PYTHON BOOT CAMP

Module 5: Loops

CS Jokes


VIA 9GAG.COM
BEING A PROGRAMMER
My mom said:
"Honey, please go to the market and buy 1 bottle of milk. If they have eggs, bring 6"
I came back with 6 bottles of milk.
She said: "Why the hell did you buy 6 bottles of milk?"
I said: "BECAUSE THEY HAD EGGS!!!!"

Objectives

- To write programs for executing statements repeatedly by using a **while** loop (§5.2).
- To develop loops following the loop design strategy (§§5.2.1-5.2.3).
- To control a loop with the user's confirmation (§5.2.4).
- To control a loop with a sentinel value (§5.2.5).
- To obtain a large amount of input from a file by using input redirection instead of typing from the keyboard (§5.2.6).
- To use **for** loops to implement counter-controlled loops (§5.3).
- To write nested loops (§5.4).
- To learn the techniques for minimizing numerical errors (§5.5).
- To learn loops from a variety of examples (**GCD**, **FutureTuition**, **MonteCarloSimulation**, **PrimeNumber**) (§§5.6, 5.8).
- To implement program control with **break** and **continue** (§5.7).
- To use a loop to control and simulate a random walk (§5.9).

Motivation

- What if you wanted to print the same sentence 100 times. How would you do that?
  - Example:
    - Print "Programming is fun!" 100 times
  - Would you really type the following 100 times???

100 times

```
print("Programming is fun!")
print("Programming is fun!")
...
print("Programming is fun!")
```

Motivation

- Loops
  - Python provides a powerful programming construct called a loop
  - Loops control how many times, in succession, an operation is performed
  - Example loop:

```
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

    - We'll explain this code shortly
    - For now, just showing that we can, in fact, printing 100 lines without having to type 100 individual statements!

Motivation

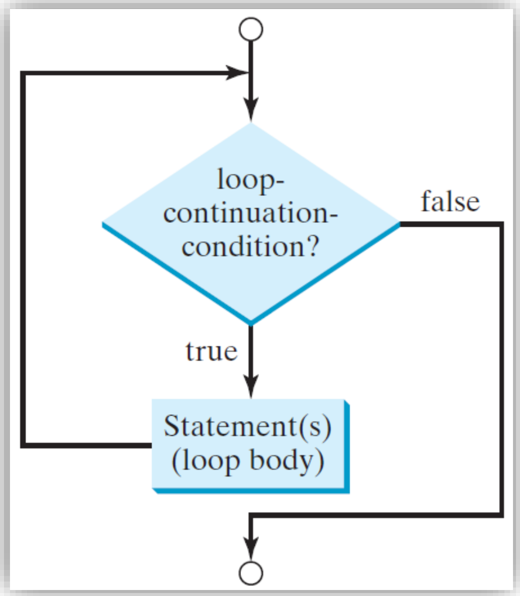
- Loops
  - Python provides two types of loop statements
  - `while` loops and `for` loops
  - `while` Loops:
    - while loops are condition-controlled loops
    - They are controlled by a true/false condition
    - Executes a statement (or statements) repeatedly so long as the given condition is true
  - `for` Loops:
    - for loops are count controlled loops that repeat a specific number of times

The `while` Loop

- Python while loop:
  - Syntax:

```
while loop-continuation-condition:
    # Loop body
    Statement(s)
```

  - Consider the flowchart on the right:
    - A single execution of the loop body is called an iteration
    - Each loop contains a loop-continuation condition
      - This controls if we execute the loop body
      - If True, the loop body is executed
      - If False, the entire loop terminates, and program control goes to the statement that follows the loop



The `while` Loop

- Using a while loop to print 100 times!

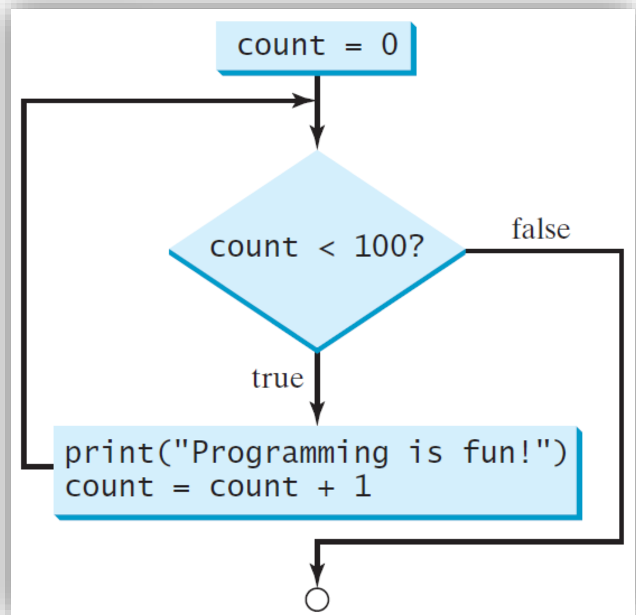
```
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

loop-continuation-condition

loop body

- Note:
  - The variable `count` is initially zero
  - The loop-continuation condition checks if `count` is less than 100
  - If `True`, it prints the message and then increments `count` by 1

    `count = count + 1`
  - At some point, it will be `False` and the loop will exit



The `while` Loop

- Another example
  - Suppose we want to sum the first 10 integers
    - 1 + 2 + 3 + … + 9 + 10
  - We can use a while loop for this!
  - Algorithm:
    - We need to loop 10 times
    - So let’s keep a CONSTANT called NUM_TIMES
    - We also keep a variable called “sum” and initialize it to 0
    - Let’s also use a “count” variable that starts at 1
      - And we will increment this variable EACH time we iterate through the loop
    - At each iteration, we take the “count” variable and add it to the “sum” variable

The `while` Loop

- Another example
  - Suppose we want to sum the first 9 integers
    - 1 + 2 + 3 + … + 9
  - We can use a while loop for this!
  - Consider the following code:

Go run this in Thonny on the Debugger!

```
sum = 0
i = 1
while i < 10:
    sum = sum + i
    i = i + 1
print("sum is", sum) # sum is 45
```

- `i` is initialized to 1
  - but is then incremented to 2, 3, 4, and so on, up to 10
- If `i < 10` is `True`, the program adds `i` to `sum`
- When `i` is 10, `i < 10` becomes false, and the loop exits

The `while` Loop

- Another example
  - What would be wrong with the following code?

```
sum = 0
i = 1
while i < 10:
    sum = sum + i
i = i + 1
```

  - Answer:
    - The loop would never exit!
    - In fact, this is called an infinite loop
    - Since the increment statement is outside the loop, `i` never increases beyond 1
      - So `i < 10` always evaluates to True

The `while` Loop

- Caution: off-by-one error
  - New programmers often execute a loop one time more (or less) than was intended
  - Consider the following code:

```
count = 0
while count <= 100:
    print("Programming is fun!")
    count = count + 1
```

  - The message is displayed 101 times
    - Here `count` started at **0**
    - And the condition was `count <= 100`
  - How to correct:
    - Make `count` start at **1**, or
    - Make the condition as `count < 100`

Program 1: Repeated Subtraction

- Remember our subtraction program. We asked the user to answer a basic subtraction question. Let's rewrite that program by repeatedly asking the same question if the user enters the incorrect result.
- Remember:
  - Step 1: Problem-Solving Phase
  - Step 2: Implementation Phase

Program 1: Repeated Subtraction

- Step 1: Problem-Solving Phase
  - Let's start by looking at some output

```
Shell
>>> %Run subtraction_repeated.py
 Please answer the following:

        9 - 5 = 4

 You got it!
>>> %Run subtraction_repeated.py
 Please answer the following:

        6 - 4 = 3

 That is incorrect. Please try again:

        6 - 4 = 2

 You got it!
>>>
```

Program 1: Repeated Subtraction

- Step 1: Problem-Solving Phase
  - Like last time:
    - We need to randomly generate two numbers
    - We need to make sure the first number is *not* smaller than the second number
      - If it is, swap them using simultaneous assignment
    - Now, we ask the user to enter an answer
      - And we save the their input a variable called `answer`
    - Next we have a `while` loop
    - In this loop, we will repeatedly tell them their answer is incorrect and will re-ask them the same question
    - What is the condition of this while loop?
    - The loop executes as long as `num1 - num2 != answer`

Program 1: Repeated Subtraction

- Step 2: Implementation Phase

```
import random

# 1. Generate two random single-digit integers
number1 = random.randint(0, 9)
number2 = random.randint(0, 9)

# 2. If number1 < number2, swap number1 with number2
if number1 < number2:
    number1, number2 = number2, number1

# 3. Prompt the student to answer the subtraction question
print("Please answer the following:\n")
print("\t{} - {} = ".format(number1, number2), end = '')
answer = int(input())

# 4. Repeatedly ask the question until the answer is correct
while number1 - number2 != answer:
    print("\nThat is incorrect. Please try again:\n")
    print("\t{} - {} = ".format(number1, number2), end = '')
    answer = int(input())

print("\nYou got it!")
```

Loop Design Strategies

- Some loops are straightforward
  - Others require some thought
- Consider the following loop-design strategy:
  1. Identify the statements that need to be repeated.
  2. Wrap these statements in a loop like this:

```
while True:
    Statements
```

  3. Code the loop-continuation-condition and add appropriate statements for controlling the loop.

```
while loop-continuation-condition:
    Statements
    Additional statements for controlling the loop
```

Program 2: Guessing Number Game

- You should write a program to play the famous number guessing game from childhood.
  - "I have a number from 1 to 100. Guess that number in as few guesses as possible."
  - Your program should randomly generate a number and then ask the user to repeatedly guess that number until they finally get it correct.
- Remember:
  - Step 1: Problem-Solving Phase
  - Step 2: Implementation Phase

Program 2: Guessing Number Game

- Step 1: Problem-Solving Phase
  - Let's start by looking at a run of the program...

```
Shell

>>> %Run number_guessing_game.py
 ******************************************
 *          Number Guessing Game          *
 ******************************************
     Guess a number between 1 and 100.
     Enter your guess: 50
     Your guess is too low
     Enter your guess: 75
     Your guess is too low
     Enter your guess: 87
     Your guess is too low
     Enter your guess: 93
     Your guess is too high
     Enter your guess: 90
     Your guess is too low
     Enter your guess: 92
     Yes, the number is 92

>>>
```

Program 2: Guessing Number Game

- Step 2: Implementation Phase
  - Remember the design strategy:

1. **Identify the statements that need to be repeated.**

Program 2: Guessing Number Game

```
**************************
*          Number Guessing G
**************************
    Guess a number between 1
    Enter your guess: 50
    Your guess is too low
    Enter your guess: 75
    Your guess is too low
    Enter your guess: 87
    Your guess is too low
    Enter your guess: 93
    Your guess is too high
    Enter your guess: 90
    Your guess is too low
    Enter your guess: 92
    Yes, the number is 92
```

- Step 2: Implementation Phase

```
import random

# STEP 0: Statements OUTSIDE the Loop
number = random.randint(1, 100)
print("*****************************************")
print("*           Number Guessing Game        *")
print("*****************************************")
print("    Guess a number between 1 and 100.")

# REMEMBER STEP 1:
# Identify the statements that must be repeated...

    # Prompt the user to guess the number
    guess = eval(input("    Enter your guess: "))

    # Use if/elif/else statement to print appropriate message
    if guess == number:
        print("    Yes, the number is", number)
    elif guess > number:
        print("    Your guess is too high")
    else:
        print("    Your guess is too low")
```

Program 2: Guessing Number Game

- Step 2: Implementation Phase
  - Remember the design strategy:

1. Identify the statements that need to be repeated.
2. **Wrap these statements in a loop like this:**

```
while True:
    Statements
```

Program 2: Guessing Number Game

- Step 2: Implementation Phase

```
import random

# STEP 0: Statements OUTSIDE the Loop
number = random.randint(1, 100)
print("**************************************")
print("*          Number Guessing Game         *")
print("**************************************")
print("    Guess a number between 1 and 100.")

# REMEMBER STEP 2:
# Wrap these statements in a while True loop
while True:
    # Prompt the user to guess the number
    guess = eval(input("    Enter your guess: "))

    # Use if/elif/else statement to print appropriate message
    if guess == number:
        print("    Yes, the number is", number)
    elif guess > number:
        print("    Your guess is too high")
    else:
        print("    Your guess is too low")
```

Program 2: Guessing Number Game

- Step 2: Implementation Phase
  - Remember the design strategy:

1. Identify the statements that need to be repeated.
2. Wrap these statements in a loop like this:

```
while True:
    Statements
```

3. **Code the loop-continuation-condition and add appropriate statements for controlling the loop.**

```
while loop-continuation-condition:
    Statements
    Additional statements for controlling the loop
```

Program 2: Guessing Number Game

- Step 2: Implementation Phase

```
import random

# Generate a random number to be guessed
number = random.randint(1, 100)
print("*************************************")
print("*          Number Guessing Game         *")
print("*************************************")
print("    Guess a number between 1 and 100.")

# Note that we must initialize guess to -1 in order to enter loop
guess = -1
while guess != number:
    # Prompt the user to guess the number
    guess = eval(input("    Enter your guess: "))

    # Use if/elif/else statement to print appropriate message
    if guess == number:
        print("    Yes, the number is", number)
    elif guess > number:
        print("    Your guess is too high")
    else:
        print("    Your guess is too low")
```

Program 3: Larger Subtraction Quiz

- Now that we know loops, we can make a better subtraction quiz!
  - Let’s ask the user how many subtraction questions they would like to answer.
  - We will then loop exactly that many times
  - At the end, we will tell them how many were correct
  - We will tell them how long they took to complete the quiz
- Remember:
  - Step 1: Problem-Solving Phase
  - Step 2: Implementation Phase

Program 3: Larger Subtraction Quiz

- Step 1: Problem-Solving Phase
  - Let’s think about what is involved here:
    1. Asking how many questions should be on the quiz.
       - That’s easy. Just ask and save answer as `num_questions`
    2. For a single iteration of the loop, what happens in the loop?
       - We must generate two random numbers
       - We must swap them if the first number is smaller than the second
       - We must ask the question, read user answer, print a correct/incorrect message, and finally update `num_correct` if necessary
    3. How do we loop that many times?
       - Use a variable `count` and loop `while count <= num_questions`
    4. And how do we time the quiz? Use `time.time()`
       - Use it once at beginning and once at the end…then subtract the difference!
       - The difference will be the number of seconds used during the quiz

Program 3:
Larger Subtraction Quiz

- Step 2: Implementation Phase

```
import random
import time

# Generate a random number to be guessed
number = random.randint(1, 100)
print("***************************************")
print("*            Subtraction Quiz          *")
print("***************************************")
print("    How many questions would you like")
print("    to answer (5 to 20): ", end = "")

num_questions = int(input())

# Setup variables
num_correct = 0 # keeps track of the number of correct answers
count = 1       # variable used to count number of questions
time_start = time.time() # get starting time in seconds
```

Program 3: Larger Subtraction Quiz

- Step 2: Implementation Phase

```
# Notice the condition of the loop using <= because started count at 1
while count <= num_questions:
    # Generate two random numbers
    num1 = random.randint(1, 9)
    num2 = random.randint(1, 9)

    # Swap the two numbers if num1 is smaller than num2
    if num1 < num2:
        num1, num2 = num2, num1

    # Ask question and save answer
    print("\n    Question {}:".format(count))
    user_answer = int(input("    {} - {} = ".format(num1, num2)))

    # Test correctness
    if user_answer == num1 - num2:
        print("      Correct!")
        num_correct += 1
    else:
        print("      Incorrect.")

    # Important: Update count variable!!!
    count += 1
```

Program 3: Larger Subtraction Quiz

- Step 2: Implementation Phase

```
# Get ending time and calculate the total time used
time_end = time.time()
time_used = time_end - time_start

# Print Closing Message
print("\n-----------------------------------------")
print("Quiz Results:")
print("    You answered {} out of {} questions correct.".format(num_correct,
                                                          num_questions))
print("    Time: {:.1f} seconds".format(time_used))
```

Program 3: Larger Subtraction Quiz

- Sample runs of program:

```
****************************************
*            Subtraction Quiz          *
****************************************
    How many questions would you like
    to answer (5 to 20): 5

    Question 1:
    6 - 1 = 5
      Correct!

    Question 2:
    6 - 6 = 0
      Correct!

    Question 3:
    9 - 3 = 4824
      Incorrect.

    Question 4:
    8 - 1 = 7
      Correct!

    Question 5:
    5 - 2 = 3
      Correct!

----------------------------------------
Quiz Results:
    You answered 4 out of 5 questions correct.
    Time: 13.4 seconds
```

```
****************************************
*            Subtraction Quiz          *
****************************************
    How many questions would you like
    to answer (5 to 20): 5

    Question 1:
    9 - 5 = 4
      Correct!

    Question 2:
    6 - 6 = 0
      Correct!

    Question 3:
    7 - 5 = 2
      Correct!

    Question 4:
    9 - 6 = 3
      Correct!

    Question 5:
    8 - 1 = 7
      Correct!

----------------------------------------
Quiz Results:
    You answered 5 out of 5 questions correct.
    Time: 5.4 seconds
```

Controlling `while` Loops

- We've seen a couple ways to control a while loop
  - Using a count variable and counting some number of iterations
  - Checking for some condition
    - Such as the number guessing game

```
while guess != number
```

- There are other ways to control the loop as well
  - We can control the loop with a user confirmation
  - And we can control the loop with a sentinel value

Controlling `while` Loops

- Controlling a Loop with User Confirmation
  - The last program (Subtraction Quiz) controlled the loop with a count
    - And we iterated between 5 or 20 times depending on the user input
  - We let the user control the number of iterations
  - How?
    - We could ask them if they want to answer another question
    - We save their answer (“Y” or “N”)
  - We then use the answer as a condition of the loop

```
while another_question == "Y":
```

Controlling `while` Loops

- Controlling a Loop with User Confirmation
  - Make a new Subtraction Quiz program
    - Copy/paste your last code
  - Edit it to make the loop user controlled
  - Here's the idea:

```
another_question = "Y"   # used in loop condition to continue quiz (or not)

# Notice we do the loop at least one time because we initialized
# the another_question variable to "Y"
while another_question.lower() == "y":
    #...Loop body here...

    # Important: UPDATE another_question loop condition variable
    print("\n    Would you like to answer another")
    print("    question (Y or N)? ", end = "")
    another_question = input()
    count += 1 # used to print Question number
```

Controlling `while` Loops

- Controlling a Loop a Sentinel Value
  - Another technique is to designate a special input value to stop the loop
    - This value is called the sentinel value
    - And a loop that uses a sentinel value is called a sentinel-controlled loop
  - Consider the following example…

Program 4:
Summing until Sentinel Value

- Write a program that repeatedly asks the user to enter integer values.
  - Your program should sum up all these values, saving the result in a variable called `sum`.
  - Your program should count how many values were entered, saving the total in a variable called `count`.
  - Your program should stop reading values once the integer 0 is entered.
- Remember:
  - Step 1: Problem-Solving Phase
  - Step 2: Implementation Phase

Program 4: Summing until Sentinel Value

- Step 1: Problem-Solving Phase
  - Use the design strategy!
  - So first think about what should be repeated inside the loop…
    - You should ask the user to enter a value
    - We need to add that to the running sum
    - And we need to increase count by 1
  - Now, wrap those statements in a While True block
  - Finally, add on the condition of the while loop

Program 4: Summing until Sentinel Value

- Step 2: Implementation Phase

```
# Variables used in program
sum = 0
count = 0

# Notice that we have to scan the data value once before the loop also
data = int(input("Enter an integer (input ends if it is a 0): "))

# The loop continues as long as data does *not* equal zero
while data != 0:
    sum += data
    count += 1
    data = int(input("Enter an integer (input ends if it is a 0): "))

print("\nYou entered {} values for a total sum of {}.".format(count, sum))
```

  - Note: we end up having to repeat a line of code
    - We prompt and scan the data value inside the loop
    - And we also prompt and scan the data once before the loop

Controlling `while` Loops

- Limitations of Python while loop structure
  - Often you will absolutely want to run your loop at least one time
  - Meaning, regardless of the condition, you want to at execute all the statements inside the loop at least once
    - And this is what we needed in the last example
    - We wanted to read a user integer at least one time
  - Most languages have a `do/while` loop
    - In short, this loop structure "does" (the do part) the loop one time
    - Then, the continue condition is checked at the end of the loop
    - This would have been a better solution to the last problem

Controlling `while` Loops

- Limitations of Python `while` loop structure
  - Python does not have a `do/while` loop structure
  - So what is a workaround?
  - If we have a problem where we absolutely want to “do” the loop one time, regardless of condition, how can we do this in Python?
  - Simple!
    - And in fact, the solution is common in programming
    - We just use a `while True:` loop
      - Meaning…the condition is always true!
    - Then, inside the loop, we have a an `if` statement
    - If the specified condition is met, we use `break` to exit the loop

Program 4:
Summing until Sentinel Value

- Step 2: Implementation Phase
  - Let's modify the last program with this new idea…

```
# Variables used in program
sum = 0
count = 0

while True:
    data = int(input("Enter an integer (input ends if it is a 0): "))

    # Check if the entered value is 0. If so, BREAK
    if data == 0:
        break

    # If we did *not* break, increase sum and increment count
    sum += data
    count += 1

print("\nYou entered {} values for a total sum of {}.".format(count, sum))
```

  - Notice that the logical order of the instructions inside the loop had to change

Controlling `while` Loops

Check Yourself

- How many times are the following loop bodies repeated? What is the printout of each loop?

```
i = 1
while i < 10:
    if i % 2 == 0:
        print(i)
```

(a)

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

(b)

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

(c)

  - (a) is infinite and prints nothing
  - (b) is infinite and prints nothing
  - (c) loops 9 times and prints 2 4 6 8 (each on a different line)

Controlling `while` Loops

Check Yourself

- Suppose the input is **2 3 4 5 0** (one number per line). What is the output of the following code?

```
number = eval(input("Enter an integer: "))
max = number

while number != 0:
    number = eval(input("Enter an integer: "))
    if number > max:
        max = number

print("max is", max)
print("number", number)
```

The `for` Loop

- Often you will use a loop to iterate a specific number of times
  - And we use a counter to count the number of iterations
  - This is called a *counter-controlled loop*
  - Example:

```
# Initialize loop-control variable
i = initial_value

# Iterate as long as i < end_value
while i < end_value:
    # Loop body
    ...
    # Adjust loop-control variable
    i += 1
```

The `for` Loop

- We can use a for loop to simplify the last example:

```
for i in range(initialValue, endValue):
    # Loop body
```

  - The general syntax is:

```
for var in sequence:
    # Loop body
    # usually, we do something with “var”
```

    - Here, `var` stands for variable
      - You can name it what you want…you are the programmer!
    - A `sequence` holds multiple items of data, stored one after another
    - We’ll study different types of sequences later in the semester

The `for` Loop

- We can use a for loop to simplify the last example:
  - Example for loop:

```
for i in range(4, 8):
    print(i)
```

    - Note what gets printed to the output:

```
Shell
>>> %Run for_loop1.py
 4
 5
 6
 7
>>>
```

    - So we loop from **`initial_value`** to **`end_value - 1`**

The `for` Loop

- We can use a for loop to simplify the last example:
  - Another example:

```
for sarah in range(5):
    print(sarah)
```

    - Output:

```
Shell
>>> %Run for_loop1.py
  0
  1
  2
  3
  4
>>>
```

    - This would have been the same as: `for i in range(0, 5):`

The `for` Loop

- We can use a for loop to simplify the last example:
  - Number of arguments of `range()` method:
    - One argument:
      - If there is only one argument, such as range(5), this assumes an initial_value of 0
    - Two arguments:
      - If there are two arguments, such as range(5, 10), this is gives the initial_value (5) and the end_value (10)
        - Although, remember, the loop does *not* execute on 10
    - Three arguments:
      - If there are three arguments, the first two are initial_value and end_value
      - The third argument is the **step size**
      - Normally, step size is assumed to be +1
        - Meaning, just add one to the counter at each iteration
      - But we can use a different step size, and even a negative step size

The `for` Loop

- We can use a for loop to simplify the last example:
  - Another example (step size 2):

```
for mike in range(1, 10, 2):
    print(mike)
```

- Output:

```
Shell
>>> %Run for_loop1.py
 1
 3
 5
 7
 9
>>>
```

- So the step size is 2 and we stop before 10

The `for` Loop

- We can use a for loop to simplify the last example:
  - Another example (step size 5):

```
for barbara in range(0, 30, 5):
    print(barbara)
```

- Output:

```
Shell
>>> %Run for_loop1.py
 0
 5
 10
 15
 20
 25

>>>
```

- So the step size is 5 and we stop before 30

The `for` Loop

- We can use a for loop to simplify the last example:
  - Another example (***<u>counting backwards</u>***):

```
for x in range(5, 0, -1):
    print(x)
```

  - Output:

```
Shell
>>> %Run for_loop1.py
 5
 4
 3
 2
 1
>>>
```

  - So the step size is -1 and we stop before 0

The `for` Loop

Check Yourself

- Suppose the input is `2 3 4 5 0` (one number per line). What is the output of the following code?

```
number = 0
sum = 0

for count in range(5):
    number = eval(input("Enter an integer: "))
    sum += number

print("sum is", sum)
print("count is", count)
```

The `for` Loop

Check Yourself

- Convert the following `for` loop into a `while` loop:

```
sum = 0
for i in range(1001):
    sum = sum + i
```

- Answer:

```
i = 0
sum = 0
while i < 1001:
    sum = sum + i
    i += 1
```

The `for` Loop

Check Yourself

- Count the number of iterations of each of the following `for` loops (assume n = 10)

```
count = 0
while count < n:
    count += 1
```

Iterations: 10 (a)

```
for count in range(n):
    print(count)
```

Iterations: 10 (b)

```
count = 5
while count < n:
    count += 1
```

Iterations: 5 (c)

```
count = 5
while count < n:
    count = count + 3
```

Iterations: 2 (d)

The `for` Loop

- Which loop should you use?
  - Each has a purpose!
  - When should **`for`** loops be used?
    - when you know how many iterations you need
    - or when you know the range of values to loop over
  - When should **`while`** loops be used?
    - When you should loop, indefinitely, as long as a given condition is true

Nested Loops

- Loops can be nested inside other loops!
  - The first loop is considered the outer loop
  - Then, inside this outer loop can be one or more inner loops
  - Each time the outer loop is repeated, the inner loops are again restarted and begin anew

Nested Loops

- Loops can be nested inside other loops!
  - Example:

```
for i in range(1, 4):
    for j in range(1, 4):
        print("i: {}    j: {}".format(i, j))
```

  - Output:

```
Shell
>>> %Run for_loop1.py
 i: 1    j: 1
 i: 1    j: 2
 i: 1    j: 3
 i: 2    j: 1
 i: 2    j: 2
 i: 2    j: 3
 i: 3    j: 1
 i: 3    j: 2
 i: 3    j: 3
```

Program 5: Multiplication Table

- Write a program that will display the following multiplication table:

```
              Multiplication Table
          1   2   3   4   5   6   7   8   9
  -----------------------------------------
  1 |     1   2   3   4   5   6   7   8   9
  2 |     2   4   6   8  10  12  14  16  18
  3 |     3   6   9  12  15  18  21  24  27
  4 |     4   8  12  16  20  24  28  32  36
  5 |     5  10  15  20  25  30  35  40  45
  6 |     6  12  18  24  30  36  42  48  54
  7 |     7  14  21  28  35  42  49  56  63
  8 |     8  16  24  32  40  48  56  64  72
  9 |     9  18  27  36  45  54  63  72  81
```

- Remember:
  - Step 1: Problem-Solving Phase
  - Step 2: Implementation Phase

Program 5: Multiplication Table

- Step 1: Problem-Solving Phase
  - How many loops do we need?
  - Answer:
    - 2 loops!
    - The outer loop will iterate 9 times
      - One for each row
        - Suggestion: use the word row as your variable name!
      - Code this first, so you can feel good about what is being printed
    - Then, for EACH iteration of the outer loop, we also have an inner loop
      - And the inner loop will also iterate 9 times
      - This inner loop prints the row values
      - Example:
        - If row = 3, then we print 3*1, 3*2, 3*3, 3*4, 3*5, etc.

Program 5: Multiplication Table

- Step 2: Implementation Phase

```
# Print Header
print("            Multiplication Table")
print("     ", end = "")
for i in range(1, 11):
    print("{:>4d}".format(i), end = "")
print("\n-----", end = "")
for i in range(1, 11):
    print("{:4s}".format("----"), end = "")
print()

# Print BODY - use two nested FOR loops
for row in range(1, 11):
    # Print Row header information
    print("{:2d} | ".format(row), end = "")
    for col in range(1, 11):
        print("{:>4d}".format(row * col), end = "")
    # Now, print a newline after each row
    print()
```

Nested Loops

- Careful!
  - Nested loops can be surprisingly short in # of lines of code
    - but they can take a long time to run!
  - Consider the following example:

```
for i in range(1000):
    for j in range(1000):
        for k in range(1000):
            print("{:>7d}{:>7d}{:>7d}".format(i, j, k))
```

- That's three nested loops!
  - And each loop, on its own, runs 1000 times…but they are nested…
- That innermost print statement will get executed 1,000,000,000 times!!!

Nested Loops

Check Yourself

- Trace the following program
  - Draw a table and show the values of i and j at each iteration of the loops

```
for i in range(1, 5):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
```

  - Output:

0 0 1 0 1 2 0 1 2 3

Program Trace

i	j
1	0
2	0
2	1
3	0
3	1
3	2
4	0
4	1
4	2
4	3

Nested Loops

Check Yourself

- Trace the following program
  - Draw a table and show the values of i and j at each iteration of the loops

```
i = 0
while i < 5:
    for j in range(i, 1, -1):
        print(j, end = " ")
    print("****")
    i += 1
```

```
****
****
2 ****
3 2 ****
4 3 2 ****
```

Program Trace

i	j
0	0
1	1
2	2
3	3
3	2
4	4
4	3
4	2

Warmup/Stretching

- Go to my repl.it
- Click on the 2280_NestedLoops_Warmup
- Fork that code
- Stretching Exercise #1:
  - Write a loop to perform the following:

```
Enter an integer: 4

Here are 4 lines, each with an asterisk:
*
*
*
*
```

Warmup/Stretching

- Stretching Exercise #2:
  - Write a loop to perform the following:

```
Enter an integer: 4

Here are 4 lines, each with an asterisk:
*
*
*
*

And now we print a triangle of asterisks:
*
**
***
****
```

Program 6: GCD

- Write a program to ask the user to enter two positive integers. You should then find the greatest common divisor (GCD) and print the result to the user.
- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 6: GCD

- **Step 1**: Problem-solving Phase
  - First question:
  - "What's a GCD???"

  - Answer:
    - Greatest Common Divisor
      - aka Greatest Common Factor (GCF)
  - For Clarity:
    - Given two integers, the GCD is the largest integer that perfectly divides into (or factors from) both of the given integers

Program 6: GCD

- **Step 1**: Problem-solving Phase
  - GCD
    - Find the largest integer that divides both numbers
    - GCD(4,2) = 2
    - GCD(16,24) = 8
    - GCD(25, 60) = 5
  - Cool, so are we ready to code?
    - NO!
  - Always, first think about the problem
  - And understand the solution 200% before coding!
  - So how do you calculate the GCD? **Discuss this in groups.**

Program 6: GCD

- **Step 1**: Problem-solving Phase
  - GCD(n1, n2)
    - You know that the number 1 is a common divisor
      - because 1 divides into everything
    - But is 1 the **greatest** common divisor**?**
    - So you can check the next values, one by one
      - Check 2, 3, 4, 5, 6, …
      - Check if that number "cleanly divides" both integers
        - How? Mod! If the mod (%) is zero, this means no remainder.
      - Keep checking all the way up to the smaller of `n1` or `n2`
    - Whenever you find a new common divisor, this becomes the new `gcd`
    - After you check all the possibilities, the value in the variable `gcd` is the GCD of n1 and n2

Program 6: GCD

- **Step 2**: Implementation Phase

LISTING 5.8 GreatestCommonDivisor.py

```
# Prompt the user to enter two integers
n1 = eval(input("Enter first integer: "))
n2 = eval(input("Enter second integer: "))

gcd = 1
k = 2
while k <= n1 and k <= n2:
    if n1 % k == 0 and n2 % k == 0:
        gcd = k
    k += 1

print("The greatest common divisor for",
    n1, "and", n2, "is", gcd)
```

Try re-coding this as a for loop!

```
Enter first integer: 125 ↵Enter
Enter second integer: 2525 ↵Enter
The greatest common divisor for 125 and 2525 is 25
```

Nested Loops

■ Check Yourself

- Trace the following program
  - Draw a table and show the values of i and j at each iteration of the loops

```
i = 5
while i >= 1:
    num = 1
    for j in range(1, i + 1):
        print(num, end = "xxx")
        num *= 2
    print()
    i -= 1
```

```
1xxx2xxx4xxx8xxx16xxx
1xxx2xxx4xxx8xxx
1xxx2xxx4xxx
1xxx2xxx
1xxx
```

Program Trace

i	j
5	1
5	2
5	3
5	4
5	5
4	1
4	2
4	3
4	4
3	1
3	2
3	3
2	1
2	2
1	1

Nested Loops

Check Yourself

- Trace the following program
  - Draw a table and show the values of i and j at each iteration of the loops

```
i = 1
while i <= 5:
    num = 1
    for j in range(1, i + 1):
        print(num, end = "G")
        num += 2
    print()
    i += 1
```

```
1G
1G3G
1G3G5G
1G3G5G7G
1G3G5G7G9G
```

Program Trace

i	j
1	1
2	1
2	2
3	1
3	2
3	3
4	1
4	2
4	3
4	4
5	1
5	2
5	3
5	4
5	5

Minimizing Numerical Errors

- Summary:
  - Use of floating-point numbers can cause numerical errors
  - Run the following code to see:

```
x = 1.0
x -= .1
x -= .1
x -= .1
x -= .1
x -= .1
print(x)
```

    - We would expect to see **0.5** printed
    - Instead, we get **0.5000000000002**
    - The answer is not perfectly accurate…it's a little bit off
    - This is due to the limitation of the hardware, something you'll learn more about in Computer Organization & Architecture

Minimizing Numerical Errors

- **<u>Never</u>** use floating-point values as loop conditions
  - For example, consider the following code:

```
# Initialize sum
sum = 0

# Add 0.01, 0.02, ..., 0.99, 1 to sum
i = 0.01
while i <= 1.0:
    sum += i          # we add i to the running sum
    i = i + 0.01      # we "increment" i by 0.01

# Display result
print("The sum is", sum)
```

- If you work this by hand, the <u>expected final value</u> for sum is **50.5**
  - But what actually gets printed is **49.5**
  - Why? Because the value of **i** does not have accurate floating-point values
  - And at the final iteration, **i** is slightly larger than 1 (although it should equal 1)

Minimizing Numerical Errors

- **<u>Never</u>** use floating-point values as loop conditions
  - If you need to sum up values similar to the last example, use a while loop or a for loop as follows:

```
# Initialize sum
sum = 0
count = 0
i = 0.01
while count < 100:
    sum += i
    i = i + 0.01
    count += 1 # Increase count

# Display result
print("The sum is", sum)
```

```
# Initialize sum
sum = 0

i = 0.01
for count in range(100):
    sum += i
    i = i + 0.01

# Display result
print("The sum is", sum)
```

  - In both cases, we simply used an integer count to serve as a counter variable, counting the 100 iterations of the loops

Program 7: Future Tuition

- A university charges $10,000 per year for study (tuition). The cost of tuition increases 7% every year. Write a program to determine how many years until the tuition will increase to $20,000.
- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 7: Future Tuition

- **Step 1**: Problem-solving Phase
  - THINK:
    - How do we solve this on paper?
      - Cost of Year0: $10,000
      - Cost of Year1: Year0*1.07
      - Cost of Year2: Year1*1.07
      - Cost of Year3: Year2*1.07
      - …
    - So keep computing the tuition <u>until</u> it is at least $20,000
    - Once you get to $20,000, print the number of years taken

Program 7: Future Tuition

- **Step 1**: Problem-solving Phase
  - THINK:
    - Now a closer look at some of the code:

```
tuition = 10000
year = 0
tuition = tuition*1.07
year += 1
tuition = tuition*1.07
year += 1
tuition = tuition*1.07
year += 1
...
```

    - So we would keep doing this until `tuition` is greater than or equal to $20,000
    - Then, at that point, we print the value in variable `year`
    - How to do this? Use a **while** loop!

Program 7: Future Tuition

- **Step 2**: Implementation Phase

LISTING 5.9 FutureTuition.py

```python
year = 0  # Year 0
tuition = 10000  # Year 1

while tuition < 20000:
    year += 1
    tuition = tuition * 1.07

print("Tuition will be doubled in", year, "years")
print("Tuition will be $" + format(tuition, ".2f"),
      "in", year, "years")
```

```
Tuition will be doubled in 11 years
Tuition will be $21048.52 in 11 years
```

`break` and `continue`

- Extra control within loops:
  - Python uses two additional keywords that provide more control within loops: **`break`** and **`continue`**
  - **`break`**:
    - We've previously jumped ahead and already saw this
    - What does **`break`** do?
      - You can use the **`break`** statement, inside a loop, to immediately terminate/stop the loop
      - Example: maybe the loop is running indefinitely
      - But you want to stop the loop once some condition is True
      - So you test for this condition, and, if True, you use **`break`**
        - This will immediately terminate/stop that specific loop

break and **continue**

- Extra control within loops:
  - **break**:
    - Example:

LISTING 5.11 TestBreak.py

```
sum = 0
number = 0

while number < 20:
    number += 1
    sum += number
    if sum >= 100:
        break

print("The number is", number)
print("The sum is", sum)
```

- The program simply adds the integers 1 through 20 to the variable sum.
- But once sum is greater or equal to 100, the loop stops by using the keyword `break`.

```
The number is 14
The sum is 105
```

break and **continue**

- Extra control within loops:
  - Python uses two additional keywords that provide more control within loops: **break** and **continue**
  - **continue**:
    - What does **continue** do?
      - You can use the **continue** statement, inside a loop, to immediately terminate/stop the current iteration of the loop
      - For clarity:
        - **continue** does NOT terminate the entire loop
        - **continue** only stops the current iteration of the loop
        - So while **break** breaks out of the entire loop
        - You can consider **continue** as breaking out of the current iteration
      - What really happens?
        - The program jumps to “after” the last line of the loop
        - Which really means it goes back to the beginning of the loop

break and **continue**

- Extra control within loops:
  - **continue**:
    - Example:

LISTING 5.12 TestContinue.py

```python
sum = 0
number = 0

while number < 20:
    number += 1
    if number == 10 or number == 11:
        continue
    sum += number

print("The sum is", sum)
```

- The program adds integers 1 through 20 to the variable sum
- But, the program SKIPS the integers 10 and 11
- So when number is 10 or number is 11, the iteration terminates and those values are not added to the sum.

```
The sum is 189
```

`break` and `continue`

- Extra control within loops:
  - So when do we use `break` and `continue`?
  - Well, you are the programmer! So you choose!
  - But when is it a good idea?
    - Whenever it simplifies the logic and the code
  - We'll show two more examples of the same problem
    - One coded with a `break`
    - And the other without a `break`
  - And on this problem, the `break` most certainly simplifies the logic and the code

break and continue

- Extra control within loops:
  - Example:
    - Given an integer as input, write a program to find the smallest factor of that integer other than 1.
    - You could write this as follows:

```
n = eval(input("Enter an integer >= 2: "))
factor = 2
while True:
    # IF this is an actual factor...remainder is 0
    if n % factor == 0:
        break # so we break!
    # otherwise, increment factor and try again
    factor += 1
print("The smallest factor other than 1 for", n, "is", factor)
```

`break` and `continue`

- Extra control within loops:
  - Example:
    - Given an integer as input, write a program to find the smallest factor of that integer other than 1.
    - Or you can write it without a break statement:

```
n = eval(input("Enter an integer >= 2: "))
found = False
factor = 2
while factor <= n and not found:
    if n % factor == 0:
        found = True
    else:
        factor += 1
print("The smallest factor other than 1 for", n, "is", factor)
```

      - So this works
      - But the code with `break` works cleaner and makes more sense

Program 8: First 50 Primes

- Write a program to find (and print out) the first 50 prime numbers, printing exactly ten prime numbers per line.

```
The first 50 prime numbers are

   2    3    5    7   11   13   17   19   23   29
  31   37   41   43   47   53   59   61   67   71
  73   79   83   89   97  101  103  107  109  113
 127  131  137  139  149  151  157  163  167  173
 179  181  191  193  197  199  211  223  227  229
```

- Remember:
  - Step 1: Problem-solving Phase
  - Step 2: Implementation Phase

Program 8: First 50 Primes

- Step 1: Problem-solving Phase
  - Break this into two parts
  - Start by solving the problem of testing if a given number is a prime number
    - We’ve done that before and you likely have the code
  - Then, once that is done, wrap that in a Loop
    - Problem says to find the first 50 primes
    - How many numbers will we need to test to find the first 50 prime numbers?
    - Who knows!
    - Thus, we need an open-ended while loop!

PYTHON BOOT CAMP

Module 5: Loops