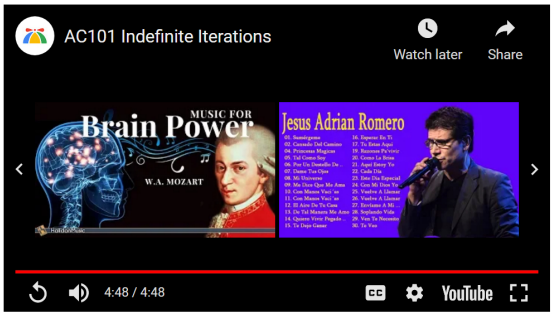
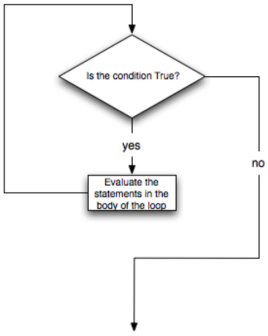
14.2. The `while` Statement

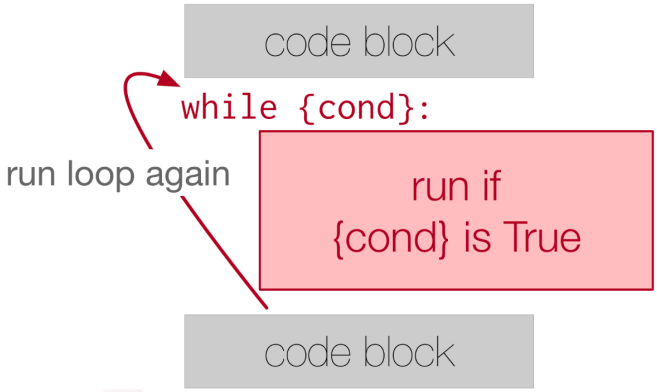


Activity: 1 -- Video: (whileloop)

There is another Python statement that can also be used to build an iteration. It is called the `while` statement. The `while` statement provides a much more general mechanism for iterating. Similar to the `if` statement, it uses a boolean expression to control the flow of execution. The body of while will be repeated as long as the controlling boolean expression evaluates to `True`.

The following two figures show the flow of control. The first focuses on the flow inside the while loop and the second shows the while loop in context.





We can use the `while` loop to create any type of iteration we wish, including anything that we have previously done with a `for` loop. For example, the program in the previous section could be rewritten using `while`. Instead of relying on the `range` function to produce the numbers for our summation, we will need to produce them ourselves. To do this, we will create a variable called `aNumber` and initialize it to 1, the first number in the summation. Every iteration will add `aNumber` to the running total until all the values have been used. In order to control the iteration, we must create a boolean expression that evaluates to `True` as long as we want to keep adding values to our running total. In this case, as long as `aNumber` is less than or equal to the bound, we should keep going.

Here is a new version of the summation program that uses a while statement.

Save & Run | Original - 1 of 1 | Show in CodeLens

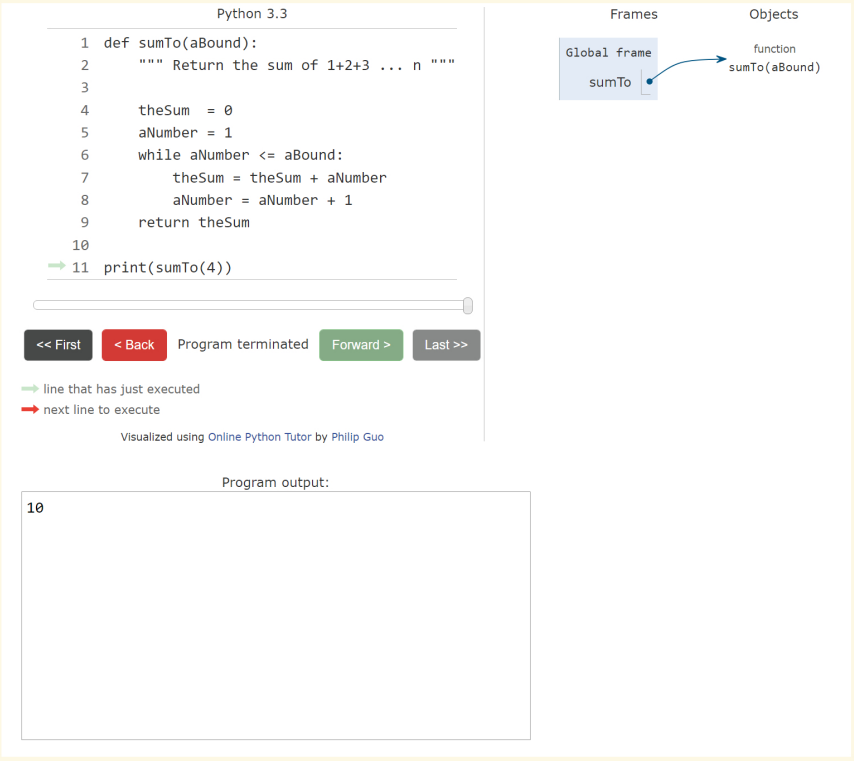
```
def sumTo(aBound):
    """ Return the sum of 1+2+3 ... n """

    theSum  = 0
    aNumber = 1
    while aNumber <= aBound:
        theSum = theSum + aNumber
        aNumber = aNumber + 1
    return theSum

print(sumTo(4))

print(sumTo(1000))

```

```
10
500500
```

Activity: 2 -- ActiveCode (ac14_2_1)

You can almost read the `while` statement as if it were in natural language. It means, while `aNumber` is less than or equal to `aBound`, continue executing the body of the loop. Within the body, each time, update `theSum` using the accumulator pattern and increment `aNumber`. After the body of the loop, we go back up to the condition of the `while` and reevaluate it. When `aNumber` becomes greater than `aBound`, the condition fails and flow of control continues to the `return` statement.

The same program in codelens will allow you to observe the flow of execution.



Activity: 3 -- CodeLens: (clens14_2_1)

Note

The names of the variables have been chosen to help readability.

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `False` or `True`.
2. If the condition is `False`, exit the `while` statement and continue execution at the next statement.
3. If the condition is `True`, execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is `False` the first time through the loop, the statements inside the loop are never executed.

The body of the loop should change the value of one or more variables so that eventually the condition becomes `False` and the loop terminates. Otherwise the loop will repeat forever. This is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions written on the back of the shampoo bottle (lather, rinse, repeat) create an infinite loop.

In the case shown above, we can prove that the loop terminates because we know that the value of `aBound` is finite, and we can see that the value of `aNumber` increments each time through the loop, so eventually it will have to exceed `aBound`. In other cases, it is not so easy to tell.

Note

Introduction of the while statement causes us to think about the types of iteration we have seen. The `for` statement will always iterate through a sequence of values like the list of names for the party or the list of numbers created by `range`. Since we know that it will iterate once for each value in the collection, it is often said that a `for` loop creates a **definite iteration** because we definitely know how many times we are going to iterate. On the other hand, the `while` statement is dependent on a condition that needs to evaluate to `False` in order for the loop to terminate. Since we do not necessarily know when this will happen, it creates what we call **indefinite iteration**. Indefinite iteration simply means that we don't know how many times we will repeat but eventually the condition controlling the iteration will fail and the iteration will stop. (Unless we have an infinite loop which is of course a problem)

What you will notice here is that the `while` loop is more work for you — the programmer — than the equivalent `for` loop. When using a `while` loop you have to control the loop variable yourself. You give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates. That also makes a while loop harder to read and understand than the equivalent for loop. So, while you *can* implement definite iteration with a while loop, it's not a good idea to do that. Use a for loop whenever it will be known at the beginning of the iteration process how many times the block of code needs to be executed.

Check your understanding

moreiter-2-1: True or False: You can rewrite any for-loop as a while-loop.

- (●) A. True
- ( ) B. False

Check me Compare me

✔ Although the while loop uses a different syntax, it is just as powerful as a for-loop and often more flexible.

Activity: 4 -- Multiple Choice (question14_2_1)

moreiter-2-2: The following code contains an infinite loop. Which is the best explanation for why the loop does not terminate?

```
n = 10
answer = 1
while ( n > 0 ):
  answer = answer + n
  n = n + 1
print(answer)
```

- (●) A. n starts at 10 and is incremented by 1 each time through the loop, so it will always be positive
- ( ) B. answer starts at 1 and is incremented by n each time, so it will always be positive
- ( ) C. You cannot compare n to 0 in while loop. You must compare it to another variable.
- ( ) D. In the while loop body, we must set n to False, and this code does not do that.

Check me Compare me

✔ The loop will run as long as n is positive. In this case, we can see that n will never become non-positive.

Activity: 5 -- Multiple Choice (question14_2_2)

moreiter-2-3: Which type of loop can be used to perform the following iteration: You choose a positive integer at random and then print the numbers from 1 up to and including the selected integer.

- (●) A. a for-loop or a while-loop
- ( ) B. only a for-loop
- ( ) C. only a while-loop

Check me Compare me

✔ Although you do not know how many iterations you loop will run before the program starts running, once you have chosen your random integer, Python knows exactly how many iterations the loop will run, so either a for-loop or a while-loop will work.

Activity: 6 -- Multiple Choice (question14_2_3)

Write a while loop that is initialized at 0 and stops at 15. If the counter is an even number, append the counter to a list called `eve_nums`.

Save & Run 5/14/2021, 3:13:14 PM - 2 of 2 Show in CodeLens

```
count = 0

eve_nums = []
while count <= 15:
    if count % 2 == 0:
        eve_nums.append(count)
    count = count + 1

```

Activity: 7 -- ActiveCode (ac14_2_2)

Result	Actual Value	Expected Value	Notes
Pass	[0, 2..., 14]	[0, 2..., 14]	Testing that eve_nums has been assigned the correct elements

Expand Differences

You passed: 100.0% of the tests

Below, we've provided a for loop that sums all the elements of `list1`. Write code that accomplishes the same task, but instead uses a while loop. Assign the accumulator variable to the name `accum`.

Save & Run 5/14/2021, 3:14:50 PM - 3 of 3 Show in CodeLens

```

list1 = [8, 3, 4, 5, 6, 7, 9]

tot = 0
for elem in list1:
```

```
    tot = tot + elem

idx = 0
accum = 0
while idx < len(list1):
    accum = accum + list1[idx]
    idx = idx + 1
```

Activity: 8 -- ActiveCode (ac14_2_3)

Result	Actual Value	Expected Value	Notes
Pass	42	42	Testing that accum has the correct value.
Pass	'while'	'\nlist... + 1\n'	Testing your code (Don't worry about actual and expected values).

Expand Differences

You passed: 100.0% of the tests

Write a function called `stop_at_four` that iterates through a list of numbers. Using a while loop, append each number to a new list until the number 4 appears. The function should return the new list.

Save & Run 5/14/2021, 3:17:22 PM - 3 of 3 Show in CodeLens

```
def stop_at_four(my_list):
    accum_lst=[]
    accum_var=0

    while (accum_var < len(my_list)) and (my_list[accum_var] != 4):
        accum_lst.append(my_list[accum_var])
        accum_var+=1
    return accum_lst

print(stop_at_four([3,6,4,1,3]))
```

```
[3, 6]
```

Activity: 9 -- ActiveCode (ac14_2_4)

Result	Actual Value	Expected Value	Notes
Pass	[0, 9...1, 7]	[0, 9...1, 7]	Testing the function stop_at_four on the input [0, 9, 4.5, 1, 7, 4, 8, 9, 3].
Pass	[]	[]	Testing the function stop_at_four on the input [4, 1, 2, 8].
Pass	[]	[]	Testing the function stop_at_four on the input [4].
Pass	[1, 6, 2, 3, 9]	[1, 6, 2, 3, 9]	Testing that stop_at_four([1, 6, 2, 3, 9]) returns ([1, 6, 2, 3, 9])

Expand Differences

You passed: 100.0% of the tests

You have attempted 10 of 9 activities on this page

14.1. Introduction">
troduction">
14.3. The Listener Loop">
✔ Completed. Well Done!
14.3. The Listener Loop">Next Section - 14.3. The Listener Loop

| Back to top