Bonus Slide

Ternary Operator

Sometimes you need a conditional, but what it does is very simple. A ternary operator lets you condense a conditional into a single expression.

```
value = result_a if test else result_b
```

is equivalent to

Boolean expression

```
if test:
    value = result_a
else:
    value = result_b
```

Unit 1: Fundamentals of Programming

Loops

15-110 – Monday 09/14

Announcements

- Hw3 was due today
- Hw4 released
- Hw2 revisions due **tomorrow at noon**
- Quizlet2 on Wednesday

Learning Goals

- Identify the **start, stop,** and **update** values of a **loop control variable** that allows repetition over specific values.

- Trace **loops** that repeat actions on data.

For Loops

Repeating Actions is Annoying

Let's write a program that prints out the numbers from 1 to 10. Up to now, that would look like:

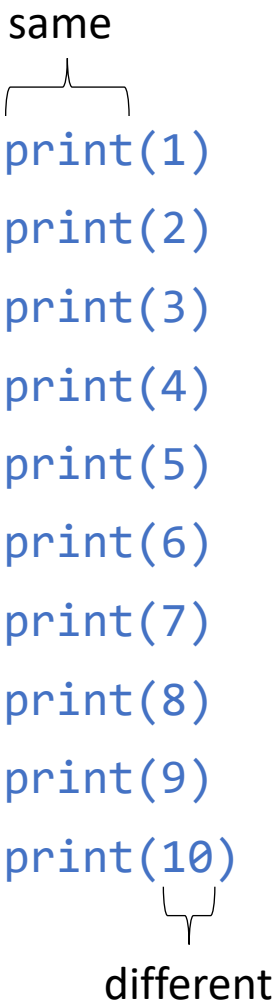
```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
```

Loops Repeat Actions Automatically

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. Noticing patterns 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.



For Loops Implement Repeated Actions

A **for loop** is a control structure that repeats code.

This is a variable with a value that changes in each iteration

This is a value that can be iterated over. Often something that can be broken into smaller parts.

```
for <loopControlVariable> in <iterableValue>:
    <loopBody>
```

This is the part of the algorithm that's the same each iteration.
It uses the loop control variable, which makes the behavior change.

`range` iterates over numbers

The built-in command `range` generates a sequence of numbers based on its arguments. This makes it a common iterable value within for loops.

Using `range`, we can print `1` to `10` in a for loop:

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

Generates numbers 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Loop Control Variables and Range

Start, Stop, and Update

Getting a loop to produce the results you want requires careful design of the **loop control variable**. This is the variable set to a new value in each iteration of the loop.

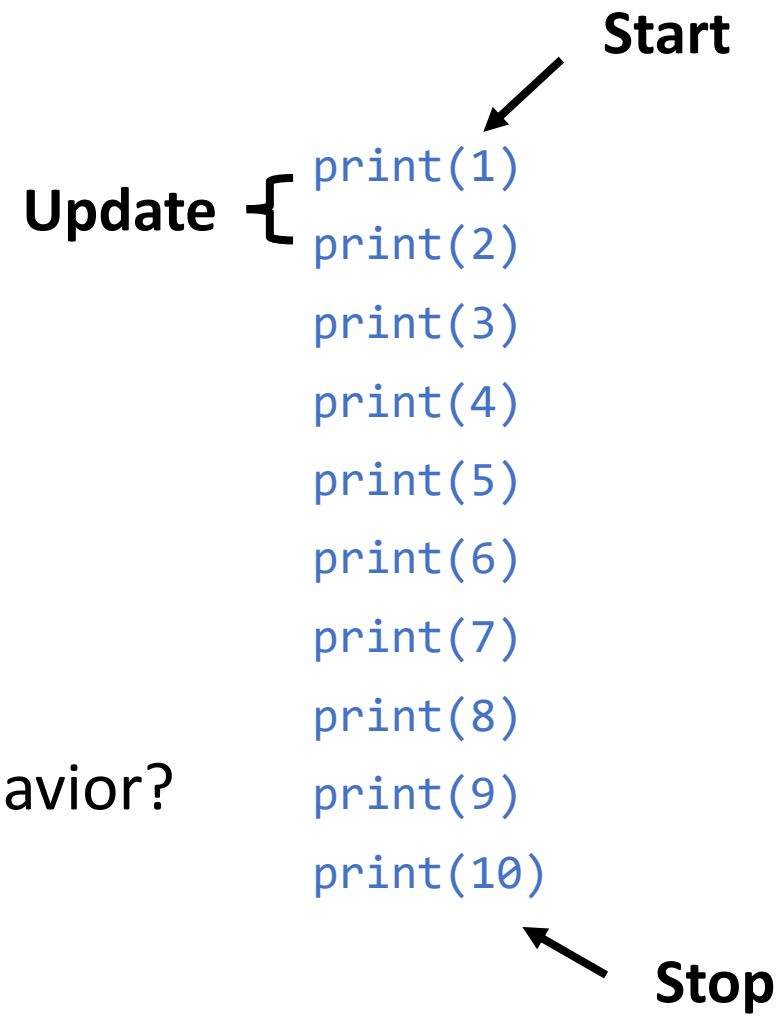
Consider the part of the algorithm that changes in each iteration. Where does that value **start**? Where does the value **stop**? How does the value **update** between iterations?

Example – Print 1 to 10

In the Print 1 to 10 example…

- **Start:** 1
- **Stop:** after 10
- **Update:** add 1 to the variable

Why does `range(1, 11, 1)` produce this behavior?



`range` has start, stop, and step

`range(start, stop, update)` sets up the loop control variable.

start gives the initial value.

stop tells the loop when to stop and is not included in the loop values. This should be immediately **after** the last value.

update tells the value how to change between iterations. Can either add or subtract an integer number each iteration.

`range` Default Arguments

Very often range has basic update of +1. In these cases we call range with just two arguments, **start** and **stop**.

start, stop

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

There are also many cases where the update is +1 and the start is 0. In this case range can take just one argument, **stop**.

stop

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

We'll see this shorthand come up in the next lecture…

Activity: Predict the Printed Values

Each slide will show a loop with a different `range`. Predict what each loop will print.

Raise a number of fingers for the answer you think is right! A=1, B=2, C=3, D=4.

Q1

```
for i in range(3):
    print(i)
```

A: 1, 2, 3

B: 0, 1, 2

C: 0, 1, 2, 3

D: 3

Q2

```
for i in range(1, 5):
    print(i)
```

A: 1, 2, 3, 4

B: 0, 1, 2, 3, 4

C: 1, 2, 3, 4, 5

D: 2, 3, 4, 5

Q3

```
for i in range(5, 1):
    print(i)
```

A: 5, 4, 3, 2

B: 1, 2, 3, 4

C: Nothing is printed

D: 4, 3, 2, 1

Q4

```
for i in range(2, 8, 2):
    print(i)
```

A: 2, 4, 6, 8, 6, 4

B: 2, 4, 6

C: 2, 3, 4, 5, 6, 7

D: 2, 4, 6, 8

Q5

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

A: 5, 4, 3, 2, 1, 0

B: Nothing is printed

C: 0, 1, 2, 3, 4

D: 5, 4, 3, 2, 1

Example: Counting Backwards

Let's go back to loop control variables. How could we count backwards from 10 to 1 instead of from 1 to 10? The loop control variable is the same as before (the number being printed), but its components change.

- **Start:** 10
- **Stop:** after 1
- **Update:** subtract 1 from the variable

```
for i in range(10, 0, -1):
    print(i)
```

Example: Sum Even Numbers from 2 to 100

Let's make it more complicated. What if we want to sum together all the even numbers from 2 to 100?

We'll track multiple data values; which is our loop control variable? **The one where we know the start, stop, and update from the beginning**.

- **Start:** 2
- **Stop:** after 100
- **Update:** add 2 to the variable

```
total = 0
for num in range(2, 101, 2):
    total = total + num
print("Sum:", total)
```

Activity: Compute Factorial

You do: your task is to compute the factorial of some integer variable `n`. This is `1*2*3*…*(n-2)*(n-1)*n`.

What is the loop control variable? What are its **start**, **stop**, and **update** values?

Bonus: see if you can write a short piece of code to compute this value.

Reading Code with Loops

As we start working with more complex loops, we may need to trace how they execute to understand what a piece of code does or find a bug in an algorithm.

Tracing loops gets tricky as you need to repeat the same lines of code with different values. Use **variable tables** to keep track of your work.

Tracing Loops

```
s = ""
for num in range(1, 6):
    numstr = str(num) + "-"
    s = s + numstr
print(s)
```

step	num	numstr	s
pre-loop	---	---	""
iteration 1	1	"1-"	"1-"
iteration 2	2	"2-"	"1-2-"
iteration 3	3	"3-"	"1-2-3-"
iteration 4	4	"4-"	"1-2-3-4-"
iteration 5	5	"5-"	"1-2-3-4-5-"

While Loops

While Loops Repeat While a Condition is `True`

What if we need to use a loop but the loop control variable needs to update in a more complex way?

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

```
while <booleanExpression>:
    <LoopBody>
```

This works like an if statement, except that we keep checking the Boolean expression until it changes from `True` to `False`.

Loop Control Variables in While Loops

While loops are less structured than for loops, which means we need to set up and maintain the loop control variable ourselves. Here's a basic structure we can use:

```
var = <start value>
while <var has not reached stop value>:
    <loopBody>
    var = <update var>
```

Example: Powers of 2

We want to write a loop that prints out all the powers of 2 from 1 to 1000.

- **Start:** 1
- **Stop:** after 1000
- **Update:** multiply the number by 2

```
num = 1
while num <= 1000:
    print(num)
    num = num * 2
```

Infinite Loops Run Forever

What happens if we don't ensure that the while loop's condition eventually becomes `False`? The `while` loop will just keep looping forever! This is called an **infinite loop**.

```
num = 1
while num > 0:
    print(num)
    num = num * 2
```

If you get stuck in an infinite loop, press the 

button to make the program stop. Then investigate your program to figure out why the variable never makes the condition `False`. Printing out the variable that changes can help pinpoint the issue.

Example: Input Loop

We can get multiple data entries directly from the user with the `input` function and a while loop!

- **Start:** initial input request
- **Stop:** when the user gives some signal, maybe `"q"`
- **Update:** ask for another input

Let's ask the user for a stream of numbers and find the total sum.

```
result = 0
val = input("Enter a #, or q to quit:")
while val != "q":
    num = int(val)
    result = result + num
    val = input("Enter a #, or q to quit:")
print("Total sum:", result)
```

For Loops vs. While Loops

When do we use each type of loop?

For loops suffice when iteration is straightforward, which is the majority of cases. It's usually the default.

While loops are best when more direct control over the loop control variable is needed.

Nesting with Loops

Nesting Conditionals in Loops

We showed previously how we can nest conditionals in other conditionals or function definitions. We can do the same thing with loops!

Use **conditionals** inside loops if you want to vary behavior between iterations, or only take actions on certain iterations. For example, we could make an ascii art pattern with the following:

```
for row in range(20):
    if row % 2 == 0:
        print("x-x-x")
    else:
        print("-o-o-")
```

Use % 2 to alternate outcomes – even produces 0, odd produces 1, so rows alternate between if and else.

Nesting Loops in Functions

We can also nest loops in functions.

If we `return` inside a loop, Python **immediately** exits the function and no further iterations will run.

Example: check whether or not a number is prime using a loop over all the number's possible factors.

- **Start:** 2 (why not 1?)
- **Stop:** the number itself
- **Update:** add 1

```
def isPrime(num):
    if num < 2:
        return False
    for factor in range(2, num):
        if num % factor == 0:
            return False
    return True
```

Normally you return in a conditional nested inside the loop, not in the loop body itself. If you return directly in the loop body, it will exit on the first iteration!

Nesting Loops

Importantly, we can also nest loops inside of loops!

We mostly do this with for loops, and mostly when we want to loop over *multiple dimensions*.

```
for <LoopVar1> in range(<endNum1>):
    for <LoopVar2> in range(<endNum2>):
        <bothLoopsBody>
    <justOuterLoopBody>
```

In nested loops, the inner loop is repeated **every time** the outer loop takes a step.

Example: Multiplication Table

Suppose we want to print a multiplication table from 1x1 to 3x2.

```
for x in range(1, 4):
  for y in range(1, 3):
    print(x, "*", y, "=", x * y)
```

Note that the inner loop belongs to the **body** of the outer loop. Every iteration of y happens anew in each iteration of x.

Tracing Nested Loops

We can use another variable table to find the values at each iteration of the loops.

```
for x in range(1, 4):
  for y in range(1, 3):
    print(x, "*", y, "=", x * y)
```

Iteration	x	y	x*y
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	4
5	3	1	3
6	3	2	6

You do: mystery function

You do: read through the following function, then trace the call `mystery(9, 15, 5)` using the variable table to the right. In general terms, what does this function do?

```
def mystery(a, b, n):
    c = 0
    for i in range(a, b+1):
        if i % n == 0:
            c = c + 1
    return c
```

step	i	c

Learning Goals

- Identify the **start, stop,** and **update** values of a **loop control variable** that allows repetition over specific values.

- Trace **loops** that repeat actions on data.