Loops and List Comprehensions

SHARP SIGHT

What you'll learn

- What are loops and why do we use them
- Two main kinds of loops: **`for`** loop and **`while`** loop
- The **`range()`** function
- Using **`break`** statements to discontinue loop execution
- List comprehensions

Loop basics

What are loops?

- Loops are pieces of code that repeat code several times before moving on
  - they execute the code and “loop” back to the beginning of the loop

- More technically: loops “iterate over a sequence”

- Different types of loops repeat differently
  - **`for`** loops repeat the code *n* times
  - **`while`** loops repeat *until condition is met*

"for" LOOPS

`for` LOOP BASICS

- `for` loops repeat a piece of code for each item of a sequence
  - they “iterate” over sequences

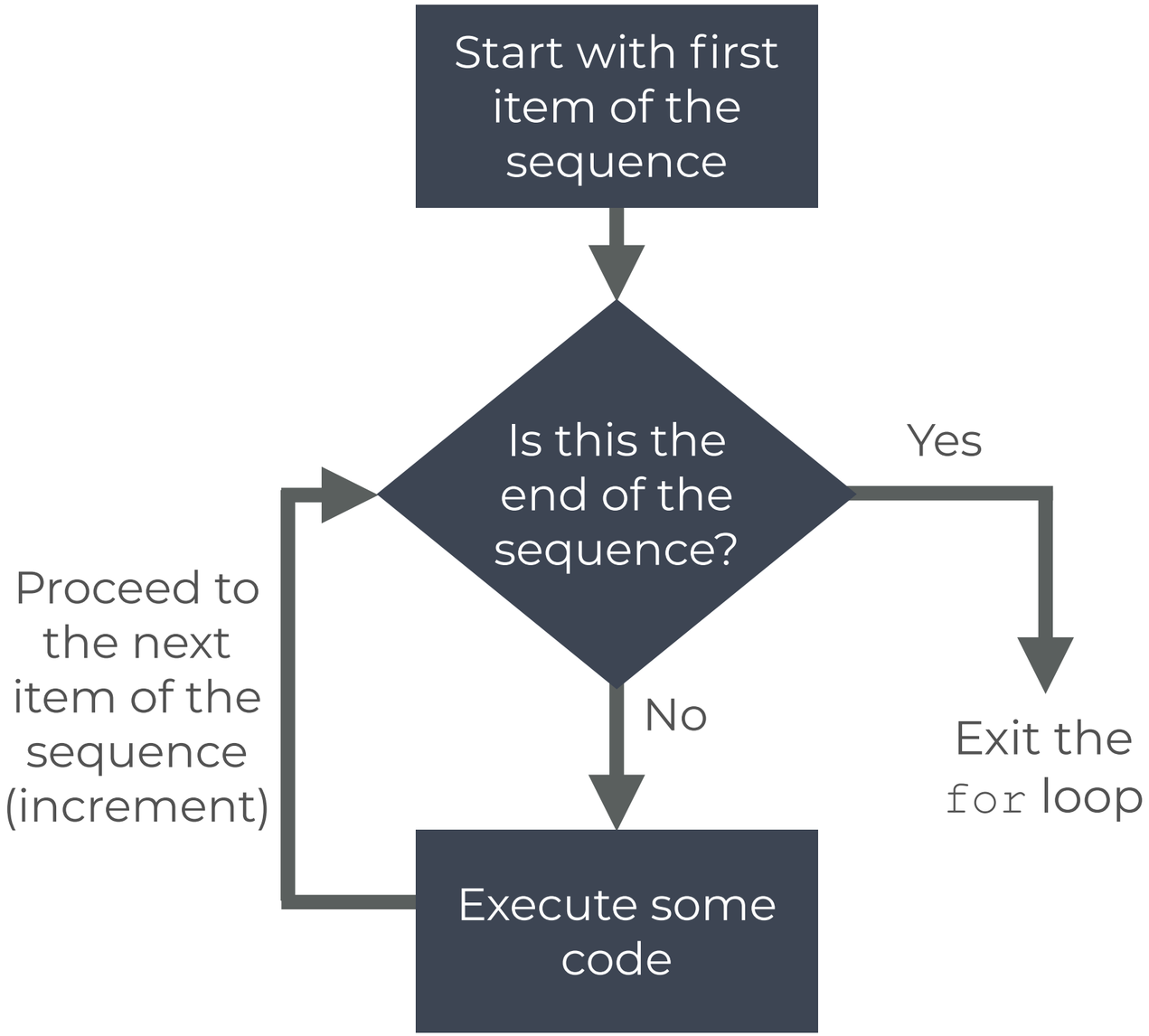
- `for` loops in Python will iterate over any ‘iterable’ object
  - lists
  - tuples
  - strings

- Iterating is also known as ‘traversing’

How `for` loops work

- This is a simple flow chart that shows how `for` loops work

- The important point:
  - `for` loops repeat a code block for every item of a sequence



`for` LOOPS ITERATE OVER ITERABLES AND SEQUENCES

WTF is an iterable?

(good question...)

Iterables are objects that can be iterated over

- I.e., something we can loop over

- Examples of iterables:
  - lists
  - strings
  - tuples
  - dictionaries
  - sets
  - ... and others

"for" LOOP SYNTAX

`for` LOOP SYNTAX

Use the `for` keyword to start a for loop

This is a Python sequence or iterable (i.e., something you can iterate over, like a list, tuple, dictionary, or string)

```
for item in sequence:
    code block to repeat
```

This code block will only be executed for every iteration of the loop

... i.e., the code will execute for every item in the sequence

`for` LOOP SYNTAX

Here, `item` is a variable that we define exclusively for the `for` loop

```
for item in sequence:
    code block to repeat
```

`item` is just a placeholder that holds the element of the sequence, as we iterate through the sequence

We can name this variable anything we want!

Code blocks of a `for` loop must be indented

```
for item in sequence:
    code block to repeat
```

This indentation must be present ....

The white space (i.e., indentation) is syntactically meaningful in Python

The best practice is to <u>use 4 spaces to indent code blocks</u>

Code blocks of a `for` loop must be indented

```
for item in sequence:
    code block to repeat
```

The code block can be 1 line long or hundreds of lines long.

As long as we want!

... but, all of the lines of a code block must be indented properly

A QUICK `for` LOOP SYNTAX EXAMPLE

```
mylist = [1,2,3]

for item in mylist:
    print(item)

1
2
3
```

Here, the list `mylist` is a type of iterable

We iterate over the elements of the sequence (`1`, `2` and `3`)

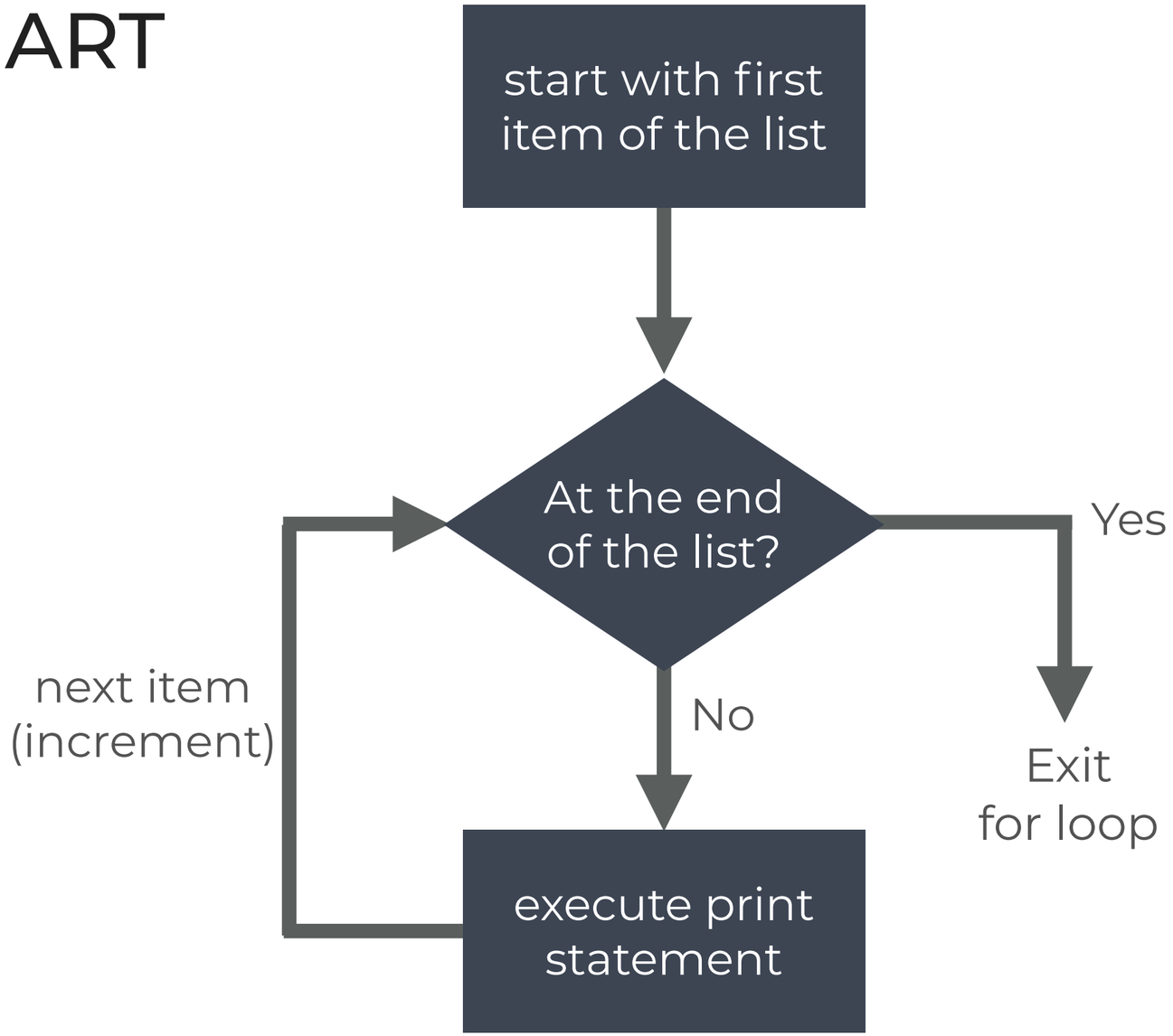
For every element of the sequence, we print out the element using a `print()` statement

Remember: think of `for` loop syntax like a flow chart

```
mylist = [1,2,3]

for item in mylist:
    print(item)

1
2
3
```



"for" LOOP EXAMPLES

Some examples of `for` loops

- For loops operate similarly on different sequences
  - lists
  - tuples
  - strings

- The following slides will show you concrete examples
  - show you how the code iterates through the sequence

Example: loop over a list

```
car_list = ['ferrari','porsche','bugatti']

for car in car_list:
    print(car)
```

- This code will iterate over every element of `car_list`
  - it will print out the car

- Note: we used the variable name "`car`" as our placeholder
  - It just represents the items of `car_list`

For every item in car_list, we print out the list item

CODE:

```
car_list = ['ferrari','porsche','bugatti']

for car in car_list:
    print(car)
```

car =

car_list:	ferrari	porsche	bugatti

OUTPUT:

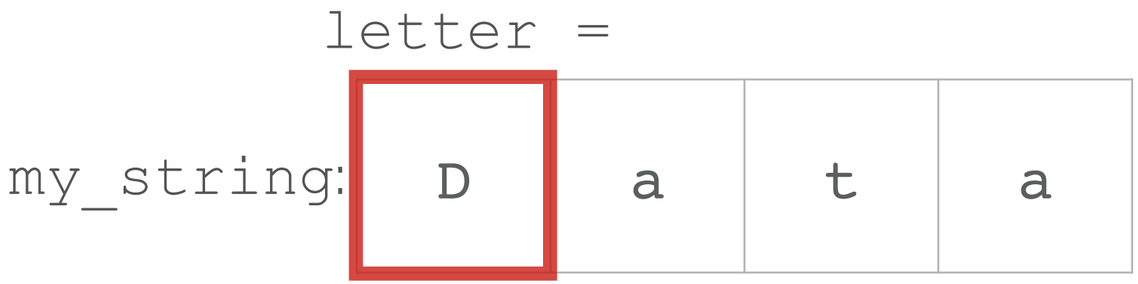
```
ferrari
porsche
bugatti
```

For every character in my_string, we print out the character

CODE:

```
my_string = "Data"

for letter in my_string:
    print(letter)
```



OUTPUT:

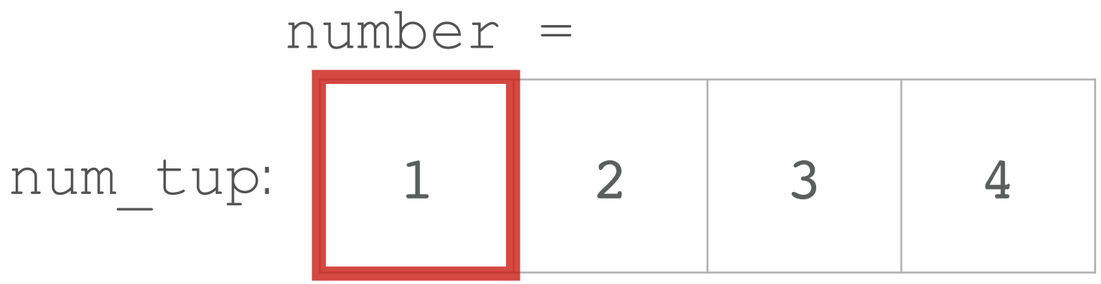
```
D
a
t
a
```

For every number num_tup, we print out the number

CODE:

```
num_tup = (1,2,3,4)

for number in num_tup:
    print(number)
```



OUTPUT:

```
1
2
3
4
```

In each of these examples, we iterated over an "iterable"

- In every case, a the for loop iterated over a sequence/iterable
  - the list is an iterable
  - the string is an iterable
  - the tuple is an iterable

- For every element of the sequence, we do something
  - execute the code block

- Note: the code block in a for loop can be much more complicated!
  - These examples used simple `print()` statements

The `range()` function

The `range()` function

- The `range()` function generates sequences of numbers
- For example, `range(5)` generates the sequence `0,1,2,3,4`
- We can use these sequences in our `for` loops

`range()` SYNTAX

Initiate the `range()` function

```
range(size-of-sequence)
```

Inside of `range()`, we specify the *size* of the numeric sequence we want to create

For example, `range(5)` will create a sequence that is 5 numbers long

A few important detail about the `range()` function

- The output of `range()` starts at 0 by default

- The output ends at 1 less than the size of the output
  - Example: `range(5)` produces the output `0,1,2,3,4`
  - `range(5)` does not include 5!

- `range()` can be used to generate wide variety of arithmetic sequences
  - note: we won't cover the complex uses of `range()`

Example: using `range()` in a `for` loop

The `for` loop iterates over the items of the numeric sequence and prints them out

```
for number in range(4):
    print(number)

0
1
2
3
```

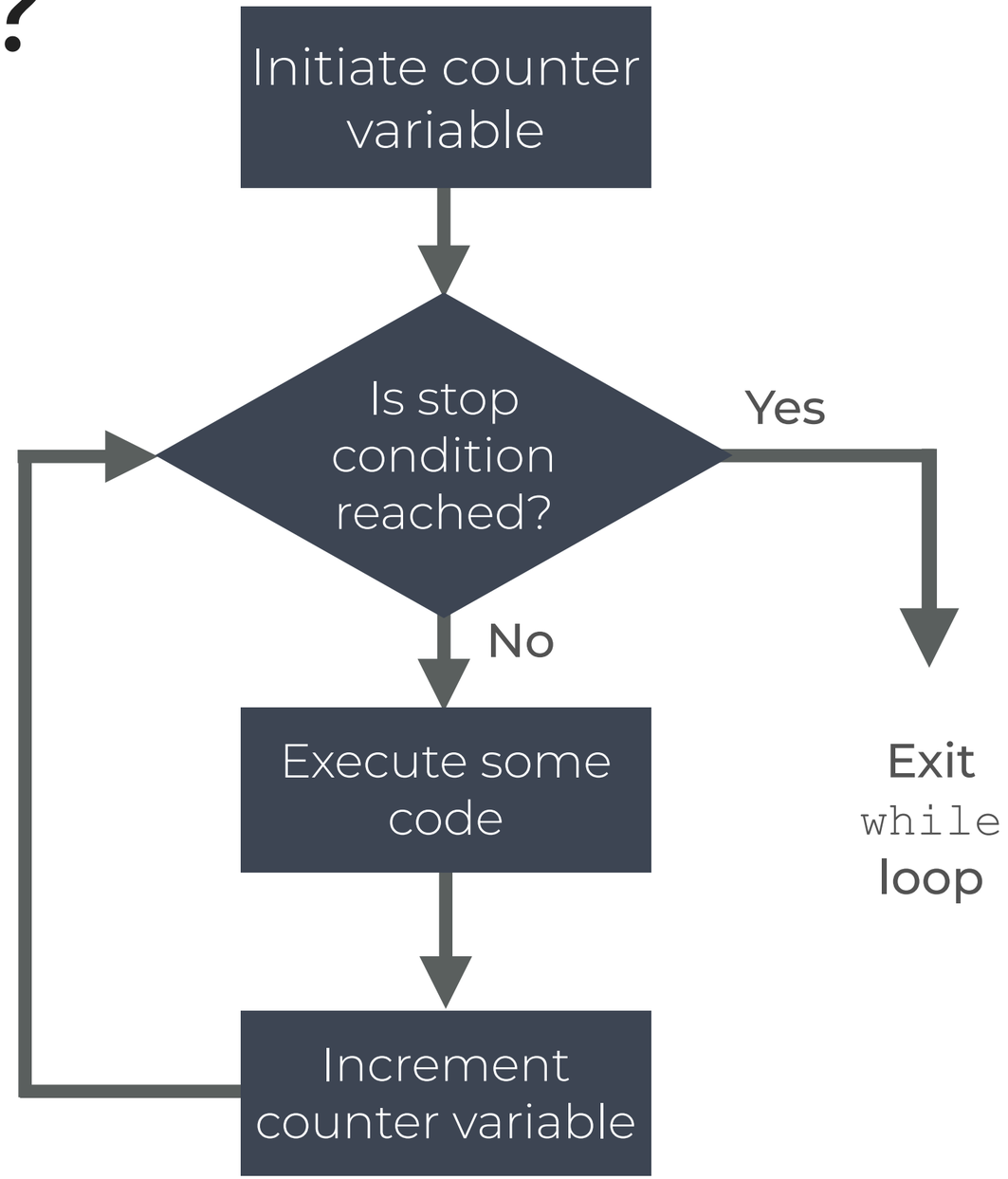
The `range()` function creates a sequence of 4 numbers that we can iterate over

`while` LOOPS

What is a `while` loop?

- **`while`** loops are similar to `for` loops

- They "loop" through and repeatedly execute some code until a stoping condition is reached



`while` LOOP SYNTAX

The `while` keyword initiates a `while` loop

A conditional statement specifies the stoping condition of the loop

```
while condition:
    code block to perform
```

The code indented underneath the `while` statement will be executed for every iteration of the loop, until the halting condition is reached

Remember ... the code block must be indented!

Example: simple `while` loop

Here, we're creating a variable called `counter`

The code block will print the value of `counter` and then increment `counter` by 1

Code:

```
counter = 0

while counter < 4:
    print(counter)
    counter = counter + 1
```

This stopping condition will cause the loop to run until `counter == 4`

Out:

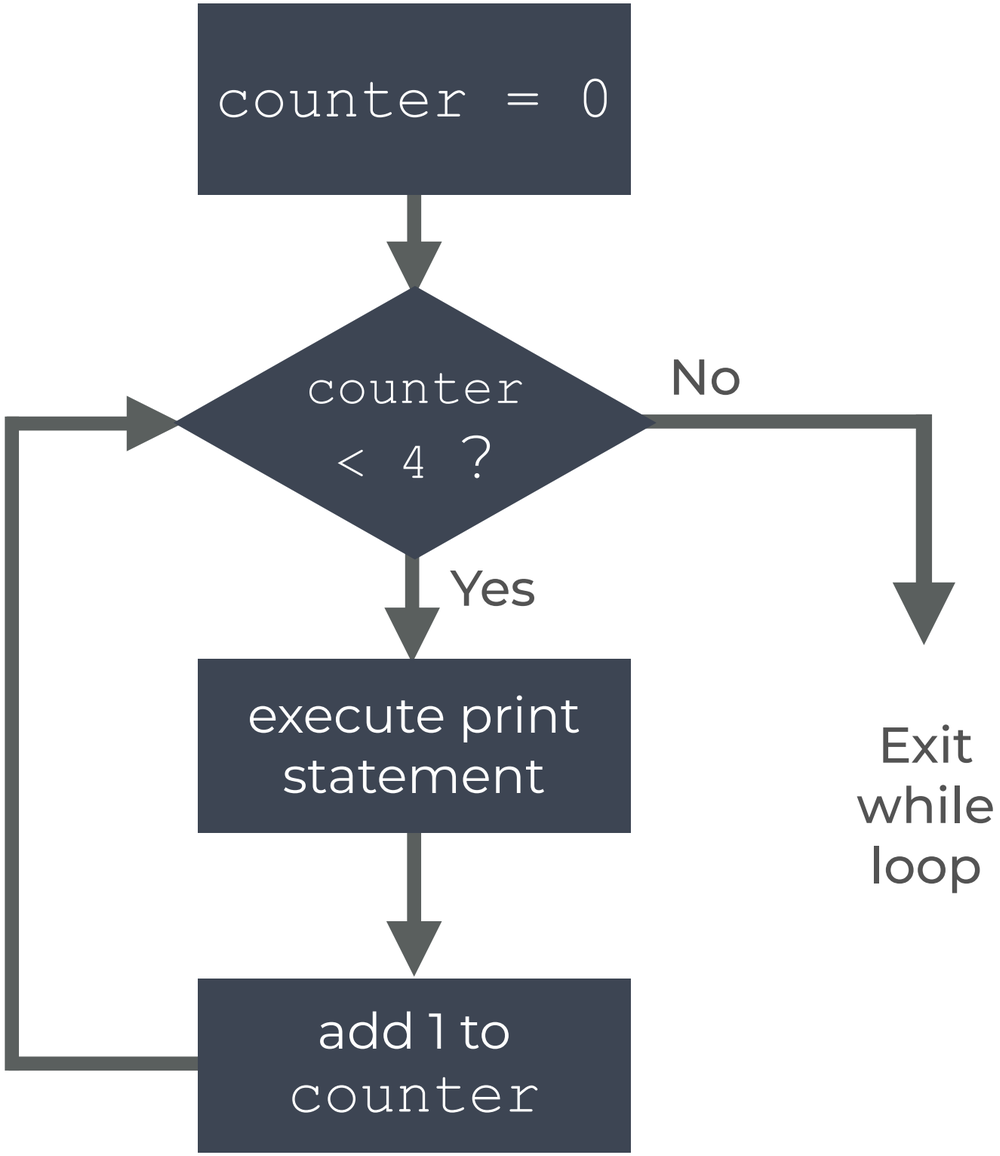
```
0
1
2
3
```

How this loop works

```
counter = 0

while counter < 4:
    print(counter)
    counter = counter + 1
```

- Every loop iteration
  - check the halt condition
  - execute print
  - add 1 to `counter`
  - repeat



Notes about `while` loops

- **`while`** loops continue to loop until a halting condition is reached

- The body code should include a variable that we use in the halting condition
  - we need to create a way to halt the loop
  - without a halting condition, the loop will continue to repeat
    - an “infinite loop”

`break` STATEMENTS

`break` STATEMENTS

- We can use **`break`** to “break out” of a **`for`** loop or **`while`** loop

- Note about nested `for` loops:
  - **`break`** will break out of the “smallest” **`for`** loop
  - i.e., the nearest **`for`** enclosing the **`break`** statement

Example: `break`

```
num_list = [1,2,3,42,5]
for number in num_list:
    print(number)
    if number == 42:
        print('we found it!')
        break
1,2,3,42, we found it!
```

- If `number == 42`, then we break out of the **`for`** loop
  - **`break`** stops execution
  - the loop won't continue on to **5**

for LOOPS VS while LOOPS

When to use ‘`for`’ vs ‘`while`’

- **`for`** is best when …
  - you have ‘iterable’ objects
  - you have sequences: lists, tuples, sets, strings

- **`while`** is best when …
  - you don’t have iterable objects or sequence to iterate through
    - no simple data structure to drive the looping process
  - you have logical conditions that can’t be represented by a sequence

List comprehensions

List comprehensions are a concise way to create lists

- List comprehensions have a compact syntax to create lists
  - list comprehensions put a `for` loop inside brackets

- The output of a list comprehension is a list

Syntax: list comprehension

This is a `for` loop that defines how we will repeat the *`expression`* for every value of *`iterable`*

```
new_list = [expression for x in iterable]
```

`Expression` is a piece of code that will execute for every iteration of the `for` loop

Syntax: list comprehension

Notice that all of this is enclosed inside of brackets

```
new_list = [expression for x in iterable]
```

So the output of a list comprehension is a `list`

Example: list comprehension

```
even_list = [x*2 for x in range(6)]

print(even_list)

[0,2,4,6,8,10]
```

Example: list comprehension

The code `range(6)` generates the sequence of integers 0, 1, 2, 3, 4, 5

```
even_list = [x*2 for x in range(6)]

print(even_list)

[0,2,4,6,8,10]
```

Example: list comprehension

For every value `x` of `range(6)`, this expression will output `x` times 2

This `for` loop will iterate over every value of `range(6)`

```
even_list = [x*2 for x in range(6)]

print(even_list)

[0,2,4,6,8,10]
```

Example: list comprehension

```
even_list = [x*2 for x in range(6)]

print(even_list)

[0,2,4,6,8,10]
```

Notice that the output is 2 times every value of `range(6)`

Example: list comprehension

```
even_list = [x*2 for x in range(6)]

print(even_list)

[0,2,4,6,8,10]
```

```
type(even_list)
list
```

Also notice that the output is a `list`

Remember: list comprehensions are just a concise way to create lists

Syntax comparison: `for` loop vs. list comprehension

Remember … list comprehensions are like `for` loops that generate a `list`

List Comprehension

```
new_list = [x * 2 for x in range(6)]
```

`for` Loop

```
new_list = []
for x in range(6):
    new_list.append(x * 2)
```

These two pieces of code produce the same output

List Comprehension

```
new_list = [x * 2 for x in range(6)]
```

for Loop

```
new_list = []
for x in range(6):
    new_list.append(x * 2)
```

When to use list comprehensions

- List comprehensions replace `for` loops in some instances

- List comprehensions are good for creating lists that contain sequences
  - regular sequences that can be described mathematically

Be careful with list comprehensions!

- List comprehensions are more concise

- BUT, list comprehensions are harder to debug!
  - Be careful!

Recap

Recap of what we learned

- Loops enable you to repeat a piece of code many times
- Two main kinds of loops: **`for`** loop and **`while`** loop
- The **`range()`** function creates sequences of integers
- Use **`break`** statements to discontinue loop execution
- List comprehensions are a concise way to create lists