#3-1: Nesting and Top-Down Design

CS SCHOLARS – PROGRAMMING

Learning Goals

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

Implement and use **helper functions** in code to break up large problems into solvable subtasks

Nesting Control Structures

Nesting Control Structures

We showed previously how to nest conditionals inside conditionals, and we've already nested both conditionals and loops inside functions.

We can use this same approach to nest loops in loops, or conditionals in loops, or loops in conditionals! We can also nest multiple times when needed, like putting a conditional in a loop in a function definition.

By **composing** these control structures in different ways, we can create new and complex behavior and can implement any algorithm we want!

Combining Conditionals and Loops

First, you may want to use conditionals in loops to **change behavior** in different iterations based on some property.

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 control variable**.

If it's even (0, 2, and 4) print x; if it's odd (1 and 3) print o.

```
for row in range(5):
    if row % 2 == 0:
        print("x-x-x")
    else:
        print("-o-o-")
```

Exiting Loops Early

We can also use conditionals in loops to **exit loops early**, which can be helpful in cases when you aren't sure when a loop should end.

If we **return** inside a loop, it immediately exits the function- no further iterations will run.

For example, we can determine whether or not a number is prime using a loop over all of the number's possible factors. Exit early if you find a factor.

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
```

Activity: countFactors

You do: write a program `countFactors` that takes a number `x` and returns the number of unique factors between [1, x] that the number has.

Hint*:* you can start with the template from `isPrime`. What needs to change so that you can count factors instead of checking if there's at least one?

Nesting Loops

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 3x2 multiplication table. 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

Activity: Trace the Nested Loop

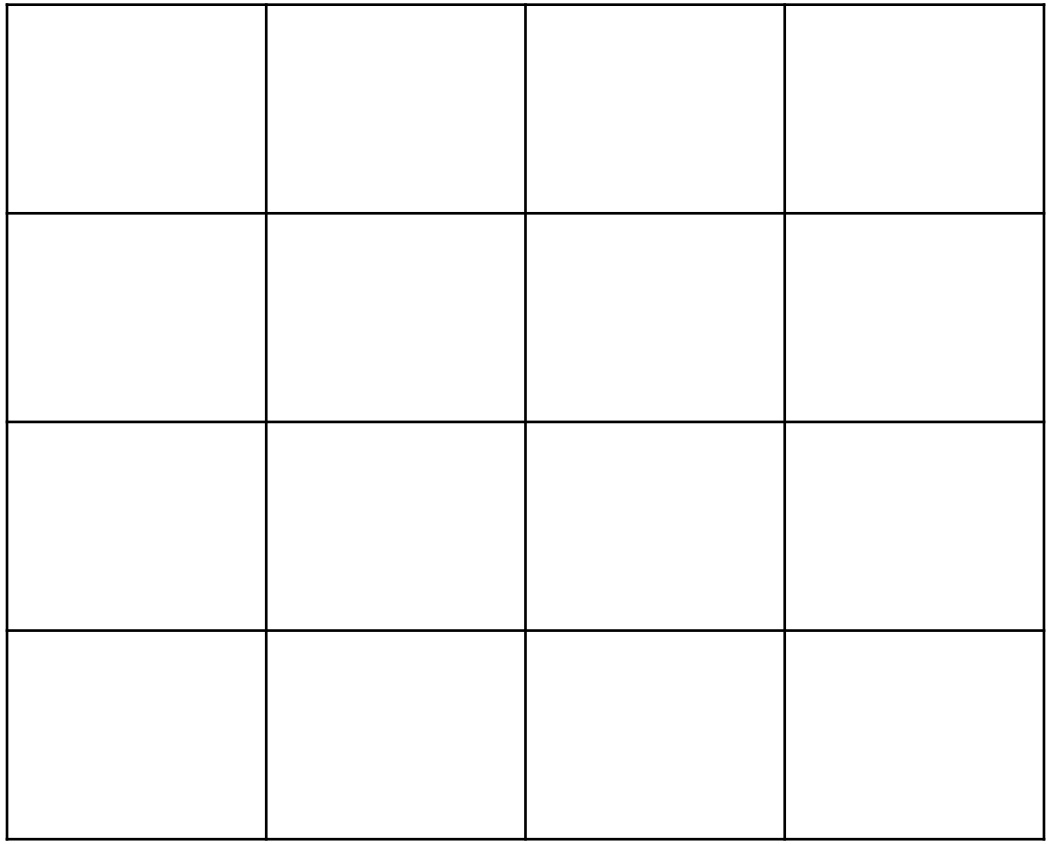
You do: what will the following loop print? Try using a table to keep track of the two loop control variables, then for each pair determine whether or not it meets the condition.

```
for x in range(2, 6):
    for y in range(10, 15):
        if y % x == 0:
            print(x, "divides", y)
```

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 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!

```
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

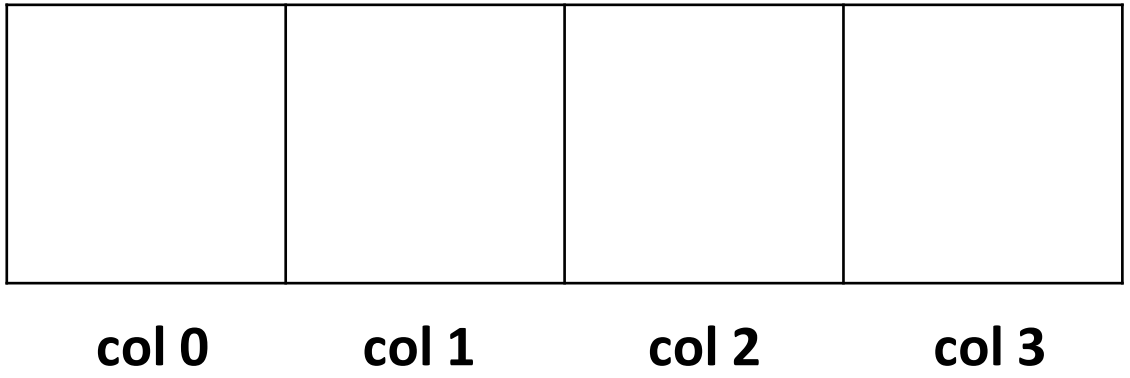
Let's start simple by drawing a row of cells instead of a whole grid. Note that a row **repeats** cells over the X axis. Each square will be 50 x 50 pixels in size.

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

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

Discuss: 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.

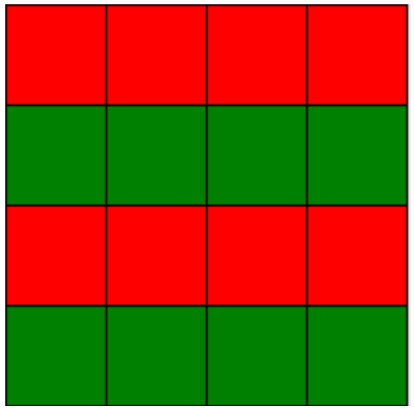
```
def drawGrid(canvas, size):
    for row in range(size):
        top = row * 50
        bottom = top + 50
        for col in range(size):
            left = col * 50
            right = left + 50
            canvas.create_rectangle(left, top,
                                    right, bottom)
```

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)
```



Activity: Vertical Stripes

You do: update the `drawGrid` code we just wrote to draw **three columns** of stripes instead of **two rows**.

What needs to change?

Top-Down Design

Helper Functions

As you start creating more complex programs, you'll often need to write many small functions that work together to solve a larger problem. We call a function that solves part of a larger problem this way a **helper function.**

By breaking up a large problem into multiple smaller problems and solving those problems with helper functions, we can make complicated tasks more approachable. This is called **top-down design**.

Designing Helper Functions

How can you determine which helper functions are needed to solve a problem?

Try to identify **subtasks** that are repeated or are separate from the main goal; break down the problem into smaller parts. Have **one subtask per function** to keep things simple.

Example: Achilles Number Sequence

Let's work through an example of top-down design as a class by writing a program to print the first n numbers of the Achilles number sequence.

An Achilles number is a number that is **powerful**, but not a **perfect power**. This is named after the ancient Greek hero Achilles, who was also very powerful but not perfect!

Definitions

Powerful: a number where for every prime number x that divides it, x^2 also divides it.

- 36 is powerful. Prime factors: 2 and 3. 4 and 9 both divide 36.
- 2700 is powerful. Prime factors: 2, 3 and 5. 4, 9, and 25 all evenly divide 2700.
- 10 is not powerful. Prime factors: 2 and 5. 4 and 25 do **not** evenly divide it.

Perfect power: a number that can be represented as an integer raised to some power greater than 1.

- 36 is a perfect power. It can be represented as 6^2.
- 2700 is not a perfect power. It is factored into $2^2 * 3^3 * 5^2$; $30^2 * 3$ is not perfect.
- 10 is not a perfect power. It isn't even powerful!

Achilles Sequence Subtasks

First, let's determine the subtasks we need to solve to generate the first n Achilles numbers.

You do: what smaller tasks could be useful here?

Possible Subtasks

Here's one way to break the problem down into subtasks:

`generateAchillesNumbers(n)` – generates the first `n` numbers that are powerful but not perfect powers.

`isPowerful(x)` – determines whether `x` is powerful

`isPerfectPower(x)` – determines whether `x` is a perfect power

`isPrime(x)` – determines whether `x` is prime

generateAchillesNumbers

Let's start with our main function, `generateAchillesNumbers`. We know that we want to generate n Achilles numbers. There's no systematic way to generate these numbers, so let's loop through **all** numbers until we've found n that meet the criteria.

Use the `isPowerful` and `isPerfectPower` helper functions we've planned to write in order to simplify the program. We don't have to worry about **how** we'll determine if the numbers are Achilles numbers – the functions will do that for us!

```
def generateAchillesNumbers(n):
    count = 0
    num = 1
    while count < n:
        if isPowerful(num) and \
           (not isPerfectPower(num)):
            count += 1
            print(num)
        num += 1
```

isPowerful

Now we need to implement the helper functions so we can actually run the main function, starting with `isPowerful`.

A number is powerful if for every prime number x that divides it, x^2 also divides it. Find every prime factor of the number (using mod and the `isPrime` function we wrote earlier), and check if the square of that number divides it too.

If **any** prime factor's square doesn't divide it, we can return `False` right away. If **all** of the prime factors obey the rules, return `True` at the end.

```
def isPowerful(x):
    for factor in range(2, x+1):
        if x % factor == 0 and \
           isPrime(factor):
            if x % (factor**2) != 0:
                return False
    return True
```

isPerfectPower

Now we can implement the second helper function, `isPerfectPower`.

A number is a perfect power if it can be represented as an integer raised to some power greater than 1. How can we check this? Maybe try taking the square root, then the cube root, etc, and see whether each result creates an integer.

When can we stop? When the root produced rounds to 1, we know the number is not a perfect power, because 1 raised to any power is still 1.

```
import math
def isPerfectPower(x):
    pow = 2
    root = x ** (1/pow)
    while round(root) > 1:
        if math.isclose(root, round(root)):
            return True
        pow += 1
        root = x ** (1/pow)
    return False
```

Put It All Together

Once we've implemented all the needed helper functions, we can run `generateAchillesNumbers`, and it should work!

`generateAchillesNumbers` does just one thing, but relies on `isPowerful` and `isPerfectPower` to work. Each of those helpers does just one thing, but they each rely on other functions (`isPrime` and `math.isclose`) to work.

By using top-down-design, we can produce complex behavior out of a set of simple functions.

Learning Goals

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

Implement and use **helper functions** in code to break up large problems into solvable subtasks