The script should first prompt the user to enter a temperature in degrees Fahrenheit and then display the temperature converted to Celsius.

Then prompt the user to enter a temperature in degrees Celsius and display the temperature converted to Fahrenheit.

All converted temperatures should be rounded to 2 decimal places.

Here's a sample run of the program:

```
Enter a temperature in degrees F: 72
72 degrees F = 22.22 degrees C

Enter a temperature in degrees C: 37
37 degrees C = 98.60 degrees F
```

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

Leave feedback on this section »

6.4 Run in Circles

One of the great things about computers is that you can make them do the same thing over and over again, and they rarely complain or get tired.

A **loop** is a block of code that gets repeated over and over again either a specified number of times or until some condition is met. There are two kinds of loops in Python: **`while` loops** and **`for` loops**. In this section, you'll learn how to use both.

Let's start by looking at how `while` loops work.

The `while` Loop

`while` loops repeat a section of code while some condition is true. There are two parts to every `while` loop:

1. The **`while` statement** starts with the `while` keyword, followed by a **test condition**, and ends with a colon (:).
2. The **loop body** contains the code that gets repeated at each step of the loop. Each line is indented four spaces.

When a `while` loop is executed, Python evaluates the test condition and determines if it is true or false. If the test condition is true, then the code in the loop body is executed. Otherwise, the code in the body is skipped and the rest of the program is executed.

If the test condition is true and the body of the loop is executed, then once Python reaches the end of the body, it returns to the `while` statement and re-evaluates the test condition. If the test condition is still true, the body is executed again. If it is false, the body is skipped.

This process repeats over and over until the test condition fails, causing Python to loop over the code in the body of the `while` loop.

Let's look at an example. Type the following code into the interactive window:

```
>>> n = 1
>>> while n < 5:
...     print(n)
...     n = n + 1
...
1
2
3
4
```

First, the integer `1` is assigned to the variable `n`. Then a `while` loop is created with the test condition `n < 5`, which checks whether or not the value of `n` is less than `5`.

If `n` is less than `5`, the body of the loop is executed. There are two lines of code in the loop body. In the first line, the value of `n` is printed on the screen, and then `n` is **incremented** by `1` in the second line.

The loop execution takes place in five steps, described in the following table:

Step #	Value of `n`	Test Condition	What Happens
1	`1`	`1 < 5` (true)	`1` printed; `n` incremented to `2`
2	`2`	`2 < 5` (true)	`2` printed; `n` incremented to `3`
3	`3`	`3 < 5` (true)	`3` printed; `n` incremented to `4`
4	`4`	`4 < 5` (true)	`4` printed; `n` incremented to `5`
5	`5`	`5 < 5` (false)	Nothing printed; loop ends.

If you aren't careful, you can create an **infinite loop**. This happens when the test condition is always true. An infinite loop never terminates. The loop body keeps repeating forever.

Here's an example of an infinite loop:

```
>>> n = 1
>>> while n < 5:
...     print(n)
...
```

The only difference between this `while` loop and the previous one is that `n` is never incremented in the loop body. At each step of the loop, `n` is equal to `1`. That means the test condition `n < 5` is always true, and the number `1` is printed over and over again forever.

Note

Infinite loops aren't inherently bad. Sometimes they are exactly the kind of loop you need.

For example, code that interacts with hardware may use an infinite loop to constantly check whether or not a button or switch has been activated.

If you run a program that enters an infinite loop, you can force Python to quit by pressing `Ctrl+C`. Python stops running the program and raises a `KeyboardInterrupt` error:

```
Traceback (most recent call last):
  File "<pyshell#8>", line 2, in <module>
    print(n)
KeyboardInterrupt
```

Let's look at an example of a `while` loop in practice. One use of a `while` loop is to check whether or not user input meets some condition and, if not, repeatedly ask the user for new input until valid input is received.

For instance, the following program continuously asks a user for a positive number until a positive number is entered:

```
num = float(input("Enter a positive number: "))

while num <= 0:
    print("That's not a positive number!")
    num = float(input("Enter a positive number: "))
```

First, the user is prompted to enter a positive number. The test condition `num <= 0` determines whether or not `num` is less than or equal to `0`.

If `num` is positive, then the test condition fails. The body of the loop is skipped and the program ends.

Otherwise, if `num` is `0` or negative, the body of the loop executes. The

user is notified that their input was incorrect, and they are prompted again to enter a positive number.

`while` loops are perfect for repeating a section of code while some condition is met. They aren't well-suited, however, for repeating a section of code a specific number of times.

The `for` Loop

A `for` loop executes a section of code once for each item in a collection of items. The number of times that the code is executed is determined by the number of items in the collection.

Like its `while` counterpart, the `for` loop has two main parts:

1. The **`for` statement** begins with the `for` keyword, followed by a **membership expression**, and ends in a colon (:).
2. The **loop body** contains the code to be executed at each step of the loop, and is indented four spaces.

Let's look at an example. The following `for` loop prints each letter of the string `"Python"` one at a time:

```
for letter in "Python":
    print(letter)
```

In this example, the `for` statement is `for letter in "Python"`. The membership expression is `letter in "Python"`.

At each step of the loop, the variable `letter` is assigned the next letter in the string `"Python"`, and then the value of `letter` is printed.

The loops runs once for each character in the string `"Python"`, so the loop body executes six times. The following table summarizes the execution of this `for` loop:

Step #	Value of `letter`	What Happens
1	`"P"`	`P` is printed

Step #	Value of `letter`	What Happens
2	`"y"`	`y` is printed
3	`"t"`	`t` is printed
4	`"h"`	`h` is printed
5	`"o"`	`o` is printed
6	`"n"`	`n` is printed

To see why `for` loops are better for looping over collections of items, let's re-write the `for` loop in previous example as a `while` loop.

To do so, we can use a variable to store the index of the next character in the string. At each step of the loop, we'll print out the character at the current index and then increment the index.

The loop will stop once the value of the index variable is equal to the length of the string. Remember, indices start at `0`, so the last index of the string `"Python"` is `5`.

Here's how you might write that code:

```
word = "Python"
index = 0

while index < len(word):
    print(word[index])
    index = index + 1
```

That's significantly more complex than the `for` loop version!

Not only is the `for` loop less complex, the code itself looks more natural. It more closely resembles how you might describe the loop in English.

Note

You may sometimes hear people describe some code as being particularly "Pythonic." The term **Pythonic** is generally used to describe code that is clear, concise, and uses Python's built-in features to its advantage.

In these terms, using a `for` loop to loop over a collection of items is more Pythonic than using a `while` loop.

Sometimes it's useful to loop over a range of numbers. Python has a handy built-in function `range()` that produces just that — a range of numbers!

For example, `range(3)` returns the range of integers starting with `0` and up to, but not including, `3`. That is, `range(3)` is the range of numbers `0`, `1`, and `2`.

You can use `range(n)`, where `n` is any positive number, to execute a loop exactly `n` times. For instance, the following `for` loop prints the string `"Python"` three times:

```
for n in range(3):
    print("Python")
```

You can also give a range a starting point. For example, `range(1, 5)` is the range of numbers `1`, `2`, `3`, and `4`. The first argument is the starting number, and the second argument is the endpoint, which is not included in the range.

Using the two-argument version of `range()`, the following `for` loop prints the square of every number starting with `10` and up to, but not including, `20`:

```
for n in range(10, 20):
    print(n * n)
```

Let's look at a practical example. The following program asks the user to input an amount and then displays how to split that amount be-

tween 2, 3, 4, and 5 people:

```
amount = float(input("Enter an amount: "))

for num_people in range(2, 6):
    print(f"{num_people} people: ${amount / num_people:,.2f} each")
```

The `for` loop loops over the number `2`, `3`, `4`, and `5`, and prints the number of people and the amount each person should pay. The formatting specifier `,.2f` is used to format the amount as fixed-point number rounded to two decimal places and commas every three digits.

Running the program with the input `10` produces the following output:

```
Enter an amount: 10
2 people: $5.00 each
3 people: $3.33 each
4 people: $2.50 each
5 people: $2.00 each
```

`for` loops are generally used more often than `while` loops in Python. Most of the time, a `for` loop is more concise and easier to read than an equivalent `while` loop.

Nested Loops

As long as you indent the code correctly, you can even put loops inside of other loops.

Type the following into IDLE's interactive window:

```
for n in range(1, 4):
    for j in range(4, 7):
        print(f"n = {n} and j = {j}")
```

When Python enters the body of the first `for` loop, the variable `n` is assigned the value `1`. Then the body of the second `for` loop is executed and `j` is assigned the value `4`. The first thing printed is `n = 1 and j = 4`.

After executing the `print()` function, Python returns to the *inner* `for` loop, assigns to `j` the value of `5`, and then prints `n = 1 and j = 5`. Python doesn't return the outer `for` loop because the inner `for` loop, which is inside the body of the outer `for` loop, isn't done executing.

Next, `j` is assigned the value `6` and Python prints `n = 1 and j = 6`. At this point, the inner `for` loop is done executing, so control returns to the outer `for` loop.

The variable `n` gets assigned the value `2`, and the inner `for` loop executes a second time. That is, `j` is assigned the value `4` and `n = 2 and j = 4` is printed to the console.

The two loops continue to execute in this fashion, and the final output looks like this:

```
n = 1 and j = 4
n = 1 and j = 5
n = 1 and j = 6
n = 2 and j = 4
n = 2 and j = 5
n = 2 and j = 6
n = 3 and j = 4
n = 3 and j = 5
n = 3 and j = 6
```

A loop inside of another loop is called a **nested loop**, and they come up more often than you might expect. You can nest `while` loops inside of `for` loops, and vice versa, and even nest loops more than two levels deep!

Important

Nesting loops inherently increases the complexity of your code, as you can see by the dramatic increase in the number of steps run in the previous example compared to examples with a single `for` loop.

Using nested loops is sometimes the only way to get something done, but too many nested loops can have a negative effect on a program's performance.

Loops are a powerful tool. They tap into one of the greatest advantages computers provide as tools for computation: the ability to repeat the same task a vast number of times without tiring and without complaining.

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 `for` loop that prints out the integers 2 through 10, each on a new line, by using the `range()` function.
2. Use a `while` loop that prints out the integers 2 through 10 (Hint: You'll need to create a new integer first.)
3. Write a function called `doubles()` that takes one number as its input and doubles that number. Then use the `doubles()` function in a loop to double the number `2` three times, displaying each result on a separate line. Here is some sample output:

```
4
8
16
```

Leave feedback on this section »