While Loops

15-110 – Friday 01/31

Learning Goals

- Use **while loops** when reading and writing algorithms to repeat actions while a certain condition is met

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

- Translate algorithms from **control flow charts** to Python code

- Use **nesting** of statements to create complex control flow

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.

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 <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 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 use a variable as part of the Boolean expression. We can then change the variable inside the while loop. For example, the variable `i` changes in the loop 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**.

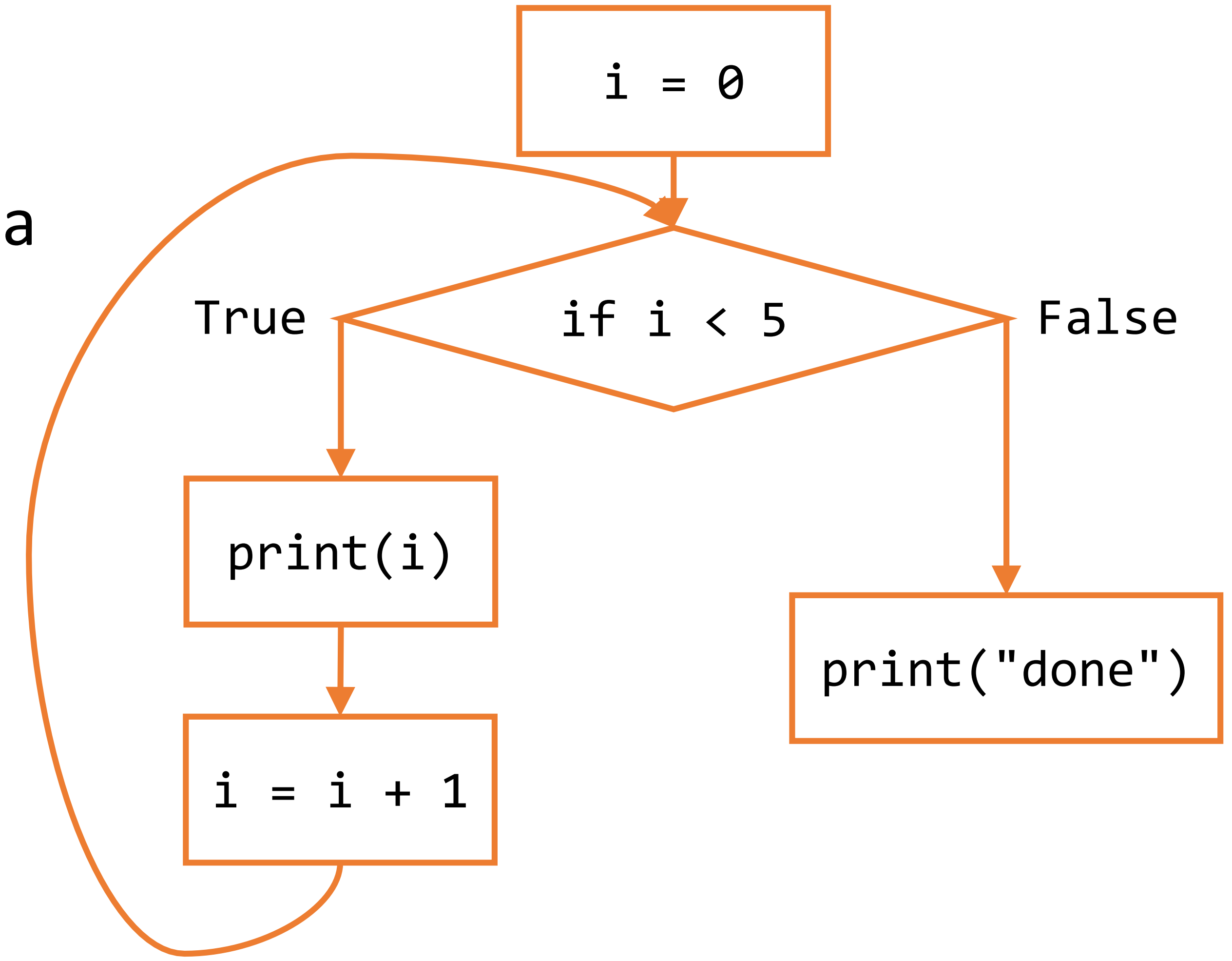
```
i = 1
while i > 0:
    print(i)
    i = i + 1
```

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 variable that changes can help pinpoint the issue.

While Loop Flow Chart

Unlike an if statement, a while loop flow chart needs to include a transition from the while loop's body back to itself.

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



Use Loop Control Variables to Design Algorithms

Now that we know the basics of how loops work, we need to write while 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.

To use this loop variable, we'll need to give it a **start value**, an **update action**, and a **continuing condition**. All three need to be coordinated for the loop to work correctly.

Loop Control Variables - Example

In our print 1-to-10 example, we want to **start** the variable at 1, and **continue while** the variable is less than or equal to 10. Set `num = 1` at the beginning of the loop and continue looping while `num <= 10`. The loop ends when `num` is 11.

Each printed number is one larger from the previous, so the **update** should set the variable to the next number (`num = num + 1`) in each iteration.

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

Activity: Print Even Numbers

You do: your task is to print the even numbers from 2 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.

Submit your start/continue/update values and your program when you're done.

Implement Algorithms by Changing Loop Body

Suppose we want to add the numbers from 1 to 10.

We need to keep track of two different numbers:

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

Both numbers need to be updated inside the loop body, but only one (the current number) needs to be checked in the condition.

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

Which is the loop control variable?

Tracing Loops

Sometimes it gets difficult to understand what a program is doing when there are 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
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

Update Order Matters

When updating multiple variables in a loop, **order matters.** If we update `num` before we update `result`, it changes the value held in `result`.

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

Note: Python checks the condition only at the start of the loop; it doesn't exit the loop as soon as `num` becomes 11.

step	result	num
pre-loop	0	1
iteration 1	2	2
iteration 2	5	3
iteration 3	9	4
iteration 4	14	5
iteration 5	20	6
iteration 6	27	7
iteration 7	35	8
iteration 8	44	9
iteration 9	54	10
iteration 10	65	11
post-loop	65	11

Loop Control Variables – Advanced Example

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

Example: how would you count the number of digits in an integer?

Loop control variable: the number itself

Start value: the original value

Continuing condition: while number is not 0

Update action: integer-divide the number by 10

A separate variable can track the actual number of digits counted.

```
num = 2020
digits = 0
while num > 0:
    digits = digits + 1
    num = num // 10
print(digits)
```

Loop Variables – Advanced Example

Another example: simulate a zombie apocalypse. Every day, each zombie finds and bites a human, 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?

Loop control variable: # of zombies
Start value: 1
Continuing condition: while the number of zombies is less than the population
Update action: double the number of zombies every day

We use a separate variable to count the number of days passed, as that's our output.

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

Nesting in While Loops

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

For example, let's make ascii art. Write code to produce the following printed string:

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

The loop will iterate over the rows that are printed. The program decides 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) print x; if it's odd (1 and 3) 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 **while loops** when reading and writing algorithms to repeat actions while a certain condition is met

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

- Translate algorithms from **control flow charts** to Python code

- Use **nesting** of statements to create complex control flow