Introduction

What Is Programming?

Programming is simply the act of entering instructions for the computer to perform.
These instructions might crunch some numbers, modify text, look up information in files,
or communicate with other computers over the internet.

All programs use basic instructions as building blocks. Here are a few of the most common ones, in English:

1. “Do this; then do that.”
2. “If this condition is true, perform this action; otherwise, do that action.”
3. “Do this action exactly 27 times.”
4. “Keep doing that until this condition is true.”

What Is Python?

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991.

Why Python for beginners?

- Easy to learn
- Code is 3-5 times shorter than Java
- 5-10 times shorter than C++
- Stepping Stone to Programming universe
- Python's methodologies can be used in a broad range of applications
- Bridging the Gap between abstract computing and real world applications
- Python is used as main programming language to do projects using Raspberry Pi
- Rising Demand for Python Programmers
- Google, Nokia, Disney, Yahoo, IBM use Python
- Open- Source, Object – Oriented, procedural and functional
- 5-10 times shorter than C++
- Not only a Scripting language, also supports Web Development and Database Connectivity

Watch this video to know different areas where Python can be used:
What can you do with Python? https://www.youtube.com/watch?v=hxGB7LU4i1I (https://www.youtube.com/watch?v=hxGB7LU4i1I) (Duration: 3:56)

Python Features:

- Beginners Language
- Extensive Standard Library
- Cross Platform Compatibility

- Interactive Mode
- Portable and Extendable
- Databases and GUI Programming
- Scalable and Dynamic Semantics
- Automatic Garbage Collection

PYTHON BASICS

The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features. Fortunately, you can ignore most of that;
you just need to learn enough to write some handy little programs.

Entering Expressions into the Interactive Shell

In [1]:

```
3+5
```

Out[1]:

8

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.

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

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

The order of operations (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:

In [2]:

```
2*(3//5)
```

Out[2]:

```
0
```

In [3]:

```
(2 + 3) * 6
```

Out[3]:

```
30
```

In [4]:

```
 48565878 * 578453
```

Out[4]:

```
28093077826734
```

In [5]:

```
2 ** 8
```

Out[5]:

```
256
```

In [6]:

```
23 / 7
```

Out[6]:

```
3.2857142857142856
```

In [7]:

```
23 // 7
```

Out[7]:

```
3
```

In [8]:

```
23 % 7
```

Out[8]:

```
2
```

In [9]:

```
2      +           2
```

Out[9]:

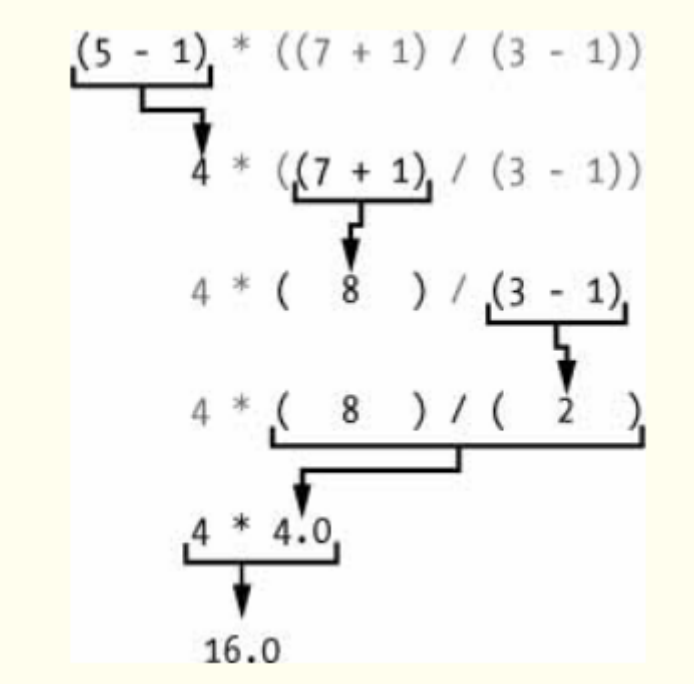
```
4
```

In [10]:

```
3**3/9*(1+2)
```

Out[10]:

```
9.0
```



The Integer, Floating-Point, and String Data Types

A data type is a category for values, and every value belongs to exactly one data type.
The most common data types in Python are listed in Table 1-2.

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'

Python programs can also have text values called strings, or strs (pronounced “stirs”).
Always surround your string in single quote (') characters (as in 'Hello' or 'Goodbye cruel world!')
so Python knows where the string begins and ends.

In [11]:

```
'Hello, world!'
```

Out[11]:

```
'Hello, world!'
```

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:

In [12]:

```
'Alice' + 'Bob'
```

Out[12]:

```
'AliceBob'
```

In [13]:

```
'Alice' + 42
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-4e9a9e71481a> in <module>
----> 1 'Alice' + 42

TypeError: can only concatenate str (not "int") to str
```

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.

In [ ]:

```
'Alice' * 5
```

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:

In [ ]:

```
 'Alice' * 'Bob'
```

In [ ]:

```
'Alice' * 5.0
```

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.

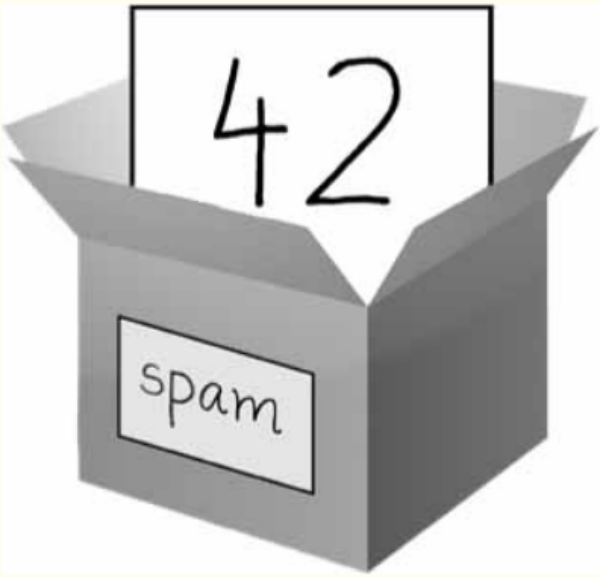


Figure 1-1: spam = 42 is like telling the program, "The variable spam now has the integer value 42 in it."

Variable Names

A good variable name describes the data it contains. Though you can name your variables almost anything, Python does have some naming restrictions. Table 1-3 has examples of legal variable names.
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.

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)

Variable names are case-sensitive, meaning that spam, SPAM, Spam, and sPaM are four different variables. Though Spam is a valid variable you can use in a program, it is a Python convention to start your variables with a lowercase letter.

Your First Program

In [ ]:

```
 # 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.')
```

In [ ]:

```
"hello"+'2'
```

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.

Comments

The following line is called a comment.

In [ ]:

```
# 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.

In [ ]:

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

The input() Function

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

In [ ]:

```
myName = input()
```

Printing the User's Name

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

In [14]:

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

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-14-2559d4ddf736> in <module>
----> 1 print('It is good to meet you, ' + myName)

NameError: name 'myName' is not defined
```

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.

In [ ]:

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

In [ ]:

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

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:

In [ ]:

```
str(29)
```

In [ ]:

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

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.

In [ ]:

```
str(0)
```

In [ ]:

```
str(-3.14)
```

In [15]:

```
int('42')
```

Out[15]:

```
42
```

In [16]:

```
int(1.25)
```

Out[16]:

```
1
```

In [17]:

```
float('3.14')
```

Out[17]:

```
3.14
```

In [18]:

```
 float(10)
```

Out[18]:

```
10.0
```

The str() function is handy when you have an integer or float that you want to concatenate to a string.
The int() function is also helpful if you have a number as a string value that you want to use in some

mathematics.
For example, the input() function always returns a string, even if the user enters a number.
Enter spam = input() into the interactive shell and enter 101 when it waits for your text.

In [ ]:

```
spam = input()
```

In [ ]:

```
spam
```

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

In [ ]:

```
spam = int(spam)
```

In [ ]:

```
spam
```

Now you should be able to treat the spam variable as an integer instead of a string.

In [ ]:

```
spam * 10 / 5
```

Note that if you pass a value to int() that it cannot evaluate as an integer, Python will display an error message.

In [ ]:

```
int('99.99')
```

In [ ]:

```
int('twelve')
```

The int() function is also useful if you need to round a floating-point number down.

In [ ]:

```
int(7.7)
```

In [ ]:

```
 int(7.7) + 1
```

You used the int() and str() functions in the last three lines of your program to get a value of the appropriate data type for the code.

In [ ]:

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

TEXT AND NUMBER EQUIVALENCE

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

>>> **42 == '42'**

False

>>> **42 == 42.0**

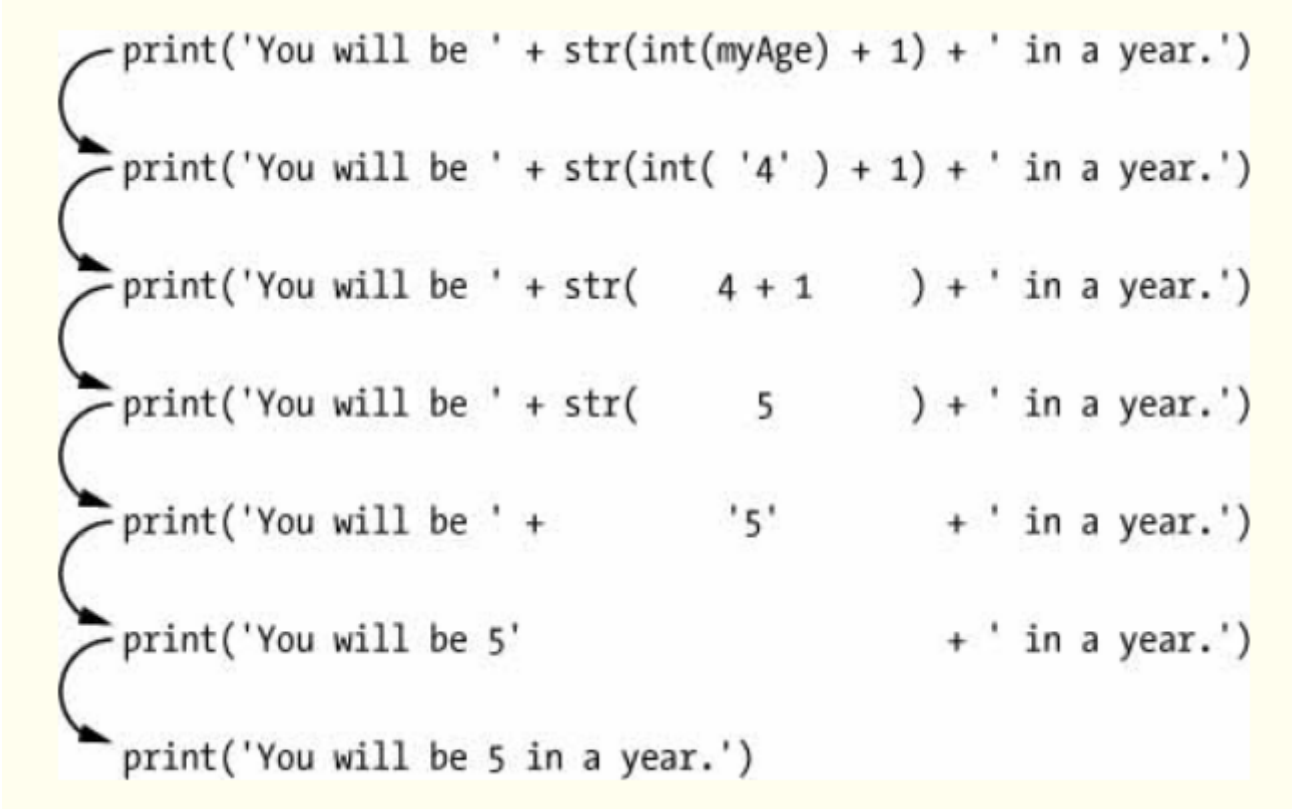
True

>>> **42.0 == 0042.000**

True

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

Let's say the user enters the string '4' for myAge. The string '4' is converted to an integer, so you can add one to it.
The result is 5. The str() function converts the result back to a string, so you can concatenate it with the second string,
'in a year.', to create the final message. These evaluation steps would look something like the following:



Summary by Quiz

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, so I'll provide flowchart versions of the code discussed in this chapter.
Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.

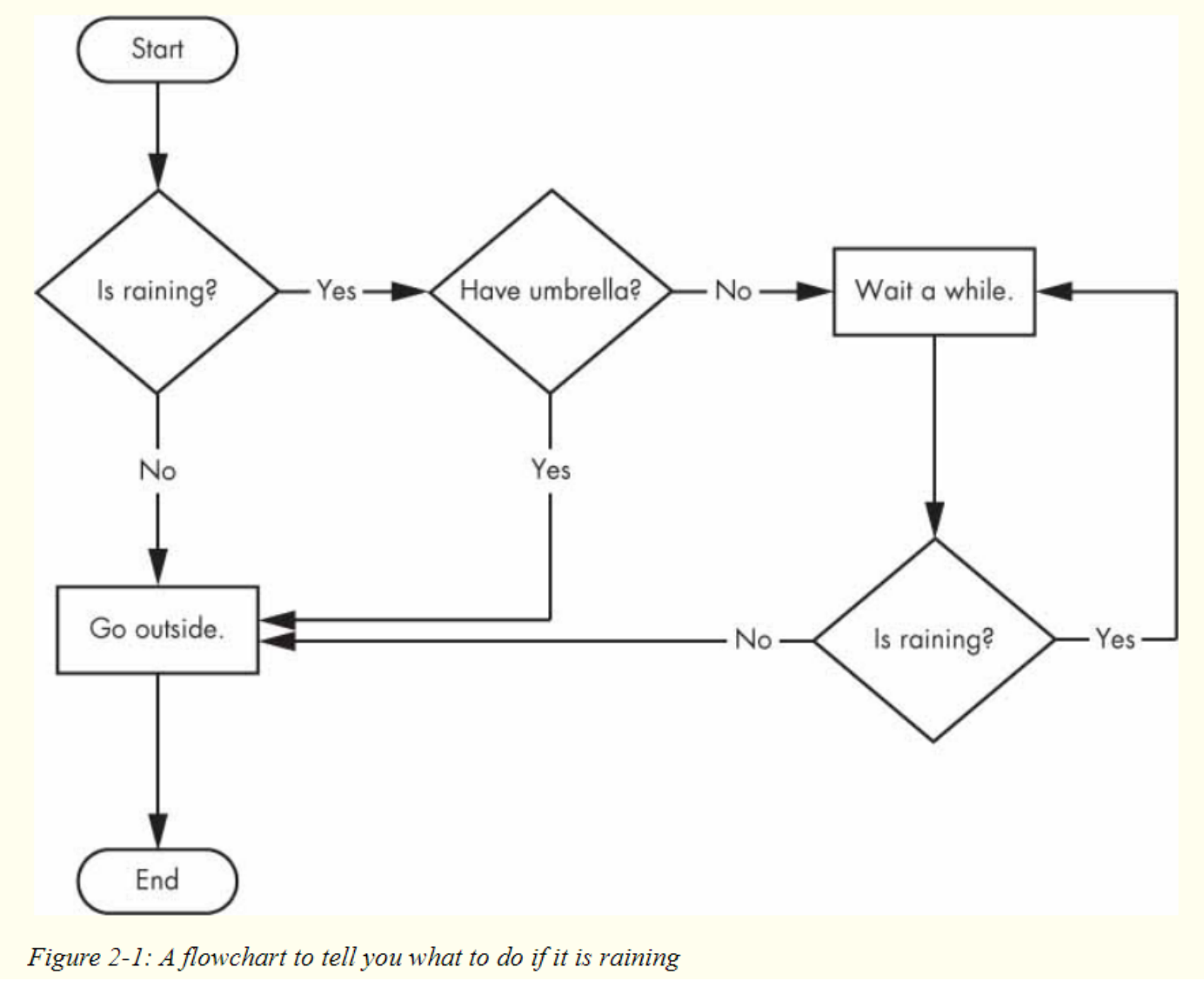


Figure 2-1: A flowchart to tell you what to do if it is raining

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.

In [ ]:

```
spam = True
```

In [ ]:

```
spam
```

In [ ]:

```
true
```

In [ ]:

```
True = 2 + 2
```

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

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

In [ ]:
```
42 == 42
```

In [ ]:
```
42 == 99
```

In [ ]:
```
2 != 3
```

In [ ]:
```
2 != 2
```

In [ ]:
```
'hello' == 'hello'
```

In [ ]:

```
'hello' == 'Hello'
```

In [ ]:

```
'dog' != 'cat'
```

In [ ]:

```
True == True
```

In [ ]:

```
True != False
```

In [ ]:

```
42 == 42.0
```

In [ ]:

```
42 == '42'
```

The <, >, <=, and >= operators, on the other hand, work properly only with integer and floating-point values.

In [ ]:

```
42 < 100
```

In [ ]:

```
42 > 100
```

In [ ]:

```
42 < 42
```

In [ ]:

```
eggCount = 42
```

In [ ]:

```
eggCount <= 42
```

In [ ]:

```
myAge = 29
```

In [ ]:

```
myAge >= 10
```

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.
Enter some expressions using and into the interactive shell to see it in action.

In [ ]:

```
True and True
```

In [ ]:

```
True and 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

You can see every possible outcome of the or operator in its truth table, shown in Table 2-3.

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

Unlike and and or, the not operator operates on only one Boolean value (or expression). This makes it a unary operator.
The not operator simply evaluates to the opposite Boolean value.

In [ ]:

```
not True
```

In [ ]:

```
not False
```

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

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

Mixing Boolean and Comparison Operators

While expressions like 4 < 5 aren't Boolean values, they are expressions that evaluate down to Boolean values.
Try entering some Boolean expressions that use comparison operators into the interactive shell.

In [ ]:

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

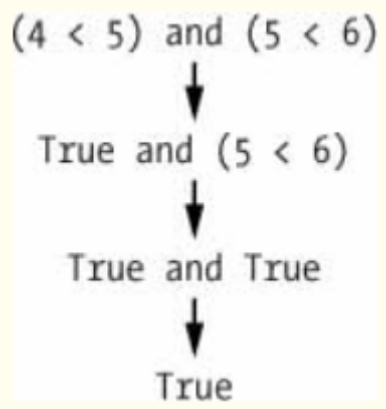
In [ ]:

```
(4 < 5) and (9 < 6)
```

In [ ]:

```
(1 == 2) or (2 == 2)
```

The computer will evaluate the left expression first, and then it will evaluate the right expression.
When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value.
You can think of the computer's evaluation process for (4 < 5) and (5 < 6) as the following:



In [ ]:

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

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

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.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program,
shown here:

In [ ]:

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

Flow Control Statements

The statements represent the diamonds in the flowchart are the actual decisions your programs will make.

if Statements

The most common type of flow control statement is the if statement. An if statement's clause will execute if the statement's condition is True.
The clause is skipped if the condition is False.

In plain English, an if statement could be read as, “If this condition is true, execute the code in the clause.”
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)

For example, let’s say you have some code that checks to see whether someone’s name is Alice.

In [ ]:

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

All flow control statements end with a colon and are followed by a new block of code (the clause). This if statement’s clause is the block with print('Hi, Alice.').
Figure 2-2 shows what a flowchart of this code would look like.

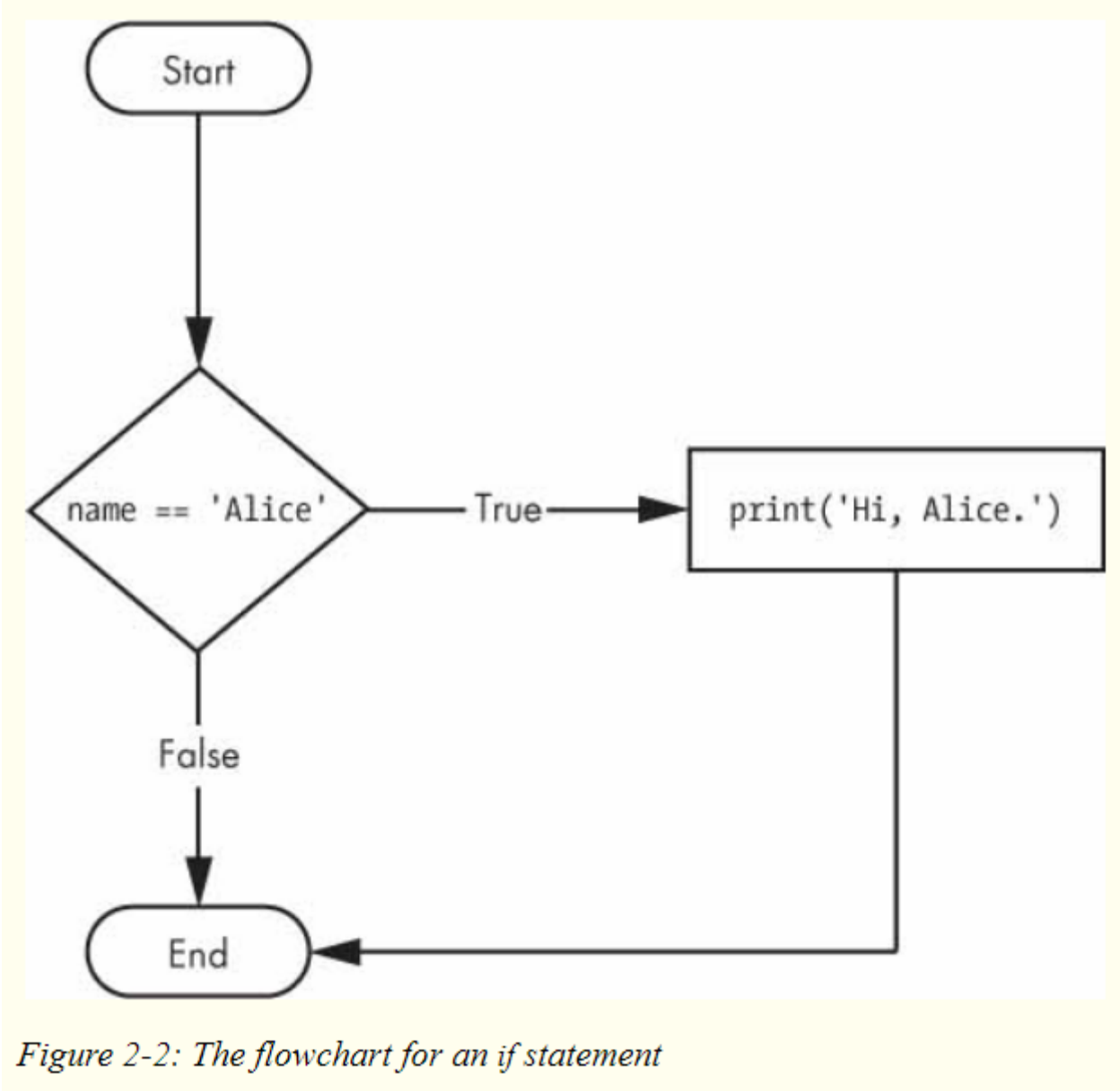


Figure 2-2: The flowchart for an if statement

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, and in code, 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)

Returning to the Alice example, let's look at some code that uses an else statement to offer a different greeting if the person's name isn't Alice.

In [ ]:

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

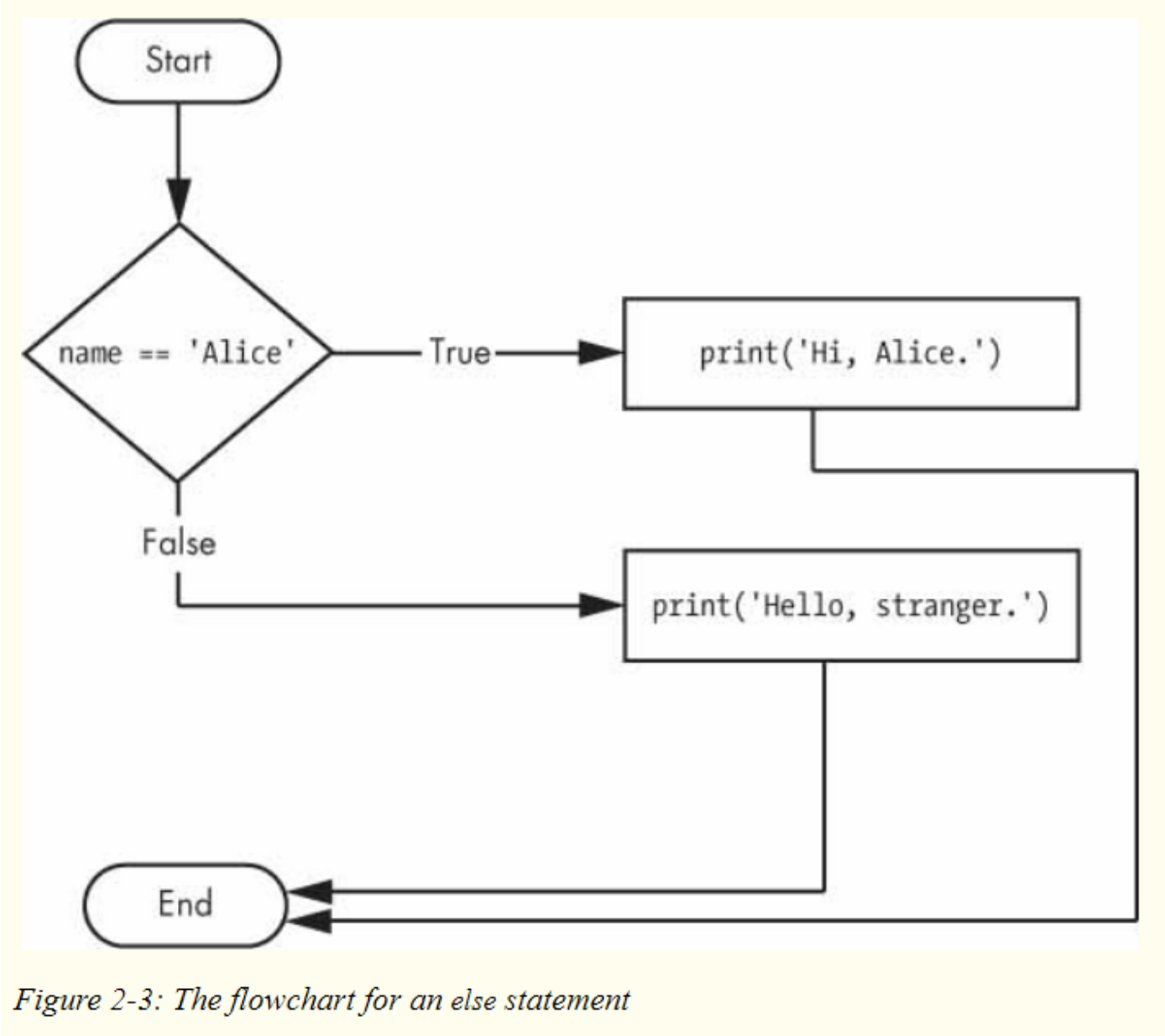


Figure 2-3: The flowchart for an else statement

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:

- 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.

In [ ]:

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

This time, you check the person's age, and the program will tell them something different if they're younger than 12. You can see the flowchart for this in Figure 2-4.

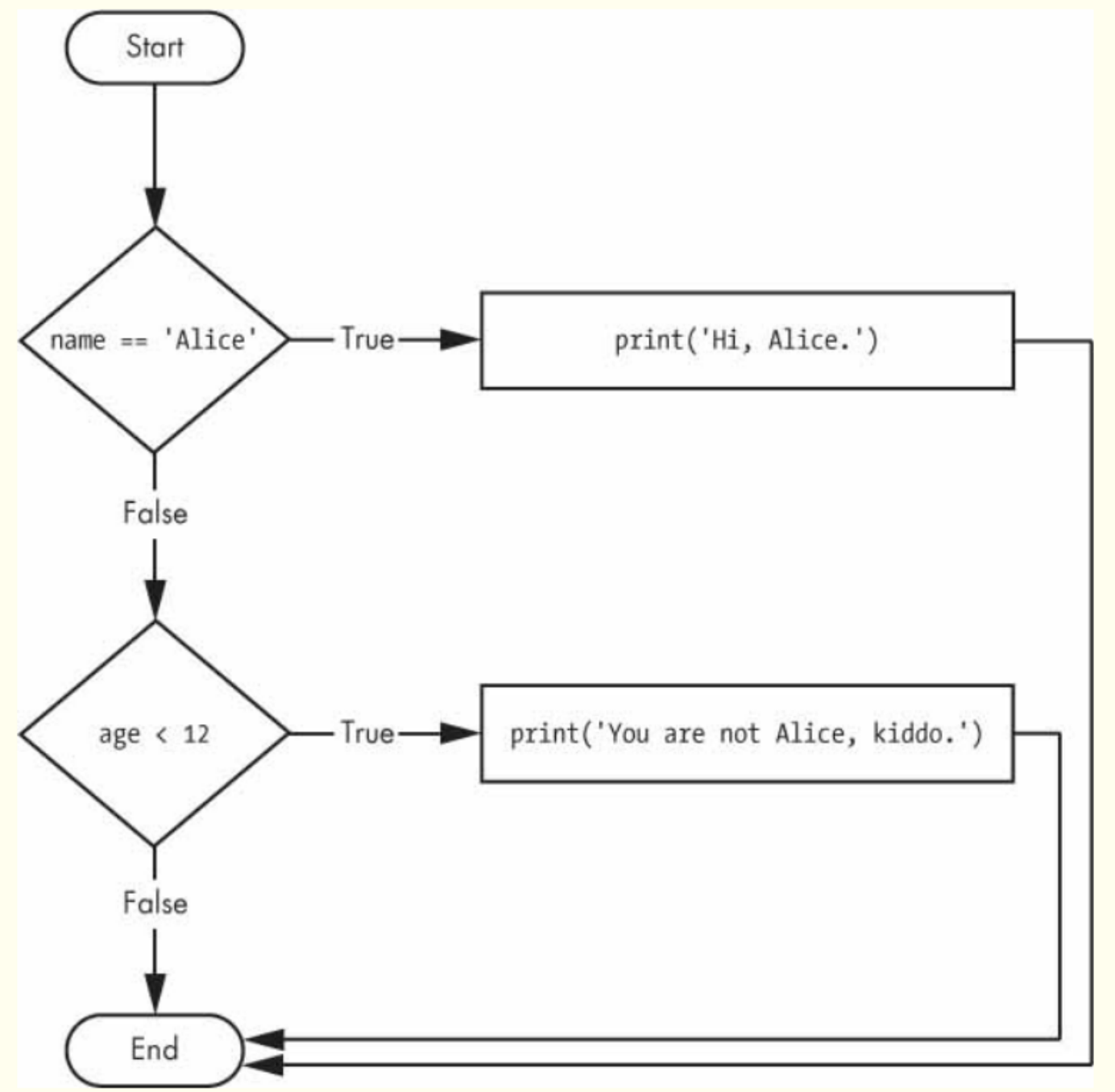


Figure 2-4: The flowchart for an elif statement

When there is a chain of elif statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be True,
the rest of the elif clauses are automatically skipped. For example, open a new file editor window and enter the following code

In [ ]:

```
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.')
```

The order of the elif statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the
elif clauses are automatically skipped once a True condition has been found, so if you swap around some of the clauses in
the above code you run into a problem. Change the code to look like the following:

In [ ]:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 100:
    print('You are not Alice, grannie.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
```

Figure 2-6 shows the flowchart for the previous code. Notice how the diamonds for age > 100 and age > 2000 are swapped.

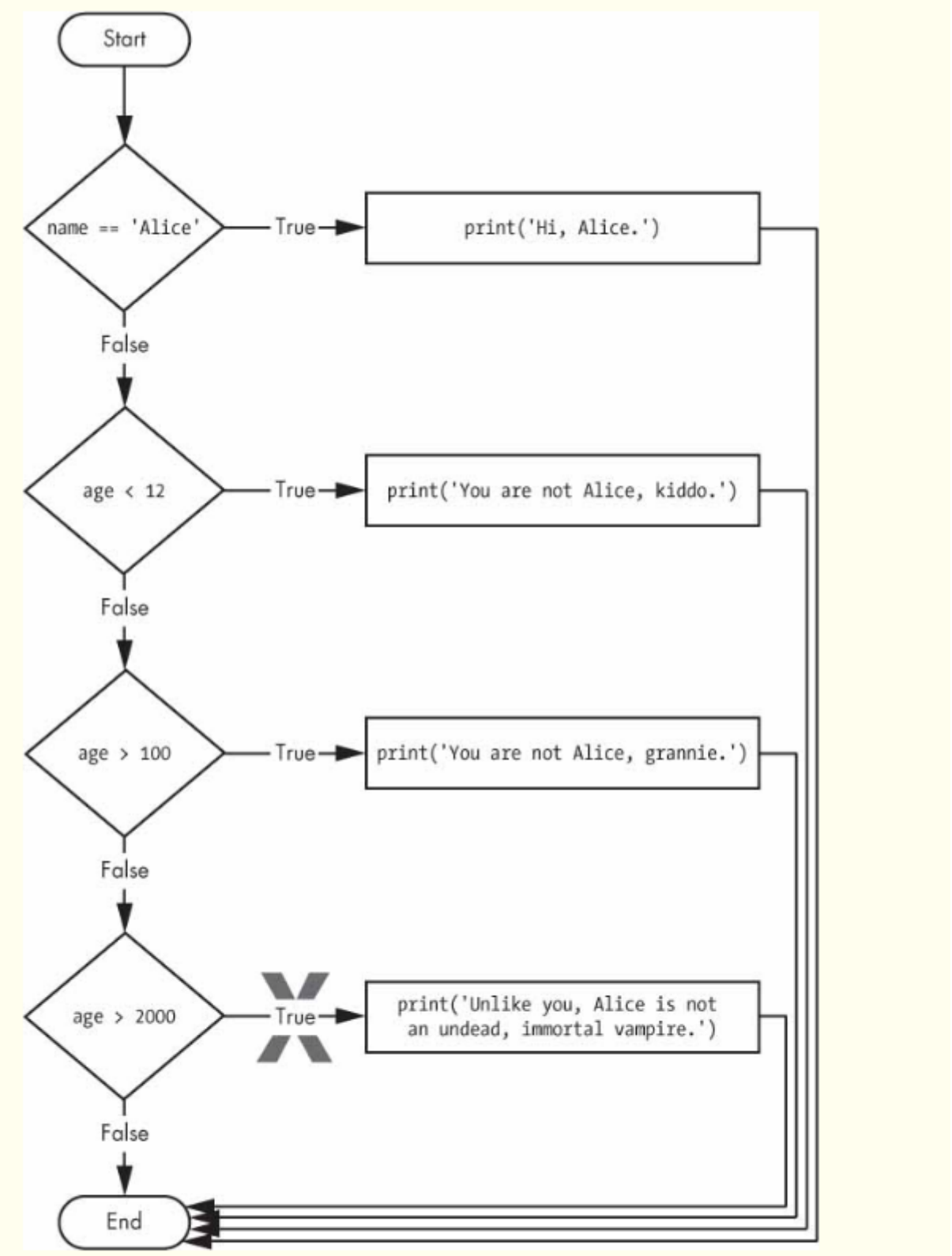


Figure 2-6: The flowchart for the vampire2.py *program. The X path will logically never happen, because if age were greater than 2000, it would have already been greater than 100.*

Optionally, you can have an else statement after the last elif statement. In that case, it is guaranteed that at least one
(and only one) of the clauses will be executed. If the conditions in every if and elif statement are False, then the else
clause is executed. For example, let's re-create the Alice program to use if, elif, and else clauses.

In [ ]:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

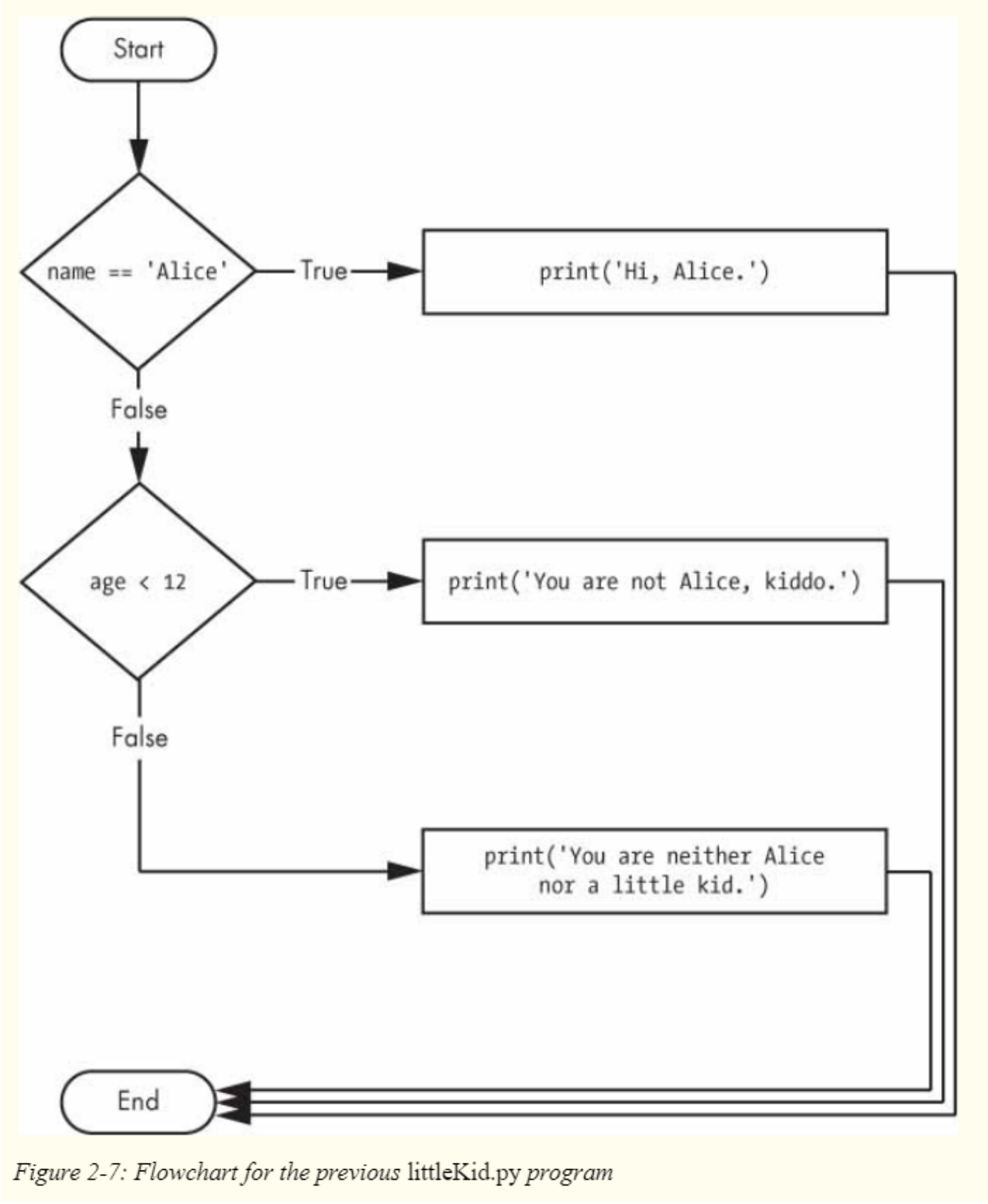


Figure 2-7: Flowchart for the previous littleKid.py *program*

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:

- 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)

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.

Let's look at an if statement and a while loop that use the same condition and take the same actions based on that condition.
Here is the code with an if statement:

In [ ]:

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

Here is the code with a while statement:

In [ ]:

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

Take a look at the flowcharts for these two pieces of code, Figures 2-8 and 2-9, to see why this happens.

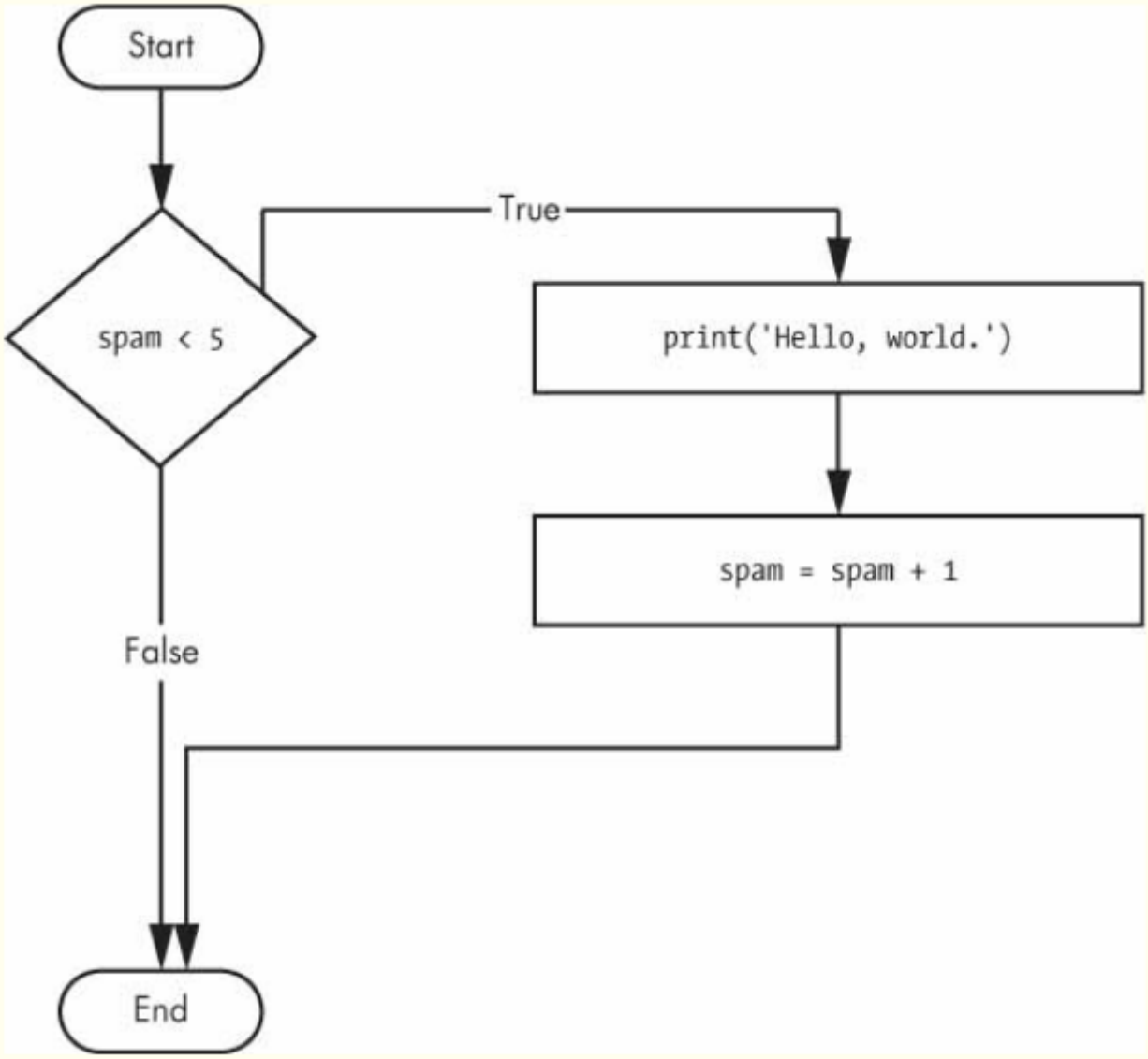


Figure 2-8: The flowchart for the if statement code

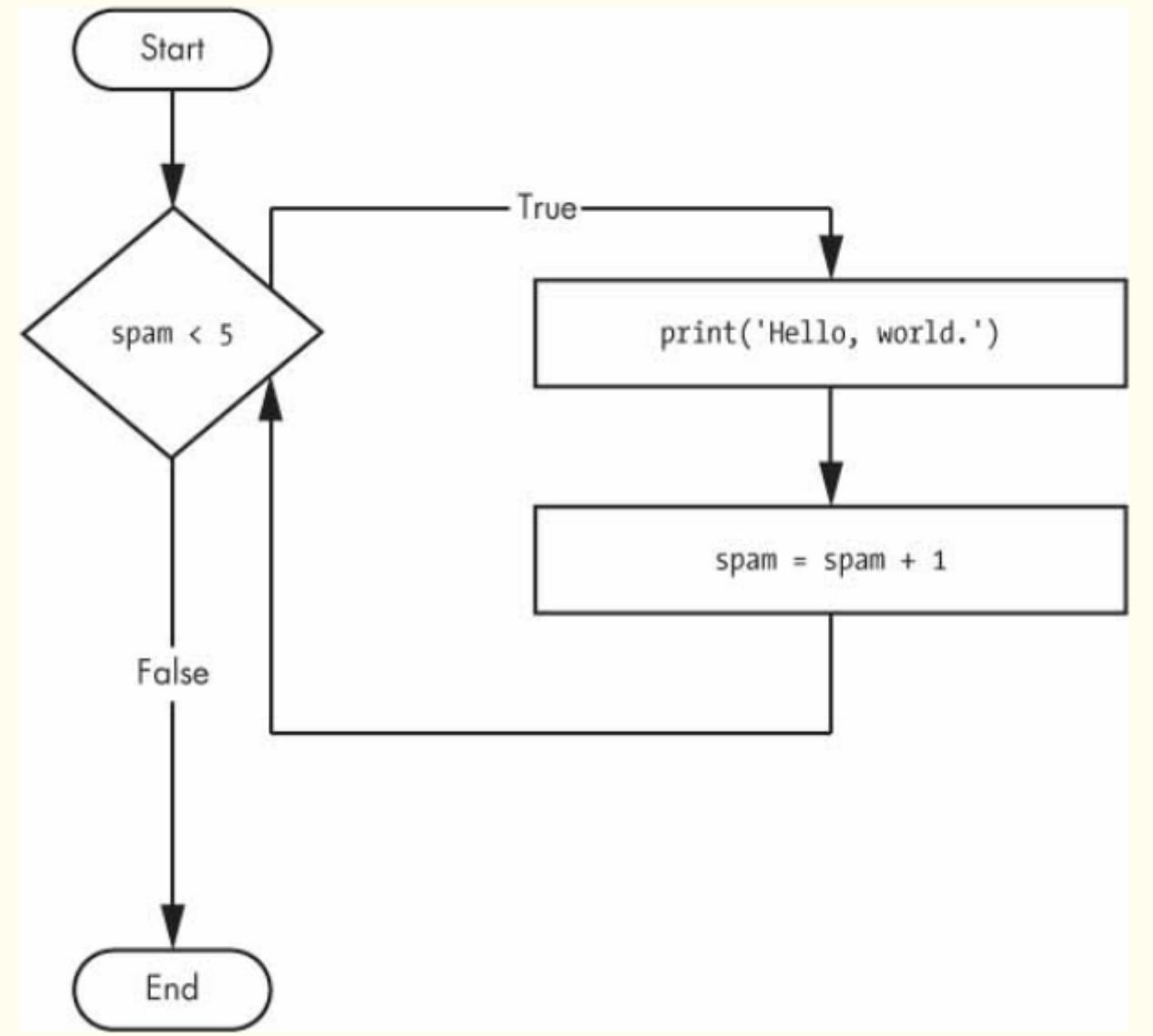


Figure 2-9: The flowchart for the while statement code

An Annoying while Loop

Here's a small example program that will keep asking you to type

In [ ]:

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

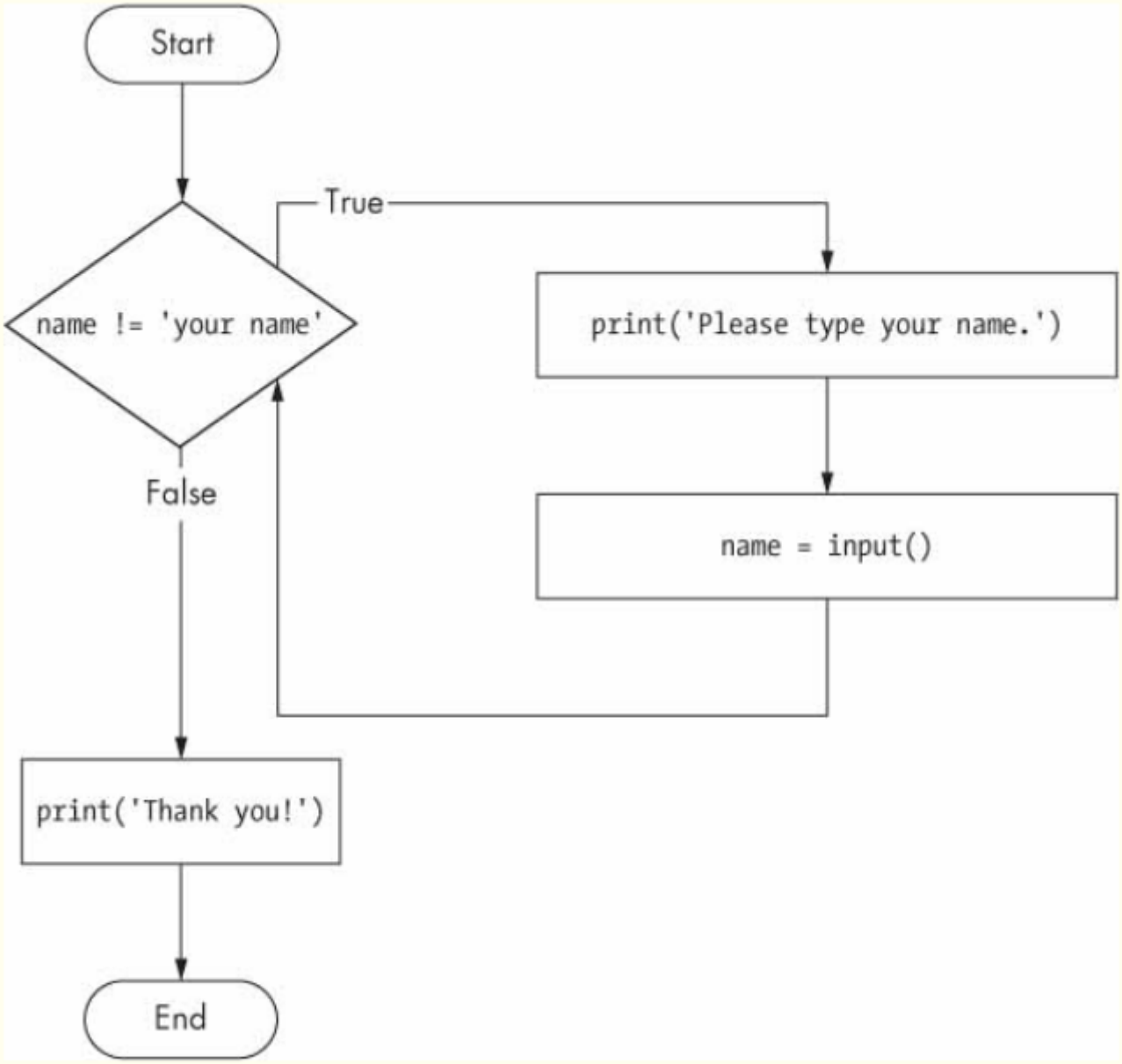


Figure 2-10: A flowchart of the yourName.py *program*

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.

Pretty simple, right? Here's a program that does the same thing as the previous program, but it uses a break statement to
escape the loop. Enter the following code

In [ ]:

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

See Figure 2-11 for this program's flowchart.

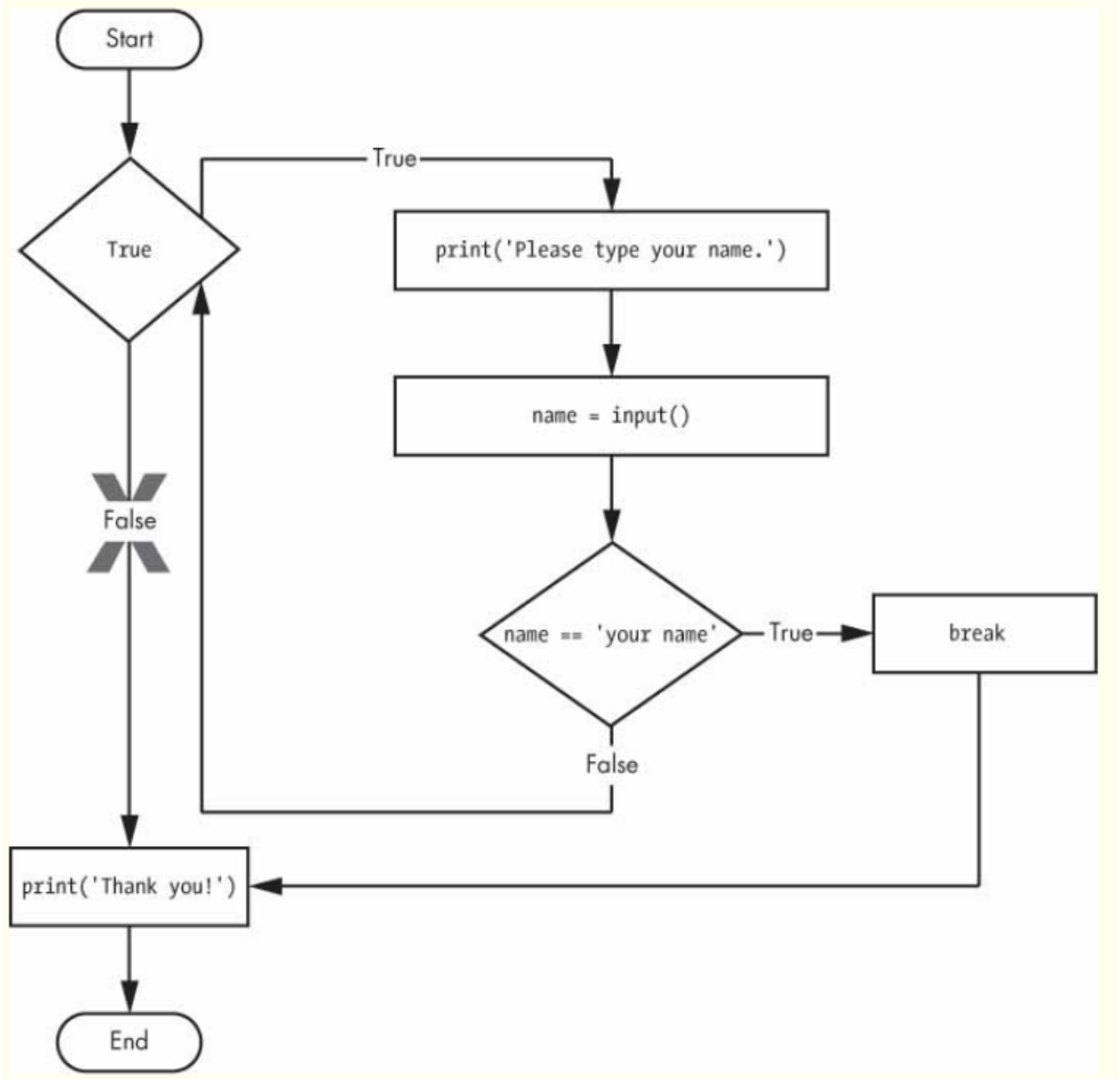


Figure 2-11: The flowchart for the yourName2.py *program with an infinite loop. Note that the X path will logically never happen, because the loop condition is always True.*

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.

In [ ]:

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

See Figure 2-12 for this program's flowchart.

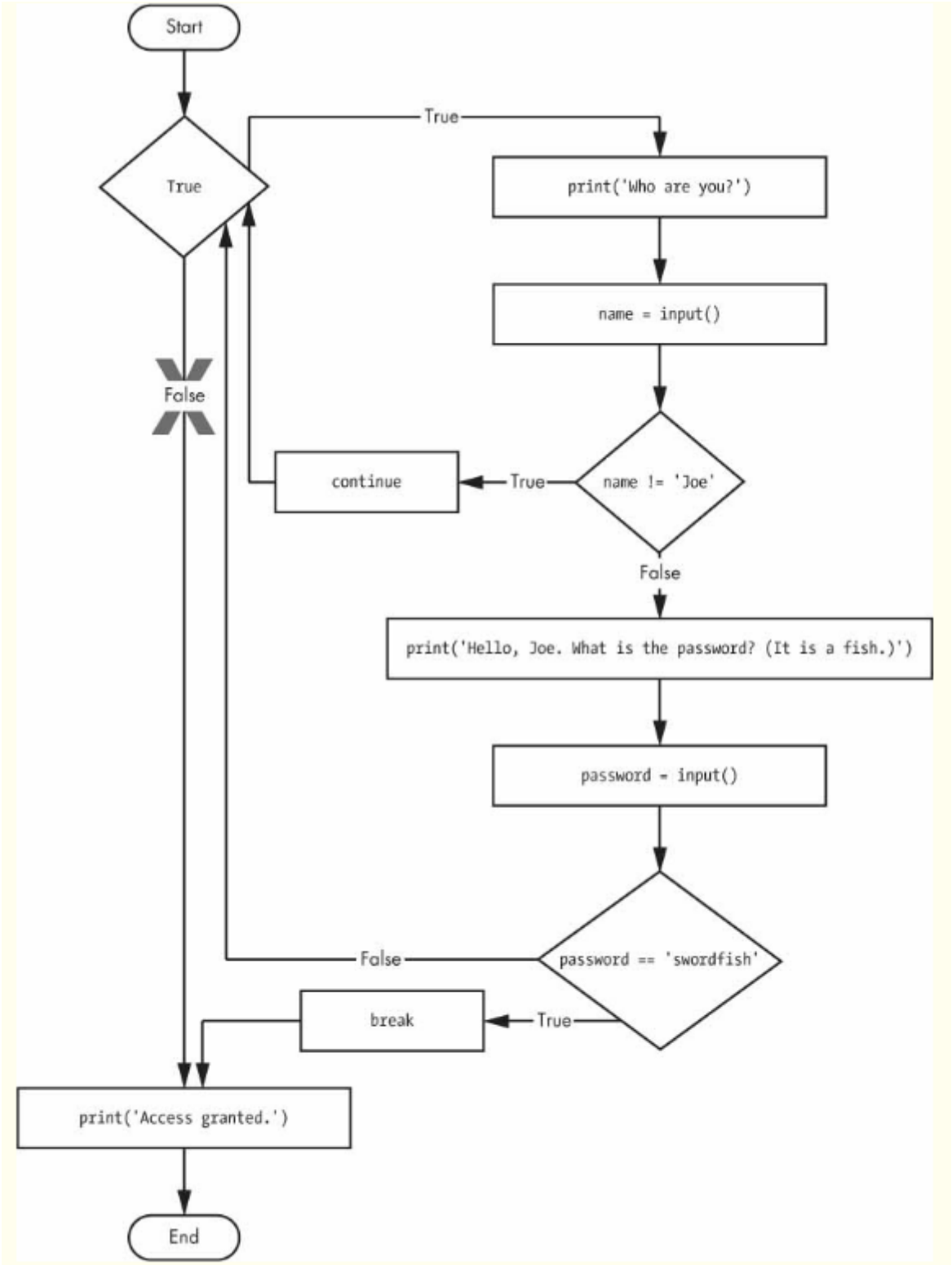


Figure 2-12: A flowchart for swordfish.py. *The X path will logically never happen, because the loop condition is always True.*

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:

- 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)

Let's create a new program called fiveTimes.py to help you see a for loop in action.

In [ ]:

```
```

Figure 2-13 shows a flowchart for the above program

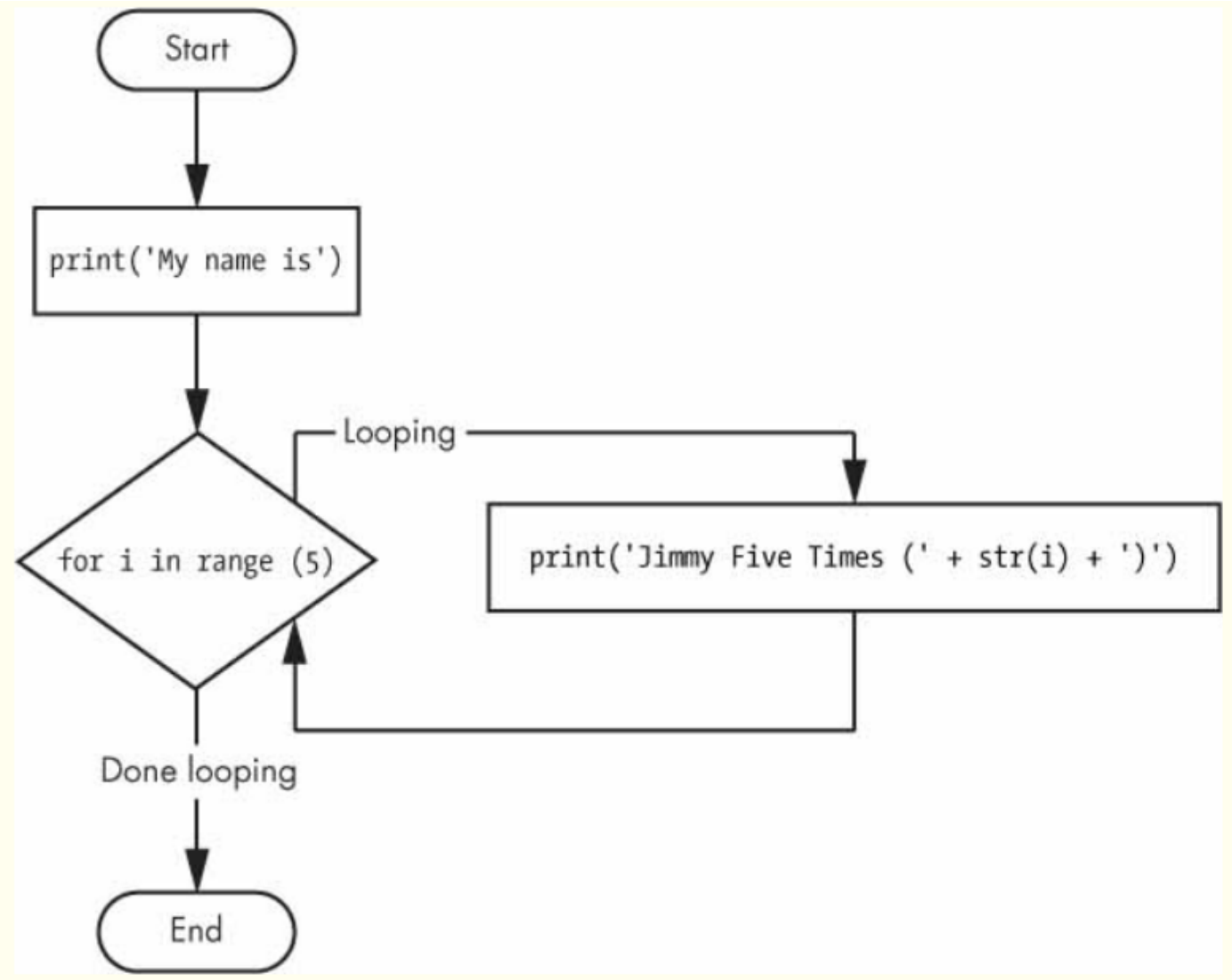


Figure 2-13: The flowchart for fiveTimes.py

In [ ]:

```
total = 0
for num in range(101):
    total = total + num
print(total)
```

An Equivalent while Loop

You can actually use a while loop to do the same thing as a for loop; for loops are just more concise.

In [ ]:

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

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

Some functions can be called with multiple arguments separated by a comma, and range() is one of them. This lets you change the integer passed to range() to follow any sequence of integers, including starting at a number other than zero.

In [ ]:

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

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

In [ ]:

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

The range() function is flexible in the sequence of numbers it produces for for loops. For example (I never apologize for my puns),
you can even use a negative number for the step argument to make the for loop count down instead of up.

In [ ]:

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

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.

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. Enter the following code:

In [ ]:

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

In [ ]:

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

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

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.

In [ ]:

```
import sys

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

A Short Program: Guess the Number

I'll show you a simple "guess the number" game. When you run this program, the output will look something 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!

In [ ]:

```
## This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break    # This condition is the correct guess!

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + 'guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

A Short Program: Rock, Paper, Scissors

Let’s use the programming concepts we’ve learned so far to create a simple rock, paper, scissors game. The output will look like this:

```
ROCK, PAPER, SCISSORS
0 Wins, 0 Losses, 0 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
p
PAPER versus...
PAPER
It is a tie!
0 Wins, 1 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
s
SCISSORS versus...
PAPER
You win!
1 Wins, 1 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
q
```

In [ ]:

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of wins, losses, and ties.
wins = 0
losses = 0
ties = 0

while True: # The main game loop.
    print('%s Wins, %s Losses, %s Ties' % (wins, losses, ties))
    while True: # The player input loop.
        print('Enter your move: (r)ock (p)aper (s)cissors or (q)uit')
        playerMove = input()
        if playerMove == 'q':
            sys.exit() # Quit the program.
        if playerMove == 'r' or playerMove == 'p' or playerMove == 's':
            break # Break out of the player input loop.
        print('Type one of r, p, s, or q.')

    # Display what the player chose:
    if playerMove == 'r':
        print('ROCK versus...')
    elif playerMove == 'p':
        print('PAPER versus...')
    elif playerMove == 's':
        print('SCISSORS versus...')

    # Display what the computer chose:
    randomNumber = random.randint(1, 3)
    if randomNumber == 1:
        computerMove = 'r'
        print('ROCK')
    elif randomNumber == 2:
        computerMove = 'p'
        print('PAPER')
    elif randomNumber == 3:
        computerMove = 's'
        print('SCISSORS')

    # Display and record the win/loss/tie:
    if playerMove == computerMove:
        print('It is a tie!')
        ties = ties + 1
    elif playerMove == 'r' and computerMove == 's':
        print('You win!')
        wins = wins + 1
    elif playerMove == 'p' and computerMove == 'r':
        print('You win!')
        wins = wins + 1
    elif playerMove == 's' and computerMove == 'p':
        print('You win!')
        wins = wins + 1
    elif playerMove == 'r' and computerMove == 'p':
        print('You lose!')
        losses = losses + 1
    elif playerMove == 'p' and computerMove == 's':
        print('You lose!')
        losses = losses + 1
```

```
    elif playerMove == 's' and computerMove == 'r':
        print('You lose!')
        losses = losses + 1
```

Summary by Quiz

FUNCTIONS

A function is like a miniprogram within a program.

In [ ]:

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

hello()
```

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

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.

In [ ]:

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

hello('Alice')
hello('Bob')
```

Define, Call, Pass, Argument, Parameter

In [ ]:

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

Return Values and return Statements

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:

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

When an expression is used with a return statement, the return value is what this expression evaluates to.

In [ ]:

```
import random

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

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

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.

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.

In [ ]:

```
spam = print('Hello!')
```

In [ ]:

```
None == spam
```

Keyword Arguments and the print() Function

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.

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

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.

Think of a scope as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten.
There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed,
and all its variables are forgotten. Otherwise, the next time you ran a program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in the function exist within the function's local scope.
When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you

call the function,
the local variables will not remember the values stored in them from the last time the function was called.

Scopes matter for several reasons:

- 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 spam.

Local Variables Cannot Be Used in the Global Scope

In [ ]:

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

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function.

In [ ]:

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

def bacon():
    ham = 101
    eggs = 0

spam()
```

Global Variables Can Be Read from a Local Scope

In [ ]:

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

Local and Global Variables with the Same Name

In [ ]:

```
def spam():
    eggs = 'spam local'
    print(eggs)    # prints 'spam local'

def bacon():
    eggs = 'bacon local'
    print(eggs)    # prints 'bacon local'
    spam()
    print(eggs)    # prints 'bacon local'

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

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.”

In [ ]:

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

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

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.

In [ ]:

```
def spam():
    global eggs
    eggs = 'spam' # this is the global

def bacon():
    eggs = 'bacon' # this is a local

def ham():
    print(eggs) # this is the global

eggs = 42 # this is the global
spam()
print(eggs)
```

In a function, a variable will either always be global or always be local. The code in a function can't use a local variable named eggs and
then use the global eggs variable later in that same function.

In [ ]:

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

eggs = 'global'
spam()
```

FUNCTIONS AS "BLACK BOXES"

Often, all you need to know about a function are its inputs (the parameters) and output value; you don't always have to burden yourself with how the function's code actually works. When you think about functions in this high-level way, it's common to say that you're treating a function as a "black box."

This idea is fundamental to modern programming. Later chapters in this book will show you several modules with functions that were written by other people. While you can take a peek at the source code if you're curious, you don't need to know how these functions work in order to use them. And because writing functions without global variables is encouraged, you usually don't have to worry about the function's code interacting with the rest of your program.

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.

In [ ]:

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

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

Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause.
The program execution moves to the start of a following except clause if an error happens.

You can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens
when this error occurs.

In [ ]:

```
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))
```

In [ ]:

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

try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

A Short Program: Zigzag

In [ ]:

```
import time, sys
indent = 0 # How many spaces to indent.
indentIncreasing = True # Whether the indentation is increasing or not.

try:
    while True: # The main program loop.
        print(' ' * indent, end='')
        print('********')
        time.sleep(0.1) # Pause for 1/10 of a second.

        if indentIncreasing:
            # Increase the number of spaces:
            indent = indent + 1
            if indent == 20:
                # Change direction:
                indentIncreasing = False

        else:
            # Decrease the number of spaces:
            indent = indent - 1
            if indent == 0:
                # Change direction:
                indentIncreasing = True
except KeyboardInterrupt:
    sys.exit()
```

Summary by Quiz

End of Module-1

In [ ]:

In [ ]: