Chapter 8

Conditional Logic and Control Flow

Nearly all of the code you have seen in this book is **unconditional**. That is, the code does not make any choices. Every line of code is executed in the order that is written or that functions are called, with possible repetitions inside of loops.

In this chapter, you will learn how to write programs that perform different actions based on different conditions using **conditional logic**. Paired with functions and loops, conditional logic allows you to write complex programs that handle many different situations.

In this chapter, you will learn how to:

- Compare the values of two or more variables
- Write `if` statements to control the flow of your programs
- Handle errors with `try` and `except`
- Apply conditional logic to create simple simulations

Let's get started!

Leave feedback on this section »

8.1 Compare Values

Conditional logic is based on performing different actions depending on whether or not some expression, called a **conditional**, is true or false. This idea is not specific to computers. Humans use conditional logic all the time to make decisions.

For example, the legal age for purchasing alcoholic beverages in the United States is 21. The statement "If you are at least 21 years old, then you may purchase a beer" is an example of conditional logic. The phrase "you are at least 21 years old" is a conditional because it may be either true or false.

In computer programming, conditionals often take the form of comparing two values, such as determining if one value is greater than another, or whether or not two values are equal to each other. A standard set of symbols called **boolean comparators** are used to make comparisons, and most of them may already be familiar to you.

The following table describes these boolean comparators:

Boolean Comparator	Example	Meaning
`>`	`a > b`	`a` greater than `b`
`<`	`a < b`	`a` less than `b`
`>=`	`a >= b`	`a` greater than or equal to `b`
`<=`	`a <= b`	`a` less than or equal to `b`
`!=`	`a != b`	`a` not equal to `b`
`==`	`a == b`	`a` equal to `b`

The term **boolean** is derived from the last name of the English mathematician George Boole, whose works helped lay the foundations of modern computing. In Boole's honor, conditional logic is sometimes called **boolean logic**, and conditionals are sometimes called **boolean expressions**.

There is also a fundamental data type called the **boolean**, or `bool` for short, which can have only one of two values. In Python, these values

are conveniently named `True` and `False`:

```
>>> type(True)
<class 'bool'>

>>> type(False)
<class 'bool'>
```

Note that `True` and `False` both start with capital letters.

The result of evaluating a conditional is always a boolean value:

```
>>> 1 == 1
True

>>> 3 > 5
False
```

In the first example, since `1` is equal to `1`, the result of `1 == 1` is `True`. In the second example, `3` is not greater than `5`, so the result is `False`.

Important

A common mistake when writing conditionals is to use the assignment operator `=`, instead of `==`, to test whether or not two values are equal.

Fortunately, Python will raise a `SyntaxError` if this mistake is encountered, so you'll know about it before you run your program.

You may find it helpful to think of boolean comparators as asking a question about two values. `a == b` asks whether or not `a` and `b` have the same value. Likewise, `a != b` asks whether or not `a` and `b` have different values.

Conditional expressions are not limited to comparing numbers. You may also compare values such as strings:

```
>>> "a" == "a"
True

>>> "a" == "b"
False

>>> "a" < "b"
True

>>> "a" > "b"
False
```

The last two examples above may look funny to you. How could one string be greater than or less than another?

The comparators < and > represent the notions of greater than and less than when used with numbers, but more generally they represent the notion of order. In this regard, `"a" < "b"` checks if the string `"a"` comes before the string `"b"`. But how are string ordered?

In Python, strings are ordered **lexicographically**, which is a fancy way to say they are ordered as they would appear in a dictionary. So you can think of `"a" < "b"` as asking whether or not the letter `a` comes before the letter `b` in the dictionary.

Lexicographic ordering extends to strings with two or more characters by looking at each component letter of the string:

```
>>> "apple" < "astronaut"
True

>>> "beauty" > "truth"
False
```

Since strings can contain characters other than letters of the alphabet, the ordering must extend to those other characters as well.

We won't go in to the details of how characters other than letters are

ordered. In practice, the < and > comparators are most often used with numbers, not strings.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. For each of the following conditional expressions, guess whether they evaluate to `True` or `False`. Then type them into the interactive window to check your answers:

```
1 <= 1
1 != 1
1 != 2
"good" != "bad"
"good" != "Good"
123 == "123"
```

2. For each of the following expressions, fill in the blank (indicated by __) with an appropriate boolean comparator so that the expression evaluates to `True`:

```
3 __ 4
10 __ 5
"jack" __ "jill"
42 __ "42"
```

Leave feedback on this section »

8.2 Add Some Logic

In addition to boolean comparators, Python has special keywords called **logical operators** that can be used to combine boolean expressions. There are three logical operators: `and`, `or`, and `not`.

Logical operators are used to construct compound logical expressions. For the most part, these have meanings similar to their meaning in the English language, although the rules regarding their use in Python are much more precise.

The `and` Keyword

Consider the following statements:

1. Cats have four legs.
2. Cats have tails.

In general, both of these statements are true.

When we combine these two statements using `and`, the resulting sentence "cats have four legs and cats have tails" is also a true statement. If both statements are negated, the compound statement "cats do not have four legs and cats do not have tails" is false.

Even when we mix and match false and true statements, the compound statement is false. "Cats have four legs and cats do not have tails" and "cats do not have four legs and cats have tails" are both false statements.

When two statements *P* and *Q* are combined with `and`, the **truth value** of the compound statement "*P* and *Q*" is true if and only if both *P* and *Q* are true.

Python's `and` operator works exactly the same way. Here are four example of compound statements with `and`:

```
>>> 1 < 2 and 3 < 4  # Both are True
True
```

Both statements are `True`, so the combination is also `True`.

```
>>> 2 < 1 and 4 < 3  # Both are False
False
```

Both statements are `False`, so their combination is also `False`.

```
>>> 1 < 2 and 4 < 3  # Second statement is False
False
```

`1 < 2` is `True`, but `4 < 3` is `False`, so their combination is `False`.

```
>>> 2 < 1 and 3 < 4  # First statement is False
False
```

`2 < 1` is `False`, and `3 < 4` is `True`, so their combination is `False`.

The following table summarizes the rules for the `and` operator:

Combination using `and`	Result
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

You can test each of these rules in the interactive window:

```
>>> True and True
True

>>> True and False
False

>>> False and True
False

>>> False and False
False
```

The `or` Keyword

When we use the word "or" in everyday conversation, sometimes we mean an **exclusive or**. That is, only the first option or the second option can be true.

For example, the phrase "I can stay or I can go" uses the exclusive or. I can't both stay and go. Only one of these options can be true.

In Python the `or` keyword is inclusive. That is, if *P* and *Q* are two ex-

pressions, the statement "P or Q" is true if any of the following are true:

1. P is true
2. Q is true
3. Both P and Q are true

Let's look at some examples using numerical comparisons:

```
>>> 1 < 2 or 3 < 4  # Both are True
True

>>> 2 < 1 or 4 < 3  # Both are False
False

>>> 1 < 2 or 4 < 3  # Second statement is False
True

>>> 2 < 1 or 3 < 4  # First statement is False
True
```

Note that if any part of a compound statement is `True`, even if the other part is `False`, the result is always true `True`. The following table summarizes these results:

Combination using `or`	Result
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Again, you can verify all of this in the interactive window:

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

```
>>> True or False
True

>>> False or True
True

>>> False or False
False
```

The `not` Keyword

The `not` keyword reverses the truth value of a single expression:

Use of `not`	Result
`not True`	`False`
`not False`	`True`

You can verify this in the interactive window:

```
>>> not True
False

>>> not False
True
```

One thing to keep in mind with `not`, though, is that it doesn't always behave the way you might expect when combined with comparators like `==`. For example, `not True == False` returns `True`, but `False == not True` will raise an error:

```
>>> not True == False
True

>>> False == not True
  File "<stdin>", line 1
```

```
    False == not True
             ^
SyntaxError: invalid syntax
```

This happens because Python parses logical operators according to an **operator precedence**, just like arithmetic operators have an order of precedence in everyday math.

The order of precedence for logical and boolean operators, from highest to lowest, is described in the following table. Operators on the same row have equal precedence.

Operator Order of Precedence (Highest to Lowest)
<, <=, ==, >=, >
not
and
or

Looking again at the expression `False == not True`, `not` has a lower precedence than `==` in the order of operations. This means that when Python evaluates `False == not True`, it first tries to evaluate `False == not` which is syntactically incorrect.

You can avoid the `SyntaxError` by surrounding `not True` with parentheses:

```
>>> False == (not True)
True
```

Grouping expressions with parentheses is a great way to clarify which operators belong to which part of a compound expression.

Building Complex Expressions

You can combine the `and`, `or` and `not` keywords with `True` and `False` to create more complex expressions. Here's an example of a more complex expression:

```
True and not (1 != 1)
```

What do you think the value of this expression is?

To find out, break the expression down by starting on the far right side. `1 != 1` is `False`, since `1` has the same value as itself. So you can simplify the above expression as follows:

```
True and not (False)
```

Now, `not (False)` is the same as `not False`, which is `True`. So you can simplify the above expression once more:

```
True and True
```

Finally, `True and True` is just `True`. So, after a few steps, you can see that `True and not (1 != 1)` evaluates to `True`.

When working through complicated expressions, the best strategy is to start with the most complicated part of the expression and build outward from there.

For instance, try evaluating the following expression:

```
("A" != "A") or not (2 >= 3)
```

Start by evaluating the two expressions in parentheses. `"A" != "A"` is `False` because `"A"` is equal to itself. `2 >= 3` is also `False` because `2` is smaller than `3`. This gives you the following equivalent, but simpler, expression:

```
(False) or not (False)
```

Since `not` has a higher precedence than `or`, the above expression is equivalent to the following:

```
False or (not False)
```

`not False` is `True`, so you can simplify the expression once more:

```
False or True
```

Finally, since any compound expression with `or` is `True` if any one of the expressions on the left or right of the `or` is `True`, you can conclude that `("A" != "A") or not (2 >= 3)` is `True`.

Grouping expressions in a compound conditional statement with parentheses improves readability. Sometimes, though, parenthesis are required to produce the expected value.

For example, upon first inspection, you may expect the following to output `True`, but it actually returns `False`:

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

The reason this is `False` is that the `==` operator has a higher precedence than `and`, so Python interprets the expression as `True and (False == True) and False`. Since `False == True` is `False`, this is equivalent to `True and False and False`, which evaluates to `False`.

The following shows how to add parentheses so that the expression evaluates to `True`:

```
>>> (True and False) == (True and False)
True
```

Logical operators and boolean comparators can be confusing the first time you encounter them, so if you don't feel like the material in this section comes naturally, don't worry!

With a little bit of practice, you'll be able to make sense of what's going on and build your own compound conditional statements when you need them.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Figure out what the result will be (`True` or `False`) when evaluating the following expressions, then type them into the interactive window to check your answers:

```
(1 <= 1) and (1 != 1)
not (1 != 2)
("good" != "bad") or False
("good" != "Good") and not (1 == 1)
```

2. Add parentheses where necessary so that each of the following expressions evaluates to `True`:

```
False == not True
True and False == True and False
not True and "A" == "B"
```

Leave feedback on this section »

8.3 Control the Flow of Your Program

Now that we can compare values to one other with boolean comparators and build complex conditional statements with logical operators, we can add some logic to our code so that it performs different actions for different conditions.

The `if` Statement

An `if` statement tells Python to only execute a portion of code if a condition is met.

For example, the following `if` statement will print `2 and 2 is 4` if the conditional `2 + 2 == 4` is `True`:

```
if 2 + 2 == 4:
    print("2 and 2 is 4")
```

In English, you can read this as "if 2 + 2 is 4, then print the string `'2 and 2 is 4'`."

Just like `while` loops, an `if` statement has three parts:

1. The `if` keyword
2. A test condition, followed by a colon
3. An indented block of code that is executed if the test condition is `True`

In the above example, the test condition is `2 + 2 == 4`. Since this expression is `True`, executing the `if` statement in IDLE displays the text `2 and 2 is 4`.

If the test condition is `False` (for instance, `2 + 2 == 5`), Python skips over the indented block of code and continues execution on the next non-indented line.

For example, the following `if` statement does not print anything:

```
if 2 + 2 == 5:
    print("Is this the mirror universe?")
```

A universe where `2 + 2 == 5` is `True` would be pretty strange indeed!

Note

Leaving off the colon (`:`) after the test condition in an `if` statement raises a `SyntaxError`:

```
>>> if 2 + 2 == 4
SyntaxError: invalid syntax
```

Once the indented code block in an `if` statement is executed, Python will continue to execute the rest of the program.

Consider the following script:

```
grade = 95

if grade >= 70:
    print("You passed the class!")
```

```
print("Thank you for attending.")
```

The output looks like this:

```
You passed the class!
Thank you for attending.
```

Since `grade` is `95`, the test condition `grade >= 70` is `True` and the string `"You passed the class!"` is printed. Then the rest of the code is executed and `"Thank you for attending."` is printed.

If you change the value of `grade` to `40`, the output looks like this:

```
Thank you for attending.
```

The line `print("Thank you for attending.")` is executed whether or not `grade` is greater than or equal to `70` because it is after the indented code block in the `if` statement.

A failing student will not know that they failed if all they see from your code is the text `"Thank you for attending."`.

Let's add another `if` statement to tell the student they did not pass if their grade is less than `70`:

```
grade = 40

if grade >= 70:
    print("You passed the class!")

if grade < 70:
    print("You did not pass the class :(")

print("Thank you for attending.")
```

The output now looks like this:

```
You did not pass the class :(
Thank you for attending.
```

In English, we can describe an alternate case with the word "otherwise." For instance, "If your grade is 70 or above, you pass the class. Otherwise, you do not pass the class."

Fortunately, there is a keyword that does for Python what the word "otherwise" does in English.

The `else` Keyword

The `else` keyword is used after an `if` statement in order to execute some code only if the `if` statement's test condition is `False`.

The following script uses `else` to shorten the code in the previous script for displaying whether or not a student passed a class:

```
grade = 40

if grade >= 70:
    print("You passed the class!")
else:
    print("You did not pass the class :(")

print("Thank you for attending.")
```

In English, the `if` and `else` statements together read as "If the grade is at least 70, then print the string `"You passed the class!"`; otherwise, print the string `"You did not pass the class :("`.

Notice that the `else` keyword has no test condition, and is followed by a colon. No condition is needed, because it executes for any condition that fails the `if` statement's test condition.

Important

Leaving off the colon (`:`) from the `else` keyword will raise a `SyntaxError`:

```
>>> if 2 + 2 == 5:
...     print("Who broke my math?")
... else
SyntaxError: invalid syntax
```

The output from the above script is:

```
You did not pass the class :(
Thank you for attending.
```

The line that prints `"Thank you for attending."` still runs, even if the indented block of code after `else` is executed.

The `if` and `else` keywords work together nicely if you only need to test a condition with exactly two states.

Sometimes, you need to check three or more conditions. For that, you use `elif`.

The `elif` Keyword

The `elif` keyword is short for "else if" and can be used to add additional conditions after an `if` statement.

Just like `if` statements, `elif` statements have three parts:

1. The `elif` keyword
2. A test condition, followed by a colon
3. An indented code block that is executed if the test condition evaluates to `True`

Important

Leaving off the colon (`:`) at the end of an `elif` statement raises a `SyntaxError`:

```
>>> if 2 + 2 == 5:
...     print("Who broke my math?")
... elif 2 + 2 == 4
SyntaxError: invalid syntax
```

The following script combines `if`, `elif`, and `else` to print the letter grade a student earned in a class:

```
grade = 85  # 1

if grade >= 90:  # 2
    print("You passed the class with a A.")
elif grade >= 80:  # 3
    print("You passed the class with a B.")
elif grade >= 70:  # 4
    print("You passed the class with a C.")
else:  # 5
    print("You did not pass the class :(")

print("Thanks for attending.")  # 6
```

Both `grade >= 80` and `grade >= 70` are `True` when `grade` is `85`, so you might expect both `elif` blocks on lines `3` and `4` to be executed.

However, only the first block for which the test condition is `True` is executed. All remaining `elif` and `else` blocks are skipped, so executing the script has the following output:

```
You passed the class with a B.
Thanks for attending.
```

Let's break down the execution of the script step-by-step:

1. `grade` is assigned the value `85` in the line marked `1`.
2. `grade >= 90` is `False`, so the `if` statement marked `2` is skipped.
3. `grade >= 80` is `True`, so the block under the `elif` statement in line `3` is executed, and `"You passed the class with a B."` is printed.
4. The `elif` and `else` statements in lines `4` and `5` are skipped, since the condition for the `elif` statement on line `3` was met.
5. Finally, line `6` is executed and `"Thanks for attending."` is printed.

The `if`, `elif`, and `else` keywords are some of the most commonly used keywords in the Python language. They allow you to write code that responds to different conditions with different behavior.

The `if` statement allows you to solve more complex problems than code without any conditional logic. You can even nest an `if` statement inside another one to write code that handles tremendously complex logic!

Nested `if` Statements

Just like `for` and `while` loops can be nested within one another, you nest an `if` statement inside another to create complicated decision making structures.

Consider the following scenario. Two people play a one-on-one sport against one another. You must decide which of two players wins depending on the players' scores and the sport they are playing:

- If the two players are playing basketball, the player with the greatest score wins.
- If the two players are playing golf, then the player with the lowest score wins.
- In either sport, if the two scores are equal, the game is a draw.

The following program solves this using nested `if` statements:

```
sport = input("Enter a sport: ")
p1_score = int(input("Enter player 1 score: "))
p2_score = int(input("Enter player 2 score: "))

# 1
if sport.lower() == "basketball":
    if p1_score == p2_score:
        print("The game is a draw.")
    elif p1_score > p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")
# 2
elif sport.lower() == "golf":
    if p1_score == p2_score:
        print("The game is a draw.")
    elif p1_score < p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")
# 3
else:
    print("Unknown sport")
```

This program first asks the user to input a sport and the scores for two players.

In (#1), the string assigned to `sport` is converted to lowercase using `.lower()` and is compared it to the string `"basketball"`. This ensures that user input such as `"Basketball"` or `"BasketBall"` all get interpreted as the same sport.

Then the players scores are compared. If they are equal, the game is a draw. If player 1's score is larger than player 2's score, then player 1 wins the basketball game. Otherwise, player 2 wins the basketball game.

In (#2), the string assigned to `sport` is converted to lowercase gain and compared to the string `"golf"`. Then the players scores are checked again. If the two scores are equal, the game is a draw. If player 1's score is less than player 2's score, then player 1 wins. Otherwise, player 2 wins.

Finally, in (#3), if the `sport` variable is assigned to a string other than `"basketball"` or `"golf"`, the message `"Unknown sport"` is displayed.

The output of the script depends on the input value. Here's a sample execution using `"basketball"` as the sport:

```
Enter a sport: basketball
Player 1 score: 75
Player 2 score: 64
Player 1 wins.
```

Here's the output with the same player scores and the sport changed to `"golf"`:

```
Enter a sport: golf
Player 1 score: 75
Player 2 score: 64
Player 2 wins.
```

If you enter anything besides `basketball` or `golf` for the sport, the program displays `Unknown sport`.

All together, there are seven possible ways that the program can run, which are described in the following table:

Sport	Score values
`"basketball"`	`p1_score == p2_score`
`"basketball"`	`p1_score > p2_score`
`"basketball"`	`p1_score < p2_score`
`"golf"`	`p1_score == p2_score`
`"golf"`	`p1_score > p2_score`
`"golf"`	`p1_score < p2_score`

Sport	Score values
everything else	any combination

Nested `if` statements can create many possible ways that your code can run. If you have many deeply nested if statements (more than two levels), then the number of possible ways the code can execute grows quickly.

Note

The complexity that results from using deeply nested if statements may make it difficult to predict how your program will behave under given conditions.

For this reason, nested `if` statements are generally discouraged.

Let's see how we simplify the previous program by removing nested `if` statements.

First, regardless of the sport, the game is a draw if `p1_score` is equal to `p2_score`. So, we can move the check for equality out from the nested `if` statements under each sport to make a single `if` statement:

```
if p1_score == p2_score:
    print("The game is a draw.")

elif sport.lower() == "basketball":
    if p1_score > p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")

elif sport.lower() == "golf":
    if p1_score < p2_score:
        print("Player 1 wins.")
    else:
```

```
        print("Player 2 wins.")

else:
    print("Unknown sport.")
```

Now there are only six ways that the program can execute.

That's still quite a few ways. Can you think of any way to make the program simpler?

Here's one way to simplify it. Player 1 wins if the sport is basketball and their score is greater than player 2's score, or if the sport is golf and their score is less than player 2's score.

We can describe this with compound conditional expressions:

```
sport = sport.lower()
p1_wins_basketball = (sport == "basketball") and (p1_score > p2_score)
p1_wins_golf = (sport == "golf") and (p1_score < p2_score)
p1_wins = player1_wins_basketball or player1_wins_golf
```

This code is pretty dense, so let's walk through it one step at a time.

First the string assigned to `sport` is converted to all lowercase so that we can compare the value to other strings without worrying about errors due to case.

On the next line, we have a structure that might look a little strange. There is an assignment operator (`=`) followed by an expression with the equality comparator (`==`). This line evaluates the following compound logical expression and assigns its value to the `p1_wins_basketball` variable:

```
(sport == "basketball") and (p1_score > p2_score)
```

If `sport` is `"basketball"` and player 1's score is larger than player 2's score, then `p1_wins_basketball` is `True`.

Next, a similar operation is done for the `p1_wins_golf` variable. If `score`

is `"golf"` and player 1's score is less than player 2's score, then `p1_wins_golf` is `True`.

Finally, `p1_wins` will be `True` if player 1 wins the basketball game or the golf game, and will be `False` otherwise.

Using this code, you can simplify the program quite a bit:

```
if p1_score == p2_score:
    print("The game is a draw.")
elif (sport.lower() == "basketball") or (sport.lower() == "golf"):
    sport = sport.lower()
    p1_wins_basketball = (sport == "basketball") and (p1_score > p2_score)
    p1_wins_golf = (sport == "golf") and (p1_score < p2_score)
    p1_wins = p1_wins_basketball or p1_wins_golf
    if p1_wins:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")
else:
    print("Unknown sport")
```

In this revised version of the program, there are only four ways the program can execute, and the code is easier to understand.

Nested `if` statements are sometimes necessary. However, if you find yourself writing lots of nested `if` statements, it might be a good idea to stop and think about how you might simplify your code.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Write a script that prompts the user to enter a word using the `input()` function, stores that input in a variable, and then displays whether the length of that string is less than 5 characters, greater than 5 characters, or equal to 5 characters by using a set of `if`, `elif`