

# Introducing while Loops

The `for` loop takes a collection of items and executes a block of code once for each item in the collection. In contrast, the `while` loop runs as long as, or *while*, a certain condition is true.

## The `while` Loop in Action

You can use a `while` loop to count up through a series of numbers. For example, the following `while` loop counts from 1 to 5:

`counting.py`

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

In the first line, we start counting from 1 by assigning `current_number` the value 1. The `while` loop is then set to keep running as long as the value of `current_number` is less than or equal to 5. The code inside the loop prints the value of `current_number` and then adds 1 to that value with `current_number += 1`. (The `+=` operator is shorthand for `current_number = current_number + 1`.)

Python repeats the loop as long as the condition `current_number <= 5` is true. Because 1 is less than 5, Python prints 1 and then adds 1, making the current number 2. Because 2 is less than 5, Python prints 2 and adds 1 again, making the current number 3, and so on. Once the value of `current_number` is greater than 5, the loop stops running and the program ends:

```
1
2
3
4
5
```

The programs you use every day most likely contain `while` loops. For example, a game needs a `while` loop to keep running as long as you want to keep playing, and so it can stop running as soon as you ask it to quit. Programs wouldn't be fun to use if they stopped running before we told them to or kept running even after we wanted to quit, so `while` loops are quite useful.

## Letting the User Choose When to Quit

We can make the `parrot.py` program run as long as the user wants by putting most of the program inside a `while` loop. We'll define a *quit value* and then keep the program running as long as the user has not entered the quit value:

`parrot.py`

```
❶ prompt = "\nTell me something, and I will repeat it back to you."
prompt += "\nEnter 'quit' to end the program."
```

```
❶ message = ""  
❷ while message != 'quit':  
    message = input(prompt)  
    print(message)
```

---

At ❶, we define a prompt that tells the user their two options: entering a message or entering the quit value (in this case, 'quit'). Then we set up a variable `message` ❷ to keep track of whatever value the user enters. We define `message` as an empty string, "", so Python has something to check the first time it reaches the `while` line. The first time the program runs and Python reaches the `while` statement, it needs to compare the value of `message` to 'quit', but no user input has been entered yet. If Python has nothing to compare, it won't be able to continue running the program. To solve this problem, we make sure to give `message` an initial value. Although it's just an empty string, it will make sense to Python and allow it to perform the comparison that makes the `while` loop work. This `while` loop ❸ runs as long as the value of `message` is not 'quit'.

The first time through the loop, `message` is just an empty string, so Python enters the loop. At `message = input(prompt)`, Python displays the prompt and waits for the user to enter their input. Whatever they enter is assigned to `message` and printed; then, Python reevaluates the condition in the `while` statement. As long as the user has not entered the word 'quit', the prompt is displayed again and Python waits for more input. When the user finally enters 'quit', Python stops executing the `while` loop and the program ends:

---

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello everyone!  
Hello everyone!
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello again.  
Hello again.
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. quit  
quit
```

---

This program works well, except that it prints the word 'quit' as if it were an actual message. A simple `if` test fixes this:

---

```
prompt = "\nTell me something, and I will repeat it back to you:"  
prompt += "\nEnter 'quit' to end the program."  
  
message = ""  
while message != 'quit':  
    message = input(prompt)  
  
    if message != 'quit':  
        print(message)
```

---

Now the program makes a quick check before displaying the message and only prints the message if it does not match the quit value:

---

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello everyone!  
Hello everyone!
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. Hello again.  
Hello again.
```

```
Tell me something, and I will repeat it back to you:  
Enter 'quit' to end the program. quit
```

---

## Using a Flag

In the previous example, we had the program perform certain tasks while a given condition was true. But what about more complicated programs in which many different events could cause the program to stop running?

For example, in a game, several different events can end the game. When the player runs out of ships, their time runs out, or the cities they were supposed to protect are all destroyed, the game should end. It needs to end if any one of these events happens. If many possible events might occur to stop the program, trying to test all these conditions in one `while` statement becomes complicated and difficult.

For a program that should run only as long as many conditions are true, you can define one variable that determines whether or not the entire program is active. This variable, called a *flag*, acts as a signal to the program. We can write our programs so they run while the flag is set to True and stop running when any of several events sets the value of the flag to False. As a result, our overall `while` statement needs to check only one condition: whether or not the flag is currently True. Then, all our other tests (to see if an event has occurred that should set the flag to False) can be neatly organized in the rest of the program.

Let's add a flag to *parrot.py* from the previous section. This flag, which we'll call `active` (though you can call it anything), will monitor whether or not the program should continue running:

---

```
prompt = "\nTell me something, and I will repeat it back to you:  
prompt += "\nEnter 'quit' to end the program. "
```

```
❶ active = True  
❷ while active:  
    message = input(prompt)  
  
    ❸ if message == 'quit':  
        active = False  
    ❹ else:  
        print(message)
```

---

We set the variable `active` to `True` ❶ so the program starts in an active state. Doing so makes the `while` statement simpler because no comparison is made in the `while` statement itself; the logic is taken care of in other parts of the program. As long as the active variable remains `True`, the loop will continue running ❷.

In the `if` statement inside the `while` loop, we check the value of `message` once the user enters their input. If the user enters 'quit' ❸, we set `active` to `False`, and the `while` loop stops. If the user enters anything other than 'quit' ❹, we print their input as a message.

This program has the same output as the previous example where we placed the conditional test directly in the `while` statement. But now that we have a flag to indicate whether the overall program is in an active state, it would be easy to add more tests (such as `elif` statements) for events that should cause `active` to become `False`. This is useful in complicated programs like games in which there may be many events that should each make the program stop running. When any of these events causes the active flag to become `False`, the main game loop will exit, a *Game Over* message can be displayed, and the player can be given the option to play again.

## Using `break` to Exit a Loop

To exit a `while` loop immediately without running any remaining code in the loop, regardless of the results of any conditional test, use the `break` statement. The `break` statement directs the flow of your program; you can use it to control which lines of code are executed and which aren't, so the program only executes code that you want it to, when you want it to.

For example, consider a program that asks the user about places they've visited. We can stop the `while` loop in this program by calling `break` as soon as the user enters the 'quit' value:

`cities.py`

```
prompt = "\nPlease enter the name of a city you have visited:"  
prompt += "\nEnter 'quit' when you are finished.) "
```

```
❶ while True:  
    city = input(prompt)  
  
    if city == 'quit':  
        break  
    else:  
        print(f"I'd love to go to {city.title()}!")
```

A loop that starts with `while True` ❶ will run forever unless it reaches a `break` statement. The loop in this program continues asking the user to enter the names of cities they've been to until they enter 'quit'. When they enter 'quit', the `break` statement runs, causing Python to exit the loop:

```
Please enter the name of a city you have visited:  
(Enter 'quit' when you are finished.) New York  
I'd love to go to New York!
```

Please enter the name of a city you have visited:  
(Enter 'quit' when you are finished.) **San Francisco**  
I'd love to go to San Francisco!

Please enter the name of a city you have visited:  
(Enter 'quit' when you are finished.) **quit**

**NOTE**

You can use the `break` statement in any of Python's loops. For example, you could use `break` to quit a `for` loop that's working through a list or a dictionary.

## Using `continue` in a Loop

Rather than breaking out of a loop entirely without executing the rest of its code, you can use the `continue` statement to return to the beginning of the loop based on the result of a conditional test. For example, consider a loop that counts from 1 to 10 but prints only the odd numbers in that range:

*counting.py*

```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue
    print(current_number)
```

First we set `current_number` to 0. Because it's less than 10, Python enters the `while` loop. Once inside the loop, we increment the count by 1 at ❶, so `current_number` is 1. The `if` statement then checks the modulo of `current_number` and 2. If the modulo is 0 (which means `current_number` is divisible by 2), the `continue` statement tells Python to ignore the rest of the loop and return to the beginning. If the current number is not divisible by 2, the rest of the loop is executed and Python prints the current number:

```
1
3
5
7
9
```

## Avoiding Infinite Loops

Every `while` loop needs a way to stop running so it won't continue to run forever. For example, this counting loop should count from 1 to 5:

*counting.py*

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

But if you accidentally omit the line `x += 1` (as shown next), the loop will run forever:

```
# This loop runs forever!
x = 1
while x <= 5:
    print(x)
```

Now the value of `x` will start at 1 but never change. As a result, the conditional test `x <= 5` will always evaluate to True and the `while` loop will run forever, printing a series of 1s, like this:

```
1
1
1
1
1
--snip--
```

Every programmer accidentally writes an infinite `while` loop from time to time, especially when a program's loops have subtle exit conditions. If your program gets stuck in an infinite loop, press **CTRL-C** or just close the terminal window displaying your program's output.

To avoid writing infinite loops, test every `while` loop and make sure the loop stops when you expect it to. If you want your program to end when the user enters a certain input value, run the program and enter that value. If the program doesn't end, scrutinize the way your program handles the value that should cause the loop to exit. Make sure at least one part of the program can make the loop's condition `False` or cause it to reach a `break` statement.

**NOTE**

*Sublime Text and some other editors have an embedded output window. This can make it difficult to stop an infinite loop, and you might have to close the editor to end the loop. Try clicking in the output area of the editor before pressing **CTRL-C**, and you should be able to cancel an infinite loop.*

### TRY IT YOURSELF

**7-4. Pizza Toppings:** Write a loop that prompts the user to enter a series of pizza toppings until they enter a 'quit' value. As they enter each topping, print a message saying you'll add that topping to their pizza.

**7-5. Movie Tickets:** A movie theater charges different ticket prices depending on a person's age. If a person is under the age of 3, the ticket is free; if they are between 3 and 12, the ticket is \$10; and if they are over age 12, the ticket is \$15. Write a loop in which you ask users their age, and then tell them the cost of their movie ticket.

*(continued)*

**7-6. Three Exits:** Write different versions of either Exercise 7-4 or Exercise 7-5 that do each of the following at least once:

- Use a conditional test in the `while` statement to stop the loop.
- Use an active variable to control how long the loop runs.
- Use a `break` statement to exit the loop when the user enters a 'quit' value.

**7-7. Infinity:** Write a loop that never ends, and run it. (To end the loop, press **CTRL-C** or close the window displaying the output.)

## Using a `while` Loop with Lists and Dictionaries

So far, we've worked with only one piece of user information at a time. We received the user's input and then printed the input or a response to it. The next time through the `while` loop, we'd receive another input value and respond to that. But to keep track of many users and pieces of information, we'll need to use lists and dictionaries with our `while` loops.

A `for` loop is effective for looping through a list, but you shouldn't modify a list inside a `for` loop because Python will have trouble keeping track of the items in the list. To modify a list as you work through it, use a `while` loop. Using `while` loops with lists and dictionaries allows you to collect, store, and organize lots of input to examine and report on later.

### Moving Items from One List to Another

Consider a list of newly registered but unverified users of a website. After we verify these users, how can we move them to a separate list of confirmed users? One way would be to use a `while` loop to pull users from the list of unconfirmed users as we verify them and then add them to a separate list of confirmed users. Here's what that code might look like:

```
confirmed  
_users.py # Start with users that need to be verified,  
# and an empty list to hold confirmed users.  
❶ unconfirmed_users = ['alice', 'brian', 'candace']  
confirmed_users = []  
  
❷ while unconfirmed_users:  
❸     current_user = unconfirmed_users.pop()  
  
❹     print(f"Verifying user: {current_user.title()}")  
     confirmed_users.append(current_user)
```

```
# Display all confirmed users.
print("\nThe following users have been confirmed:")
for confirmed_user in confirmed_users:
    print(confirmed_user.title())
```

We begin with a list of unconfirmed users at ❶ (Alice, Brian, and Candace) and an empty list to hold confirmed users. The while loop at ❷ runs as long as the list `unconfirmed_users` is not empty. Within this loop, the `pop()` method at ❸ removes unverified users one at a time from the end of `unconfirmed_users`. Here, because Candace is last in the `unconfirmed_users` list, her name will be the first to be removed, assigned to `current_user`, and added to the `confirmed_users` list at ❹. Next is Brian, then Alice.

We simulate confirming each user by printing a verification message and then adding them to the list of confirmed users. As the list of unconfirmed users shrinks, the list of confirmed users grows. When the list of unconfirmed users is empty, the loop stops and the list of confirmed users is printed:

---

```
Verifying user: Candace
Verifying user: Brian
Verifying user: Alice
```

The following users have been confirmed:

```
Candace
Brian
Alice
```

---

## **Removing All Instances of Specific Values from a List**

In Chapter 3 we used `remove()` to remove a specific value from a list. The `remove()` function worked because the value we were interested in appeared only once in the list. But what if you want to remove all instances of a value from a list?

Say you have a list of pets with the value 'cat' repeated several times. To remove all instances of that value, you can run a while loop until 'cat' is no longer in the list, as shown here:

---

```
pets.py      pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
              print(pets)

              while 'cat' in pets:
                  pets.remove('cat')

              print(pets)
```

---

We start with a list containing multiple instances of 'cat'. After printing the list, Python enters the while loop because it finds the value 'cat' in the list

at least once. Once inside the loop, Python removes the first instance of 'cat', returns to the while line, and then reenters the loop when it finds that 'cat' is still in the list. It removes each instance of 'cat' until the value is no longer in the list, at which point Python exits the loop and prints the list again:

```
['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
['dog', 'dog', 'goldfish', 'rabbit']
```