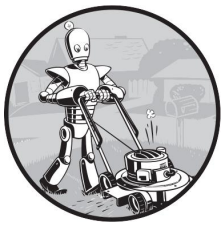
# 2

# IF-ELSE AND FLOW CONTROL



So, you know the basics of individual instructions and that a program is just a series of such instructions. But programming's real strength isn't just running one instruction after another like a weekend errand list. Based on how expressions evaluate, a program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *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.

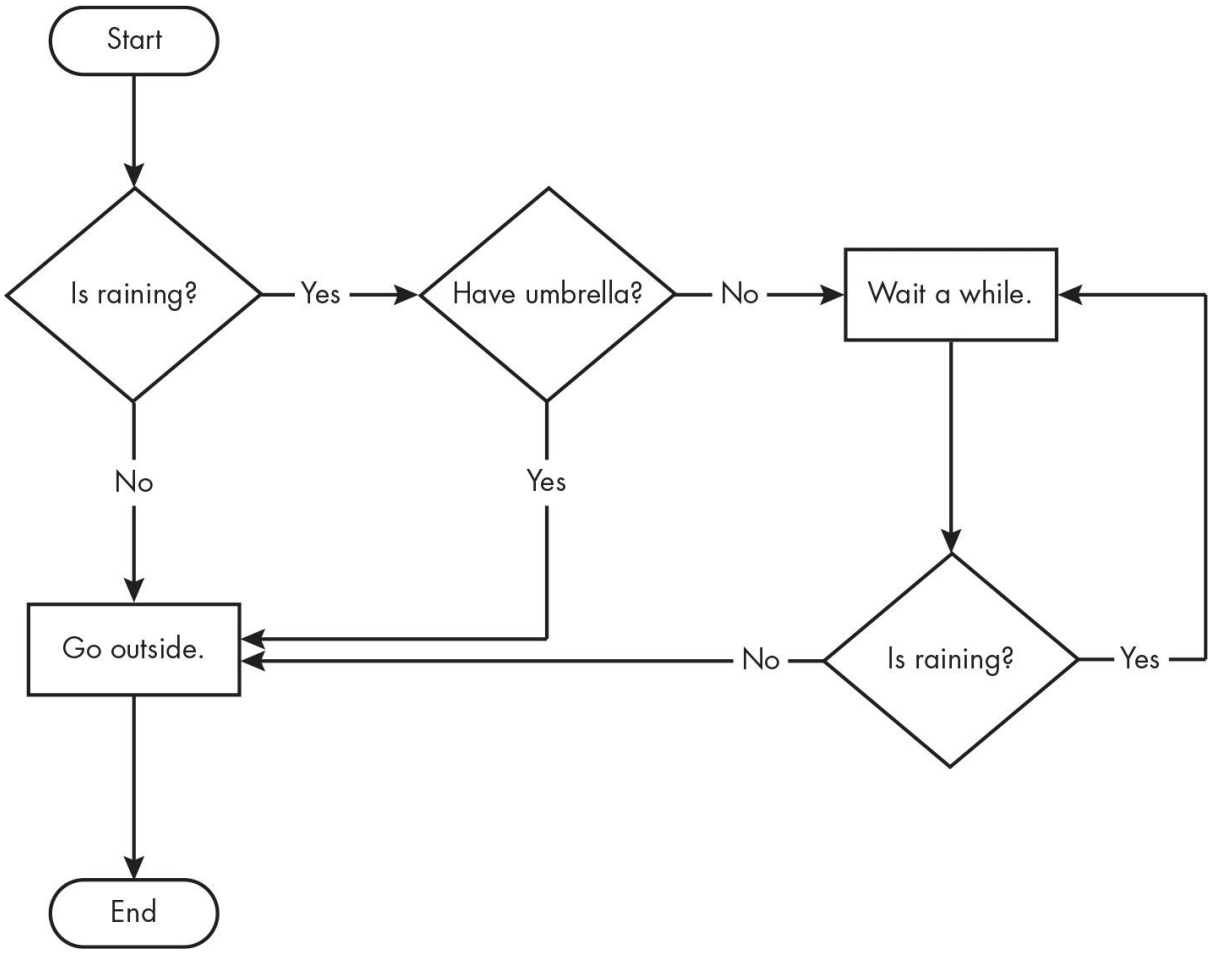



*Figure 2-1: A flowchart to tell you what to do if it is raining*
Description

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, the other steps with rectangles, and the starting and ending steps with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

## 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 quotation marks you place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase. Note that these Boolean values don't have quotes, because they are different from the string values `'True'` and `'False'`. Enter the following into the interactive shell:

```
❶ >>> spam = True
  >>> spam
  True
❷ >>> true
  Traceback (most recent call last):
    File "<python-input-0>", line 1, in
  <module>
  NameError: name 'true' is not defined
❸ >>> False = 2 + 2
    File "<python-input-0>", line 1, in
  <module>
  SyntaxError: can't assign to False
```

Some of these instructions are intentionally incorrect, and they'll cause error messages to appear. Like any other value, you can use Boolean values in expressions and store them in variables ❶. If you don't use the proper case ❷ or if you try to use `True` and `False` for variable names ❸, Python will give you an error message.

## 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 | Examples |
|---|---|---|
| `==` | Equal to | `5 == 5` evaluates to `True`. `4 == 2 + 2` evaluates to `True`. |
| `!=` | Not equal to | `1 != 2` evaluates to `True`. `'Hello' != 'Hello'` evaluates to `False`. |
| `<` | Less than | `10 < 5` evaluates to `False`. `1.999 < 5` evaluates to `True`. |
| `>` | Greater than | `1 + 1 > 4 + 8` evaluates to `False`. `99 > 4 + 8` evaluates to `True`. |
| `<=` | Less than or equal to | `4 <= 5` evaluates to `True`. `5 <= 5` evaluates to `True`. |

| Operator | Meaning | Examples |
|---|---|---|
| >= | Greater than or equal to | 5 >= 4 evaluates to True. 5 >= 5 evaluates to True. |

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
```

As you might expect, == (equal to) evaluates to `True` when the values on both sides are the same, and != (not equal to) evaluates to `True` when the two values are different. The == and != operators can actually work with values of any data type:

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

```
❶ >>> 42 == '42'
False
```

Note that an integer or floating-point value will never equal a string value. The expression `42 == '42'` ❶ evaluates to `False` because Python considers the integer `42` to be different from the string `'42'`. However, Python does consider the integer `42` to be the same as the float `42.0`.

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

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggs = 42
❶ >>> eggs <= 42
True
>>> my_age = 29
❷ >>> my_age >= 10
True
```

You'll often use comparison operators to compare a variable's value to some other value, like in the `eggs <= 42` ❶ and `my_age >= 10` ❷ examples, or to compare the values in two variables to each other. (After all, comparing two literal values like `'dog' != 'cat'` always has the same result.) You'll see more examples of this later when you learn about flow control statements.

### 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 asks whether two values are the same as each other.
- The = operator 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.

The `and` operator always takes two Boolean values (or expressions), so it's considered to be a *binary* Boolean operator. 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:

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

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

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

| Expression | Evaluates to ... |
|---|---|
| `True and True` | `True` |
| `True and False` | `False` |
| `False and True` | `False` |
| `False and False` | `False` |

Like the `and` operator, the `or` operator also always takes two Boolean values (or expressions), and therefore is considered to be a binary Boolean operator. However, the `or` operator evaluates an

expression to `True` if *either* of the two Boolean values is `True`. If both are `False`, it evaluates to `False`:

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

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` |

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:

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

Much like using double negatives in speech and writing, you can use multiple `not` operators ❶, though there's never not no reason to do this in real programs. Table 2-4 shows the truth table for `not`.

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

| Expression | Evaluates to ... |
|---|---|
| `not True` | `False` |
| `not False` | `True` |

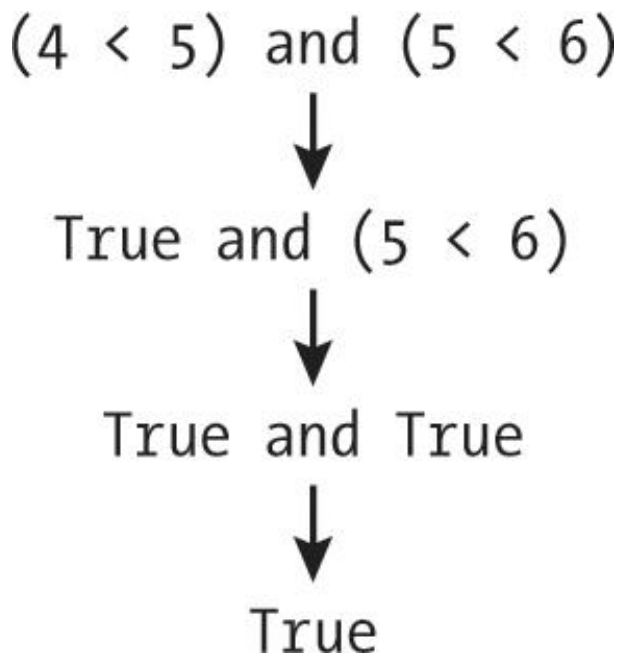
## Mixing Boolean and Comparison Operators

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

Recall that the `and`, `or`, and `not` operators are called Boolean operators because they always operate on the Boolean values `True` and `False`. 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:

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

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will 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:




Description

You can also use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> spam = 4
>>>
2 + 2 == spam and not 2 + 2 == (spam + 1) and
2 * 2 == 2 + 2
True
```

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

## Components 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*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

### *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. A condition always evaluates 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. You'll frequently write code that could be described in English as follows: "If this condition is true, do this thing, or else do this other thing." Other code you'll write is the same as saying, "Keep repeating these instructions as long as this condition continues to be true."

### *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 four rules for blocks:

- A new block begins when the indentation increases.
- Blocks can contain other blocks.

- A block ends when the indentation decreases to zero or to a containing block's indentation.
- Python expects a new block immediately after any statement that ends with a colon.

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

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

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

## *Program Execution*

In the previous chapter's *hello.py* program, Python started executing instructions at the top of the program going down, one after another. *Program execution* (or simply, *execution*) is a term for the current instruction being executed. If you put your finger on each line on your screen as the line is executed, you can think of your finger as the program execution.

Not all programs execute by simply going straight down, however. If you use your finger to trace through a program with flow control statements, you'll likely find your finger jumping around to different places in the source code based on conditions.

# Flow Control Statements

Now let's explore the most important piece of flow control: the statements themselves. The statements represent the diamonds you saw in the flowchart in Figure 2-1, and they are the actual decisions your programs will make.

## *if*

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`.

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 or `if` block)

For example, let's say you have some code that checks whether someone's name is Alice:

```
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 the flowchart of this code would look like.

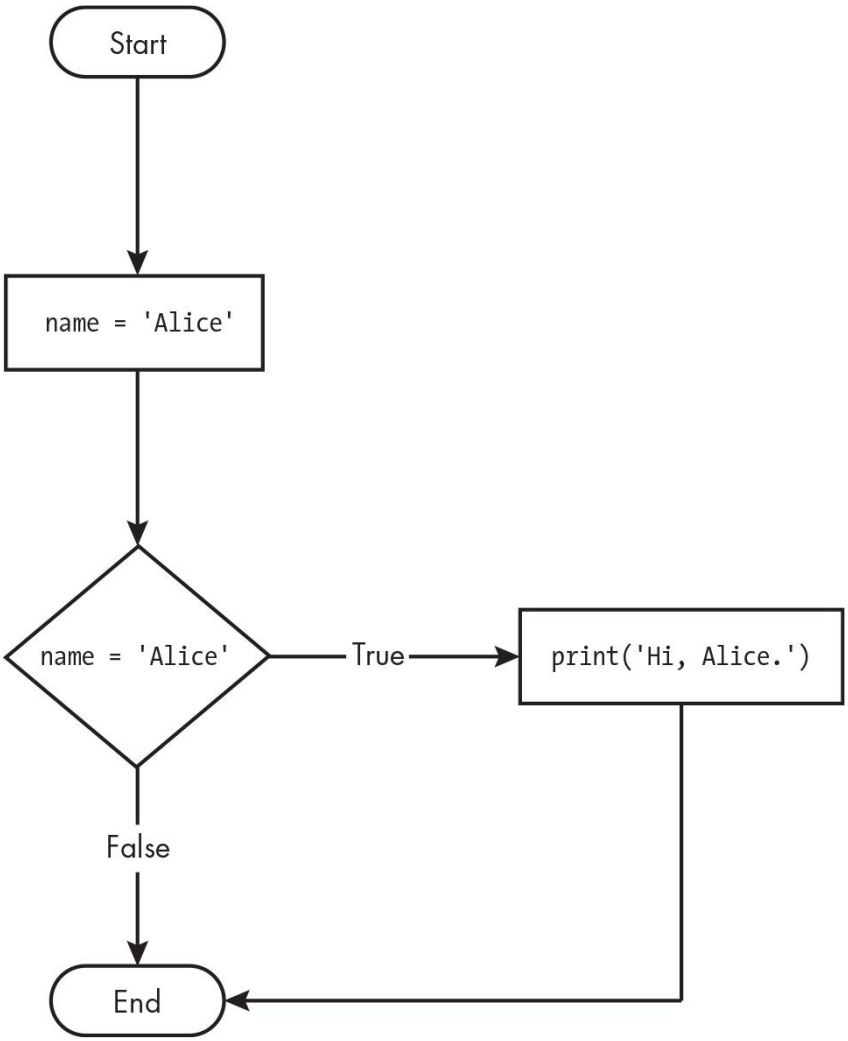



*Figure 2-2: The flowchart for an `if` statement* Description

Try changing the `name` variable to another string besides `'Alice'`, and run the program again. Notice that "Hi, Alice." doesn't appear on the screen, because that code was skipped over.

## *else*

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 or `else` block)

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:

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

Figure 2-3 shows what the flowchart of this code would look like.

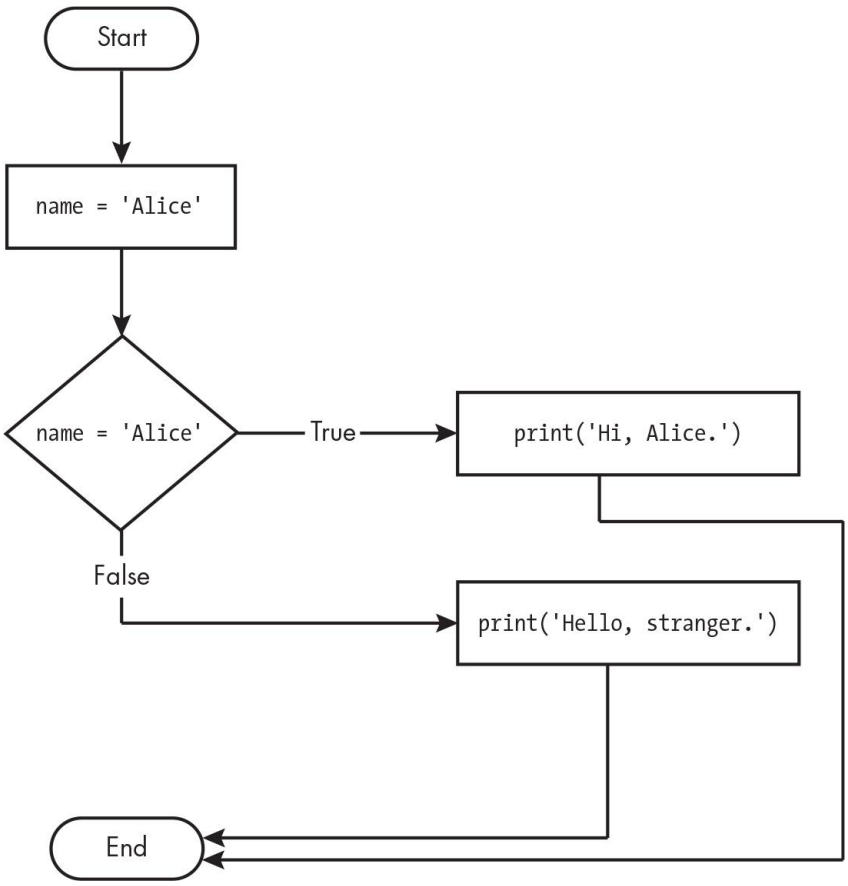



*Figure 2-3: The flowchart for an `else` statement* Description

Try changing the name variable to a string besides `'Alice'`, and rerun the program. Instead of `'Hi, Alice.'`, you will see `'Hello, stranger.'` on the screen.

## elif

You would use `if` or `else` when you want only one of the clauses to execute. But 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 or `elif` block)

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

```
name = 'Alice'
age = 33
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

This time, the program checks the person's age and tells them something different if they're younger than 12. You can see the corresponding flowchart in Figure 2-4.

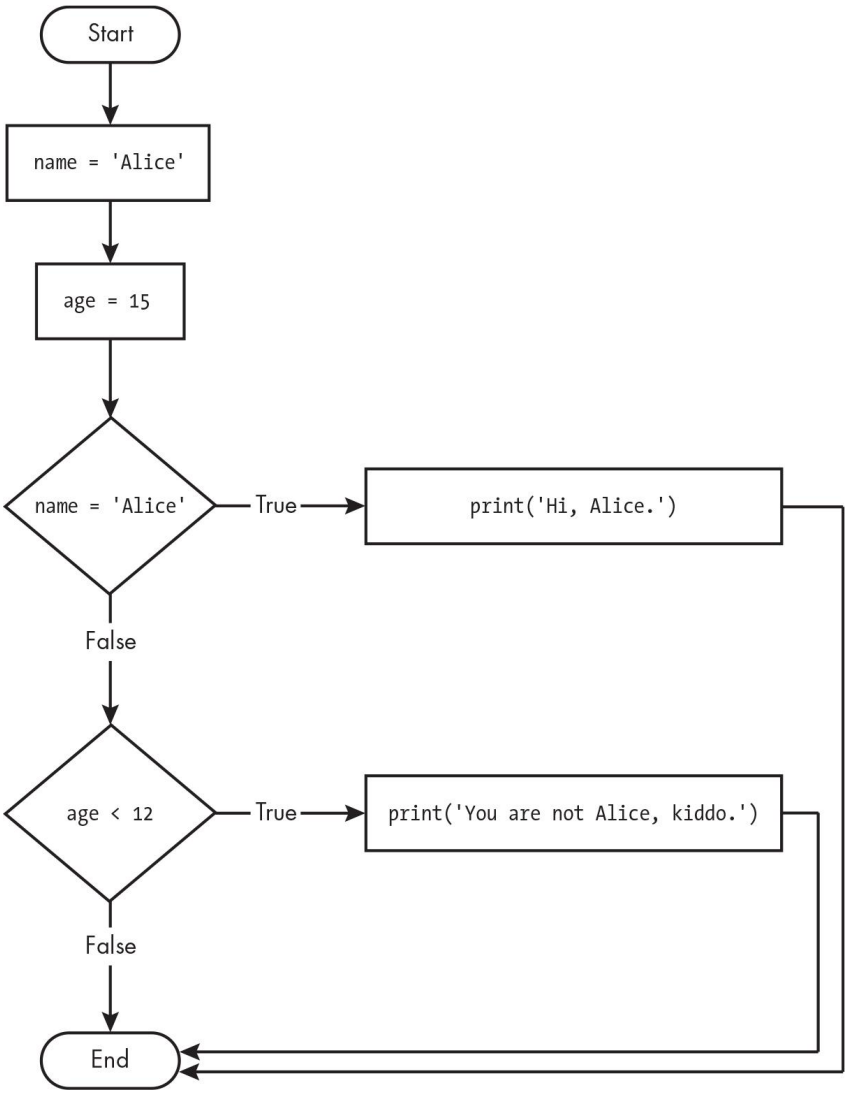



*Figure 2-4: The flowchart for an `elif` statement* Description

The `elif` clause executes if `age < 12` is `True` and `name == 'Alice'` is `False`. However, if both of the conditions are `False`, Python skips both of the clauses. There is *no* guarantee that at least one of the clauses will be executed; in 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, saving it as *vampire.py*:

```
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.')
```

Here, I've added two more `elif` statements to make the name checker greet a person with different answers based on `age`. Figure 2-5 shows the flowchart for this.

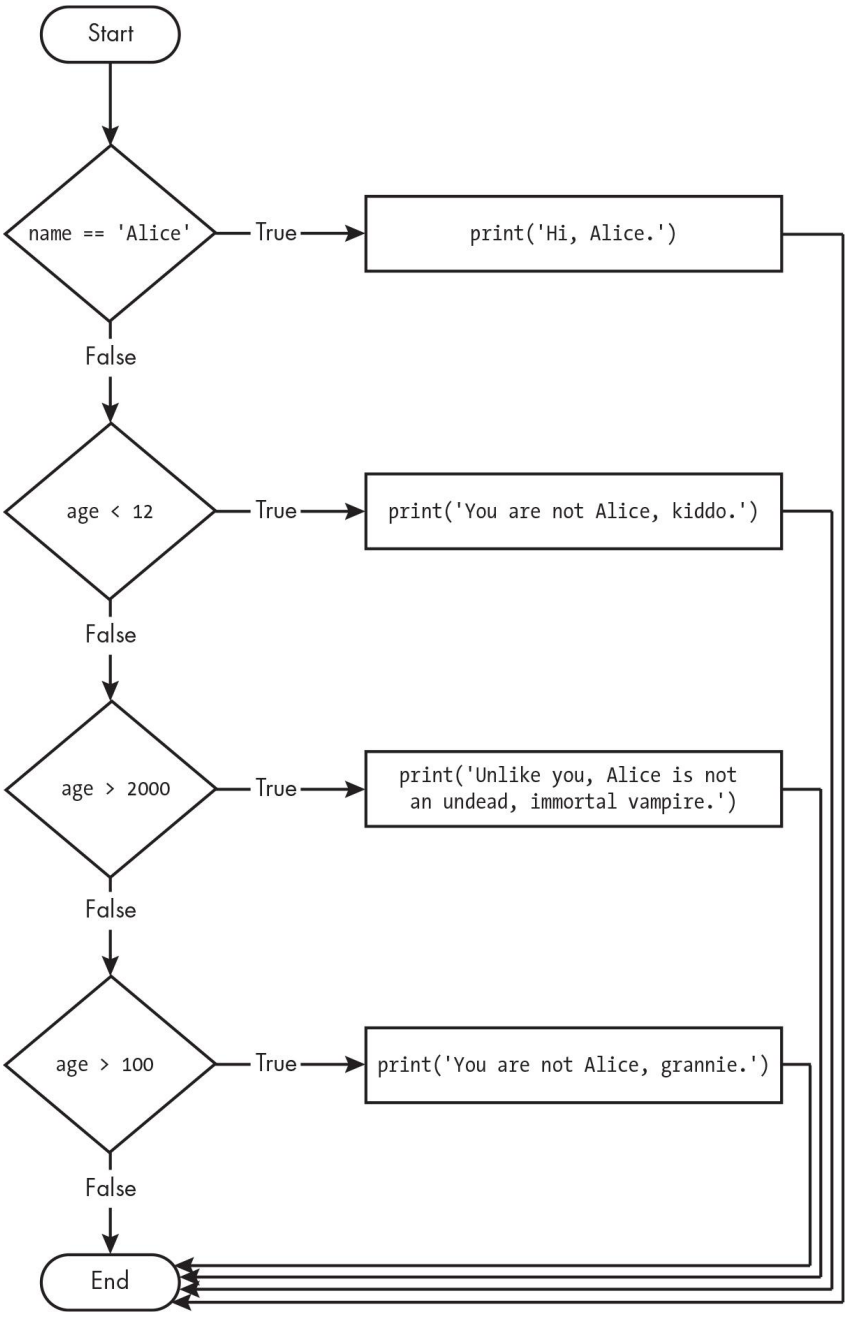



*Figure 2-5: The flowchart for multiple `elif` statements in the* vampire.py *program* Description

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 *vampire.py*, you

will run into a problem. Change the code to look like the following, and save it as *vampire2.py*:

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

Say the `age` variable contains the value `3000` before this code is executed. You might expect the code to print the string `'Unlike you, Alice is not an undead, immortal vampire.'` However, because the `age > 100` condition is `True` (after all, 3,000 *is* greater than 100) ❶, the string `'You are not Alice, grannie.'` is printed, and the rest of the `elif` statements are automatically skipped. Remember that at most only one of the clauses will be executed, and for `elif` statements, the order matters!

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

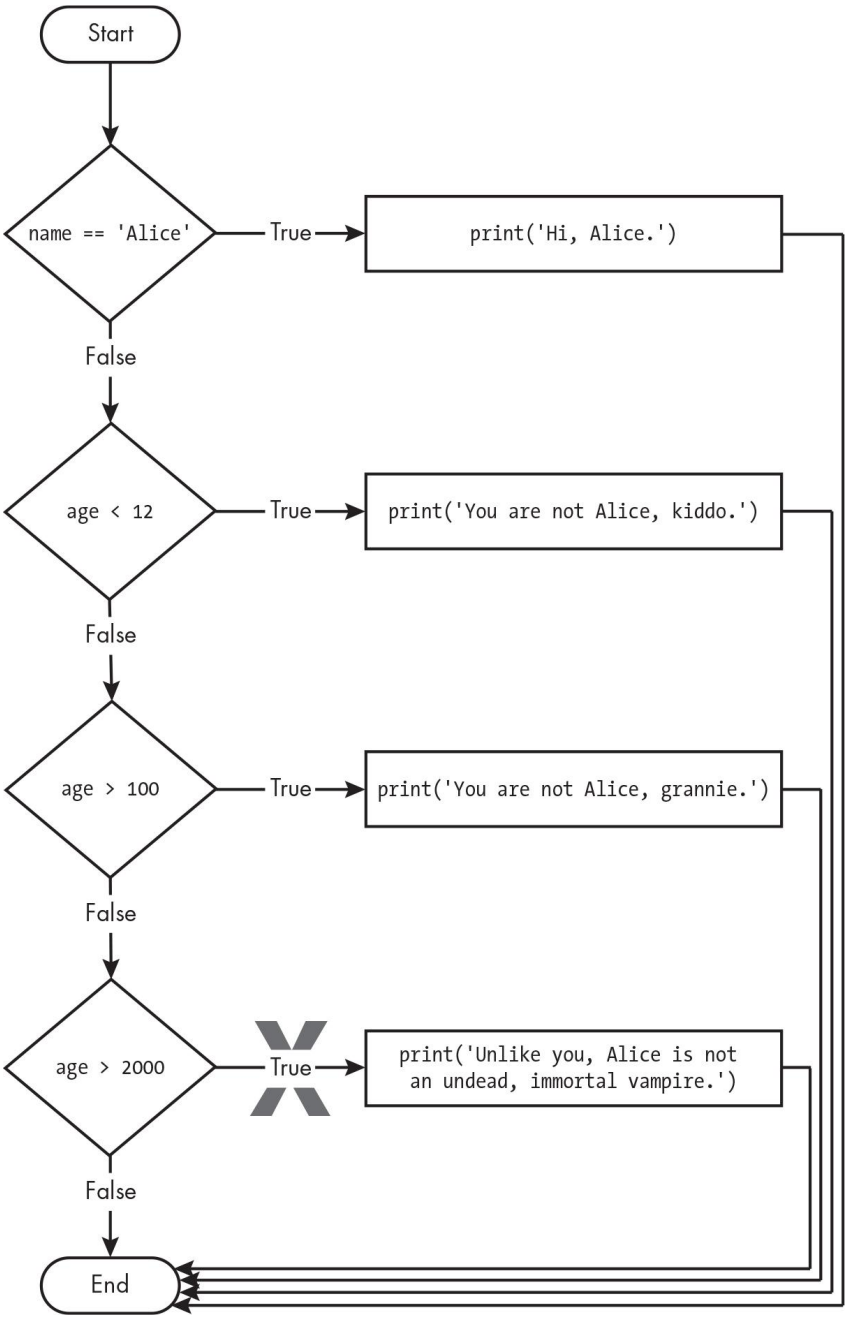



*Figure 2-6: The* vampire2.py *program flowchart. The X path will logically never happen, because if* `age` *were greater than* `2000`*, it would have already been greater than 100.* Description

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:

```
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 shows the flowchart for this new code, which we'll save as *littleKid.py*.

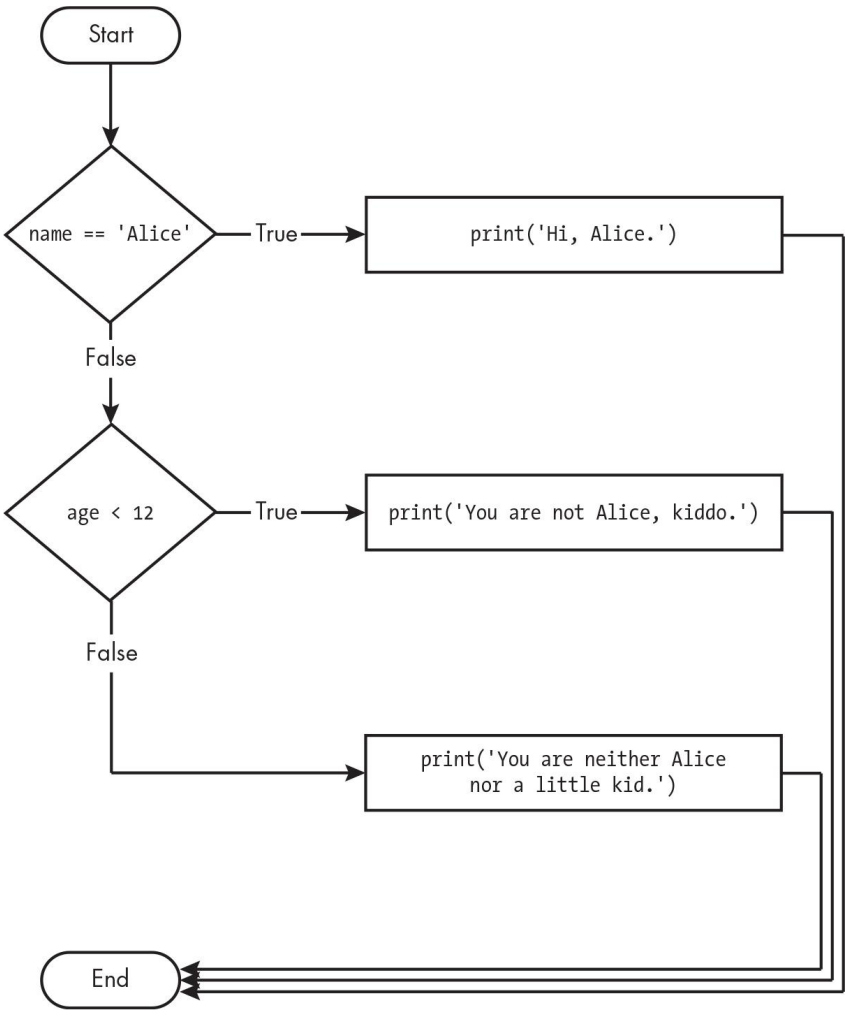



*Figure 2-7: The flowchart for the* littleKid.py *program* Description

In plain English, this type of flow control structure would be, "If the first condition is true, do this. Else, if the second condition is true, do that. Otherwise, do something else." When you use `if`, `elif`, and `else` statements together, remember these rules about how to order them to avoid bugs like the one in Figure 2-6. First, there is always exactly one `if` statement; any `elif` statements you need should follow the `if` statement. Second, if you want to be sure that at least one clause is executed, close the structure with an `else` statement.

As you can see, flow control statements can make your programs more sophisticated but also more complicated. Don't despair; you will become more comfortable with this complexity as you practice writing code. And all true programmers have at some point spent an hour to find

out their program doesn't work because they accidentally typed < instead of <=. These little mistakes happen to everyone.

## A Short Program: Opposite Day

With your knowledge of Boolean values and `if-else` statements, enter the following code into a new file and save it as *oppositeday.py*:

```
today_is_opposite_day = True

# Set say_it_is_opposite_day based on
today_is_opposite_day:
❶ if today_is_opposite_day == True:
    say_it_is_opposite_day = True
else:
    say_it_is_opposite_day = False

# If it is opposite day, toggle
say_it_is_opposite_day:
if today_is_opposite_day == True:
  ❷ say_it_is_opposite_day = not
say_it_is_opposite_day

# Say what day it is:
if say_it_is_opposite_day == True:
    print('Today is Opposite Day.')
else:
    print('Today is not Opposite Day.')
```

When you run this program, it outputs `'Today is not Opposite Day.'` There are two variables in this code. At the start of the program, the `today_is_opposite_day` variable is set to `True`. The next `if` statement checks if this variable is `True` (it is) ❶ and sets the `say_it_is_opposite_day` variable to `True`; otherwise, it would set the variable to `False`. The second `if` statement checks if `today_is_opposite_day` is set to `True` (it still is), and if so, the code *toggles* (that is, sets to the opposite Boolean value) the

variable ❷. Finally, the third `if` statement checks if `say_it_is_opposite_day` is `True` (it isn't) and prints `'Today is Opposite Day.'`; otherwise, it would have printed `'Today is not Opposite Day.'`

If you change the first line of the program to `today_is_opposite_day = False` and run the program again, it still prints `'Today is not Opposite Day.'` If we look through the program, we can figure out that the first `if-else` statements set `say_it_is_opposite_day` to `False`. The second `if` statement's condition is `False`, so it skips its block of code. Finally, the third `if` statement's condition is again `False` and prints `'Today is not Opposite Day.'`

So, if today is not Opposite Day, the program correctly prints `'Today is not Opposite Day.'` And if today is Opposite Day, the program (also correctly) prints `'Today is not Opposite Day.'` as one would say on Opposite Day. Logically, this program will never print `'Today is Opposite Day.'` no matter if the variable is set to `True` or `False`. Really, you could replace this entire program with just `print('Today is not Opposite Day.')` and it would be the same program. This is why programmers should not be paid per line of code written.

## A Short Program: Dishonest Capacity Calculator

In Chapter 1, I showed how hard drive and flash memory manufacturers lie about the advertised capacities of their products by using a different definition of TB and GB. Let's write a program to calculate how misleading their advertised capacities are. Enter the following code into a new file and save it as *dishonestcapacity.py*:

```
print('Enter TB or GB for the advertised
unit:')
unit = input('>')

# Calculate the amount that the advertised
capacity lies:
if unit == 'TB' or unit == 'tb':
    discrepancy = 1000000000000 /
1099511627776
```

```
elif unit == 'GB' or unit == 'gb':
    discrepancy = 1000000000 / 1073741824

print('Enter the advertised capacity:')
advertised_capacity = input('>')
advertised_capacity =
float(advertised_capacity)

# Calculate the real capacity, round it to
the nearest hundredths,
# and convert it to a string so it can be
concatenated:
real_capacity = str(round(advertised_capacity
* discrepancy, 2))

print('The actual capacity is ' +
real_capacity + ' ' + unit)
```

This program asks the user to enter what unit the hard drive advertises itself as having, either TB or GB:

```
# Calculate the amount that the advertised
capacity lies:
if unit == 'TB' or unit == 'tb':
    discrepancy = 1000000000000 /
1099511627776
elif unit == 'GB' or unit == 'gb':
    discrepancy = 1000000000 / 1073741824
```

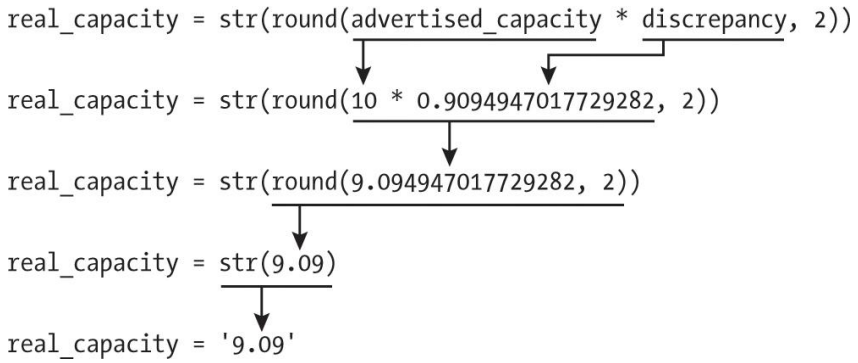
TBs are larger than GBs, and the larger the unit, the wider the discrepancy between advertised and real capacities. The `if` and `elif` statements use a Boolean `or` operator so that the program works no matter whether the user enters the unit in lowercase or uppercase. If the user enters something else for the unit, then neither the `if` clause nor the `elif` clause runs, and the discrepancy variable is never assigned.

Later, when the program tries to use the discrepancy variable, this will cause an error. We'll cover that case later.

Next, the user enters the advertised size in the given units:

```
# Calculate the real capacity, round it to
the nearest hundredths,
# and convert it to a string so it can be
concatenated:
real_capacity = str(round(advertised_capacity
* discrepancy, 2))
```

We do a lot in this single line. Let's use the example of the user entering 10TB for the advertised size and unit. If we look at the innermost part of the line of code, we see that `advertised_capacity` is multiplied by `discrepancy`. This is the real capacity, but it may have several digits, as in `9.094947017729282`. So this number is passed as the first argument to `round()` with `2` as the second argument. This function call to `round()` returns, in our example, `9.09`. This is a floating-point value, but we want to get a string form of it to concatenate to a message string in the next line of code. To do this, we pass it to the `str()` function. Python evaluates this one line as the following:




Description

If the user failed to enter *TB*, *tb*, *GB*, or *gb* as the unit, the conditions for both the `if` and `elif` statements would be `False` and the `discrepancy` variable would never be created. But the user wouldn't know anything was wrong until Python tried to use the nonexistent variable. Python would raise a `NameError: name 'discrepancy' is not defined` error and point to the line where `real_capacity` is assigned.

The true origin of this bug, however, is the fact that the program doesn't handle the case where the user enters an invalid unit. There are many ways to handle this error, but the simplest would be to have an `else` clause that displays a message like "You must enter TB or GB" and then calls the `sys.exit()` function to quit the program. (You'll learn about this function in the next chapter.)

The final line in the program displays the actual hard drive capacity by concatenating a message string to the `real_capacity` and `unit` strings:

```
print('The actual capacity is ' +
real_capacity + ' ' + unit)
```

As it turns out, hard drives and flash memory manufacturers lie even more: I have a 256GB SD card in my laptop that I use for backups. In real GBs, this should be 274,877,906,944 bytes. In fake GBs, it should be 256,000,000,000 bytes. But my computer reports that the actual capacity is 255,802,212,352 bytes. It's funny how the actual size is always inaccurate in a way that makes it less than the advertised size, and never more.

## Summary

By using expressions that evaluate to `True` or `False` (also called conditions), you can write programs that make decisions on what code to execute and what code to skip. These conditions are expressions that compare two values with the ==, `!=`, <, >, <=, and >= comparison operators to evaluate to a Boolean value. You can also use the `and`, `or`, and `not` Boolean operators to connect expressions into more complicated expressions. Python uses indentation to create blocks of code. In this chapter, we used blocks as part of `if`, `elif`, and `else` statements, but as you'll see, several other Python statements use blocks as well. These flow control statements will let you write more intelligent programs.

## Practice Questions

1. What are the two values of the Boolean data type? How do you write them?
2. What are the three Boolean operators?
3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).

4. What do the following expressions evaluate to?

---

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3 == 5)
not ((5 > 4) or (3 == 5))
(True and True) and (True == False)
(not False) or (not True)
```

---

5. What are the six comparison operators?
6. What is the difference between the equal to operator and the assignment operator?
7. Explain what a condition is and where you would use one.
8. Identify the three blocks in this code:

---

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
       print('bacon')
    else:
        print('ham')
    print('spam')
print('Done')
```

---

9. Write code that prints `Hello` if `1` is stored in `spam`, prints `Howdy` if `2` is stored in `spam`, and prints `Greetings!` if anything else is stored in `spam`.