05-while-loops-and-user-input

September 2, 2023

1 Loops, Iteration Schemas and Input

While loops are really useful because they let your program run until a user decides to quit the program. They set up an infinite loop that runs until the user does something to end the loop. This section also introduces the first way to get input from your program's users.

2 Contents

2.1 The FOR (iteration) loop

The `for` loop statement is the most widely used iteration mechanisms in Python.

- Almost every structure in Python can be iterated (*element by element*) by a `for` loop
  - a list, a tuple, a dictionary, … (more details will follows)
- In Python, also `while` loops are permitted, but `for` is the one you would see (and use) most of the time!

3 What is a while loop?

A while loop tests an initial condition. If that condition is true, the loop starts executing. Every time the loop finishes, the condition is reevaluated. As long as the condition remains true, the loop keeps executing. As soon as the condition becomes false, the loop stops executing.

3.1 General syntax

[ ]:
```python
# Set an initial condition.
game_active = True

# Set up the while loop.
while game_active:
    # Run the game.
    # At some point, the game ends and game_active will be set to False.
    #   When that happens, the loop will stop executing.

# Do anything else you want done after the loop runs.
```

- Every while loop needs an initial condition that starts out true.
- The `while` statement includes a condition to test.
- All of the code in the loop will run as long as the condition remains true.
- As soon as something in the loop changes the condition such that the test no longer passes, the loop stops executing.
- Any code that is defined after the loop will run at this point.

3.2 Example

Here is a simple example, showing how a game will stay active as long as the player has enough power.

[3]:
```python
# The player's power starts out at 5.
power = 5

# The player is allowed to keep playing as long as their power is over 0.
while power > 0:
    print("You are still playing, because your power is %d." % power)
    # Your game code would go here, which includes challenges that make it
    #   possible to lose power.
    # We can represent that by just taking away from the power.
    power = power - 1

print("\nOh no, your power dropped to 0! Game Over.")
```

```
You are still playing, because your power is 5.
You are still playing, because your power is 4.
You are still playing, because your power is 3.
You are still playing, because your power is 2.
You are still playing, because your power is 1.

Oh no, your power dropped to 0! Game Over.
```

top

3.3 Exercises

Growing Strength

- Make a variable called strength, and set its initial value to 5.
- Print a message reporting the player's strength.
- Set up a while loop that runs until the player's strength increases to a value such as 10.
- Inside the while loop, print a message that reports the player's current strength.
- Inside the while loop, write a statement that increases the player's strength.
- Outside the while loop, print a message reporting that the player has grown too strong, and that they have moved up to a new level of the game.
- Bonus: Play around with different cutoff levels for the value of *strength*, and play around with different ways to increase the strength value within the while loop.

[ ]:
```
# Ex 6.1 : Growing Strength

# put your code here
```

top

4 Accepting user input

Almost all interesting programs accept input from the user at some point. You can start accepting user input in your programs by using the `input()` function. The input function displays a messaget to the user describing the kind of input you are looking for, and then it waits for the user to enter a value. When the user presses Enter, the value is passed to your variable.

4.1 General syntax

The general case for accepting input looks something like this:

[ ]:
```
# Get some input from the user.
variable = input('Please enter a value: ')
# Do something with the value that was entered.
```

You need a variable that will hold whatever value the user enters, and you need a message that will be displayed to the user.

4.2 Example

In the following example, we have a list of names. We ask the user for a name, and we add it to our list of names.

[7]:
```
# Start with a list containing several names.
names = ['guido', 'tim', 'jesse']

# Ask the user for a name.
new_name = input("Please tell me someone I should know: ")

# Add the new name to our list.
```

```
names.append(new_name)

# Show that the name has been added to the list.
print(names)
```

```
Please tell me someone I should know: jessica
['guido', 'tim', 'jesse', 'jessica']
```

4.3 Exercises

Game Preferences

- Make a list that includes 3 or 4 games that you like to play.
- Print a statement that tells the user what games you like.
- Ask the user to tell you a game they like, and store the game in a variable such as `new_game`.
- Add the user's game to your list.
- Print a new statement that lists all of the games that we like to play (*we* means you and your user).

[ ]:
```
# Ex 6.2 : Game Preferences

# put your code here
```

top

5 Using while loops to keep your programs running

Most of the programs we use every day run until we tell them to quit, and in the background this is often done with a while loop.

Here is an example of how to let the user enter an arbitrary number of names.

[9]:
```
# Start with an empty list. You can 'seed' the list with
#  some predefined values if you like.
names = []

# Set new_name to something other than 'quit'.
new_name = ''

# Start a loop that will run until the user enters 'quit'.
while new_name != 'quit':
    # Ask the user for a name.
    new_name = input("Please tell me someone I should know, or enter 'quit': ")

    # Add the new name to our list.
    names.append(new_name)

# Show that the name has been added to the list.
print(names)
```

```
Please tell me someone I should know, or enter 'quit': guido
Please tell me someone I should know, or enter 'quit': jesse
Please tell me someone I should know, or enter 'quit': jessica
Please tell me someone I should know, or enter 'quit': tim
Please tell me someone I should know, or enter 'quit': quit
['guido', 'jesse', 'jessica', 'tim', 'quit']
```

That worked, except we ended up with the name 'quit' in our list. We can use a simple `if` test to eliminate this bug:

[10]:
```python
# Start with an empty list. You can 'seed' the list with
#  some predefined values if you like.
names = []

# Set new_name to something other than 'quit'.
new_name = ''

# Start a loop that will run until the user enters 'quit'.
while new_name != 'quit':
    # Ask the user for a name.
    new_name = input("Please tell me someone I should know, or enter 'quit': ")

    # Add the new name to our list.
    if new_name != 'quit':
        names.append(new_name)

# Show that the name has been added to the list.
print(names)
```

```
Please tell me someone I should know, or enter 'quit': guido
Please tell me someone I should know, or enter 'quit': jesse
Please tell me someone I should know, or enter 'quit': jessica
Please tell me someone I should know, or enter 'quit': tim
Please tell me someone I should know, or enter 'quit': quit
['guido', 'jesse', 'jessica', 'tim']
```

This is pretty cool! We now have a way to accept input from users while our programs run, and we have a way to let our programs run until our users are finished working.

5.1 Exercises

Many Games

- Modify *Game Preferences* so your user can add as many games as they like.

[ ]:
```python
# Ex 6.3 : Many Games

# put your code here
```

top

6 Using while loops to make menus

You now have enough Python under your belt to offer users a set of choices, and then respond to those choices until they choose to quit.

Let's look at a simple example, and then analyze the code:

[33]:
```python
# Give the user some context.
print("\nWelcome to the nature center. What would you like to do?")

# Set an initial value for choice other than the value for 'quit'.
choice = ''

# Start a loop that runs until the user enters the value for 'quit'.
while choice != 'q':
    # Give all the choices in a series of print statements.
    print("\n[1] Enter 1 to take a bicycle ride.")
    print("[2] Enter 2 to go for a run.")
    print("[3] Enter 3 to climb a mountain.")
    print("[q] Enter q to quit.")

    # Ask for the user's choice.
    choice = input("\nWhat would you like to do? ")

    # Respond to the user's choice.
    if choice == '1':
        print("\nHere's a bicycle. Have fun!\n")
    elif choice == '2':
        print("\nHere are some running shoes. Run fast!\n")
    elif choice == '3':
        print("\nHere's a map. Can you leave a trip plan for us?\n")
    elif choice == 'q':
        print("\nThanks for playing. See you later.\n")
    else:
        print("\nI don't understand that choice, please try again.\n")

# Print a message that we are all finished.
print("Thanks again, bye now.")
```

```

Welcome to the nature center. What would you like to do?

[1] Enter 1 to take a bicycle ride.
[2] Enter 2 to go for a run.
[3] Enter 3 to climb a mountain.
[q] Enter q to quit.

What would you like to do? 1
```

```
Here's a bicycle. Have fun!


[1] Enter 1 to take a bicycle ride.
[2] Enter 2 to go for a run.
[3] Enter 3 to climb a mountain.
[q] Enter q to quit.

What would you like to do? 3

Here's a map. Can you leave a trip plan for us?


[1] Enter 1 to take a bicycle ride.
[2] Enter 2 to go for a run.
[3] Enter 3 to climb a mountain.
[q] Enter q to quit.

What would you like to do? q

Thanks for playing. See you later.

Thanks again, bye now.
```

Our programs are getting rich enough now, that we could do many different things with them. Let's clean this up in one really useful way. There are three main choices here, so let's define a function for each of those items. This way, our menu code remains really simple even as we add more complicated code to the actions of riding a bicycle, going for a run, or climbing a mountain.

[34]:
```python
# Define the actions for each choice we want to offer.
def ride_bicycle():
    print("\nHere's a bicycle. Have fun!\n")

def go_running():
    print("\nHere are some running shoes. Run fast!\n")

def climb_mountain():
    print("\nHere's a map. Can you leave a trip plan for us?\n")

# Give the user some context.
print("\nWelcome to the nature center. What would you like to do?")

# Set an initial value for choice other than the value for 'quit'.
choice = ''

# Start a loop that runs until the user enters the value for 'quit'.
while choice != 'q':
    # Give all the choices in a series of print statements.
    print("\n[1] Enter 1 to take a bicycle ride.")
    print("[2] Enter 2 to go for a run.")
    print("[3] Enter 3 to climb a mountain.")
    print("[q] Enter q to quit.")

    # Ask for the user's choice.
    choice = input("\nWhat would you like to do? ")

    # Respond to the user's choice.
    if choice == '1':
        ride_bicycle()
    elif choice == '2':
        go_running()
    elif choice == '3':
        climb_mountain()
    elif choice == 'q':
        print("\nThanks for playing. See you later.\n")
    else:
        print("\nI don't understand that choice, please try again.\n")

# Print a message that we are all finished.
print("Thanks again, bye now.")
```

```
Welcome to the nature center. What would you like to do?

[1] Enter 1 to take a bicycle ride.
[2] Enter 2 to go for a run.
[3] Enter 3 to climb a mountain.
[q] Enter q to quit.

What would you like to do? 1

Here's a bicycle. Have fun!


[1] Enter 1 to take a bicycle ride.
[2] Enter 2 to go for a run.
[3] Enter 3 to climb a mountain.
[q] Enter q to quit.

What would you like to do? 3

Here's a map. Can you leave a trip plan for us?


[1] Enter 1 to take a bicycle ride.
```

```
[2] Enter 2 to go for a run.
[3] Enter 3 to climb a mountain.
[q] Enter q to quit.

What would you like to do? q

Thanks for playing. See you later.

Thanks again, bye now.
```

This is much cleaner code, and it gives us space to separate the details of taking an action from the act of choosing that action.

top

7 Using while loops to process items in a list

In the section on Lists, you saw that we can `pop()` items from a list. You can use a while list to pop items one at a time from one list, and work with them in whatever way you need.

Let's look at an example where we process a list of unconfirmed users.

[21]:
```python
# Start with a list of unconfirmed users, and an empty list of confirmed users.
unconfirmed_users = ['ada', 'billy', 'clarence', 'daria']
confirmed_users = []

# Work through the list, and confirm each user.
while len(unconfirmed_users) > 0:

    # Get the latest unconfirmed user, and process them.
    current_user = unconfirmed_users.pop()
    print("Confirming user %s...confirmed!" % current_user.title())

    # Move the current user to the list of confirmed users.
    confirmed_users.append(current_user)

# Prove that we have finished confirming all users.
print("\nUnconfirmed users:")
for user in unconfirmed_users:
    print('- ' + user.title())

print("\nConfirmed users:")
for user in confirmed_users:
    print('- ' + user.title())
```

```
Confirming user Daria…confirmed!
Confirming user Clarence…confirmed!
Confirming user Billy…confirmed!
Confirming user Ada…confirmed!
```

```
Unconfirmed users:

Confirmed users:
- Daria
- Clarence
- Billy
- Ada
```

This works, but let's make one small improvement. The current program always works with the most recently added user. If users are joining faster than we can confirm them, we will leave some users behind. If we want to work on a 'first come, first served' model, or a 'first in first out' model, we can pop the first item in the list each time.

[22]:

```python
# Start with a list of unconfirmed users, and an empty list of confirmed users.
unconfirmed_users = ['ada', 'billy', 'clarence', 'daria']
confirmed_users = []

# Work through the list, and confirm each user.
while len(unconfirmed_users) > 0:

    # Get the latest unconfirmed user, and process them.
    current_user = unconfirmed_users.pop(0)
    print("Confirming user %s...confirmed!" % current_user.title())

    # Move the current user to the list of confirmed users.
    confirmed_users.append(current_user)

# Prove that we have finished confirming all users.
print("\nUnconfirmed users:")
for user in unconfirmed_users:
    print('- ' + user.title())

print("\nConfirmed users:")
for user in confirmed_users:
    print('- ' + user.title())
```

```
Confirming user Ada…confirmed!
Confirming user Billy…confirmed!
Confirming user Clarence…confirmed!
Confirming user Daria…confirmed!

Unconfirmed users:

Confirmed users:
- Ada
- Billy
- Clarence
- Daria
```

This is a little nicer, because we are sure to get to everyone, even when our program is running under a heavy load. We also preserve the order of people as they join our project. Notice that this all came about by adding *one character* to our program!

top

8 Accidental Infinite loops

Sometimes we want a while loop to run until a defined action is completed, such as emptying out a list. Sometimes we want a loop to run for an unknown period of time, for example when we are allowing users to give as much input as they want. What we rarely want, however, is a true 'runaway' infinite loop.

Take a look at the following example. Can you pick out why this loop will never stop?

[ ]:
```python
current_number = 1

# Count up to 5, printing the number each time.
while current_number <= 5:
    print(current_number)
```

[ ]:
```
1
1
1
1
1
...
```

I faked that output, because if I ran it the output would fill up the browser. You can try to run it on your computer, as long as you know how to interrupt runaway processes:

- On most systems, Ctrl-C will interrupt the currently running program.
- If you are using Geany, your output is displayed in a popup terminal window. You can either press Ctrl-C, or you can use your pointer to close the terminal window.

The loop runs forever, because there is no way for the test condition to ever fail. The programmer probably meant to add a line that increments current_number by 1 each time through the loop:

[23]:
```python
current_number = 1

# Count up to 5, printing the number each time.
while current_number <= 5:
    print(current_number)
    current_number = current_number + 1
```

```
1
2
3
```

```
4
5
```

You will certainly make some loops run infintely at some point. When you do, just interrupt the loop and figure out the logical error you made.

Infinite loops will not be a real problem until you have users who run your programs on their machines. You won't want infinite loops then, because your users would have to shut down your program, and they would consider it buggy and unreliable. Learn to spot infinite loops, and make sure they don't pop up in your polished programs later on.

Here is one more example of an accidental infinite loop:

[ ]:
```
current_number = 1

# Count up to 5, printing the number each time.
while current_number <= 5:
    print(current_number)
    current_number = current_number - 1
```

[ ]:
```
1
0
-1
-2
-3
...
```

In this example, we accidentally started counting down. The value of `current_number` will always be less than 5, so the loop will run forever.

8.1 Exercises

Marveling at Infinity

- Use one of the examples of a failed while loop to create an infinite loop.
- Interrupt your output.
- Marvel at the fact that if you had not interrupted your output, your computer would have kept doing what you told it to until it ran out of power, or memory, or until the universe went cold around it.

[ ]:
```
# Ex 6.4 : Marveling at Infinity

# put your code here
```

top

9 Overall Challenges

Gaussian Addition This challenge is inspired by a story about the mathematician Carl Frederich Gauss. As the story goes, when young Gauss was in grade school his teacher got mad at his class one day.

"I'll keep the lot of you busy for a while", the teacher said sternly to the group. "You are to add the numbers from 1 to 100, and you are not to say a word until you are done."

The teacher expected a good period of quiet time, but a moment later our mathematician-to-be raised his hand with the answer. "It's 5050!" Gauss had realized that if you list all the numbers from 1 to 100, you can always match the first and last numbers in the list and get a common answer:

```
1, 2, 3, ..., 98, 99, 100
1 + 100 = 101
2 + 99 = 101
3 + 98 = 101
```

Gauss realized there were exactly 50 pairs of numbers in the range 1 to 100, so he did a quick calculation: 50 * 101 = 5050.

- Write a program that passes a list of numbers to a function.
  - The function should use a while loop to keep popping the first and last numbers from the list and calculate the sum of those two numbers.
  - The function should print out the current numbers that are being added, and print their partial sum.
  - The function should keep track of how many partial sums there are.
  - The function should then print out how many partial sums there were.
  - The function should perform Gauss' multiplication, and report the final answer.
- Prove that your function works, by passing in the range 1-100, and verifying that you get 5050.
  - `gauss_addition(list(range(1,101)))`
- Your function should work for any set of consecutive numbers, as long as that set has an even length.
  - Bonus: Modify your function so that it works for any set of consecutive numbers, whether that set has an even or odd length.

[1]:
```
# Overall Challenge: Gaussian Addition

# put your code here
```

top