For Loops

15-110 – Monday 09/15

Announcements

- Check2 was due today
- Week1-2 revision deadline is **tomorrow (09/16) at noon**
- **Start Hw2 early**! It's heavier than Hw1
  - On the assignment you'll start **combining** code constructs you've learned about before, so your solutions will be more algorithmically complex.
- You're encouraged to attend **small group sessions** to get help with learning the course content. In particular, the TAs will provide **more help than usual** on one of the Hw2 problems in small group sessions this week (`drawIllusion`). Contact your TA to learn more!
- Quizlet2 on Wednesday

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/decrease 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>
```

Example: Print 1 to 10

Previously we showed how to print the numbers from 1 to 10 with a while loop. Doing this with a for loop is super easy! The loop control variable starts at 0 and automatically increases by 1 each loop iteration.

```
for i in range(10):
    print(i+1) # starts at 0 and ends at 9, so add 1
```

While Loops vs. For Loops

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

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

In a **for** loop using a range expression, we'd write the following:

```
n = 10
result = 0
for i in range(n + 1):
    result = result + i
print(result)
```

We have to use `n + 1` because `range` goes **up to but not including** the given number. It's like writing

```
while i < n + 1:
```

Range

`range` Manages the Loop Control Variable

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

Because range generates numbers this way, you **can't update the loop control variable in the loop body**.

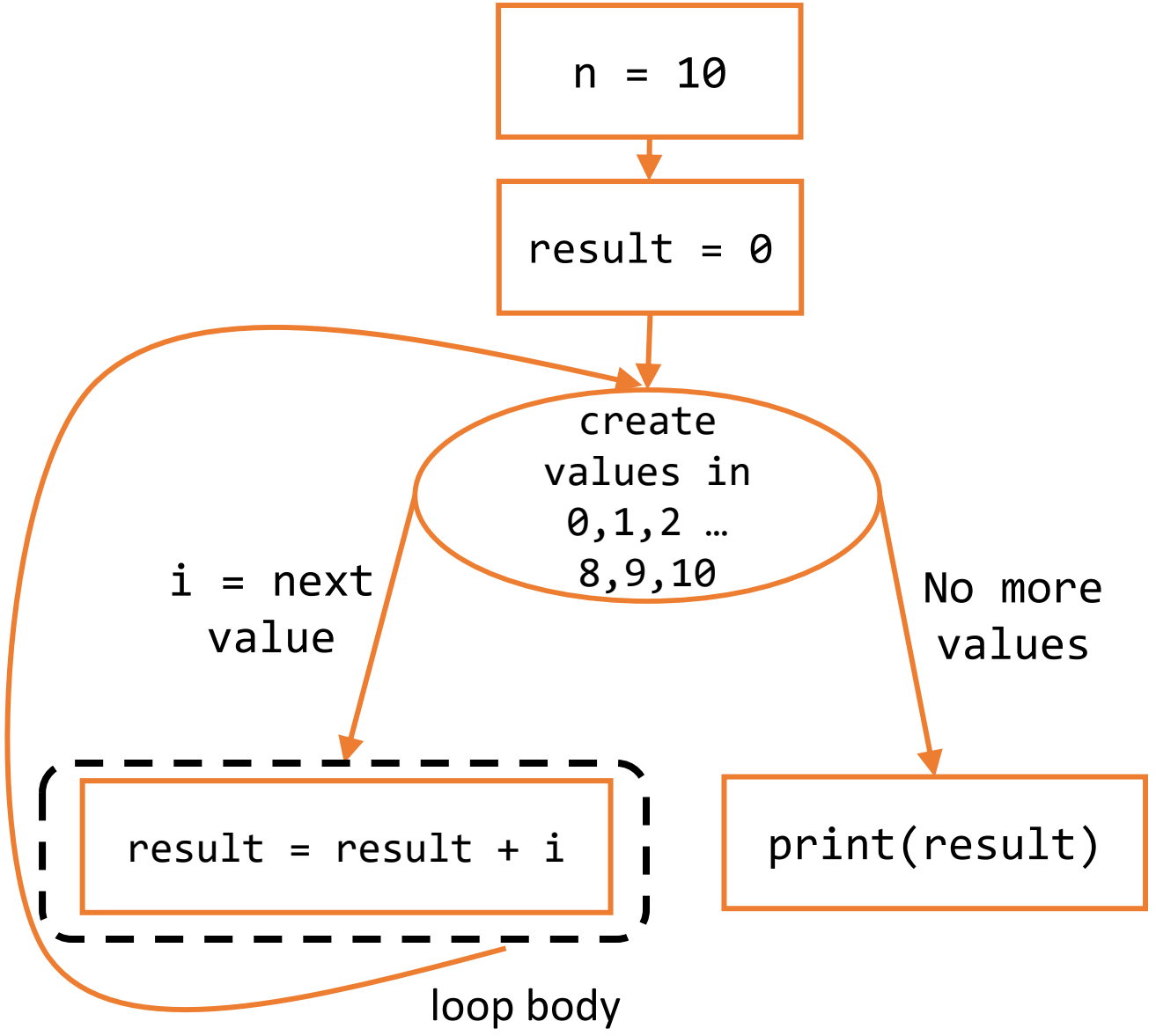
If you try to change the loop control variable, it will revert back to the next expected value on the following iteration.

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

For Loop Flow Chart

Unlike while loops, we don't initialize or update the loop control variable. `range` covers that part instead.

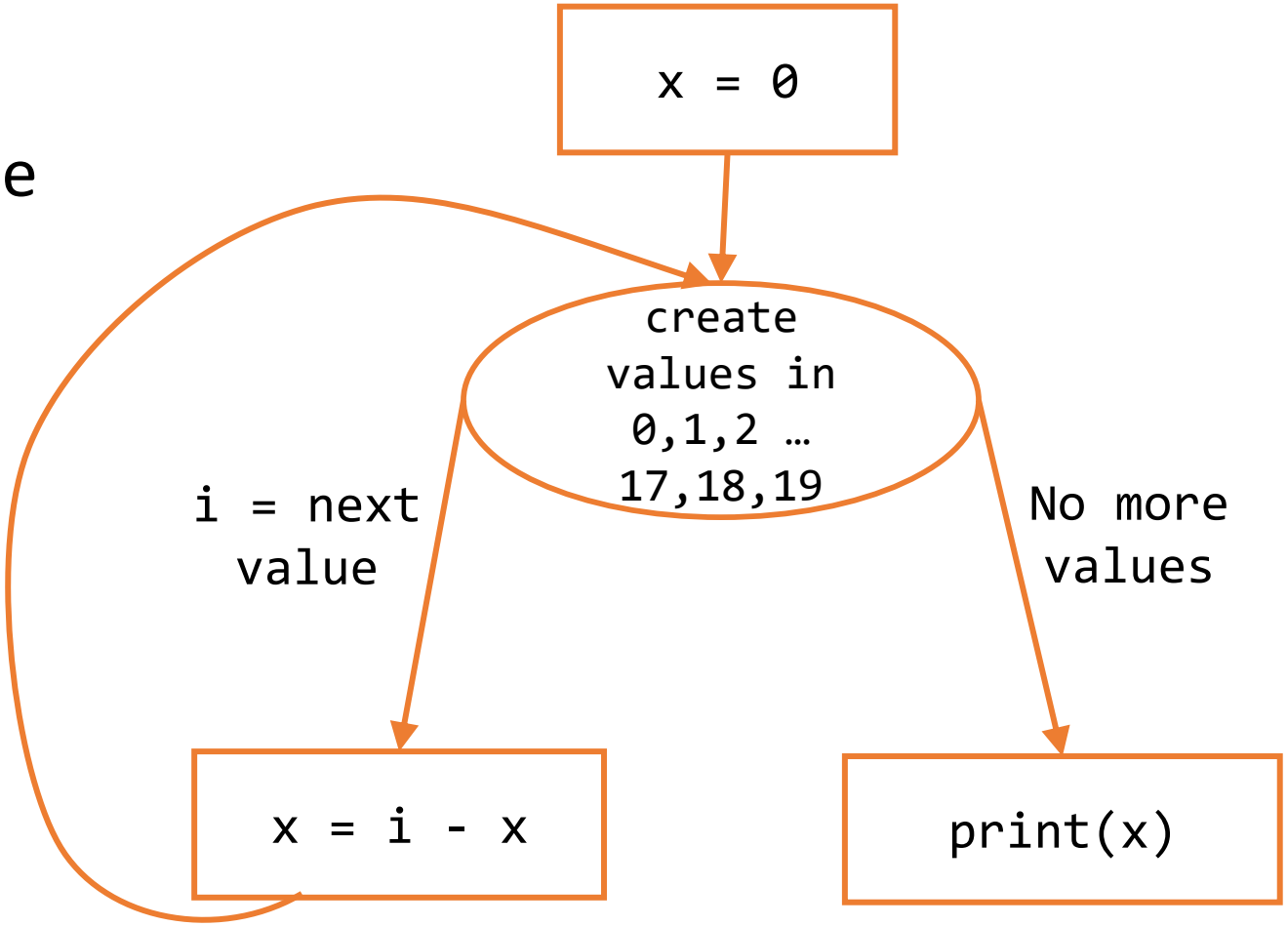
```
n = 10
result = 0
for i in range(n + 1):
    result = result + i
print(result)
```



Activity: Translate the 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` Arguments: start, end, step

`range` can be used with up to three arguments: start, end, and step. As we've seen before, if you use only one argument, it's the **end**.

end

```
for i in range(10):
    print(i)
```

When range is given two arguments, the first is the **start** and the second is the end. Now the loop begins at the start instead of `0`. The second loop will print 3, 4, 5, 6, 7

start, end

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

If we use three arguments, the last argument is the **step** of the range (how much the loop control variable should change in each iteration). The third loop prints the odd numbers from 1 to 11, because it updates `i` by `2` each iteration instead of `1`.

start, end, step

```
for i in range(1, 12, 2):
    print(i)
```

While Loops Work Too

```
for i in range(1, 12, 2):
    print(i)
```

Any looping over numbers we can do in a for loop can also be done in a while loop. In a while loop, the above code could be written as:

```
i = 1
while i < 12:
    print(i)
    i = i + 2
```

`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 `i` has to end at `0` in order to make `1` the last number that is printed.

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

Coding with For Loops

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)
```

Nesting with For Loops

We can also nest for loops in functions and conditionals in for loops, just like with while loops.

For example, we can determine whether or not a number is prime using a for loop over all the number's possible factors (from 2 up to but not including the number itself).

Make sure to also check that the number is positive and not 1!

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

Nested Loops

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 **code tracing** 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)
```

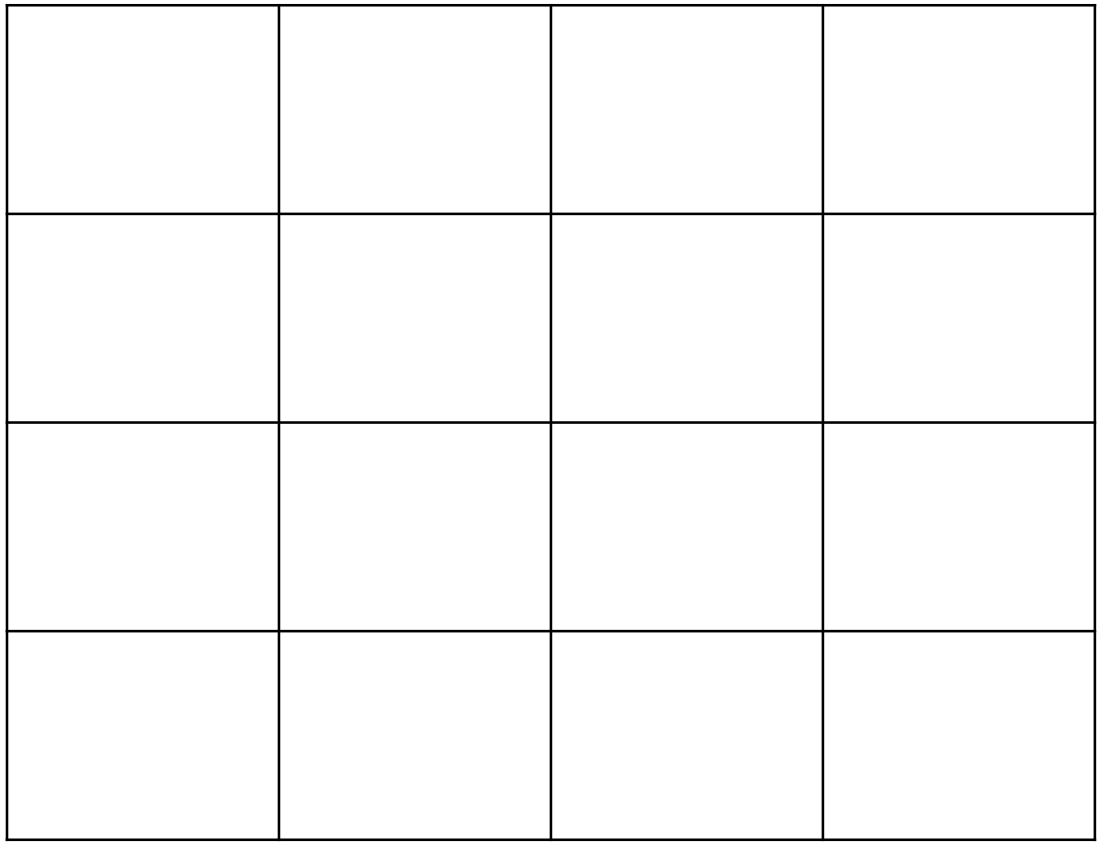
Iteration	x	y	x*y
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	4
5	3	1	3
6	3	2	6

Example: `drawGrid(canvas, gridSize)`

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

If we call `drawGrid(canvas, 4)`, we create a 4x4 grid.

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.



Sidebar: Function Call Canvas

Let's use a bit of code to generate a new canvas in a function call.

We just need to add in our own call to our drawing function in the middle!

```
def drawGrid(canvas, gridSize):
    pass # add code here
```

```
import tkinter
def runDrawGrid():
  root = tkinter.Tk()
  canvas = tkinter.Canvas(root, width=400,
                          height=400)
  canvas.configure(bd=0,
                   highlightthickness=0)
  canvas.pack()

  drawGrid(canvas, 4) # your call here!

  root.mainloop()
```

First, Draw a Row

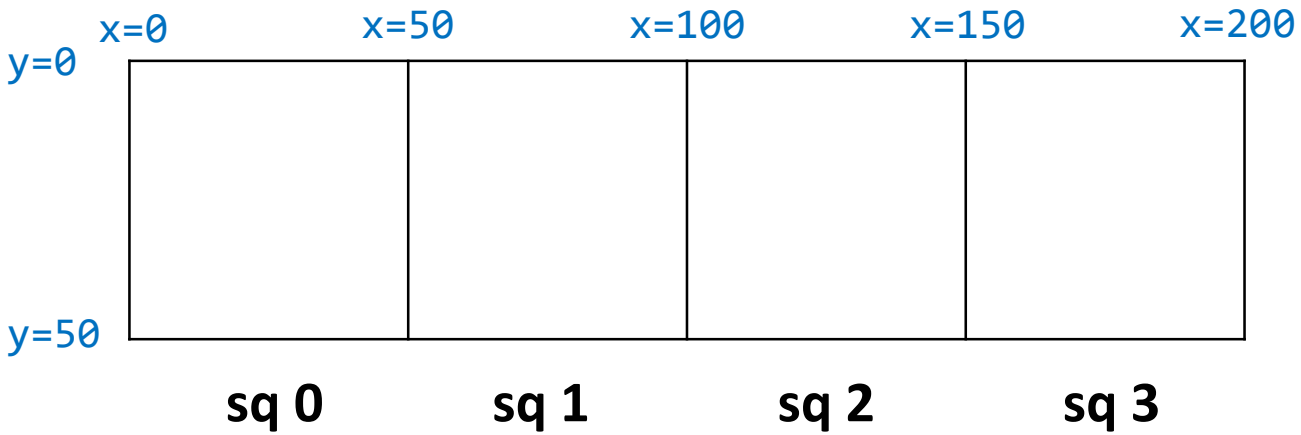
Let's start simple by drawing a row of cells instead of a whole grid. Note that a row **repeats** cells over the X axis. Each square will be 50 x 50 pixels in size.

Each square's top and bottom will be 0 and 50. The first square's left and right are 0 and 50, second are 50 and 100, etc.

We'll want to loop over all possible columns from 0 to `gridSize-1`. We'll then draw a square for each.

Discuss: How can we calculate a square's left and right positions **abstractly** 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 3, 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, gridSize):
  for col in range(gridSize):
    leftSq = col * 50
    rightSq = leftSq + 50
    canvas.create_rectangle(leftSq, 0,
                            rightSq, 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.

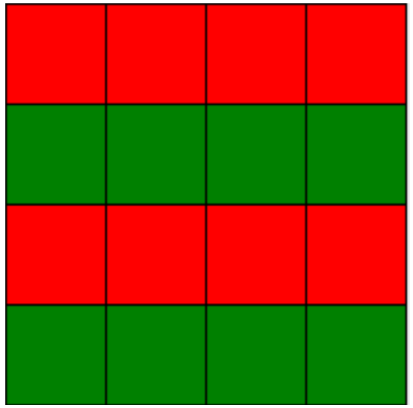
```
def drawGrid(canvas, gridSize):
    for row in range(gridSize):
        topSq = row * 50
        bottomSq = topSq + 50
        for col in range(gridSize):
            leftSq = col * 50
            rightSq = leftSq + 50
            canvas.create_rectangle(leftSq,
                                    topSq,
                                    rightSq,
                                    bottomSq)
```

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(leftSq, topSq,
                        rightSq, bottomSq,
                        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