For Loops

15-110 – Monday 09/21

Learning Goals

- Use **for loops** when reading and writing algorithms to repeat actions a specified number of times
- Recognize which numbers will be produced by a **range** expression
- Translate algorithms from **control flow charts** to Python code
- Use **nesting** of statements to create complex control flow

For Loops

For Loops Implement Repeated Actions

We've learned how to use while loops and loop control variables to iterate until a certain condition is met. When that loop control is straightforward (increase a number until it reaches a certain limit), we can use a more standardized structure instead.

A **for loop** over a **range** tells the program exactly how many times to repeat an action. The loop control variable is updated by the loop itself!

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

Do This Five Times

```
for i in range(5):
    print("I really like Python.")
```

```
I really like Python.
I really like Python.
I really like Python.
I really like Python.
I really like Python.
```

range(n) generates numbers from 0..n-1

```
for i in range(5):
    print("Now i is", i)
print("Finally i is", i)

Now i is 0
Now i is 1
Now i is 2
Now i is 3
Now i is 4
Finally i is 4
```

While Loops vs. For Loops

To sum the numbers from 0 to n in a **while** loop, we'd write the following:

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

In a **for** loop using a range expression, the loop control variable starts at 0 and automatically increases by 1 each loop iteration.

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

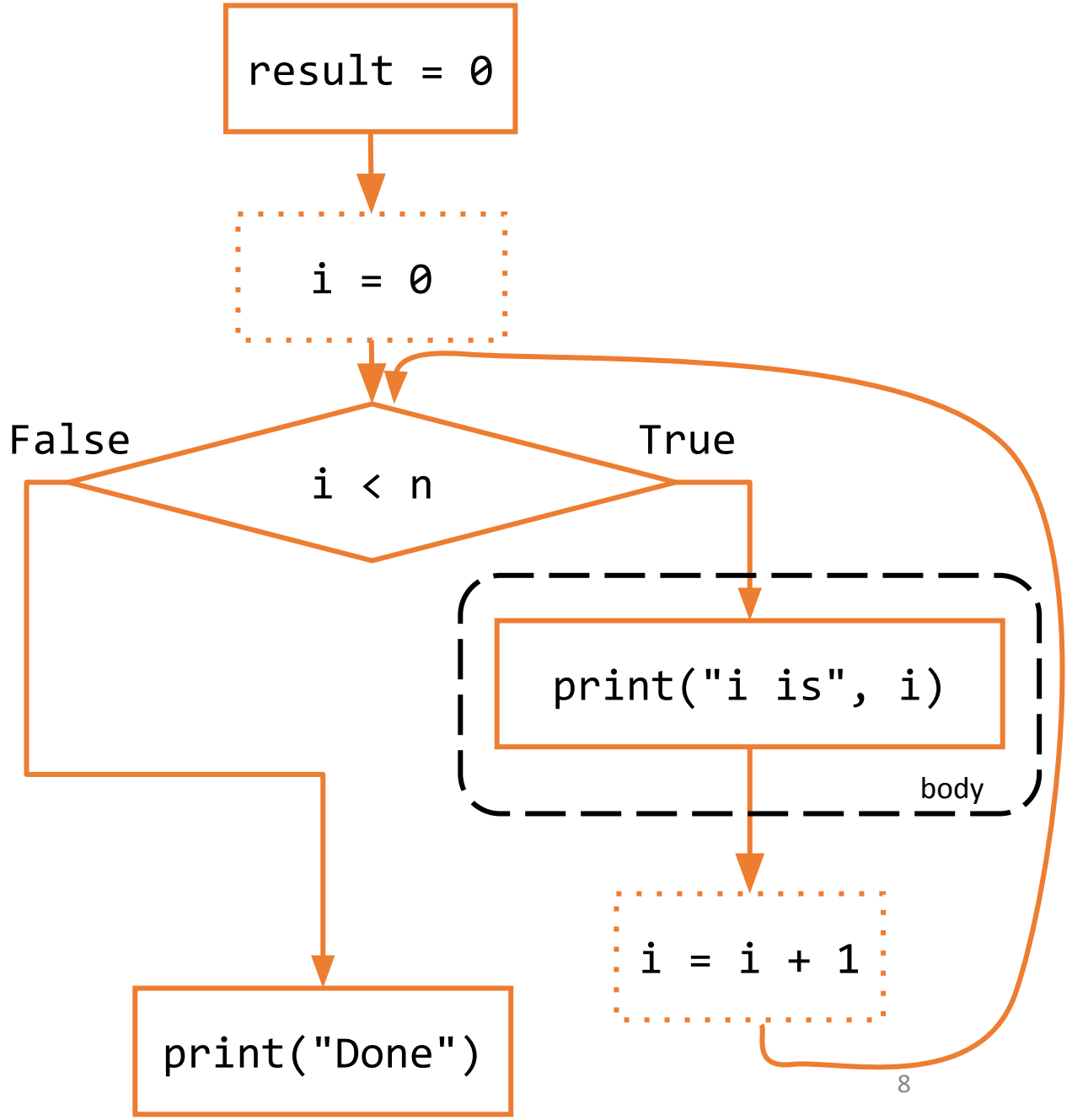
We have to specify `n + 1` because range goes **up to but not including** the given number.

For Loop Flow Chart

Unlike **while** loops, we don't initialize or update the loop control variable. The **for** loop will do that automatically.

We show actions done by the for loop with a dotted outline here, because they're **implicit**, not written directly.

```
for i in range(n):
    print("i is", i)
print("Done")
```

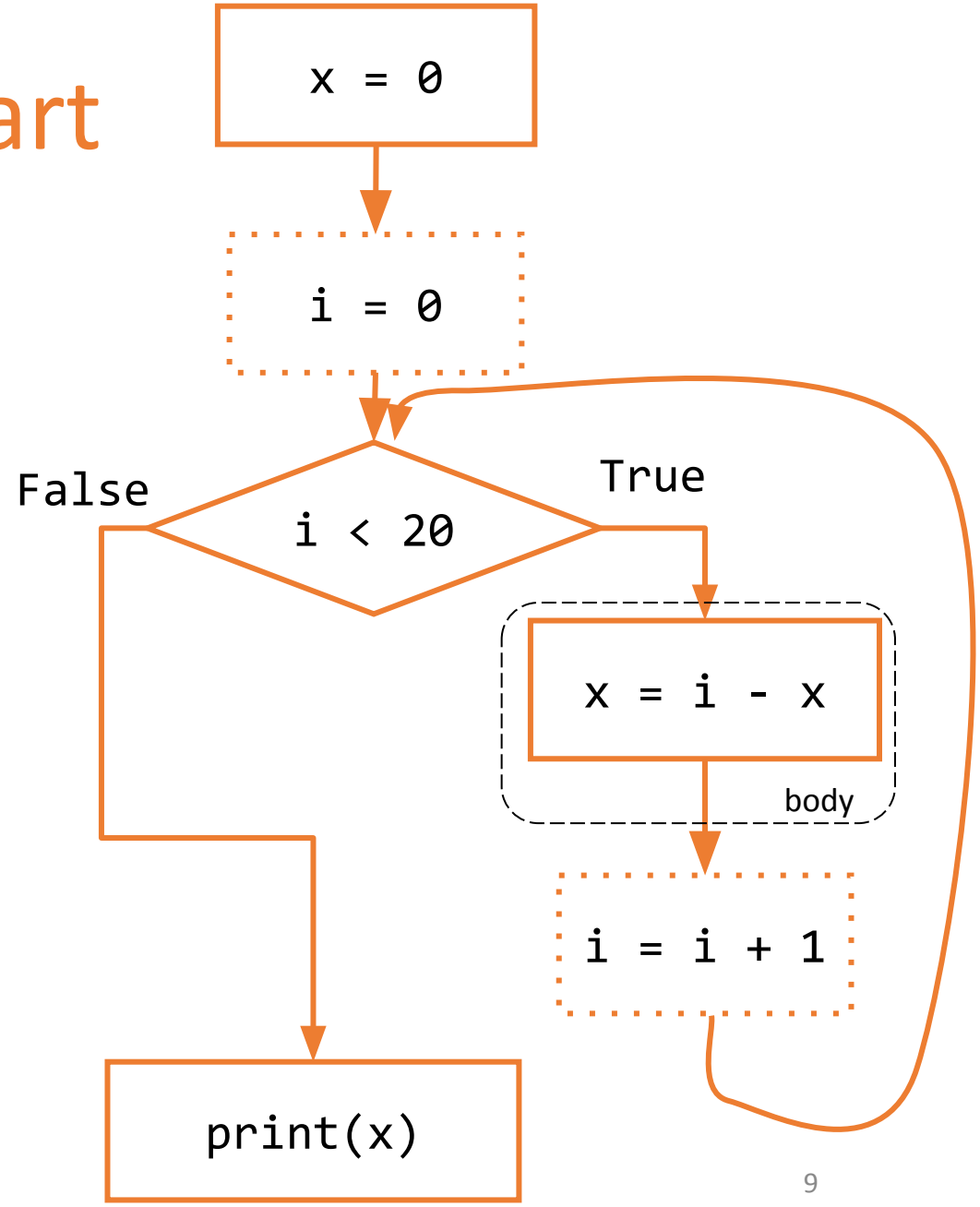


Note: this flow chart translation is not quite accurate because after the **for** loop, i will be equal to n-1, not n. After the **while** loop i will be equal to n.

Activity: Translate Flow Chart

You do: given the flow chart to the right, write a program that matches the flow chart. Use a **for** loop, not a **while** loop.

What does the program print?



Range

`range()` Generates Values

When we run `for i in range(10)`, `range(10)` generates the consecutive values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, one value each iteration.

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() Also Has a Step

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)
```

Anything we can do in a for loop can also be done in a while loop. In a while loop, the above would be (almost) equivalent to:

```
i = 2
while i < 11:
    print(i)
    i = i + 2
```

What will be the value of i after the **for** loop?
What will be the value of i after the **while** loop?

`range()` Example: Countdown

Let's write a program that counts backwards from 10 to 1, using `range()`.

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

Note that the range has to end at `0` in order to make `1` the last number that is generated and printed.

Activity: Predict the Printed Values

In this Kahoot quiz, predict what the loop will print based on its `range()`.

Link: https://kahoot.it/

Link for asynchronous Kahoot: https://kahoot.it/challenge/06107185?challenge-id=a750a494-3baa-4c36-81d2-898b6309e430_1600722536420

Problem Solving with For Loops

Problem solving with `for` loops is similar to problem solving with `while` loops. You need to identify the loop control variable, then find the correct start, end, and step for it.

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

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

Nested Loops

Nesting Loops

We've already used nesting to put conditionals in other conditionals and in loops. More 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: Coordinate Plane with Nested Loops

Suppose we want to print all the coordinates on a plane from (0,0) to (4,3).

```
for x in range(5):
    for y in range(4):
        print("(", x, ",", y, ")")
```

Note that every iteration of y happens anew in each iteration of x.

Tracing Nested Loops

The following code prints out a 4x3 multiplication table. We can use **code tracing** to find the values at each iteration of the loops.

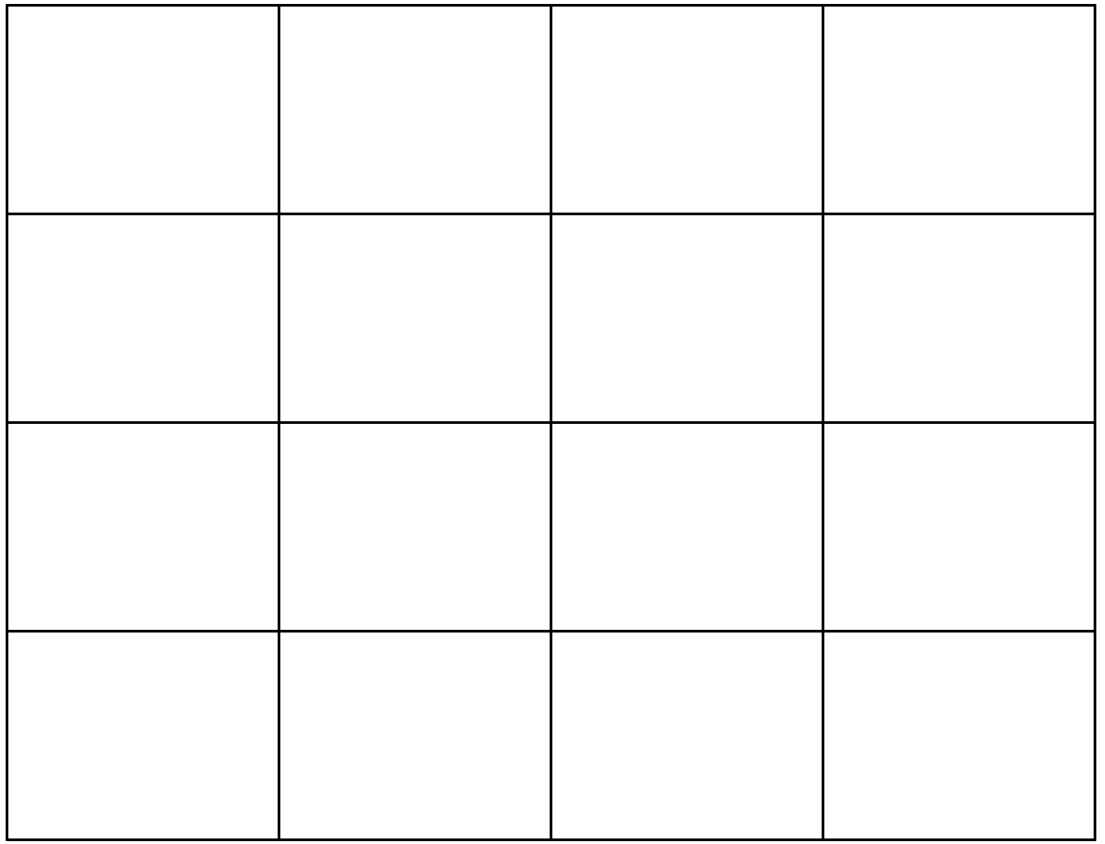
```
for x in range(1, 5):
  for y in range(1, 4):
    print(x, "*", y, "=", x * y)
```

Iteration	x	y	x*y
1	1	1	1
2	1	2	2
3	1	3	3
4	2	1	2
5	2	2	4
6	2	3	6
7	3	1	3
8	3	2	6
9	3	3	9
10	4	1	4
11	4	2	8
12	4	3	12

Example: `drawGrid(canvas, size)`

Let's write a function that draws a grid using Tkinter.

Instead of repeating calls of `create_rectangle`, we'll use **nested for loops** (along with math and logic) to determine where to draw each square.



First, Draw a Row

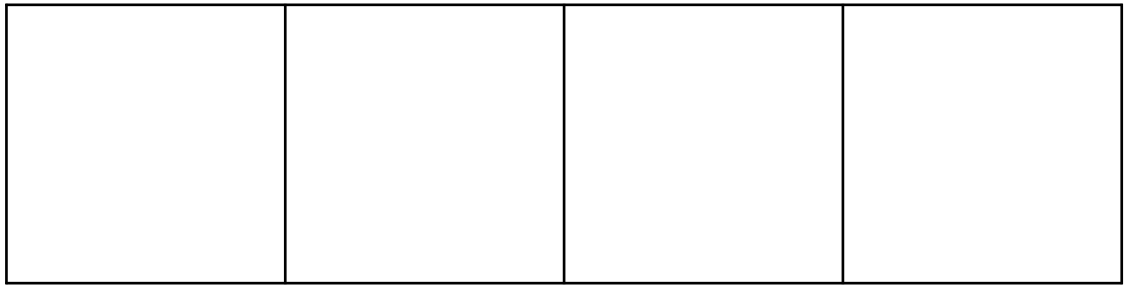
Let's start simple, by drawing a row of cells instead of a whole grid. Note that a row **repeats** cells over the X axis.

Loop over all possible columns from 0 to size. We'll then draw a square for each.

Each square's top and bottom will be 0 and 50.

You do: How can we calculate a square's left and right positions, using only its column number?

Desired outcome:



Loop Over Columns

The first square starts at x coordinate 0; the next is one square over, so it starts at 50. The third square has two squares before it, so it starts at 2 * 50; etc..

If we number the squares from 0 to 4, each square's left side starts at `col * 50`, where 50 is the size of the square. Add 50 to that coordinate to get the right side.

```
def drawGrid(canvas, size):
  for col in range(size):
    left = col * 50
    right = left + 50
    canvas.create_rectangle(left, 0,
                            right, 50)
```

Draw Multiple Rows for a Grid

Now we just need to repeat the logic that drew the first row. Take the code from before and put it inside an outer loop. Note that the outer loop represents a cell's **row**, while the inner loop represents a cell's **column**.

Calculate the top of each cell based on the value's row, using the same logic that found the column coordinates.

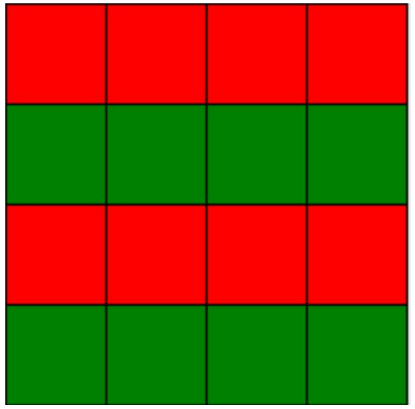
```
def drawGrid(canvas, size):
  for row in range(size):
    top = row * 50
    bottom = top + 50
    for col in range(size):
      left = col * 50
      right = left + 50
      canvas.create_rectangle(left,
                              top,
                              right,
                              bottom)
```

Add Stripes with Conditionals

We can make the grid more exciting by adding colors to the cells, to draw stripes.

Stripes alternate by **row** or by **column**. Check whether the row/column is **odd** or **even** using the mod operator.

```
if row % 2 == 0:
  color = "red"
else:
  color = "green"
canvas.create_rectangle(left, top,
                        right, bottom,
                        fill=color)
```



Learning Goals

- Use **for loops** when reading and writing algorithms to repeat actions a specified number of times
- Recognize which numbers will be produced by a **range** expression
- Translate algorithms from **control flow charts** to Python code
- Use **nesting** of statements to create complex control flow
- **Feedback: https://bit.ly/110-feedback**