PYTHON MODULE 1 NOTES

Syllabus: **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

CHAPTER 1

1. DEFINITION : **Python** is a high-level, interpreted, and general-purpose programming language that emphasizes simplicity and readability. It is designed to be easy to learn and use, making it an excellent choice for both beginners and experienced developers. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

Python is widely used in various fields such as web development, data analysis, artificial intelligence (AI), machine learning (ML), scientific computing, automation, and more. Its extensive standard library and vast ecosystem of third-party packages make it a powerful and versatile tool for solving diverse problems efficiently

In simple terms, Python is a programming language that's like a set of instructions you give to a computer to make it do something. Imagine teaching someone how to bake a cake step-by-step; Python lets you give instructions to a computer in a similar way.

Here's why Python is great and easy to use:

Simple to Read and Write: Python uses plain English-like words, so it's easier to understand, even for beginners. For example: print("Hello, world!"). This simply tells the computer to display "Hello, world!" on the screen.

Versatile: You can use Python to do many things, like:

a. Build websites
b. Analyze data
c. Create games
d. Control robots or devices
e. Work on Artificial Intelligence (AI) and Machine Learning (ML)

Beginner-Friendly: Python takes care of many complicated details for you, so you can focus on solving problems rather than getting stuck on complex coding rules.

Huge Community: If you ever get stuck, there are lots of tutorials, forums, and people who can help.

Python is like a friendly tool that helps you bring your ideas to life through a computer!

NOTE: Python was created by **Guido van Rossum** in 1991. He is often referred to as Python's **"Benevolent Dictator For Life" (BDFL)** because he guided the development of Python for many years.

The name "Python" was inspired by the British comedy series *Monty Python's Flying Circus*, not the snake, reflecting van Rossum's preference for a fun and approachable language.

2. ENTERING EXPRESSIONS INTO THE INTERACTIVE SHELL

What is the Interactive Shell?

The interactive shell is a place where you can type Python code and immediately see the result. It's like a calculator but can do much more.

How to Open It?

On **Windows**: Go to the Start menu, find the Python program group, and click **IDLE (Python GUI)**.

How to Use It?

At the prompt `>>>`, you can type commands or math expressions. For example:

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

When you press **Enter**, Python calculates the result and shows `4`.

What is an Expression?

- An **expression** is a piece of code that Python can calculate and give back a value.
- For example, `2 + 2` is an expression. Python calculates it and gives the result, `4`.

Values and Operators

- A **value** is just a number or a piece of data, like `2`.
- An **operator** is something that tells Python what to do with the values. For example:
  - `+` adds two numbers.
  - `-` subtracts one number from another.

Single Values are Also Expressions

If you type just a single value, like `2`, Python will still recognize it as an expression. It simply evaluates to itself:

```
>>> 2
2
```

Conclusion: **In short:** You can use the interactive shell to quickly do calculations or test Python code. Expressions combine values and operators, and Python calculates their result for you!

There are plenty of other operators you can use in Python expressions, too. For example, 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

When doing math in Python, **operators** (like `+`, `-`, `*`, `/`) follow specific rules, just like in regular math. These rules are called the **order of operations** or **precedence**. Python uses the same rules as you learned in school to decide what to calculate first.

Order of Operations (Precedence)

1. Exponents (`**`) come first
Example: >>> 2 ** 3

8 # (2 raised to the power of 3)

2. Multiplication (`*`), Division (`/`), Floor Division (`//`), and Modulus (`%`) come next

These are calculated from **left to right**.

Examples: >>> 10 * 2 / 5

4.0 # First, 10 * 2 = 20, then 20 / 5 = 4.0

3. Addition (`+`) and Subtraction (`-`) come last

These are also calculated from **left to right**.

Example: >>> 5 + 3 - 2

6 # First, 5 + 3 = 8, then 8 - 2 = 6

Overriding the Order with Parentheses

If you want to change the order Python calculates things, use **parentheses**. Python will always calculate what's inside parentheses first.

With parentheses

>>> (2 + 3) * 4

20 # First, 2 + 3 = 5, then 5 * 4 = 20

Without parentheses:

>>> 2 + 3 * 4

14 # First, 3 * 4 = 12, then 2 + 12 = 14

Python won't be able to understand it and will display a SyntaxError error message, as shown here:

```
>>> 5 +
  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<stdin>", line 1
    42 + 5 + * 2
             ^
SyntaxError: invalid syntax
```

3. THE INTEGER, FLOATING-POINT, AND STRING DATA TYPES

When you work with Python, you'll deal with values (numbers, text, etc.). Each value belongs to a specific category, which is called a data type

What is a Data Type?

A **data type** defines the kind of value you're working with. Every value in Python belongs to one (and only one) data type.

Common Data Types in Python

Here are the most common ones:

1. **Integer (int)**
   - Whole numbers (no decimal points).
   - Examples: -2, 0, 30, 42
   - These are called **integers** or **ints**.
2. **Floating-Point Number (float)**
   - Numbers that have a **decimal point**.
   - Examples: 3.14, 42.0, -7.5
   - These are called **floats**.
3. **String (str)**
   - Text or characters inside quotes.
   - Examples: "Hello", 'Python', "123" (even though it looks like a number, it's text because of the quotes).

Difference Between int and float

- **int**: Whole numbers.
  Example: 42
- **float**: Numbers with decimals.
  Example: 42.0

Even though both 42 and 42.0 represent the same value, Python treats them differently because one has a decimal point.

In Simple Terms

- A **data type** is like a label that tells Python, "Hey, this is a number" or "This is text."
- Common data types:
  - **int**: Whole numbers (e.g., 5, -2).
  - **float**: Numbers with decimals (e.g., 3.14, 42.0).
  - **str**: Text (e.g., "hello", "123").

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'

4. STRING CONCATENATION AND REPLICATION

1. The + Operator

- For numbers: Adds them.
  Example: 2 + 3 → 5
- For strings: Combines (concatenates) them.
  Example: 'Alice' + 'Bob' → 'AliceBob'
- Mixing strings and numbers: Causes an error.
  Example: 'Alice' + 42 → TypeError
  Solution: Convert the number to a string using str().
  Example: 'Alice' + str(42) → 'Alice42'

2. The * Operator

- For numbers: Multiplies them.
  Example: 2 * 3 → 6
- For a string and a number: Repeat the string.
  Example: 'Alice' * 5 → 'AliceAliceAliceAliceAlice'
- Mixing incompatible types: Causes an error.
  Example: 'Alice' * 'Bob' → TypeError

Key Points

- Operators behave differently depending on the data types of the values.
- Mixing incompatible types (e.g., string + number) gives an error unless you explicitly convert the data type.
- Use + for string concatenation and * for string repetition.

5. STRING CONCATENATION AND REPLICATION

What is a Variable?

A variable is like a **labeled box** in your computer's memory where you can store a value.

spam = 42

Example:Here, spam is the variable, and 42 is the value stored in it.

How to Use Variables

Use the = sign (assignment operator) to store a value in a variable.

Example:
spam = 40
eggs = 2
print(spam + eggs) # Output: 42

Updating Variables (Overwriting)

You can change the value of a variable anytime, and the old value will be forgotten.

Example:
spam = 'Hello'
spam = 'Goodbye'
print(spam) # Output: Goodbye

Rules for Variable Names

- Must be one word.
- Can use letters, numbers, and _ (underscore).
- Cannot start with a number.
  Examples:
  Valid: spam, eggs2, _hello
  Invalid: 2spam, spam!, hello world

Valid Variable Names

1. my_var
2. age42

Invalid Variable Names

1. 42age *(Cannot start with a number)*
2. my-var *(Hyphens are not allowed)*

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)

6. YOUR FIRST PROGRAM

1. Opening the File Editor: The interactive shell is where you type one command at a time, but to write full programs, you'll use the file editor.

To open the file editor in IDLE:

- On Windows: Go to File > New Window.
- On macOS: Select File > New File.

The file editor looks like a regular text editor, but for Python code. It doesn't have the >>> prompt, unlike the interactive shell.

2. Writing the Code:
Type the following code into the file editor:

```
# This program says hello and asks for my name.
print('Hello world!')
print('What is your name?')  # ask for their name
myName = input()
print('It is good to meet you, ' + myName)
print('The length of your name is:')
print(len(myName))
print('What is your age?')  # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

What this code does:

1. **Prints a greeting**: Hello world!
2. **Ask for your name**: What is your name?
3. **Prints a greeting with your name**: It is good to meet you, [Your Name]
4. **Shows the length of your name**: How many letters your name has.
5. **Asks for your age**: What is your age?
6. **Tells you what your age will be next year**: It adds 1 to your age.

Saving the Program:

a. After writing the code, **save your program** so you don't lose it.

i. Go to **File** > **Save As**.
ii. Name your file hello.py and click **Save**.

Running the Program:

- To run the program, press **F5** or go to **Run** > **Run Module**.
- Your program will run in the interactive shell (where you first started IDLE).
- It will ask you for your name and age and then give you the responses like this:

Closing and Reopening the Program:

- Once the program finishes, it stops running.
- You can close the file editor, but don't forget to save your program before closing it.

- If you want to reopen it later, go to **File** > **Open**, choose `hello.py`, and click **Open**. Your program will appear again in the editor.

In Summary:

- **Write** your code in the file editor.
- **Save** your program with a `.py` file extension.
- **Run** it to see the output.
- **Save** often to avoid losing your work!

This is how you create and run a SIMPLE Python program.

7. DISSECTING YOUR PROGRAM

Comments (#):

- Anything after a # is a comment. Python ignores comments, so they're just for you to make notes or explain your code.
- For example, # This program says hello is a comment, and Python doesn't run it.

The print() Function:

- print() displays text on the screen.
- For example: print('Hello world!')
- This will show "Hello world!" on the screen.
- You can also print empty lines with `print()` (without anything inside the parentheses).

The input() Function:

- input() asks the user to type something and then stores it.
- Example: myName = input()

Using + to Combine Text:

- You can **combine** text and variables. For example print('It is good to meet you, ' + myName)
- If `myName` is 'Al', it will print: `It is good to meet you, Al`.

The `len()` Function:

- `len()` finds out how many characters are in a string.
- Example: print(len(myName))
- If `myName` is 'Al', it will print `2` because "Al" has 2 letters.

Using `str()` and `int()`:

- If you want to **convert** between different types of data (like numbers and text), you use `str()` and `int()`.
- `str()` turns numbers into text:

print('I am ' + str(29) + ' years old.')
This prints: I am 29 years old.
int() turns text (like '29') into a number so you can do math:
age = input() # If user enters '4'
print('You will be ' + str(int(age) + 1) + ' in a year.')

Error Handling:

- If you try to convert something that's not a number (like 'hello') using int(), Python will give an error:

int('hello') # This will give an error.

Why Use str() and int():

The input() function always gives you text (even if you type a number). To do math, you need to convert it to a number with int(). When printing results, you might need to turn numbers back into text with str() to combine them.

CHAPTER 2

1. BOOLEAN VALUES

Boolean values in Python are a special type of data that can only be either True or False. These values are not enclosed in quotes, and the first letter must always be capitalized (e.g., True, False).

For example:

- When you write spam = True, the variable spam now holds the Boolean value True.
- If you try to write true (with a lowercase "t"), Python will give an error because it doesn't recognize it as the Boolean value True.

Additionally, True and False cannot be used as variable names. For example, if you try to assign a value like True = 2 + 2, Python will give an error because True is a reserved word in Python.

```
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
```

2. COMPARISON OPERATORS

Table 2-1: Comparison Operators

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

These operators evaluate to `True` or `False` depending on the values you give them. Let's try some operators now, starting with `==` and `!=`.

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

NOTE: The == operator checks if two values are equal. For example, 5 == 5 is True because both sides are the same.

The = operator is used to assign a value to a variable. For example, x = 5 means the variable x now holds the value 5.

To remember:

- == checks if two things are the same (equality).
- = is for assigning a value to a variable.

3. BOOLEAN OPERATORS

Boolean operators and, or, and not are used to work with Boolean values (True or False).

- The and operator checks if both values are True. If both are True, the result is True; otherwise, it's False.
  For example:
  - True and True gives True
  - True and False gives False

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

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the and operator.

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

On the other hand, 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.

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

The `not` operator reverses a Boolean value:

- If the value is **True**, not True gives **False**.
- If the value is **False**, not False gives **True**.

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

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

-

4. MIXING BOOLEAN AND COMPARISON OPERATORS

You can combine comparison operators (like <, >, ==) with Boolean operators (and, or, not) in a single expression.

- Comparison operators give True or False.
- Boolean operators then use those True or False values to give a final result.

For example, 4 < 5 and 3 > 2 will first compare the numbers and then apply the and operator to check if both comparisons are True.

x = 4
y = 7
z = 3

```
# Using comparison and 'and' operator
result = (x < y) and (z > 2)
print(result)  # This will print True because both conditions are true: 4 < 7 and 3 > 2
```

Explanation:

- `x < y` checks if 4 is less than 7, which is **True**.
- `z > 2` checks if 3 is greater than 2, which is also **True**.
- The `and` operator combines these results, and since both are **True**, the final result is **True**.

Another example with `or`:

```
result2 = (x > y) or (z > 2)
print(result2)  # This will print True because one condition is true: 3 > 2
```

Explanation:

- `x > y` checks if 4 is greater than 7, which is **False**.
- `z > 2` checks if 3 is greater than 2, which is **True**.
- The `or` operator gives **True** if at least one condition is **True**. So, the final result is **True**.

5. ELEMENTS OF FLOW CONTROL

In Python, flow control helps decide which code to run based on certain conditions. Here's a breakdown:

1. Conditions: A condition is a statement that evaluates to either True or False. For example, 4 < 5 is a condition that is True.
2. Blocks of Code: A block is a group of lines of code that are executed together. In Python, blocks are defined by indentation. Indentation means adding spaces or tabs at the beginning of a line to indicate that it belongs to the same block.
    - Rule 1: A block starts when indentation increases.
    - Rule 2: A block can contain other blocks (nested).
    - Rule 3: A block ends when indentation decreases.

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

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.')`.

6. PROGRAM EXECUTION

In a basic program, Python executes instructions one by one from top to bottom. However, with flow control statements, the program can jump to different parts of the code or skip sections, depending on conditions, altering the sequence of execution. (It is saved as .py)

7. FLOW CONTROL STATEMENTS

(a) if statement

An if statement checks if a condition is True. If it is, the code inside the block (called the clause) runs. If the condition is False, the code inside the block is skipped. The statement starts with "if", followed by the condition, a colon, and then an indented block of code.

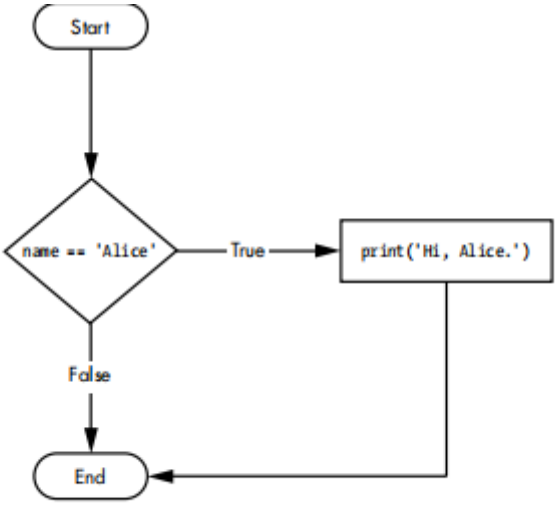


Figure 2-3: The flowchart for an if statement

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

(b) else statement

An **else statement** provides an alternative block of code to run when the condition in the **if statement** is **False**. It doesn't need a condition and is used to say, "If this condition is true, run this code; otherwise, run this code." It starts with the "else" keyword, followed by a colon and an indented block of code.

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

```
else:
    print('Hello, stranger.')
```

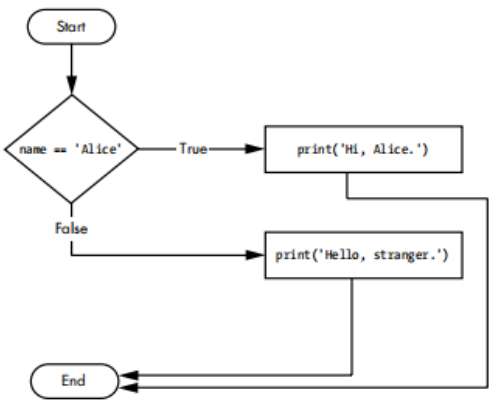


Figure 2-4: The flowchart for an else statement

(c) elif statement

An elif (short for "else if") statement lets you check multiple conditions. It comes after an if or another elif statement, and only gets checked if the previous conditions were False. If the elif condition is True, its block of code will run. It starts with the "elif" keyword, followed by a condition, a colon, and an indented block of code.

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

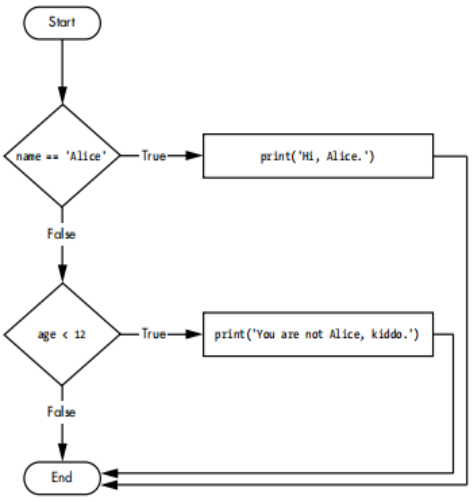


Figure 2-5: The flowchart for an elif statement

(d) while loop

A while loop runs a block of code repeatedly as long as a condition is True. It starts with the "while" keyword, followed by a condition, a colon, and then an indented block of code. If the condition becomes False, the loop stops.

```
count = 0
while count < 5:
   print(count)
   count += 1
```

This code will print the numbers 0 to 4. The loop continues as long as `count` is less than 5. After each loop, `count` is increased by 1.

Initialization:

- count = 0 is executed. This means count is now set to 0.

First Check of the While Loop:

- The while loop checks if count < 5. Since count is 0, the condition is **True**.

First Loop Iteration:

- Inside the loop, print(count) is executed, so it prints 0 (the current value of count).
- Then, count += 1 is executed. This increases count from 0 to 1.

Second Check of the While Loop:

- The loop checks again if count < 5. Since count is now 1, the condition is still **True**.

Second Loop Iteration:

- print(count) is executed, so it prints 1.
- Then, count += 1 increases count from 1 to 2.

Third Check of the While Loop:

- The loop checks if count < 5. Since count is now 2, the condition is still **True**.

Third Loop Iteration:

- print(count) prints 2.
- count += 1 increases count from 2 to 3.

Fourth Check of the While Loop:

- The loop checks if count < 5. Since count is now 3, the condition is still **True**.

Fourth Loop Iteration:

- print(count) prints 3.
- count += 1 increases count from 3 to 4.

Fifth Check of the While Loop:

- The loop checks if count < 5. Since count is now 4, the condition is still **True**.

Fifth Loop Iteration:

- print(count) prints 4.
- count += 1 increases count from 4 to 5.

Sixth Check of the While Loop:

- The loop checks if count < 5. Since count is now 5, the condition is **False**.

End of Loop:

- Since the condition is now **False**, the loop exits and the program ends.

OUTPUT

```
0
1
2
3
4
```

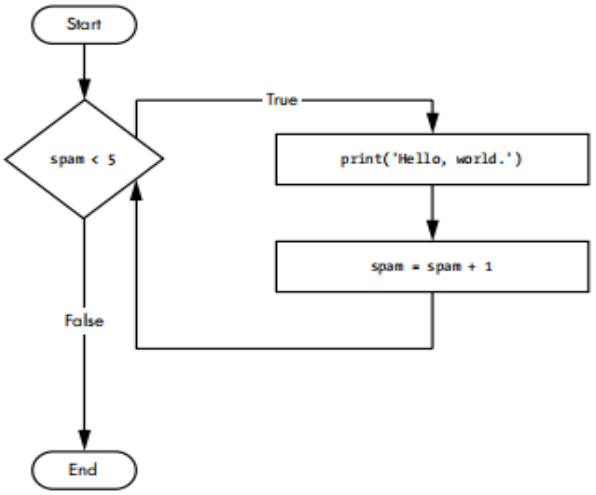


Figure 2-10: The flowchart for the while statement code

(d) break statement

The break statement is used to immediately exit from a loop (like a while or for loop) before it naturally finishes. When the program reaches the break statement, it stops executing the loop and moves to the next part of the code.

```
count = 0
while True:  # Infinite loop
   count += 1
   print(count)
   if count == 5:
      break  # Exit the loop when count is 5
```

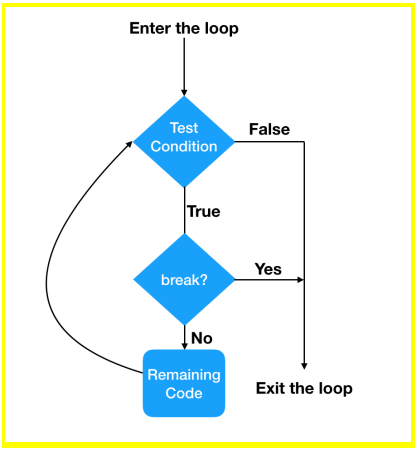
Explanation:

1. The loop starts and count is incremented.
2. It prints the current count.
3. When count reaches 5, the if count == 5: condition becomes True.
4. The break statement is triggered, and the loop is immediately stopped.
5. The program continues after the loop.

OUTPUT

```
1
2
3
4
5
```

The loop stops once it reaches 5 because of the break.



(e) for loop and range function

A for loop is used in Python when you want to repeat a block of code a specific number of times. The loop iterates over a sequence (like a list, string, or range) and performs an action for each item in that sequence.

The range() function: The range() function is often used in a for loop to generate a sequence of numbers. It can accept up to three arguments:

1. Start (optional): The starting number (default is 0).
2. Stop: The number where the loop stops (this number is not included in the sequence).
3. Step (optional): The number to increment between each value (default is 1).

Example

```
for i in range(5):
    print(i)
```

range(5) creates a sequence of numbers from 0 to 4 (it does not include 5).
The loop runs 5 times, and each time the variable i takes the next value in the sequence (0, 1, 2, 3, 4).
It prints each value of i one by one.

OUTPUT

```
0
1
2
3
4
```

Example with different range()

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

range(1, 10, 2) starts at 1, stops before 10, and increments by 2.
So, the sequence will be: 1, 3, 5, 7, 9.
It prints these values

OUTPUT

1
3
5
7
9

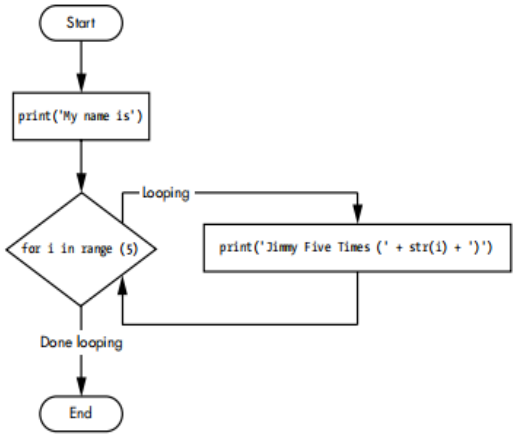


Figure 2-14: The flowchart for fiveTimes.py

(f) continue statement

The continue statement in Python is used to skip the rest of the code inside the current loop iteration and move to the next iteration of the loop. It is often used when you want to avoid executing certain code for specific conditions while continuing the loop.

Why is it used?

- To skip unwanted parts of the loop for specific conditions.
- To improve readability and avoid deeply nested code.

Example:

```
for number in range(1, 6):
   if number == 3:  # Skip when number is 3
      continue
   print(number)
```

- The loop iterates through numbers from 1 to 5.
- When number == 3, the continue statement skips the print() and moves to the next iteration.
- All other numbers are printed.

OUTPUT

1
2
4
5

The number **3** is skipped because of the `continue` statement.

8. IMPORTING MODULES

In Python, a module is simply a file that contains Python code, including functions, classes, and variables. It helps to organize your code into smaller, reusable pieces, making your program easier to manage.

Why are modules needed?

- Code Reusability: You can write a piece of code once in a module and reuse it in multiple programs.
- Organization: It helps to break your code into smaller, manageable parts.
- Avoid Duplication: If you need the same function or class in multiple programs, you can import the module instead of rewriting the code.

Examples of Python modules:

1. math: Provides mathematical functions (like square root, power, etc.).

```
import math
print(math.sqrt(16))  # Output: 4.0
```

2. random: Helps to generate random numbers.

```
import random
print(random.randint(1, 10))  # Output: A random number between 1 and 10
```

3. os: Interacts with the operating system (like working with files and directories).

```
import os
print(os.getcwd())  # Output: Current working directory
```

4. **time**: Used for working with time-related tasks.

```
import time
print(time.time())  # Output: Current time in seconds since the epoch
```

9. ENDING A PROGRAM EARLY WITH SYS.EXITT)

In Python, you can end a program early using sys.exit(). This is helpful when you want to stop the program immediately if a certain condition is met, such as an error or invalid input.

How does sys.exit() work?

- The sys.exit() function stops the program wherever it is called.
- To use it, you must first import the sys module.

```
import sys
while True:
    print("Type 'exit' to quit the program.")
    user_input = input("Enter something: ")
    if user_input == "exit":
```

```
        print("Goodbye!")
        sys.exit()  # Ends the program here
    print(f"You entered: {user_input}")
```

Explanation of the example:

1. The program runs inside a loop and repeatedly asks the user to type something.
2. If the user types `'exit'`, the program prints "Goodbye!" and calls `sys.exit()`, which stops the program immediately.
3. If the user types anything else, the program continues.

CHAPTER 3

1. def STATEMENTS WITH PARAMETERS

In Python, functions can take parameters to make them more flexible and useful. A parameter is a placeholder for the value (called an argument) that you pass to the function when calling it. This helps functions perform specific tasks based on the input you provide.

Key Concepts:

1. Defining a function with parameters:
   When you define a function, you can specify one or more parameters in parentheses. These act like temporary variables within the function.
2. Passing arguments:
   When calling the function, you provide the actual value (argument) for the parameter. The function then uses that value while it runs.
3. Scope of parameters:
   Parameters only exist inside the function. Once the function finishes, the parameter variables are no longer available.

Example:

Here's how this works:

```
def hello(name):  # 'name' is the parameter
    print('Hello ' + name)

hello('Alice')  # 'Alice' is the argument
hello('Bob')    # 'Bob' is the argument
```

Explanation:

1. Defining the function (def hello(name):):
   - The function hello has one parameter: name.
   - When the function is called, the value you pass (Alice or Bob) is temporarily stored in the variable name.
2. Calling the function (hello('Alice')):

- When hello('Alice') is called, the value 'Alice' is assigned to name. The function prints 'Hello Alice'.
- Similarly, hello('Bob') prints 'Hello Bob'.

OUTPUT

```
Hello Alice
Hello Bob
```

NOTE: If you try to use the parameter variable (name) outside the function, it will cause an error because the parameter exists only inside the function.

```
hello('Alice')
print(name)  # This will cause an error because 'name' does not exist outside the function.
```

2. RETURN VALUESLAND RETURN STATEMENTS

In Python, a return statement is used in a function to send a value back to the code that called it. This returned value is called the return value. The `return` keyword is followed by the value or expression that you want to return.

Key Points:

1. A function with a `return` statement evaluates to the value specified in the `return`.
2. If a function has no `return` statement, it returns `None` by default.
3. You can use the return value of a function in other parts of your program.

Example 1: Returning a value

```
def add(a, b):
    return a + b  # Returns the sum of a and b

result = add(3, 5)
print(result)  # Output: 8
```

Example 2: Using an expression with return

```
def square(num):
    return num * num  # Returns the square of the number

print(square(4))  # Output: 16
```

Example 3: Conditional return

```
def magic8Ball(answerNumber):
    if answerNumber == 1:
        return "Yes"
    elif answerNumber == 2:
        return "No"
```

```
    else:
        return "Ask again later"

response = magic8Ball(1)
print(response)  # Output: Yes
```

Why is return useful?

- It lets your function send results back to where it was called.
- You can use the returned value in other calculations or decisions.

3. THE NONE VALUE

In Python, None is a special value that means "no value" or "nothing." It is the only value of the NoneType data type and is written with a capital N. Other programming languages may call this value null, nil, or undefined.

Key Points:

1. None is not the same as 0, False, or an empty string. It specifically means "nothing."
2. Functions that do not have a return statement automatically return None.
3. You can use None as a placeholder value in your code when you need to indicate "nothing" or "not yet set."

Example 1: None as a return value

```
def my_function():
    print("This function does something but doesn't return a value.")

result = my_function()  # The function prints something but doesn't return anything
print(result)  # Output: None
```

Explanation:

- The my_function() prints text, but it does not explicitly return a value.
- Python automatically returns None, which is stored in the variable result.

Example 2: Using None as a placeholder

```
value = None  # Placeholder value to indicate "nothing"
if value is None:
    print("Value is not set yet.")
```

Output: Value is not set yet.

Example 3: Comparing with None

```
spam = print("Hello!")  # The print() function returns None
print(spam == None)  # Output: True
```

OUTPUT

```
Hello!
True
```

Summary:

- **None** represents "no value."
- Functions without a return statement automatically return **None**.
- It is often used to indicate a placeholder or missing value.

4. KEYWORD WORD ARGUMENTS AND print( )

When calling a function, arguments can be passed in two ways:

1. Positional Arguments: Based on the order in which they are passed.
2. Keyword Arguments: By explicitly naming the parameter and assigning it a value.

1. Positional Arguments

The position of the argument matters.
Example:

```
import random

print(random.randint(1, 10))  # Correct: 1 is the low end, 10 is the high end

# random.randint(10, 1) would cause an error
```

2. Keyword Arguments

- Keyword arguments are written as parameter=value.
- They are useful for **optional parameters**.

3. Examples with the print() Function

Default behavior:

```
print("Hello")

print("World")
```

OUTPUT

```
Hello

World
```

4. Changing the end keyword:

The `end` parameter controls what is printed at the end of the line.

```
print("Hello", end=" ")  # Replace newline with a space
print("World")
```

OUTPUT

```
Hello World
```

5. Using the `sep` keyword:

The `sep` parameter controls how multiple arguments are separated.

```
print("cats", "dogs", "mice", sep=",")  # Use a comma instead of space
```

OUTPUT

```
cats,dogs,mice
```

Why Use Keyword Arguments?

- **Flexibility**: You can customize the behavior of functions.
- **Clarity**: It makes your code more readable by showing what each argument means.

5. LOCAL AND GLOBAL SCOPE

1. Global Scope

- Variables defined outside all functions are global variables.
- They are accessible anywhere in the program.
- There is only one global scope, and it lasts for the entire program.

2. Local Scope

- Variables defined inside a function are local variables.
- They are accessible only inside that function.
- A local scope is created when the function is called and is destroyed when the function ends.

Why Use Local Variables?

- They keep the function independent and make debugging easier.
- Using global variables everywhere can make it hard to track bugs in larger programs.

Key Rules:

1. Global variables cannot be modified directly inside a function unless declared using the `global` keyword.

2. Local variables exist only inside their function and are destroyed after the function ends.
3. Functions can access global variables, but local variables from one function cannot be accessed in another function.

Example 1: Local and Global Variables

```
x = 10  # Global variable

def my_function():

    y = 5  # Local variable

    print("Inside function: x =", x)  # Access global variable

    print("Inside function: y =", y)

my_function()

# print(y)  # This will cause an error because y is local and not accessible here

print("Outside function: x =", x)  # Global variable is still accessible
```

OUTPUT

```
Inside function: x = 10

Inside function: y = 5

Outside function: x = 10
```

Example 2: Modifying Global Variables

```
z = 20  # Global variable

def change_global():

    global z  # Declare z as global

    z = 50  # Modify global variable

    print("Inside function: z =", z)

change_global()

print("Outside function: z =", z)  # Global variable modified
```

OUTPUT

```
Inside function: z = 50

Outside function: z = 50
```

Scope means where a variable can be used in your program.

1. Global Scope

- Variables created **outside functions** are **global variables**.
- You can use them **anywhere** in the program.

2. Local Scope

- Variables created **inside functions** are **local variables**.
- You can use them **only inside that function**.

Why Is This Important?

- **Global variables** are shared by the whole program.
- **Local variables** exist only inside a function. This makes functions safer and easier to debug!

6. EXCEPTION HANDLING

What is Exception Handling?

- When your program runs into an error (called an exception), it can stop working and crash.
- Exception handling lets you catch those errors, fix them, and keep the program running smoothly instead of crashing.

How Does It Work?

- try block: You write the code that might cause an error.
- except block: If an error happens in the try block, Python will jump to the except block and handle the error instead of crashing.

Simple Example:

Let's say we want to divide a number by another number. What if the user enters 0? This will cause a Zero Division Error, and the program will crash. We can prevent that using exception handling.

```
def spam(divideBy):
    try:
        return 42 / divideBy  # This might cause an error
    except ZeroDivisionError:
        return "You can't divide by zero!"  # Handle the error
    return 42 / divideBy

# Test with different values
print(spam(2))   # Normal division
print(spam(12))  # Normal division
print(spam(0))   # Error case, will be handled
print(spam(1))   # Normal division
```

Explanation of the Code:

- The try block contains the division code, which may cause an error if divideBy is 0.
- The except block catches the **ZeroDivisionError** and returns a friendly message instead of crashing.
- If there is no error, the division happens normally.

OUTPUT:

21.0

3.5

You can't divide by zero!

42.0

Summary:

- Use try to attempt risky code.
- Use except to handle any errors that happen in the try block.
- This prevents your program from crashing when errors occur.

7. A SHORT PROGRAM:GUESS THE NUMBER

In this simple game, the computer thinks of a number between 1 and 20, and you have to guess it. After each guess, the program will tell you if your guess is too high or too low, until you get the correct number. Let's break it down in simple steps:

Game Flow:

1. The computer picks a random number between 1 and 20 (you don't know it).
2. You make a guess by typing a number.
3. The computer compares your guess to its number and tells you:
    - If your guess is too low (meaning the number is higher).
    - If your guess is too high (meaning the number is lower).
4. You keep guessing until you find the correct number.
5. Once you guess it right, the program congratulates you and tells you how many guesses it took.

Example:

When you run the program, the output looks like this:

I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15

```
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
16
Good job! You guessed my number in 4 guesses!
```

You start by guessing 10. The computer says it's too low.

Then, you guess 15. It's still too low.

Then, you guess 17. The computer says it's too high.

Finally, you guess 16, and the computer says "Good job! You guessed my number in 4 guesses!"

How Does It Work?

- The program uses a loop to let you keep guessing until you get it right.
- It compares your guess with the computer's number using if statements.
- It counts how many guesses it takes using a counter.

Summary:

- The program thinks of a number.
- You guess numbers, and it tells you if your guess is too high or too low.
- The game ends when you guess correctly!