
        if letter == "A":
            print("letter is A")
        else:
            print("letter isn't A, B and C")
```



Output

```
letter is A
```

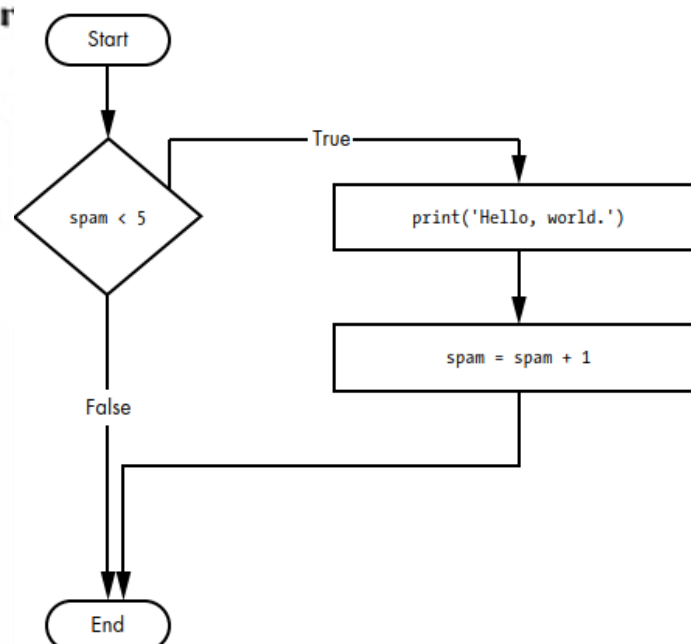
- Remember, at most only one of the clauses will be executed, and for *elif* statements, the order matters!

4. While loop Statements: (June 2024-6M, July 2024-6M)

- Python While Loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.
- In code, a while statement always consists of the following:
 1. *The while keyword*
 2. *A condition (that is an expression that evaluates to True or False.*
 3. *A colon*
 4. *Starting on the next line, an indented block of code (called the while clause)*
- **Example:** for a while loop statement

Using if statement	Using while statement
<pre>spam = 0 if spam < 5: print('Hello, world.') spam = spam + 1</pre>	<pre>spam = 0 while spam < 5: print('Hello, world.') spam = spam + 1</pre>

- **Description:**
- These statements are similar—both if and while checking the value of spam, and if it's less than five, they print a message.
- But when we run these two code snippets, for the if statement, the output is simply "Hello, world."
- But for the *while* statement, it's "Hello, world." repeated five times.
- **Flowchar**



while statements.

5. for loop :: (June 2024-6M, July 2024-6M)

- A for loop is used for iterating over a sequence.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.
- In code, a for statement always consists of the following:
 1. *The for keyword*
 2. *A variable name*
 3. *The in Keyword*
 4. *A call on the range*
 5. *A Colon*

○ **Example:**

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

5. break Statements:

- 'Break' in Python is a loop control statement. It is used to control the sequence of the loop.
- The break keyword is used to **break out a for loop, or a while loop**. 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:** for break statement

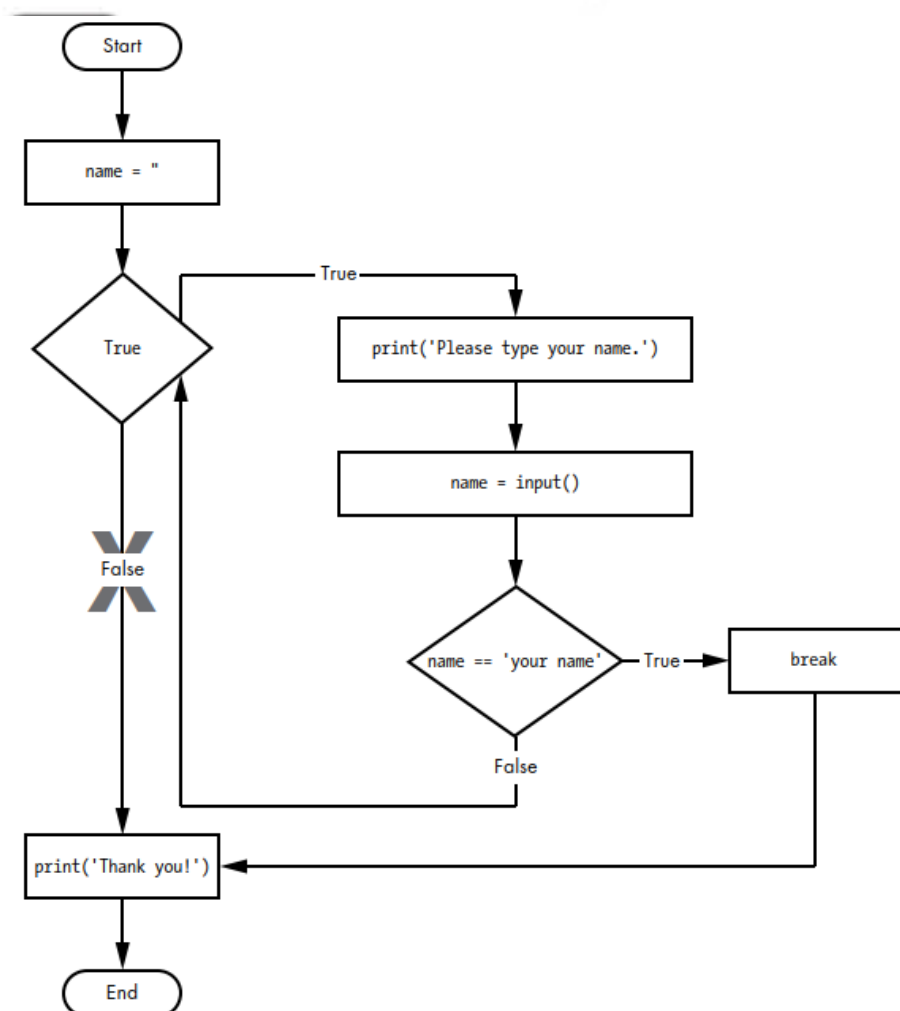
```
❶ while True:
    print('Please type your name.')
❷ name = input()
❸ if name -- 'your name':
❹     break
❺ print('Thank you!')
```

○ **Description:**

- The first line ❶ creates an infinite loop; it is a while loop whose condition is always True.
- The program execution will always enter the loop and will exit it only when a break statement is executed.
- 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 your name is equal to your name.
 - If this condition is True, the break statement is run ❹, and the execution moves out of the loop to 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 2:** The flowchart for the program with an infinite loop. Note that the X path will logically never happen because the loop condition is always True.

○ **Flowchart 2:**



6. Continue statements:

- The continue keyword is used to end the current iteration in a for loop (or a while loop) and continues to the next iteration. It is used inside the loop.
- When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.
- **Example:** for continue statements

```

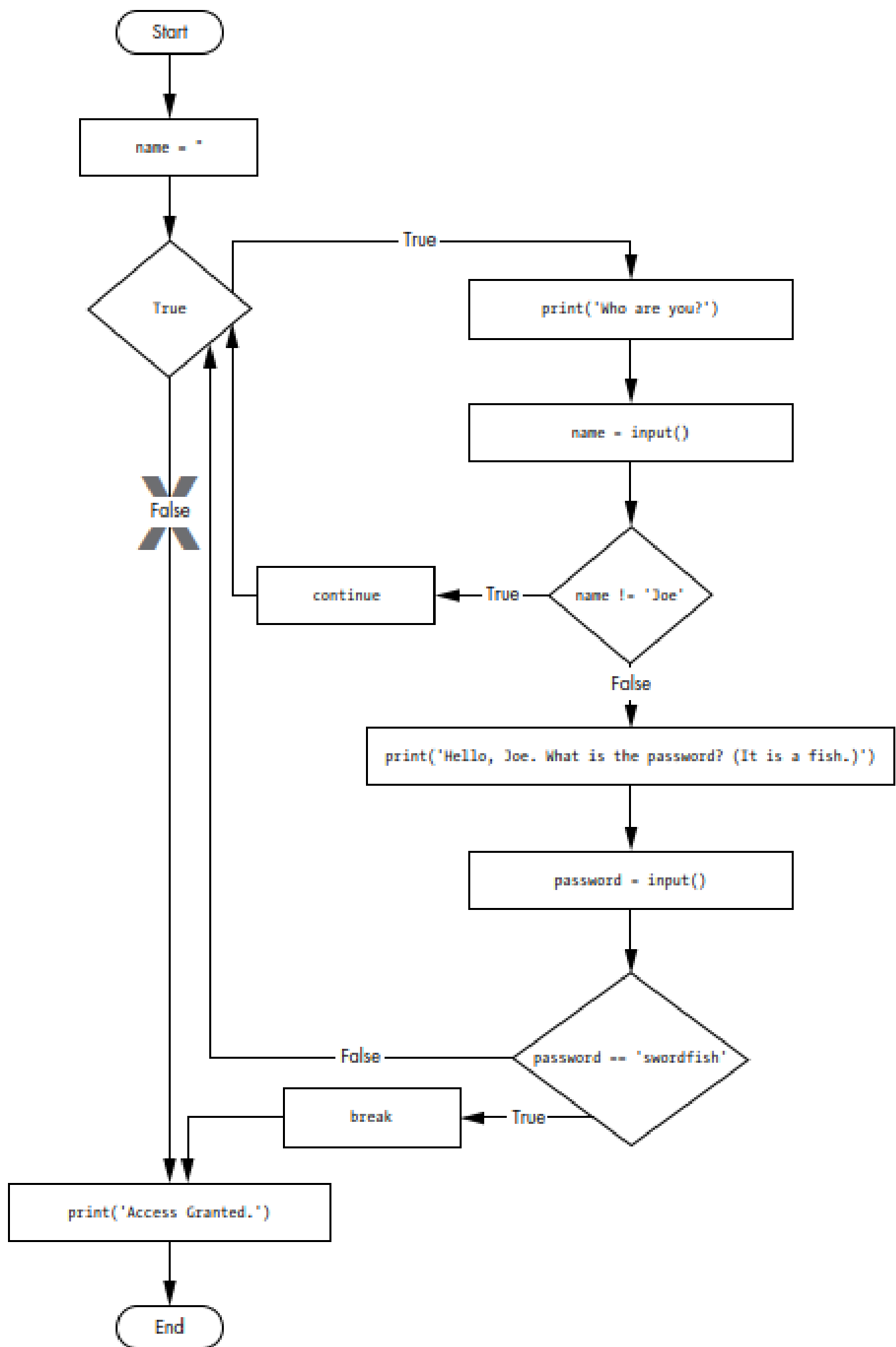
while True:
    print('Who are you?')
    name = input()
    ❶ if name != 'Joe':
    ❷     continue
    print('Hello, Joe. What is the password? (It is a fish.)')
    ❸ password = input()
    if password == 'swordfish':
    ❹     break
    ❺ print('Access granted.')
  
```

- **Output:**

```

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.
  
```

- **Description:**
- If the user enters any name besides Joe ❶, the continue statement ❷ causes the program execution to jump back to the start of the loop.
- When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the 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 of the 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 of the loop.



7. for loops and the range() function:

- Execute a block of code only a certain **number of times** 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

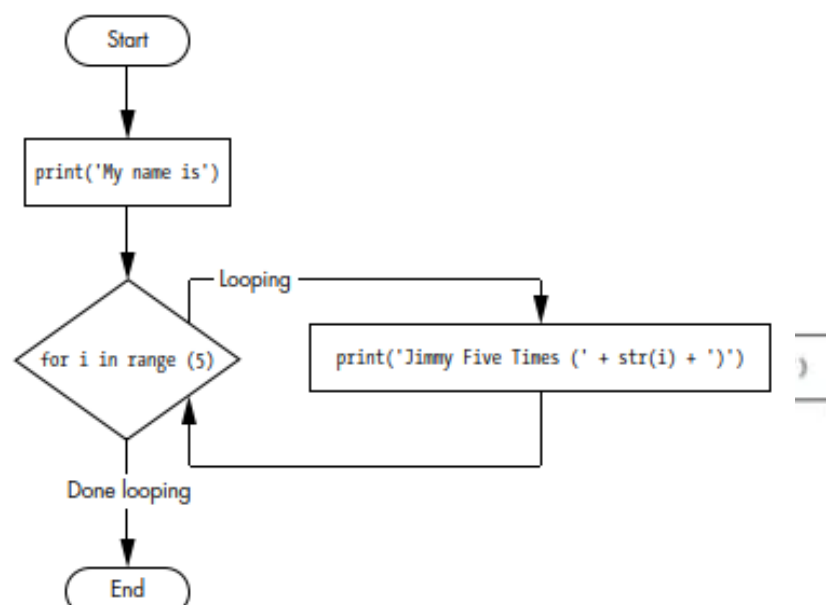
○ Example1 and output:

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

- 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 (0).
- After Python finishes an iteration through all the code inside the for loop's clause, the execution goes back to the top of the loop, & 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().

▪ Flowchart: for range function



▪ Example 1:

```

1 total = 0
2 for num in range(101):
3     total = total + num
4 print(total)

```

- The result should be 5,050. When the program first starts, the total variable is set to 0 **1**.
- The for loop **2** then executes `total = total + num` **3** 100 times.
- By the time the loop has finished all of its 100 iterations, every integer from 0 to 100 will have been added to the total. At this point, the total is printed on the screen **4**.

▪ An equivalent while loop: For the first example of a for loop.

```

print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1

```

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

- Functions that can be called with multiple arguments separated by a comma are called **range()** functions.
- This lets us change the integer passed to **range()** to follow any sequence of integers, including starting at a number other than zero.

```

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

```

- 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.

```

12
13
14
15

```

- The **range()** function can also be called with three arguments. The first two arguments will be the *start* and *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 loops. We can even use a 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)
```

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

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

2.8 Importing Modules

- All Python programs can call a basic set of functions called built-in functions, including the **`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 in the programs.
- For example, the `math` module has mathematics-related functions, the `random` module has random number-related functions, and so on.
- Before using the functions in a module, we must import the module with an import statement. In code, 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 you pass it.
- 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

FUNCTIONS

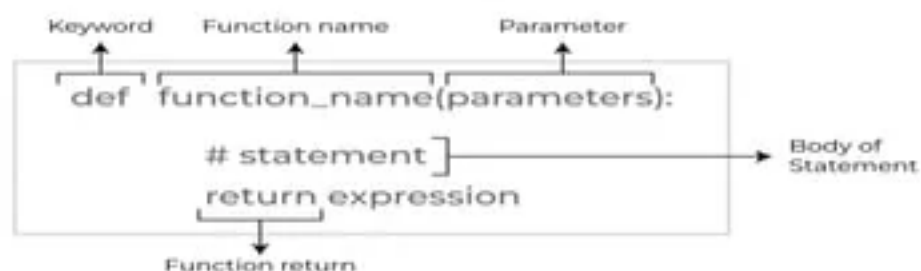
○ Introduction: (July 2024-6M)

- Function is a block of code that only runs when called. It is like a mini-program.
- The idea is to use the same block of code repeatedly during programming in terms of function whenever required.
- We can pass data, known as parameters, into a function. It can return data as a result. A major purpose of functions is to group code that gets executed multiple times
- Python provides several built-in functions like print(), len(), input(). Etc. but you can also write your functions.
- In Python a function is created using the def keyword.

○ Benefits of Using Functions

1. Increase Code Readability
2. Increase Code Reusability

○ Python Function Declaration: The syntax to declare a function is:



○ Types of Functions in Python:

1. **Built-in library function:** These are Standard functions in Python that are available to use. Example `print()`, `len()` etc..
2. **User-defined function:** We can create our own functions based on our requirements.

○ Example: *Program*

```

❶ def hello():
❷     print('Howdy!')
    print('Howdy!!!')
    print('Hello there.')

❸ hello()
  hello()
  hello()

```

Output

```

Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.

```

▪ Program Description:

- 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 executed when 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 number of arguments in between the parentheses.
- When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there.
- When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.
- Since this program calls `hello()` three times, the code in the `hello()` function is executed three times. When we run this program, the output looks like the above.

3.1 def Statements with Parameters:

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

○ Example with output

<pre>❶ def hello(name): ❷ print('Hello ' + name) ❸ hello('Alice') hello('Bob')</pre>	<pre>_____ Hello Alice Hello Bob _____</pre>
---	--

▪ Program Description:

- The definition of the hello() function in this program has a parameter called name ❶.
- A parameter is a variable stored in an argument 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, & the variable name is automatically set to 'Alice', which is what 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 the function returns.

3.2 Return Values and return Statements: (July 24-6M)

- The value that a function call evaluates is called the **return value** of the function. Ex: len('Hello') → Return value is 5
- The Python return statement is used to return a value from a function.
- The user can only use the return statement in a function. It cannot be used outside of the Python function.
- When creating a function using the def statement, you can specify what the return value should be with a return statement.
- A return statement consists of the following:
 1. *The return keyword*
 2. *The value or expression that the function should return.*
- When an expression is used with a return statement, the return value is what this expression evaluates to.
- For example, the following program defines a function that returns a different string depending on what number it is passed as an argument.

```

❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)

```

3.3 None value:

- In Python there is a value called None, which represents the absence of a value.
- The **None** keyword is used to define a null value or no value at all.
- None is not the same as 0, False, or an empty string. None is the only value of the **NoneType** data type.
- Python None is the function returns when there are no return statements.
- One place where None is used is as the return value of print().
- Example:

```

def check_return():
    pass
print(check_return())

```

Output:

None

3.4 Keyword Arguments and Print ()

- Python allows to passing of function arguments in the form of keywords which are called named **arguments**.
- Most arguments are identified by their position in the function call.

- 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.

<pre>print('Hello') print('World')</pre>	<pre>Hello World</pre>
--	----------------------------

- 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, can set the `end` keyword argument to change this to a different string.
- For example, if the program and its output were this:

<pre>print('Hello', end='') print('World')</pre>	<pre>HelloWorld</pre>
<pre>print('Hello', end=' ') print('World')</pre>	<pre>Hello World</pre>

- 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***(June 24-6M, July 24-8M)

- 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 your program begins. When your 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 you run it:

<i>Program</i>	<i>Output → Error</i>
<pre>def spam(): eggs = 31337 spam() print(eggs)</pre>	<pre>Traceback (most recent call last): File "C:/test3784.py", line 4, in <module> print(eggs) NameError: name 'eggs' is not defined</pre>

- 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 the variable eggs is no longer present.
- **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():
    ❶ eggs = 99
    ❷ bacon()
    ❸ print(eggs)

    def bacon():
        ham = 101
        ❹ eggs = 0

    ❺ spam()

```

- When the program starts, the spam() function is called ❺, 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 egg—which is different from 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 in the spam() function to print the value of eggs ❸, and since the local scope for the call to spam() still exists here, 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)

```

- Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs are 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 local variable.
- But technically, it's perfectly legal to do so.

Example program

```

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'

❸ eggs = 'global'
  bacon()
  print(eggs)    # prints 'global'

```

Output

```

bacon local
spam local
bacon local
global

```

- There are actually three different variables in this program, but confusingly they are all named eggs. The variables 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 The Global Statement

- Global statement is used to modify a global variable within a function.
- If we have a line such as global eggs at the top of a function, it tells Python, "In this function, eggs refer to the global variable, so don't create a local variable with this name."
- For example:

```

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

eggs = 'global'
spam()
print(eggs)

```

When you run this program, the final print() call will output this:

```
spam
```

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

3.7 Exception Handling: (June 2024-6M, July 2024-8M)

- **Definition:** Exception is an event (error) which occurs during the execution of the program.
- 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.

- For example: Program

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

output

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

- A ZeroDivisionError happens whenever we try to divide a number by zero. From the line number given in 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 the following except clause if an error happens.
- 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.

Program

```
def spam(divideBy):
    try:
        return 42 / divideBy
    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
```

- Note that any errors that occur in function calls in a try block will also be caught. Consider the following program, which instead has the spam() calls in the try block:

Program

```
def spam(divideBy):
    return 42 / divideBy

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.
```
