Loops

Learning Objectives

- Learn how to repeat some action for each element of a sequence using a `for` loop
- Become familiar with certain patterns of processing sequences:
  - aggregating,
  - mapping,
  - & filtering
- Learn how to repeat some action while a condition holds using a `while` loop
- Identify cases when a `for` loop is more appropriate than a `while` loop and vice versa

For Loops & Sequences

Counting Up

Can you write a program that counts up from 1 to 100?

```
1
2
3
4
5
6
7
...
100
```

Yes, but Slowly

Right now, our best bet would be to do it manually…

```
print(1)
print(2)
print(3)
print(4)
...
print(100)
```

😬😬😬

Yes, but Not Really

"Counting from 1 to 100" ➡️ "Printing all numbers in the **range** from 1 to 100"

- Sounds like a place where a `range` might come in handy!
- Remember: `range(start, stop)` creates a sequence of numbers between `[start, stop)`
- Not so easy to print, though...

```python
numbers = range(1, 101)    # stop at 101 so that 100 is the last number included.
print(numbers)
```

🖨️ 👇

```
range(1, 101)
```

Oookay...

Printing Values in a Range

If `range(1, 101)` has all of the values, we could actually get them one-by-one using indices:

```
numbers = range(1, 101)   # stop at 101 so that 100 is the last number included.
print(numbers[0])
print(numbers[1])
print(numbers[2])
print(numbers[3])
```

🖨️👇

```
1
2
3
4
```

But now the program is one line longer than our first solution!

Printing All Members of a Range

Python provides a way of proceeding through all members of a sequence **in order:** the `for` loop.

```
numbers = range(1, 101)
for number in numbers:
  print(number)
```

🖨️👇

```
1
2
3
4
...
100
```

Success! And in three lines.

The `for` Loop

A `for` loop allows you to write a block of code that is executed **once per element** in an **iterable.**

- For now, think **iterable** ≈ **sequence**
- "Plucks out" elements in sequence order, one-by-one, and gives each a variable name
    - We call this "iterating over" elements of the sequence
- The code block executed each time can be written in terms of this variable name

Syntax of `for`

```
for element in sequence:
  do_something()
  do_something_else()
```

- `sequence` is the name of the sequence that we're iterating over
- `element` is the name of a variable that stores each value from the sequence
  - If `element` is not already declared, it will be declared here
  - `element` will remain "in scope" (available) even after the loop
- The first time we execute the body of the loop, `element == sequence[0]`.
  - The next time, `element == sequence[1]`
  - The next time, `element == sequence[2]`
  - and so on

Unravelling a `for` Loop

A shorter version of counting to 100:

```python
count_off = range(1, 4) # contains 1, 2, 3
for number in count_off:
  print(number)
```

🖨️👇

```
1
2
3
```

We can write an "unravelled" version of this program
that shows exactly what happens with this loop.

Unravelling a `for` Loop

No loop, but logically equivalent:

```python
count_off = range(1, 4) # contains 1, 2, 3
number = count_off[0]
print(number)
number = count_off[1]
print(number)
number = count_off[2]
print(number)
```

🖨️ 👇

```
1
2
3
```

The body of the loop is repeated verbatim for each iteration we do. The value that `number` gets with each iteration is the next value stored in the sequence.

Loops on Other Sequences

Loops over strings go character-by-character:

```python
song_title = "respect"
for letter in song_title:
  print(song_title)
```

🖨️👇

```
r
e
s
p
e
c
t
```

Loops on Other Sequences

Loops over lists/tuples pull out each element from left to right.

```python
personal_data = ("Harry", "Smith", 27, 19147, False)
for datum in personal_data:
  print(datum)
```

🖨️ 👇

```
Harry
Smith
27
19147
False
```

Loops on Other Sequences

Loops over lists/tuples pull out each element from left to right.

```python
top_restaurants = ["Clubhouse", "UTown", "Han Dynasty", "Loco Pez"]
for favorite in top_restaurants:
  print(favorite)
```

🖨️ 👇

```
Clubhouse
UTown
Han Dynasty
Loco Pez
```

Looping Idioms: Repetition

Printing Values of a Sequence

- Strings, tuples, and lists can be printed out to reveal their contents, but ranges and other iterables don't have this convenience.
- Inspect a sequence by printing out each value contained inside.

```
for element in sequence:
  print(element)
```

Do Something `n` Times

- `range(n)` is a sequence that contains all integers from `0` to `n - 1`.
  - `len(range(n)) == n` always.
  - A `for` loop over `range(n)` will execute the body `n` times.

```
print("you're so funny.")
for x in range(8):
  print("ha")
print("lol")
```

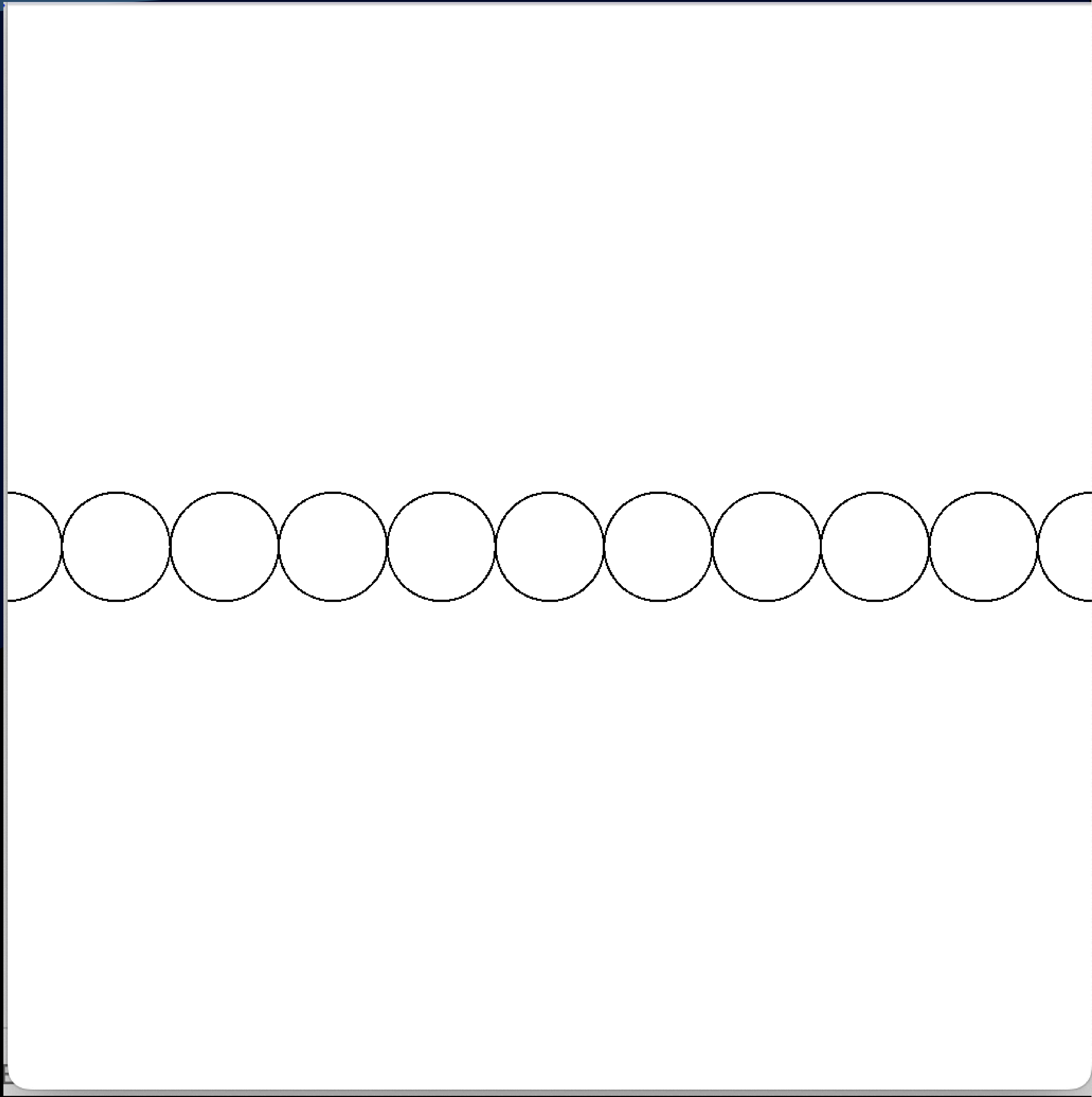
```
you're so funny.
ha
ha
ha
ha
ha
ha
ha
ha
lol
```

Note: didn't even use the variable `x` in the loop body. That's OK.

Do Something n Times

```python
import penndraw as pd
for x_position in range(11):
  pd.circle(x_position / 10, 0.5, 0.05)
pd.run()
```

🖌️ ➡️



Here, we use the value of `x_position` to evenly space eleven circles across a PennDraw canvas.

- Circles at `(0.0, 0.5)`, `(0.1, 0.5)`, `(0.2, 0.5)`, …

Looping Idioms: Copying & Filtering

Copy a Sequence into a List

Lists are the only mutable sequences we have so far. They can be more flexible than other sequences, which is nice.

- To create a list version of another sequence, you can create a new list and write a `for` loop to fill it.

```
new_list = []                # [] is a list with no contents
for value in sequence:       # For each value in the source sequence,
  new_list.append(value)     # add that value to the end of the new list.
```

Copy a Sequence into a List

```
new_list = []
dna_sequence = "ACGTCAGTAGACGACAT"
for base_pair in dna_sequence:      # For each value in the source sequence,
  new_list.append(base_pair)            # add that value to the end of the new list.
print(new_list)
```

🖨️👇

```
['A', 'C', 'G', 'T', 'C', 'A', 'G', 'T', 'A', 'G', 'A', 'C', 'G', 'A', 'C', 'A', 'T']
```

Now we could simulate modifications to the DNA sequence:

```
new_list[3] = "C" # Change the first "T" to a "C"
```

An Aside: Built-in Redundancy

Many of the common idioms we're covering here are *so common* that Python has built in shorter ways of doing them right into the language!

```
new_list = []             # [] is a list with no contents
for value in sequence:    # For each value in the source sequence,
  new_list.append(value)  # add that value to the end of the new list.
```

is logically equivalent to:

```
new_list = list(sequence)
```

I'll identify the "short", loopless versions of the idioms we cover here. It's useful to know both.

Filter Values Out of a Sequence

We can extend the previous idiom by only copying values that meet a certain condition. This is called **filtering.**

```
new_list = []                 # [] is a list with no contents
for value in sequence:        # For each value in the source sequence,
  if condition(value):        # if that value meets some condition
    new_list.append(value)  # add that value to the end of the new list.
```

`condition()` is a placeholder here to represent some boolean expression that helps decide whether or not to include `value`.

Filter Values Out of a Sequence

```
exam_scores = [100, 0, 89, 93, 78, 67, 0]
non_zeroes = []                # [] is a list with no contents
for score in exam_scores:      # For each score from the list,
  if score > 0:                # if that score is not zero,
    non_zeroes.append(score)   # add that score to the end of the new list.
print(non_zeroes)
```

🖨️👇

```
[100, 89, 93, 78, 67]
```

Filter Values Out of a Sequence

```python
names = ["haRry", "Adi", "molly", "jared", "cEDRIc", "Sukya", "TraviS"]
proper_caps = []              # [] is a list with no contents
for name in names:            # For each name from the list,
  if name.istitle():          # if that name is in "title case"
    proper_caps.append(name)  # add that name to the end of the new list.
print(proper_caps)
```

🖨️ 👇

```
["Adi", "Sukya"]
```

Looping Idioms: Aggregating

Aggregating Information

Sometimes, we only want to learn some *property* of a sequence instead of creating a whole new sequence.

- Commonly accomplished with an **accumulator variable:**
  - a variable that has its value updated over successive iterations of the loop
  - important to declare accumulator variables *outside* of the loop so we don't overwrite its value each time.

Counting Elements (`len()`)

As a simple example, what if we didn't have `len()` available to us?

```python
my_tuple = (10, 20, -10, -20, "Yes", "OK") # This is the sequence we'll iterate over
counter = 0                                # This is our accumulator variable starting at 0
for element in my_tuple:                   # For each value in our tuple,
  counter = counter + 1                    # add 1 to our counter.
print(counter)                             # 🖨️ ➡️ 6
```

- `counter` starts counting at `0`—before we've counted any elements, that's how many we've counted!
- Within each loop, we increment `counter` by `1`.
- We don't actually use each `element` in the tuple, we're just counting them as they "pass by" in the iteration.

Adding Elements (`sum()`)

Imagine that I write down how much money I spend per day over a few days. How can I figure out how much I spent overall?

```
my_tuple = (10.54, 11.90, 203.10, 0, 5.0) # This is the sequence we'll iterate over
total = 0                                 # This is our accumulator variable starting at 0
for price in my_tuple:                    # For each price in our tuple,
  total = total + price                   # add that price to our total.
print(counter)                            # 🖨️ ➡️ 230.54
```

Equivalent to `sum(my_tuple)`

Counting Elements That Meet a Condition

What if we only want to count those elements that match some condition we care about?

```
my_tuple = (10, 20, -10, -20, 0, 40)          # This is the sequence we'll iterate over
counter = 0                                   # This is our accumulator variable starting at 0
for element in my_tuple:                      # For each value in our tuple,
  if element >= 0:                            # if that element is not negative,
    counter = counter + 1                     # add 1 to our counter.
print(counter)                                # 🖨️ ➡️ 4
```

- This time, we only increment `counter` when a condition is met
- This time, we actually use the value of `element`

Be Cautious About Accumulator Variables

Make sure to pick the initial value of the accumulator **outside of the loop** so that we don't accidentally start over each loop!

```
my_tuple = (10, 20, -10, -20, 0, 40)        # This is the sequence we'll iterate over
for element in my_tuple:                     # For each value in our tuple,
  counter = 0                                # set counter to be equal to 0
  if element >= 0:                           # if that element is not negative,
    counter = counter + 1                    # add 1 to our counter.
print(counter)                               # 🖨️ ➡️ 1
```

The value of `counter` resets back to `0` for each element we look at.

Finding the Largest/Smallest Values (`max()`/`min()`)

Accumulator variables don't have to always increase.

To find the largest (smallest) value in a sequence:

- Look at each value and compare it to the largest (smallest) *so far.*
- If we find a new largest (smallest), write that down!
- At the end, the largest (smallest) so far is the also the largest (smallest) overall!

```python
exam_scores = [92, 99, 100, 98.5] # This is the sequence we'll iterate over
largest = exam_scores[0]          # We'll just "guess" that the first score is the largest.
for score in exam_scores:         # For each score,
  if score > largest:             # if that score is higher than the largest we've seen,
    largest = score               # that score is now the largest we've seen so far.
print(largest)                    # 🖨️ ➡️ 100
```

Finding the Largest/Smallest Values (`max()`/`min()`)

```
exam_scores = [92, 99, 100, 98.5] # This is the sequence we'll iterate over
largest = exam_scores[0]          # We'll just "guess" that the first score is the largest.
for score in exam_scores:         # For each score,
  if score > largest:             # if that score is higher than the largest we've seen,
    largest = score               # that score is now the largest we've seen so far.
print(largest)                    # 🖨 ➡ 100
```

is equivalent to:

```
exam_scores = [92, 99, 100, 98.5]
print(max(exam_scores))
```

Looping Idioms: Mapping

Mapping 🌏

We can modify the values in a list, one by one, using the same rule each time. For example, curving exam scores by adding 10 points:

```
curved_scores = []
exam_scores = [92, 99, 100, 98.5]
for score in exam_scores:
  curved_scores.append(score + 10)
```

Here, we are appending a value that is not just the same as the one that we're pulling out of the list.

Mapping In Place

We can apply the same curve to the list without creating a new list at all using `enumerate()`.

- Remove `for score in exam_scores`
- Replace it with `for index, score in enumerate(exam_scores)`
- Within the loop body,
  - `index` will store the index of the current element (i.e. `0`, `1`, `2`, ...)
  - `score` will store the current element itself

```python
exam_scores = [92, 99, 100, 98.5]
for index, score in enumerate(exam_scores):
  exam_scores[index] = score + 10
```

Caution with Mapping In Place

Careful! We have permanently changed the list `exam_scores`.

```
exam_scores = [92, 99, 100, 98.5]
print(exam_scores)
for index, score in enumerate(exam_scores):
  exam_scores[index] = score + 10
print(exam_scores)
```

🖨️ 👇

```
[92, 99, 100, 98.5]
[102, 109, 110, 108.5]
```

While Loops

`while` Loops

`while` loops are a more general form of looping: specify a condition and as long as that condition is met, repeat a body of statements

- like an `if` statement that checks its condition more than once
- everything that you accomplish with a `for` loop can be accomplished with a `while` loop, but in a more verbose way

Syntax:

```
while condition:
  statement_one
  statement_two
```

while Loops: Animation

We've already seen while loops as a way to run an animation loop forever and ever:

```
import penndraw as pd
x_center = 0.5 # SETUP
while True:
    pd.clear()                                # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)  # 2. draw this frame
    x_center += 0.01                          # 3. update shapes for next frame
    pd.advance()
```

Infinite `while` Loops

`while True:` is a tricky construct--its condition is always true by definition!

```python
while True:
    print("stuck :(")    # This will happen infinitely
print("I'm free!")       # This will never be reached
```

🖨️ 👇

```
stuck :(
stuck :(
stuck :(
stuck :(
stuck :(
stuck :(
stuck :(
stuck :(
stuck :(
stuck :(
...
```

Counting with `while`

We could use a `while` loop to solve our original counting problem:

```python
counter = 0
while counter < 5:
    print(counter)
    counter += 1
```

🖨️ 👇

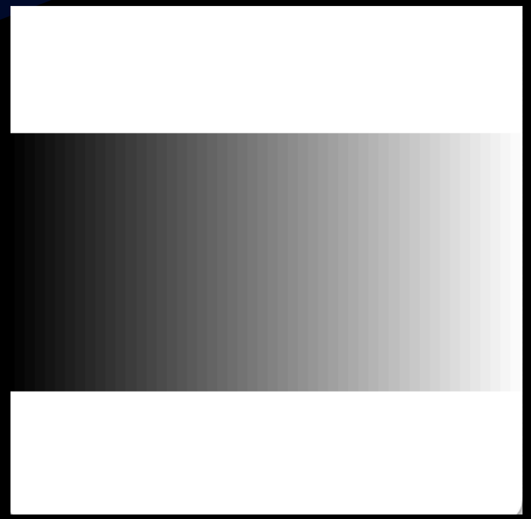
```
0
1
2
3
4
```

Recipe for a `while` Loop

1. Define a loop control variable
2. Define your loop condition in terms of the loop control variable
3. Make sure to update your loop control variable to eventually reach a case when your condition will go from true to false.

Example: Drawing with `while`

```python
import penndraw as pd
pd.set_canvas_size(256, 256)
x = 0   # define a loop control variable
while x < 256: # write condition in terms of l.c.v.
    pd.set_pen_color(x, x, x)
    pd.filled_rectangle(x / 255, 0.5, 1 / 255, 0.25)
    x += 1 # update the l.c.v., bringing loop closer to end
pd.run() # we do eventually get here!
```



`for` vs. `while`

- Use `for` loops to iterate over sequences
- Use `while` loops for animation, or when you're not sure how many iterations you need to go for
- Both kinds of loops can often be "replaced" with built-ins, but this takes practice to remember them all!