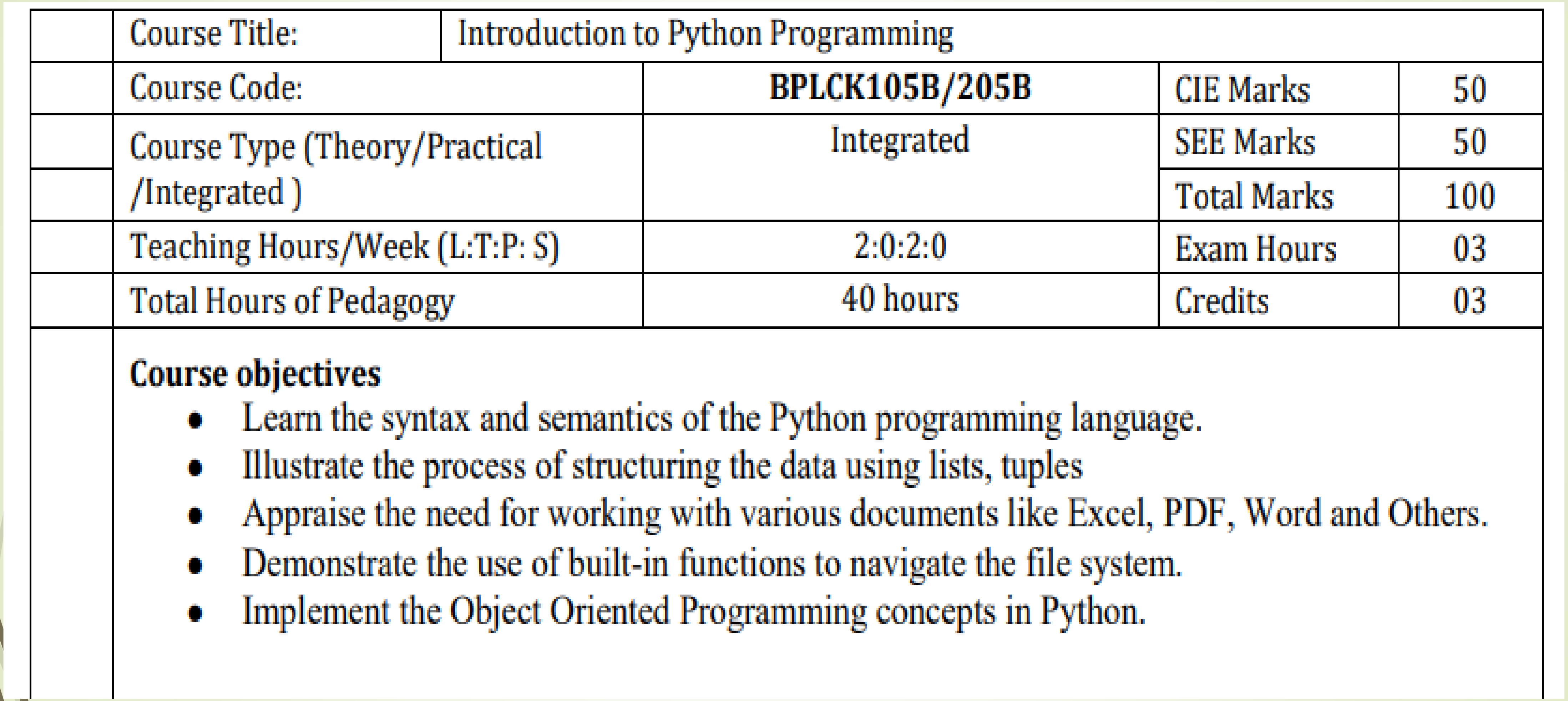
Introduction to Python Programming

Dr. Sandeep Bhat
Professor,CSE Dept
SIT,Mangaluru

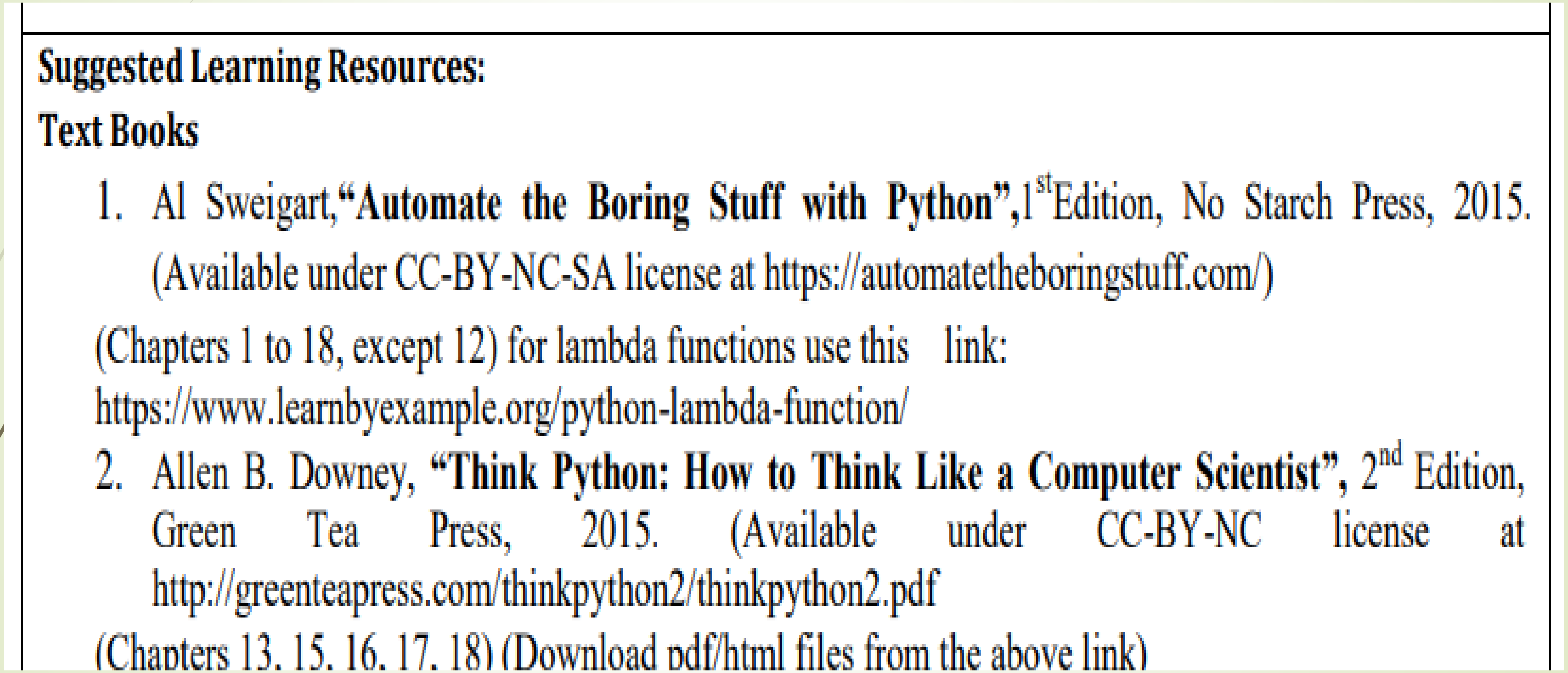


	Course Title:	Introduction to Python Programming		
	Course Code:	**BPLCK105B/205B**	CIE Marks	50
	Course Type (Theory/Practical /Integrated)	Integrated	SEE Marks	50
			Total Marks	100
	Teaching Hours/Week (L:T:P: S)	2:0:2:0	Exam Hours	03
	Total Hours of Pedagogy	40 hours	Credits	03

Course objectives

- Learn the syntax and semantics of the Python programming language.
- Illustrate the process of structuring the data using lists, tuples
- Appraise the need for working with various documents like Excel, PDF, Word and Others.
- Demonstrate the use of built-in functions to navigate the file system.
- Implement the Object Oriented Programming concepts in Python.

Cont...



Suggested Learning Resources:

Text Books

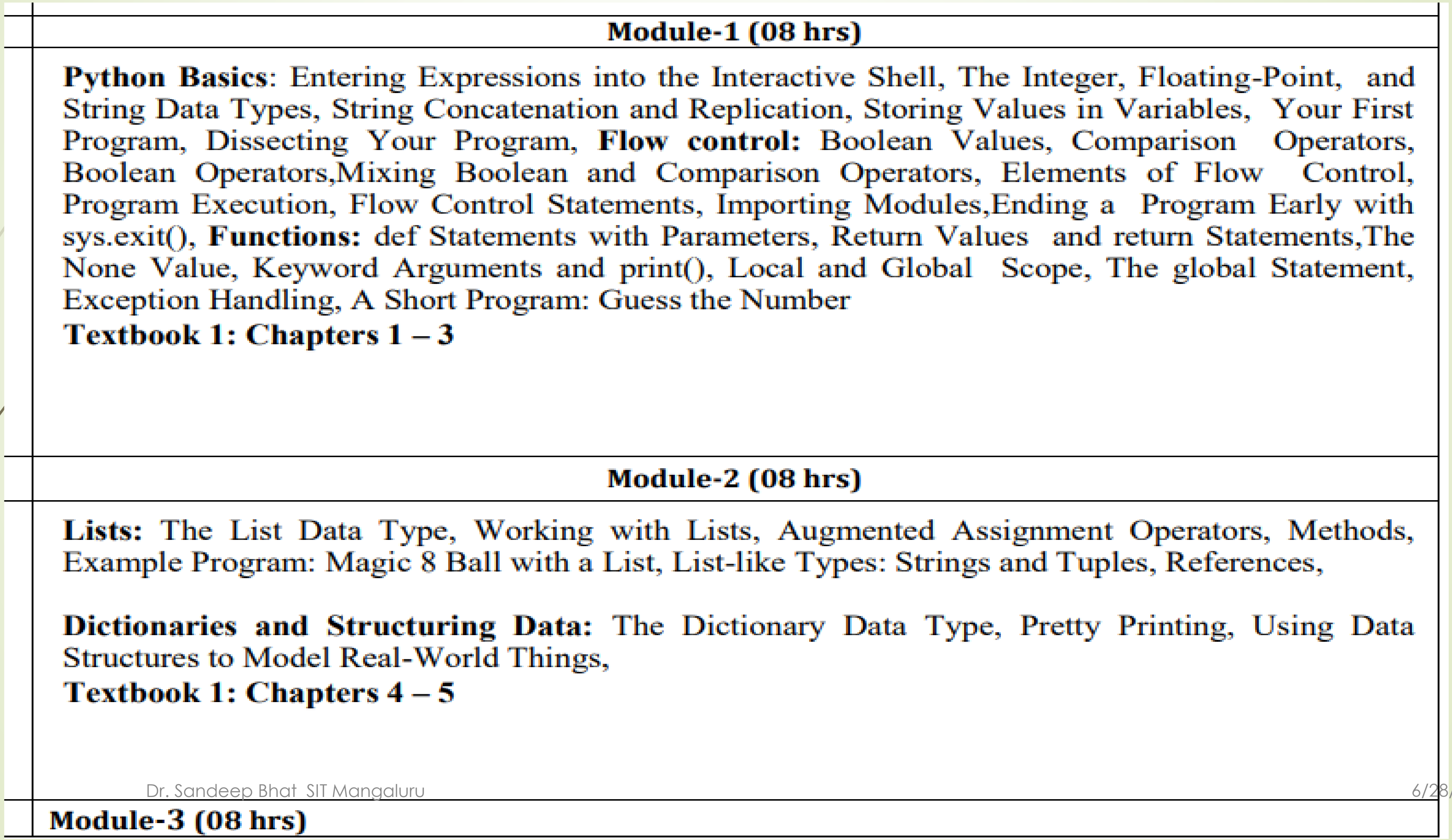
1. Al Sweigart,**"Automate the Boring Stuff with Python",**1st Edition, No Starch Press, 2015. (Available under CC-BY-NC-SA license at https://automatetheboringstuff.com/)

(Chapters 1 to 18, except 12) for lambda functions use this link: https://www.learnbyexample.org/python-lambda-function/

2. Allen B. Downey, **"Think Python: How to Think Like a Computer Scientist",** 2nd Edition, Green Tea Press, 2015. (Available under CC-BY-NC license at http://greenteapress.com/thinkpython2/thinkpython2.pdf

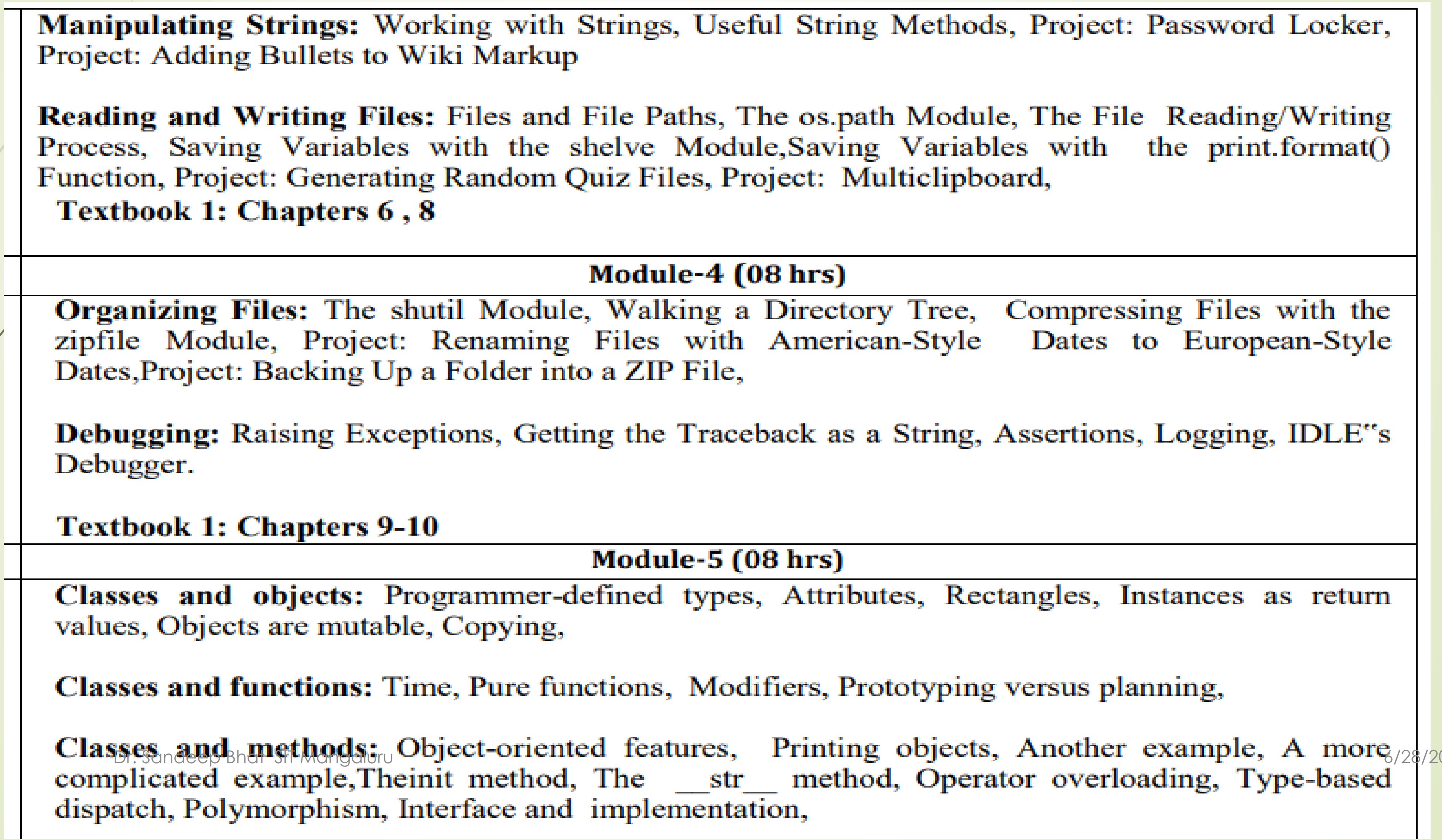
(Chapters 13, 15, 16, 17, 18) (Download pdf/html files from the above link)

Cont..



Module-1 (08 hrs)
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**
Module-2 (08 hrs)
Lists: The List Data Type, Working with Lists, Augmented Assignment Operators, Methods, Example Program: Magic 8 Ball with a List, List-like Types: Strings and Tuples, References, **Dictionaries and Structuring Data:** The Dictionary Data Type, Pretty Printing, Using Data Structures to Model Real-World Things, **Textbook 1: Chapters 4 – 5**
Module-3 (08 hrs)

Cont..



Manipulating Strings: Working with Strings, Useful String Methods, Project: Password Locker, Project: Adding Bullets to Wiki Markup

Reading and Writing Files: Files and File Paths, The os.path Module, The File Reading/Writing Process, Saving Variables with the shelve Module,Saving Variables with the print.format() Function, Project: Generating Random Quiz Files, Project: Multiclipboard,

Textbook 1: Chapters 6 , 8

Module-4 (08 hrs)

Organizing Files: The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates,Project: Backing Up a Folder into a ZIP File,

Debugging: Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE"s Debugger.

Textbook 1: Chapters 9-10

Module-5 (08 hrs)

Classes and objects: Programmer-defined types, Attributes, Rectangles, Instances as return values, Objects are mutable, Copying,

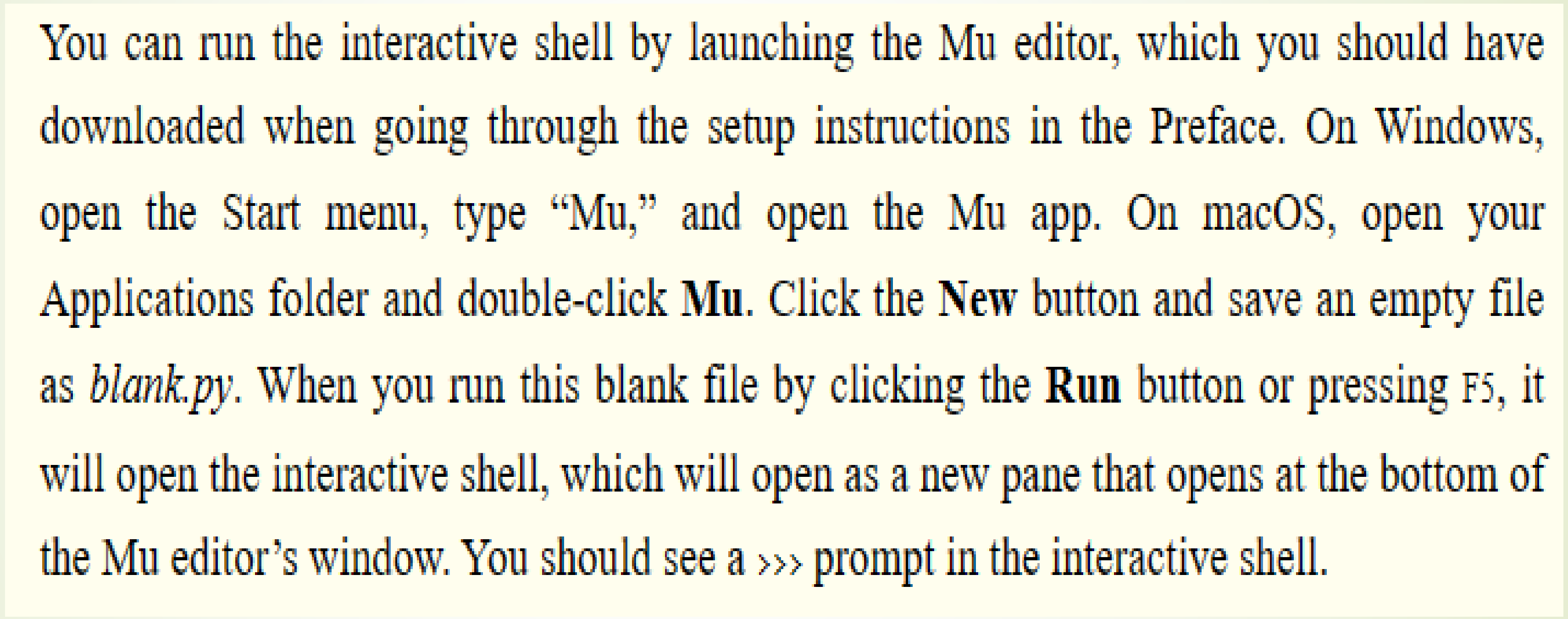
Classes and functions: Time, Pure functions, Modifiers, Prototyping versus planning,

Classes and methods: Object-oriented features, Printing objects, Another example, A more complicated example,Theinit method, The __str__ method, Operator overloading, Type-based dispatch, Polymorphism, Interface and implementation,

MODULE-1

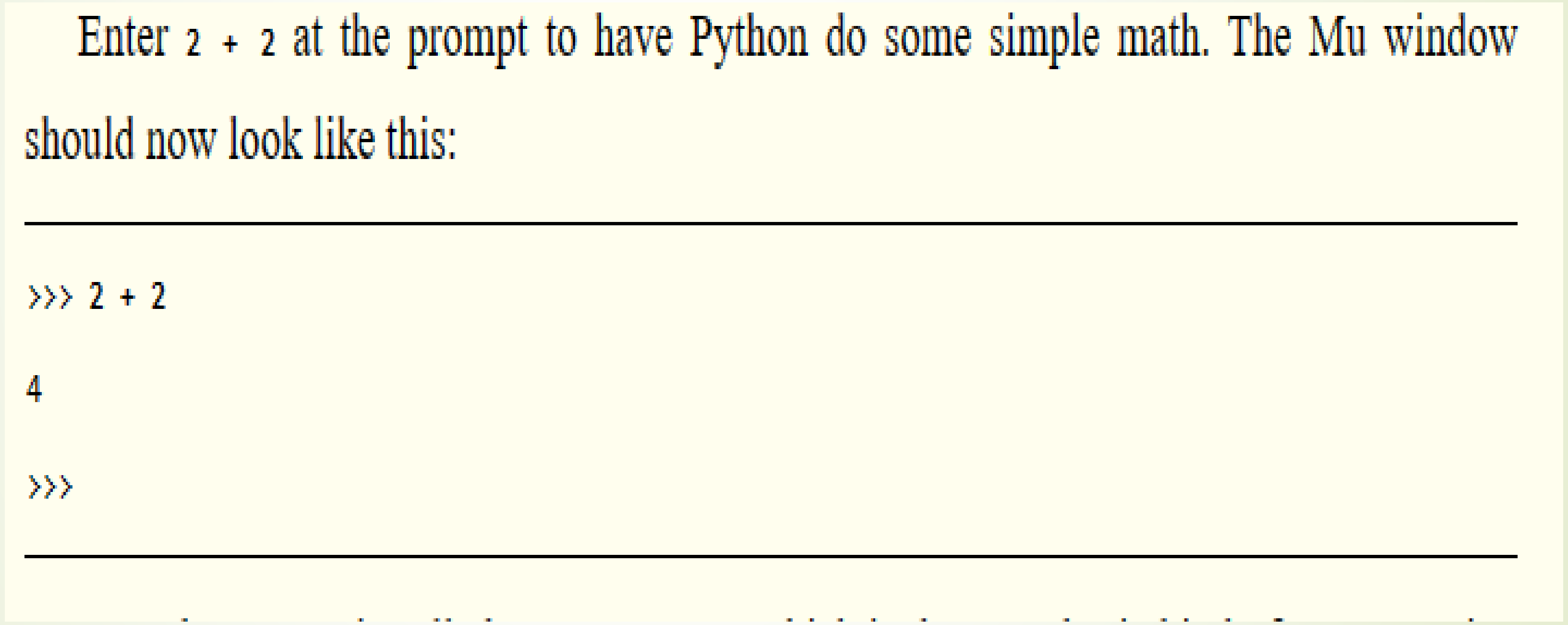
- PYTHON BASICS

Entering Expressions into the Interactive Shell



You can run the interactive shell by launching the Mu editor, which you should have downloaded when going through the setup instructions in the Preface. On Windows, open the Start menu, type "Mu," and open the Mu app. On macOS, open your Applications folder and double-click **Mu**. Click the **New** button and save an empty file as *blank.py*. When you run this blank file by clicking the **Run** button or pressing F5, it will open the interactive shell, which will open as a new pane that opens at the bottom of the Mu editor's window. You should see a >>> prompt in the interactive shell.

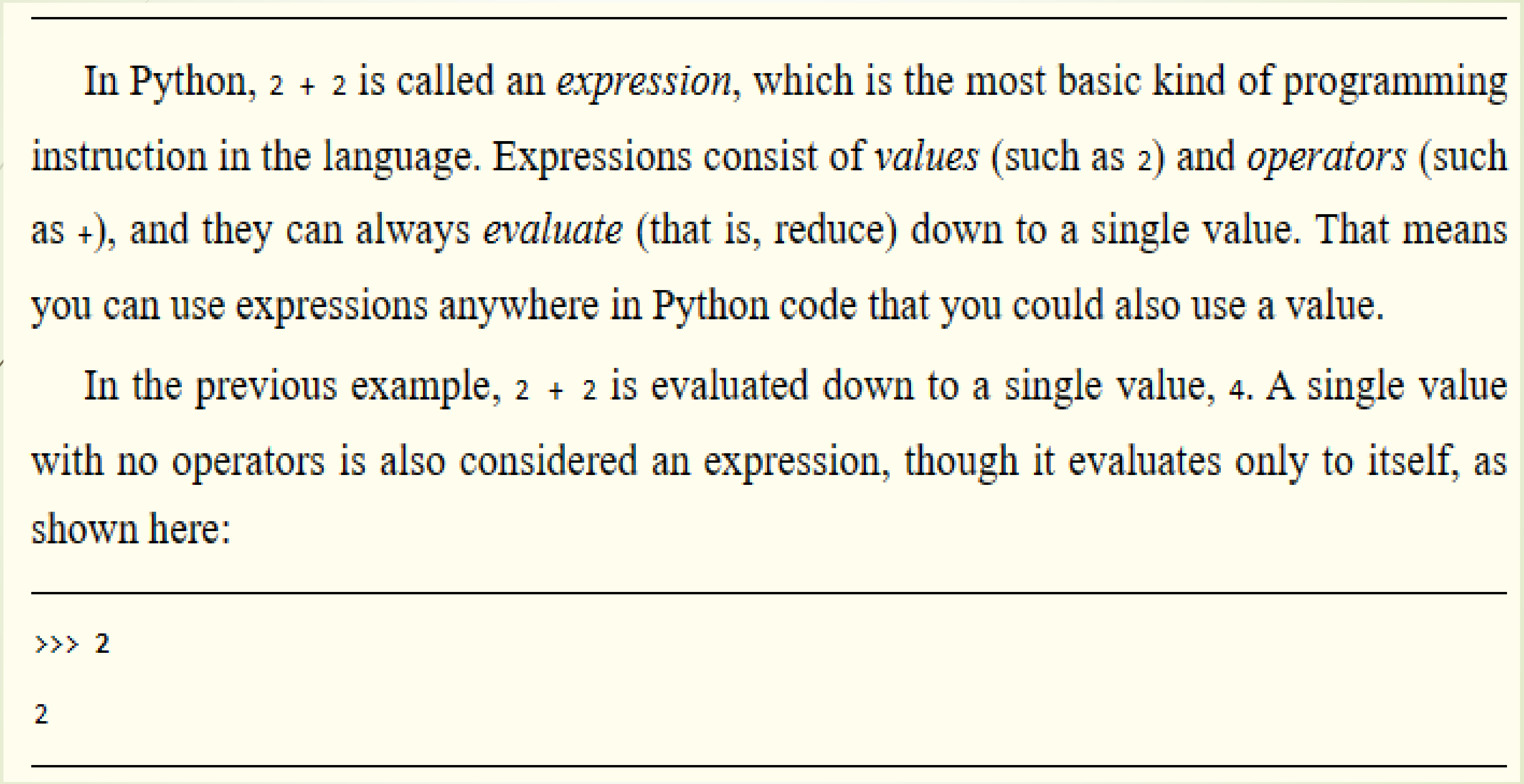
Cont…



Enter 2 + 2 at the prompt to have Python do some simple math. The Mu window should now look like this:

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

Cont..



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 you can use expressions anywhere in Python code that you could also use a value.

In the previous example, `2 + 2` is evaluated down to a single value, `4`. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

Math operators

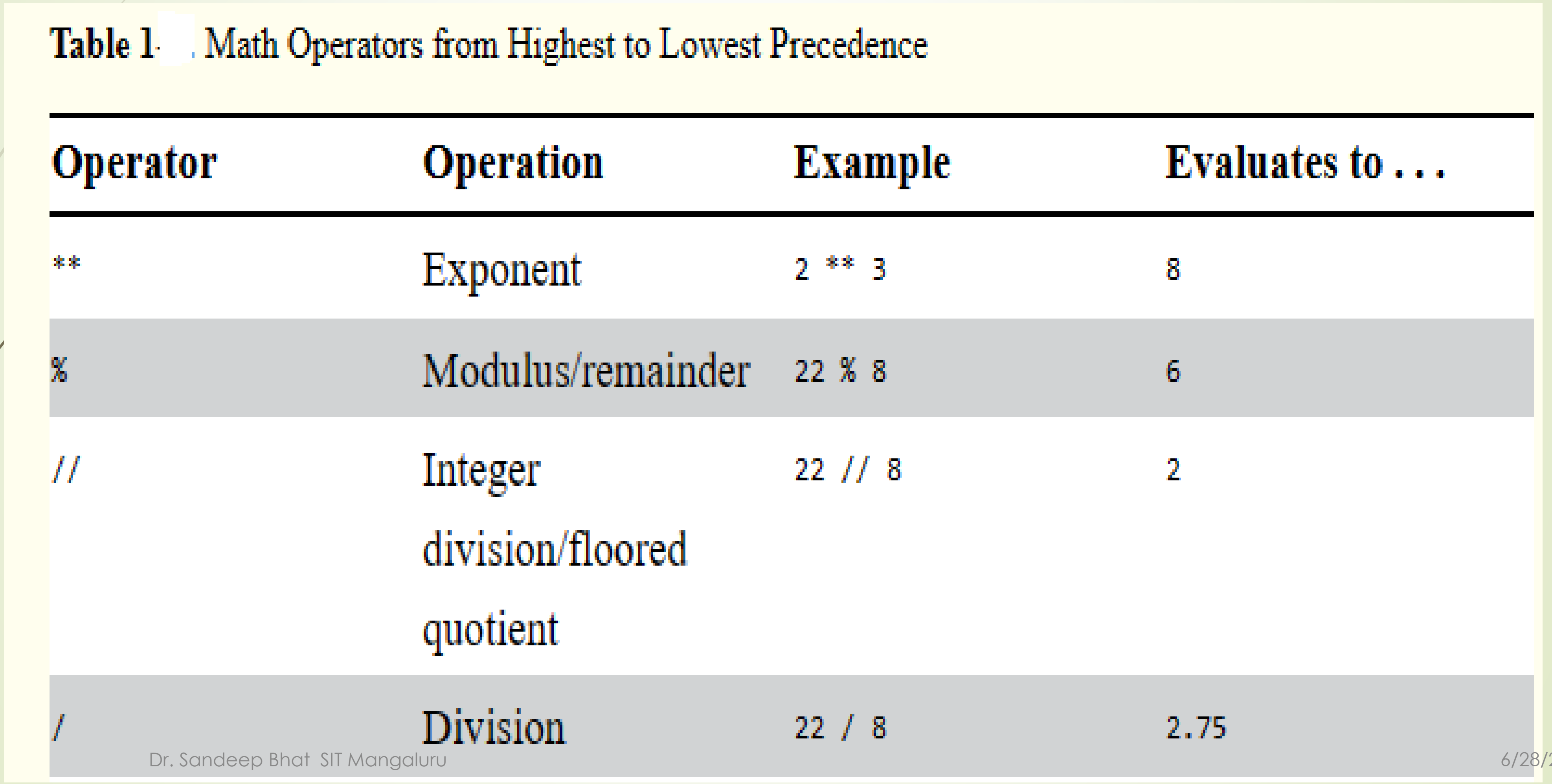
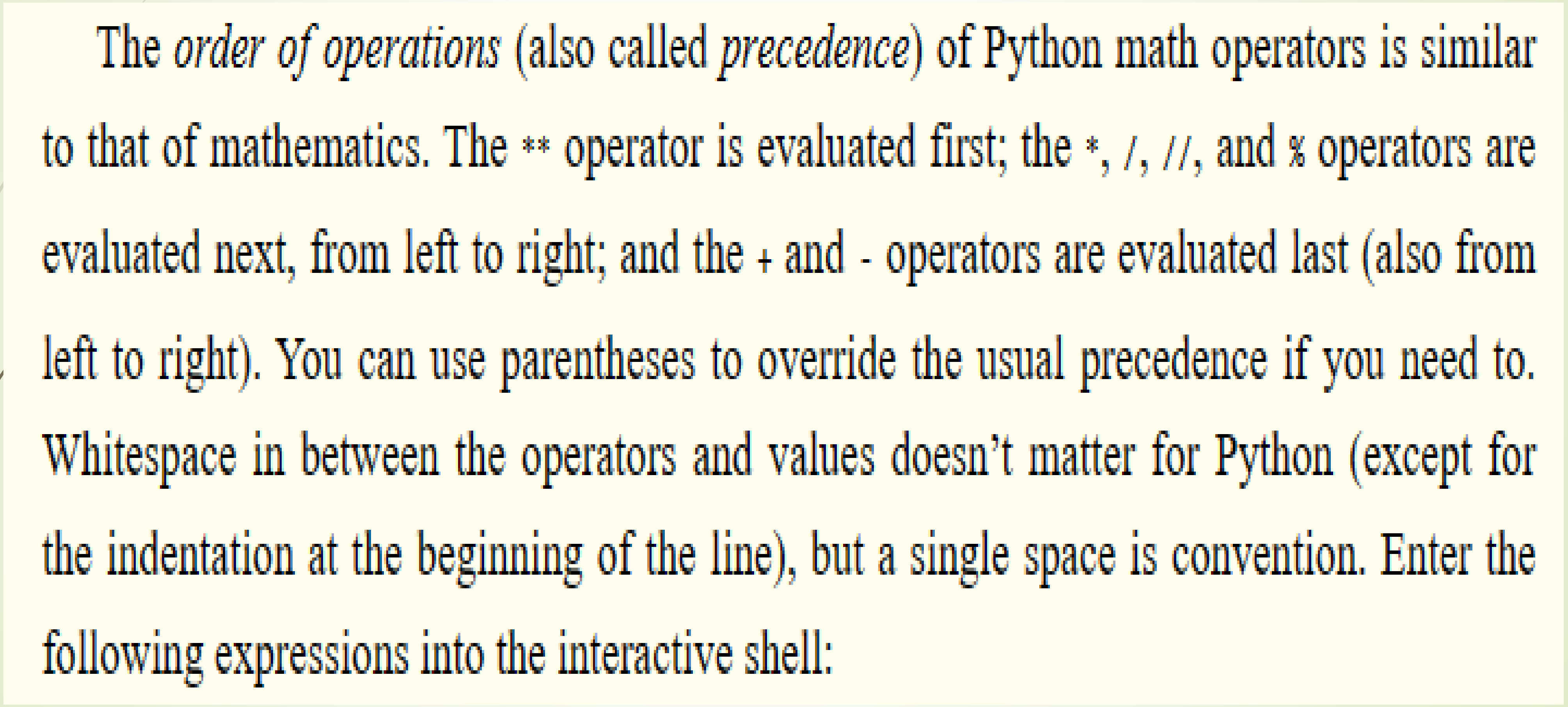


Table 1- . Math Operators from Highest to Lowest Precedence

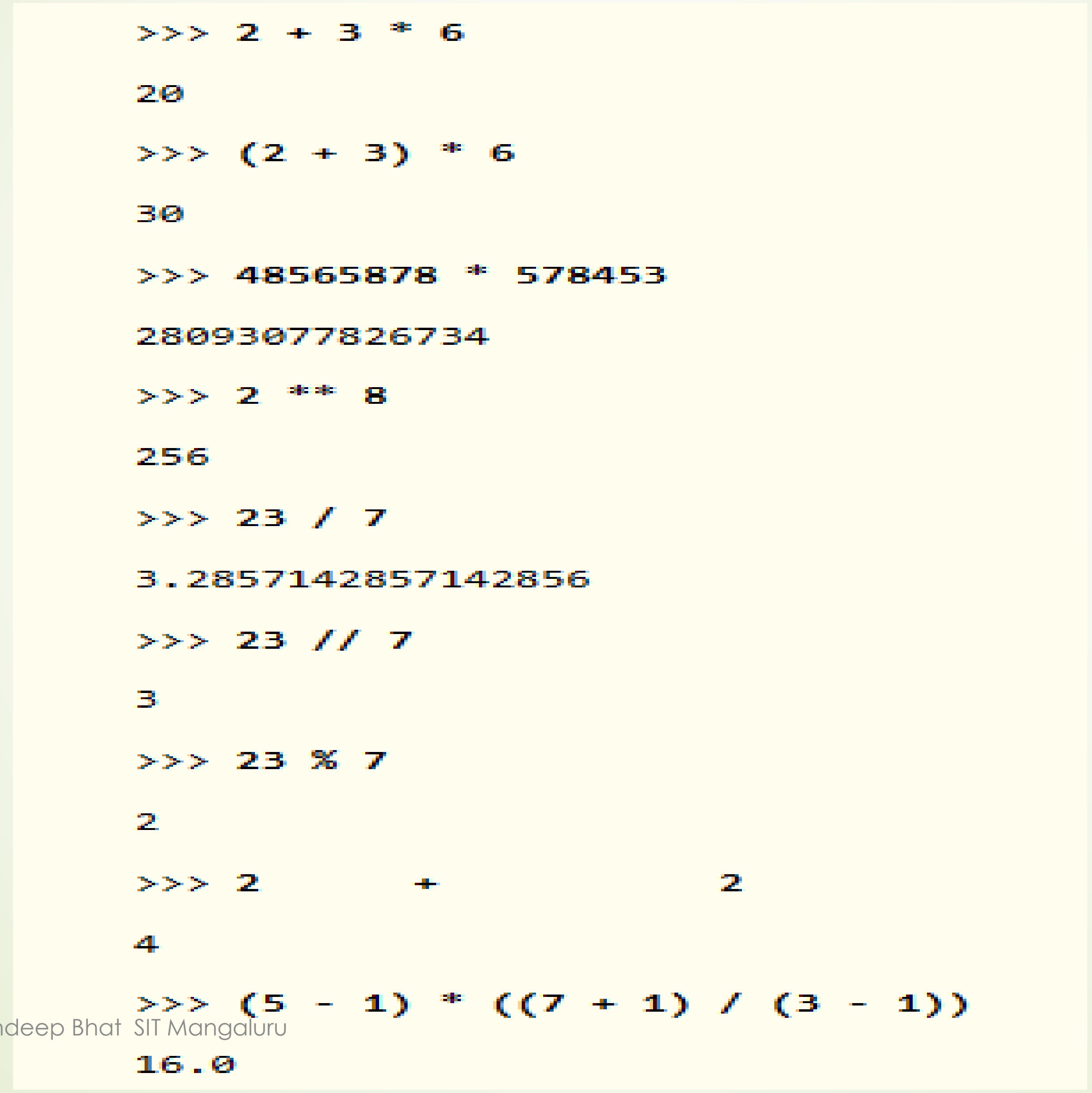
Operator	Operation	Example	Evaluates to . . .
`**`	Exponent	`2 ** 3`	`8`
`%`	Modulus/remainder	`22 % 8`	`6`
`//`	Integer division/floored quotient	`22 // 8`	`2`
`/`	Division	`22 / 8`	`2.75`

cont...



The *order of operations* (also called *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). You can use parentheses to override the usual precedence if you need to. Whitespace in between the operators and values doesn't matter for Python (except for the indentation at the beginning of the line), but a single space is convention. Enter the following expressions into the interactive shell:

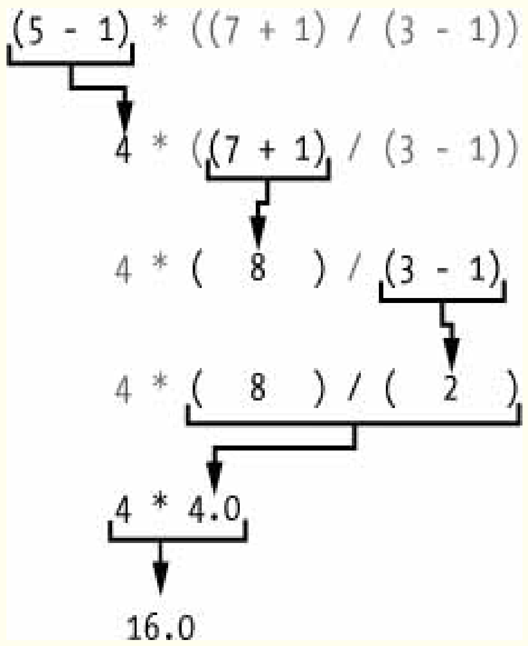
Cont...



```
>>> 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
```

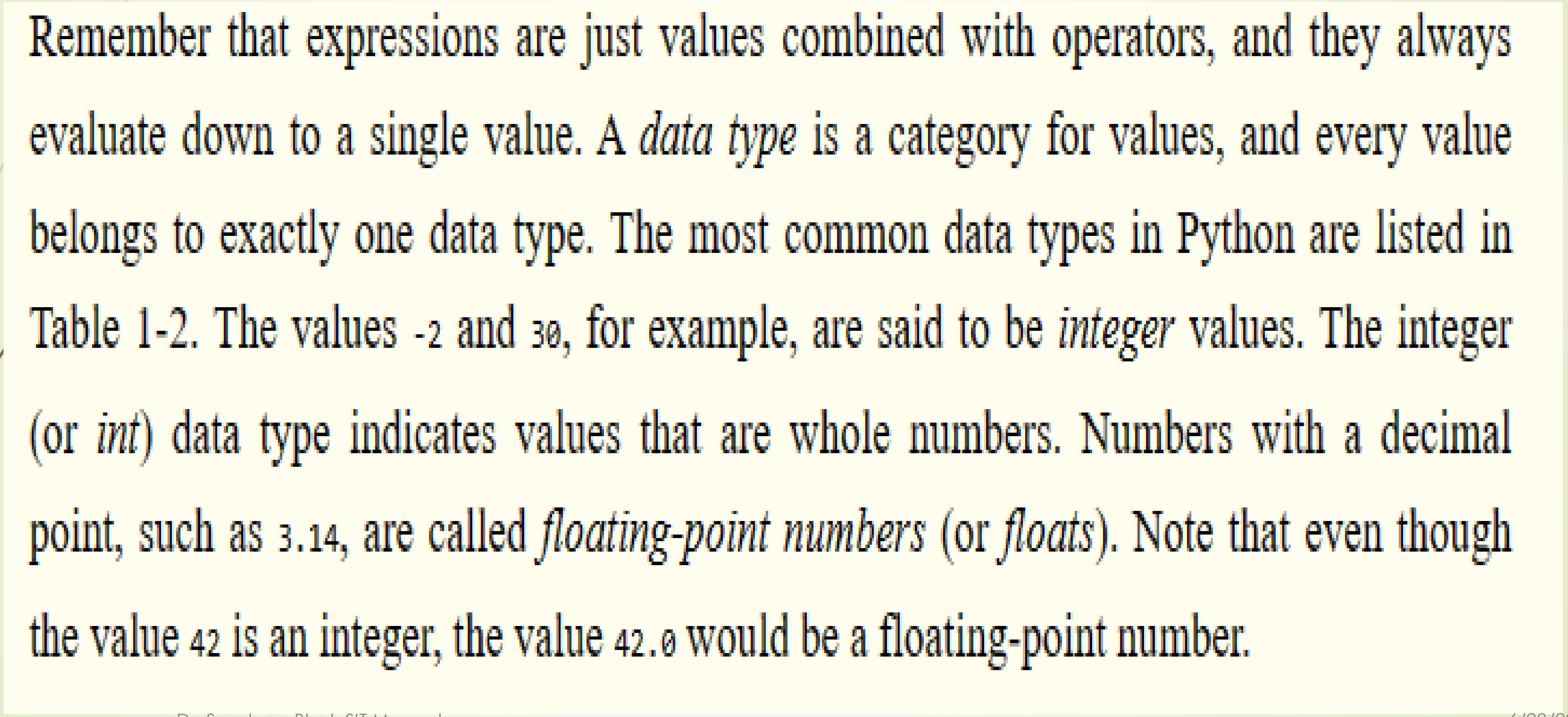
Cont..

In each case, you as the programmer must enter the expression, but Python does the hard part of evaluating it down to a single value. Python will keep evaluating parts of the expression until it becomes a single value, as shown here:



These rules for putting operators and values together to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate. Here's an example:

The Integer, Floating-Point, and String Data Types



Remember that expressions are just values combined with operators, and they always evaluate down to a single value. A *data type* is a category for values, and every value belongs to exactly one data type. The most common data types in Python are listed in Table 1-2. The values `-2` and `30`, for example, are said to be *integer* values. The integer (or *int*) data type indicates values that are whole numbers. Numbers with a decimal point, such as `3.14`, are called *floating-point numbers* (or *floats*). Note that even though the value `42` is an integer, the value `42.0` would be a floating-point number.

Cont...

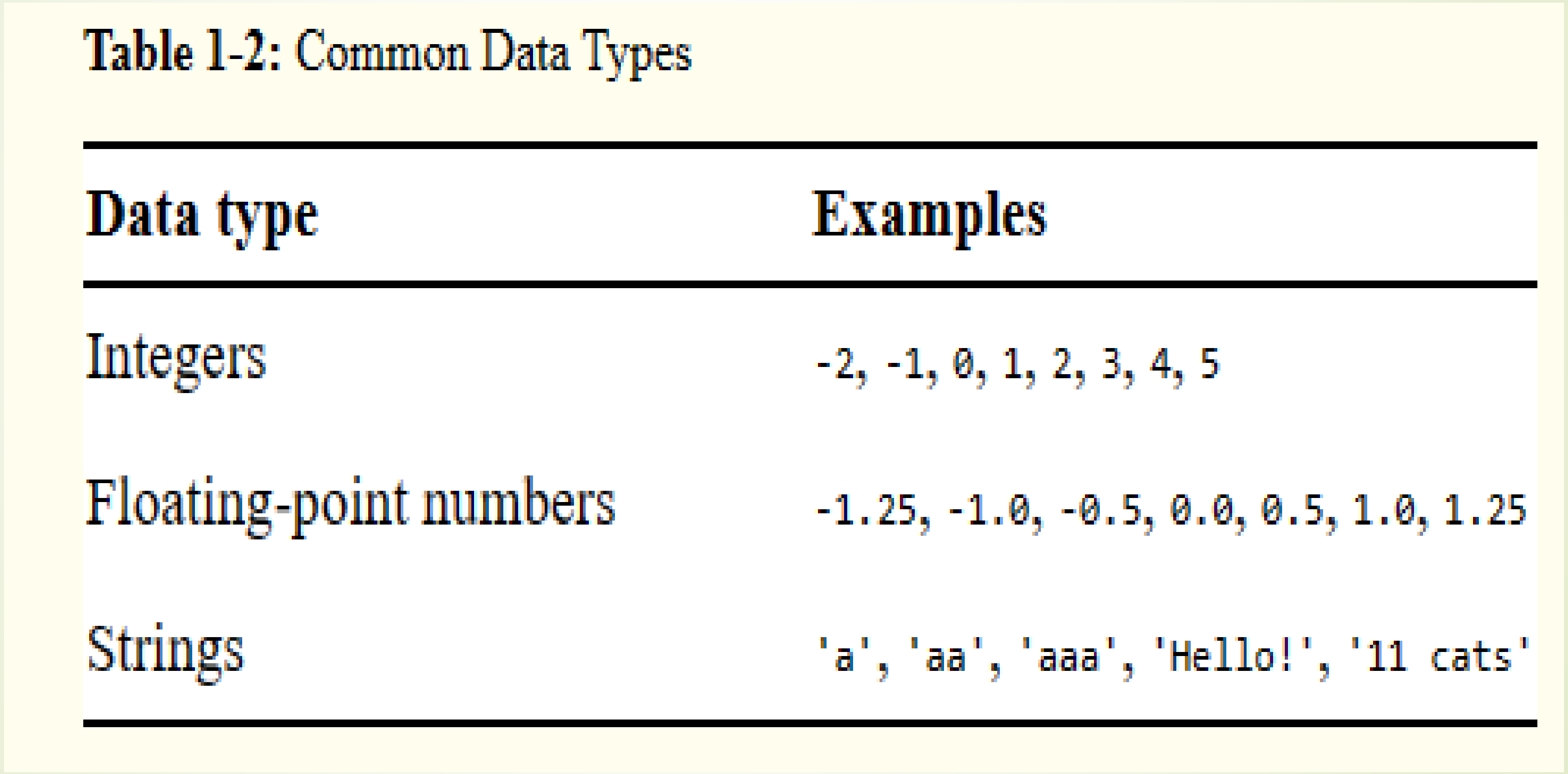
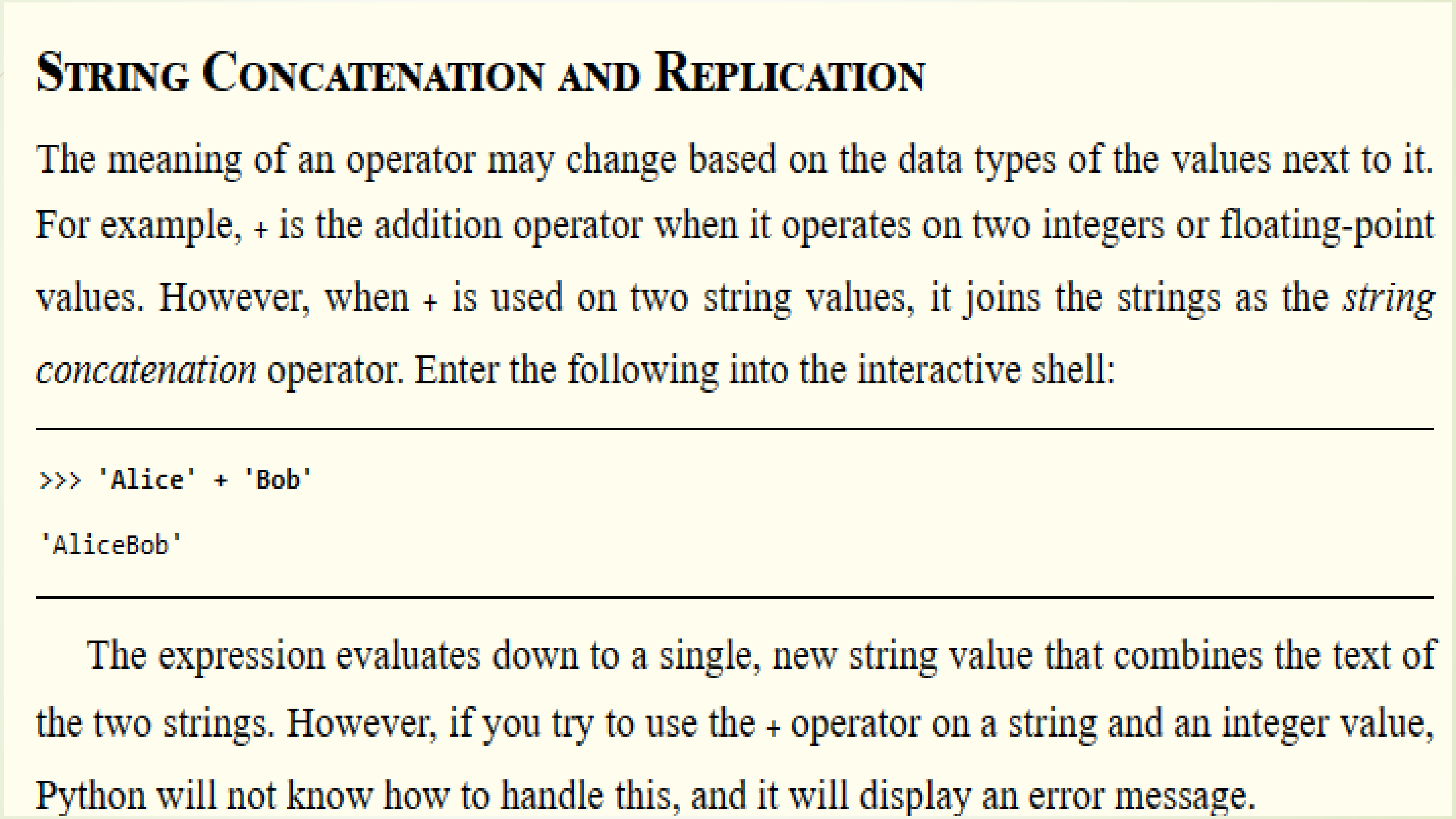


Table 1-2: Common Data Types

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'`

Cont…



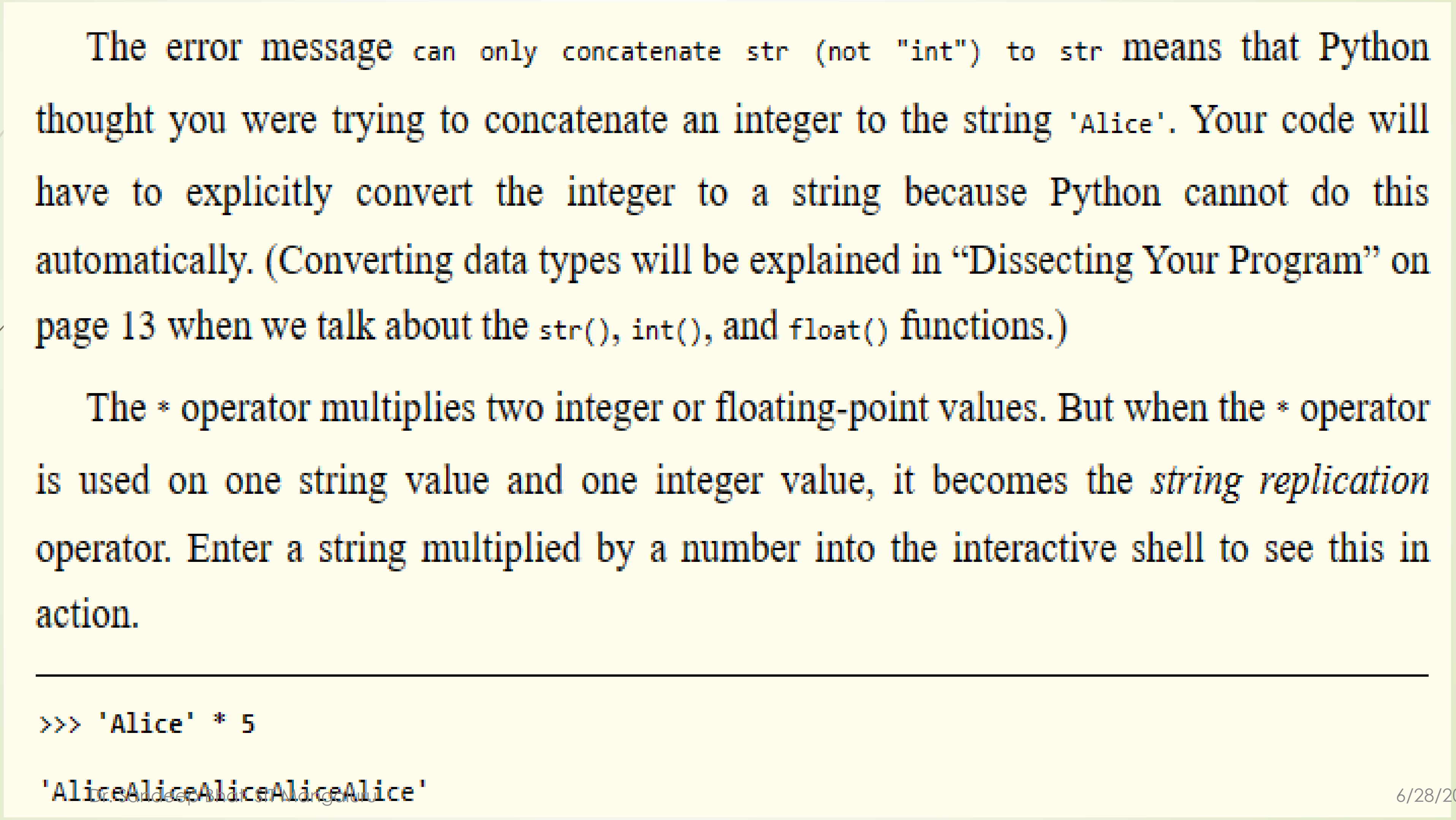
String Concatenation and Replication

The meaning of an operator may change based on the data types of the values next to it. 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. Enter the following into the interactive shell:

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

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you 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.

Cont..

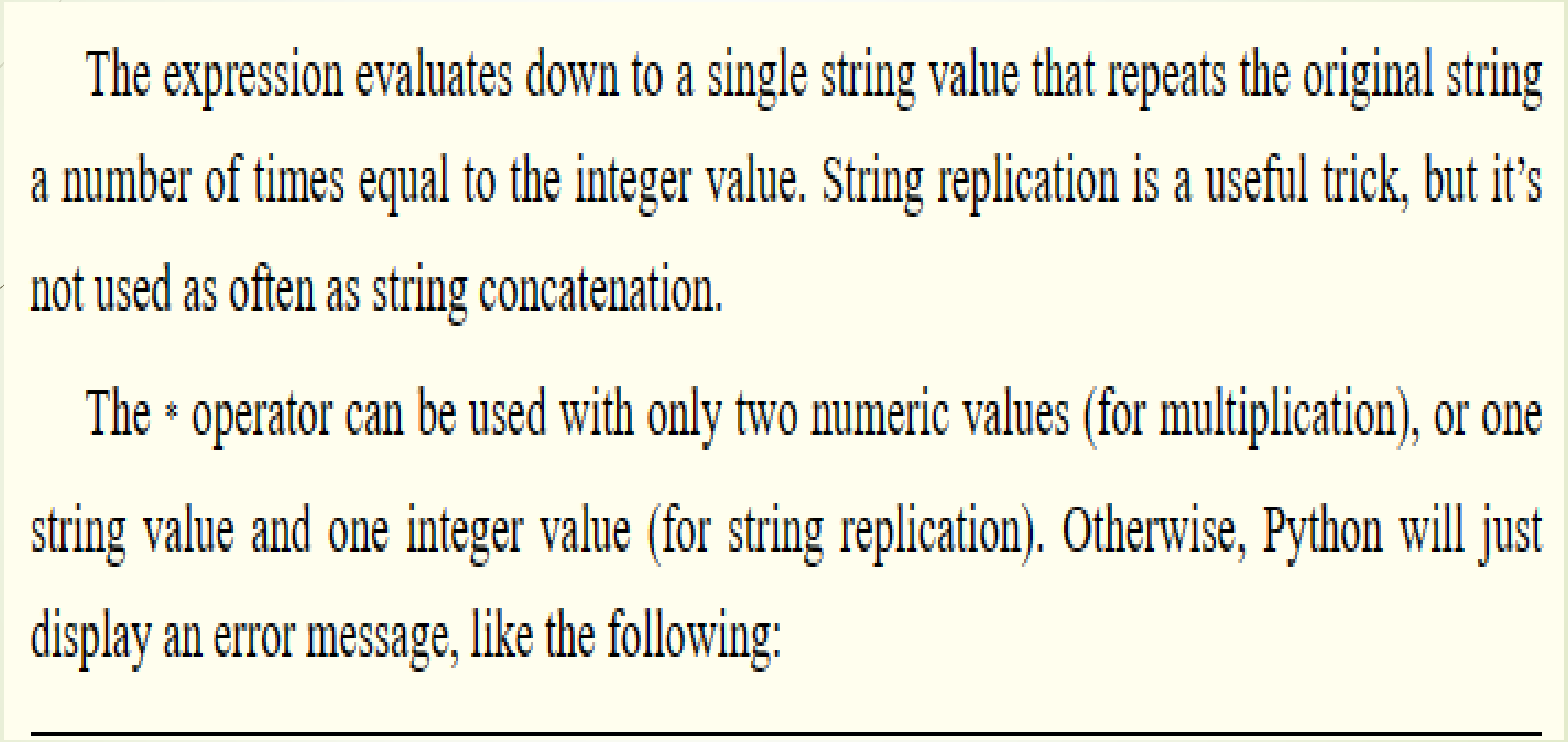


The error message `can only concatenate str (not "int") to str` means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the integer to a string because Python cannot do this automatically. (Converting data types will be explained in "Dissecting Your Program" on page 13 when we talk about the `str()`, `int()`, and `float()` functions.)

The `*` operator multiplies two integer or floating-point values. But when the `*` operator is used on one string value and one integer value, it becomes the *string replication* operator. Enter a string multiplied by a number into the interactive shell to see this in action.

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

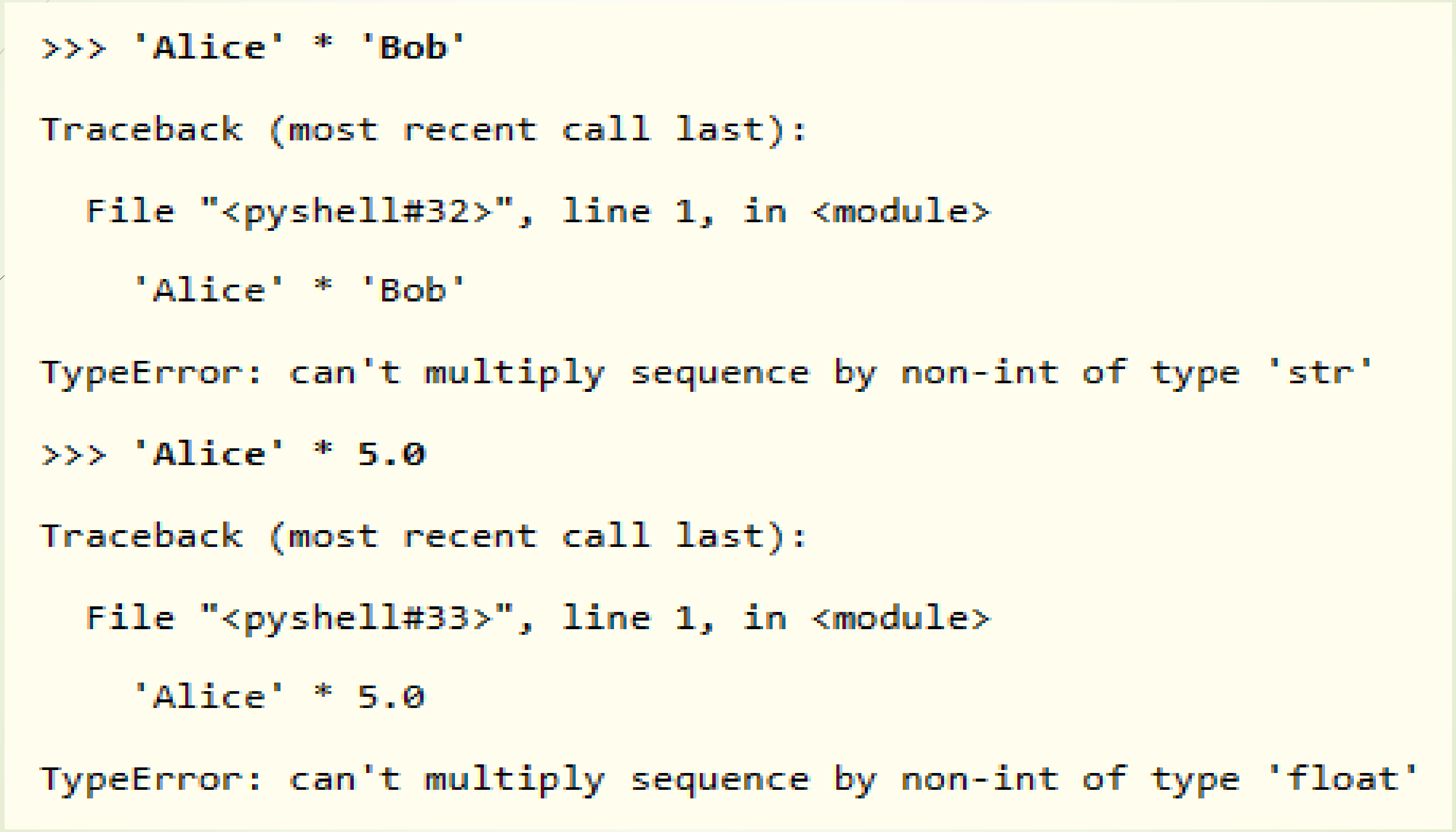
Cont..



The expression evaluates down to a single string value that repeats the original string a number of times equal to the integer value. String replication is a useful trick, but it's not used as often as string concatenation.

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, like the following:

Cont...



```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Cont…

- It makes sense that Python wouldn’t understand these expressions: you can’t multiply two words, and it’s hard to replicate an arbitrary string a fractional number of times.

Storing Values in Variables

- A variable is like a box in the computer's memory where you can store a single value.
- If you want to use the result of an evaluated expression later in your program, you can save it inside a variable.

Assignment Statements

- You'll store values in variables with an assignment statement.
- An assignment statement consists of a variable name, an equal sign (called the assignment operator), and the value to be stored. If you enter the assignment statement spam = 42, then a variable named spam will have the integer value 42 stored in it.
- Think of a variable as a labeled box that a value is placed in, as in Figure 1-1.

Cont…

Think of a variable as a labeled box that a value is placed in, as in Figure 1-1.

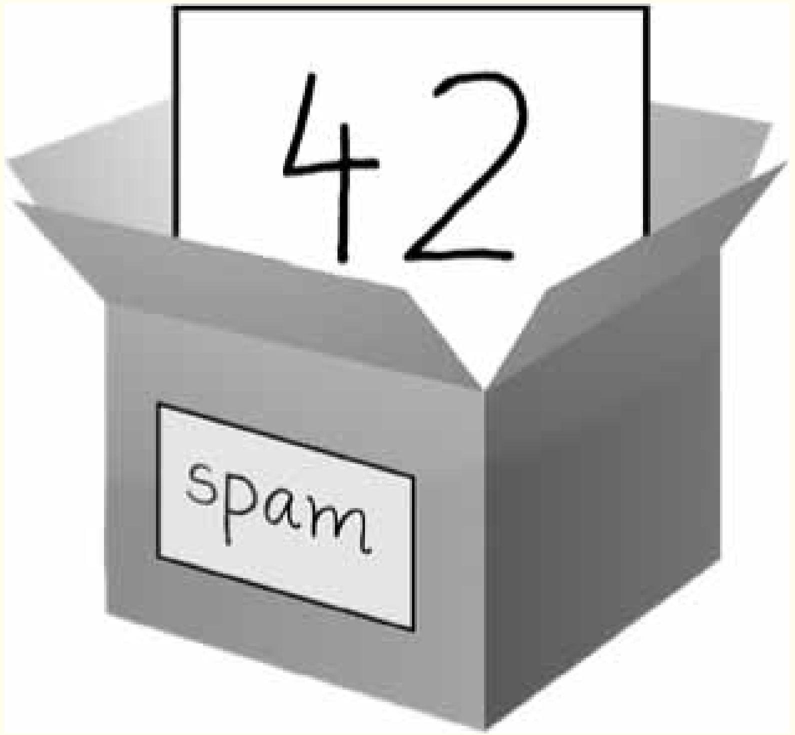
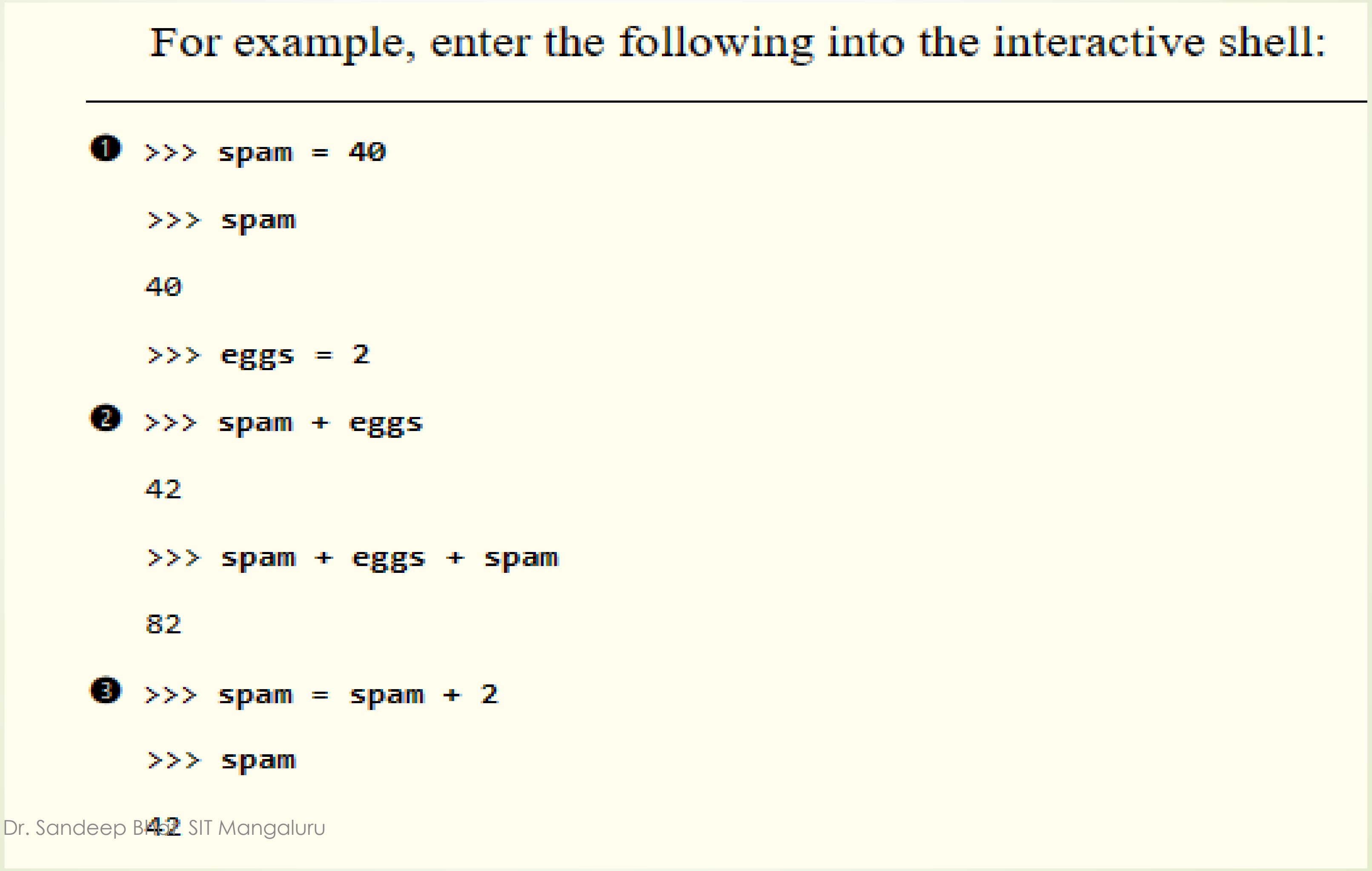


Figure 1-1: `spam = 42` *is like telling the program, "The variable* `spam` *now has the integer value* `42` *in it."*

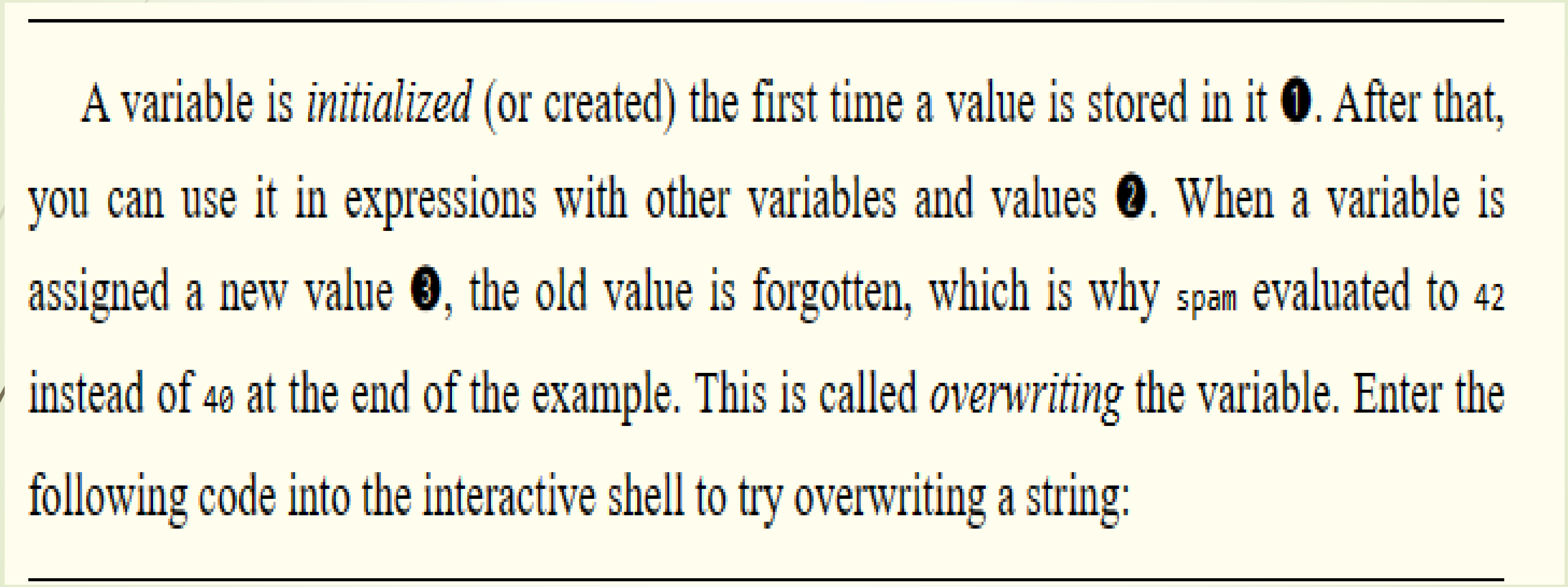
Cont...



For example, enter the following into the interactive shell:

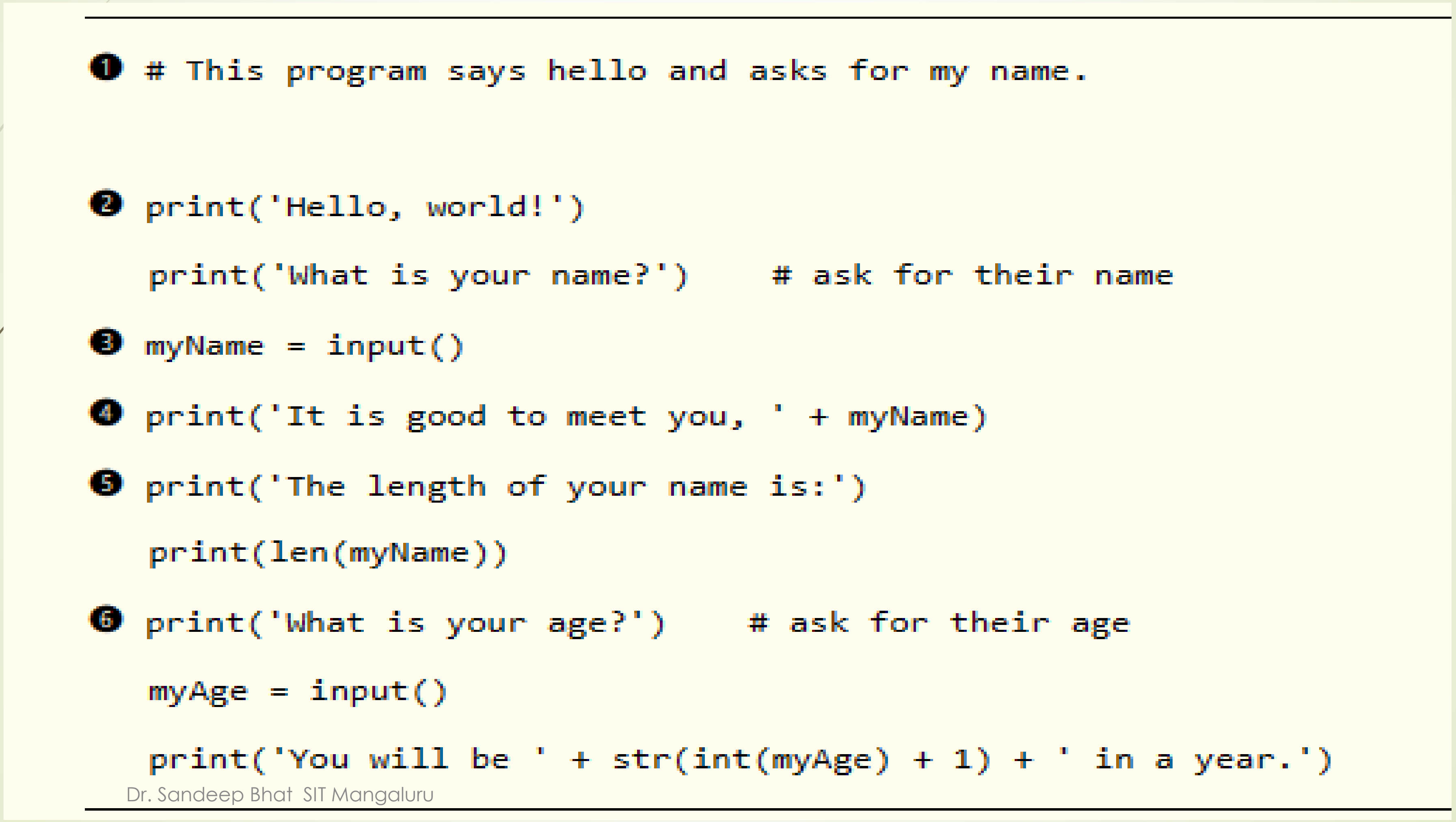
```
❶ >>> spam = 40
   >>> spam
   40
   >>> eggs = 2
❷ >>> spam + eggs
   42
   >>> spam + eggs + spam
   82
❸ >>> spam = spam + 2
   >>> spam
   42
```

Cont..



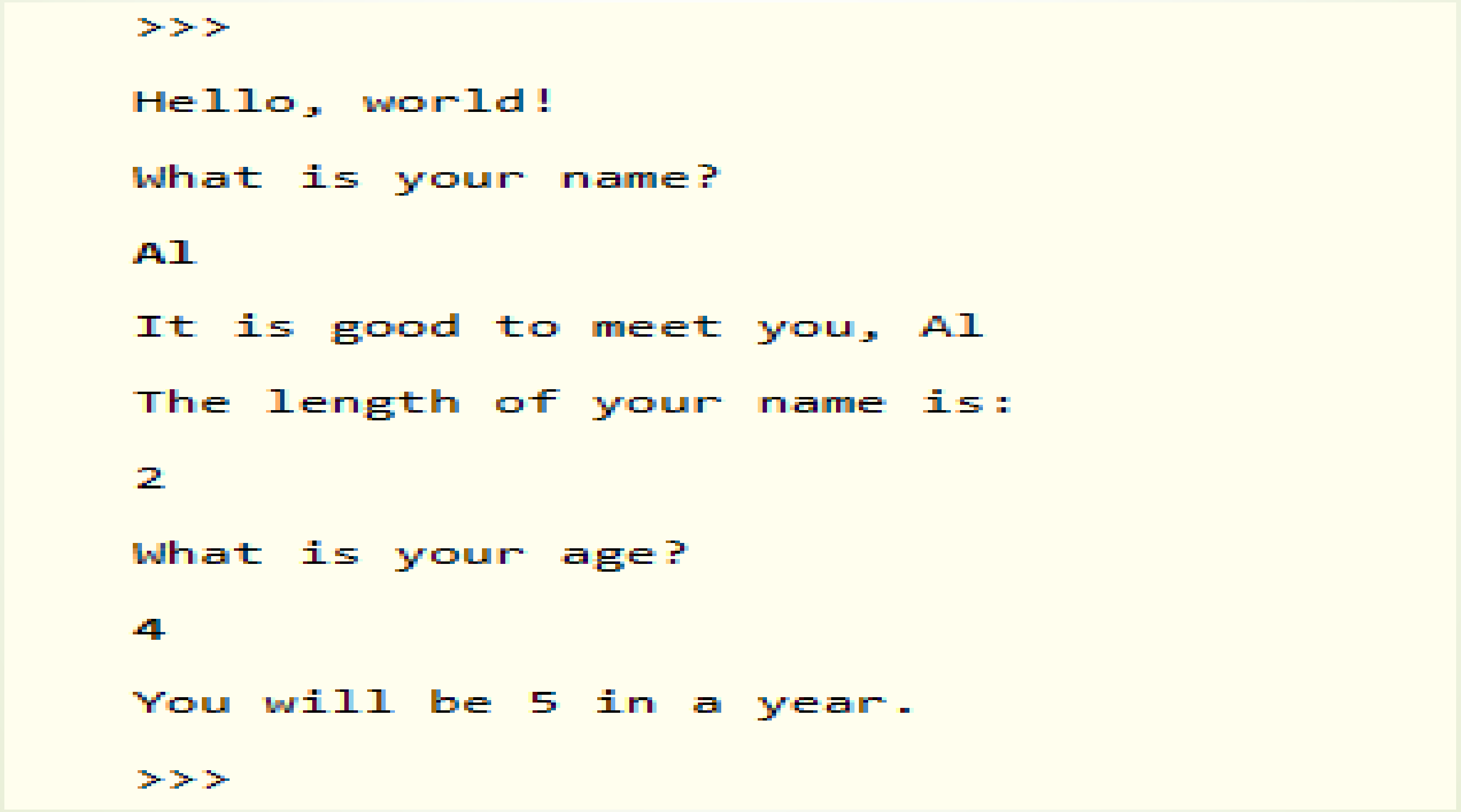
A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is why `spam` evaluated to `42` instead of `40` at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

Your first programme



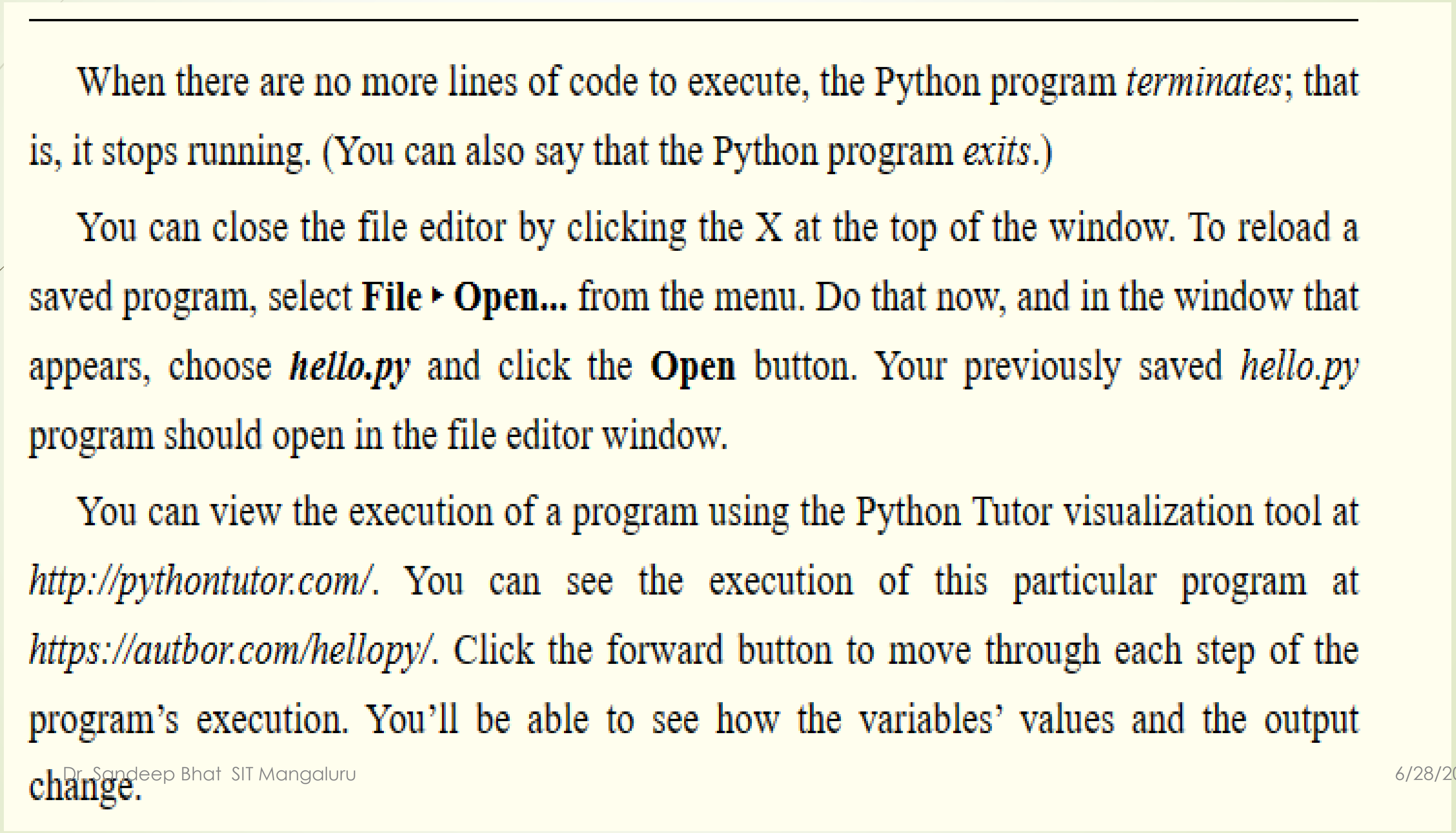
```
❶ # This program says hello and asks for my name.

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.')
```

output



```
>>>
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.
>>>
```

Cont..

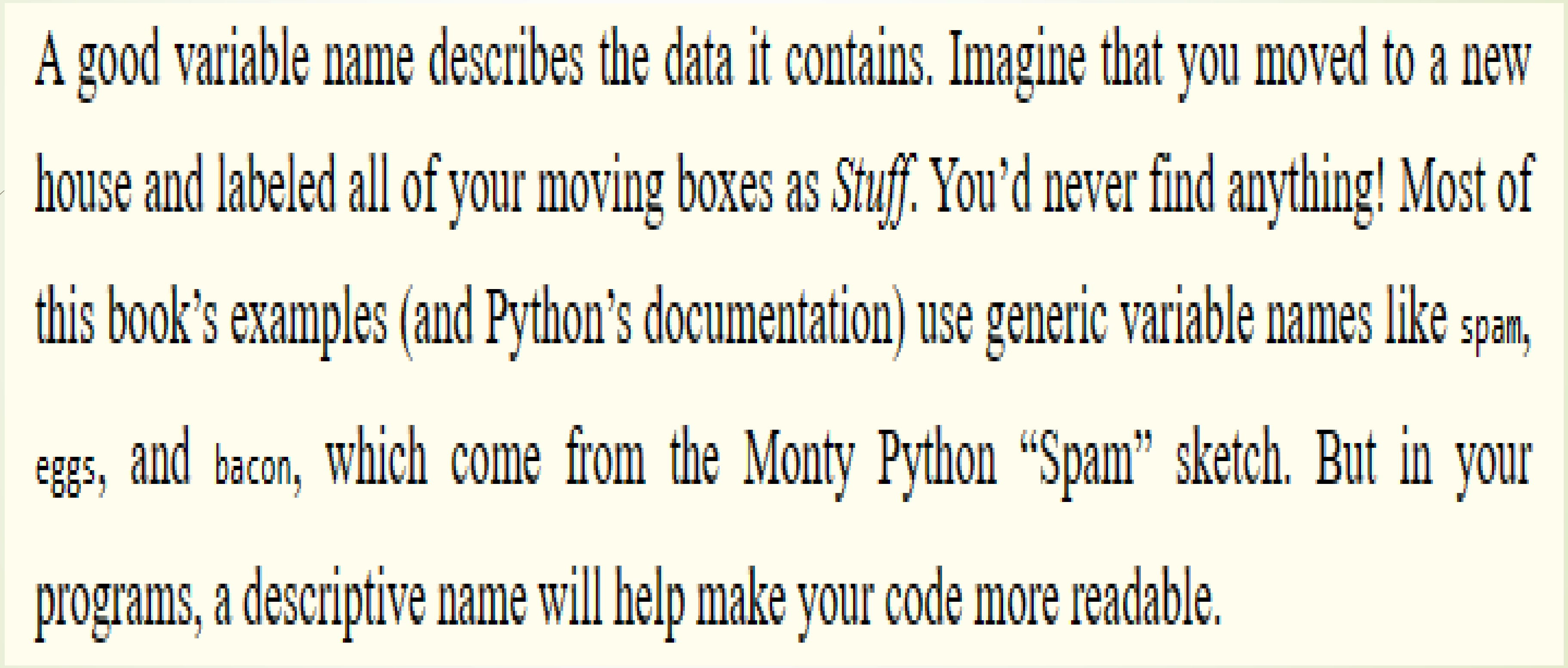


When there are no more lines of code to execute, the Python program *terminates*; that is, it stops running. (You can also say that the Python program *exits*.)

You can close the file editor by clicking the X at the top of the window. To reload a saved program, select **File ▸ Open...** from the menu. Do that now, and in the window that appears, choose ***hello.py*** and click the **Open** button. Your previously saved *hello.py* program should open in the file editor window.

You can view the execution of a program using the Python Tutor visualization tool at *http://pythontutor.com/*. You can see the execution of this particular program at *https://autbor.com/hellopy/*. Click the forward button to move through each step of the program's execution. You'll be able to see how the variables' values and the output change.

Variable Names



A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as *Stuff*. You'd never find anything! Most of this book's examples (and Python's documentation) use generic variable names like `spam`, `eggs`, and `bacon`, which come from the Monty Python "Spam" sketch. But in your programs, a descriptive name will help make your code more readable.

Cont..

- Though you can name your variables almost anything, Python does have some naming restrictions.

- You can name a variable anything as long as it obeys the following three rules:

- It can be only one word with no spaces.
- It can use only letters, numbers, and the underscore (_) character.
- It can’t begin with a number.

Cont..

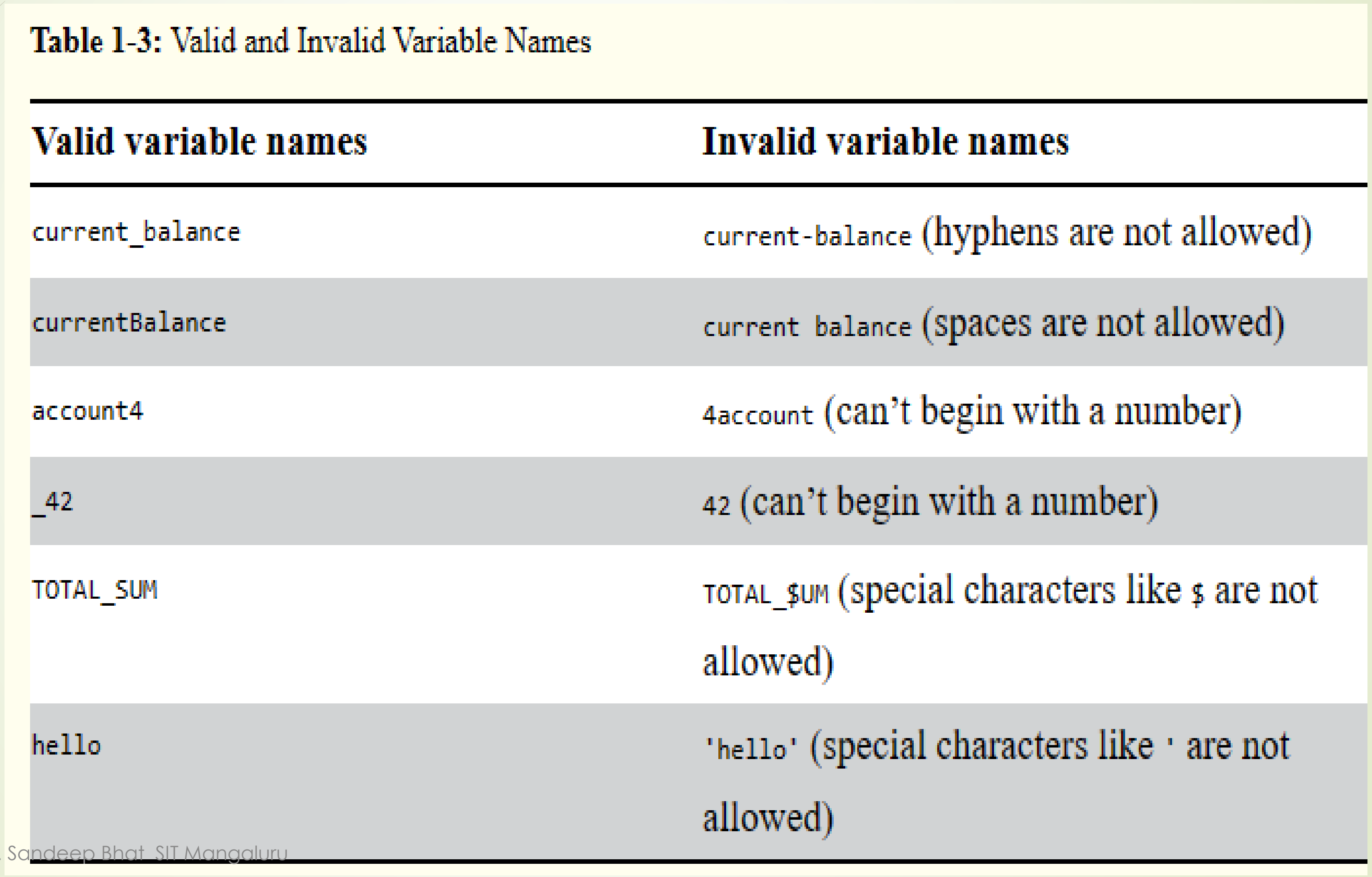


Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
`current_balance`	`current-balance` (hyphens are not allowed)
`currentBalance`	`current balance` (spaces are not allowed)
`account4`	`4account` (can't begin with a number)
`_42`	`42` (can't begin with a number)
`TOTAL_SUM`	`TOTAL_$UM` (special characters like `$` are not allowed)
`hello`	`'hello'` (special characters like `'` are not allowed)

Your first programme

- 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)

Cont…

- 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.')

results

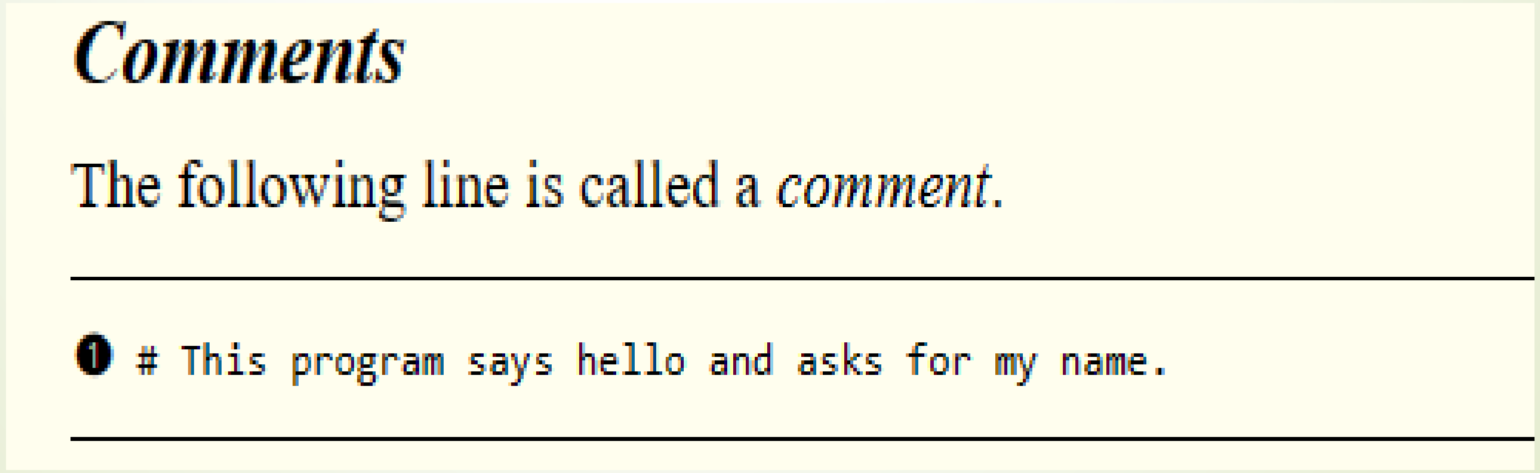
```
>>>
Hello, world!
What is your name?
Al
It is good to meet you, Al
```

```
The length of your name is:
2
What is your age?
4
You will be 5 in a year.
>>>
```

Dissecting Your Program

- With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.



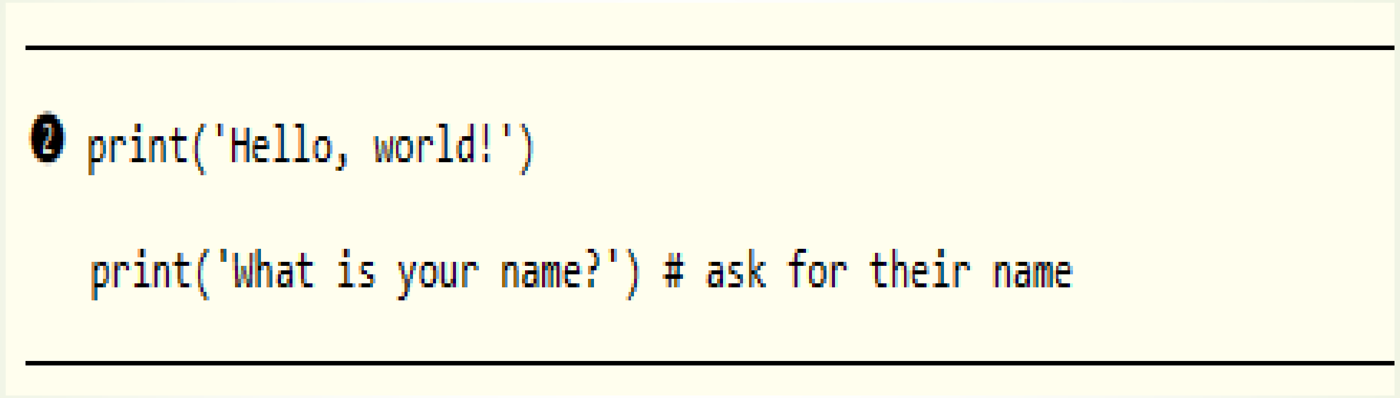
Comments

The following line is called a *comment*.

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

The print() Function

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



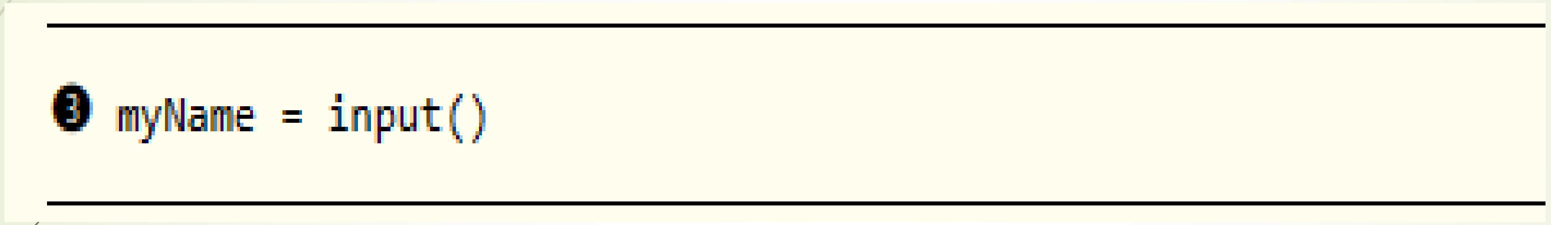
```
❷ print('Hello, world!')
  print('What is your name?') # ask for their name
```

- 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 input() Function

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

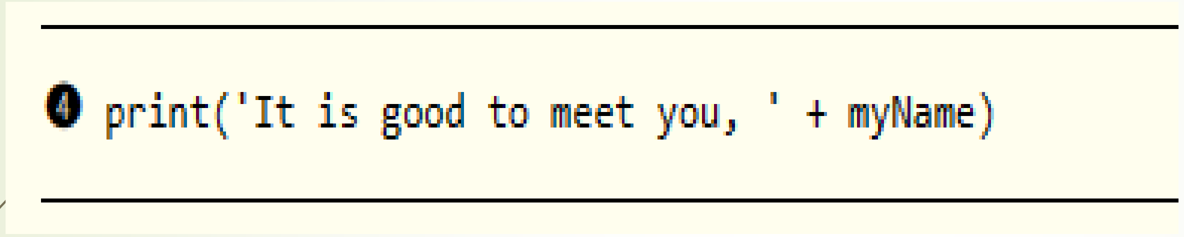


```
❸ myName = input()
```

- This function call evaluates to a string equal to the user's text, and the line of code assigns the myName variable to this string value.
- You can think of the input() function call as an expression that evaluates to whatever string the user typed in. If the user entered 'Al', then the expression would evaluate to myName = 'Al'.

Printing the User’s Name

- The following call to print() actually contains the expression 'It is good to meet you, ' + myName between the parentheses.

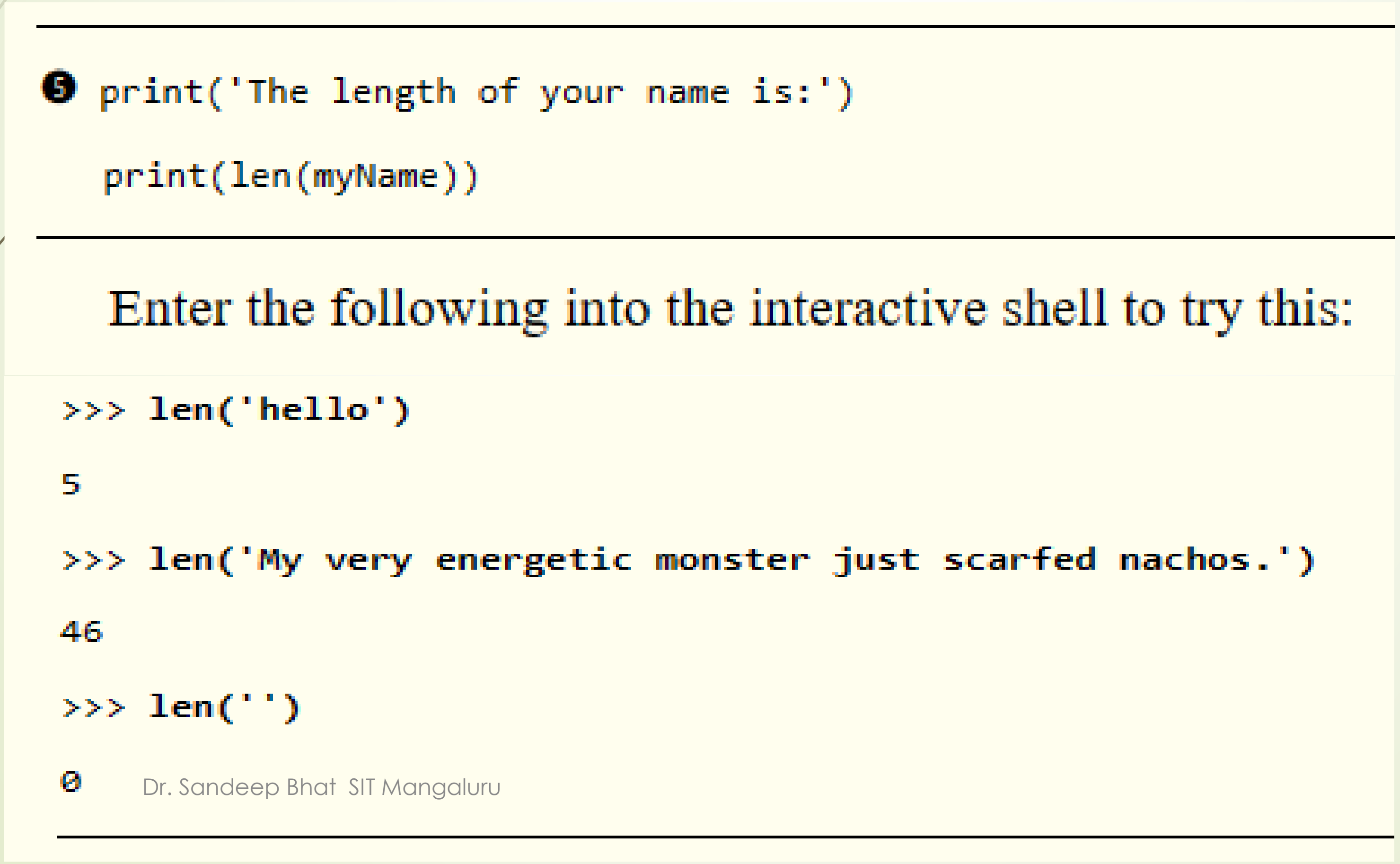


```
❹ print('It is good to meet you, ' + myName)
```

- Remember that expressions can always evaluate to a single value. If 'Al' is the value stored in myName on line ❸, then this expression evaluates to 'It is good to meet you, Al'.
- This single string value is then passed to print(), which prints it on the screen.

The len() Function

- You can pass the len() function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.



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

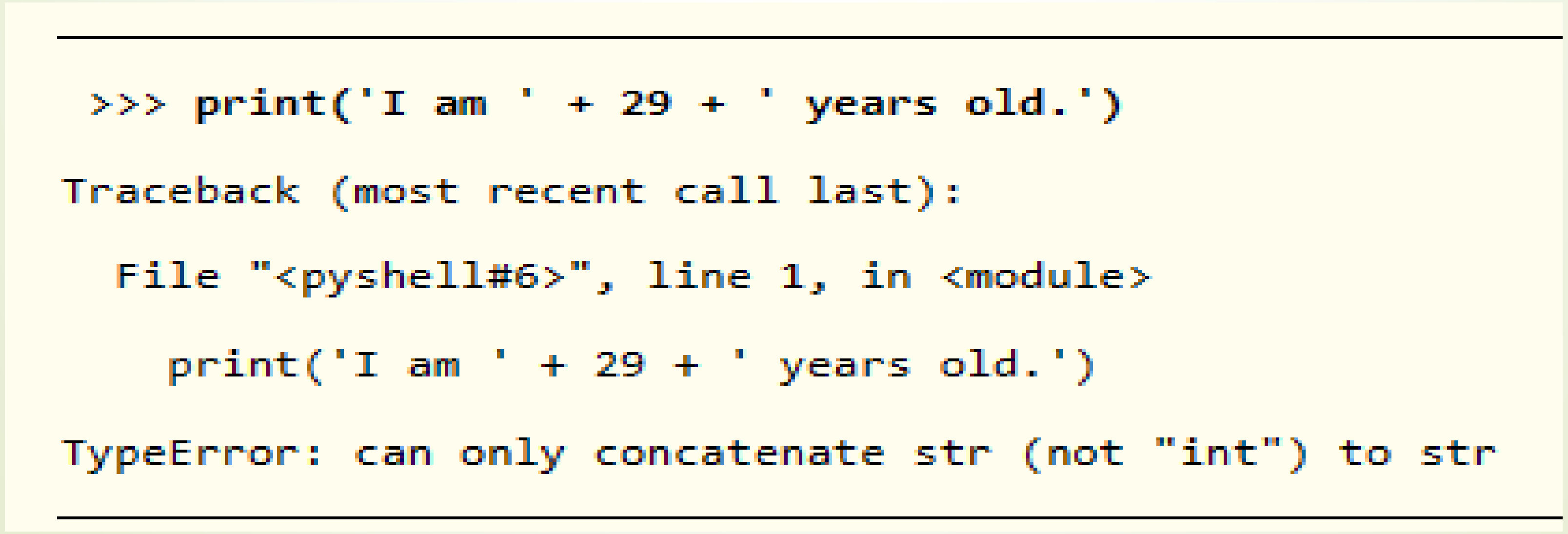
Enter the following into the interactive shell to try this:

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

Cont…

- Just like those examples, len(myName) evaluates to an integer. It is then passed to print() to be displayed on the screen.

- The print() function allows you to pass it either integer values or string values, but notice the error that shows up when you type the following into the interactive shell:



```
 >>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: can only concatenate str (not "int") to str
```

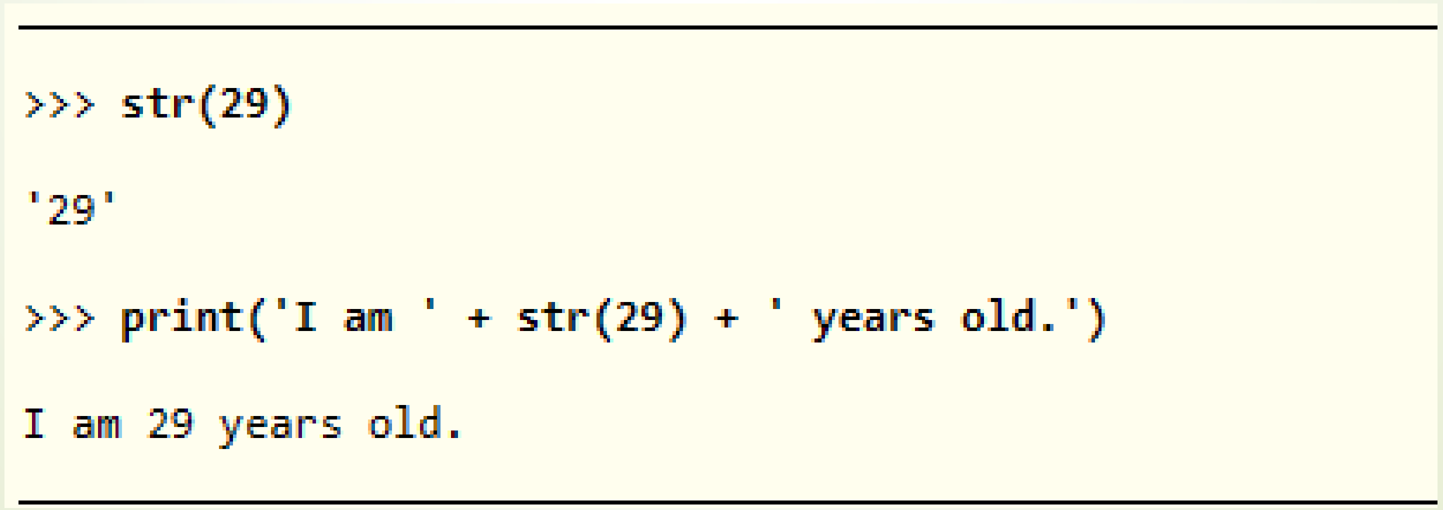
Cont..

- Python gives an error because the + operator can only be used to add two integers together or concatenate two strings.
- You can't add an integer to a string, because this is ungrammatical in Python.

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

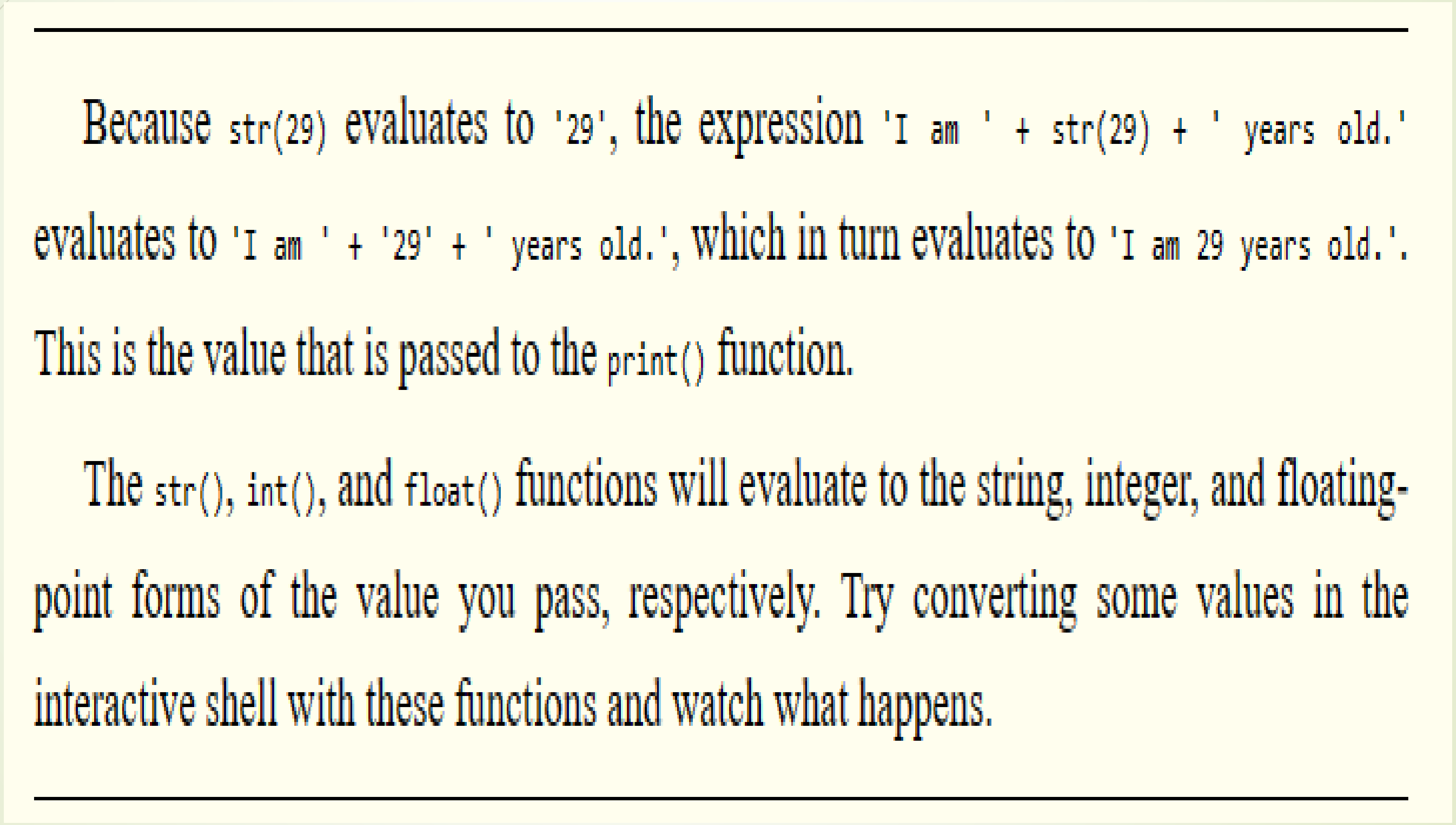
- If you want to concatenate an integer such as 29 with a string to pass to print(), you'll need to get the value '29', which is the string form of 29.

- The str() function can be passed an integer value and will evaluate to a string value version of the integer, as follows:



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

Cont..



Because `str(29)` evaluates to `'29'`, the expression `'I am ' + str(29) + ' years old.'` evaluates to `'I am ' + '29' + ' years old.'`, which in turn evaluates to `'I am 29 years old.'`. This is the value that is passed to the `print()` function.

The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively. Try converting some values in the interactive shell with these functions and watch what happens.

Practice Questions

1. Which of the following are operators, and which are values?

*

'hello'

-88.8

-

/

+

5

Cont..

- 3. Name three data types.
- 4. What is an expression made up of? What do all expressions do?
- 5. This chapter introduced assignment statements, like `spam = 10`. What is the difference between an expression and a statement?

ANS:

- **Q.4&5:**
- Expression: An expression is a combination of values, variables, operators, and function calls that evaluates to a single value.

- Examples of expressions:

- 5 + 3
- x * y
- Math.sqrt(16)

Cont..

- A statement is a complete instruction or action that performs a specific task or operation.

Examples of statements:

- Assignment statement: x = 5;
- Conditional statement (if-else):

```
if x > 0:
    print("Positive")
else:
    print("Negative")
```

FLOW CONTROL

- 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.

Flow Chart

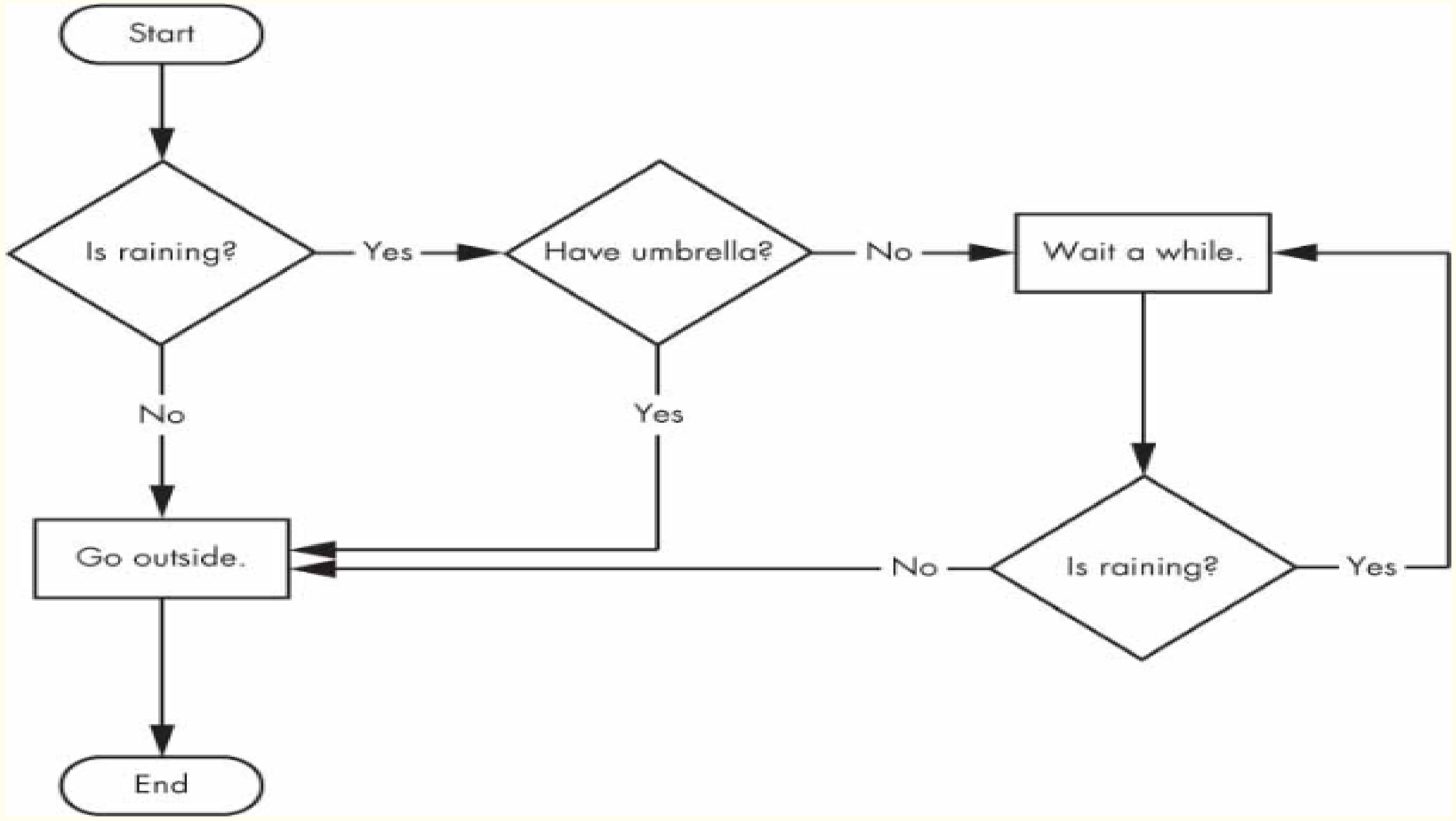


Figure 2-1: A flowchart to tell you what to do if it is raining

Cont..

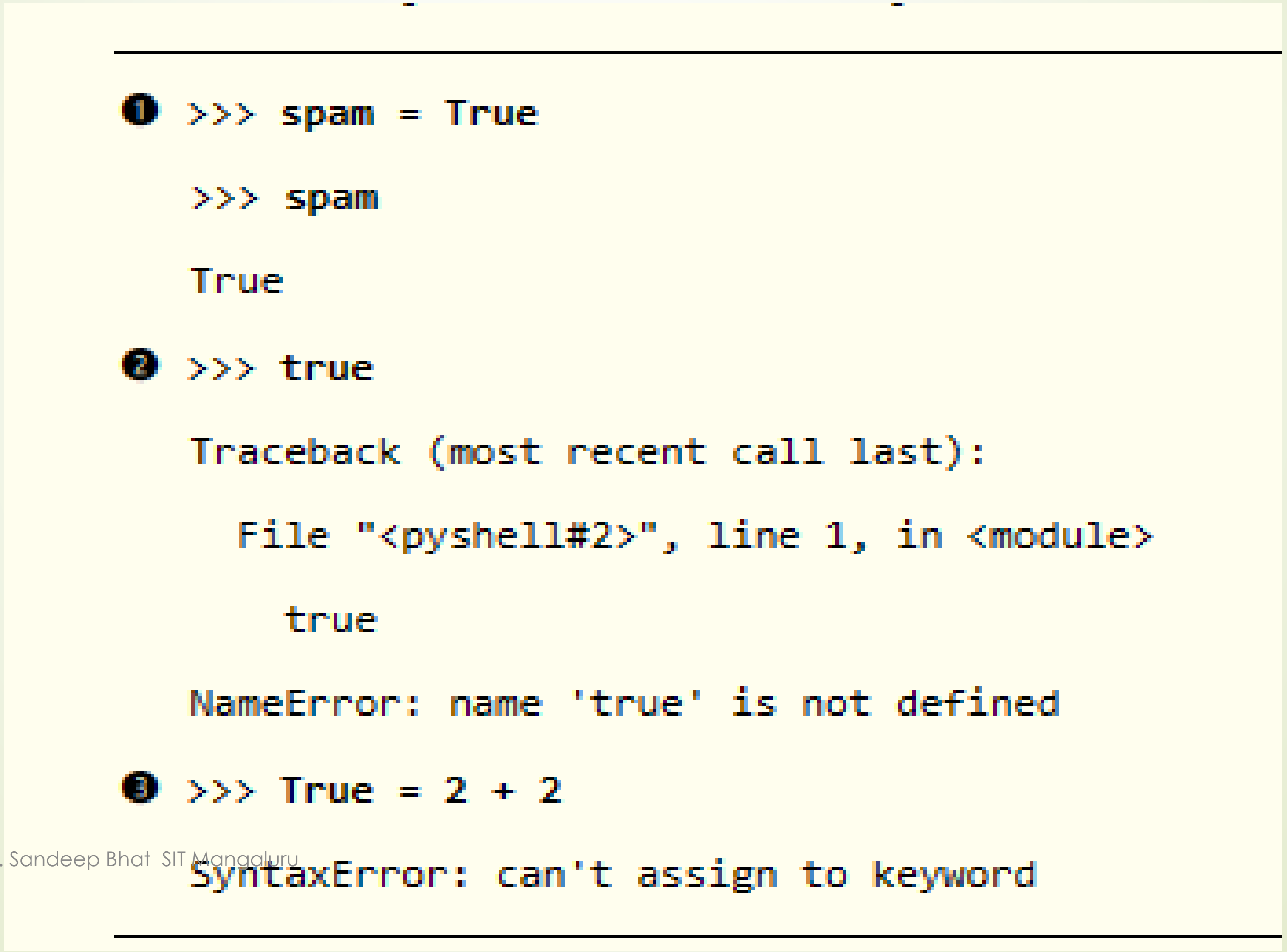
- In a flowchart, there is usually more than one way to go from the start to the end.
- The same is true for lines of code in a computer program.
- Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles.
- The starting and ending steps are represented with rounded rectangles.

Boolean Values

- While 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. (Boolean is capitalized because the data type is named after mathematician George Boole.)

- When entered as Python code, the Boolean values True and False lack the quotes you place around strings, and they always start with a capital T or F, with the rest of the word in lowercase.

Cont..



```
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: can't assign to keyword
```

Comparison Operators

- Comparison operators, also called relational operators, compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

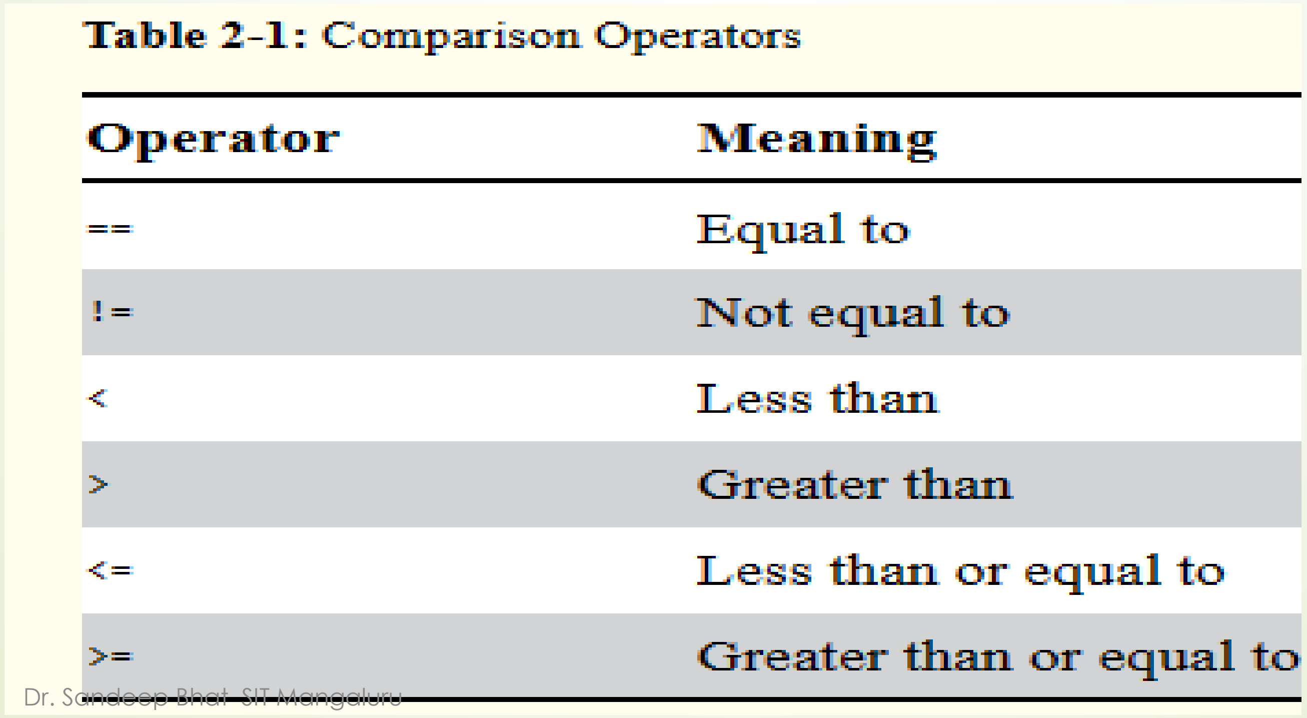
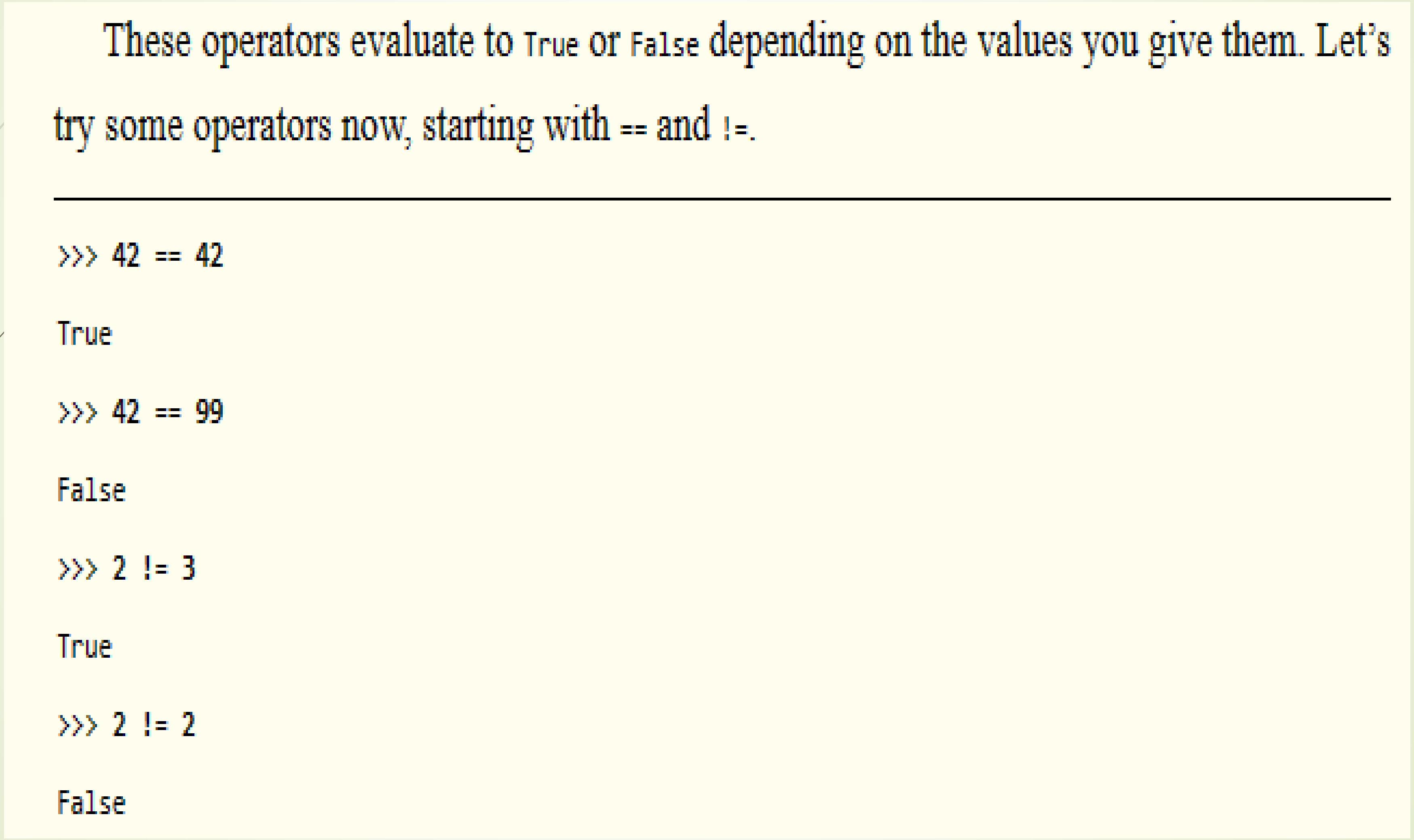


Table 2-1: Comparison Operators

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

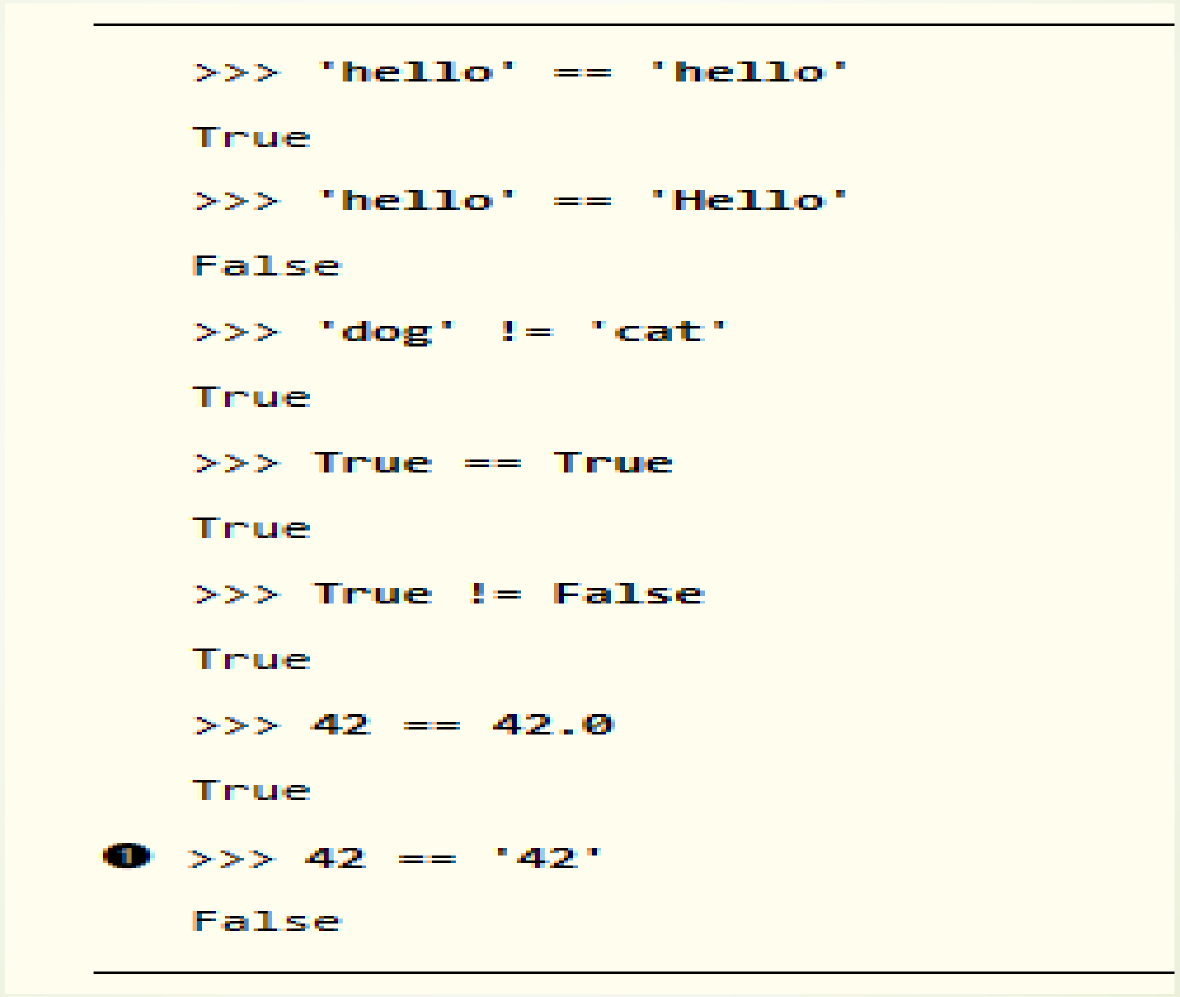
Cont..



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

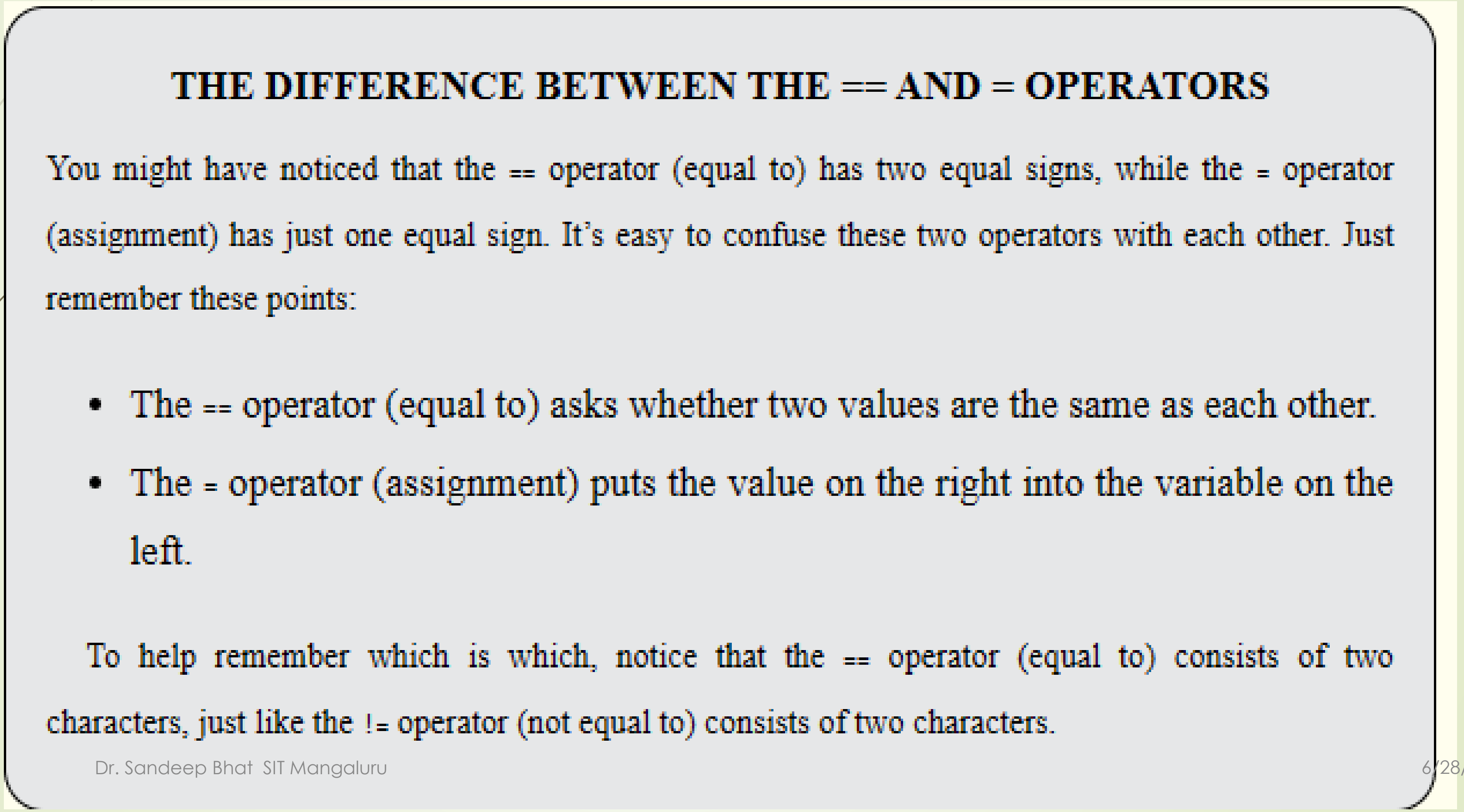
```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

Cont..



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

Cont..



THE DIFFERENCE BETWEEN THE == AND = OPERATORS

You might have noticed that the == operator (equal to) has two equal signs, while the = operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- 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.

To help remember which is which, notice that the == operator (equal to) consists of two characters, just like the != operator (not equal to) consists of two characters.

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. Let's explore these operators in detail, starting with the and operator.

Binary Boolean Operators

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

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

Cont..

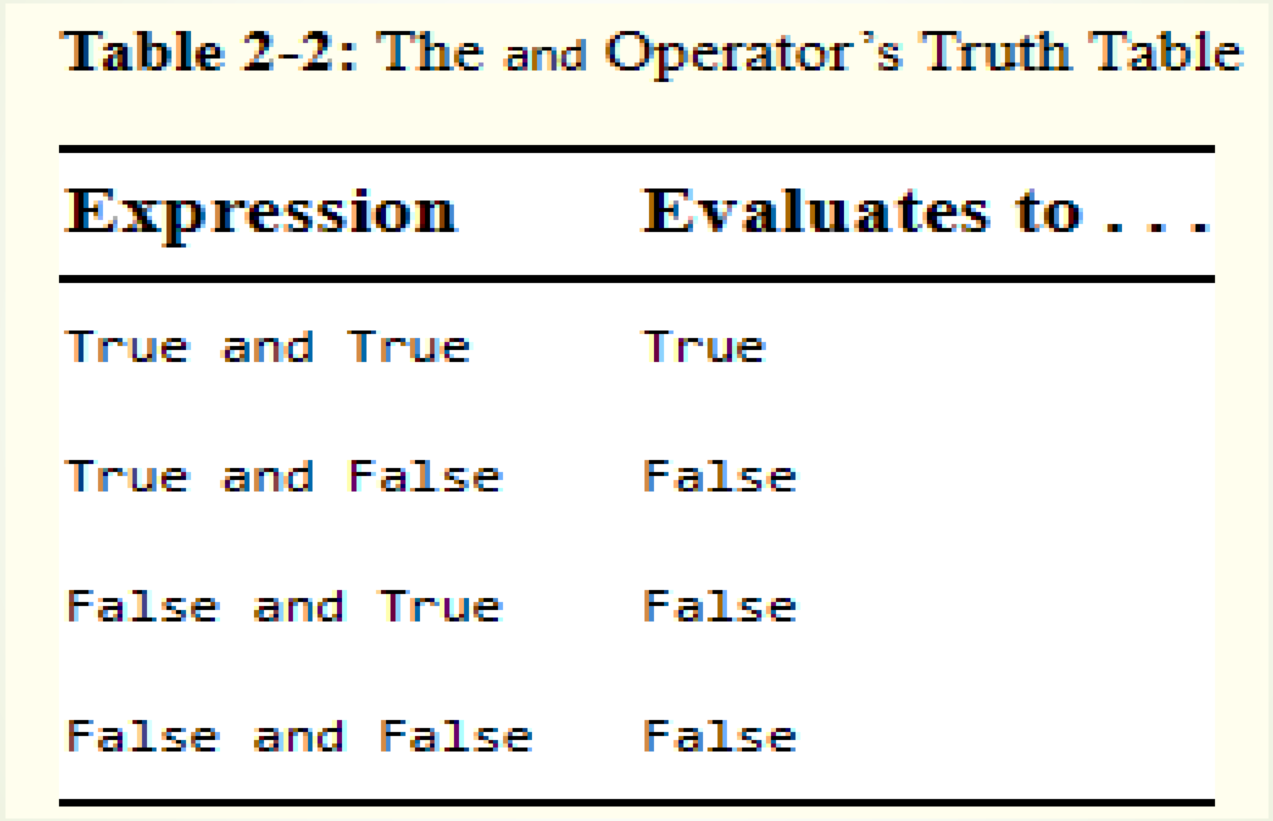


Table 2-2: The and Operator's Truth Table

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

Cont..

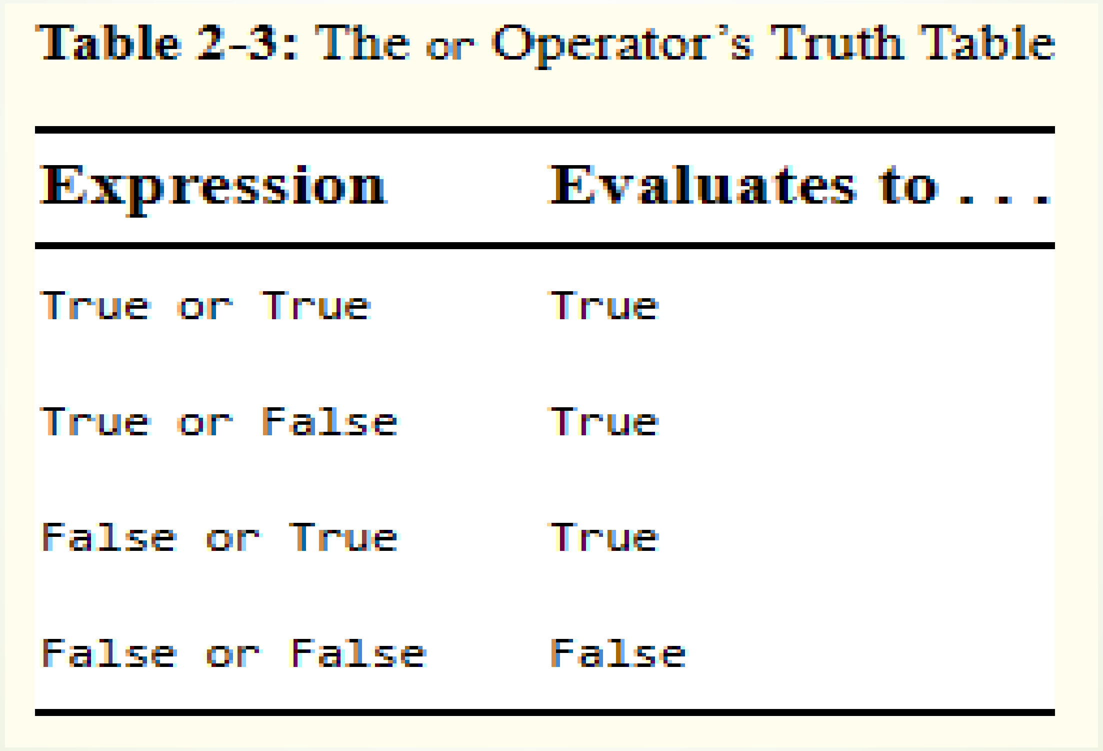


Table 2-3: The or Operator's Truth Table

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

Cont..

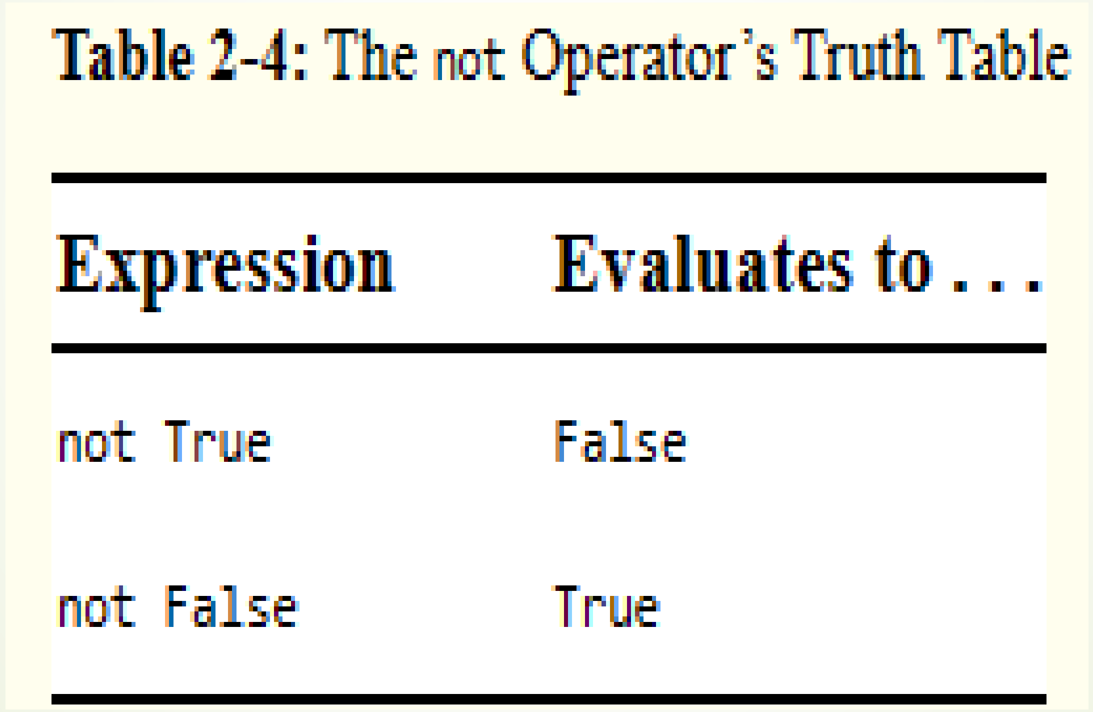
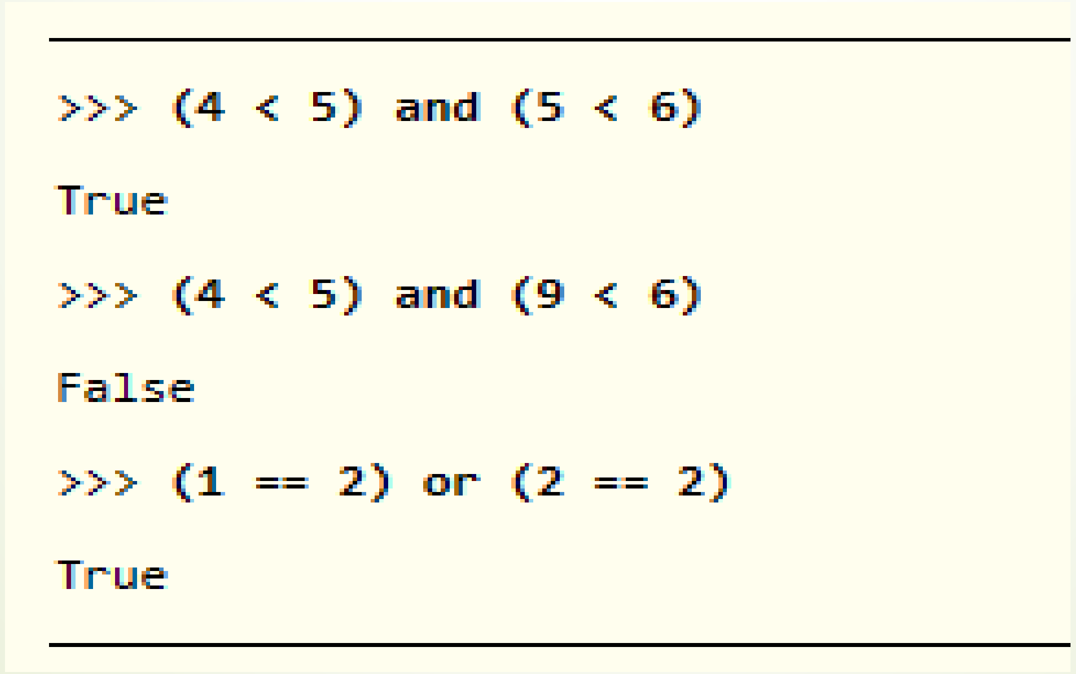


Table 2-4: The not Operator's Truth Table

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

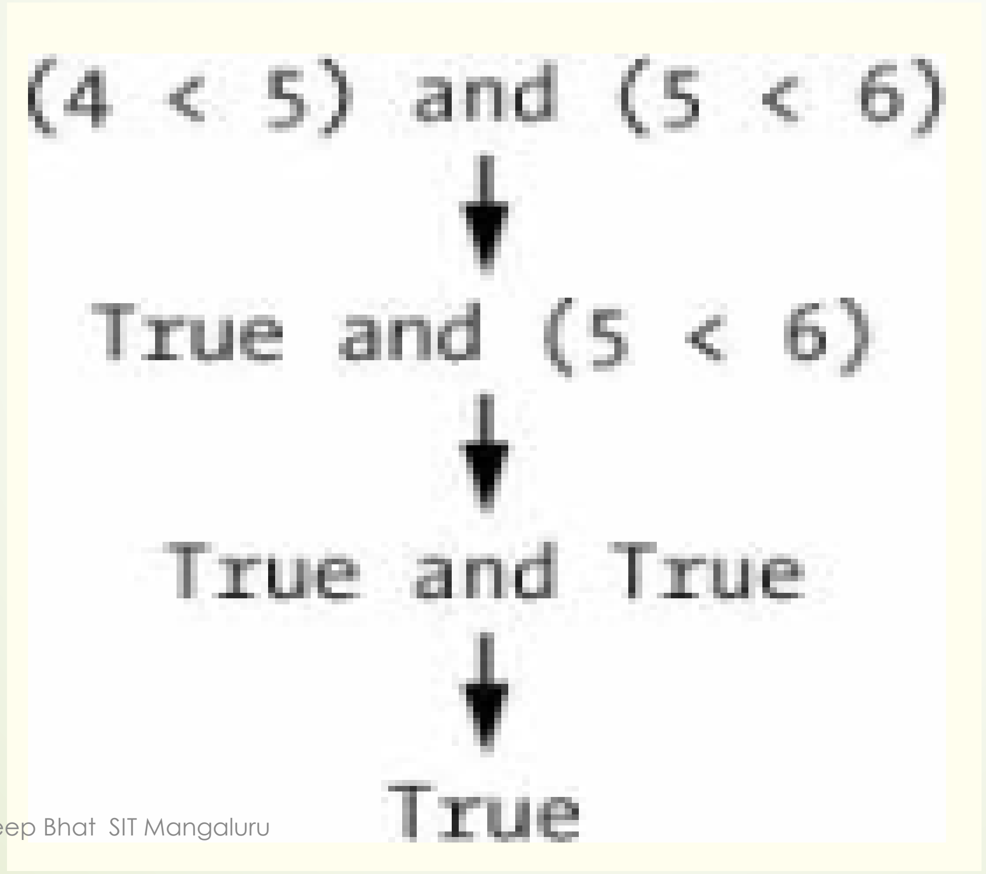
Mixing Boolean and Comparison Operators

- Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.



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

Cont..



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

Elements of Flow Control

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

Conditions

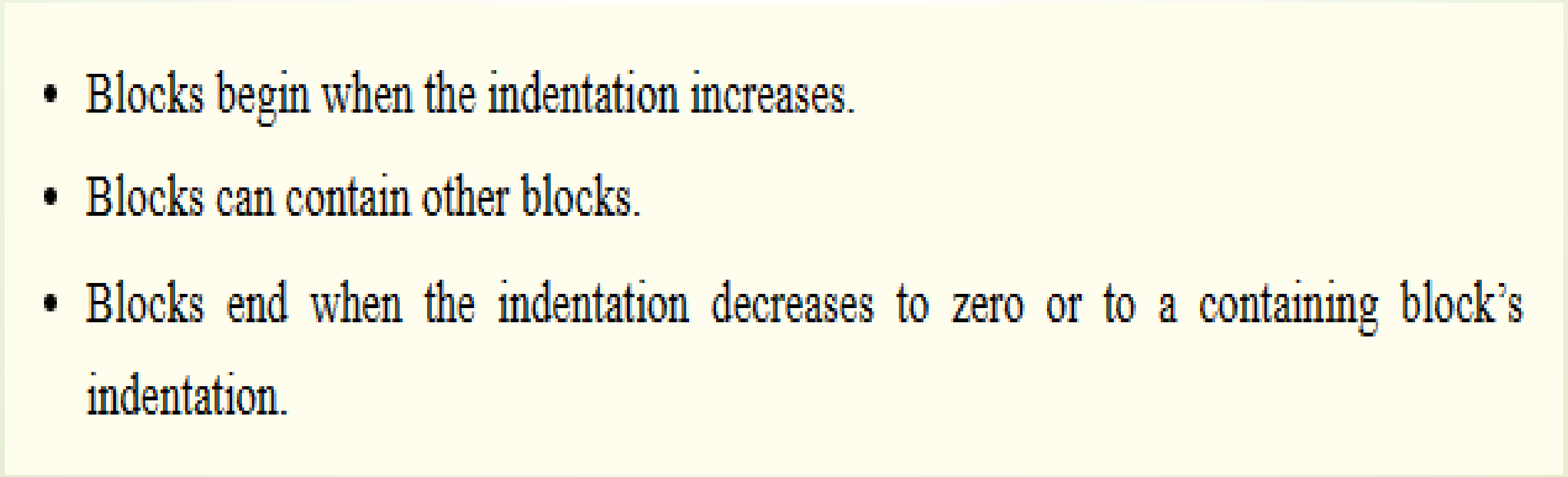
- The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; condition is just a more specific name in the context of flow control statements.

- 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 almost every flow control statement uses a condition.

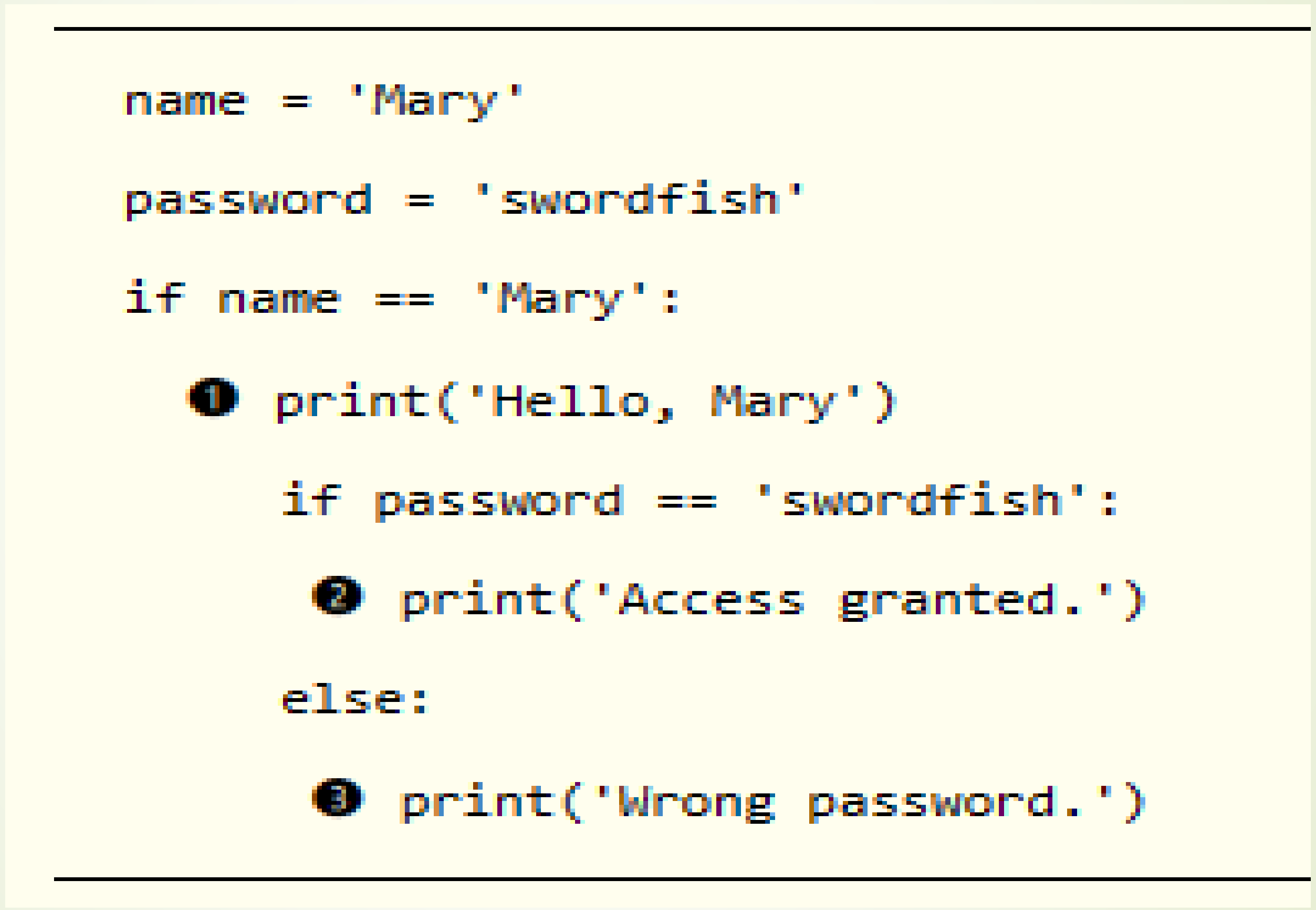
Blocks of Code

- Lines of Python code can be grouped together in blocks.
- You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.



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

Cont..



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

Program Execution

- The *program execution* (or simply, *execution*) is a term for the current instruction being executed.
- If you print the source code on paper and put your finger on each line as it is executed, you can think of your finger as the program execution.

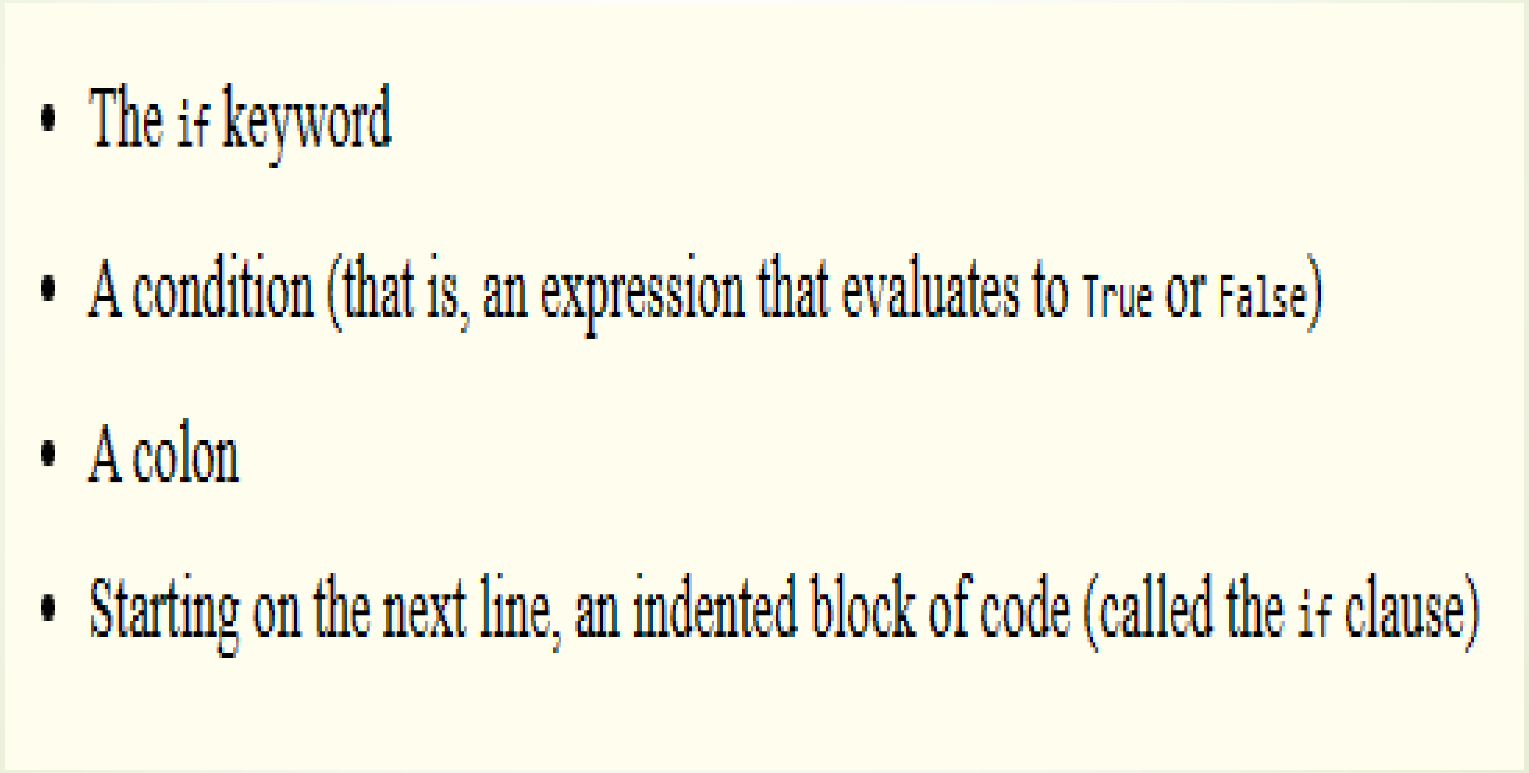
FLOW CONTROL STATEMENTS

if Statements

- 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's condition is True.
- The clause is skipped if the condition is False.

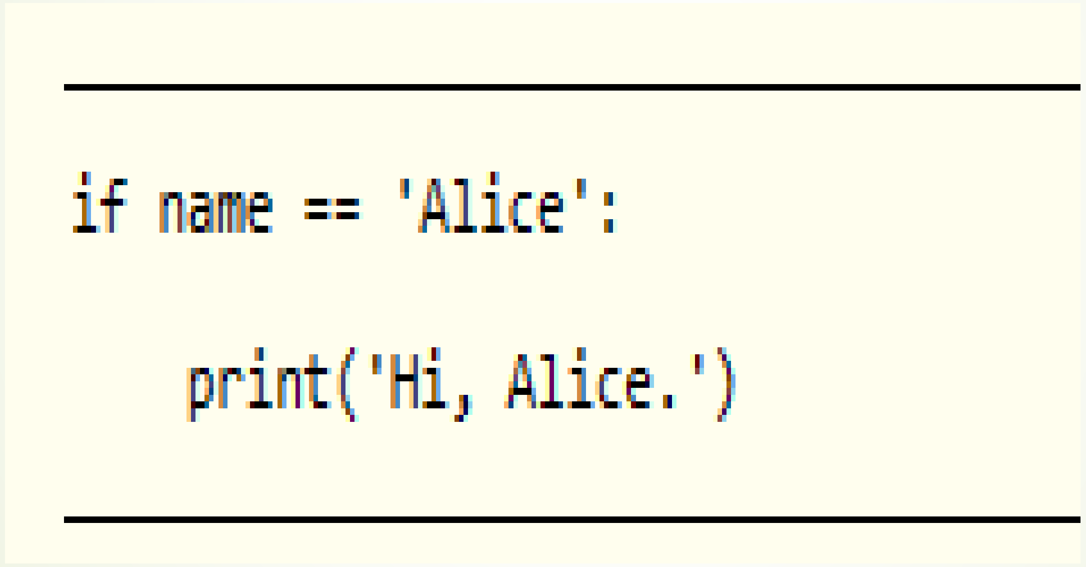
Cont..

- In Python, an if statement consists of the following:



- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `if` clause)

Cont..



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

Cont..

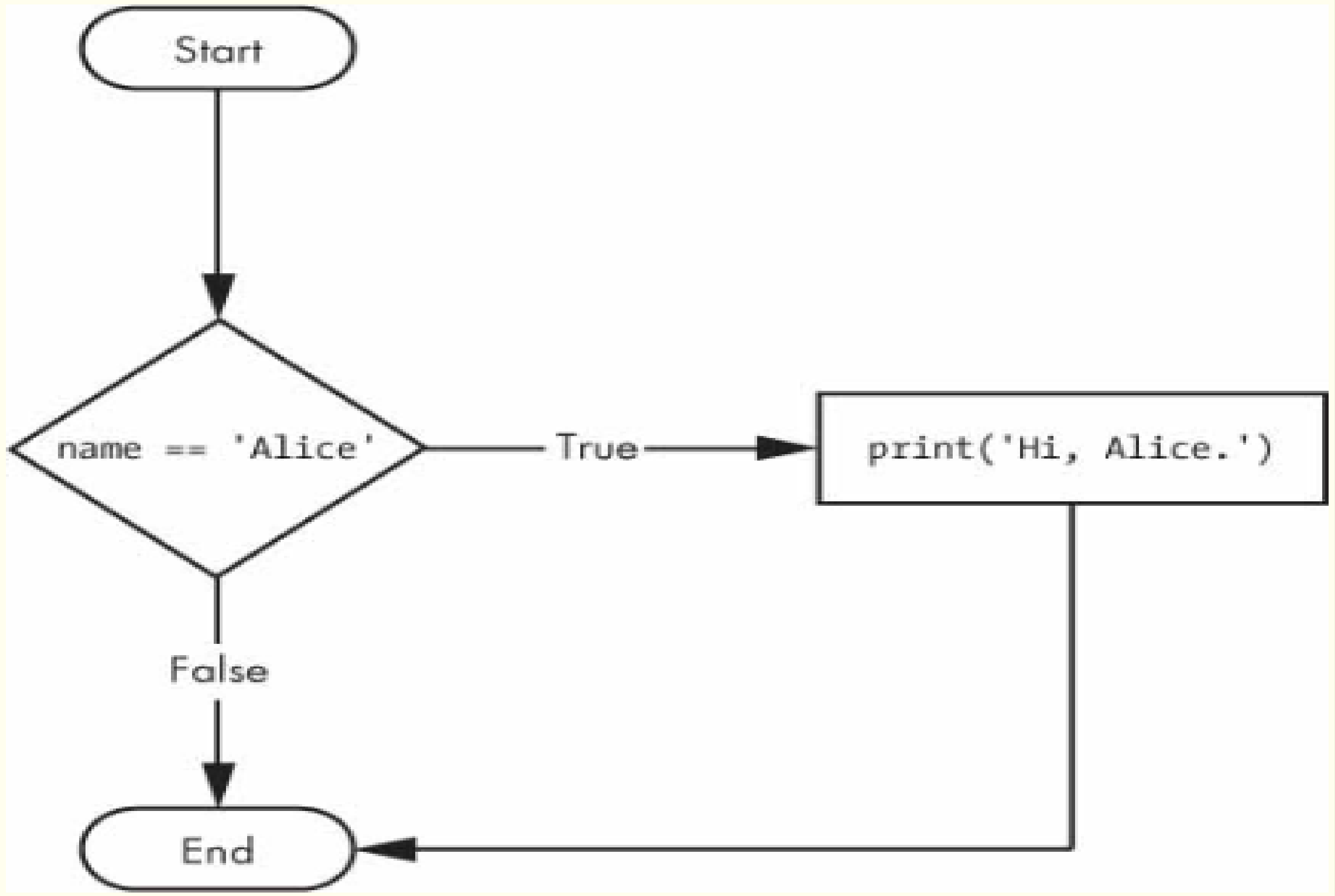


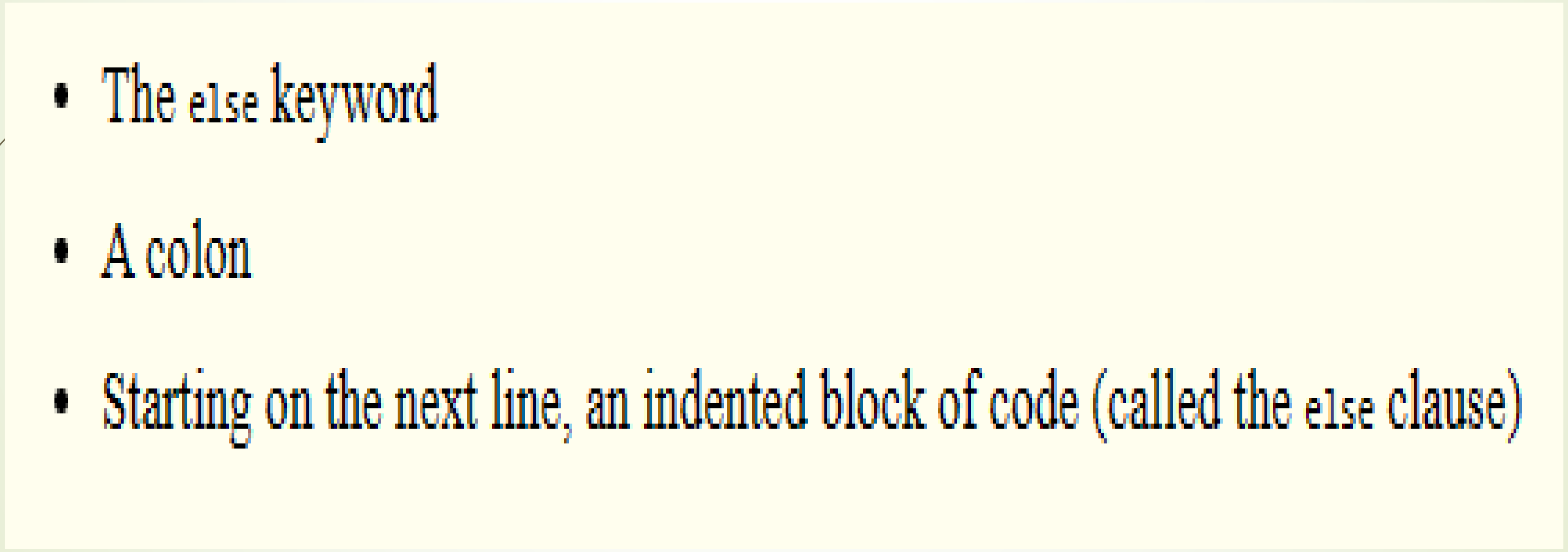
Figure 2-2: The flowchart for an if statement

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.
- Or else, execute that code." An else statement doesn't have a condition

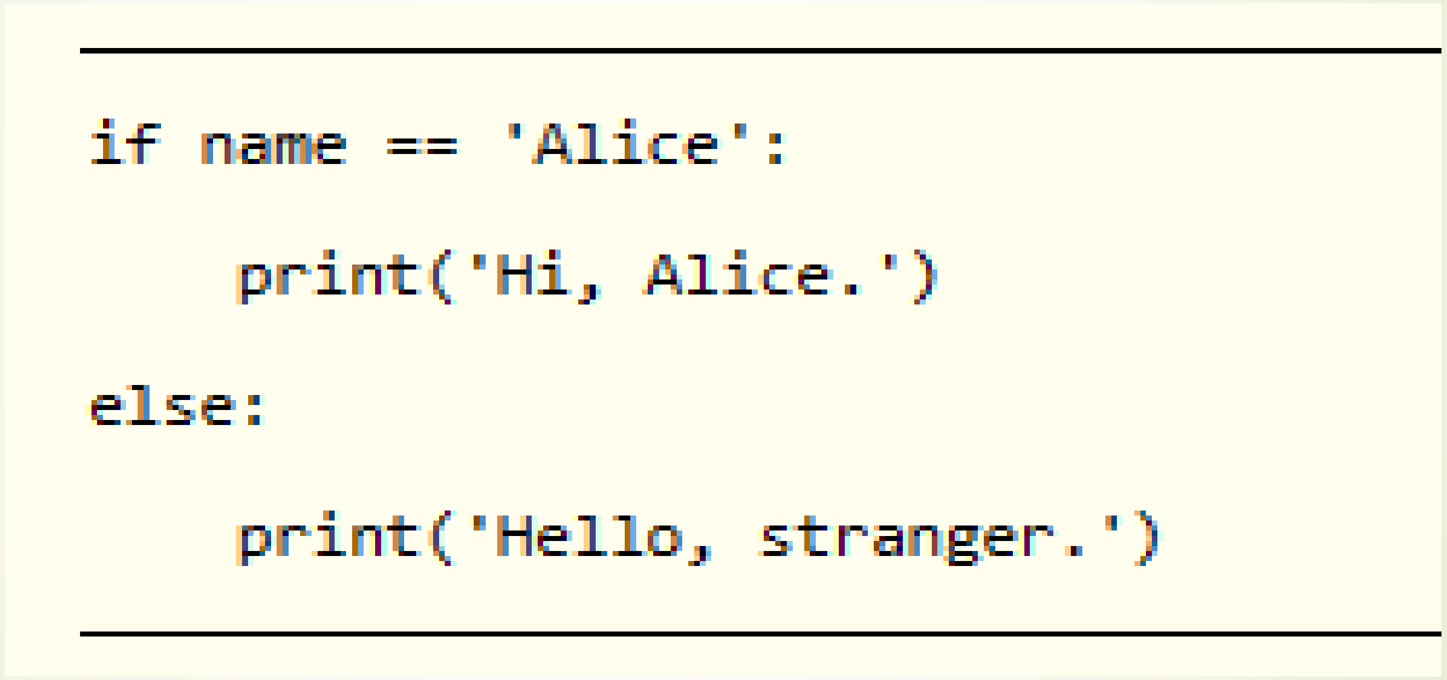
Cont..

- An else statement always consists of the following:



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

Cont..



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

Cont…

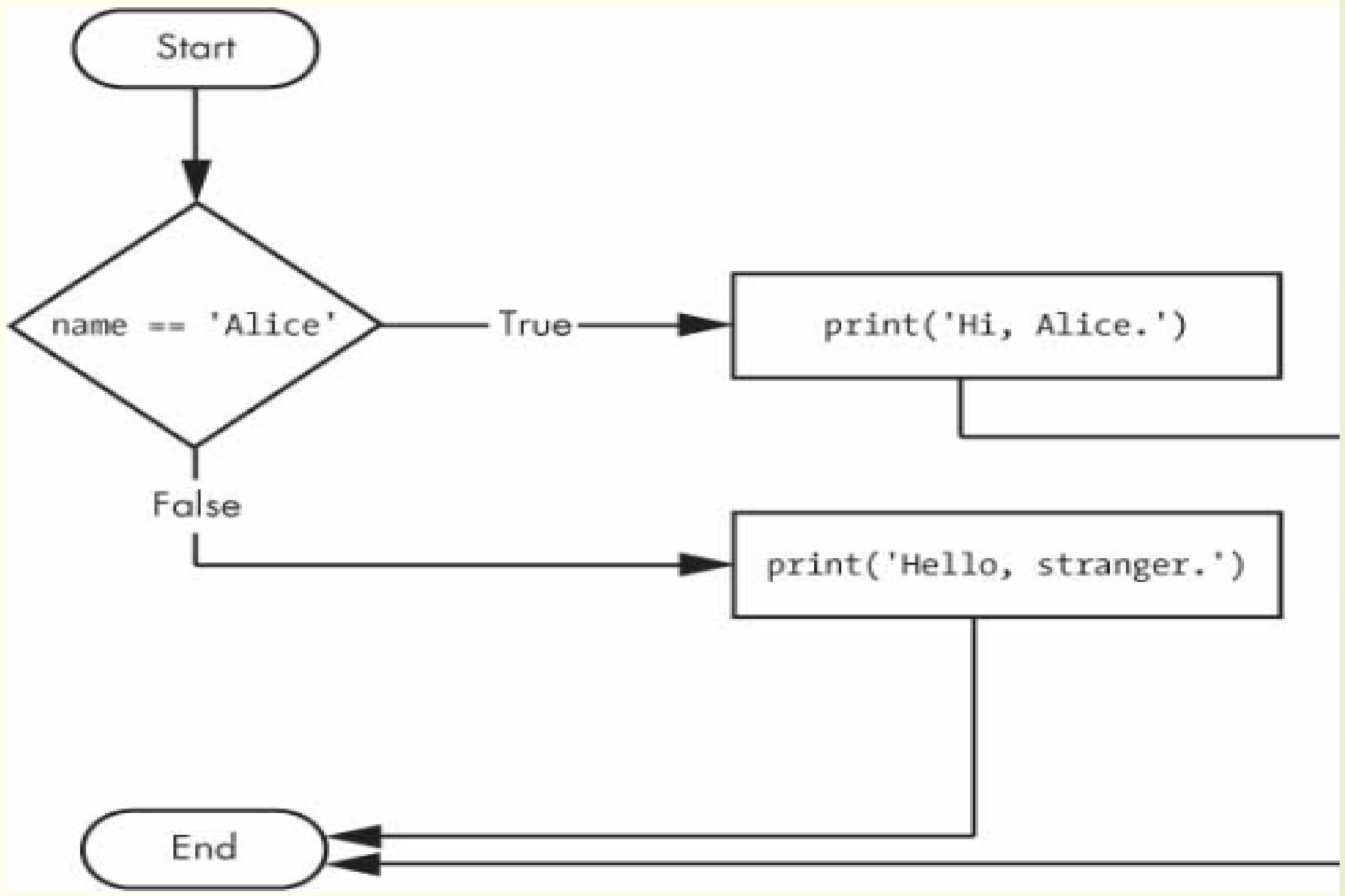
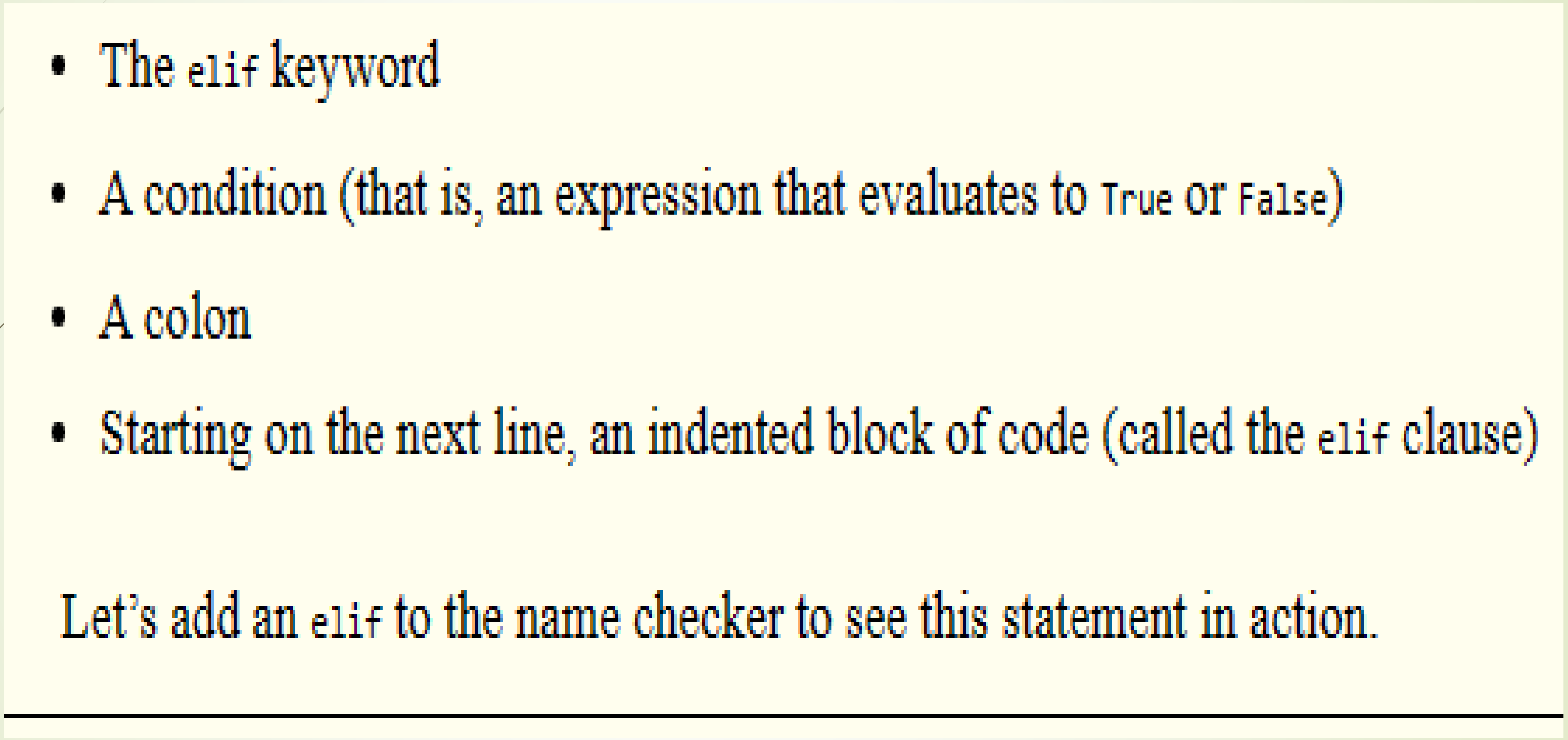


Figure 2-3: The flowchart for an `else` *statement*

elif Statements

- While only one of the if or else clauses will execute, you may have a case where you want one of 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:

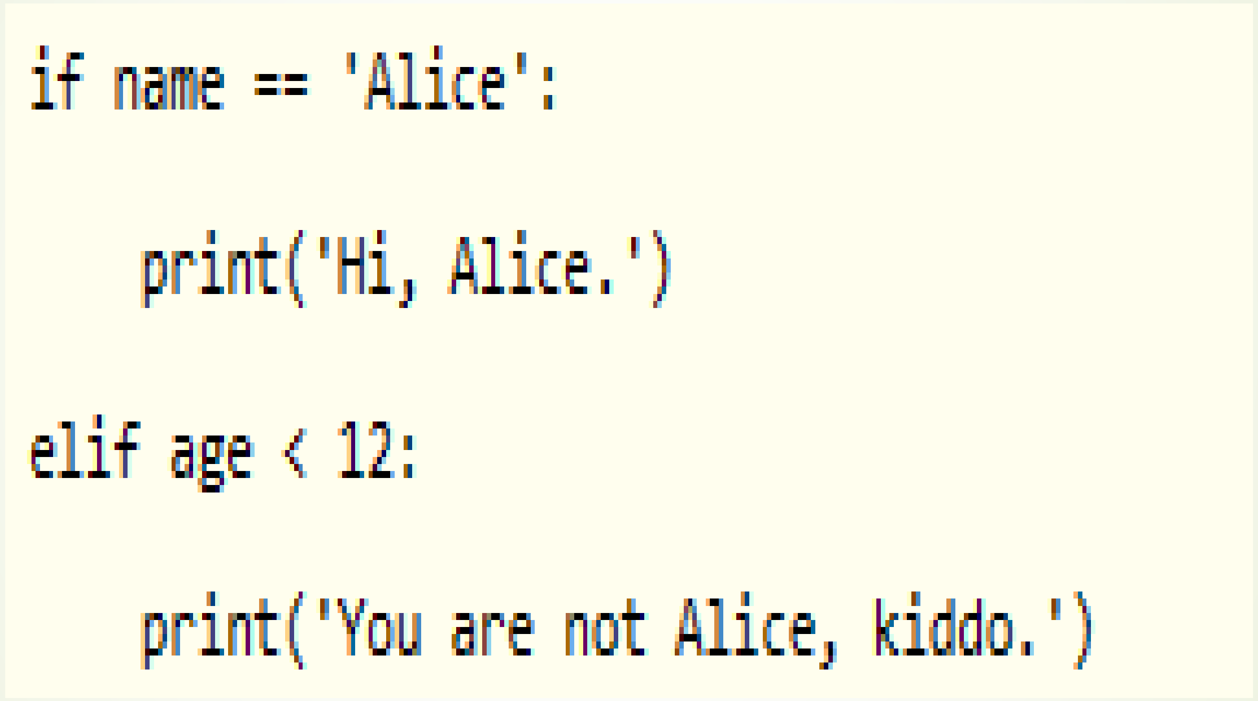
Cont..



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

Let's add an `elif` to the name checker to see this statement in action.

Cont..



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

Cont..

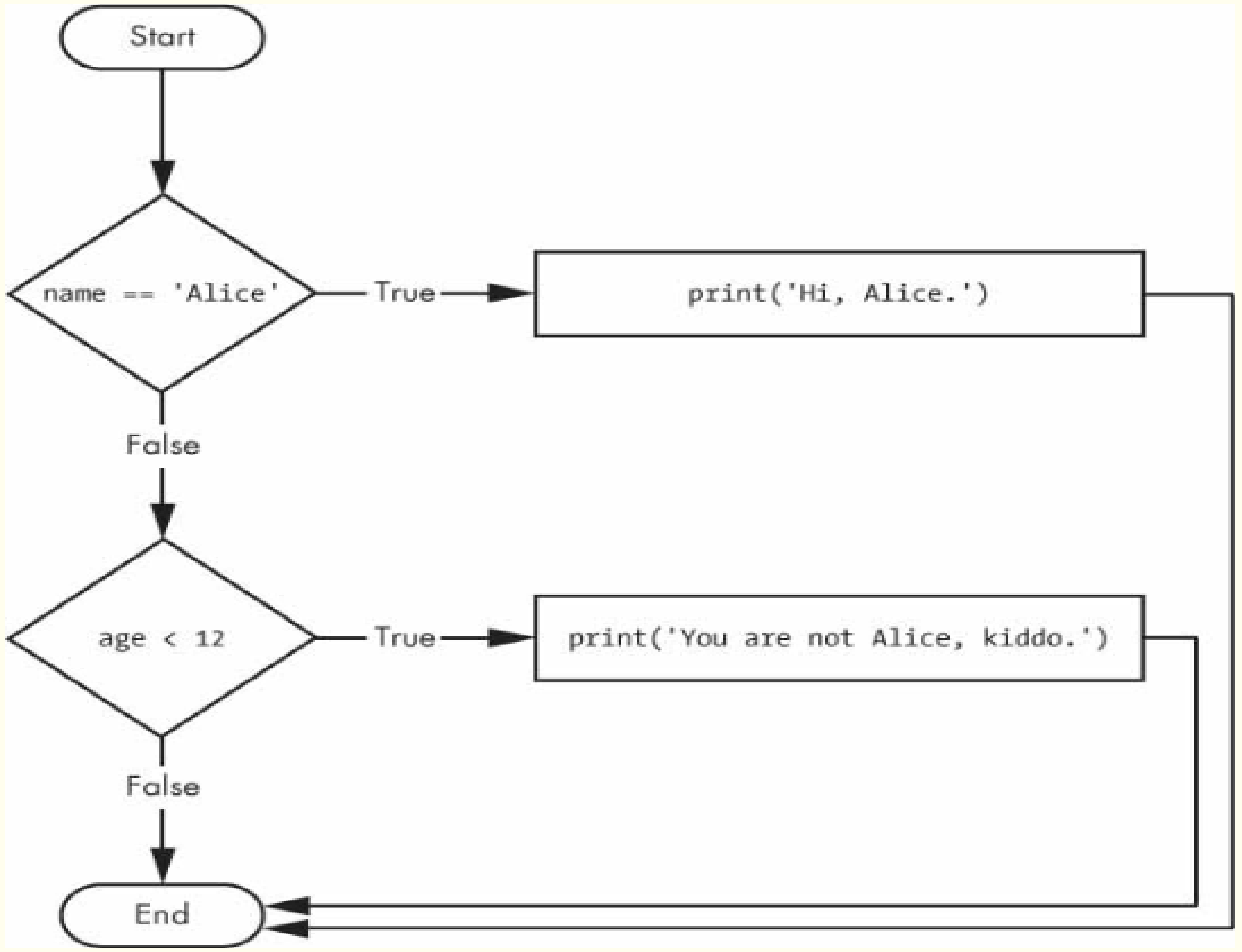
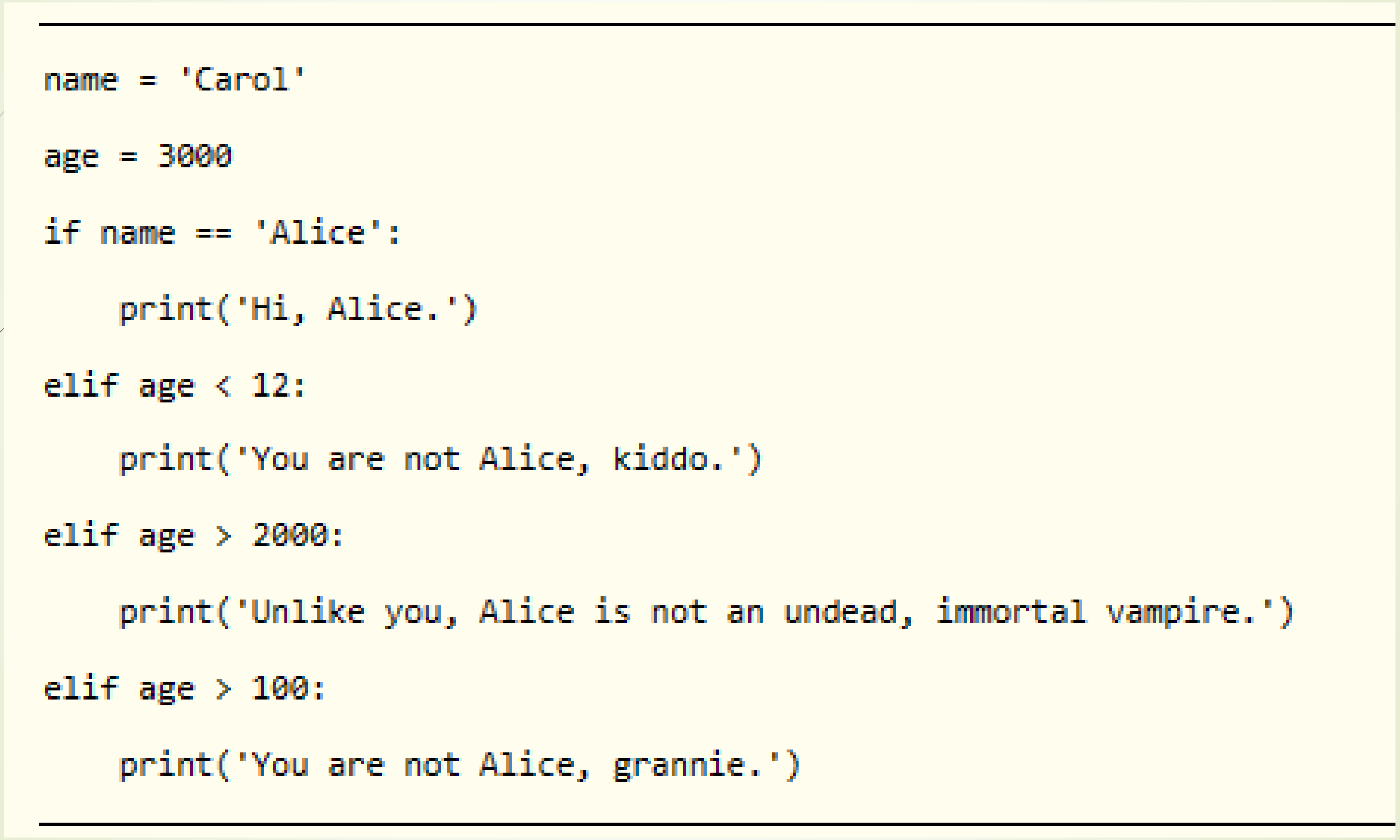


Figure 2-4: The flowchart for an `elif` *statement*

Cont..



```
name = 'Carol'

age = 3000

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.')
```

Cont..

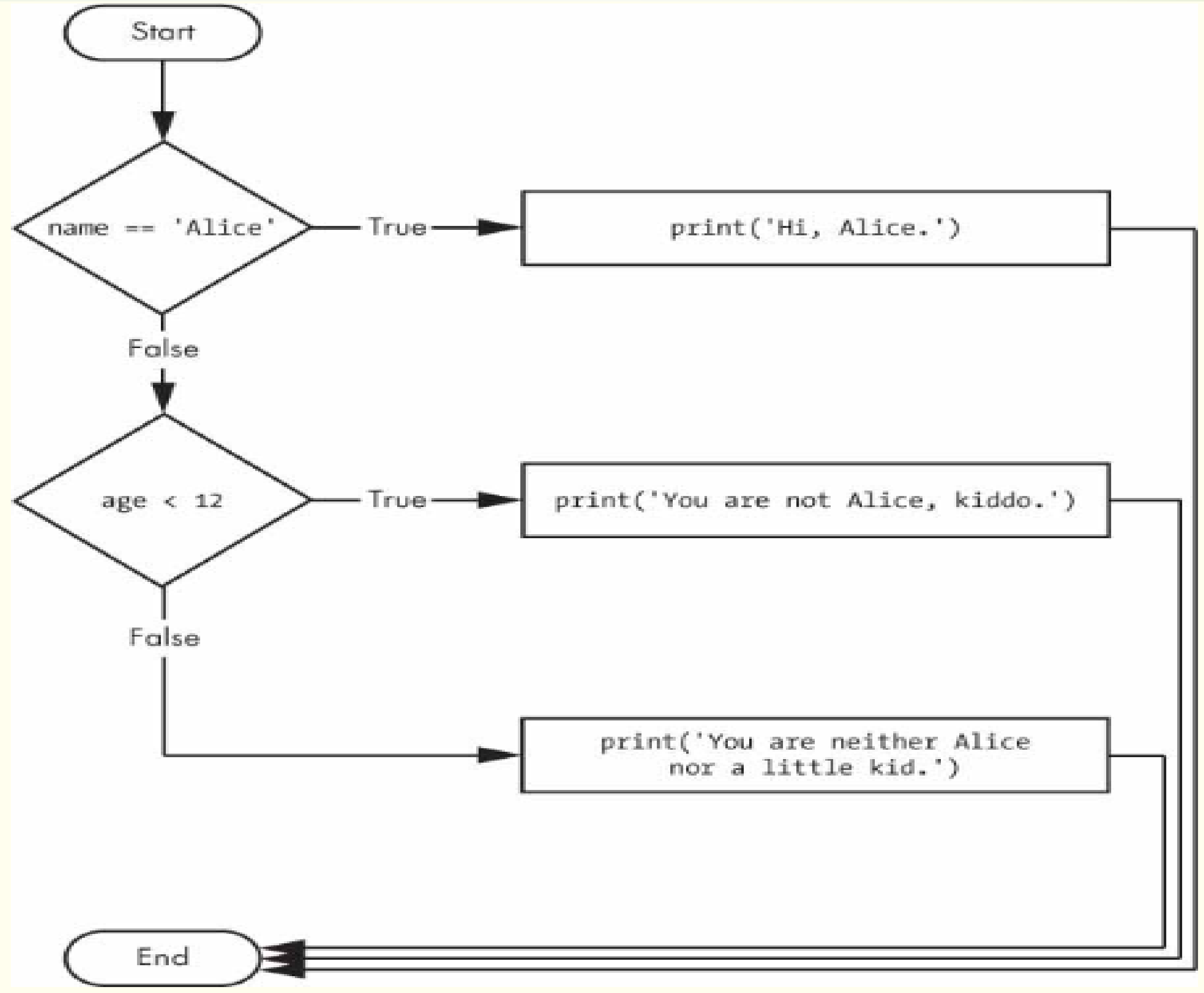

Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
print('You are neither Alice nor a little kid.')
End

Figure 2-7: Flowchart for the previous littleKid.py *program*

while Loop Statements

- You can make a block of code execute over and over again using 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:

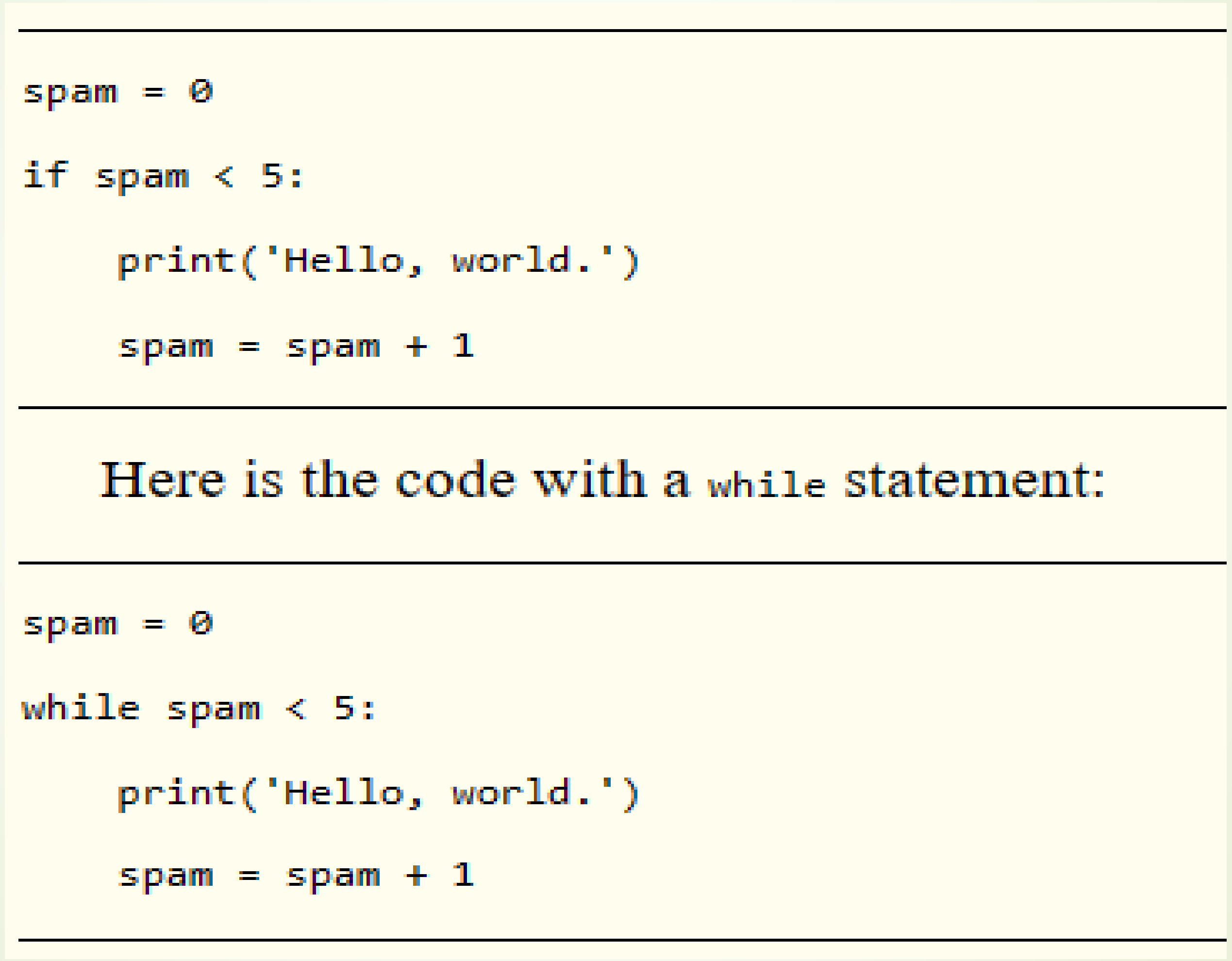
Cont..

- The while keyword
- A condition (that is, an expression that evaluates to True or False)
- A colon
- Starting on the next line, an indented block of code (called the while clause)

Cont..

- You 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.
- But at the end of a while clause, the program execution jumps back to the start of the while statement.
- The while clause is often called the *while loop* or just the *loop*.

Cont..

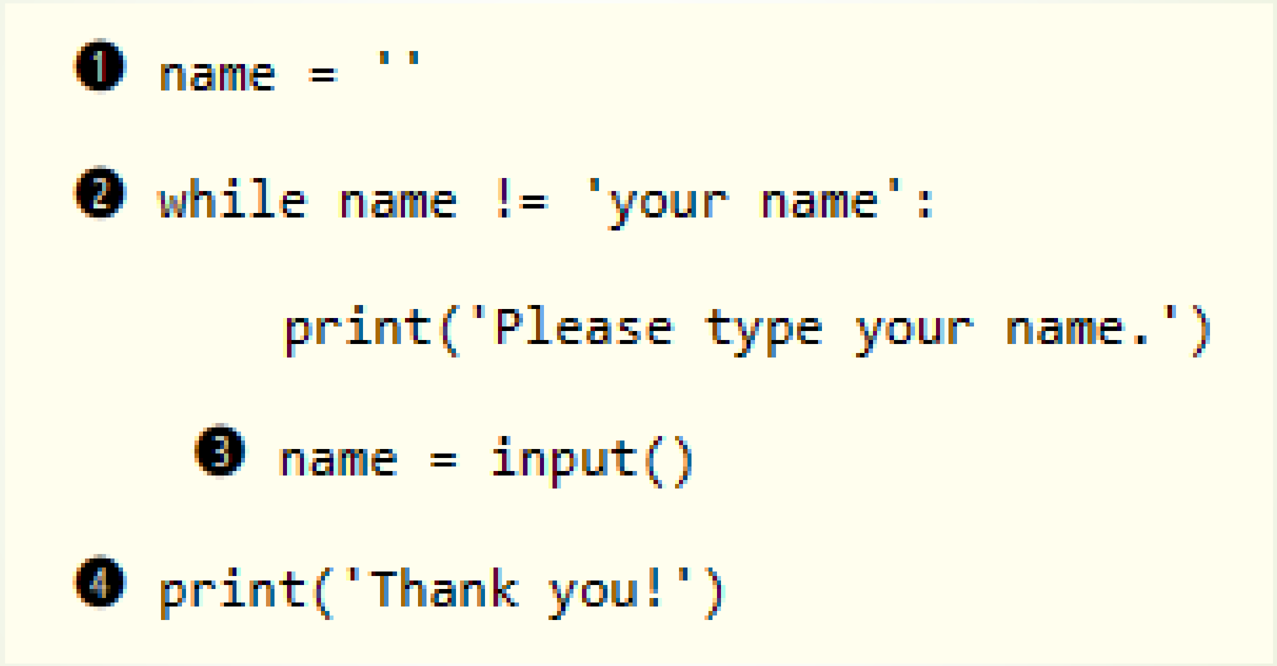


```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a `while` statement:

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

An Annoying while Loop



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

Cont..

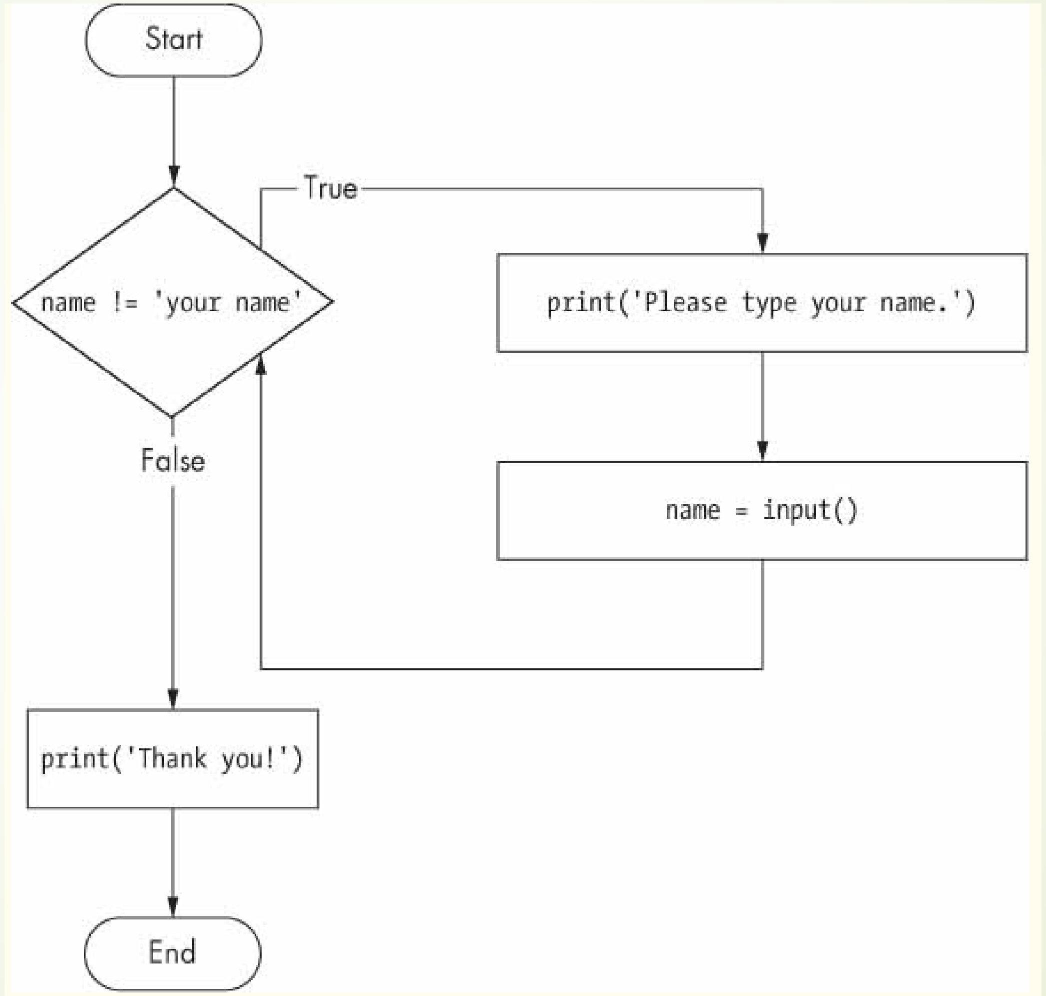
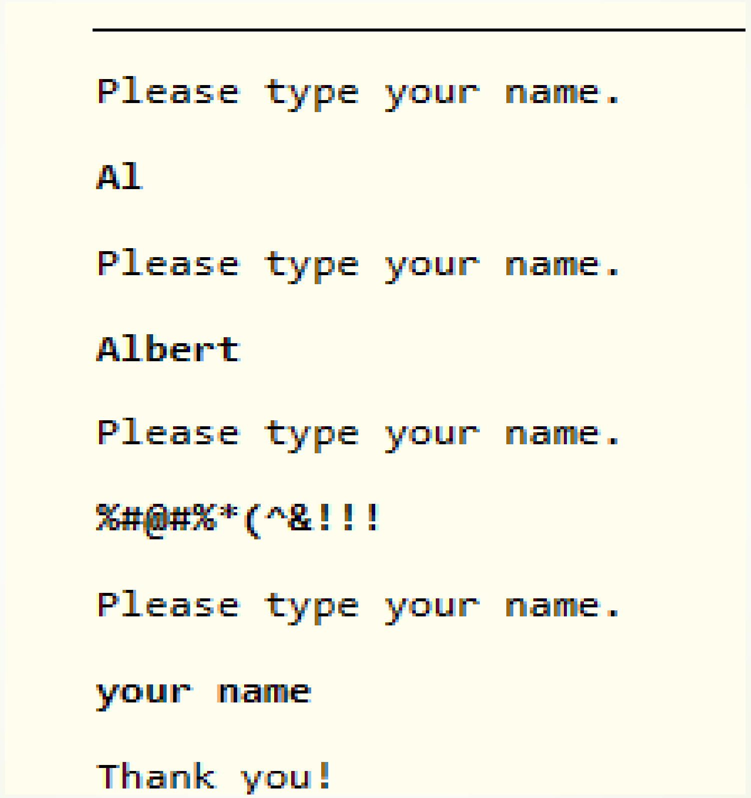


Figure 2-10: A flowchart of the yourName.py *program*

Cont..

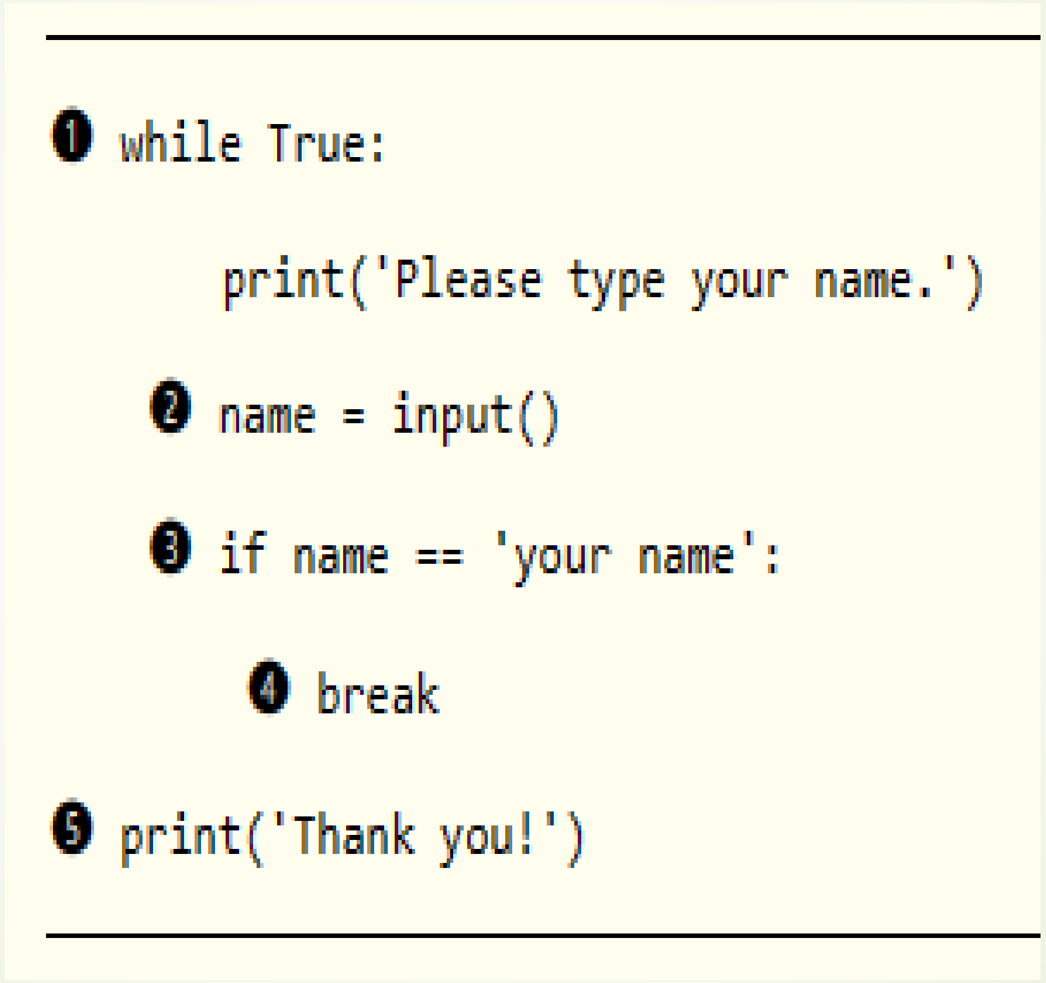


```
Please type your name.
Al
Please type your name.
Albert
Please type your name.
%#@#%*(^&!!!
Please type your name.
your name
Thank you!
```

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.

Cont..



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

Cont..

- The first line ❶ creates an infinite loop; it is a while loop whose condition is always True. (The expression True, after all, always evaluates down to the value True.)

- After the program execution enters this loop, it will exit the loop only when a break statement is executed. (An infinite loop that never exits is a common programming bug.)

Flow chart

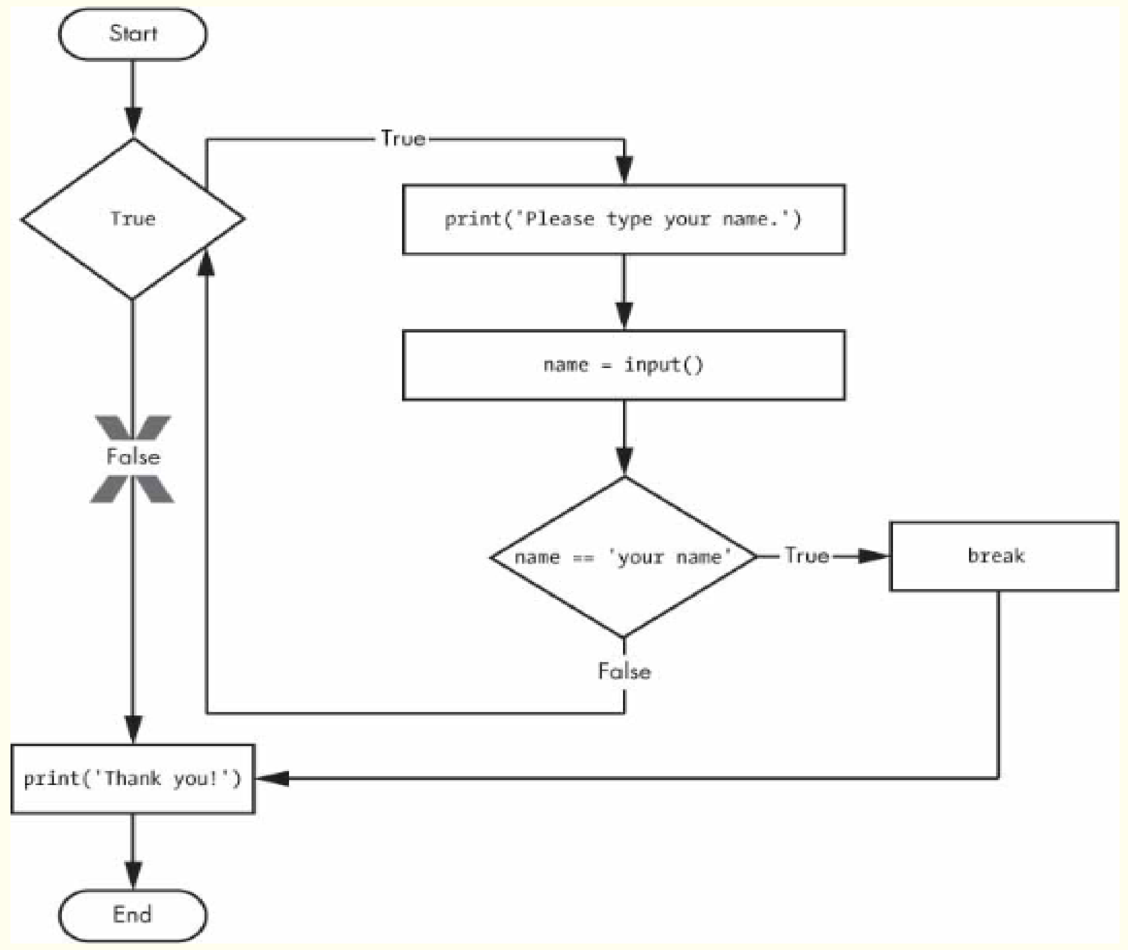


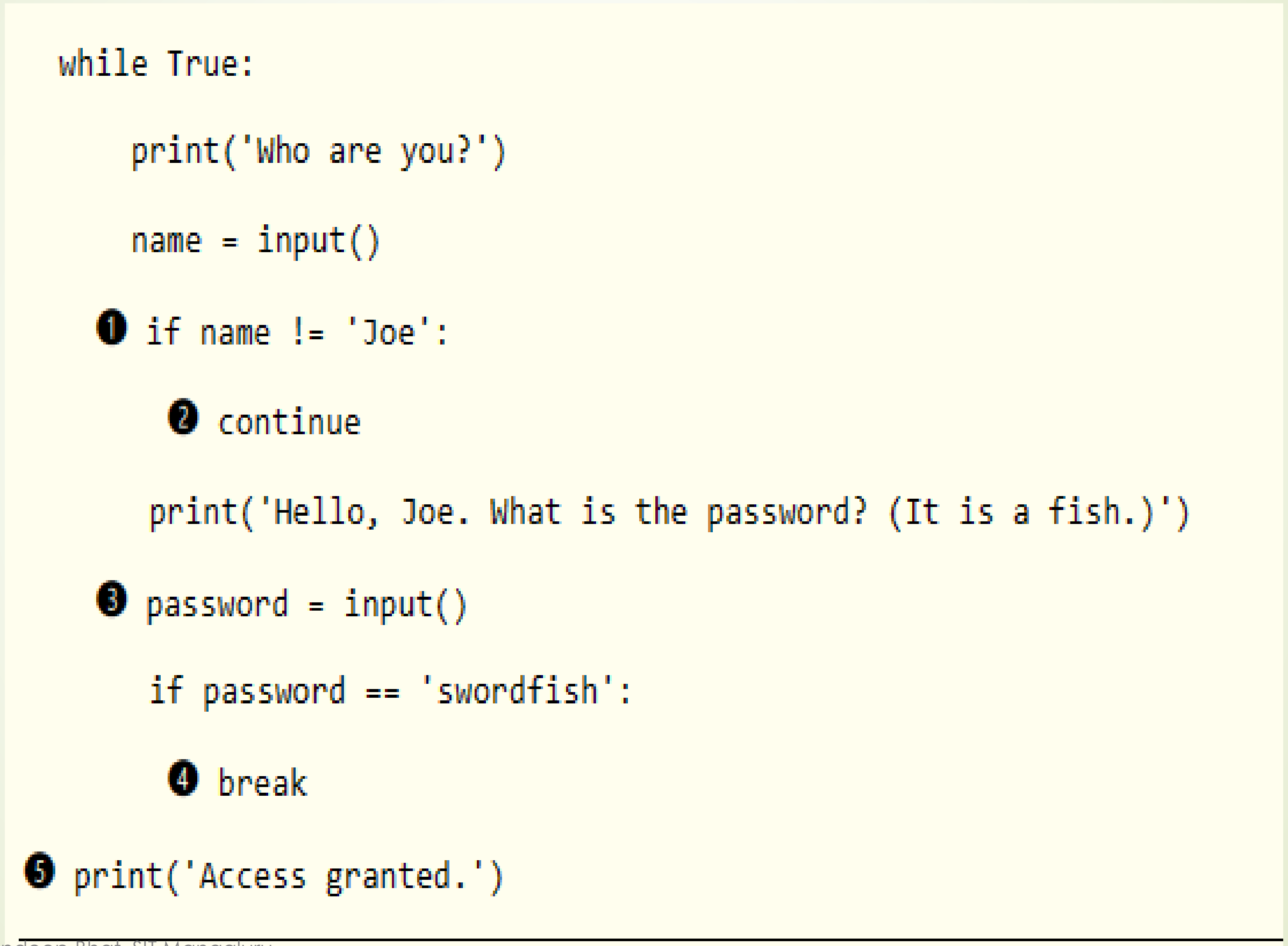
Figure 2-11: The flowchart for the yourName2.py *program with an infinite loop. Note that the X path will*

Continue Statements

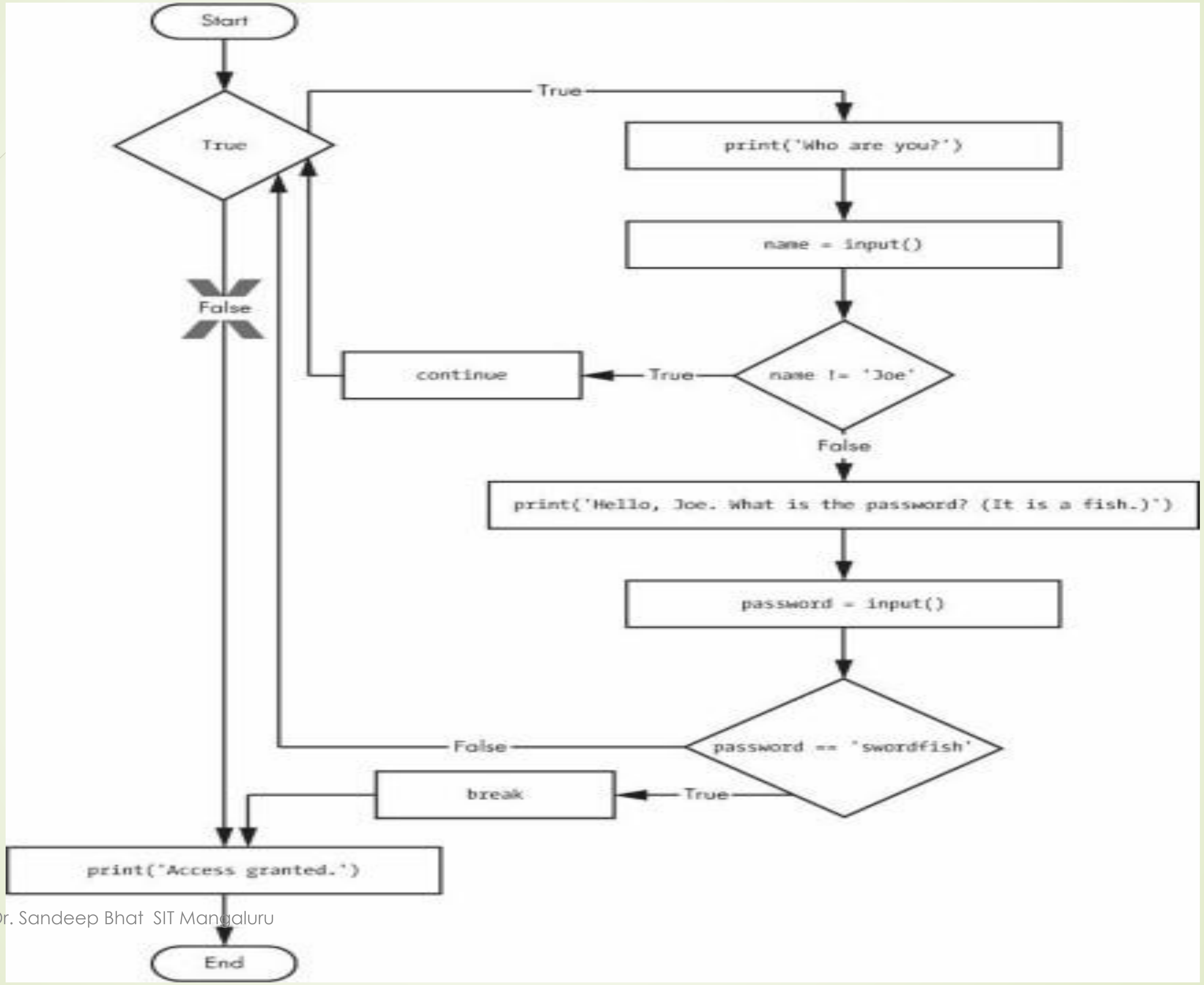
- Like break statements, continue statements are used inside loops.

- 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. (This is also what happens when the execution reaches the end of the loop.)

Cont..



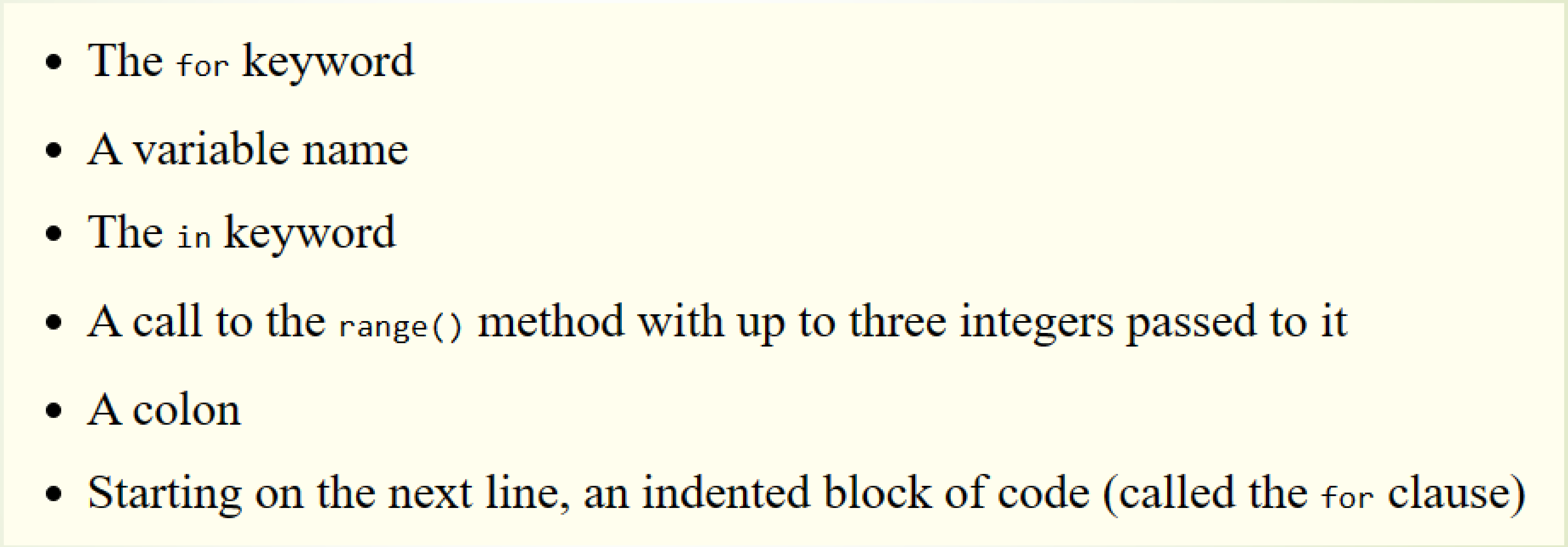
```
while True:
    print('Who are you?')
    name = input()
 ❶ if name != 'Joe':
     ❷ continue
    print('Hello, Joe. What is the password? (It is a fish.)')
 ❸ password = input()
    if password == 'swordfish':
     ❹ break
❺ print('Access granted.')
```


Start
True
True
print('Who are you?')
name = input()
name != 'Joe'
True
continue
False
False
print('Hello, Joe. What is the password? (It is a fish.)')
password = input()
password == 'swordfish'
False
True
break
print('Access granted.')
End

for Loops and the range() Function

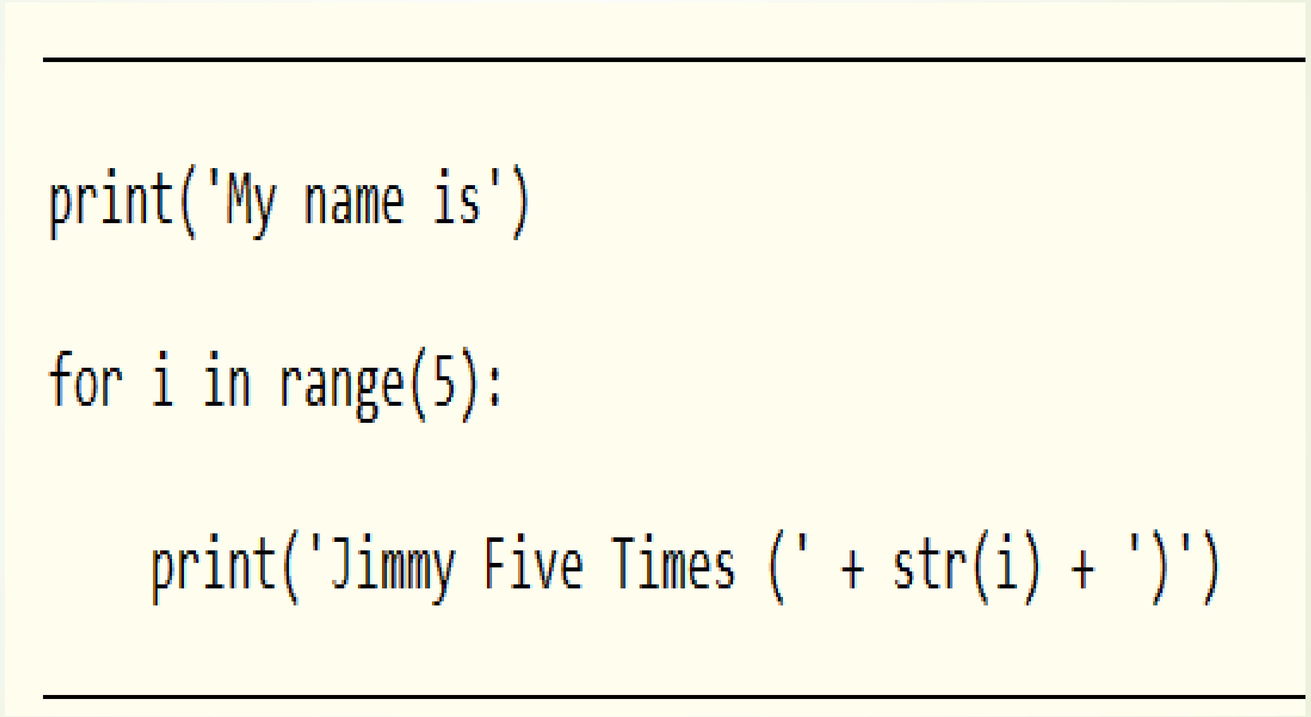
- The while loop keeps looping while its condition is True (which is the reason for its name), but what if you want to execute a block of code only a certain number of times?
- You can do this with a for loop statement and the range() function.
- In code, a for statement looks something like for i in range(5): and includes the following:

Cont..



- The `for` keyword
- A variable name
- The `in` keyword
- A call to the `range()` method with up to three integers passed to it
- A colon
- Starting on the next line, an indented block of code (called the `for` clause)

Cont..



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

Cont..

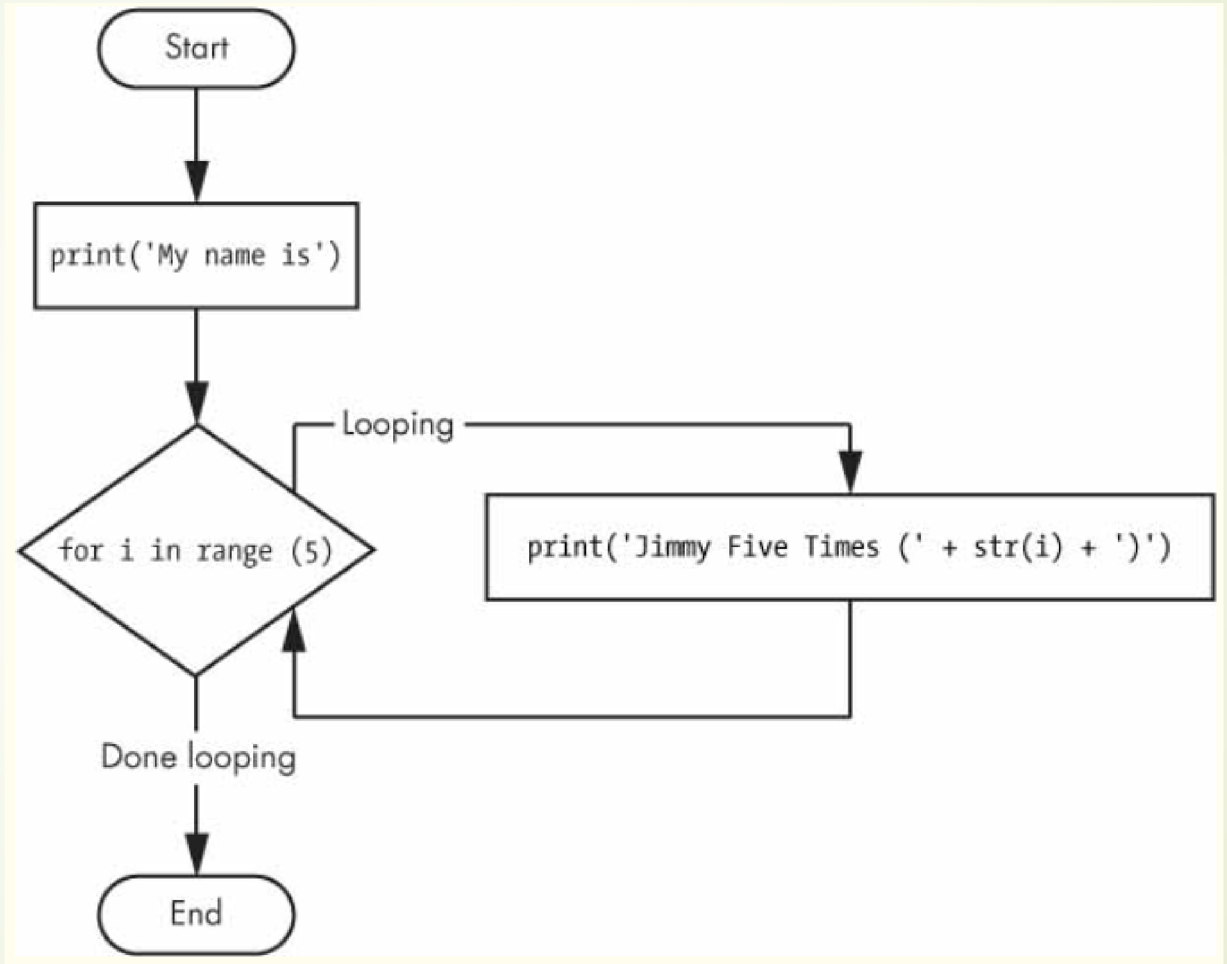
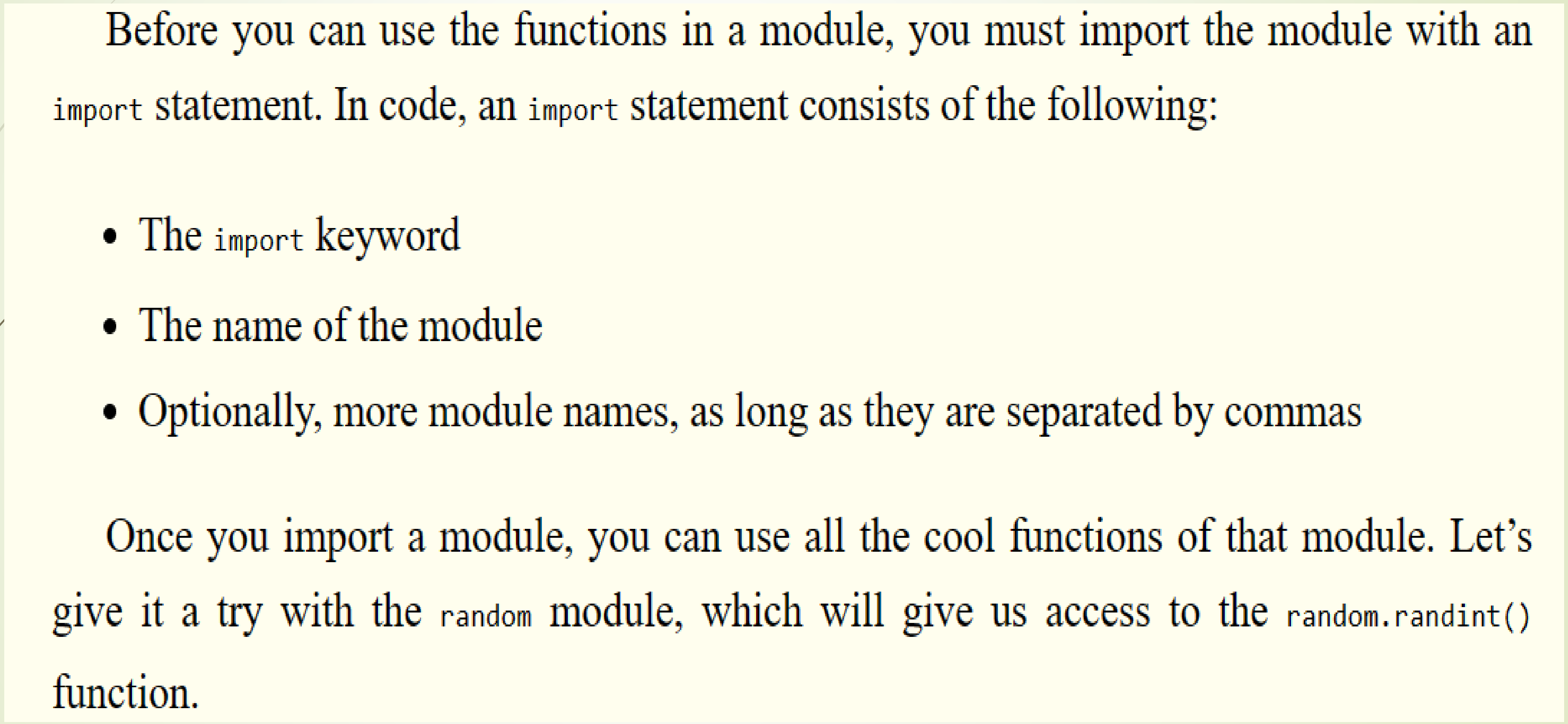


Figure 2-13: The flowchart for fiveTimes.py

Importing Modules

- All Python programs can call a basic set of functions called built-in functions, including the print(), input(), and len() functions you've seen before.
- 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 your programs.
- For example, the math module has mathematics-related functions, the random module has random number-related functions, and so on.

Cont..



Before you can use the functions in a module, you must import the module with an `import` statement. In code, an `import` statement consists of the following:

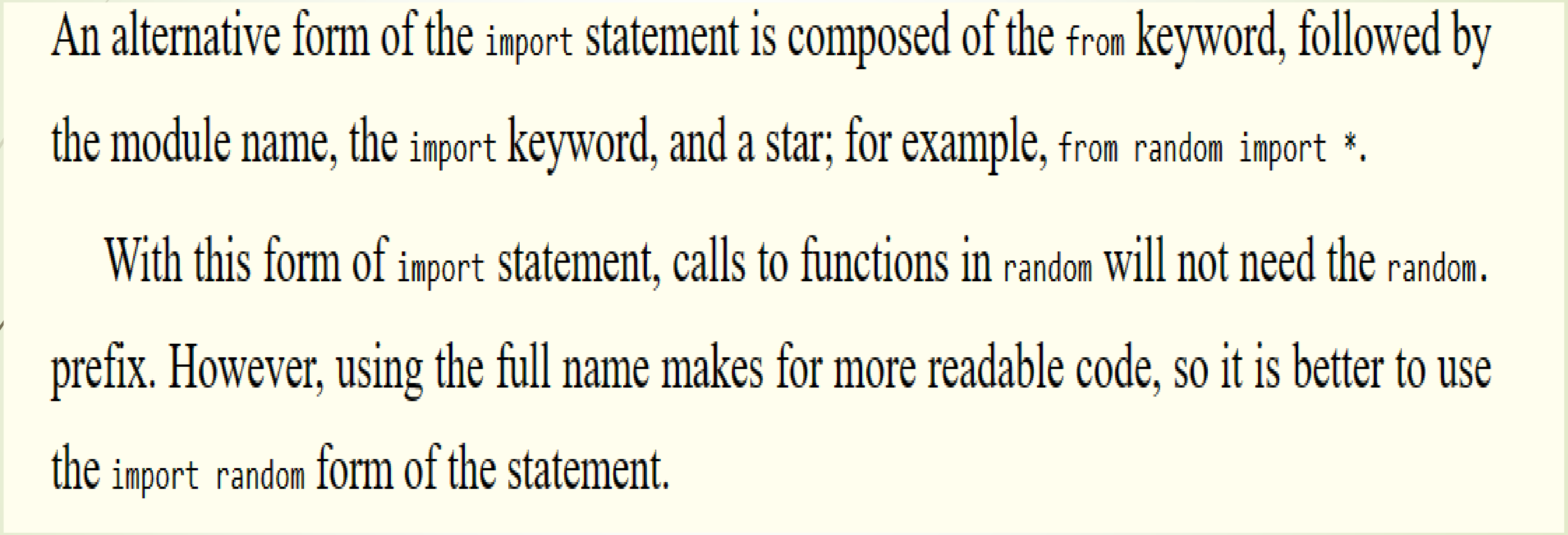
- The `import` keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

Cont..

```
import random

for i in range(5):

    print(random.randint(1, 10))
```

from import Statements



An alternative form of the `import` statement is composed of the `from` keyword, followed by the module name, 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 `import random` form of the statement.

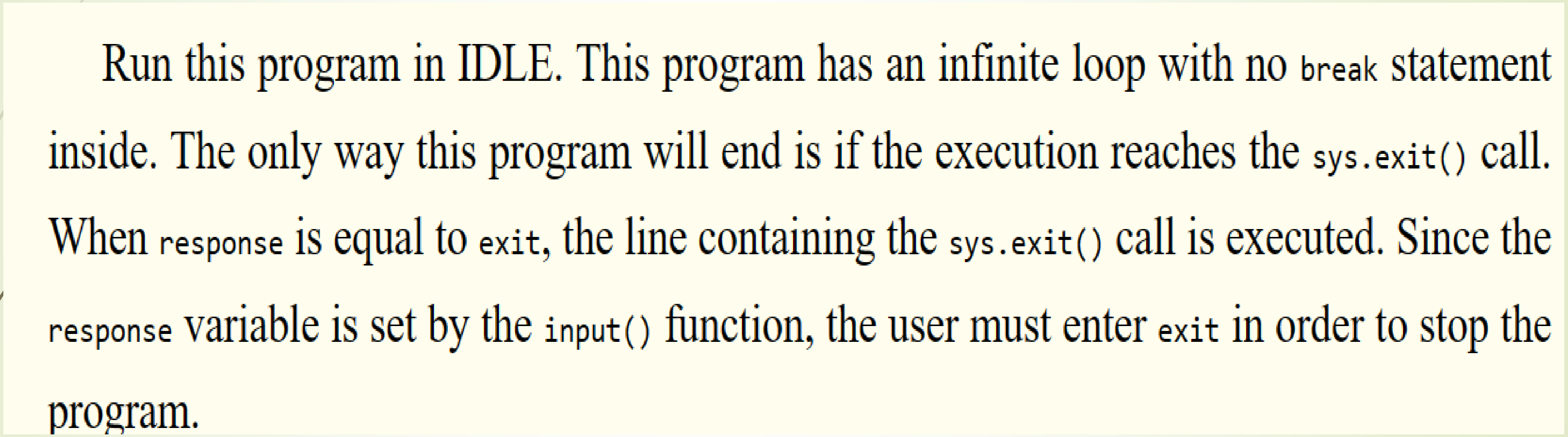
Ending a Program Early with the sys.exit() Function

- The last flow control concept to cover is how to terminate the program.
- Programs always terminate if the program execution reaches the bottom of the instructions.
- However, you can cause the program to terminate, or exit, before the last instruction by calling the sys.exit() function.
- Since this function is in the sys module, you have to import sys before your program can use it.

Cont..

```
import sys


while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

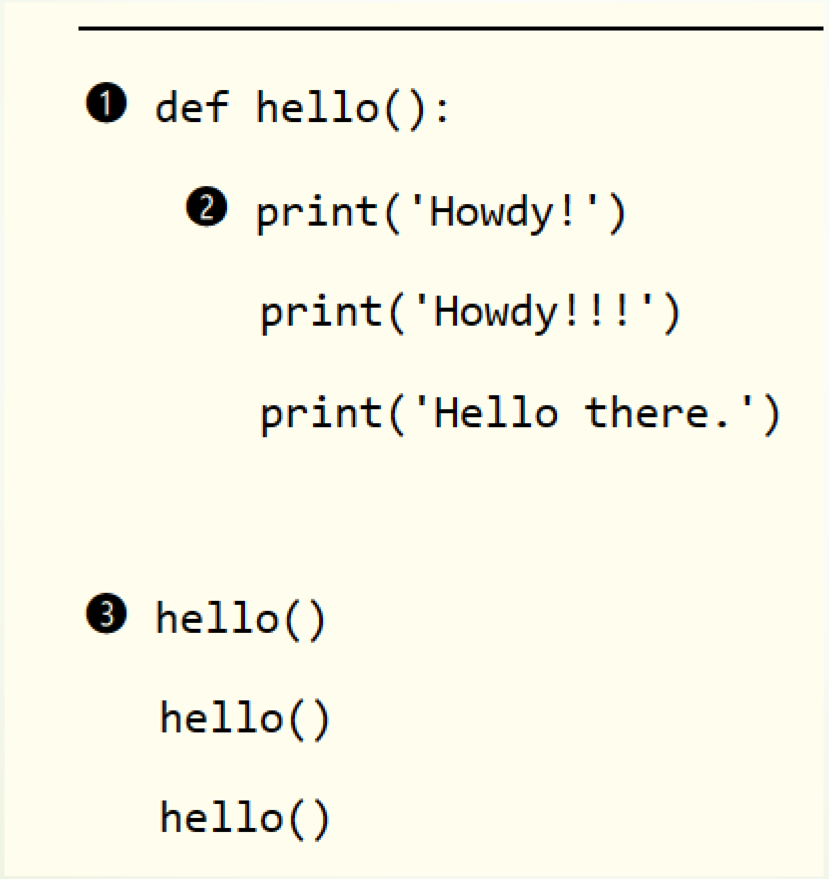
Cont..



Run this program in IDLE. This program has an infinite loop with no `break` statement inside. The only way this program will end is if the execution reaches the `sys.exit()` call. When `response` is equal to `exit`, the line containing the `sys.exit()` call is executed. Since the `response` variable is set by the `input()` function, the user must enter `exit` in order to stop the program.

Function

- A function is like a miniprogram within a program.



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

❸ hello()
  hello()
  hello()
```

Cont..

- 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.

Cont..

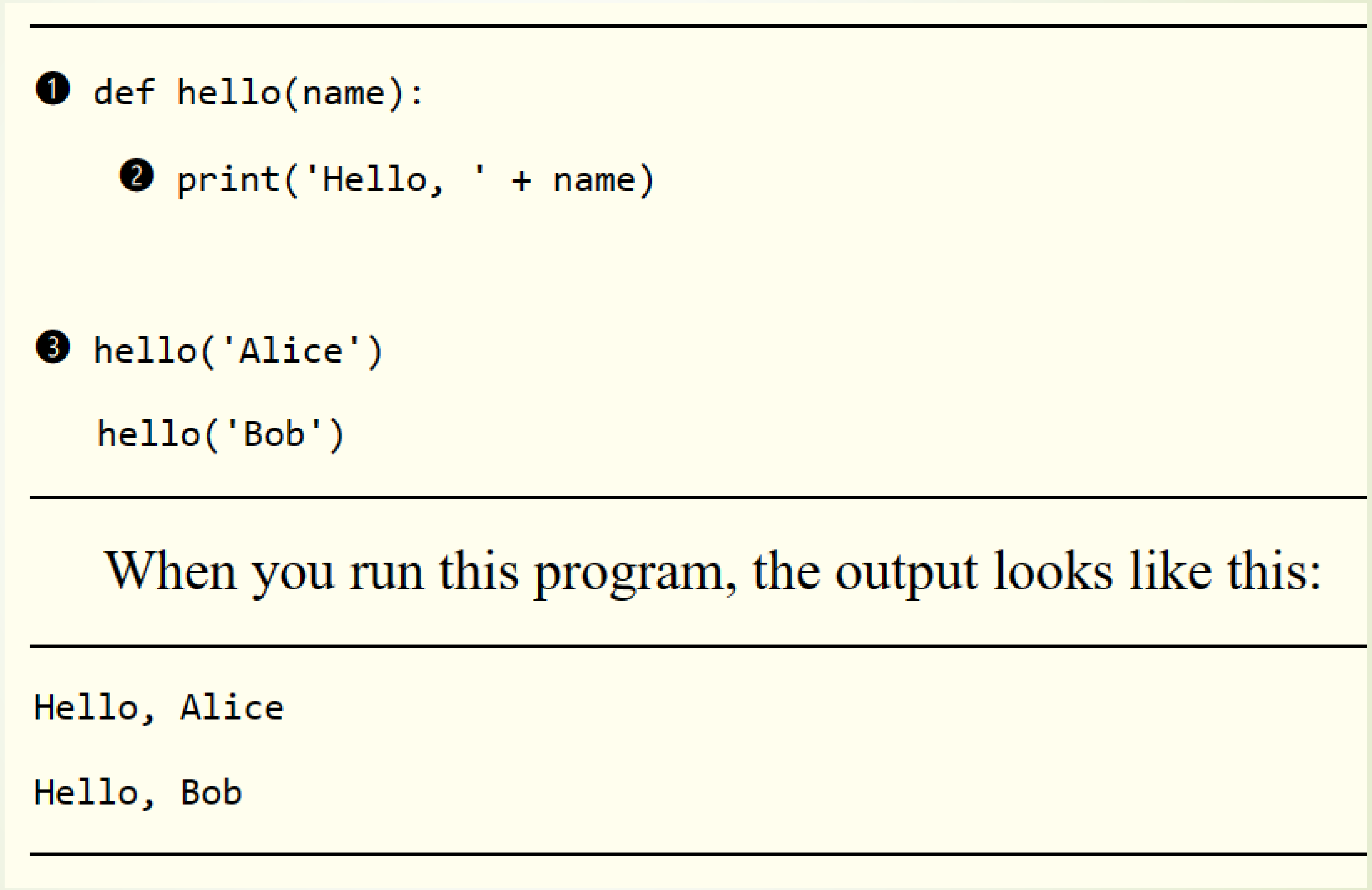
- In code, a function call is just the function’s name followed by parentheses, possibly with some number of arguments in between the parentheses.

- 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.

- def Statements with Parameters
- When you call the print() or len() function, you pass them values, called arguments, by typing them between the parentheses.
- You can also define your own functions that accept arguments.

Cont..



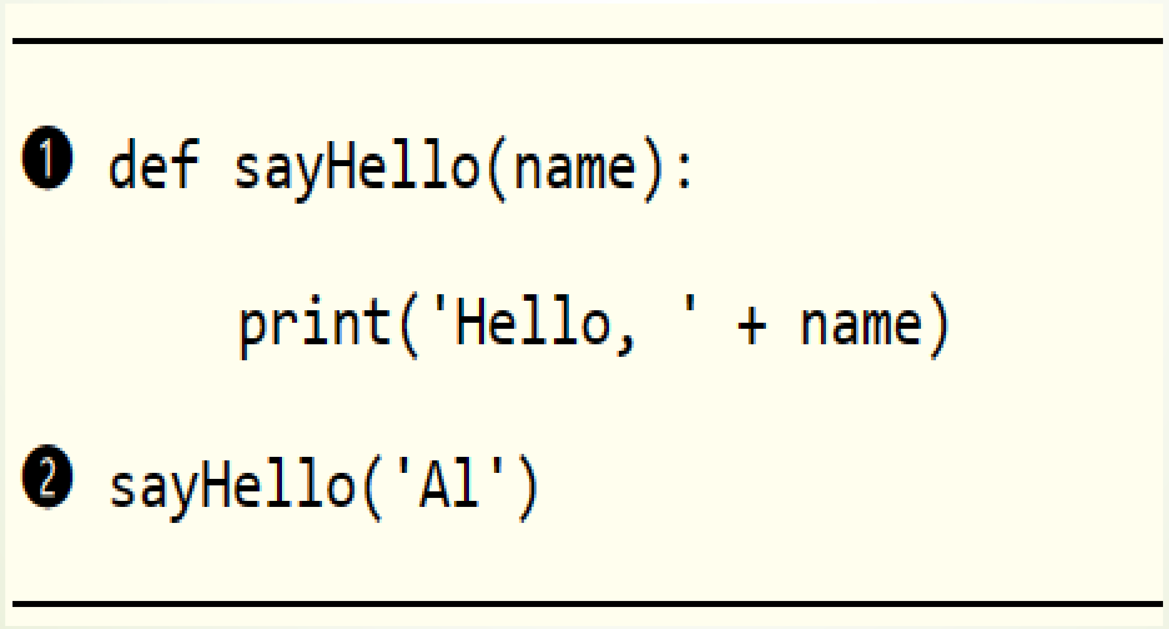
```
❶ def hello(name):
   ❷ print('Hello, ' + name)

❸ hello('Alice')
  hello('Bob')
```

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

```
Hello, Alice
Hello, Bob
```

Define, Call, Pass, Argument, Parameter

- The terms define, call, pass, argument, and parameter can be confusing.
- Let’s look at a code example to review these terms:



```
❶ def sayHello(name):
      print('Hello, ' + name)
❷ sayHello('Al')
```

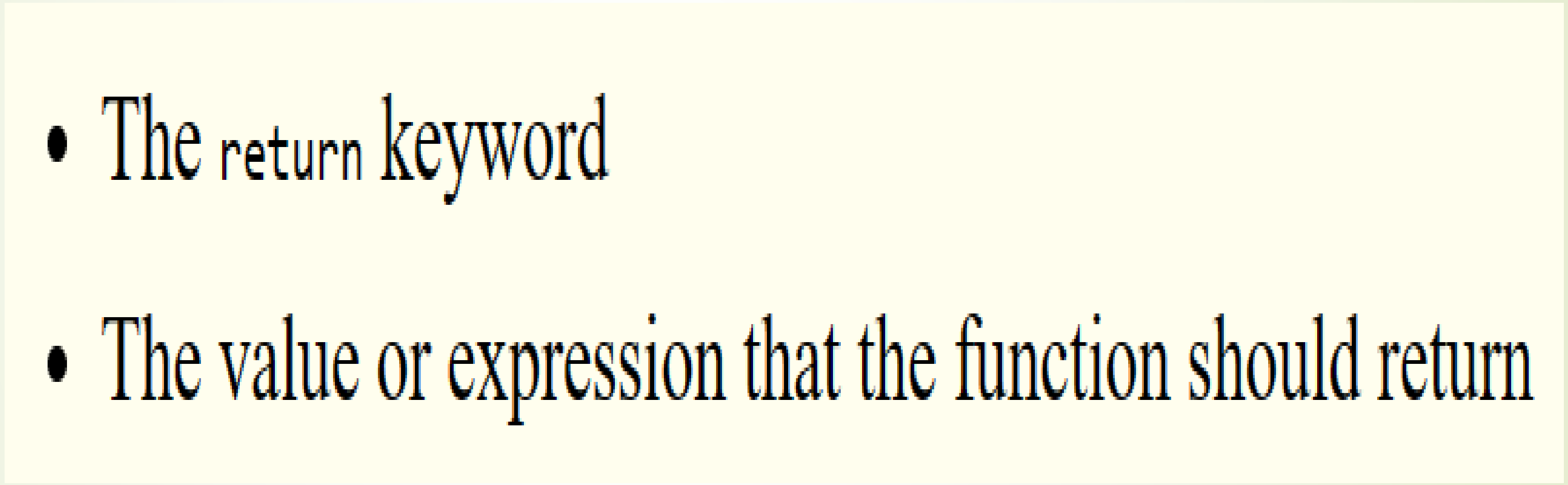
Cont..

- 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 sayHello() function ❶.
- The sayHello('Al') 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 'Al' to the function. A value being passed to a function in a function call is an argument.
- The argument 'Al' is assigned to a local variable named name. Variables that have arguments assigned to them are parameters.

Return Values and return Statements

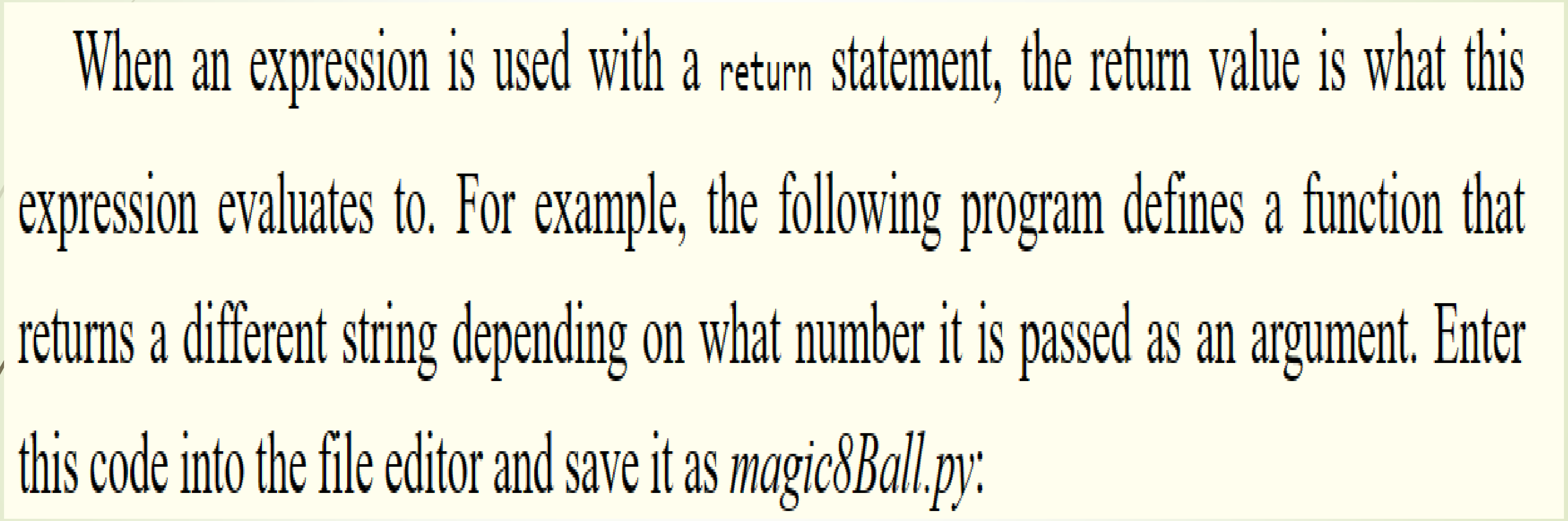
- When you call the len() function and pass it an argument such as 'Hello', the function call evaluates to the integer value 5, which is the length of the string you passed it.
- In general, the value that a function call evaluates to is called the return value of the function.
- When creating a function using the def statement, you can specify what the return value should be with a return statement.
- A return statement consists of the following:

Cont..



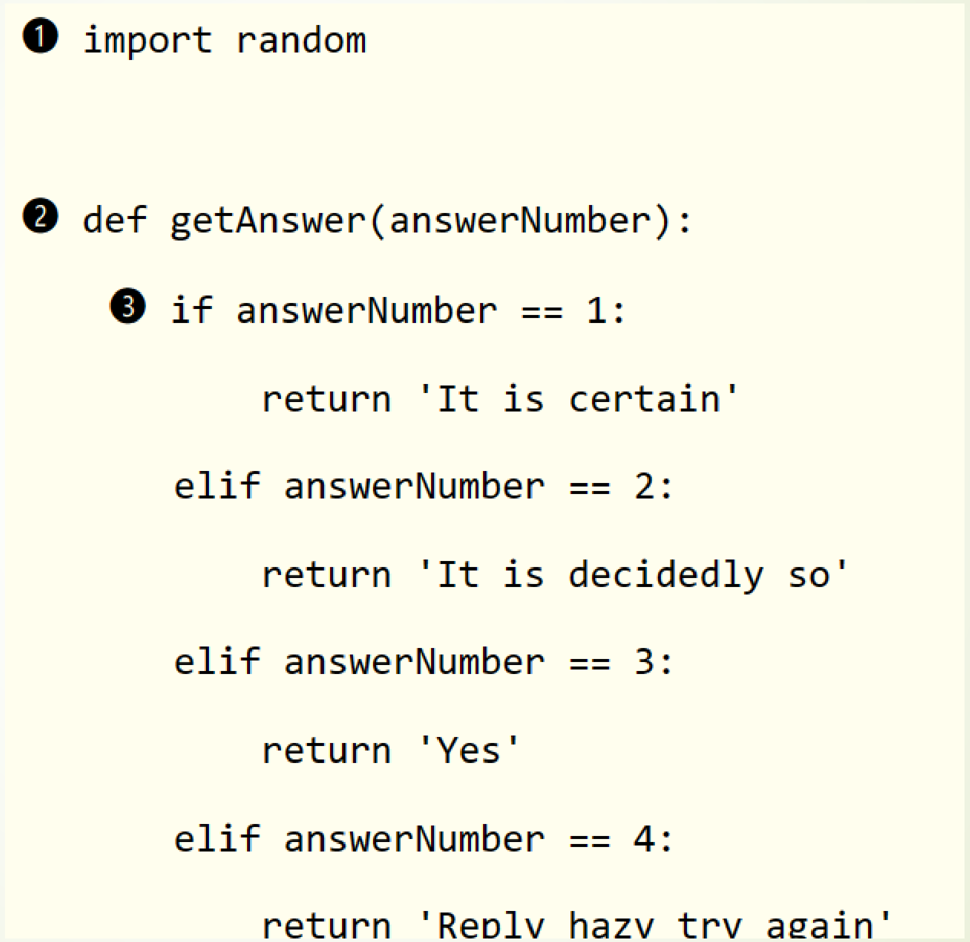
- The return keyword
- The value or expression that the function should return

Cont..



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. Enter this code into the file editor and save it as *magic8Ball.py*:

Cont..

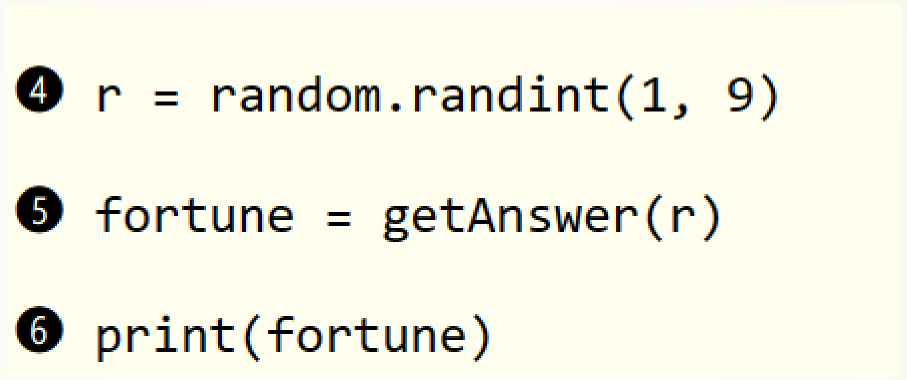


```
❶ import random


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'
```

Cont..

```
        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'
```

Cont..



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

Cont..

- When this program starts, Python first imports the random module ❶.
- Then the getAnswer() function is defined ❷.
- Because the function is being defined (and not called), the execution skips over the code in it.
- Next, the random.randint() function is called with two arguments: 1 and 9 ❹.
- It evaluates to a random integer between 1 and 9 (including 1 and 9 themselves), and this value is stored in a variable named r.

Cont..

- The getAnswer() function is called with r as the argument ❺.
- The program execution moves to the top of the getAnswer() function ❸, and the value r is stored in a parameter named answerNumber.
- Then, depending on the value in answerNumber, the function returns one of many possible string values.
- The program execution returns to the line at the bottom of the program that originally called getAnswer() ❺.
- The returned string is assigned to a variable named fortune, which then gets passed to a print() call ❻ and is printed to the screen.

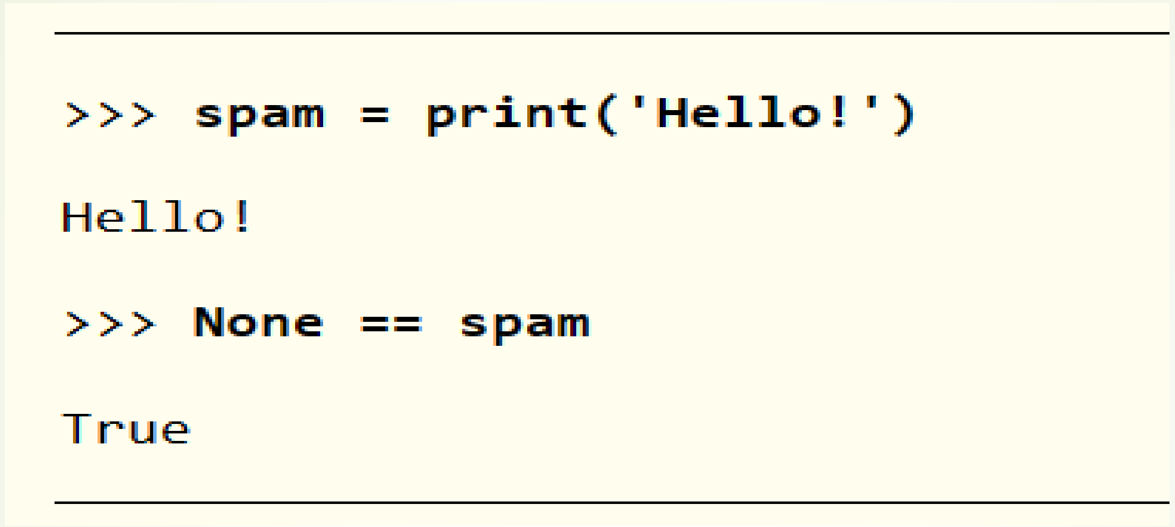
The None Value

- In Python, there is a value called None, which represents the absence of a value.
- The None value is the only value of the NoneType data type. (Other programming languages might call this value null, nil, or undefined.)
- Just like the Boolean True and False values, None must be typed with a capital N.

Cont..

- This value-without-a-value can be helpful when you 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.

Cont..



```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

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

Keyword Arguments and the print() Function

- 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).

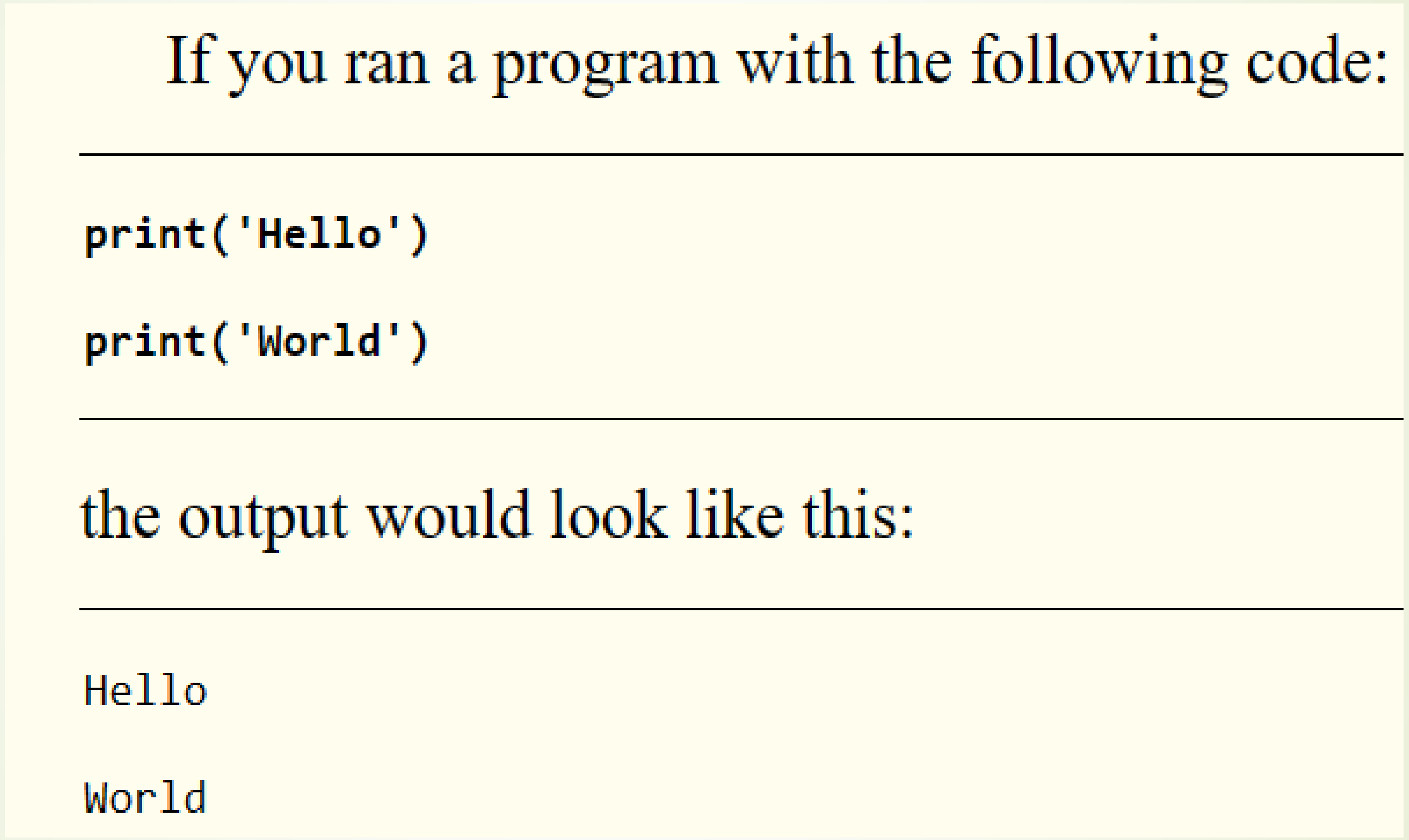
Cont..

- However, 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.

Cont..



If you ran a program with the following code:

```
print('Hello')

print('World')
```

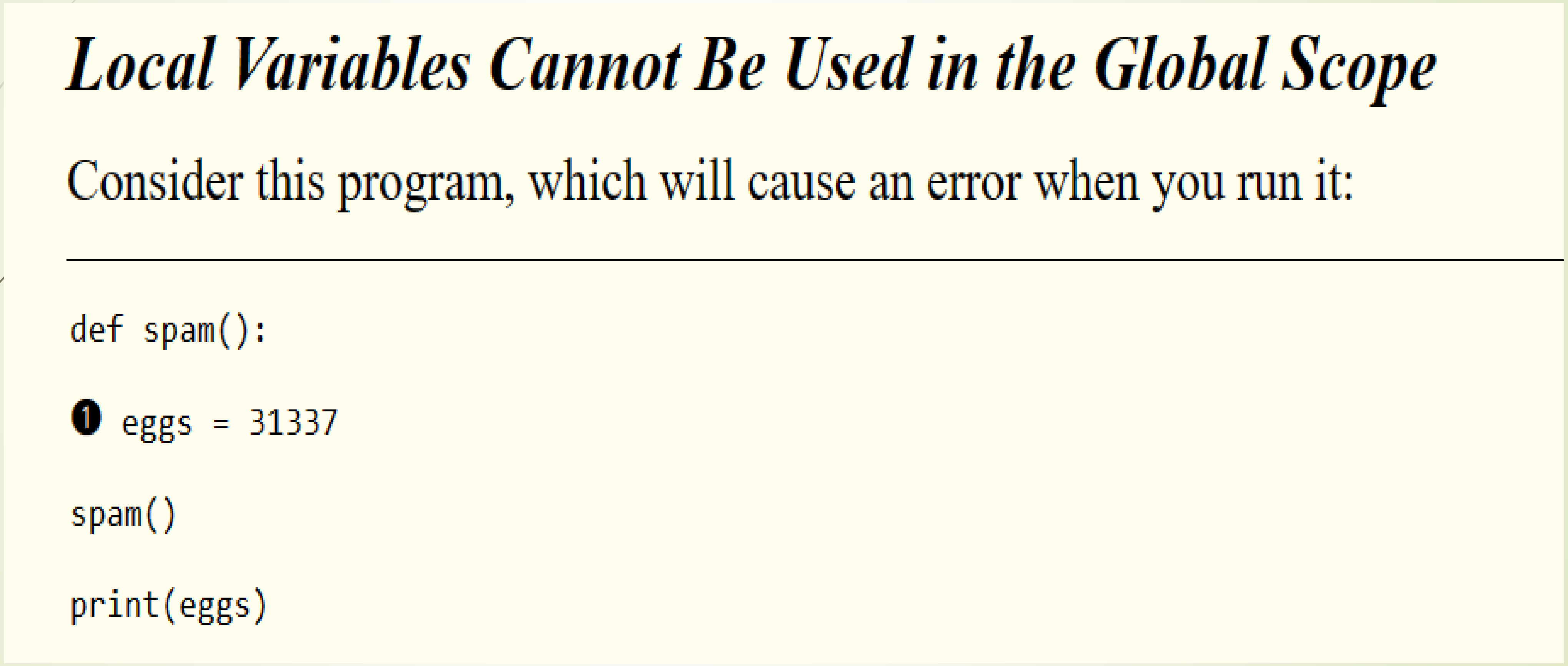
the output would look like this:

```
Hello

World
```

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.

Cont..

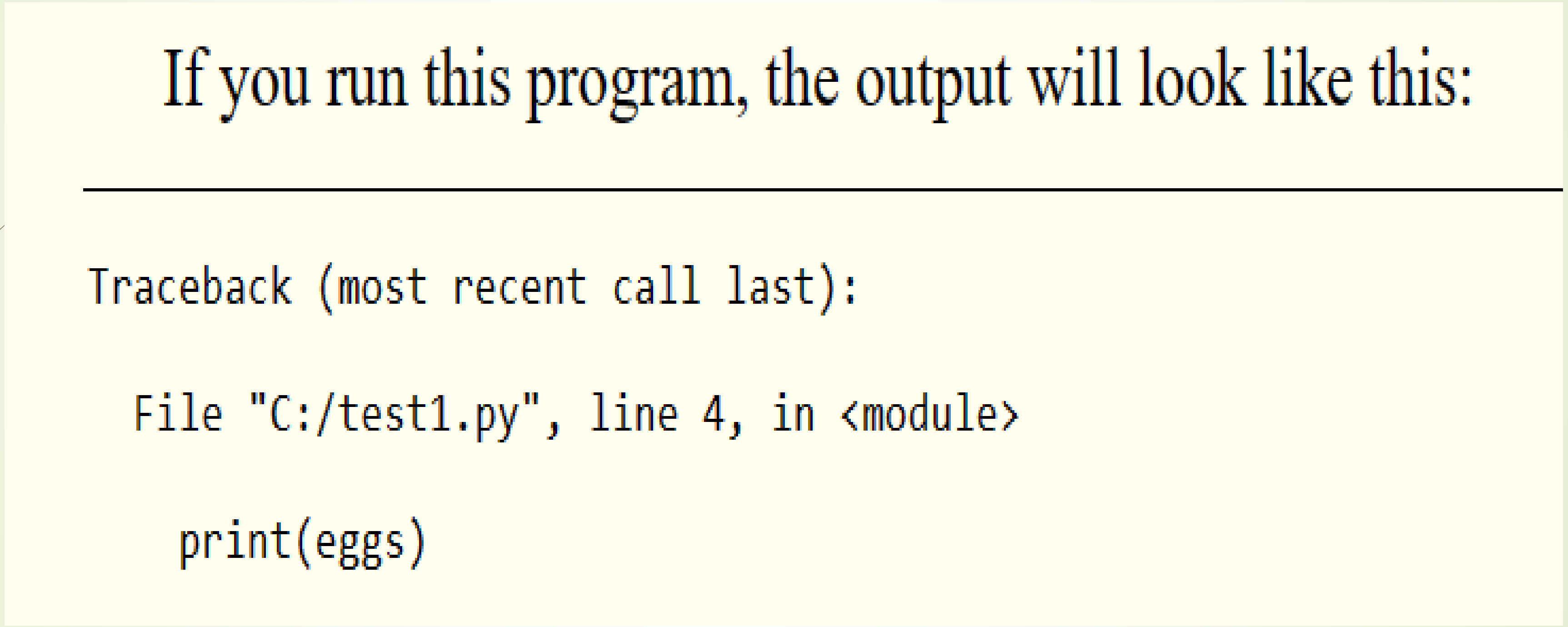


Local Variables Cannot Be Used in the Global Scope

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

```
def spam():
❶ eggs = 31337
spam()
print(eggs)
```

Cont..

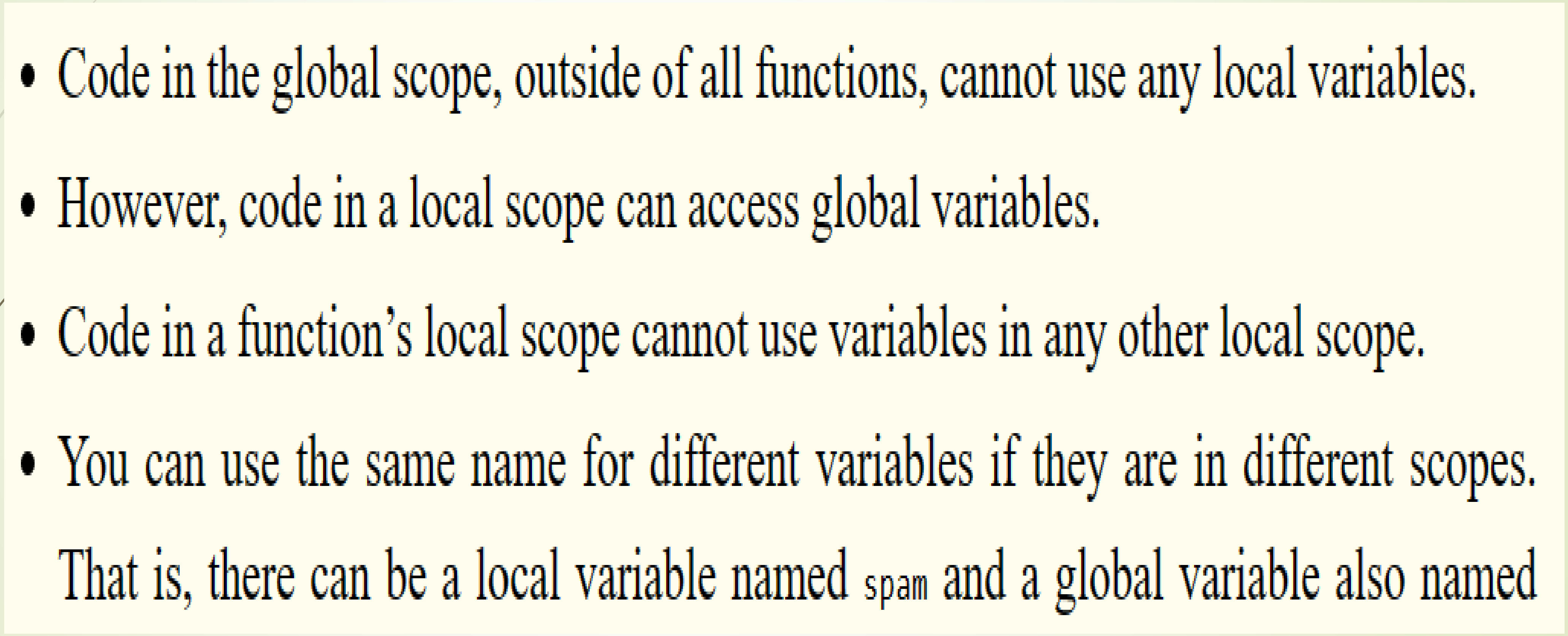

If you run this program, the output will look like this:

```
Traceback (most recent call last):
  File "C:/test1.py", line 4, in <module>
    print(eggs)
```

Cont..

- The error happens because the eggs variable exists only in the local scope created when spam() is called ❶.
- Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs.
- So when your program tries to run print(eggs), Python gives you an error saying that eggs is not defined.

Cont..



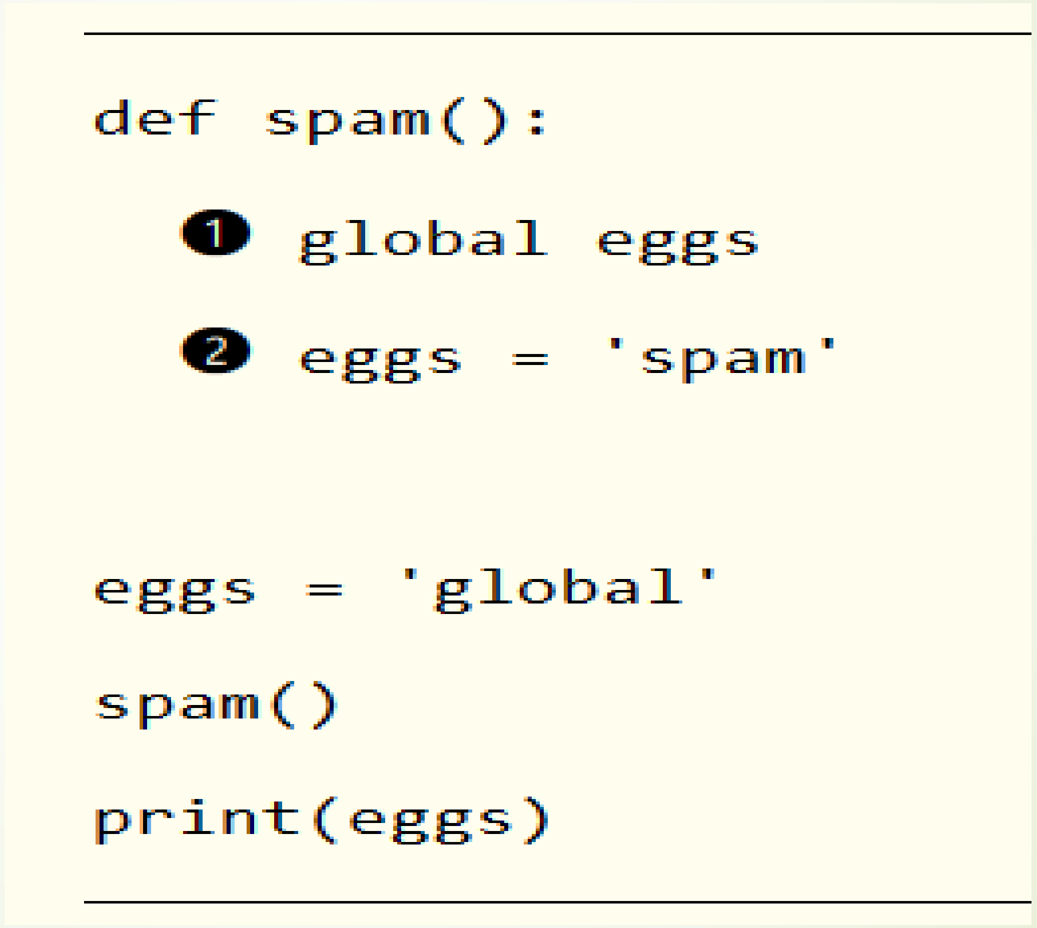
- Code in the global scope, outside of all functions, cannot use any local variables.
- However, code in a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named

The global Statement

- If you need to modify a global variable from within a function, use the global statement.

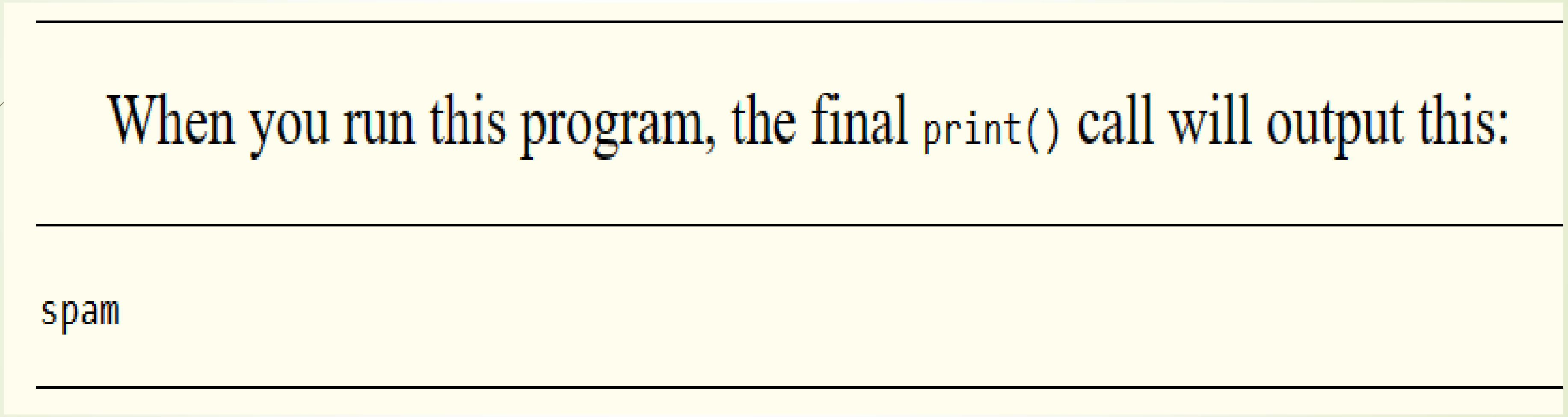
- If you have a line such as global eggs at the top of a function, it tells Python, “In this function, eggs refers to the global variable, so don’t create a local variable with this name.”

Cont..



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


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

Cont..



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

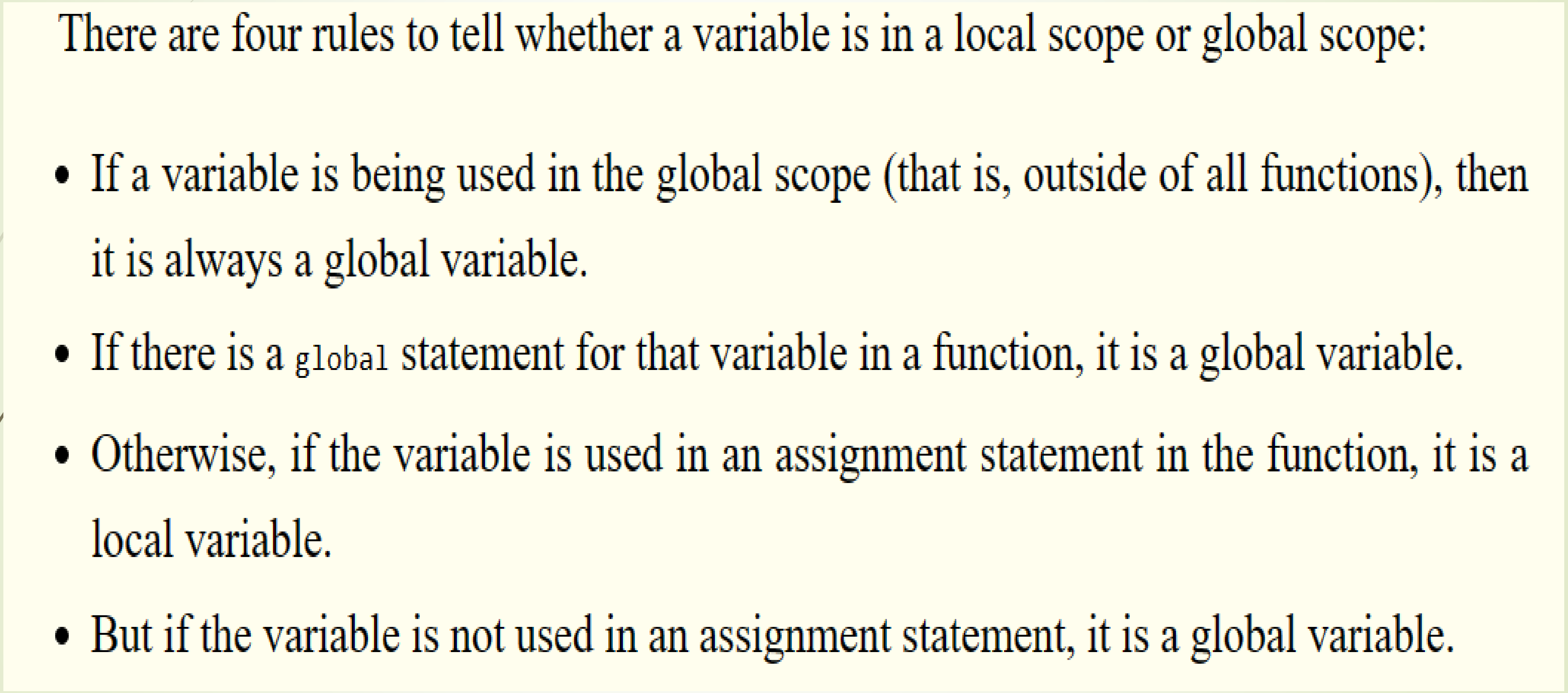
```
spam
```

Cont..

- 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.

Cont..



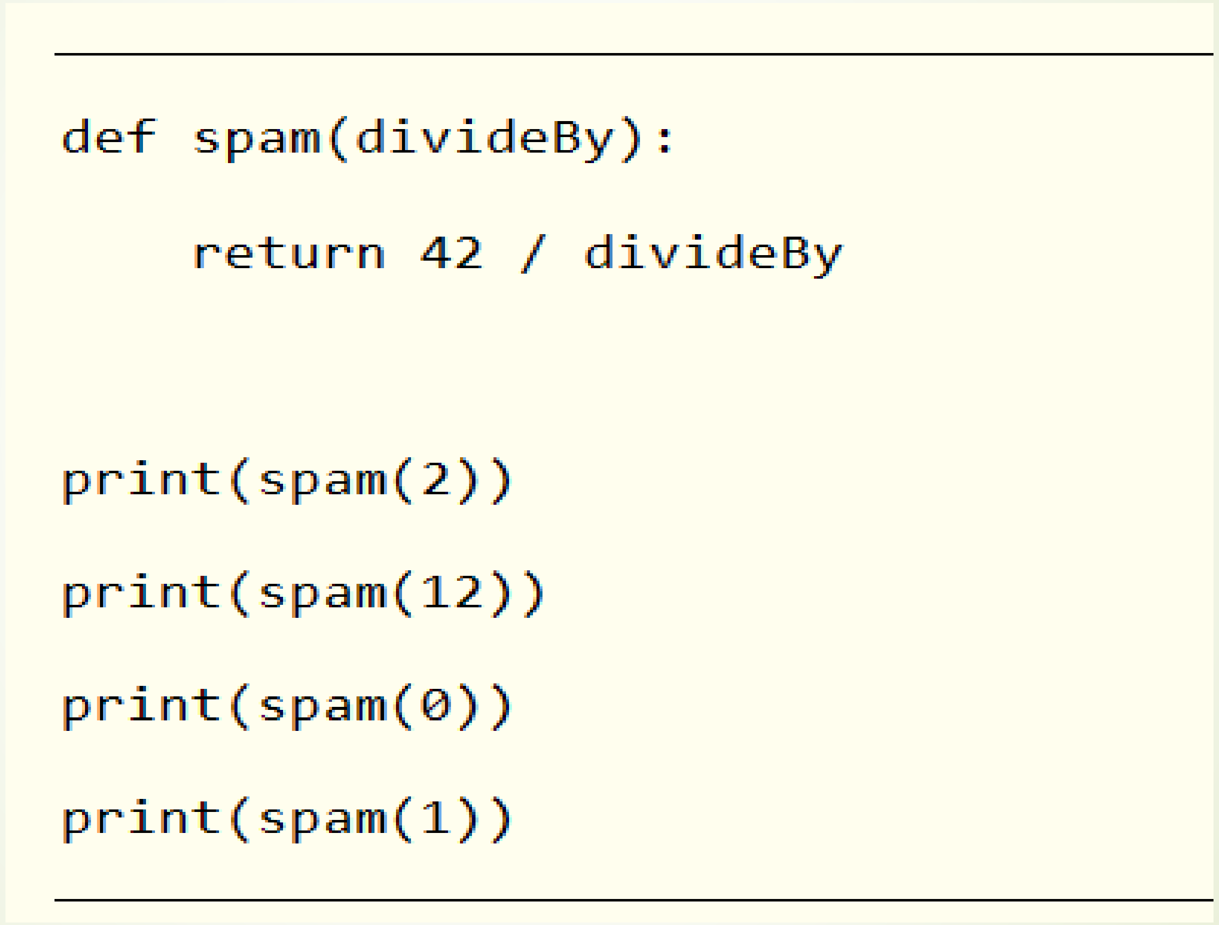
There are four rules to tell whether a variable is in a local scope or global scope:

- If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
- If there is a `global` statement for that variable in a function, it is a global variable.
- Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
- But if the variable is not used in an assignment statement, it is a global variable.

Exception Handling

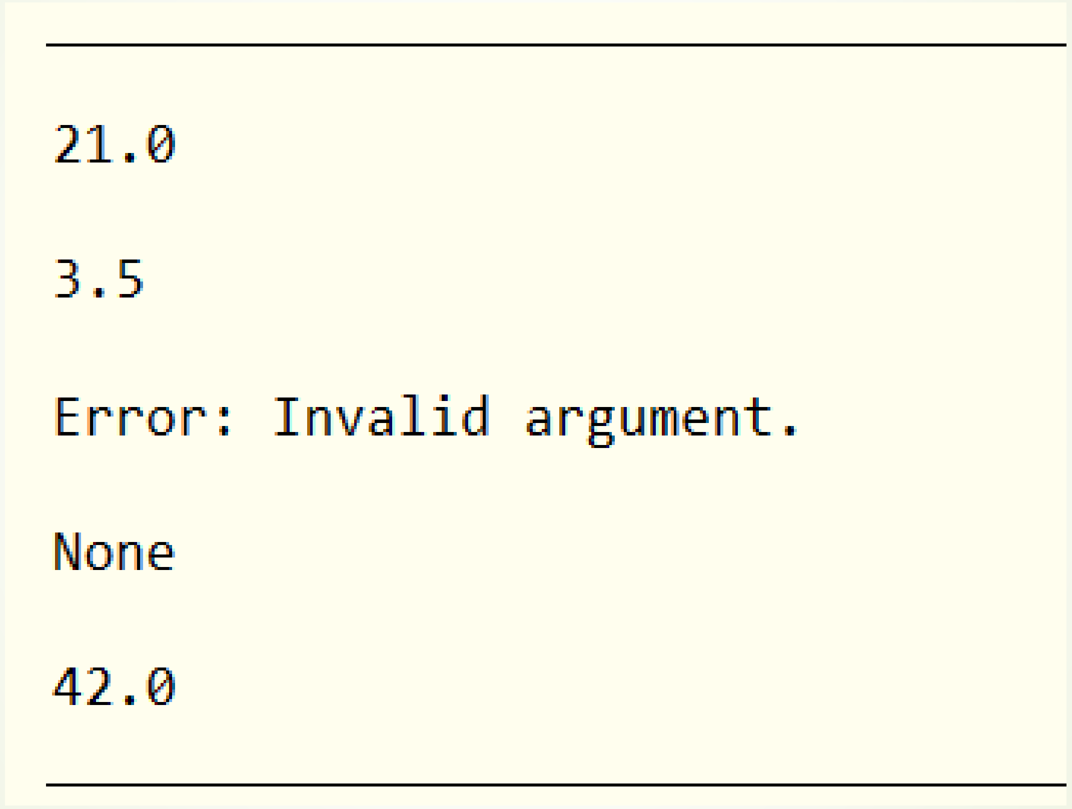
- Right now, getting an error, or *exception*, in your Python program means the entire program will crash.
- You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

Cont..

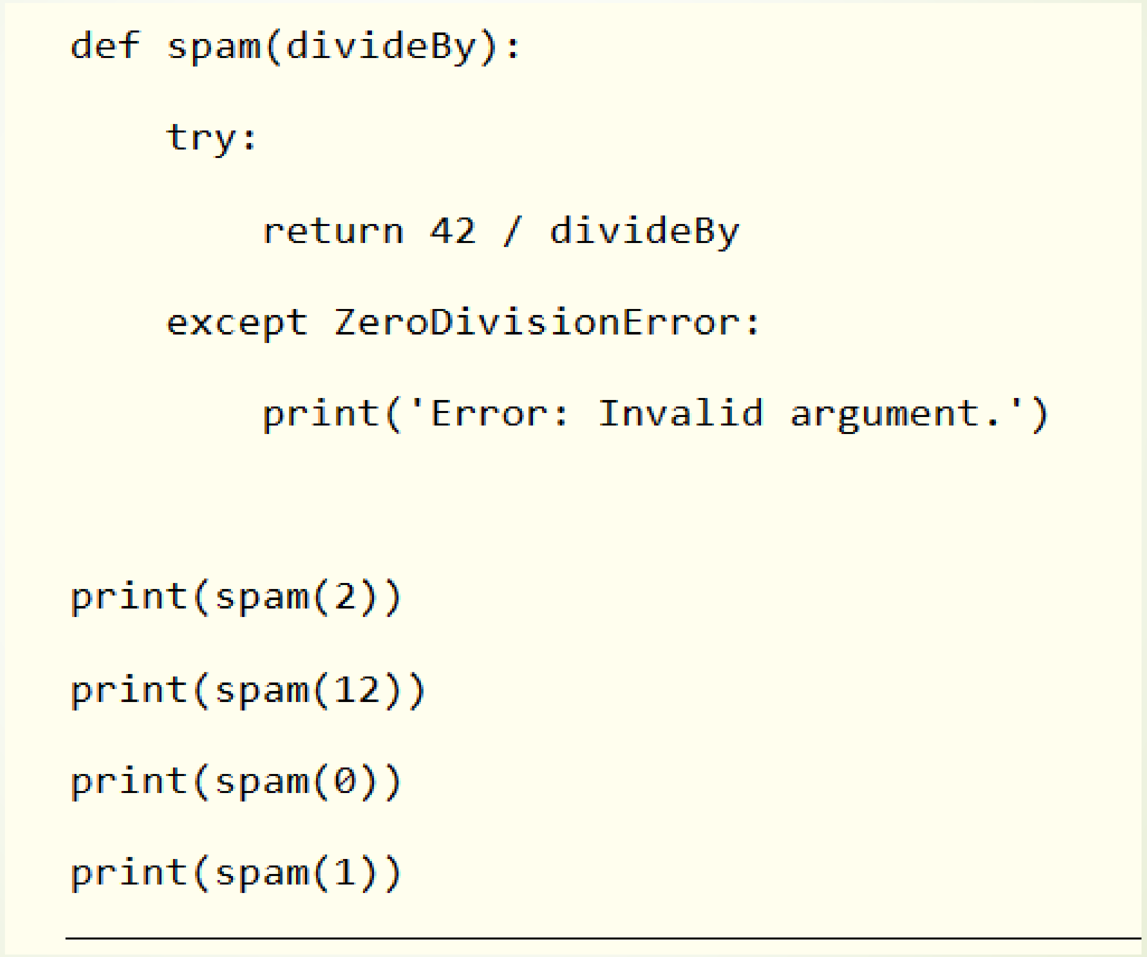


```
def spam(divideBy):
    return 42 / divideBy


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

Cont..


```
21.0

3.5

Error: Invalid argument.

None

42.0
```

It can be solved as:



```python
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))
```

Guess a number:A short program

- import random

- def guess_the_number():
-     secret_number = random.randint(1, 100)
-     attempts = 0

-     print("Welcome to Guess the Number!")
-     print("I have chosen a secret number between 1 and 100. Can you guess it?")

-     while True:

- guess_the_number()

```
guess = int(input("Enter your guess: "))
        attempts += 1

        if guess < secret_number:
            print("Too low! Try again.")
        elif guess > secret_number:
            print("Too high! Try again.")
        else:
            print(f"Congratulations! You guessed the number in {attempts} attempts.")
            break
```