# Chapter 4: Python Loops

## Contents

Time required: 60 minutes

## DRY

**D**on't **R**epeat **Y**ourself

## Learning Outcomes

Students will be able to:

- Write different types of loops, such as for and while

- Select the correct type of loop based on a given problem

- Use loops to solve common problems, such as, finding sum, finding max, etc.

- Write nested loops

## Loops

Computers do not get bored. If you want the computer to repeat a certain task hundreds of thousands of times, it does not protest. Humans hate too much repetition. Therefore, repetitious tasks should be performed by computers. All programming languages support repetitions. The general class of programming constructs that allow the definition of repetitions are called "iterations." A term which is even more common for such tasks is "loops."

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well, and people do poorly. Because iteration is so common, Python provides several language features to make it easier.

## Control Flow

Control flow is generally considered to include the following features plus some others:

- repetition

- modularity

- decision logic

## Repetition

Repetition is the ability to cause specified portions of the code to be executed repeatedly under tightly controlled conditions. Repetition almost always embeds decision logic in some form. This module will introduce you to two of the mechanisms for implementing repetition in Python -- the for and while loop.

## Decision Logic

Decision logic is the ability to make decisions based on the program state and to determine which code will be executed as well as when and how it will be executed. Repetition almost always embeds decision logic in some form.

# Updating Variables

A common pattern is an assignment statement that updates a variable, where the new value of the variable depends on the old.

```
x = x + 1
```

This means "get the current value of x, add 1, and then update x with the new value."

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
>>> x = x + 1
NameError: name "x" is not defined
```

Before you can update a variable, you must initialize it, usually with a simple assignment:

```
>>> x = 0
```

```
>>> x = x + 1
```

Updating a variable by adding 1 is called an increment; subtracting 1 is called a decrement.

# For Loops

Sometimes we want to loop through a set of things such as a list of words, the lines in a file, or a list of numbers. When we have a list of things to loop through, we can construct a definite loop using a for statement. We call the while statement an indefinite loop because it simply loops until some condition becomes False, whereas the for loop is looping through a known set of items so it runs through as many iterations as there are items in the set.

## Examples

**Example 1:** The following will print Hello ten times:

```
for i in range(10):
    print("Hello")
```

1. When control enters the loop, the first item in the sequence is assigned to i.

2. The statements in the body of the loop are executed. Those statements may or may not refer to the contents of i, but very often will.

3. The next item in the sequence is assigned to i and the statements in the body of the loop are executed. This process continues until every item in the sequence has been processed.

4. After all of the items in the sequence have been processed, control transfers to the next statement following the for loop statement.

The structure of a for loop is as follows:

```
for variable name in range(number of times to repeat):
    statements to be repeated
```

The syntax is important. The word `for` must be in lowercase, the first line must end with a colon, and the statements to be repeated must be indented. Indentation is used to tell Python which statements will be repeated.

The flow chart below shows the program flow. The range range(10) generates 10 numbers from 0-9 and stops at 10.
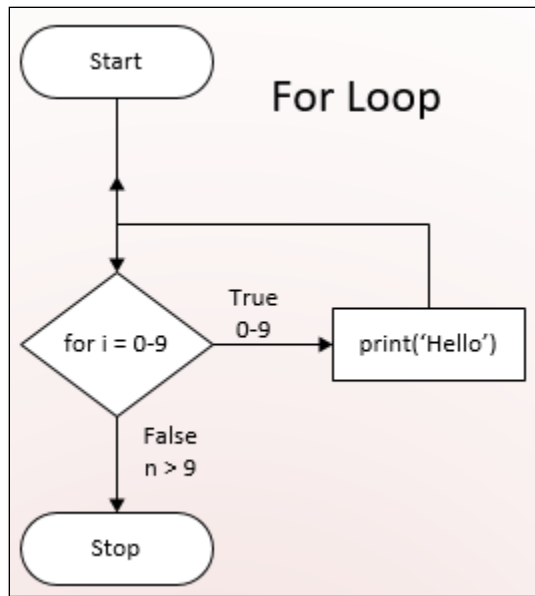
*Figure 1 For loop*

**Example 2:** The program below asks the user for a number and prints its square, then asks for another number and prints its square, etc. It does this three times and then prints that the loop is done.

```
for i in range(3):
    num = eval(input("Enter a number: "))
    print ("The square of your number is", num * num)
print("The loop is now done.")
```

Example run:

```
Enter a number: 3
The square of your number is 9
Enter a number: 5
The square of your number is 25
Enter a number: 23
The square of your number is 529
The loop is now done.
```

Since the second and third lines are indented, Python knows that these are the statements to be repeated. The fourth line is not indented, so it is not part of the loop and only gets executed once, after the loop has completed.

Looking at the above example, we see where the term for loop comes from: we can picture the execution of the code as starting at the `for` statement, proceeding to the second and third lines, then looping back up to the `for` statement.

**Example 3**: The program below will print A, then B, then it will alternate C's and D's five times and then finish with the letter E once.

```python
print("A")
print("B")
for i in range(5):
    print("C")
    print('D')
print('E')
```

1. The first two print statements get executed once, printing an A followed by a B.

2. The C's and D's alternate five times. Note that we don't get five C's followed by five D's. The way the loop works is we print a C, then a D, then loop back to the start of the loop and print a C and another D, etc.

3. When the program is done looping with the C's and D's, it prints one E.

**Example 4**: If we wanted the above program to print five C's followed by five D's, instead of alternating C's and D's, we could do the following:

```python
print("A")
print("B")
for i in range(5):
    print("C")
for i in range(5):
    print("D")
print("E")
```

**Example 5:** The next example shows a for loop applied to a list and then to a string extracted from the list. A list is a data structure.

```python
dogs = ["Affenpinscher", "Afgan Hound", "Akita"]

for breed in dogs:
    print(breed)
print(f"Length of dogs list = {len(dogs)}")

for letter in dogs[2]:
    print(letter)
print(dogs[2])
```

```
Python 3.6
(known limitations)

1  dogs = ["Affenpinscher","Afgan Hound","Akita"]
2
3  for breed in dogs:
4      print(breed)
5  print("Length of dogs list = " + str(len(dogs)))
6
7  for letter in dogs[2]:
8      print(letter)
9  print(dogs[2])
```

Edit this code

that just executed
line to execute

<< First    < Prev    Next >    Last >>

Done running (21 steps)

e visualization (NEW!)

Print output (drag lower right corner to resize)
```
k
i
t
a
Akita
```

Frames          Objects

Global frame                          str
    dogs                          "Affenpinscher"
    breed
    letter                            str
                                  "Afgan Hound"

                                      str
                                   "Akita"
                        list
                        0   1   2

                        str
                        "a"

1.  The script begins by creating and populating a list with the names of three breeds of dogs.

2.  The first for loop in Listing 1 iterates through the list extracting and printing the name of each breed on a new line. When the first for loop in Listing 1 terminates, the code gets and prints the length of the list.

3.  The second for loop gets and iterates through the string at index value 2 in the list ("Akita"). It prints each letter from that string on a new line. When that for loop terminates, the code in Listing 1 prints the contents of the string.

## Loop Variable

One part of a for loop that is a little tricky is the loop variable. In the example below, the loop variable is the variable i. The output of this program will be the numbers 0, 1, . . . , 9, each printed on its own line.

```
for i in range(10)
    print(i)
```

1.  When the loop first starts, Python sets the variable i to 0.

2.  Each time we loop back up, Python increases the value of i by 1.

3.  The program loops 10 times, each time increasing the value of i by 1, until we have looped 10 times. At this point the value of i is 9.

You may be wondering why `i` starts with 0 instead of 1. Well, there doesn't seem to be any good reason why other than that starting at 0 was useful in the early days of computing and it has stuck with us. Most things in computer programming start at 0 instead of 1. This does take some getting used to.

Since the loop variable, `i`, gets increased by 1 each time through the loop, it can be used to keep track of where we are in the looping process.

Consider the example below:

```
for i in range(3):
    print(i+1, "-- Hello")
```

Example program run:

```
1 – Hello
2 -- Hello
3 – Hello
```

**Names:** There's nothing too special about the name i for our variable. Programs will have the exact same result regardless of the variable name used.

```
for i in range(100):        for wacky_name in range(100):
    print(i)                    print(wacky_name)
```

It's a convention in programming to use the letters x, y, i, j, and k for loop variables, unless there's a good reason to give the variable a more descriptive name.

You can also use a name for the target variable if the variable is part of the calculation in the loop. Notice the use of the `\t` escape sequence to insert a tab character to align the columns nicely. The f-strings formatting is used to convert the number variables to strings.

Here is an example.

## Tutorial 4.1 – Numbers and Squares

```python
1  """
2      Name: numbers_and_squares.py
3      Author:
4      Created:
5      Purpose:  Use a loop to print the squares of the numbers 1 to 10
6  """
7
8  # Constant for number of dashes printed
9  NUMBER_OF_DASHES = 14
10
11 # Print the heading
12 print("Number\tSquare")
13
14 # Print dashes using string multiplication
15 print(NUMBER_OF_DASHES * "-")
16
17 # Print the numbers 1-10 and their squares
18 for number in range(1, 11):
19
20     # Calculate the square of the number
21     square = number ** 2
22
23     # Print result using \t to tab
24     print(f"{number}\t{square}")
```

Example run:

```
Number  Square
--------------
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
```

## The Range Function

For loops in many other programming languages deal strictly with numeric sequences. The built-in range function, when used with a Python for loop makes it possible to emulate that behavior.

The range function returns an immutable sequence containing a series of increasing integer values. There are two overloaded versions of the range function:

range(stop)

range(start, stop, step)

The version with a single parameter returns a sequence containing the integers beginning with 0 and ending at one less than the value of stop. For example range(6) returns the following sequence:

[0, 1, 2, 3, 4, 5]

The version of the range function with two and optionally three parameters returns a sequence that begins with start, ends just short of stop , and optionally increments by step. For example, range(1, 6, 2) returns the following sequence:

[1, 3, 5]

The sequence returned by range can be used to cause a for loop to iterate based on a numeric index in much the same way that the primary for loop in C, C++, C#, and Java behaves.

The value we put in the range function determines how many times we will loop. The way range works is it produces a list of numbers from zero to the value minus one. For instance, `range(5)` produces five values: 0, 1, 2, 3, and 4.

If we want the list of values to start at a value other than 0, we can do that by specifying the starting value. The statement `range(1, 5)` will produce the list 1, 2, 3, 4. This brings up one quirk of the range function—it stops one short of where we think it should. If we wanted the list to contain the numbers 1 through 5 (including 5), then we would have to do `range(1, 6)`.

We can also get the list of values to go up by more than one at a time. To do this, we can specify an optional step as the third argument. The statement `range(1, 10, 2)` will step through the list by twos, producing 1, 3, 5, 7, 9.

To get the list of values to go backwards, we can use a step of -1.
For instance, `range(5, 1, -1)` will produce the values 5, 4, 3, 2, in that order. (Note that the range function stops one short of the ending value 1).

Here are a few more examples:

| Statement | Values Generated |
|---|---|
| range(10) | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| range(1, 10) | 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| range(3, 7) | 3, 4, 5, 6 |

| range(2, 15, 3) | 2, 5, 8, 11, 14 |
|---|---|
| range(9, 2, -1) | 9, 8, 7, 6, 5, 4, 3 |

Here is an example program that counts down from 5 and then prints a message. We imported the sleep function from the time module, to count down 1 second at a time.

```python
# Import the sleep function from the time module
from time import sleep

for i in range(5, 0, -1):
    # Print the counter variable
    print(i, end=" ")
    # Stop for 1 second
    sleep(1)
print("Blast off!!")
```

Example program run:

```
5 4 3 2 1 Blast off!!
```

The print argument, end='' keeps everything on the same line.

```python
# Illustrates the for loop and the range function
#------------------------------------------------

sum = 0
for counter in range(5):
    sum += counter
    print(f"counter = {counter} sum = {sum}")
print(22 * "=")
#-------------------------------------------------------------
# A string is a type of list
breed = "Affenpinscher"
for index in range(1, len(breed) ,2):
    print(f"Letter at index {index} {breed[index]}")
print(breed)
```

The code in the top half of the previous program calls the range function to get an immutable sequence containing the integers from 0 through 4 inclusive. The for loop iterates through that sequence, extracting, summing, and printing the integer values and the sum of the values contained in the sequence.

Note that the body of this for loop contains two statements at the same indentation level.

The code in the bottom half of Listing 2 calls the range function to get an immutable sequence containing the following integers:

[1, 3, 5, 7, 9, 11]

The for loop iterates through that sequence and uses the integers contained in the sequence to extract and print corresponding letters from the string "Affenpinscher".

## Running Totals

There are times when you want to accumulate a running sum. You might be calculating the total sales for the month, the total of a bank account at the end of the week, or the total of a restaurant bill.

You would declare an accumulator variable that adds the value being read or calculated to itself.

```
total = total + item_price
```

The flowchart below shows the program flow to accumulating a running total.
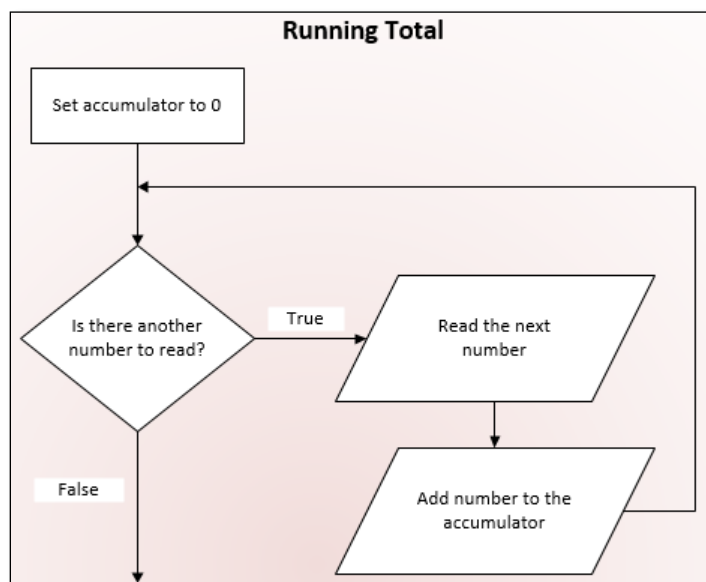


# Tutorial 4.2 - Running Total

The following program demonstrates how to keep a running total of numbers entered by the user. Save the program as **running_total.py**.

```
1  """
2      Name: running_total.py
3      Author:
4      Created:
5      Sum a series of numbers entered by the user
6  """
7
8  # Constant for the maximum number of user entries
9  MAX = 5
10
11 # Initialize an accumulator variable
12 running_total = 0
13
14 # Explain to the user what the program does
15 print(f"Let's calculate the sum of {MAX} whole numbers you will enter.")
16
17 # Get the numbers and accumulate them into a total
18 for x in range(MAX):
19     # Get the number from the user
20     number = int(input("Enter a number: "))
21
22     # Add number input to the total
23     running_total = running_total + number
24
25 # Display the running total of the numbers
26 print(f"The total is {running_total}.")
```

Example run:

```
Let's calculate the sum of 5 whole numbers you will enter.
Enter a number: 6
Enter a number: 8
Enter a number: 3
Enter a number: 90
Enter a number: 34
The total is 141.
```

## Tutorial 4.3 – And Now for Something Completely Different

You can use this web site to put your name in ascii art into your Python program.

https://www.patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%20

1. Copy the ascii characters into a Python file.

2. Some of the fonts don't work with particular words. The example shown is with the Doom font.

```python
 2  p = """
 3   _____          _   _
 4  |  __  \        | | | |
 5  | |__/ /        | | | |       __       __
 6  |  ___/ | | | __|  |_ |_  __  \  /  \ |   \
 7  | |     | | | |  _  | | | | |  | | |  | | | |
 8  \_|      \_, |\___| |_|\__/|_| |_|
 9            __/ |
10           |___/
11  """
12  print(p)
```

Example run:

```
 _____          _   _
|  __  \        | | | |
| |__/ /        | | | |       __       __
|  ___/ | | | __|  |_ |_  __  \  /  \ |   \
| |     | | | |  _  | | | | |  | | |  | | | |
\_|      \_, |\___| |_|\__/|_| |_|
          __/ |
         |___/
```

## While Loops

Sometimes, we need to repeat something, but we don't know ahead of time exactly how many times it must be repeated. For instance, a game of Tic-tac-toe keeps going until someone wins or there are no more moves to be made. The number of turns will vary from game to game. This is a situation that would call for a while loop.

A while statement is like an if statement. The syntax is:

```
while boolean expression == True:
    statements
```

Just like an if statement, the while statement tests a Boolean expression. If the expression evaluates to True, it executes the code block below it. However, contrary to the if statement, once the code block has finished, the code "loops" back to the Boolean expression to test it again. If it still evaluates to True, the code block below it gets executed once more. And after it has finished, it loops back again, and again, and again...

**Note:** If the boolean expression immediately evaluates to False, then the code block below the while is skipped completely, just like with an if statement.

Let's take a simple example: printing the numbers 1 to 5. With a while loop, that can be done as follows:

```
num = 1
while num <= 5:
    print(num)
    num += 1
print("Done" )
```

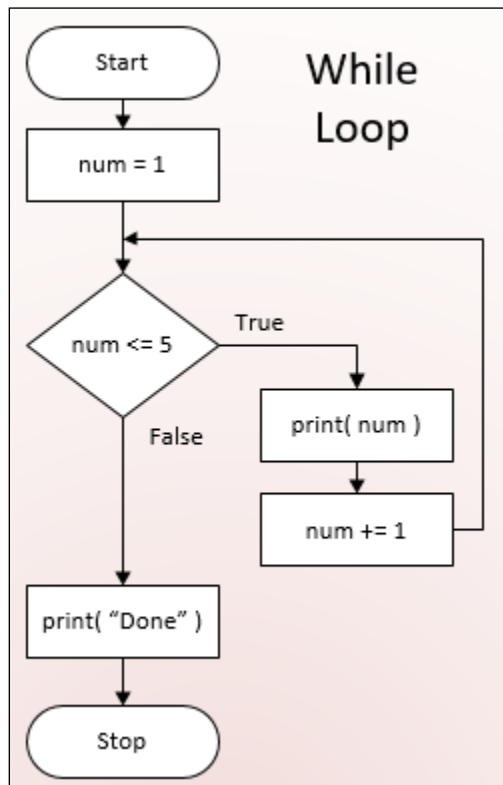This code is also expressed by the following flow chart.



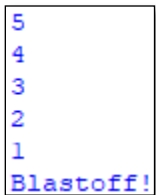*Figure 2 Flow chart expressing a while loop.*

1. The first line initializes a variable num. This is the variable that the code will print, so it is initialized to 1, as 1 is the first value that must be printed.

2. The while loop starts. The boolean expression says num <= 5. Since num is 1, and 1 is smaller than (or equal to) 5, the boolean expression evaluates to True. Therefore, the code block below the while gets executed.

3. The first line of the code block below the while prints the value of num, which is 1.

4. The second line adds 1 to the value of num, which makes num hold the value 2.

5. The code loops back to the boolean expression

6. num is 2, the boolean expression still evaluates to True. The code block gets executed once more. 2 is displayed, num gets the value 3, and the code loops back to the Boolean expression.

7. num is 3, the boolean expression still evaluates to True. The code block gets executed once more. 3 is displayed, num gets the value 4, and the code loops back to the Boolean expression.

8. num is 4, the Boolean expression still evaluates to True. The code block gets executed once more. 4 is displayed, num gets the value 5, and the code loops back to the Boolean expression.

9. When num reaches 5, the Boolean expression evaluates to False. The loop is exited.

10. The last line of the code, the printing of "Done," is executed.

Here is another program that uses a while loop to count down from five and then says "Blastoff!".

```
from time import sleep
n = 5
while n > 0:
    print(n)
    n = n - 1
    sleep(1)
print("Blastoff!")
```

Example program run:

```
5
4
3
2
1
Blastoff!
```

You can almost read the while statement as if it were English. It means, "While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, exit the while statement and display the word Blastoff!"

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

1. Evaluate the condition, yielding True or False.

2. If the condition is false, exit the while statement and continue execution at the next statement.

3. If the condition is true, execute the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top. We call each time we execute the body of the loop an iteration. For the above loop, we would say, "It had five iterations", which means that the body of the loop was executed five times.

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. We call the variable that changes each time the loop executes and controls when the loop finishes the iteration variable. If there is no iteration variable, the loop will repeat forever, resulting in an infinite loop.

## Augmented Assignment Operators

When a program accumulates something, you might have something like this.

```
x = x + 2
```

The number 2 is added to the variable x on the right and assigned to the variable x on the left. Our previous program showed an example of this.

```
total = total + number
```

Here are the augmented assignment operators with some examples.

| Operator | Example Usage | Equivalent To |
|----------|---------------|---------------|
| += | x += 10 | x = x + 10 |
| -= | y -= 3 | y = y - 3 |
| *= | z *= 14 | z = z * 4 |
| /= | a /= b | a = a / b |
| %= | c %= 3 | c = c % 3 |
| //= | x //= 3 | x = x // 3 |
| **= | y **= 2 | y = y**2 |

Here is another way to do the running total program with a while loop using augmented assignment operators.

```
# Declare running total variable
total = 0
# Declare loop counter
count = 0
# Loop from 0 - 4
while count < 5:
    # Get input, add to the running total
    total += int(input("Enter a number: "))
    # Increment the loop counter
    count += 1
# Display the running total
print(f"Total is {total}")
```

Example program run:

```
Enter a number: 1
Enter a number: 52
Enter a number: 3
Enter a number: 47
Enter a number: 5
Total is 108
```

There are two variables used. `total` is used to add up the five numbers that the user enters. It is started at zero, as at the start of the code the user has not yet entered any numbers, so the total is still zero.

**count** is used to count how often the code has gone through the loop. Since the loop must be done five times, count is started at 0 and the Boolean expression tests if count smaller is than 5. Since in the loop count gets increased by 1 at the end of every cycle through the loop, the loop gets processed five times before the Boolean expression changes to False.

You may wonder why count is started at 0 and the Boolean expression checks if count < 5. Why not start `count` at 1 and check if `count <= 5`? The reason is convention: programmers are used to start indices at 0, and if they count, they count "up to but not including." When you continue with programming, you will find that most code sticks to this convention. Most standard programming constructs that use indices or count things apply this convention.

Note: The variable **count** is what programmers call a "throw-away variable." Its only purpose is to count how often the loop has been cycled through, and it has no real meaning before the loop, in the loop, or after the loop has ended. Programmers often choose a single character variable name for such a variable, usually x. y is typically used if a second counter is needed.

## Sentinel Value

What if you want a loop that you want to keep going until the user quits? That is when we want to use a sentinel value: a piece of data that would not make sense in the regular sequence, and which is used to indicate the end of the input.

```
while answer != -1:
    The loop keeps going until the user enters -1
```

Example program:

```
answer = 0
while answer != -1:
    answer = float(input("Enter a positive number, -1 to quit.\n"))
    if(answer != -1):
        print(f"Your number is: {answer}")
print("That's all folks!")
```

Example program run:

```
Enter a positive number, -1 to quit.
45
Your number is: 45.0
Enter a positive number, -1 to quit.
-1
That's all folks!
```

# Tutorial 4.4 – Running Total under User Control

The code block below shows how to use a while loop to allow the user to enter numbers as long as they want, until they enter a zero. Once a zero is entered, the total is printed, and the program ends. This is called a sentinel value.

```
 1  # Declare running total variable
 2  total = 0
 3
 4  # Get input from the user before the loop
 5  # This allows the input before the while loop test
 6  num = int(input( "Enter a number: " ))
 7
 8  # Keep looping until the user enters 0
 9  while num != 0:
10
11      # Add up the running total
12      total += num
13
14      # Get another number from the user
15      num = int(input( "Enter a number: " ))
16
17  # Display the running total
18  print( "Total is", total )
```

Example run:

```
Enter a number: 5
Enter a number: 2
Enter a number: 0
Total is 7
```

## Input Validation and Exception Handling

Input validation code checks that values entered by the user, such as text from the input() function, are formatted correctly.

For example, if you want users to enter their ages, your code shouldn't accept nonsensical answers such as negative numbers (which are outside the range of acceptable integers) or words (which are the wrong data type). Input validation can also prevent bugs or security vulnerabilities.

If you implement a withdraw_from_account() function that takes an argument for the amount to subtract from an account, you need to ensure the amount is a positive number. If the withdraw_from_account() function subtracts a negative number from the account, the "withdrawal" will end up adding money!

Typically, we perform input validation by repeatedly asking the user for input until they enter valid text, as in the following example:

## Tutorial 4.5 – Input Validation Try Except

```python
"""
    File name: input_validation.py
    Author:
    Created:
    Purpose: Validate the correct input
"""

# Loop until both conditions return False
while True:
    # Is the input an integer?
    try:
        age = int(input("Enter your age:"))
    except:
        print("Please use whole numbers.")
        continue
    # Is the input a positive integer?
    if age < 1:
        print("Please enter a positive number.")
        continue
    break

print(f"Your age is {age}.")
```

Sample run:

```
Enter your age:d
Please use numeric digits.
Enter your age:-2
Please enter a positive number.
Enter your age:23
Your age is 23.
```

When you run this code, you'll be prompted for your age until you enter a valid one. This ensures that by the time the execution leaves the while loop, the age variable will contain a valid value that won't crash the program.

## Infinite Loops

An endless source of amusement for programmers is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop because there is no iteration variable telling you how many times to execute the loop.

In the case of countdown, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of n gets smaller each time through the loop, so eventually we get to 0. Other times a loop is obviously infinite because it has no iteration variable at all.

Sometimes you don't know it's time to end a loop until you get halfway through the body. In that case you can write an infinite loop on purpose and then use the break statement to jump out of the loop.

This loop is obviously an infinite loop because the logical expression on the while statement is simply the logical constant True:

```
n = 10
while True:
    print(n, end=" ")
    n = n - 1
print("Done! ")
```

If you make the mistake and run this code, you will learn quickly how to stop a runaway Python process on your system or find where the power-off button is on your computer. This program will run forever or until your battery runs out because the logical expression at the top of the loop is always true because the expression is the constant value `True`.

While this is a dysfunctional infinite loop, we can still use this pattern to build useful loops if we carefully add code to the body of the loop to explicitly exit the loop using break when we have reached the exit condition. For example, suppose you want to take input from the user until they type done.

You could write:

```
while True:
    line = input("> ")
    if line == "done":
        break
    print(line)
print("Done! ")
```

The loop condition is `True`, which is always true, so the loop runs repeatedly until it hits the break statement.

Each time through, it prompts the user with an angle bracket. If the user types **done**, the break statement exits the loop. Otherwise, the program echoes whatever the user types and goes back to the top of the loop.

Here's a sample run:

```
> hello there
hello there
> finished
finished
> done
Done!
```

This way of writing while loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

## Finishing Iterations with Continue

Sometimes you are in an iteration of a loop and want to finish the current iteration and immediately jump to the next iteration. In that case you can use the continue statement to skip to the next iteration without finishing the body of the loop for the current iteration.

Here is an example of a loop that copies its input until the user types "done" but treats lines that start with the hash character as lines not to be printed (kind of like Python comments).

```
while True:
    line = input(">")
    if line[0] == "#":
        continue
    if line == "done":
        break
print(line)
print("Done!")
```

Here is a sample run of this new program with continue added.

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

All the lines are printed except the one that starts with the hash sign because when the continue is executed, it ends the current iteration and jumps back to the while statement to start the next iteration, thus skipping the print statement.

# Tutorial 4.6 - Temperature Converter Looped

Let's look at the temperature converter we wrote earlier. Open the program and save it as **temperature_while.py**

1. The display concatenates a **°F** and a **°C** as shown in the example.

2. Get the Degree Symbol: www.degreesymbol.net
   At the top of this web page, you can **Copy** the degree symbol and paste it into your code.

One annoying thing about it is that the user must restart the program for every new temperature. A while loop will allow the user to repeatedly enter temperatures. A simple way for the user to indicate that they are done is to have them enter a nonsense temperature like -1000 (which is below absolute 0).

Enter and test the code below.

```python
"""
    Name: temperature_while.py
    Author:
    Created:
    Purpose: Convert Fahrenheit to Celsius
"""

# Sentinel value
QUIT = 9999

# Prime the loop, get the Fahrenheit temperature
fahrenheit = float(input(f"Enter a Fahrenheit temperature ({QUIT} to quit): "))

# Loop while fahrenheit isn't = to 9999
while fahrenheit != QUIT:

    # Calculate the Celsius equivalent
    celsius = (fahrenheit - 32.0) * 5.0 / 9.0

    # Display the convert temperature
    print(f"\n{fahrenheit}°F is equal to {celsius:.2f}°C\n")

    # Get the Fahrenheit temperature
    fahrenheit = float(
        input(f"Enter a Fahrenheit temperature ({QUIT} to quit): "))
```

Example run:

```
Enter a Fahrenheit temperature (9999 to quit): -40

-40.0°F is equal to -40.00°C

Enter a Fahrenheit temperature (9999 to quit): 32

32.0°F is equal to 0.00°C

Enter a Fahrenheit temperature (9999 to quit): 9999
```

1. Look at the while statement. It says that we will keep looping, that is, keep getting and converting temperatures, as long as the temperature entered is not 9999. As soon as 9999 is entered, the while loop stops.

2. Tracing through, the program first compares the temperature to 9999. If temperature is not 9999, then the program converts the temperature.

3. The program then loops back up and again compares temperature to 9999. If temperature is not 9999, the program will ask for another temperature, convert it, and then loop back up again and do another comparison. It continues this process until the user enters 9999.

4. We need the line `fahrenheit = 0.0` at the start, as without it, we would get a name error. The program would get to the while statement, try to see if fahrenheit is not equal to -1000 and run into a problem because fahrenheit doesn't yet exist. To take care of this, we just declare it equal to 0.0. There is nothing special about the value 0.1 here. We could set it to anything except 9999. (Setting it to -9999 would cause the condition on the while loop to be false right from the start and the loop would never run.)

It is normal to think of the while loop as continuing looping until the user enters 9999. When we construct the condition, instead of thinking about when to stop looping, we instead need to think in terms of what has to be true in order to keep going.

A while loop is a lot like an if statement. The difference is that the indented statements in an if block will only be executed once, whereas the indented statements in a while loop are repeatedly executed.

## Tutorial 4-7 – Guessing Game Revisited

Earlier we wrote a program that played a simple random number guessing game. The problem with that program is that the player only gets one guess. We can, in a sense, replace the if statement in that program with a while loop to create a program that allows the user to keep guessing until they get it right.

```
1  """
2      Name: guessing_game.py
3      Author:
4      Created:
5      Purpose: A better guessing game
6  """
7
8  # Import randint to generate random integers
9  from random import randint
10
11 # Generate a random number from 1-10 inclusive
12 secret_number = randint(1, 10)
13
14 # Prime the loop with a user entry
15 guess = int(input("Guess the secret number: "))
16
17 # While the guess is incorrect, continue the loop
18 while guess != secret_number:
19
20     # Get input from the user
21     guess = int(input("Guess the secret number: "))
22
23 # The user wins
24 print("You finally got it!")
```

The condition `guess != secret_number` says that as long as the current guess is not correct, we will keep looping. In this case, the loop consists of one statement, the input statement, and so the program will keep asking the user for a guess until their guess is correct.

We require the line `guess = 0` prior to the while loop so that the first time the program reaches the loop, there is something in guess for the program to use in the comparison. The exact value of guess doesn't really matter at this point. We just want something that is guaranteed to be different than `secret_number`.

When the user finally guesses the right answer, the loop ends and program control moves to the print statement after the loop, which prints a congratulatory message to the player.

This version is a bit frustrating as the user has no idea which direction to guess. They just have to keep guessing until they win.

## The Break Statement

The break statement can be used to break out of a for or while loop before the loop is finished.

## Example 1

Here is a program that allows the user to enter up to 10 numbers. The user can stop early by entering a negative number.

```python
# Loop 10 ten times, 0-9
for i in range(10):
    # Get input from user
    num = int(input("Enter number: "))
    # If num < 0, exit the loop
    if num < 0:
        break
```

This could also be accomplished with a while loop.

```python
# Initialize variables
i = 0
num = 1
# Loop 10 times unless num is a negative number
while i < 10 and num > 0:
    num = int(input("Enter a number: "))
```

Either method is ok. In many cases the break statement can help make your code easier to understand and less clumsy.

## Example 2

Earlier, we used a while loop to allow the user to repeatedly enter temperatures to be converted. Here is, more or less, the original version on the left compared with a different approach using the break statement.

```python
temp = 0                          while True:
while temp!=-1000:                    temp = eval(input(': '))
    temp = eval(input(': '))          if temp==-1000:
    if temp!=-1000:                       print('Bye')
        print(9/5*temp+32)                break
    else:                             print(9/5*temp+32)
        print('Bye!')
```

## The else Statement

There is an optional else that you can use with break statements. The code indented under the else gets executed only if the loop completes without a break happening.

## Example 1

This is a simple example based on Example 1 of the previous section.

```
for i in range(10):
    num = eval(input("Enter number: "))
    if num < 0:
        print("Stopped early")
        break
else:
    print("User entered all ten values")
```

The program allows the user to enter up to 10 numbers. If they enter a negative, then the program prints **Stopped early** and asks for no more numbers. If the user enters no negatives, then the program prints **User entered all ten values**.

**Example 2**

Here are two ways to check if an integer num is prime. A prime number is a number whose only divisors are 1 and itself. The first approach uses a while loop, while the second approach a for/break loop:

```
I = 2
while I < num and num % i! = 0:
    i = i + 1
if i == num:
    print("Prime")
else:
    print("Not prime")
```

```
for i in range(2, num):
    if num % I == 0:
        print("Not prime")
        break
else:
    print("Prime")
```

The idea behind both approaches is to scan through all the integers between 2 and num-1, and if any of them is a divisor, then we know num is not prime. To see if a value i is a divisor of num, we check to see if **num % i** is 0.

The idea of the while loop version is that we continue looping as long as we haven't found a divisor. If we get all the way through the loop without finding a divisor, then i will equal num, and in that case the number must be prime.

The idea of the for/break version is we loop through all the potential divisors, and as soon as we find one, we know the number is not prime and we print Not prime and stop looping. If we get all the way through the loop without breaking, then we have not found a divisor. In that case the **else** block will execute and print that the number is prime.

## Tutorial 4.7 - Yet Another Guessing Game - While Running = True

We revisit our old friend from many programming exercises, the Guessing Game.

The while statement allows you to repeatedly execute a block of statements as long as a condition is true. A while statement is an example of what is called a looping statement. A while statement can have an optional else clause.

Save as **guessing_while_true.py**

```python
1  """
2      Name: guessing_game_while_true.py
3      Author:
4      Created:
5      Purpose: Keep guessing the number until running = false
6  """
7
8  # Import randint to generate random integers
9  from random import randint
10
11 # Generate a random number from 1-10 inclusive
12 secret_number = randint(1, 10)
13
14 # Set boolean variable running as True
15 running = True
16
17 # While running = True, continue to loop
18 while running:
19
20     # Get input from user
21     guess = int(input("Guess the secret number: "))
22
23     # Determine if the user won, give hints if the guess is wrong
24     if guess == secret_number:
25         print(f"The secret number was {secret_number}, you guessed it!")
26
27         # This causes the while loop to stop
28         running = False
29
30     # Guess is low
31     elif guess < secret_number:
32         print("No, it is a little higher than that.")
33
34     # Guess is high
35     else:
36         print("No, it is a little lower than that.")
37
38 # Part of the while loop when the loop exits
39 else:
40     print("The while loop is over.")
41     # Do anything else you want to do here
42
43 print("Done")
```

Example program run:

```
Guess the secret number: 5
No, it is a little higher than that.
Guess the secret number: 7
No, it is a little higher than that.
Guess the secret number: 8
No, it is a little higher than that.
Guess the secret number: 9
The secret number was 9, you guessed it!
The while loop is over.
Done
```

**How It Works**

We are still playing the guessing game, this time with the while loop.

1. We move the input and if statements to inside the while loop and set the variable running to True before the while loop.

2. We check if the variable running is True and then proceed to execute the corresponding while-block.

3. After this block is executed, the condition is again checked which in this case is the running variable.

4. If it is true, we execute the while block again, else we continue to execute the optional else-block and then continue to the next statement.

5. The else block is executed when the while loop condition becomes False - this may even be the first time that the condition is checked. If there is an else clause for a while loop, it is always executed unless you break out of the loop with a break statement.

`running` is a Boolean variable. The True and False values are called Boolean types and you can consider them to be equivalent to the value 1 and 0 respectively.

## Tutorial 4.8 - Guessing Game Finale

It is worth going through step-by-step how to develop a program. We will modify the guessing game program from earlier to do the following:

- The player only gets five turns.

- The program tells the player after each guess if the number is higher or lower.

- The program prints appropriate messages for when the player wins and losses.

```
1  """
2      Name: guessing_game_finale.py
3      Author:
4      Created:
5      Purpose: The Guessing Game Finale!
6  """
7
8  # Import randint to generate random integers
9  from random import randint
10
11 # Create constants and variables
12 MAX_GUESSES = 5       # Maximum user guesses
13 guess = 0             # Users guess
14 number_guesses = 0    # How many guesses the user has used
15 guesses_left = 0      # How many guesses the user has left
16
17 # Generate a random number from 1-10 inclusive
18 secret_number = randint(1, 10)
19
20 # While the guess is incorrect and the user has guesses left
21 # continue the loop
22 while guess != secret_number and number_guesses < MAX_GUESSES:
23
24     # Get a guess from the user
25     guess = int(input("Guess the secret number: "))
26
27     # Increment numberGuesses
28     number_guesses += 1
29
30     # Determine if the user won, give hints if the guess is wrong
31     # Guess is too low
32     if guess < secret_number:
33
34         # Calculate how many guesses are left
35         guesses_left = MAX_GUESSES - number_guesses
36
37         # Provide user feedback
38         print(f"HIGHER: {guesses_left} guesses left.\n")
39
40     # Guess is too high
41     elif guess > secret_number:
42
43         # Calculate how many guesses are left
44         guesses_left = MAX_GUESSES - number_guesses
45
46         # Provide user feedback
47         print(f"LOWER: {guesses_left} guesses left.\n")
48
49     # The user wins!
50     else:
51         print(f"You got it! You are a winner!")
52
53 # If the loop ends before the user guesses the right number
54 if number_guesses == MAX_GUESSES and guess != secret_number:
55     print(f"You lose. The correct number is {secret_number}")
56
```

Below is an example run.

```
Guess the secret number: 5
LOWER. 4 guesses left.

Guess the secret number: 1
HIGHER. 3 guesses left.

Guess the secret number: 3
HIGHER. 2 guesses left.

Guess the secret number: 2
HIGHER. 1 guesses left.

Guess the secret number: 4
You got it! You are a winner!
```

## Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more places for bugs to hide.

One way to cut your debugging time is "debugging by bisection." For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a print statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, the problem must be in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you must search. After six steps (which is much less than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the "middle of the program" is and not always possible to check it. It doesn't make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

## Glossary

**accumulator** A variable used in a loop to add up or accumulate a result.

**counter** A variable used in a loop to count the number of times something happened. We initialize a counter to zero and then increment the counter each time we want to "count" something.

**decrement** An update that decreases the value of a variable.

**initialize** An assignment that gives an initial value to a variable that will be updated.

**increment** An update that increases the value of a variable (often by one).

**infinite loop** A loop in which the terminating condition is never satisfied or for which there is no terminating condition.

**iteration** Repeated execution of a set of statements using either a function that calls itself or a loop.

## Assignment Submission

Attach all program files to the assignment in BlackBoard.