While Loops

15-110 – Friday 09/13

Learning Goals

Use **programming** to specify algorithms to computers

- Use **while loops** to repeat actions until a certain condition is met
- Use **nesting** of statements to create complex control flow

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.

Conditions Must Eventually Become False

Unlike with if statements, we want the condition in a while loop to **change** after a certain number of iterations, from True to False. If this doesn't happen, the while loop might loop forever!

The only way to make the condition change is to use a variable as part of it. We can then change the variable inside the while loop. An example of this is shown below.

```
i = 0
while i < 5:
    print(i)
    i = i + 1
print("done")
```

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**.

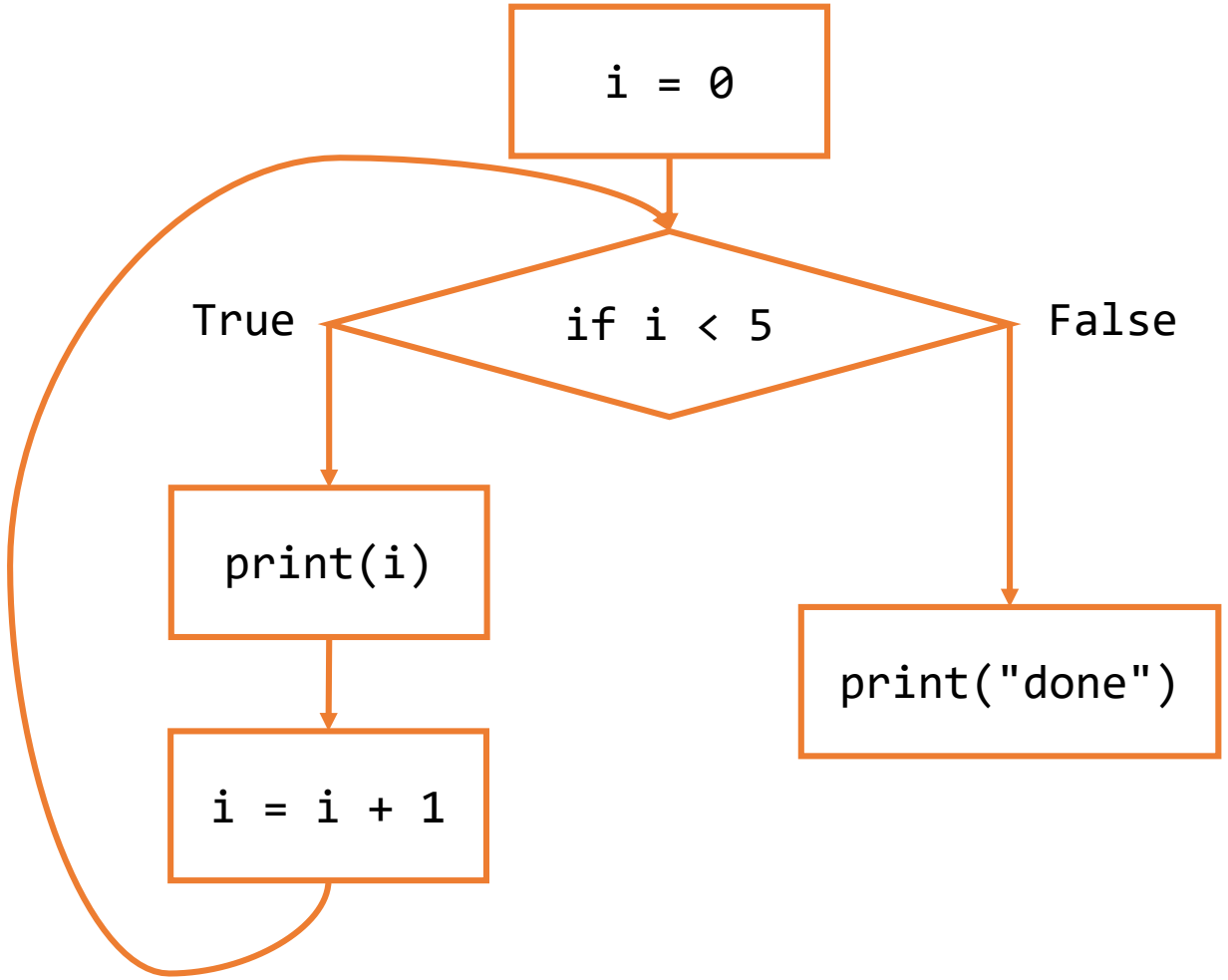
```
i = 1
while i > 0:
    print(i)
    i = i + 1
```

If you get stuck in an infinite loop, press the button that looks like a 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.

While Loop Flow Chart

To make a flow chart that runs a while loop, we need to add a transition from the while loop's body back to itself.

```
i = 0
while i < 5:
    print(i)
    i = i + 1
print("done")
```



Using Loops in Algorithms

Now that we know the basics of how loops work, we need to determine how to write a loop to produce a wanted algorithm. Usually we use loops in algorithms when we want to repeat an action.

First, we need to identify the parts of the repeated action that change in each iteration. This will become the **loop variable(s)** that is updated in the loop body.

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**.

Loop Variables - Example

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
```

Activity: print even numbers

You want to print the even numbers from 2 to 100. What is your loop variable, and what are the start/end/update values?

Identify these values, then use them to write out the code for the loop.

Loops and Algorithms – Loop Body

Setting up a loop can depend on more than just a single loop variable. For example, say we want to find the sum of the numbers from 1 to 10. How do we do this?

We need to keep track of two different numbers- the current number we're adding, and the current sum. Both numbers will need to be updated inside the loop body! However, only one (the current number) needs to be checked in the condition.

Note- when updating multiple variables in a loop, **order matters.** If we set num = num + 1 first, it will change the value held in result!

```
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

Loops and Algorithms – Loop Variables

It isn't always obvious how the start, end, and update values of a loop variable should work. Sometimes you need to think through an example to make it clear!

For example: let's say we want to simulate a zombie apocalypse. Every day, each zombie will find a human and bite them, turning them into a zombie. If we start with just one zombie, how long does it take for the whole world (7.5 billion people) to turn into zombies?

Your **start value** is 1. Your **end value** is when the number of zombies is greater than the population. And your **update value** is doubling the number of zombies every day! A separate variable can be used to count the number of days passed.

```
zombieCount = 1
population = 7.5 * 10**9
daysPassed = 0
while zombieCount < population:
    zombieCount = zombieCount * 2
    daysPassed = daysPassed + 1
print(daysPassed)
```

Nesting in While Loops

We showed previously how we can nest if statements in other if statements to combine them together. We can do the same thing with while loops!

For example, let's say we want to make ascii art. Specifically, let's write code to produce the following printed string:

```
x-x-x
-o-o-
x-x-x
-o-o-
x-x-x
```

We can make a loop that iterates over the row we're printing. We can decide whether to print the x line or the o line based on **the value of the loop variable**. If it's even (0, 2, and 4) we'll print x; if it's odd (1 and 3) we'll print o.

```
row = 0
while row < 5:
    if row % 2 == 0:
        print("x-x-x")
    else:
        print("-o-o-")
    row = row + 1
```

Learning Goals

Use **programming** to specify algorithms to computers

- Use **while loops** to repeat actions until a certain condition is met
- Use **nesting** of statements to create complex control flow