Lesson 14. Making more-complicated decisions

After reading lesson 14, you'll be able to

- Combine many decisions in one conditional statement
- Make a choice when presented with various options
- Write code that gets the computer to decide between a few choices

It's limiting and time-consuming if every decision you make is the result of asking only one question at a time. Say you want to buy a new phone. There are only three phones that you're considering, but you're not sure how much money you have in your bank account. Additionally, one other criteria is that the phone is available in green. Using yes or no questions, you could ask the following:

- Do I have between $400 and $600?
- Do I have between $200 and $400?
- Do I have between $0 and $100?
- Does Phone 1 come in green?
- Does Phone 2 come in green?
- Does Phone 3 come in green?

Because you have more than one condition you want to check, you can combine two (or more) together. For example, you could ask, "Do I have between $400 and $600, and does Phone 1 come in green?"

Consider this

You're seven years old and are trying to choose your best friend based on the sports that you both play. The order of importance of sports is soccer, basketball, and baseball. You want to have as many sports as possible in common. If that isn't possible, you want your friend to play as many sports as possible from that preferred order. List all the possible combinations of sports in order. Tommy plays soccer and baseball. How many choices down in the list is he?

Answer:

```
soccer and basketball and baseball
soccer and basketball
soccer and baseball              <----- Three choices down
basketball and baseball
soccer
basketball
baseball
```

14.1. COMBINING MULTIPLE CONDITIONS

You know how to write code that depends on whether one condition is true. This means deciding "this or not this." Sometimes, the decision you want to make might be "this or that or that or some other thing."

For example, if "It is raining" is true and "I am hungry" is false, then "It is raining and I am hungry" is false. Table 14.1 shows the truth values of statements made up of two statements.

Table 14.1. Truth values for combinations of two statements with "and" and "or"

Statement 1 (example: "It is	Word to combine the statements	Statement 2 (example: "I am	Result (example: "It is raining <_>

Statement 1 (example: “It is raining”)	Word to combine the statements (<and>, <or>, <not>)	Statement 2 (example: “I am hungry”)	Result (example: “It is raining <_> I am hungry”)
True	<and>	True	True
True	<and>	False	False
False	<and>	True	False
False	<and>	False	False
True	<or>	True	True
True	<or>	False	True
False	<or>	True	True
False	<or>	False	False
N/A	<not>	True	False
N/A	<not>	False	True

Suppose you’re making a simple pasta dinner. How do you think about making it? You ask yourself whether you have pasta and pasta sauce. If you have both, you can make your pasta dinner. Notice that a couple of ideas arise from this simple question.

One idea is that you combined two questions in one: *Do you have pasta and pasta sauce?* These questions could be asked in a different way, in a nested fashion, which would end up giving you the same final answer: *Do you have pasta? If yes, do you have pasta sauce?* But combining the two questions in one is easier to understand.

The other idea is that you used an important word, *and*, to link two questions that have yes/no answers. The word *and* and the word *or* are both Boolean operators, which are used to link two questions that have yes/no answers.

Quick check 14.1

Combine the following questions by using the Boolean operators `and`/`or`:

1

Do you need milk? If yes, do you have a car? If yes, drive to the store and buy milk.

2

Is variable `a` zero? If yes, is variable `b` zero? If yes, is variable `c` zero? If yes, then all variables are zero.

3

Do you have a jacket? Do you have a sweater? Take one of these; it’s cold outside.

The code examples so far have only one expression that evaluates to `true` or `false` inside the conditional statement. In reality, you can make decisions based on more than one condition. In programming, you can combine multiple conditional expressions in one `if` statement. This way, you don’t have to write separate `if` statements for every separate conditional. This leads to cleaner code that’s easier to read and understand.

14.1.1. Conditionals are made up of true/false expressions

You’ve seen conditionals in which only one expression evaluates to `true`/`false`; for example, `num_a < 0`. An `if` statement can check multiple conditionals and act accordingly, depending on whether the entire expression, made up of multiple conditionals, is `true`/`false`. This is where the truth table you saw in table 14.1 is useful. You use it to combine more than one expression by using the Boolean operators `and` and `or`. In Python, the words `and` and `or` are keywords.

You can have an `if` statement made up of more than one expression, as shown in the following listing.

Listing 14.1. Multiple conditional expressions in one `if` statement

```
if num_a < 0 and num_b < 0:
    print("both negative")
```

Here, two decisions must be made before entering inside the code block of the `if` statement: one decision is if `num_a < 0`, and the other decision is if `num_b < 0`.

Recall that expressions are evaluated to a value that's a Python object—for example, an integer value. After you start to combine multiple expressions, you need to be careful about the order in which expressions and parts of each expression are evaluated.

In math, you learned about the operator precedence of addition, subtraction, multiplication, and division. In programming, the same precedence exists as in math, but additional operations must be taken into account—things like comparison operators and logical operators to combine Boolean expressions.

Table 14.2 shows a complete set of operator precedence rules, which tells you which operations are done before others in Python. These precedence rules are used, among other things, for evaluating the result of a larger conditional made up of smaller conditional expressions.

Table 14.2. Order of operations, with those at the top being executed first. Operations at the same precedence level within one cell are left-associative; they're executed left to right as encountered in an expression.

Operator	Meaning
()	Parentheses
**	Exponent
* / // %	Multiplication Division Floor division Modulus
+ -	Addition Subtraction
== != > >= < <= is is not in not in	Is equal to Is not equal to Greater than Greater than or equal to Less than Less than or equal to Identity (object is another object) Identity (object is not another object) Membership (object is in another object) Membership (object isn't in another object)
not	Logical NOT
and	Logical AND
or	Logical OR

Quick check 14.2

Evaluate the following expressions by using the operator precedence in table 14.2:

1

```
3 < 2 ** 3 and 3 == 3
```

2

```
0 != 4 or (3/3 == 1 and (5 + 1) / 3 == 2)
```

3

```
"a" in "code" or "b" in "Python" and len("program")
== 7
```

Take a look at the following (incorrect) code. It's similar to the code in listing 14.1, except that the line `num_a < 0 and num_b < 0` is written as `num_a and num_b < 0`.

Listing 14.2. Code that doesn't do what you think it does

```
if num_a and num_b < 0:
    print("both negative")
```

If you run the code with the following different values for `num_a` and `num_b`, you'll get the output in table 14.3. An empty entry means no output. Notice that one of the pairs of values gives a misleading printout.

Table 14.3. Result in the console output after running the code in listing 14.2 with different values for `num_a` and `num_b`

num_a	num_b	Console output
-1	-1	both negative
-1	1	
0	-1	
0	1	
1	-1	both negative
1	1	

When `num_a = 1` and `num_b = -1`, the output printed to the console is `both negative`, which is incorrect. Use the precedence rules to see what's

By the precedence rules in table 14.2, the `and` logical operator has lower precedence than the "less than" comparison. The expression `num_a and num_b < 0` can be rewritten as `(num_a and (num_b < 0))`.

In Python, all integer values except 0 are considered `True`, and the integer value 0 is considered `False`. `if -1` evaluates to `if True`, and `if 0` evaluates to `if False`. Because of the precedence rules, whenever `num_a` is anything except 0, the expression evaluates to `True`. When `num_a = 1` and `num_b = -1`, the code incorrectly prints `both negative` because `(num_a and (num_b < 0))` evaluates to `(1 and (-1 < 0))`, which evaluates to `(True and True)`, which is `True`.

Quick check 14.3

Q1:

Go back to the code in listing 14.1. The conditional there can be rewritten, using the precedence rules and parentheses, as `((num_a < 0) and (num_b < 0))`. Draw a table for a few combinations of `num_a` and `num_b` to convince yourself that all possible pairs of values give the expected printout.

14.2. CHOOSING WHICH LINES TO EXECUTE

Now you understand the purpose of a conditional statement and how to write one in Python. Conditionals don't have to be used as only single "detours" in code. They can also be used to make a decision as to which blocks of code to execute.

14.2.1. Do this or that

Sometimes you want to perform one task but not another. For example, you might say something like "If it is sunny, then I will walk to work; otherwise, if it is cloudy, then I will take an umbrella and walk to work; but otherwise, I will drive." For this, the `elif` and `else` keywords will be used in combination with an `if` statement.

Listing 14.3 shows a simple `if-elif-else` conditional statement in code. You get an input number from the user. If the number is greater than 0, you print `positive`. Otherwise, if the number is less than zero, you print `negative`. Otherwise, you print that the number is zero. Only one of the messages will be printed.

Listing 14.3. Example of a simple `if-elif-else` conditional statement

```
num = int(input("Enter a number: "))                1
if num > 0:                                         2
    print("num is positive")                        3
elif num < 0:                                       4
    print("num is negative")                        5
else:                                               6
    print("num is zero")                            7
```

- ***1*** **User input**
- ***2*** **Condition checks that the number is greater than 0**
- ***3*** **Prints a message**
- ***4*** **When if num > 0 is False, do this condition to check that the number is less than 0.**
- ***5*** **Prints a message**
- ***6*** **When elif num < 0 is False, the else is a catchall.**
- ***7*** **Prints a message**

Here, you start a conditional with the `if` statement. Any `elif` or `else` statements that come after it are associated with that `if` statement. This kind of structure means that you'll execute the code block that belongs to the first decision that's true.

Quick check 14.4

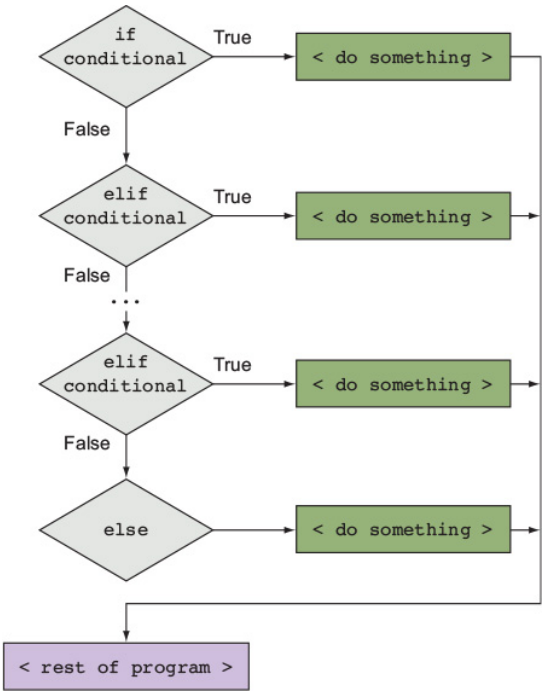
Q1:

What's printed when you run listing 14.3 with the following values for num: -3, 0, 2, 1?

Figure 14.1 shows how to visualize multiple decisions. Each decision is a conditional statement; a group of decisions are part of an `if-elif-else` code block. Follow any path in figure 14.1 by tracing the paths denoted by arrows. The main program decisions are shown by the diamonds. The first decision

gram>, you'll notice that you can deviate from the main path of the program at most only once. The path you'll deviate to is the first path whose condition evaluates to `True`.

Figure 14.1. Visualizing a generic `if-elif-else` code block. You can deviate from the main program flow by doing `<do something>` at most once. You can have zero or more `elif` blocks, and the `else` block is optional.



The `if-elif-else` code block in figure 14.1 is a generic block. You can have the following variations:

- Only one `if` statement (you saw this in the previous lesson)
- One `if` statement and one `elif` statement
- One `if` statement and many `elif` statements
- One `if` statement and one `else` statement
- One `if` statement, one or more `elif` statements, and one `else` statement

For all of these variations, the detour executed is the first one whose condition evaluates to `True`. If none evaluate to `True`, the `else` detour is executed. If the preceding variations don't include an `else` statement, it's possible that none of the detours to `<do something>` are executed.

Quick check 14.5

Q1:

Draw a flowchart for listing 14.3.

The following listing shows the generic way of writing code that does one thing or another, depending on whether certain conditions hold, as shown in figure 14.1.

Listing 14.4. A General way to write a simple `if-elif-else` conditional

```
if <conditional>:              1
    <do something>
elif <conditional>:            2
    <do something>
else:                          3
    <do something>
```

- ***1* The keyword "if" starts the conditional block.**
- ***2* The keyword "elif" starts the "else if" conditional block.**
- ***3* The keyword "else" starts the catchall other conditional cases.**

The keyword `if` starts the conditional block, as before, followed by a conditional expression and then the colon character. When the `if` statement conditional is `True`, the code block for that `if` statement is executed, and then all remaining code blocks that are part of the `if-elif-else` group are skipped. When the `if` statement conditional is `False`, you check the condi-

If the conditional in the `elif` statement is `True`, the code block for that `elif` statement is executed, and all remaining code blocks that are part of the `if-elif-else` group are skipped. You can have as many `elif` statements as you want (zero or more). Python looks at conditionals one after another and will execute the first code block that evaluates to `True`.

When none of the conditionals from the `if` or any of the `elif` statements are `True`, the code block inside the `else` is executed. You can think of the `else` as a catchall conditional for when nothing else is `True`.

When there's no `else` statement, and none of the conditionals evaluate to `True`, the conditional block won't do anything.

Quick check 14.6

Q1:

Take a look at these code snippets:

```
With if-elif-else statements          With if stat
if num < 6:                           if num < 6:
   print("num is less than 6")          print("num
elif num < 10:                        if num < 10:
   print("num is less than 10")         print("num
elif num > 3:                         if num > 3:
   print("num is greater than 3")       print("num
else:                                 print("Finis
   print("No relation found.")
print("Finished.")
```

What will the user see on the screen if `num` has the following values?

```
num         With if-elif-else         With if
-------------------------------------------------
20
-------------------------------------------------
9
-------------------------------------------------
5
-------------------------------------------------
0
```

14.2.2. Putting it all together

At this point, you can see that the structure of the programs is changing yet again:

- You can decide to do one of many things by checking different conditions.
- The `if-elif` structure is used to enter the first code block that's `True`.
- The `else` is used to do something when nothing else is `True`.

Listing 14.5 shows a simple program that checks the user input. When the user enters a noninteger value for either input, the program prints a message to the user and then moves on to the next group, at the same indentation level, of `if-elif-else` statements. It doesn't enter the `else` code block associated with the first `if` statement because it already executed the block within the `if`.

When the user enters two valid integers, you enter the `else` code block and print a message depending on the sign of the numbers inputted. Only the message associated with the first time a condition evaluates to `True` within the nested `if-elif-else` statement will be printed. After that code block finishes, you move on to check the next `if-elif-else` group, seeing whether the user guessed the lucky number.

Listing 14.5. Example of how to use `if-elif-else` statements

```
num_a = 5
num_b = 7
lucky_num = 7
if type(num_a) != int or type(num_b) != int:
    print("You did not enter integers")
else:
    if num_a > 0 and num_b > 0:
        print("both numbers are positive")
    elif num_a < 0 and num_b < 0:
        print("both numbers are negative")
    else:
        print("numbers have opposite sign")
if num_a == lucky_num or num_b == lucky_num:
```

```
else:
    print("I have a secret number in mind...")
```

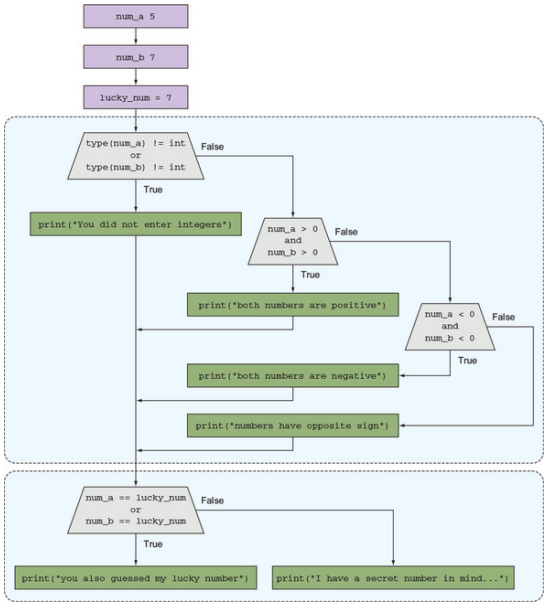
- ***1*** **One group of if-else with a nested if-elif-else group**
- ***2*** **Nested group of if-elif-else**
- ***3*** **Another group of if-else**

Thinking like a programmer

Programmers write readable code, both for others to be able to read and for themselves to look back on later. It's a good idea to create variables to store complex computations and give them descriptive names rather than including them in conditionals directly. For example, don't do `if (x ** 2 - x + 1 == 0) or (x + y ** 3 + x ** 2 == 0)`. Instead, create variables `x_eq = x ** 2 - x + 1` and `xy_eq = x + y ** 3 + x ** 2` and then check `if x_eq == 0 or xy_eq == 0`.

Python makes it easy to visualize which lines should be executed because the code blocks are indented. You can take listing 14.5 and visualize the code in terms of blocks. In figure 14.2, you see that the conditionals have a cascading look.

Figure 14.2. Visualization of listing 14.5, showing the conditional code blocks



Within the conditional group, you'll execute only the first branch that evaluates to `True`. Whenever you have another `if` statement at the same level as another `if` statement, you're starting another conditional group.

You can see in figure 14.2 that two major conditional blocks are at the main level: the one that checks the user input and the one that checks for the lucky number. Using this visualization, you can even propose a rewrite of the code in listing 14.5 to eliminate the `else` statement for the first code block that checks the user input, and convert that to an `elif` statement. The code rewrite is in the next listing.

Listing 14.6. Rewrite of listing 14.5 to convert an `else` to a series of `elif`s

```
num_a = 5
num_b = 7
lucky_num = 7
if type(num_a) != int or type(num_b) != int:
    print("You did not enter integers")
elif num_a > 0 and num_b > 0:
    print("both numbers are positive")
elif num_a < 0 and num_b < 0:
    print("both numbers are negative")
else:
    print("numbers have opposite sign")
if num_a == lucky_num or num_b == lucky_num:
    print("you also guessed my lucky number!")
else:
    print("I have a secret number in mind...")
```

- ***1*** **The else block from listing 14.5 converted to a series of elif blocks**

same.

14.2.3. Thinking in terms of code blocks

It's important to realize that when you decide which branch to execute, you look at only the particular `if-elif-else` conditional group, as shown in listing 14.7. The `if` statement has one check, to see whether the input from the user is one of the strings in the tuple `greet_en` or `greet_sp`. The other two `elif`s each have a nested `if-elif` code block.

Listing 14.7. Example with multiple `if-elif-else` code blocks

```
greeting = input("Say hi in English or Spanish! ")
greet_en = ("hi", "Hi", "hello", "Hello")
greet_sp = ("hola", "Hola")
if greeting not in greet_en and greeting not in greet
    print("I don't understand your greeting.")
elif greeting in greet_en:
    num = int(input("Enter 1 or 2: "))
    print("You speak English!")
    if num == 1:
        print("one")
    elif num == 2:
        print("two")
elif greeting in greet_sp:
    num = int(input("Enter 1 or 2: "))
    print("You speak Spanish!")
    if num == 1:
        print("uno")
    elif num == 2:
        print("dos")
```

- ***1* One code block made up of if-elif statements**
- ***2* A nested block containing an if-elif block**
- ***3* A nested block containing another if-elif block**

The program will enter only one path through the `if-elif-elif`, through one of the following:

- The `if` when the user enters a greeting that isn't in `greet_en` and not in `greet_sp`
- Through the `elif` when `greeting in greet_en`
- Through the `elif` when `greeting in greet_sp`

SUMMARY

In this lesson, my objective was to teach you how to make decisions by using the `if-elif-else` conditional statements, and to teach you how various combinations of their parts affect the program flow. The decisions you can make are now even more complex, because you can choose which code to execute. These are the major takeaways:

- Operator precedence is important when evaluating many expressions inside one conditional.
- The `if` statement indicates whether to take a detour. The `if-elif-else` statements indicate which detour to take.
- Visualize more-complicated programs, which include conditional statements, by using flowcharts.

As you're starting to write programs that involve a few concepts, it's important to actively engage in solving them. Take out a pen and paper and draw out your solution or write out your thought process. Then open your preferred IDE, type your code, and then run, test, and debug your program. Don't forget to comment your code.

Let's see if you got this...

Q14.1

Write a program that reads in two numbers from the user. The program should print the relation between the two numbers, which will be one of the following: `numbers are equal`, `first number is less than the second number`, `first number is greater than the second number`.

Q14.2

Write a program that reads in a string from the user. If the string contains at least one of every vowel (a, e, i, o, u), print `You have all the vowels!` Additionally, if the string starts with the letter *a* and ends with the letter *z*, print `And it's sort of alphabetical!`

PREV
Lesson 13. Introducing decisions in programs

NEXT
Lesson 15. Capstone project: choose your own adventure