#3: Conditionals and Loops

SAMS SENIOR CS TRACK

Last time

Learned how to use the building blocks of programming: data, variables, functions, and graphics

Used algorithmic thinking to solve new problems with programming

Today's Learning Goals

Use conditionals to change program actions based on data

Use loops to repeat actions within a program

Conditionals

Conditionals

Sometimes we need to change what a program does based on the given input. We can do this using **conditional statements**. These statements choose what the program will do next based on a boolean expression.

```
if <boolean_expression>:
    <body_if_true>
```

Conditional Example

In the following example, the code will only print "I see you!" if the boolean variable `visible` is set to `True`. However, it will always print "start" and "finish".

```
print("start")
if visible == True:
    print("I see you!")
print("finish")
```

Else conditions for alternatives

Sometimes we want the program to do one of two possible actions based on the conditions. In this case, instead of writing two if statements, we can write a single if statement and give it an **else**. The else will cover the case when the boolean expression is False.

```
if <boolean_expression>:
    <body_if_true>
else:
    <body_if_false>
```

Conditional Example

Prediction Exercise: What will the following code print?

```
x = 5
if x > 10:
    print("Up high!")
else:
    print("Down low!")
```

Question: What could we change to get the other statement to print instead?

Question: Can we get the program to print out both statements?

Multiple Branches

If we want to have more than two options for what the program can do, we can add one or more **elif** statements in between the initial if and final else. The program will only ever enter one branch of the conditional.

```
if <boolean_expression_A>:
    <body_if_A_True>
elif <boolean_expression_B>:
    <body_if_A_False_and_B_True>
else:
    <body_if_both_False>
```

Multi-Branch Example

The following example shows a three-branch conditional in a function. We don't need to add a return statement outside the conditional- why?

```
def number_sign(x):

    if x > 0:

        return "positive"

    elif x < 0:

        return "negative"

    else:

        return "zero"
```

Exercise: gradeCalculator

Write a program `gradeCalculator` that takes as input `grade` (a number) and prints the letter grade it corresponds to as a string.

90+ is an A, 80-90 is a B, 70-80 is a C, 60-70 is a D, and below 60 is an R.

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)
```

For Loops for Repeated Actions

There's an easier way to repeat actions! You can use a **for loop** to tell the program how many times to repeat a step, and even change the step based on which iteration you're on.

```
for <step_variable> in range(<min_num>, <max_num_plus_one>):
    <steps_to_repeat>
```

So our previous program could be:

```
for i in range(1, 11):
    print(i)
```

Range has multiple arguments

When range has one argument, it represents when the loop should end. This is the maximum number **plus one**. In this case, the starting argument defaults to `0`.

```
range(10) -> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

When range has two arguments, the first argument is the start point and the second is the end.

```
range(2, 10) -> 2, 3, 4, 5, 6, 7, 8, 9
```

When range has three arguments, the first is the start point, the second is the end, and the third is the **step**. The step tells us how much the numbers should change by.

```
range(2, 10, 2) -> 2, 4, 6, 8
range(10, 2, -1) -> 10, 9, 8, 7, 6, 5, 4, 3
```

Exercise: Print Even Numbers

Write a function, `printEvensToN`, that takes as input `n`, an integer, and prints out the even numbers from `0` up to and including `n`.

How could we change this program to instead **sum** the even numbers from `0` to `n`?

For Loops with Strings

We can also use for loops to iterate over data that can be thought of as multiple parts put together in a whole (**iterable**). A string can be thought of as a sequence of letters (**characters**). Using a for loop, we can write a program that loops over each of the characters in order.

```
for <character_variable> in <string>:
    <character_action_body>
```

Example String Loop

Prediction Exercise: what do you think the following code prints?

```
s = "Hello World!"
t = ""
for c in s:
    t = c + t
print(t)
```

While Loops for Uncertain Conditions

For loops are great for circumstances where we know exactly how many times we need to loop. However, this isn't always the case. Sometimes we instead tell a program to loop **until a certain condition is no longer True**. This is like having an if statement that keeps repeating until it becomes False. These are called **while loops**.

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

While loops are different from for loops in several ways, but the most important difference is that **while loops can keep looping forever**. You need to make sure that the loop body will eventually change the boolean expression to be False to avoid this!

Example: sumDigits

Say we want to sum the digits in a number. We can't use a for loop (because numbers are not iterable); we have to use a while loop in which we add each digit to a sum and then remove it.

```
def sumDigits(x):
    total = 0
    while x > 0:
        digit = x % 10
        total += digit # This is shorthand for total = total + digit.
        x = x // 10
    return total
```

Exercise: While Loop

Prediction Exercise: What will the following code return when we call mystery(16)?

```
def mystery(x):
    if x <= 1:
        return 0
    count = 0
    y = 1
    while y < x:
        y = y * 2
        count += 1
    return count
```

Today's Learning Goals

Use conditionals to change program actions based on data

Use loops to repeat actions within a program