

Python Programming

Module 7: User Input and while loops

Learning objectives

1. How to use `input()` to allow users to provide their own information in your programs.
2. Work with both text and numerical input and how to use while loops to make your programs run as long as your users want them to.
3. controlling the flow of a while loop by setting an active flag, using the `break` statement, and using the `continue` statement.
4. Using a `while` loop to move items from one list to another and how to remove all instances of a value from a list.
5. How while loops can be used with dictionaries.

User Input and while loops

How the `input()` function works

The `input()` function pauses your program and waits for the user to enter some text. Once Python receives the user's input, it stores it in a variable to make it convenient for you to work with.

For example, the following program asks the user to enter some text, then displays that message back to the user:

```
message = input("Tell me something, and I will repeat it back to  
                you:")  
print(message)
```

The `input()` function takes one argument: the *prompt*, or instructions, that we want to display to the user so they know what to do. In this example, when Python runs the first line, the user sees the prompt Tell me something, and I will repeat it back to you:. The program waits while the user enters their response and continues after the user presses enter. The response is stored in the variable `message`, then `print(message)` displays the input back to the user:

```
Tell me something, and I will repeat it back to you: Hello  
                everyone!  
Hello everyone!
```

Writing Clear Prompts

Any statement that tells the user what to enter should work. For example:

```
name = input("Please enter your name: ")  
print("Hello, " + name + "!")
```

Add a space at the end of your prompts (after the colon in the preceding example) to separate the prompt from the user's response and to make it clear to your user where to enter their text. For example:

```
Please enter your name: Eric  
Hello, Eric!
```

You can store your prompt in a variable and pass that variable to the `input()` function. This allows you to build your prompt over several lines, then write a clean `input()` statement.

```
prompt = "If you tell us who you are, we can personalize the  
          messages you see."  
prompt += "\nWhat is your first name? "  
  
name = input(prompt)  
print("\nHello, " + name + "!")
```

This example shows one way to build a multi-line string. The first line stores the first part of the message in the variable `prompt`. In the second line, the operator `+=` takes the string that was stored in `prompt` and adds the new string onto the end.

The prompt now spans two lines, again with space after the question mark for clarity:

```
If you tell us who you are, we can personalize the messages you
see.
What is your first name? Eric

Hello, Eric!
```

Using `int()` to Accept Numeric Input

When you use the `input()` function, Python interprets everything the user enters as a string. Consider the following interpreter session, which asks for the user's age:

```
>>> age = input("How old are you? ")
How old are you? 21
>>> age
'21'
```

The user enters the number 21, but when we ask Python for the value of `age`, it returns `'21'`, the string representation of the numerical value entered. But if you try to use the input as a number, you'll get an error:

```
>>> age = input("How old are you? ")
How old are you? 21
1 >>> age >= 18
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
2 TypeError: unorderable types: str() >= int()
```

We can resolve this issue by using the `int()` function, which tells Python to treat the input as a numerical value. The `int()` function converts a string representation of a number to a numerical representation, as shown here:

```
>>> age = input("How old are you? ")
How old are you? 21
1 >>> age = int(age)
>>> age >= 18 True
```

In this example, when we enter 21 at the prompt, Python interprets the number as a string, but the value is then converted to a numerical representation by `int()` at 1. Now Python can run the conditional test: it compares `age` (which now contains the numerical value 21) and 18 to see if `age` is greater than or equal to 18. This test evaluates to `True`.

How do you use the `int()` function in an actual program? Consider a program that determines whether people are tall enough to ride a roller coaster:

```
height = input("How tall are you, in inches? ")
height = int(height)

if height >= 36:
    print("\nYou're tall enough to ride!")
else:
    print("\nYou'll be able to ride when you're a little older.")
```

The program can compare height to 36 because `height = int(height)` converts the input value to a numerical representation before the comparison is made. If the number entered is greater than or equal to 36, we tell the user that they're tall enough:

```
How tall are you, in inches? 71

You're tall enough to ride!
```

The Modulo Operator

A useful tool for working with numerical information is the *modulo operator* (`%`), which divides one number by another number and returns the remainder:

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
>>> 7 % 3
1
```

The modulo operator doesn't tell you how many times one number fits into another; it just tells you what the remainder is.

When one number is divisible by another number, the remainder is 0, so the modulo operator always returns 0. You can use this fact to determine if a number is even or odd:

```
number = input("Enter a number, and I'll tell you if it's even or
odd: ")
number = int(number)

if number % 2 == 0:
    print("\nThe number " + str(number) + " is even.")
else:
```

```
print("\nThe number " + str(number) + " is odd.")
```

Even numbers are always divisible by two, so if the modulo of a number and two is zero (here, if `number % 2 == 0`) the number is even. Otherwise, it's odd.

```
Enter a number, and I'll tell you if it's even or odd: 42
The number 42 is even.
```

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:

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

In the first line, we start counting from 1 by setting the value of `current_number` to 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
```

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:

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

```

3 while message != 'quit':
    message = input(prompt)
    print(message)

```

At 1, 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 2 to store 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 3 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 stored in message and printed; then, Python re-evaluates 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 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. "

1 active = True
2 while active:
    message = input(prompt)

3 if message == 'quit':
    active = False
4 else:
    print(message)
```

We set the variable `active` to `True` 1 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 2.

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' at 3, we set active to False, and the while loop stops. If the user enters anything other than 'quit' at 4, 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:

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

1 while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I'd love to go to " + city.title() + "!")
```

A loop that starts with while True 1 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
```


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:

```
current_number = 0
while current_number < 10:
1     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 1, 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 for- ever. For example, this counting loop should count from 1 to 5:

```
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
--snip--
```

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.

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:

```
# Start with users that need to be verified,
# and an empty list to hold confirmed users.
1 unconfirmed_users = ['alice', 'brian', 'candace']
  confirmed_users = []

# Verify each user until there are no more unconfirmed users.
# Move each verified user into the list of confirmed users.
2 while unconfirmed_users:
3     current_user = unconfirmed_users.pop()

    print("Verifying user: " + current_user.title())
4     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 1 (Alice, Brian, and Candace) and an empty list to hold confirmed users. The while loop at 2 runs as long as the list `unconfirmed_users` is not empty. Within this loop, the `pop()` function at 3 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, stored in `current_user`, and added to the `confirmed_users` list at 4. 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

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 = ['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 re-enters 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']
```

Filling a Dictionary with User Input

You can prompt for as much input as you need in each pass-through a while loop. Let's make a polling program in which each pass through the loop prompts for the participant's name and response. We'll store the data we gather in a dictionary, because we want to connect each response with a particular user:

```
responses = {}

# Set a flag to indicate that polling is active.
polling_active = True

while polling_active:
    # Prompt for the person's name and response.
1    name = input("\nWhat is your name? ")
    response = input("Which mountain would you like to climb
someday? ")

    # Store the response in the dictionary:
2    responses[name] = response

    # Find out if anyone else is going to take the poll.
3    repeat = input("Would you like to let another person respond?
(yes/ no) ")
    if repeat == 'no':
        polling_active = False

# Polling is complete. Show the results.
print("\n--- Poll Results ---")
4 for name, response in responses.items():
    print(name + " would like to climb " + response + ".")
```

The program first defines an empty dictionary (`responses`) and sets a flag (`polling_active`) to indicate that polling is active. As long as `polling_active` is `True`, Python will run the code in the while loop.

Within the loop, the user is prompted to enter their username and a mountain they'd like to climb 1. That information is stored in the `responses` dictionary 2, and the user is asked whether or not to keep the poll running 3. If they enter yes, the program enters the while loop again. If they enter no, the `polling_active` flag is set to `False`, the while loop stops running, and the final code block at 4 displays the results of the poll.

If you run this program and enter sample responses, you should see output like this:

```
What is your name? Eric
Which mountain would you like to climb someday? Denali
Would you like to let another person respond? (yes/ no) yes
```

What is your name? **Lynn**

Which mountain would you like to climb someday? **Devil's Thumb**

Would you like to let another person respond? (yes/ no) **no**

--- Poll Results ---

Lynn would like to climb Devil's Thumb.

Eric would like to climb Denali.

Exercises

1. Write a Python program to find those numbers which are divisible by 7 and multiple of 5, between 1500 and 2700 (both included).
2. Write a Python program to construct the following pattern, using a nested loop number. Go to the editor

Expected Output:

```
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

3. Write a python program which mimics the calculator, and can accept inputs from user, and also can perform all the basic calculations.
4. Write a python program which takes the date as input and returns the day on that date.
5. Write a python program to find the median of three user provided numbers and print it.
6. Write a program that asks five times to guess the lucky number using `while` loop. The program asks for five guesses (no matter whether the correct number was guessed or not). If the correct number is guessed, the program outputs "Good guess!", otherwise it outputs "Try again!". After the fifth guess it stops and prints "Game over."
7. Read an integer N . For all non-negative integers $i < N$, print i^2 .