#2-3: Loops

CS SCHOLARS – PROGRAMMING

Learning Goals

Use **for loops** when reading and writing algorithms to repeat actions a specified number of times

Identify **start values, continuing conditions,** and **update actions** for **loop control variables**

Use **while loops** when reading and writing algorithms to repeat actions where the update action or continuing condition is complicated

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

For Loops Implement Repeated Actions

When the pattern represented by a loop is straightforward, we can use a simple control structure to achieve the pattern in a small amount of code.

A **for loop** over a **range** tells the program exactly how many times to repeat an action. The loop body represents the action taken in each step of the pattern.

```
for <loopVariable> in range(<numberActions>):
    <loopBody>
```

The for loop creates a new variable – *loopVariable* – that is repeatedly set to the numbers from 0 to *numberActions*-1. If we use the loop variable in the loop body, we can create different behavior across iterations!

Printing 1 to 10

Let's say we want to print the numbers from 1 to 10. There are ten actions, so we could start the loop as:

```
for number in range(10):
```

For each repetition **(iteration),** we want to print a number. But we can't print the number held in the variable – that starts at 0, not 1. Let's print `number + 1` instead.

```
for number in range(10):
    print(number + 1)
```

Activity: Predict the Result

You do: here's a slightly different for loop. What do you think it will print?

```
for x in range(5):
    print(x, x % 3)
```

Hint: recall that % is the modulo (remainder) operator

Loop Control Variables

Use Loop Control Variables to Design Algorithms

Now that we know the basics of how loops work, we want to write `for` loops that produce specific repeated actions.

First, we need to identify which parts of the repeated action must change in each iteration. This changing part is the **loop control variable(s),** which is updated in the loop body.

We can usually represent this changing part as a combination of a **start value**, an **update action**, and a **continuing condition**. All three need to be coordinated for the loop to work correctly.

Example: Print 1-to-10 loop control variable

In our prior example, we used a loop control variable like this:

- Start value: `number = 0`
- Continuing condition: `number < 10`
- Update action: `number = number + 1`

This worked because `range` defaults to using 0 for the start and +1 for the update action. But we can use non-default values for `range` as well! For example, we could have written the loop such that we could print the number directly:

- Start value: `number = 1`
- Continuing condition: `number <= 10`
- Update action: `number = number + 1`

Range with Two Arguments

We can also give `range` two arguments, a **start** and an **end** value. The loop control variable begins with the start value, is incremented by 1 each iteration, and goes up to but not including the end value.

The following code would generate the numbers 3, 4, 5, 6, and 7.

```
for i in range(3, 8):
    print(i)
```

Range with Three Arguments

If we use three arguments in the `range` function, the last argument is the **step** of the range (how much the loop control variable should change in each iteration).

The following example would print the even numbers from 1 to 10, because it updates `i` by `2` each iteration.

```
for i in range(2, 11, 2):
    print(i)
```

Range Examples

For our original example, we could have written:

```
 for number in range(1, 11): # step is default +1
     print(number)
```

What if we wanted to count backwards instead? The loop control variable is the same, but its components change. We **start** at `10`, **update** by subtracting `1`, and **continue while** `number >= 1` (`number > 0`).

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

Activity: Print Multiple Numbers

You do: your task is to print the multiples of 5 from 50 to 100.

What is your loop control variable? What is its start value, continuing condition, and update action?

Once you've determined what these values are, use them to write a short program that does this task.

For Loops Manage the Loop Control Variable

The for loop manages the loop control variable for you entirely. That's useful, but it also means you **can't update it in the loop body**.

If you try to change the loop control variable, it will revert back to the next expected value on the following iteration. This happens because of the `range`.

```
for i in range(10):
    print(i)
    i = i + 2 # should skip two ahead, but does not
```

Loops in Algorithms

Implement Algorithms by Changing Loop Body

Suppose we want to **add** the numbers from 1 to 10 instead of printing them.

We need to keep track of two different numbers:

- the current number we're adding
- the current sum

Both numbers are represented as variables (as they represent data stored over time), but only one is a loop control variable.

```
result = 0
for num in range(1, 11):
    result = result + num
print(result)
```

Which is the loop control variable?

Tracing Loops

Sometimes it gets difficult to understand what a program is doing when that program uses loops. It can be helpful to manually trace through the values in the variables at each step of the code, including each iteration of the loop.

```
result = 0
for num in range(1, 8):
    result = result + num
print(result)
```

step	result	num
pre-loop	0	--
iteration 1	1	1
iteration 2	3	2
iteration 3	6	3
iteration 4	10	4
iteration 5	15	5
iteration 6	21	6
iteration 7	28	7
post-loop	28	7

Identifying Complicated Patterns

Figuring out what the loop control variable isn't always easy! Sometimes you need to think for a while to decide how to split up an intended result into repeated steps.

Example: how would you create a program that produces the pattern `"10-11-12-13-"`?

What is repeated? The numbers followed by dashes. `"10-"`, `"11-"`, `"12-"`, and `"13-"`. Add them each to a result one at a time.

```
s = ""
for i in range(10, 14):
    s = s + str(i) + "-"
print(s)
```

Activity: Print Number pyramid

You do: write code that takes a variable n and produces a **number pyramid** with n lines. A number pyramid of 4 lines looks like this:

```
11
2**2
3****3
4******4
```

If we added a 5th line, it would be:

```
5********5
```

Hint: start by figuring out a proper **loop control variable** for the pattern.

Bonus: if you finish early, try to figure out how to center the pyramid so it looks like:

```
    11
   2**2
  3****3
 4******4
5********5
```

Limitations of For Loops

For loops are excellent for standard patterns, and you'll use them a lot! But they don't work for every possible repetition.

What if you have a continuing condition that isn't represented by a simple comparison? What if you have an update action that isn't adding or subtracting an integer number?

In these cases, we need to use **while loops** instead!

While Loops

While Loops Repeat While a Condition is `True`

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>
```

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 `while` loop finds that the Boolean expression is `False`, it skips the loop body the same way an `if` statement would skip its body.

Conditions **Must** Eventually Become `False`

Unlike `if` statements, the condition in a `while` loop **must eventually become `False`**. If this doesn't happen, the `while` loop will keep going forever!

The best way to make the condition change from `True` to `False` is to set up a reasonable **loop control variable**. If we set up the parts of a loop control variable so that the continuing condition will eventually become `False`, the loop will eventually stop.

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
print("done")
```

Note a core difference from for loops – we must define the start value and run the update action ourselves. The while loop doesn't manage the loop control variable!

Infinite Loops Run Forever

What happens if we don't ensure that the condition eventually becomes `False`? The `while` loop will just keep looping forever! This is called 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 stop sign above the editor (or refresh the page in repl.it) 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.

You Do: Trace the Program

You do: if we slightly change the code from the previous program, what happens to the program?

```
i = 1
while i <= 10:
    i = i + 1 # moved up one line
    print(i)
print("done")
```

For Loops vs While Loops

If you can write a (numeric) loop in a for loop, you can also write it in a while loop. To sum the numbers from 0 to n in a **for** loop, we'd write the following:

```
result = 0
for i in range(n + 1):
    result = result + i

print(result)
```

To sum the numbers from 0 to n in a **while** loop, we'd instead use the following. Note the extra lines to set up and update the loop control variables

```
i = 0
result = 0
while i <= n:
    result = result + i
    i = i + 1
print(result)
```

Purpose of While Loops

There's not much point in writing a while loop that could be a for loop. But some while loops can **only** be while loops!

For example, what if we wanted to print all the powers of two from 1 to 100? We can generate all the powers by multiplying each new power by 2 again.

- Start value: `n = 1`
- Continuing condition: `n <= 100`
- Update action: `n = n * 2`

We can't set up a range with an update action that multiplies! But we can write a while loop that works that way.

Example: Powers of Two

Here's our loop that prints the powers of two:

```
n = 1
while n <= 100:
    print(n)
    n = n * 2
```

Example: Number Digits

While loops are also useful if you need to repeatedly **divide** a number.

For example – how can we tell how many digits are in an integer? Each digit represents a power of 10; repeatedly divide the integer by 10 until you can't divide it anymore (when it reaches 0).

```
count = 0
n = 2022
while n > 0:
    count = count + 1
    n = n // 10
print(count)
```

Activity: Trace a While Loop

You do: What will the following code print?

Try using a table to manually trace through the code!

```
x = 16
y = 1
num = 0
while y < x:
    num += 1
    y = y * num
print(num, y)
```

Learning Goals

Use **for loops** when reading and writing algorithms to repeat actions a specified number of times

Identify **start values, continuing conditions,** and **update actions** for **loop control variables**

Use **while loops** when reading and writing algorithms to repeat actions where the update action or continuing condition is complicated