Table of Contents

Introduction

This document introduces how to create and use the compound statements: if statements, for statements and while statements. A list of videos are linked to provide more examples and explanations on compound statements.

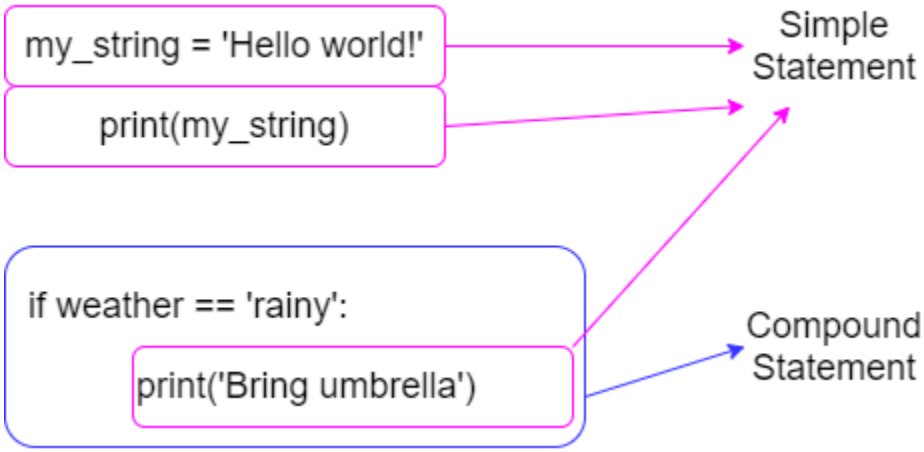
Videos

- Conditional Statements (0:00 - 7:03)
    - Covers what are conditional statements (if, elif and else), how it works, and includes examples
- Order of evaluating logical statements (0:00 - 8:04)
    - Covers the order in which conditional statements are evaluated, and includes examples
- While loop (0:00 - 9:26)
    - Covers what a while loop is and how it works by showing it through an example
- For loop (9:26 - 14:36)
    - Covers what a for loop is and how it works by showing it through an example
- Definite and indefinite repetition (0:00 - 10:07)
    - Covers what definite and indefinite repetitions mean, which loops (for and while) are suitable for which kind of repetition, and includes examples

1 Compound Statements

Recall that a program describes computations that a computer then evaluates, and Python programs are made up of statements, which are interpreted then evaluated by the machine. While there are many types of statements in Python, they are generally organized into two categories: simple statements and compound statements.

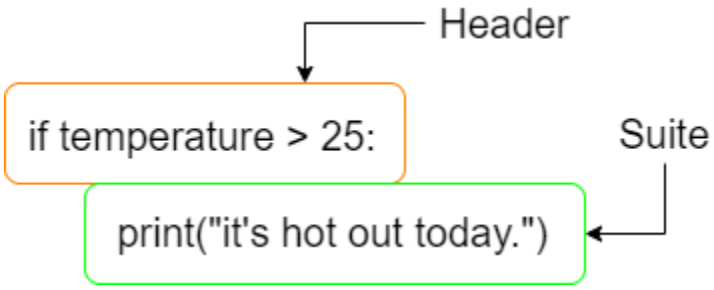
‘Simple statements’ are code that usually only spans one line, and are considered ‘simple’ in comparison to ‘compound statements’, which can span multiple lines.



Compound statements are made up of other statements, including simple statements and compound statements. They are often used to control when, if, and how many times certain lines of code should be evaluated.

Compound statements are composed of 'clauses', which are divided into a header and a 'suite'. The first line is called the header. The following lines, indented one level from the header, are called the suite.

As an illustration, an if statement, a kind of compound statement, might look like this:



We'll be covering some of the compound statements that you will encounter in this course; however, a full list of compound statements that exist in Python can be found in the compound statements section of the official Python documentation.

1.1 Conditional Statements (if, elif, else)

Conditional statements are the first kind of compound statements we will see in Python. Conditional statements allow the program to make decisions based on a given expression.

Generally, `if` statements have the following format:

`if <conditional expression>:`
`  <statements>`
`elif <conditional expression>:`
`  <statements>`
`elif <conditional expression>:`
`  <statements>`
`else:`
`  <statements>`
	Here, we have 4 clauses:
	• the `if` clause
	• an `elif` clause
	• another `elif` clause
	• and the `else` clause
	In a conditional statement, you will always have an `if` clause, but the `elif` and `else` clauses are optional. The clauses are evaluated in the order in which they are written in. Once a header of a clause evaluates to True, its suite will be evaluated, and the remaining clauses are skipped.

The suite of an `if` clause is evaluated only if the expression in its header evaluates to `True`.

`>>> temperature = -5`
`>>> if temperature < 0:`
`...  print("It's cold.")`
`... elif temperature > 30:`
`...  print("It's hot.")`
`... else:`
`...  print("It's warm.")`
`...`
`It's cold.`
	Here an int of *minus five* is bound to the identifier `temperature`. The expression in the header of the `if` clause `temperature < 0` evaluates to True since `-5` is less than `0`. The suite of the `if` clause is then evaluated, which displays the string `It's cold.` in the output. The other `elif` and `else` clauses will not be evaluated.

The suite of an `elif` (short for else if) clause is evaluated only if the expression in its header evaluates to `True`, *and* only if the expressions of the preceding `if` and `elif` headers have evaluated to `False`.

`>>> temperature = 32`
`>>> if temperature < 0:`
`...  print("It's cold.")`
`... elif temperature > 30:`
`...  print("It's hot.")`
`... else:`
`...  print("It's warm.")`
`...`
`It's cold.`
	Here an int of *thirty two* is bound to the identifier `temperature`. The expression in the header of the `if` clause `temperature < 0` evaluates to False since `32` is greater than `0`. Next, the expression in the header of the `elif` clause `temperature > 30` is checked, and it evaluates to True as `32` is larger than `30`. The suite of the `elif` clause is then evaluated, which displays the string `It's hot.` in the output. The `else` clause will not be evaluated.

The suite of an `else` clause is evaluated only if all preceding expressions of the `if` and `elif` headers have evaluated to `False`.

`>>> temperature = 20`
`>>> if temperature < 0:`
`...  print("It's cold.")`
`... elif temperature > 30:`
`...  print("It's hot.")`
`... else:`
`...  print("It's warm.")`
`...`
`It's warm.`
	Here an int of *twenty* is bound to the identifier `temperature`. The expression in the header of the `if` clause `temperature < 0` evaluates to False since `20` is greater than `0`. Next, the expression in the header of the `elif` clause `temperature > 30` is checked, and it evaluates to False as `20` is less than `30`. Lastly, the `suite` of the `else` clause is evaluated, which displays the string `It's warm.` in the output.

The order of the clauses is important if their conditions are not mutually exclusive. Consider the following example:

The intent of the code is to print `"It's hot today."` whenever the temperature is greater than 30. However in this case, even if the temperature is 40, this code segment will incorrectly print `"It's warm today."`. Once any clause in the entire if statement evaluates to True and its suite is evaluated, none of the other clauses are considered.

`>>> temperature = 35`
`>>> if temperature > 15:`
`...  print("It's warm today.")`
`... elif temperature > 30:`
`...  print("It's hot today.")`
`... elif temperature > 40:`
`...  print("It's very hot today.")`
`...`
`It's warm today.`
	Here the identifier `temperature` is bound to an `int` of *thirty-five*. The expression in the header of the `if` clause `temperature > 15` evaluates to `True` since 35 is greater than 15. The suite of this `if` clause will then be evaluated, and the string `"It's warm today."` will be displayed in the output. The remaining clauses will not be evaluated.

We can also have nested conditional statements, meaning that the suite of a conditional statement can include conditional statements. For example:

`>>> temperature = 15`
`>>> rainy = True`
`>>> if temperature > 0:`
`...  if rainy == True:`
`...  print("Bring an umbrella.")`
`... else:`
`...  print("It's cold.")`
`...`
`Bring an umbrella.`
	Here, the expression in the header of the first, outer `if` clause evaluates to True since `15` is greater than `0`. Within the suite of this outer `if` clause, there is another, inner, `if` clause. The bool True is bound to the identifier `rainy`, which makes the expression `rainy == True` evaluate to True. Thus the suite of this inner `if` clause is evaluated, displaying the string `Bring an umbrella.`. The `else` clause corresponding to the outer `if` statement will not be evaluated.

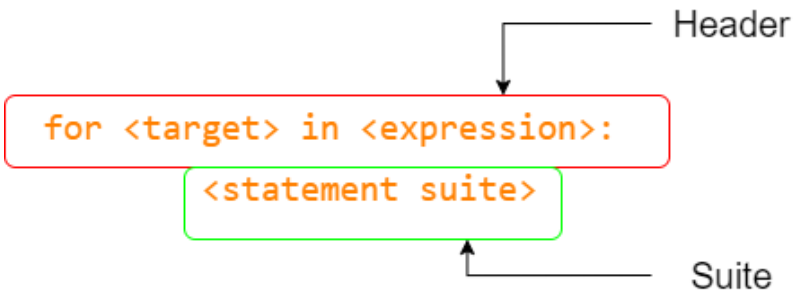
Here is an example of a slightly more complex nested conditional statement:

`>>> temperature = 15`
`>>> rainy = False`
`>>> windy = False`
`>>> if temperature > 0:`
`...  if rainy == True:`
`...  print("Bring an umbrella.")`
`...  elif windy == True:`
`...  print("It's windy.")`
`...  else:`
`...  print("It's nice.")`
`... elif temperature < 0:`
`...  print("It's cold.")`
`...`
`It's nice.`
	The expression in the header of the first, outer `if` clause evaluates to True since `15` is greater than `0`. Within the suite of this outer `if` clause, there is another, inner `if` clause. The bool False is bound to the identifier `rainy`, which makes the expression in the header of inner `if` clause `rainy == True` evaluate to False. Next, the expression in the header of the inner `elif` clause `windy == True` evaluates to False as the bool False is bound to the identifier `windy`. Lastly, the suite of the inner `else` clause is evaluated, displaying the string `It's nice.`. The `elif` clause corresponding to the outer `if` statement will not be evaluated.

1.2 `for` Statement

`for` statements are another type of compound statement. A `for` statement is a type of repetition statement (often also called "loop") that allows us to repeatedly evaluate a group of statements. For instance, we might want to perform operations on every element in a list, or repeat code a specific number of times. In both of these cases, a `for` statement is generally the best choice.

A `for` statement will look like this:



`For` statements are used to *iterate* over sequences, such as strings or lists or tuples in order. The statement suite, or "loop body", will operate on each element of the sequence.

The `for` statement works by binding the target (identifier) to each element of the sequence in order, and then evaluating the statement suite for that binding.

Syntactically, a `for` statement begins with the keyword `for`, followed by an identifier (called the target above), followed by the keyword `in`, then an expression, and then a colon (`:`) delimiter.

The expression in a `for` statement needs to evaluate to a sequence, such as a list or a string, or a range (introduced below).

1.2.1 `for` statements and strings

In the case of strings, the suite of the `for` statement will be evaluated for each character of that string.

In the following example, each time the header of the `for` loop is evaluated, the target identifier letter gets bound to the next character in the sequence `word` which is of type str. This character is printed by the statement in the suite of the `for` loop.

`>>> word = "Hello"`
`>>> for letter in word:`
`...  print("The current letter is " + letter)`
 	
`The current letter is H`
`The current letter is e`
`The current letter is l`
`The current letter is l`
`The current letter is o`
	In the **first iteration**, the identifier `letter` will be bound to the first character in the string, which is `'H'`. The suite of the `for` statement will then be evaluated, printing the result of the string concatenation expression `"The current letter is H"`.
	
	The **second iteration** of the `for` loop will follow the same pattern, except that the identifier `letter` will be bound to the second character in the string, which is `'e'`. The suite will be evaluated again to print the resulting expression `"The current letter is e"`.
	
	The same process will be repeated for each character of the string until it reaches the end of the string. In the last iteration, where `letter` is bound to the character `'o'`, once the evaluation of the suite has been completed, the `for` loop will come to an end.

1.2.2 `for` statements and lists and tuples

In the case of lists and tuples, the suite of the `for` statement will be evaluated for each element in the list or tuple. For example:

`>>> words = ['cat', 'computer', 'python']`
`>>> for word in words:`
`...  print(word + " has " + str(len(word)) + " letters")`
`...`
`cat has 3 letters`
`computer has 8 letters`
`python has 6 letters`
	For the first iteration, the identifier `word` will be bound to the 0th element of the list `words`, which in this case is `'cat'`.
	
	Next, the statement suite is evaluated. It will print `The word cat has 3 letters` as `word` is bound to `'cat'` at this time.
	
	After the statement suite is finished, the loop continues with the next element in the list, which is `'computer'`. It will bind the string object `'computer'` to the identifier `word` and evaluate the statement suite, printing off `The word computer as 8 letters`
	
	It will then do the same with `'python'`. After all the statement suite has finished evaluating with all elements of the list, then the `for` statement ends.

1.2.3 `for` statements with the `range` function

The range built-in function generates a range object, which is an **immutable sequence of numbers**.

There are two ways of creating a range object:

```
range(stop)
range(start, stop[, step])
```

The first is the more common method, and returns a sequence of numbers from 0 to `stop-1`.

You can convert the range object to a list object to verify this for yourself.

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

In the following example, the call to the built-in `range` function evaluates to an immutable sequence of 5 int objects. The suite of the for loop will be evaluated 5 times in this case.

`>>> for i in range(5):`
`...  print("Iteration", i)`
`Iteration 0`
`Iteration 1`
`Iteration 2`
`Iteration 3`
`Iteration 4`
	The call to the `range` function evaluates to a sequence of `int` objects from `0` to `4` inclusive.
	
	In the first iteration, the identifier `i` is bound to the first object in the sequence, which is `0`. The statement suite is then evaluated with `i` bound to `0`, printing `"Iteration 0"`.
	
	The second iteration continues with the next element in the sequence, which is the `int` object `1`. The statement suite will print "`Iteration 1`".
	
	This process continues until the last element, which is the `int` object 4. Once the statement suite is evaluated with `i` bound to this object, the for statement is complete.

The second way of creating a sequence using the `range` function is used when you want to begin the sequence at a number other than 0, or specify an increment other than 1. For example:

`>>> list(range(1, 9, 2))` `[1, 3, 5, 7]`	This yields a sequence of odd numbers from 1 to 9, non-inclusive. Remember that the `range` function always stops *before* the stopping number.

Using the `range` function with a for loop is handy when you need to access the index of a list, for instance if you want to change elements of the list. For example:

`>>> numbers = [1, 3, 5, 7, 9]`
`>>> for i in range(len(numbers)):`
`...  numbers[i] = numbers[i] - 1`
	This piece of code will change the elements of numbers to be:
	`[0, 2, 4, 6, 8]`

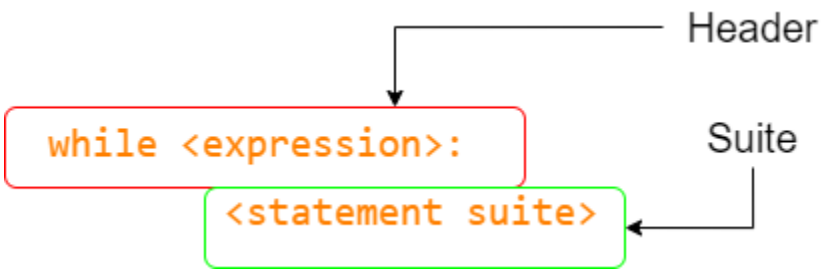
The `range` function will work as long as it is given the correct object type as an argument, an int. Remember that expressions will always evaluate to a certain type of object. Thus, we can also use expressions as arguments for the `range` function:

`>>> for i in range(len("ca" + "ke"):`
`...  print(i)`
`...`
`0`
`1`
`2`
`3`
	The expression within the len function is evaluated first, which results in a concatenated string of `cake`. Next the `len("cake")` will return an int of `4`. The for loop will then be evaluated a total of 4 times since the `range` function is given an int of `4` as its argument.

1.3 `while` Statement

`while` statements are another type of compound repetition statement that can be used to evaluate code repeatedly. The main difference between `while` statements and `for` statements is that `for` statements are used to loop a specific number of times; for instance, for every element in a list or string or tuple. In contrast, `while` statements are generally used to repeatedly evaluate their statement suites until a specific condition is reached.

The general syntax of a `while` statement is:



The header starts with the while keyword, then an expression, then a colon.

The expression in the header of the `while` statement needs to evaluate to either true or false. The statement suite will continually be evaluated while the expression is true and will not be evaluated when the expression is false.

In the following example, the suite of the `while` statement will continue to be evaluated as long as the expression `len(word) < 8` in the header of the `while` statement evaluates to True. Each time the expression evaluates to True, the statement in the suite will prompt the user to enter a word. After the user has entered a word, the program will 'loop' back to evaluate the expression in the header of the `while` statement. This will continue until the expression in the header of the while statement evaluates to false.

(Note: green will be used to show user input)
 	
`>>> word = ''`
`>>> while len(word) < 8:`
`...  word = input("Enter word: ")`
`...`
`Enter word: hello`
`Enter word: candy`
`Enter word: pen`
`Enter word: textbook`
`>>>`
	In the first iteration of the `while` loop, the identifier `word` is bound to an empty string, which has a length of 0 characters. The expression in the header of the `while` statement will evaluate to `True` since 0 is less than 8. The suite of the `while` loop will be evaluated, and the first time the user is prompted, the user enters `hello`.
	
	The program will 'loop' back to evaluate the expression in the header of the `while` statement. Since `hello` has 5 characters, the expression `len(word) < 8` in the header of the `while` statement will evaluate to `True`, as 5 is less than 8. The second iteration starts and the suite of the `while` loop is evaluated again.
	
	The same process is repeated when the user enters `candy` in the second iteration, and `pen`, in the third iteration.
	
	In the fourth iteration, the user enters the word `textbook`. This time when the program loops back to evaluate the expression in the header of the `while` statement, the expression evaluates to False since the word `textbook` has 8 characters in length, and 8 is not less than 8. Once the expression evaluates to False, the `while` loop ends.

When using `while` statements, be careful of infinite loops. Infinite loops happen when the expression in the header of the `while` statement never evaluates to False. Consequently the suite of the `while` statement will continuously be evaluated without any way to stop. Here is one example of an infinite loop:

`>>> clone = True`
`>>> while clone == True:`
`...  print("Make it double!")`
`...`
`Make it double!`
`Make it double!`
`Make it double!`
`Make it double!`
`Make it double!`
`Make it double!`
etc...
	The identifier `clone` is bound to the value `True`. The expression in the header of the `while` statement evaluates to True, which causes the suite of the `while` statement to be evaluated. However, since the value of the identifier `clone` is never changed in the suite of the `while` statement, the expression `clone == True` in the header of the `while` statement will always evaluate to `True`, and the suite will be evaluated repeatedly. The same can also happen if the `while` statement `while True:` is used.

There are many ways to change the code to prevent an infinite loop. The main idea is to ensure that there is some way to make a change within the suite of the `while` statement such that the expression in the header of the while statement evaluates to False at some point in time. Here are some ways of changing the code above such that it is no longer an infinite loop:

```
>>> clone = 5
>>> while clone > 0:
...     print("Make it double!")
...     clone = clone - 1
...
Make it double!
Make it double!
Make it double!
Make it double!
Make it double!
>>>
```

```
>>> clone = True
>>> times = 5
>>> while clone == True:
...     print("Make it double!")
...     times = times - 1
...     if times <= 0:
...         clone = False
...
Make it double!
Make it double!
Make it double!
Make it double!
Make it double!
>>>
```