CSCI 141

Lecture 10:
Modules, `random`, loops loops loops loops, `range`

CSCI 141

Lecture 10:
Modules, `random`, loops loops loops loops, `range`

fo(u)r loops, get it?

Happenings

CS Mentors Workshop: COMMAND THE LINE!

Wednesday (today!) 10/16 - 4pm CF 165

Announcements

Announcements

- A3 is out! Due next Tuesday.

Announcements

- A3 is out! Due next Tuesday.
    - Start early so you have plenty of time to study for…

Announcements

- A3 is out! Due next Tuesday.
  - Start early so you have plenty of time to study for…
- The midterm exam is a week from Friday!

Announcements

- A3 is out! Due next Tuesday.
  - Start early so you have plenty of time to study for…
- The midterm exam is a week from Friday!
  - Covers material through Monday.

Study Tips

Study Tips

Reading is not enough: **solve problems.**

Study Tips

Reading is not enough: **solve problems**.

- **Goals** slides: can you do these things? Try and see.

Study Tips

Reading is not enough: **solve problems**.

- **Goals** slides: can you do these things? Try and see.
- **Terminology**: be able to discuss the meaning of all words that appear in blue in the slides

Study Tips

Reading is not enough: **solve problems**.

- **Goals** slides: can you do these things? Try and see.
- **Terminology**: be able to discuss the meaning of all words that appear in blue in the slides
- **Socrative questions**: make sure you know how to solve them. Then, try code in Thonny or compare answers with your peers.

Study Tips

Reading is not enough: **solve problems**.

- **Goals** slides: can you do these things? Try and see.
- **Terminology**: be able to discuss the meaning of all words that appear in blue in the slides
- **Socrative questions**: make sure you know how to solve them. Then, try code in Thonny or compare answers with your peers.
- **Demo code**: solve the same problem without without looking at my code.

Study Tips

Reading is not enough: **solve problems**.

- **Goals** slides: can you do these things? Try and see.
- **Terminology**: be able to discuss the meaning of all words that appear in blue in the slides
- **Socrative questions**: make sure you know how to solve them. Then, try code in Thonny or compare answers with your peers.
- **Demo code**: solve the same problem without without looking at my code.
- **QOTDs**: still available on Canvas - make sure you know how to solve them.

Study Tips

Reading is not enough: **solve problems**.

- **Goals** slides: can you do these things? Try and see.
- **Terminology**: be able to discuss the meaning of all words that appear in blue in the slides
- **Socrative questions**: make sure you know how to solve them. Then, try code in Thonny or compare answers with your peers.
- **Demo code**: solve the same problem without without looking at my code.
- **QOTDs**: still available on Canvas - make sure you know how to solve them.
- **Exercises** from the eBook

Study Tips

Reading is not enough: **solve problems**.

- **Goals** slides: can you do these things? Try and see.
- **Terminology**: be able to discuss the meaning of all words that appear in blue in the slides
- **Socrative questions**: make sure you know how to solve them. Then, try code in Thonny or compare answers with your peers.
- **Demo code**: solve the same problem without without looking at my code.
- **QOTDs**: still available on Canvas - make sure you know how to solve them.
- **Exercises** from the eBook

A study guide including sample coding questions is coming later this week.

Goals

- Know how to import a module and call its functions
  - Know how to generate random numbers using the `random` module's `randint` function.
  - Know how to find the documentation for a module and its functions to learn what they do.
- Know the syntax and behavior of the `for` statement (`for` loop)
- Know how to use the `range` function in the header of a `for` loop.

Last time: the `while` statement

Not so different from an `if` statement:

`while` keyword

a boolean expression (the condition)

a colon :

```
while [condition]:
    [indented code block]
```

an indented code block: one or more statements to be executed **while** the boolean expression evaluates to True

The `while` statement: Semantics (Behavior)

If statement:

1. Evaluate the condition
2. If true, execute body (code block), then continue on.

While statement:

1. Evaluate the condition
2. If true, execute body, otherwise skip step 3 and continue on.
3. Go back to step 1

The `while` statement: A Working Example

```
# print account balance after each
# of five years:
balance = 100.0 # starting balance
year = 1
while year <= 5:
    balance = balance + (0.02 * balance)
    print(balance)
    year = year + 1
```

Terminology notes:

- the line with `while` and the condition is the loop header
- the code block is the loop body
- the entire construct (header and body) is a `while` statement
- usually people call them while loops instead

QOTD

What is the output of the following code?

```
count = 10
while count < 21:
    print(count, end=" ")
    count += 3
```

QOTD

What are the values of `m` and `n` after this code is executed?

```
n = 12345
m = 0
while n != 0:
    m = (10 * m) + (n % 10)
    n //= 10
```

What can you do with while loops?

- Anything you can write code to do!
- Not just counting.

Demo: Not just counting

Demo: Not just counting

- sum_inputs.py:
    - sum user-provided positive numbers until a negative number is entered

Other Peoples’ Code

We’ve already used code other people wrote by calling built-in Python functions:

- `print, input, type`

Built-in functions are special because they’re always available.

Many other functions exist in the Python Standard Library, which is a collection of modules containing many more functions.

Other Peoples’ Code

An example: I want to generate a random integer between 0 and 10.

```
import random
```

Other Peoples' Code

An example: I want to generate a random integer between 0 and 10.

I don't know how to do this.

```
import random
```

Other Peoples’ Code

An example: I want to generate a random integer between 0 and 10.

I don’t know how to do this.

Someone who does has written some functions for me. They live in the `random` module:

```
import random
```

Other Peoples' Code

An example: I want to generate a random integer between 0 and 10.

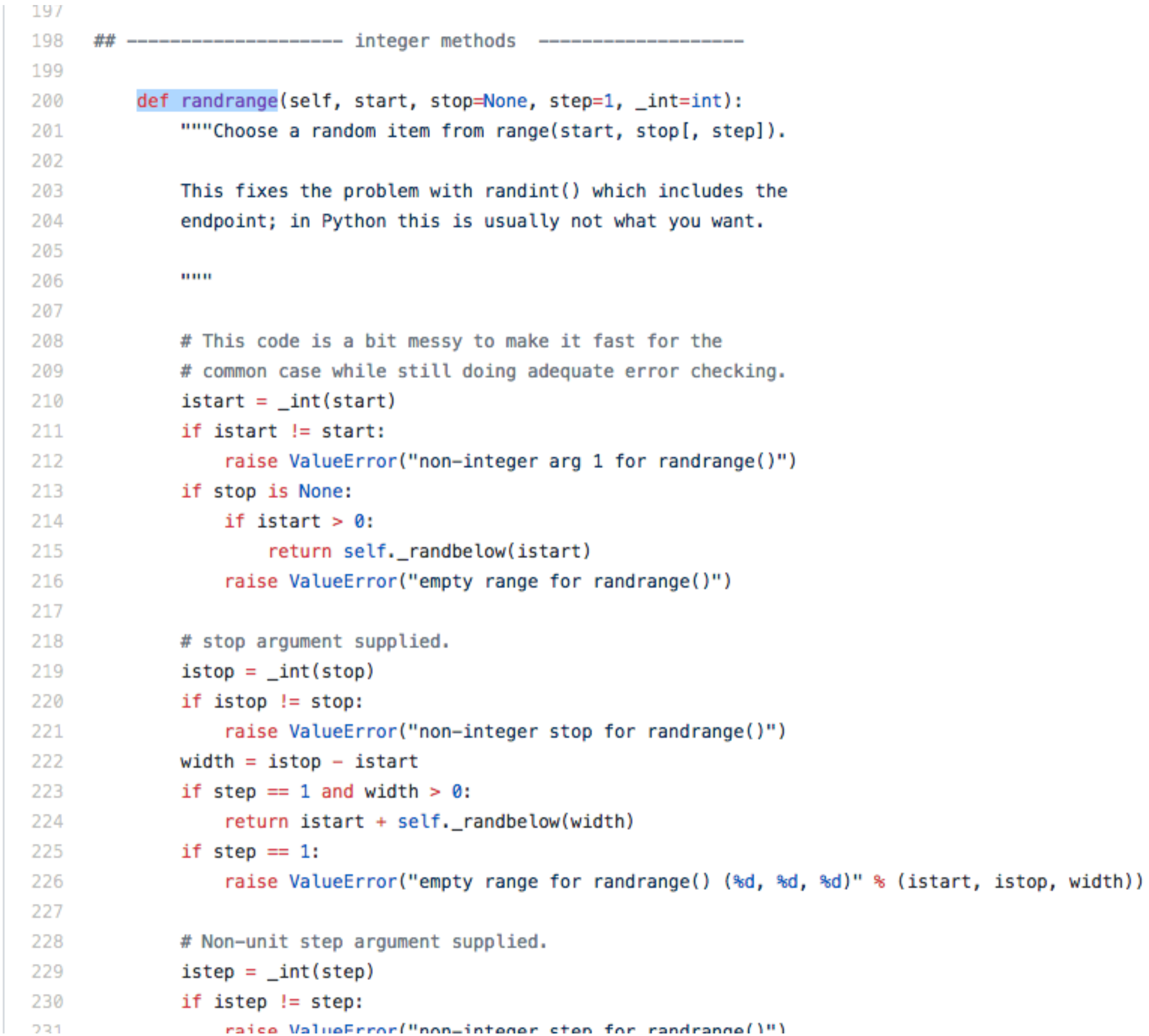
I don't know how to do this.

Someone who does has written some functions for me. They live in the `random` module:

```
import random
```

I could go look at the source code…

⋮



```python
## ------------------- integer methods  -------------------

    def randrange(self, start, stop=None, step=1, _int=int):
        """Choose a random item from range(start, stop[, step]).

        This fixes the problem with randint() which includes the
        endpoint; in Python this is usually not what you want.

        """

        # This code is a bit messy to make it fast for the
        # common case while still doing adequate error checking.
        istart = _int(start)
        if istart != start:
            raise ValueError("non-integer arg 1 for randrange()")
        if stop is None:
            if istart > 0:
                return self._randbelow(istart)
            raise ValueError("empty range for randrange()")

        # stop argument supplied.
        istop = _int(stop)
        if istop != stop:
            raise ValueError("non-integer stop for randrange()")
        width = istop - istart
        if step == 1 and width > 0:
            return istart + self._randbelow(width)
        if step == 1:
            raise ValueError("empty range for randrange() (%d, %d, %d)" % (istart, istop, width))

        # Non-unit step argument supplied.
        istep = _int(step)
        if istep != step:
            raise ValueError("non-integer step for randrange()")
```

⋮

Ar
int

I d

S
Th

e.

I c

Other Peoples' Code

An example: I want to generate a random integer between 0 and 10.

I don't know how to do this.

Someone who does has written some functions for me. They live in the `random` module:

```
import random
```

I could go look at the source code… but I'd rather just use their functions without knowing **how** they work.

```
num = random.randint(0,10)
```

Other Peoples' Code

```
import random
num = random.randint(0,10)
```

Two questions:

1. **What is this syntax about?**

2. How do I know what the function does?

Using Modules: Syntax

The Python Standard Library is a collection of modules containing many more functions.

To use functions in a module, you need to import the module using an `import` statement:

```
import module
```

By convention, we put all import statements at the **top** of programs.

Using Modules: Syntax

The Python Standard Library is a collection of modules containing many more functions.

To use functions in a module, you need to import the module using an `import` statement:

```
import module
```

(replace the in *`this font`* with the specific module name)

By convention, we put all import statements at the **top** of programs.

Using Modules: Syntax

Once you’ve imported a module:

```
    import random
```

you can call functions in that module using the following syntax:

```
     random.randint(0,10)
```

Using Modules: Syntax

Once you've imported a module:

```
import random
```

you can call functions in that module using the following syntax:

$$\underbrace{\texttt{random}}_{\text{Module name}}\texttt{.randint(0,10)}$$

Using Modules: Syntax

Once you’ve imported a module:

```
import random
```

you can call functions in that module using the following syntax:

$$\underbrace{\texttt{random}}_{\text{Module name}}\texttt{.}\underbrace{\texttt{randint(0,10)}}_{\text{Function call (the usual syntax)}}$$

Using Modules: Syntax

Once you’ve imported a module:

```python
import random
```

you can call functions in that module using the following syntax:

$$\underbrace{\texttt{random}}_{\text{Module name}}\underset{\text{Dot}}{\texttt{.}}\underbrace{\texttt{randint(0,10)}}_{\text{Function call (the usual syntax)}}$$

Other Peoples' Code

```
import random
num = random.randint(0,10)
```

Two questions:

1. What is this syntax about?

2. **How do I know what the function does?**

Other Peoples' Code

```
import random
num = random.randint(0,10)
```

Two questions:

1. What is this syntax about?

2. **How do I know what the function does?**

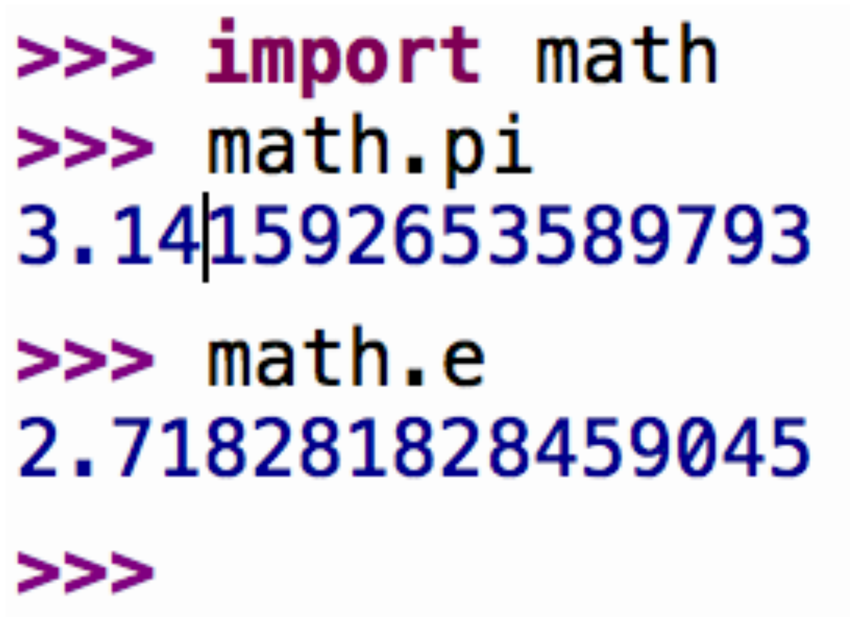
Read about it in the Python documentation.
My approach, in practice:

1. Google "python 3 <whatever>"
2. Make sure the URL is from python.org and has version python 3.x

example

`math` module

- The math module has useful stuff!
- You can read about it in the documentation.
- logarithms, trigonometry, …
- Modules can also contain values:



```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>>
```

You try it out:

Write a program to compute and print the average of 100 random numbers between 0 and 10.

Hot take: for some tasks, `while` loops are annoying.

Hot take: for some tasks, `while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”

Hot take: for some tasks, `while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

Hot take: for some tasks, `while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

```python
i = 0
while i < 10:
    some_thing()
    i += 1
```

Hot take: for some tasks, `while` loops are annoying.

- Often, you want: "Do `some_thing()` 10 times"
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don't even care about `i`, it's just bookkeeping!

Hot take: for some tasks, `while` loops are annoying.

- Often, you want: "Do `some_thing()` 10 times"
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don't even care about `i`, it's just bookkeeping!

- Wouldn't it be great if we could:

Hot take: for some tasks, `while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don’t even care about `i`, it’s just bookkeeping!

- Wouldn’t it be great if we could:

```
do 10 times:
    some_thing()
```

Hot take: for some tasks, `while` loops are annoying.

- Often, you want: "Do `some_thing()` 10 times"

- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don't even care about `i`, it's just bookkeeping!

- Wouldn't it be great if we could:

```
do 10 times:
    some_thing()
```

We (almost) can! Using `for` loops.

The `for` statement: syntax

```
for var_name in sequence:
    codeblock
```

The `for` statement: syntax

for keyword

```
for var_name in sequence:
    codeblock
```

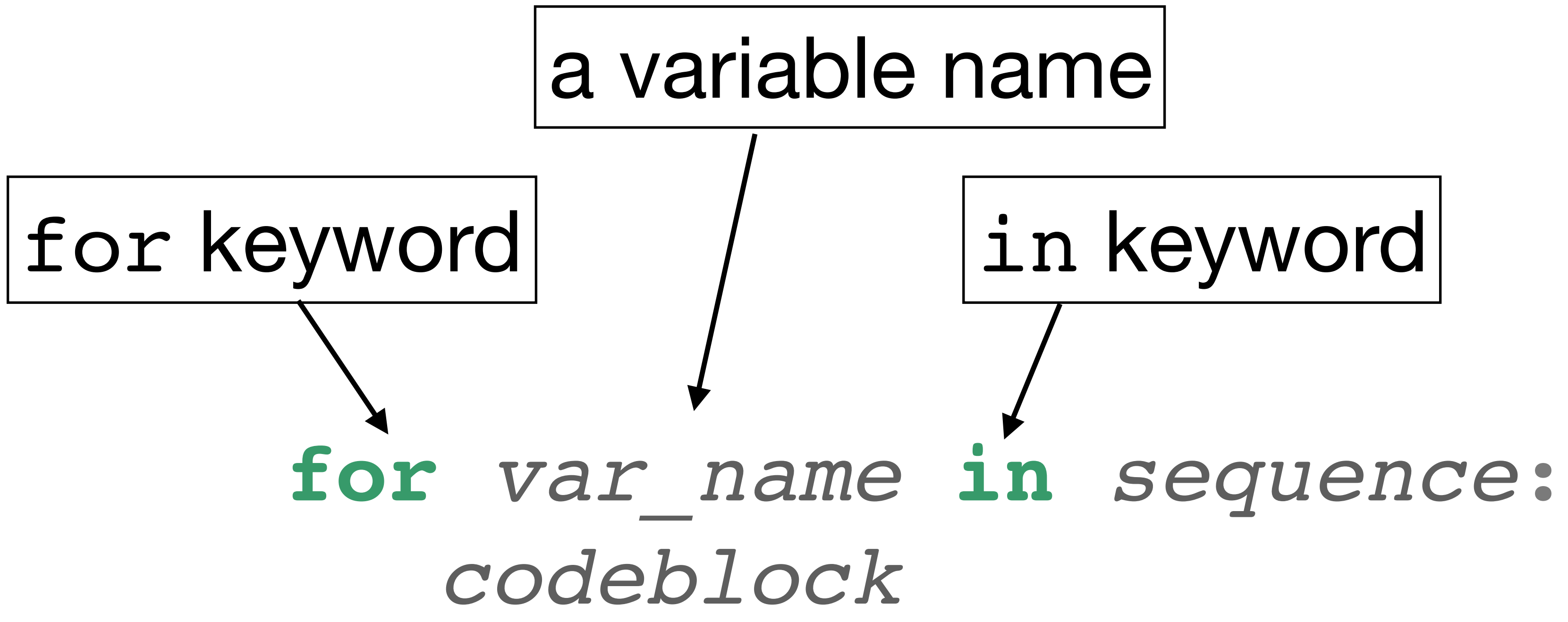
The `for` statement: syntax

a variable name

`for` keyword

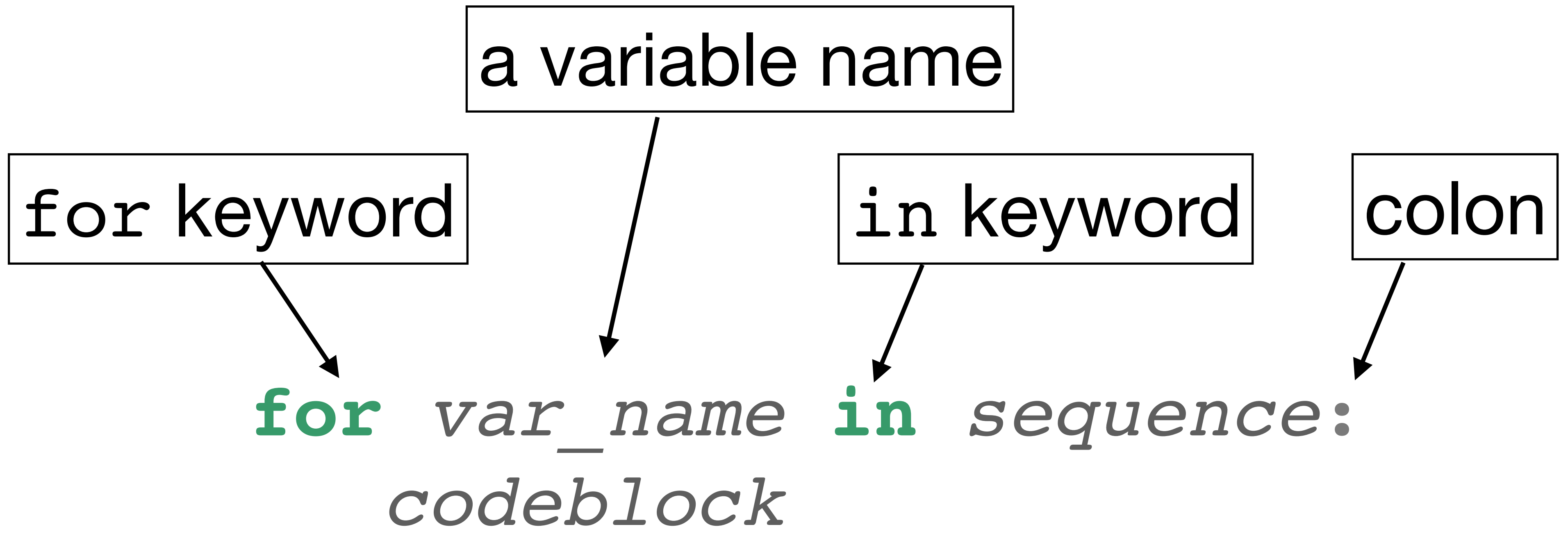
```
for var_name in sequence:
    codeblock
```

The `for` statement: syntax



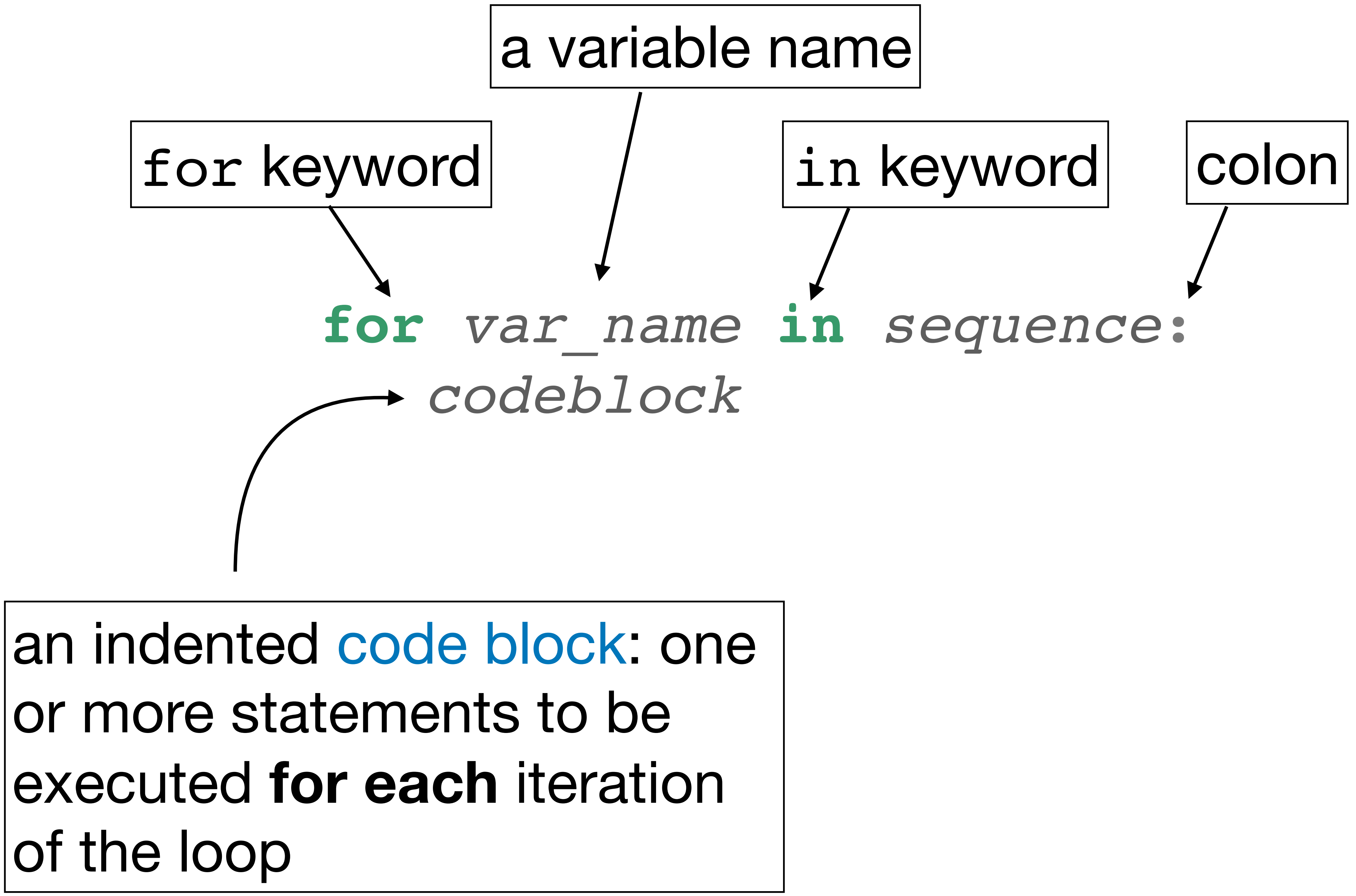
```
for var_name in sequence:
    codeblock
```

The `for` statement: syntax



```
for var_name in sequence:
    codeblock
```

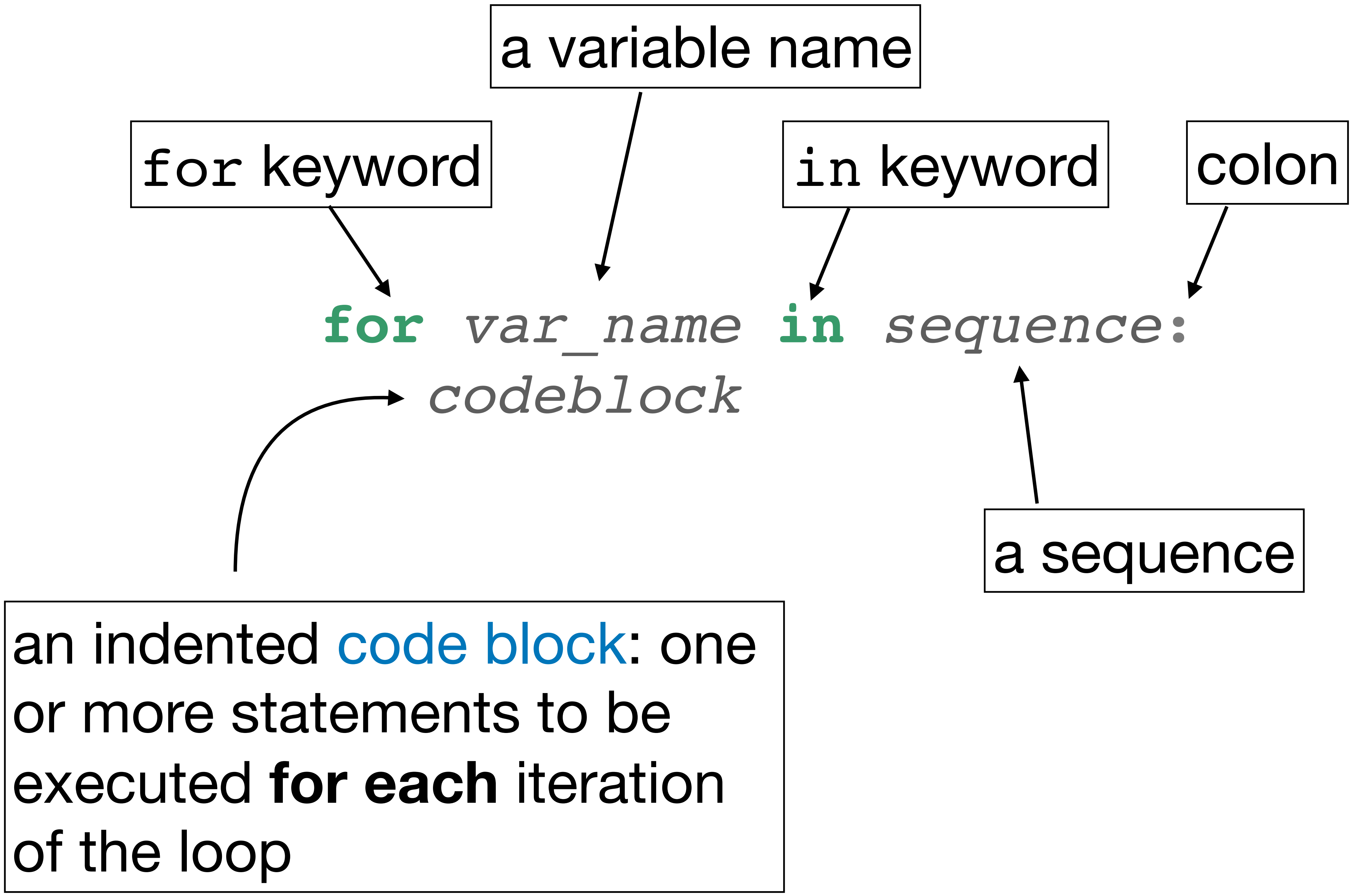
The `for` statement: syntax



```
for var_name in sequence:
    codeblock
```

an indented code block: one or more statements to be executed **for each** iteration of the loop

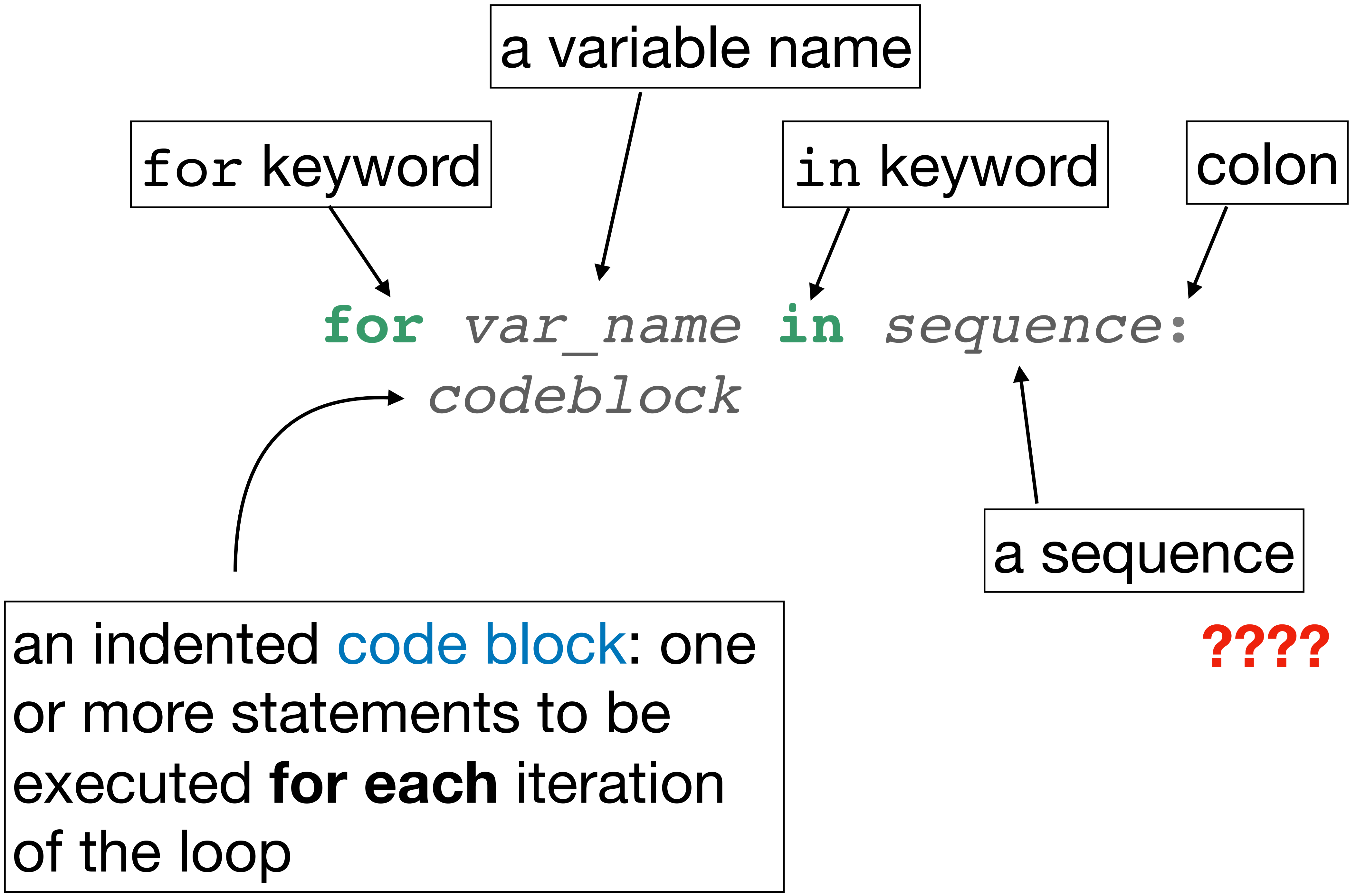
The `for` statement: syntax



```
for var_name in sequence:
    codeblock
```

an indented code block: one or more statements to be executed **for each** iteration of the loop

The `for` statement: syntax



```
for var_name in sequence:
    codeblock
```

an indented code block: one or more statements to be executed **for each** iteration of the loop

????

Sequences in Python: Lists

```python
for color in ["red", "green", "blue"]:
    print(color)
```

This code prints:

```
red
green
blue
```

Sequences in Python: Lists

```
for color in ["red", "green", "blue"]:
    print(color)
```

This is a `list`: an ordered collection of values.
Much more on these later.

This code prints:

```
red
green
blue
```

The `for` statement: behavior

```python
for color in ["red", "green", "blue"]:
    print(color)
```

This code prints:

```
red
green
blue
```

The `for` statement: behavior

```python
for color in ["red", "green", "blue"]:
    print(color)
```

The loop body is executed once **for each** value in the sequence (list).

This code prints:

```
red
green
blue
```

The `for` statement: behavior

```
for color in ["red", "green", "blue"]:
    print(color)
```

The loop body is executed once **for each** value in the sequence (list).

This code prints:

```
red
green
blue
```

In *each* iteration, the loop variable

The `for` statement: behavior

```
for color in ["red", "green", "blue"]:
    print(color)
```

The loop body is executed once **for each** value in the sequence (list).

This code prints:

```
red
green
blue
```

In *each* iteration, the loop variable (`color`)

The `for` statement: behavior

```
for color in ["red", "green", "blue"]:
    print(color)
```

The loop body is executed once **for each** value in the sequence (list).

This code prints:

```
red
green
blue
```

In *each* iteration, the loop variable (`color`) takes on a *different* value from the sequence:

The `for` statement: behavior

```
for color in ["red", "green", "blue"]:
    print(color)
```

The loop body is executed once **for each** value in the sequence (list).

This code prints:

```
red
green
blue
```

In *each* iteration, the loop variable (`color`) takes on a *different* value from the sequence:

(`"red"`, then `"green"`, then `"blue"`)

The `for` statement: behavior

```
for color in ["red", "green", "blue"]:
    print(color)
```

The loop body is executed once **for each** value in the sequence (list).

This code prints:

```
red
green
blue
```

In *each* iteration, the loop variable (`color`) takes on a *different* value from the sequence:

(`"red"`, then `"green"`, then `"blue"`)

Notice: the loop variable gets updated **automatically** after each iteration!

Sequences in Python: Ranges

Lists are great if you have a list of things, but what about:

Sequences in Python: Ranges

Lists are great if you have a list of things, but what about:

“Do `some_thing()` 10 times”?

Sequences in Python: Ranges

Lists are great if you have a list of things, but what about:

“Do `some_thing()` 10 times”? ugh.

Sequences in Python: Ranges

Lists are great if you have a list of things, but what about:

“Do `some_thing()` 10 times”? ugh.

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    some_thing()
```

Sequences in Python: Ranges

Lists are great if you have a list of things, but what about:

“Do `some_thing()` 10 times”? ugh.

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    some_thing()
```

New function to the rescue: `range` makes it easy to generate lists like this.

Sequences in Python: Ranges

```
for i in range(5):
    print(i)
```

This code prints:

```
0
1
2
3
4
```

Sequences in Python: Ranges

```
for i in range(5):
    print(i)
```

The `range` function returns a sequence of integers.

This code prints:

```
0
1
2
3
4
```

Sequences in Python: Ranges

```
for i in range(5):
    print(i)
```

The `range` function returns a sequence of integers.

Not technically a list, but acts like one: more on this later

This code prints:

```
0
1
2
3
4
```

Sequences in Python: the `range` function

Sequences in Python: the `range` function

```
for i in range(5):
    print(i, end=“ “)
```

prints: 0 1 2 3 4

Sequences in Python: the `range` function

`range(a)`: from **0** *up to* but *not including* **`a`**

```
for i in range(5):
    print(i, end=" ")
```

prints: `0 1 2 3 4`

Sequences in Python: the `range` function

`range(a)`: from **0** *up to* but *not including* **`a`**

```
for i in range(5):
    print(i, end=“ “)
```

prints: `0 1 2 3 4`

```
for i in range(2, 5):
    print(i, end=“ “)
```

prints: `2 3 4`

Sequences in Python: the `range` function

`range(`**`a`**`)`: from **0** *up to* but *not including* **`a`**

```
for i in range(5):
    print(i, end=" ")
```

prints: `0 1 2 3 4`

`range(`**`a, b`**`)`: from **a** *up to* but *not including* **`b`**

```
for i in range(2, 5):
    print(i, end=" ")
```

prints: `2 3 4`

Sequences in Python: the `range` function

`range(a)`: from **0** *up to* but *not including* **a**

```
for i in range(5):
    print(i, end=" ")
```

prints: `0  1  2  3  4`

`range(a, b)`: from **a** *up to* but *not including* **b**

```
for i in range(2, 5):
    print(i, end=" ")
```

prints: `2  3  4`

```
for i in range(1, 8, 3):
    print(i, end=" ")
```

prints: `1,  4,  7`

Sequences in Python: the `range` function

`range(`**`a`**`)`: from **0** *up to* but *not including* **`a`**

```
for i in range(5):
    print(i, end=" ")
```

prints: `0 1 2 3 4`

`range(`**`a, b`**`)`: from **a** *up to* but *not including* **`b`**

```
for i in range(2, 5):
    print(i, end=" ")
```

prints: `2 3 4`

`range(`**`a, b, c`**`)`: sequence from **a** *up to* but *not including* **`b`** counting in *increments* of **`c`**

```
for i in range(1, 8, 3):
    print(i, end=" ")
```

prints: `1, 4, 7`

Converting ranges to lists

The `range` function returns a **sequence** of integers.

It's not technically a **list**: print(range(4)) does not print `[1, 2, 3]`

To turn the range into a list (e.g., to print it), we can use the list function:

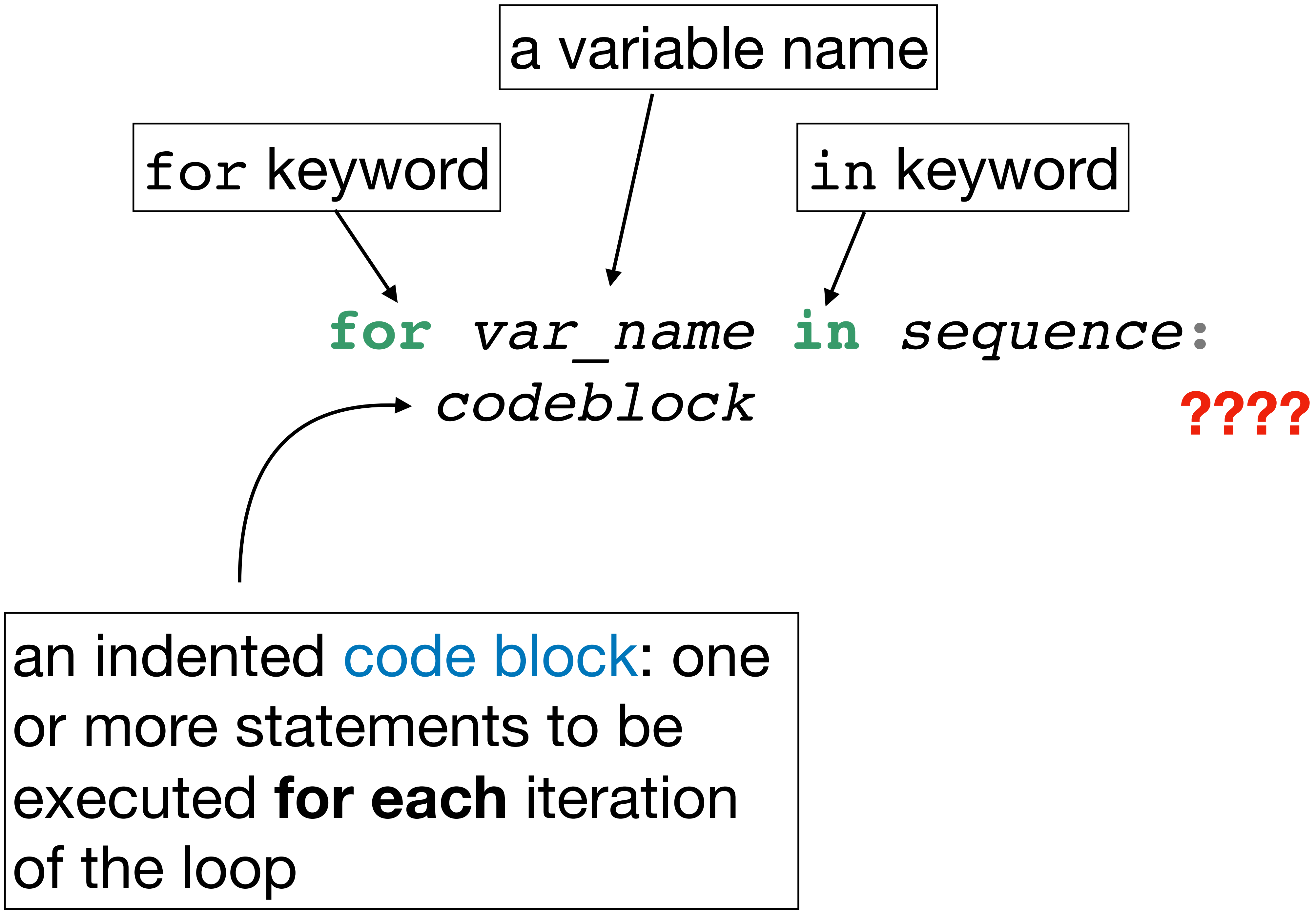
```
list(range(2, 5)) => [2, 3, 4]
```

Range function: Demo

- demo in shell
    - one, two, and three argument versions
- ranges.py - poll questions

Range function: Demo

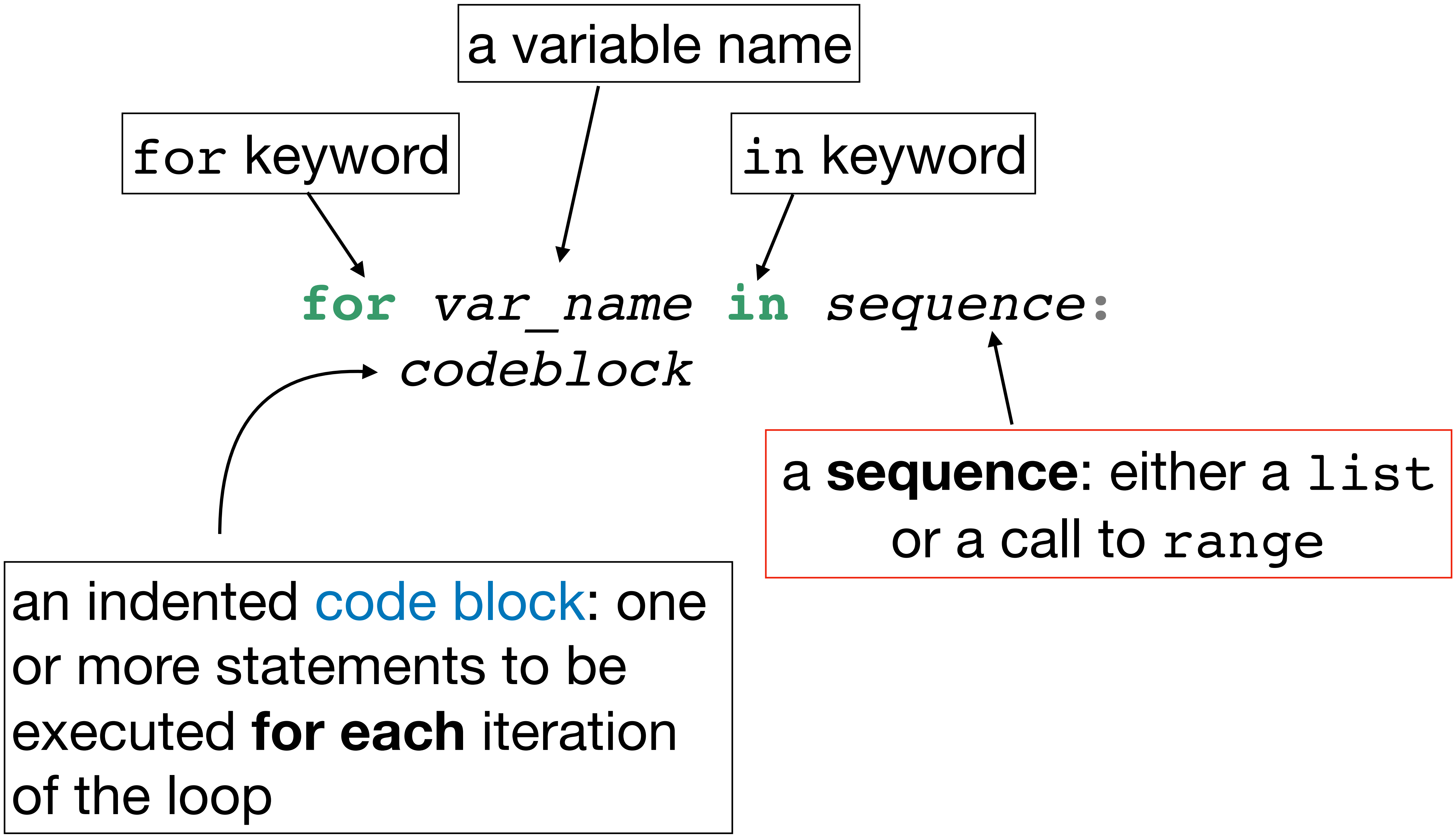
Back to `for` loops...



```
for var_name in sequence:
    codeblock
```

an indented code block: one or more statements to be executed **for each** iteration of the loop

Back to `for` loops...



```
for var_name in sequence:
    codeblock
```

`for` keyword

a variable name

`in` keyword

a **sequence**: either a `list` or a call to `range`

an indented code block: one or more statements to be executed **for each** iteration of the loop

`while` loops are annoying.

`while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”

`while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

`while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

`while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don’t even care about `i`,
it’s just bookkeeping!

`while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don’t even care about `i`, it’s just bookkeeping!

- Wouldn’t it be great if we could:

`while` loops are annoying.

- Often, you want: "Do `some_thing()` 10 times"
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don't even care about `i`, it's just bookkeeping!

- Wouldn't it be great if we could:

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

`while` loops are annoying.

- Often, you want: “Do `some_thing()` 10 times”
- With a while loop you need to:

```
i = 0
while i < 10:
    some_thing()
    i += 1
```

I don’t even care about `i`, it’s just bookkeeping!

- Wouldn’t it be great if we could:

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

We can!

Revisiting Repetition

```
for var_name in sequence:
    codeblock
```

- balance3.py - rewrite yearly bank account balance with a for loop
- Average of 100 random numbers

`while` vs `for`

Task: Generate and print random integers between 1 and 10 (inclusive) until one of the random numbers exceeds 8.

Would you use a `for` loop or a `while` loop?

`while` vs `for`

Task: Ask the user for a number (**n**), then print 100 random numbers between 0 and **n**.

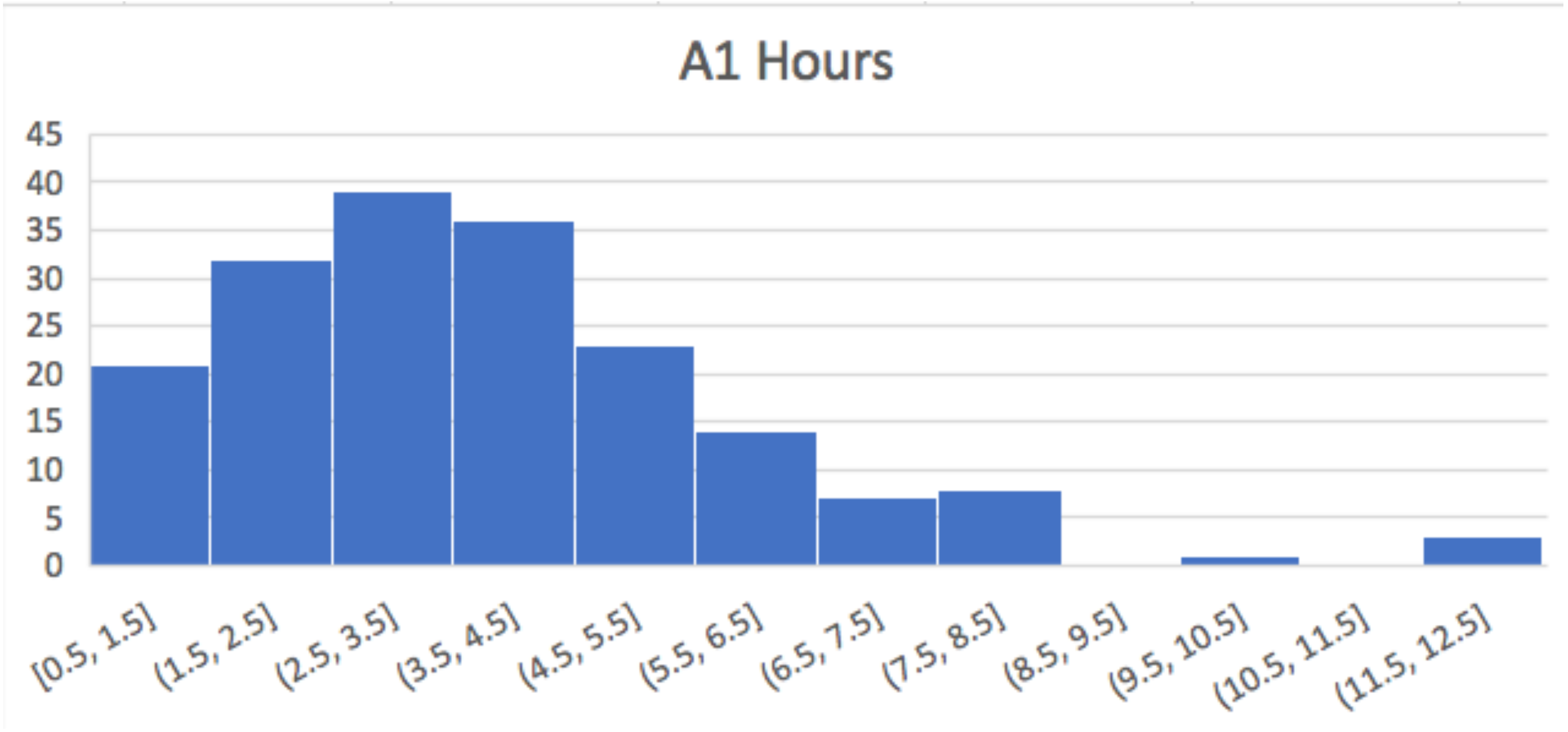
Would you use a `for` loop or a `while` loop?

`while` vs `for`

Task: Sum the numbers from 1 to 340, leaving out those divisible by 5.

Would you use a `for` loop or a `while` loop?

A1 debrief


A1 Hours
45
40
35
30
25
20
15
10
5
0
[0.5, 1.5]
(1.5, 2.5]
(2.5, 3.5]
(3.5, 4.5]
(4.5, 5.5]
(5.5, 6.5]
(6.5, 7.5]
(7.5, 8.5]
(8.5, 9.5]
(9.5, 10.5]
(10.5, 11.5]
(11.5, 12.5]