#8: While Loops and Break

SAMS SENIOR NON-CS TRACK

Last Time

Use **else** and **elif** statements to make multiple-option decisions

Use **nesting** to combine if statements with other if statements or functions

Ex 4-2 Feedback

Most of the assignment went well, but many of you didn't get to lineIntersection. Let's go over that quickly.

Today's Learning Goals

Use a **while loop** to repeat actions until a certain condition is met

Use **break** and **nesting** to change the control flow of while loops

While Loops

Repeating Actions

Say you want to write a program that prints out the numbers from 1 to 10. Right now, that would look like:

```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
```

Loops

A **loop** is a control structure that lets us repeat actions, so that we don't need to write out similar code over and over again.

Loops are generally most powerful if we can find a **pattern** between the repeated items. This lets us separate out the parts of the action that are the same each time from the parts that are different.

In printing the numbers from 1 to 10, the part that is the **same** is the action of printing. The part that is **different** is the number that is printed.

While Loops

A **while loop** is a type of loop that keeps repeating until a certain condition is met. It uses the syntax:

```
while <boolean_expression>:
    <loop_body>
```

The while loop checks the Boolean expression, and if it is True, it runs the loop body. Then it checks the Boolean expression again, and if it is still True, it runs the loop body again... etc. When the loop finds that the Boolean expression is False, it exits the loop immediately.

Loop Variables

Let's say we want to program our print-1-to-10 example. The part that stays the same (printing) can be written directly, but the part that changes (the number) will need to be a **variable**, so we can change it as we loop.

To use this loop variable, we'll need to give it an **initial value**, a **way to update**, and a **time to end the loop**. This last part can also be thought of as **when to keep looping**.

```
<initial value>
while <when to keep looping>:
    print(num) # the print is the same
    <way to update>
```

In our 1-to-10 example, we want to **start** the variable at 1, and **end** after it has printed 10. So we set num = 1 at the beginning of the loop and continue looping while num <= 10.

Each number we print is one apart from the previous, so we'll want to set the variable to the next number (num + 1) at each iteration.

```
num = 1
while num <= 10:
    print(num)
    num = num + 1
```

Infinite Loops

Make sure to always update your loop variable in a way that will eventually make the loop condition False! Otherwise, you might end up with an **infinite loop**.

```
i = 1
while i > 0:
    print(i)
    i = i + 1
```

If you get stuck in an infinite loop, press the button that looks like a lightning bolt above the interpreter to make the program stop. Then investigate your program to figure out why the variable never makes the condition False. Printing out the loop variable can help with this.

Exercise 1: print evens

Go to the schedule page and download the starter file for today's lecture. You'll write exercise code under the comment with the exercise's number.

Exercise 1: write a few lines of code that print the **even** numbers from 2 to 10, as shown to the right. You must use a loop in your solution for full credit.

Think about how you can update the variable to make this problem easier...

```
2
4
6
8
10
```

Loop Conditions and Variables

We can make the condition of the loop **any Boolean expression we want**. This means that we can update variables in many different ways to solve different problems.

For example, what if we wanted to find the largest power of 3 that is less than 100? Start at the first power (0), loop while the next number is less than 100 (num*3 < 100), and multiply by 3 each time!

```
num = 1
while num * 3 < 100:
    num = num * 3
print(num)
```

Using Multiple Variables

We can also update variables that aren't the loop variable in the loop, in order to generate new kinds of data. As long as we check at least one variable in the condition, we can do whatever we want with the rest!

For example, let's write a program that sums the numbers from 1 to 10.

```
result = 0
num = 1
while num <= 10:
    result = result + num
    num = num + 1
print(result)
```

Tracing Loops

Sometimes it gets difficult to track what a program is doing when we add in loops. We can make this simpler by manually tracing through the values in the variables at each step of the code, including each iteration of the loop.

```
result = 0
num = 1
while num <= 10:
    result = result + num
    num = num + 1
print(result)
```

step	result	num
pre-loop	0	1
iteration 1	1	2
iteration 2	3	3
iteration 3	6	4
iteration 4	10	5
iteration 5	15	6
iteration 6	21	7
iteration 7	28	8
iteration 8	36	9
iteration 9	45	10
iteration 10	55	11
post-loop	55	11

Exercise 2: factorial

Exercise 2: write a few lines of code that set a variable num to hold the value 10, then uses a loop to compute 10! (or 10 factorial) and print it out at the end. Your code should still work if num is changed to hold a different positive integer.

Recall that 10! = 10*9*8*7*6*5*4*3*2*1

Hint: you may need up to three different variables to solve this problem. Think carefully about what they should hold!

Loop Control Flow

Input-Output Loops

Sometimes, we want to write a program that will constantly take input from the user and respond with meaningful output. When dealing with user interaction, we don't know how many times we need to loop- it all depends on what kind of input the user gives!

To do this, we'll tell the loop to keep going forever by making the condition expression True. But we'll still need to add something to make the loop stop eventually...

```
print("Let's introduce everyone in the class!")
while True:
    name = input("What's your name?")
    print("Hi, " + name + "!")
```

Break Statements

In order to exit an infinite input-output loop, we'll use a new statement called **break**. As soon as the program reaches a break statement, it will 'break' out of the loop body and immediately move on to the next statement after the loop.

We'll usually put break inside an if statement, so that we only break when a certain condition is met. We'll talk more about using conditionals in while loops in a bit. For now, you can just plan to use the following structure:

```
print("Let's introduce everyone in the class!")
while True:
    name = input("What's your name?")
    if name == "done":
        break # we just put break on a line by itself
    print("Hi, " + name + "!")
print("Nice to meet everyone!")
```

Exercise 3: number guessing game

Exercise 3: write a line of code that sets a variable num to hold a number between 1 and 10 that you choose. Then write a few lines of code using a loop that asks the user to guess a number between 1 and 10. It should continue asking them to guess until they get it right. When they get it right, print a congratulations message.

A possible interaction is shown to the right.

I'm thinking of a number between 1 and 10...

Guess a number: 8

Try again!

Guess a number: 3

Try again!

Guess a number: 4

You got it!

Nesting in Loops

Just as we could nest conditionals in conditionals and conditionals in functions, we can nest conditionals in loops!

In general, we use conditionals in loops to change the behavior of different iterations based on the loop variable. This can help end the loop early (as with break) or can change the output based on properties of the iteration.

```
while <boolean expression>:
    <while body>
    if <boolean expression>:
        <if body>
    <while body>
```

Example: alternating Boolean

Let's say we want to write a program that alternates A and B with the numbers 1 to 10, as shown to the right. We can do this by using a Boolean to tell whether we should print A or B at each iteration of the loop.

```
aTurn = True
i = 1
while i <= 10:
    if aTurn == True:
        print("A" + str(i))
    else:
        print("B" + str(i))
    i = i + 1
    aTurn = not aTurn
```

A1

B2

A3

B4

A5

B6

A7

B8

A9

B10

Nesting Example: Variable case

We could also write a program that only performs certain statements on a special set of values. This code prints out special statements for numbers between 3 and 7.

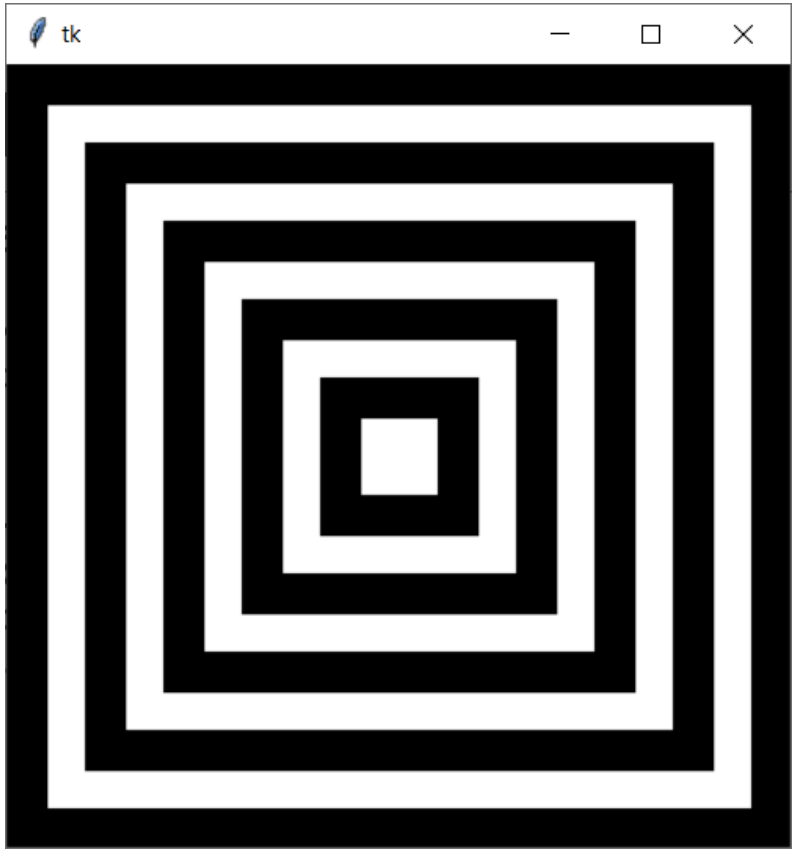
```
i = 1
while i <= 10:
    print(i)
    if i == 5:
        print("That's five!")
    elif 3 <= i and i <= 7:
        print("That's close to five")
    i = i + 1
```

```
1
2
3
That's close to five
4
That's close to five
5
That's five!
6
That's close to five
7
That's close to five
8
9
10
```

Exercise 4: illusion

Exercise 4: write a few lines of tkinter code that create the image shown on the right. Note that you'll need to use a loop to get full credit.

Hint: it's easiest to make this illusion by **overlapping shapes**. Start with the largest black square, then draw the next-largest white square, etc. You'll need to draw 10 squares total.



Today's Learning Goals

Use a **while loop** to repeat actions until a certain condition is met

Use **break** and **nesting** to change the control flow of while loops