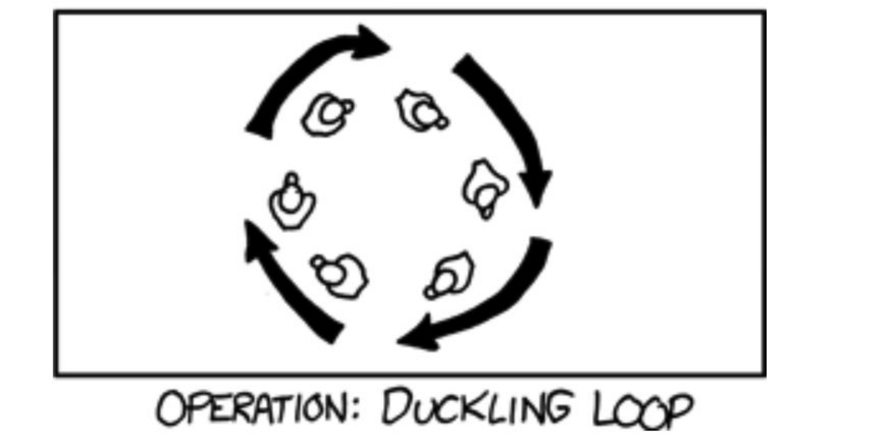
CS 1: Intro to CS

Wrapping up Modules and Lists; Loops and Intro to Conditionals



Administrivia

MP 2 is out, due Thursday 11:30PM

HW1 feedback is published (read through your TAs feedback, even -0's!)

Tomorrow's lab will cover lists and loops

Agenda

- More about lists and modules
- Loops
  - The `for` statement
- Decision-making with conditionals
  - The `if` statement
  - The `else` and `elif` statements

Note: Class went through lists/modules in depth with posted code, recording is posted to supplement loops/conditionals

Learning Objectives

- Be able to define, access, and modify list sequences
- Motivate the importance of loops and **iterative** programming
- Know how to evaluate boolean expressions
- Be able to use boolean expressions to conditionally-execute code with for if statements
- Identify the difference between if, elif, and else conditional statements (more next time)

Returning to List Code Demo

There’s quite a lot we can do with lists, so let’s jump into some code to explore…

See **`.py`** files under today’s lecture for the code and some practice exercises!

- **`loops_lists.py`** (list demo code)
- **`extra_list_practice.py`** (3 practice exercises)
- **`duck_loop.py`** (complete duck loop program with emoji module, previewing loops for Monday)

We have also provided a **`Lec05Lists.java`** analog to **`lec05_lists.py`** (you aren't expected to know the Java code, but students have shared it's been helpful to compare the two languages!)

Loops

So far, we’ve seen multiple kinds of data

- Ints, floats, strings, lists

We’ve also learned how to write functions with def and return

Today, we introduce another fundamental concept: **a loop**

Loops

Code that executes repeatedly

Python has two kinds of loop statements:

- **`for`** loops (this lecture)
- **`while`** loops (next lecture)

Loops over lists

Loops are often associated with lists

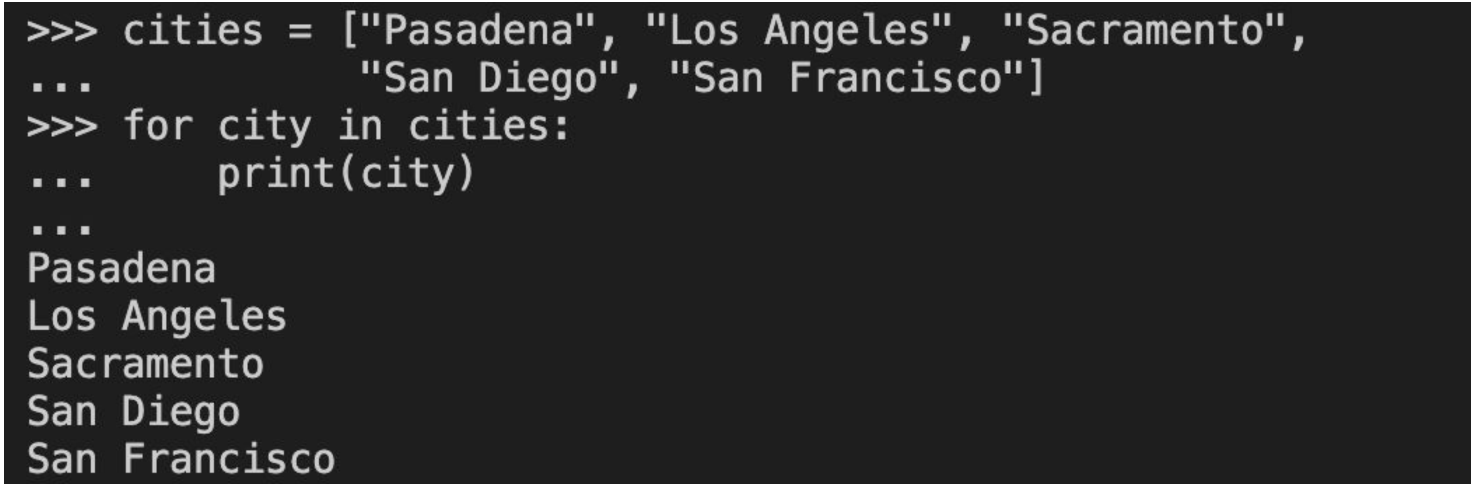
Basic idea:

- For each element of the list
- Do the following … [chunk of code]

Example:

- For each element of a list
- Print the element

Example



```
>>> cities = ["Pasadena", "Los Angeles", "Sacramento",
...           "San Diego", "San Francisco"]
>>> for city in cities:
...     print(city)
...
Pasadena
Los Angeles
Sacramento
San Diego
San Francisco
```

Structure of a `for` Loop

```
for <name> in <list>:
    <chunk of code>
```

Similar to def and return, for and in are keywords (reserved words)

Chunk of code is repeated for each element in the list

Each time though, the next element of `<list>` is assigned to `<name>` and `<chunk of code>` is executed

The chunk of code is called a **block**

Technical Note

```
for <name> in <list>:
    <block>
```

Technically, the thing that comes after `in` does not have to be a list

It can be any Python value that is *iterable*

We will explain this in more detail later

The `+=` operator and friends

When you see a line in the form

```
x = x + 10
```

You can write

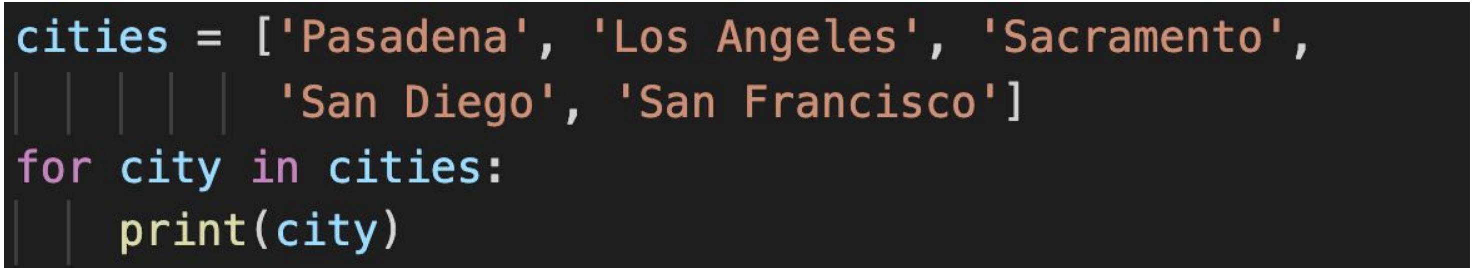
```
x += 10
```

Many operators op have op= counterparts

- `+=`, `-=`, `*=`, `/=`, `&=`

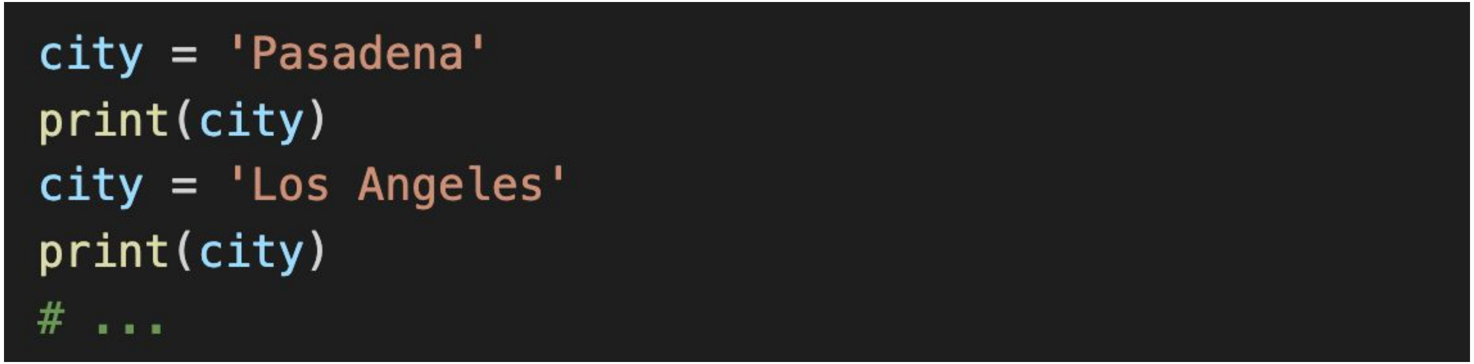
You should use them where applicable as they male code more concise and readable

Unravelling the Loop



```
cities = ['Pasadena', 'Los Angeles', 'Sacramento',
          'San Diego', 'San Francisco']
for city in cities:
    print(city)
```

Is equivalent to:



```
city = 'Pasadena'
print(city)
city = 'Los Angeles'
print(city)
# ...
```

`for` *`item`* `in` *`lst`* Syntax

The variable name for the *item* does not matter:

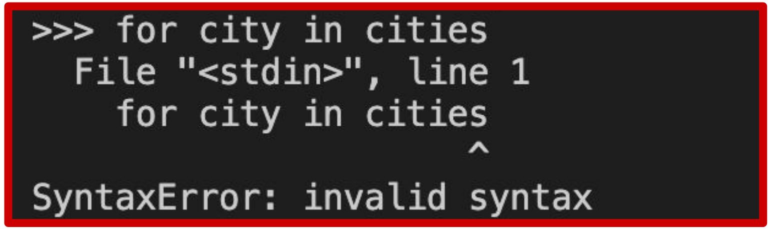
```
for foo in cities:
    print(foo)
```

But we like to use descriptive variable names, so the following equivalent is preferred:

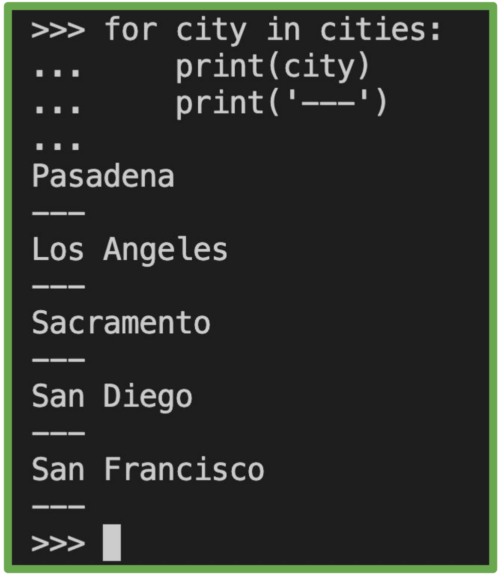
```
for city in cities:
    print(city)
```

Loop Syntax Rules

1. Must have a colon `(:)` at the end of the ***`for item in lst`*** declaration



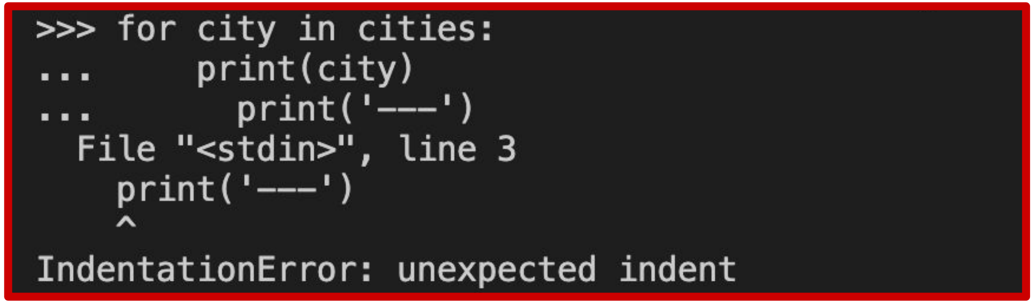
```
>>> for city in cities
  File "<stdin>", line 1
    for city in cities
                     ^
SyntaxError: invalid syntax
```



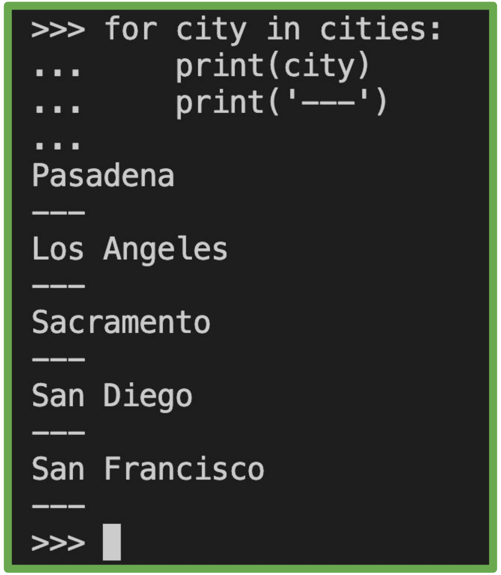
```
>>> for city in cities:
...     print(city)
...     print('---')
...
Pasadena
---
Los Angeles
---
Sacramento
---
San Diego
---
San Francisco
---
>>> 
```

Loop Syntax Rules

2. Every line in a block must be indented the same amount, otherwise an error occurs



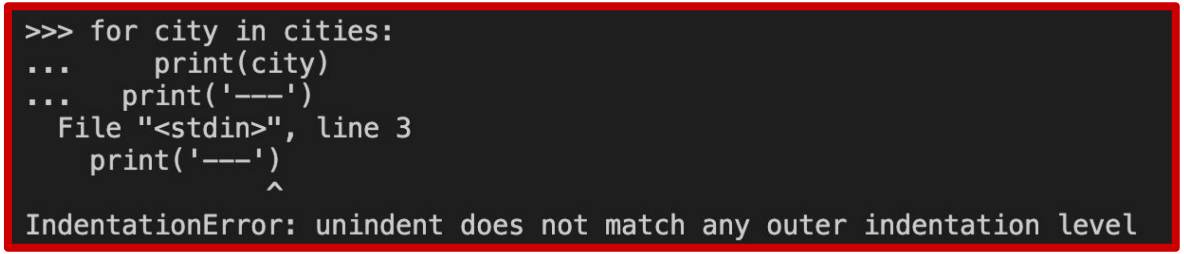
```
>>> for city in cities:
...     print(city)
...       print('---')
  File "<stdin>", line 3
    print('---')
    ^
IndentationError: unexpected indent
```



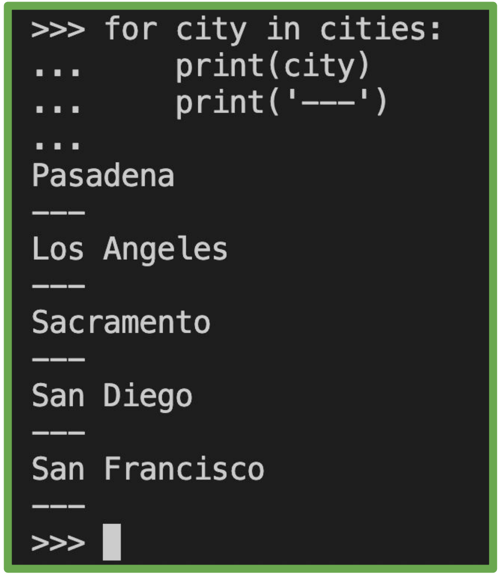
```
>>> for city in cities:
...     print(city)
...     print('---')
...
Pasadena
---
Los Angeles
---
Sacramento
---
San Diego
---
San Francisco
---
>>> 
```

Loop Syntax Rules

3. The end of the block is indicated when indent goes back to what it was before the **for** loop began



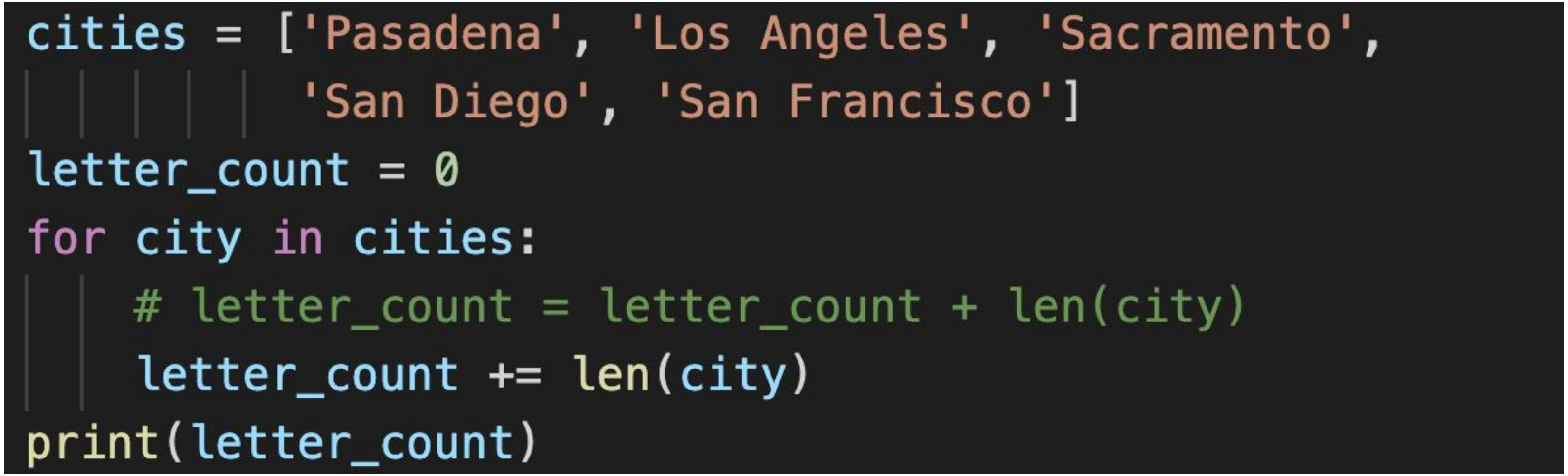
```
>>> for city in cities:
...     print(city)
...   print('---')
  File "<stdin>", line 3
    print('---')
               ^
IndentationError: unindent does not match any outer indentation level
```



```
>>> for city in cities:
...     print(city)
...     print('---')
...
Pasadena
---
Los Angeles
---
Sacramento
---
San Diego
---
San Francisco
---
>>> █
```

Application: Summing word lengths

Here is the code we practiced at the end of Friday's lecture:



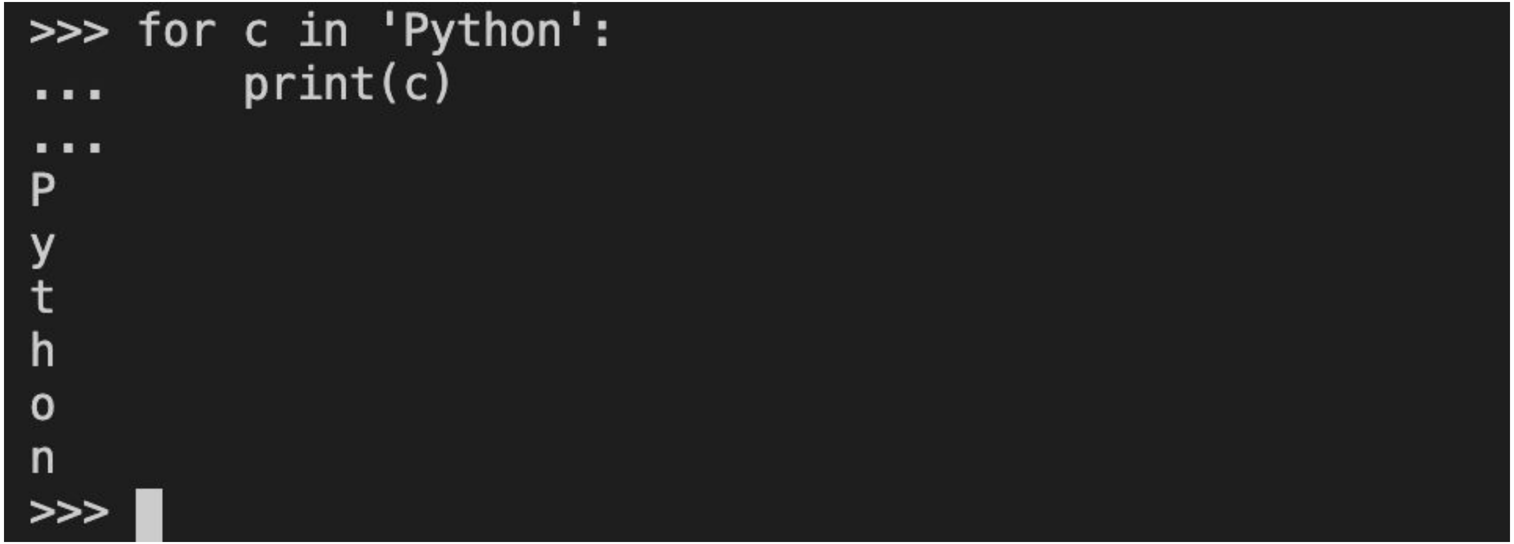
```python
cities = ['Pasadena', 'Los Angeles', 'Sacramento',
          'San Diego', 'San Francisco']
letter_count = 0
for city in cities:
    # letter_count = letter_count + len(city)
    letter_count += len(city)
print(letter_count)
```

Output:

```
51
```

Loops and Strings

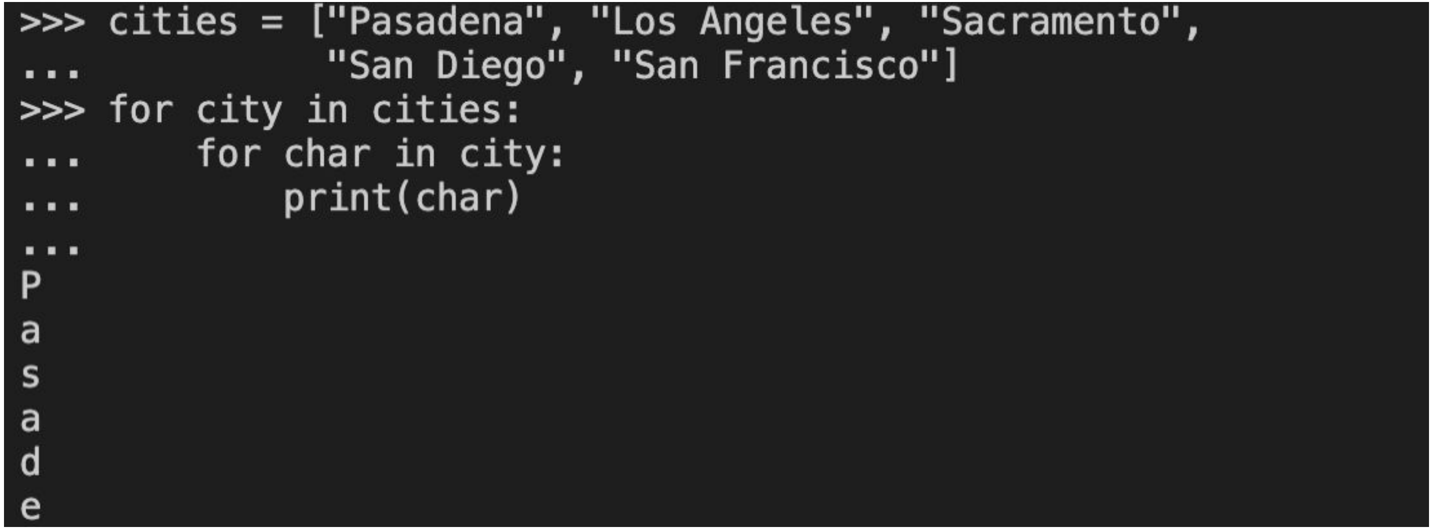
We can also use a **`for`** loop to loop over characters of a string (a string is iterable!)



```
>>> for c in 'Python':
...     print(c)
...
P
y
t
h
o
n
>>> 
```

Nested Loops

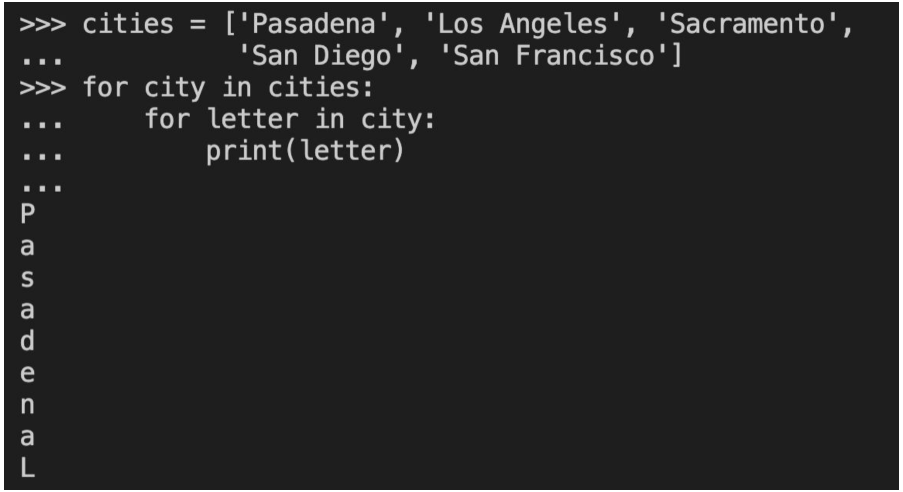
Can nest one for loop inside another:



```
>>> cities = ["Pasadena", "Los Angeles", "Sacramento",
...           "San Diego", "San Francisco"]
>>> for city in cities:
...     for char in city:
...         print(char)
...
P
a
s
a
d
e
```

etc.

Nested Loops



```
>>> cities = ['Pasadena', 'Los Angeles', 'Sacramento',
...           'San Diego', 'San Francisco']
>>> for city in cities:
...     for letter in city:
...         print(letter)
...
P
a
s
a
d
e
n
a
L
```

First time through outer loop: `city` is 'Pasadena'

Inner loop: `char` is 'P', then 'a', etc.

Second time through outer loop: `city` is 'Los Angeles'

Inner loop: `char` is 'L', then 'o', etc.

Check Your Understanding

What is the output of the following code?

```
for i in [1, 2, 3]:
    for j in [1, 2, 3]:
        print(i + j)
```

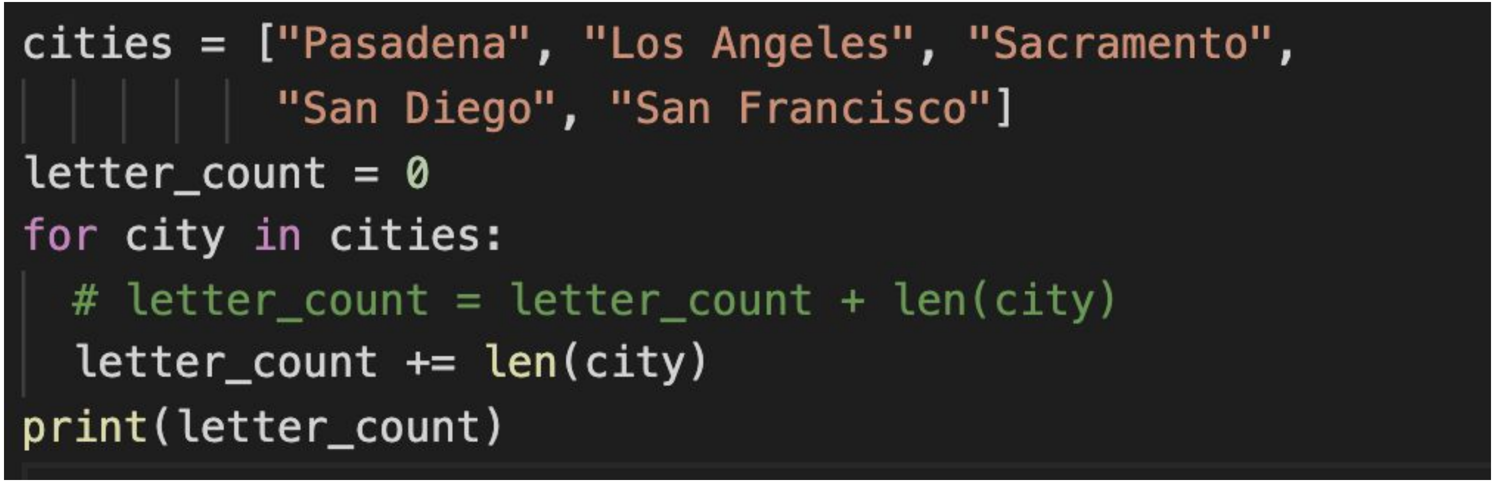
Check Your Understanding

What is the output of the following code (from Lecture Check)?

```
nums = [1, 2, 1]
chars = ['^', '_', '^']
for n in nums:
  result = ''
  for c in chars:
    result += (c * n)
  print(result)
```

Application: Summing word lengths

Another way to do this:



```
cities = ["Pasadena", "Los Angeles", "Sacramento",
          "San Diego", "San Francisco"]
letter_count = 0
for city in cities:
  # letter_count = letter_count + len(city)
  letter_count += len(city)
print(letter_count)
```

Output:
51

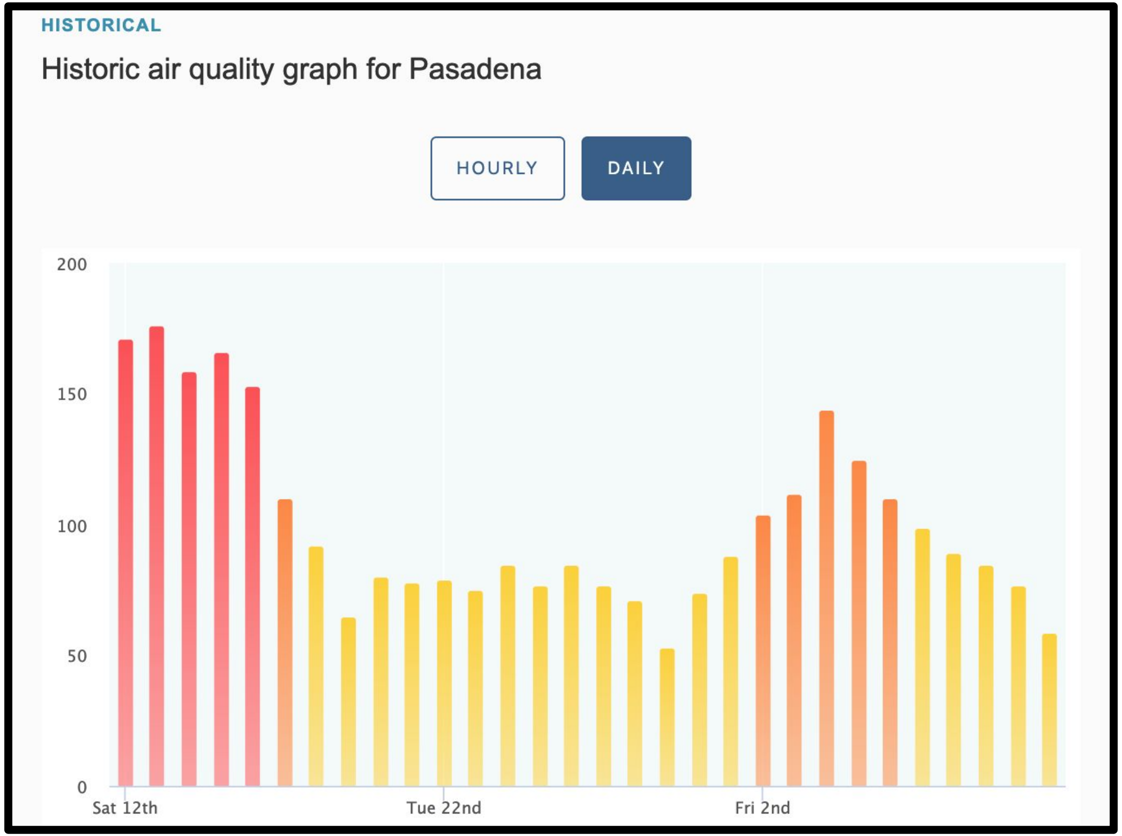
Decision Making in Programs

So far, our programs have always done the same thing no matter what

But there are many scenarios where programs need to handle decision-making based on certain conditions.

What are some examples of problems you might need to make different decisions based on data?

An Application Close to Home...

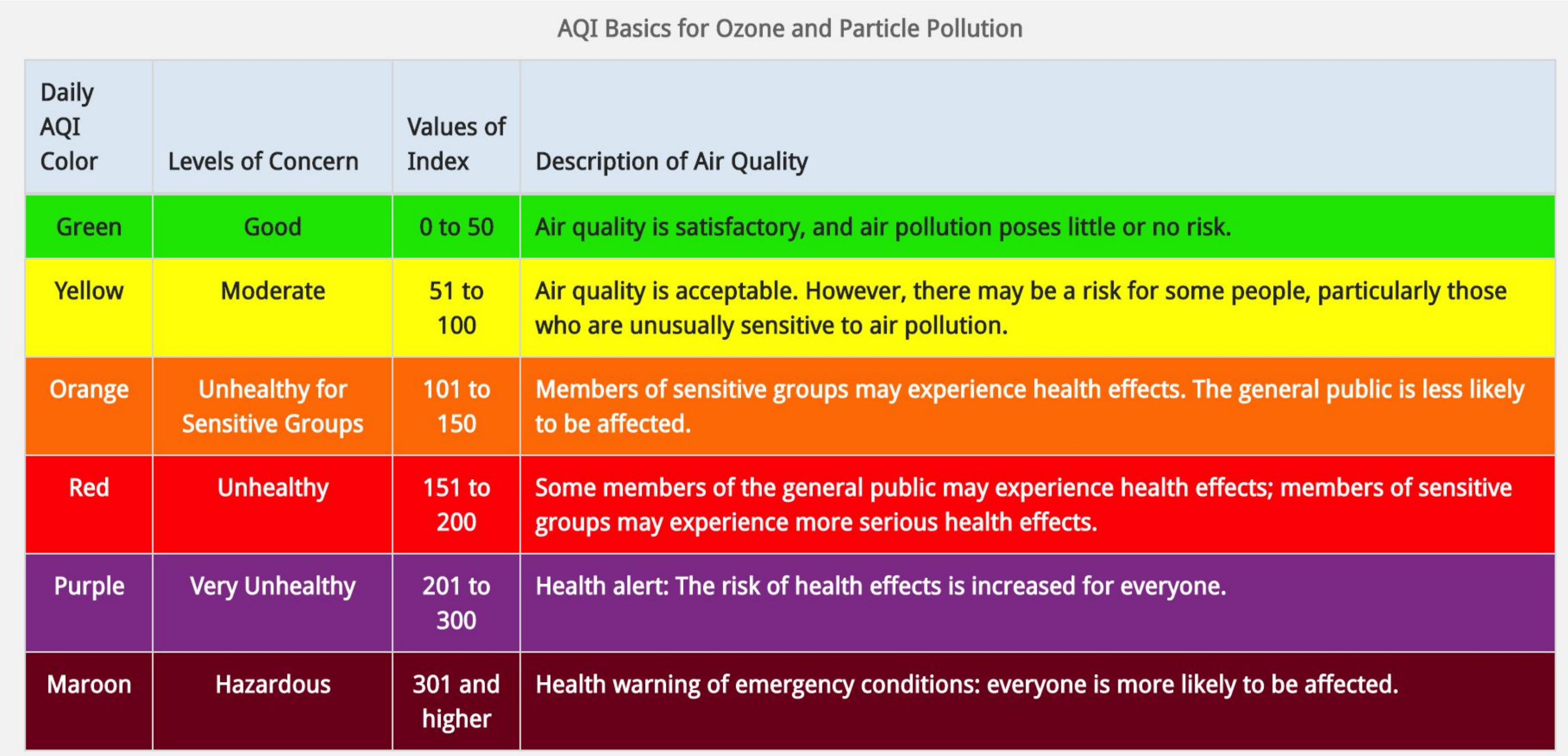


HISTORICAL

Historic air quality graph for Pasadena

Source:
https://www.iqair.com/usa/california/pasadena

When is an AQI Unhealthy?



AQI Basics for Ozone and Particle Pollution

Daily AQI Color	Levels of Concern	Values of Index	Description of Air Quality
Green	Good	0 to 50	Air quality is satisfactory, and air pollution poses little or no risk.
Yellow	Moderate	51 to 100	Air quality is acceptable. However, there may be a risk for some people, particularly those who are unusually sensitive to air pollution.
Orange	Unhealthy for Sensitive Groups	101 to 150	Members of sensitive groups may experience health effects. The general public is less likely to be affected.
Red	Unhealthy	151 to 200	Some members of the general public may experience health effects; members of sensitive groups may experience more serious health effects.
Purple	Very Unhealthy	201 to 300	Health alert: The risk of health effects is increased for everyone.
Maroon	Hazardous	301 and higher	Health warning of emergency conditions: everyone is more likely to be affected.

Analyzing AQI

Suppose we wanted to write a program to know which days of a week (let’s say, Sept. 12th to 19th 2020 in Pasadena) have “unhealthy” air quality, as determined by the EPI air quality index (AQI)

How might we write code to count how many AQI values are above 150?

Need to break down into two subproblems:

1. How do we *test* to see whether or not a particular AQI is greater than 150?
2. How do we *use* that information to control our program?

Relational Operators

To test a number against another number, we need a **relational operator**

- Examples: `<`, `<=`, `>`, `>=`, `==`

Relational operators return a boolean value (`True` or `False`)

Relational Operators

x == y (is x equal to y?)

x != y (is x not equal to y?)

x < y (is x less than y?)

x <= y (is x less than or equal to y?)

x > y (is x greater than y?)

x >= y (is x greater than or equal to y?)

== VS. =

Note: The == operator is completely different from the = (assignment) operator

- It’s very easy to mix these up when first learning them!

```
a = 10  # assign a the value 10
a == 10 # is a equal to 10?
```

Testing the AQI

For the first part of our problem, we need to test if an AQI value is greater than 100 (unhealthy for sensitive groups to go outside)

```
>>> aqi = 161
>>> aqi > 150
True
```

The `if` statement

Using the condition, we can use an **`if`** statement to:

- Execute a block of code if some condition is true
- Otherwise do nothing

```
>>> aqi = 161
>>> if aqi > 150:
        print('It\'s unhealthy to go outside!')
It's unhealthy to go outside!
```

Structure of an `if` statement

```
if <boolean expression>:
    <block of code>
```

Like a `for` loop, `if` statements:

- Colon (`:`) must come at end of if line
- Block of code can consist of multiple lines (with correct indentation)

Interpreting an `if` statement

```
if <boolean expression>:
    <block of code>
```

- If the `<boolean expression>` evaluates to `True`, then execute the `<block of code>`
- Otherwise, don’t

In either case, continue by executing the code after the `if` statement

Back to our Problem

For any given AQI value, we now know how to compare it with 150 and do something based on the result

Since we have a whole list of items, we will need a `for` loop as well!

Also need to keep track of number of AQI values seen so far which exceed 150

Back to our Problem: Pseudocode

```
Initialize a count of AQI values over 150 to 0

For each value in our list:
    ●  If the value is greater than 150 (unhealthy) update our counter by 1

Print the number of values found to the console
```

Back to our Problem: Pseudocode

```
Initialize a count of AQI values over 150 to 0
For each value in our list:
    ● If the value is greater than 150 (unhealthy) update our counter by 1
Print the number of values found to the console
```

```
aqis_above_150 = 0
```

Back to our Problem: Pseudocode

```
Initialize a count of AQI values over 150 to 0
For each value in our list:
    If the value is greater than 150 (unhealthy) update our counter by 1
Print the number of values found to the console
```

```
aqis_above_150 = 0
for value in aqis:
```

Back to our Problem: Pseudocode

Initialize a count of AQI values over 150 to 0

For each value in our list:

- **If the value is greater than 150 (unhealthy) update our counter by 1**

Print the number of values found to the console

```
aqis_above_150 = 0
for value in aqis:
    if value > 150:
        aqis_above_150 += 1
```

Back to our Problem: Pseudocode

Initialize a count of AQI values over 150 to 0

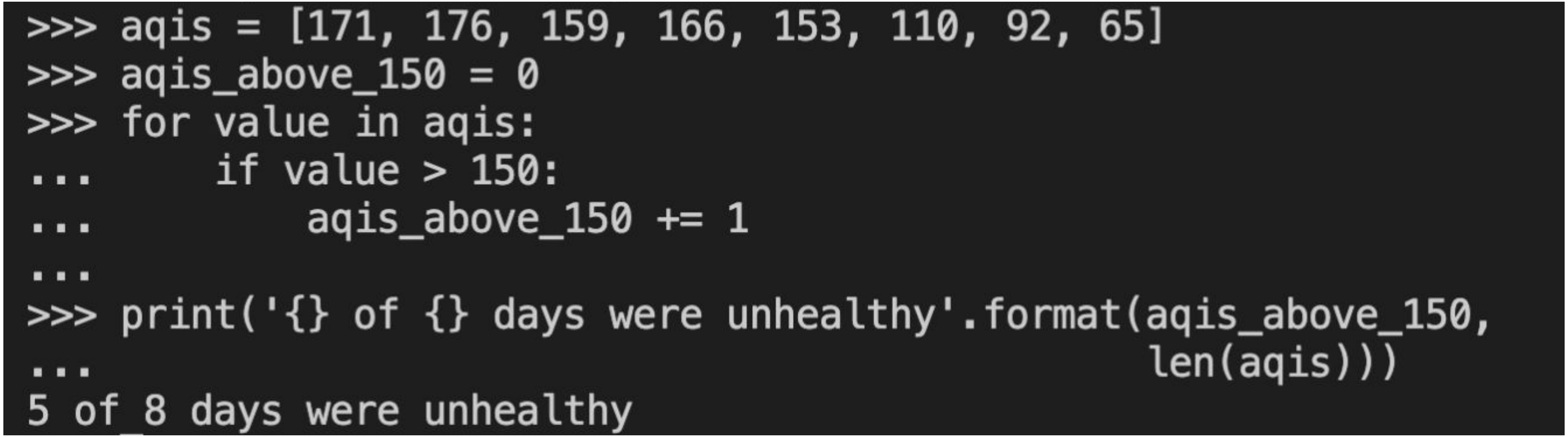
For each value in our list:

- If the value is greater than 150 (unhealthy) update our counter by 1

Print the number of values found to the console

```
aqis_above_150 = 0
for value in aqis:
    if value > 150:
        aqis_above_150 += 1
print(f'{aqis_above_150} of {len(aqis)} days were unhealthy')
```

Testing it Out



```
>>> aqis = [171, 176, 159, 166, 153, 110, 92, 65]
>>> aqis_above_150 = 0
>>> for value in aqis:
...     if value > 150:
...         aqis_above_150 += 1
...
>>> print('{} of {} days were unhealthy'.format(aqis_above_150,
...                                             len(aqis)))
5 of 8 days were unhealthy
```

More Decisions

Recall that using an condition, we can use an **if** statement to:

- Execute a block of code if some condition is true
- Otherwise do **nothing**

What if we instead want to do something **else** when the condition isn’t true?

Practice On Your Own: `generate_email` 2.0

Recall the **`generate_email`** function from last week. Let’s take what we’ve learned so far and generalize the program to take a list of 2-value lists (**`[firstname, lastname]`**) and return a new list of Caltech email addresses. The list argument should remain unchanged.

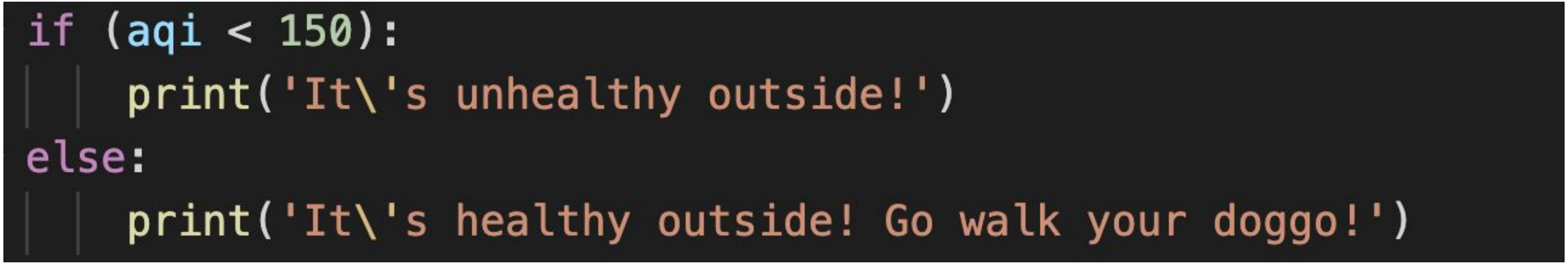
This is a good exercise to practice using a helper function within another function!

Extending Conditionals: `if` and `else`

An **`if`** statement can optionally include a second part called the **`else`** clause, which is executed only if the boolean expression in the **`if`** statement evaluates to **`False`**

```
if <boolean expression>:
    <block of code>
else:
    <different block of code>
```

`if` and `else` with our AQI example



```
if (aqi < 150):
    print('It\'s unhealthy outside!')
else:
    print('It\'s healthy outside! Go walk your doggo!')
```

Multi-way Tests

```
aqi = 161
if aqi < 150:
    aqis_below_150 += 1
if aqi == 150
    aqis_at_150 += 1
if aqi > 150
    aqis_above_150 += 1
```

What’s wrong with this code?

Multi-way Tests

The problem:

- For many aqi values, some of the three **`if`** statement conditions may be evaluated unnecessarily
- We can use **`else`** to handle this

Second Try

Use **else**:

```
aqi = 161
if aqi < 150:
    aqis_below_150 += 1
else:
    if aqi == 150:
        aqis_at_150 += 1
    else:
        aqis_above_150 += 1
```

This works and is efficient, but nested **if**s like this are not very readable...

Third Try

How would we say this in English?

“*If* the aqi is less than 150, do <thing1>, *else* if the aqi is 150, do <thing2>, *else* do <thing3>”

We can express this in Python using an **`elif`** statement inside an **`if`** (**`elif`** is short for “else if”)

Third Try

This leads to:

```
aqi = 161
if aqi < 150:
    aqis_below_150 += 1
elif aqi == 150:
    aqis_at_150 += 1
else:
    aqis_above_150 += 1
```

This is both efficient *and* readable!

Using Conditions in `while` Loops

So far, we've introduced iterative programming with the **`for`** loop over lists, strings, and range sequences

Sometimes:

- We’re not working with lists or strings
- We don’t have a fixed number of things to loop over
- We don’t know in advance how many times we will have to loop

`for` vs. `while` Loops

Structure of a **`for`** loop:

```
for <item> in <seq>:
    <chunk of code>
```

Structure of a **`while`** loop:

```
while <boolean expression>:
    <block of code>
```

The **`while`** loop shares similar syntax rules with **`if`** and **`for`** statements (e.g. **`:`** at end, indentation rules)

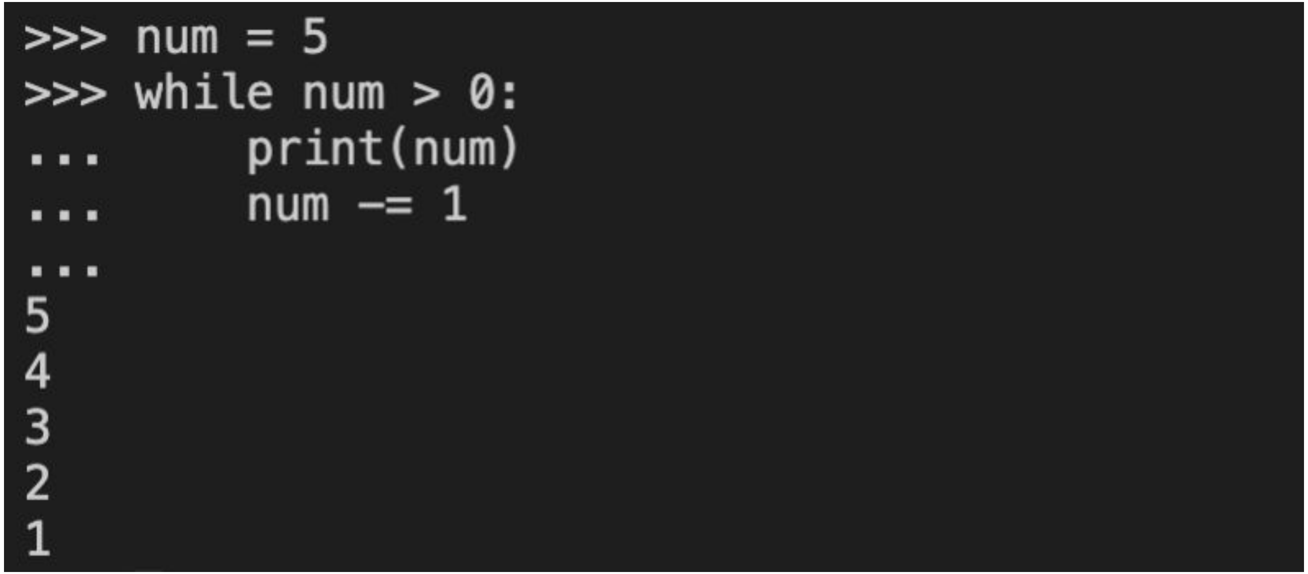
Evaluation of the `while` loop

```
while <boolean expression>:
    <block of code>
```

1. Evaluate the `<boolean expression>`
2. If it evaluates to **`True`**, execute `<block of code>` and repeat from the beginning
3. Otherwise, continue with the next line after the **`while`** loop

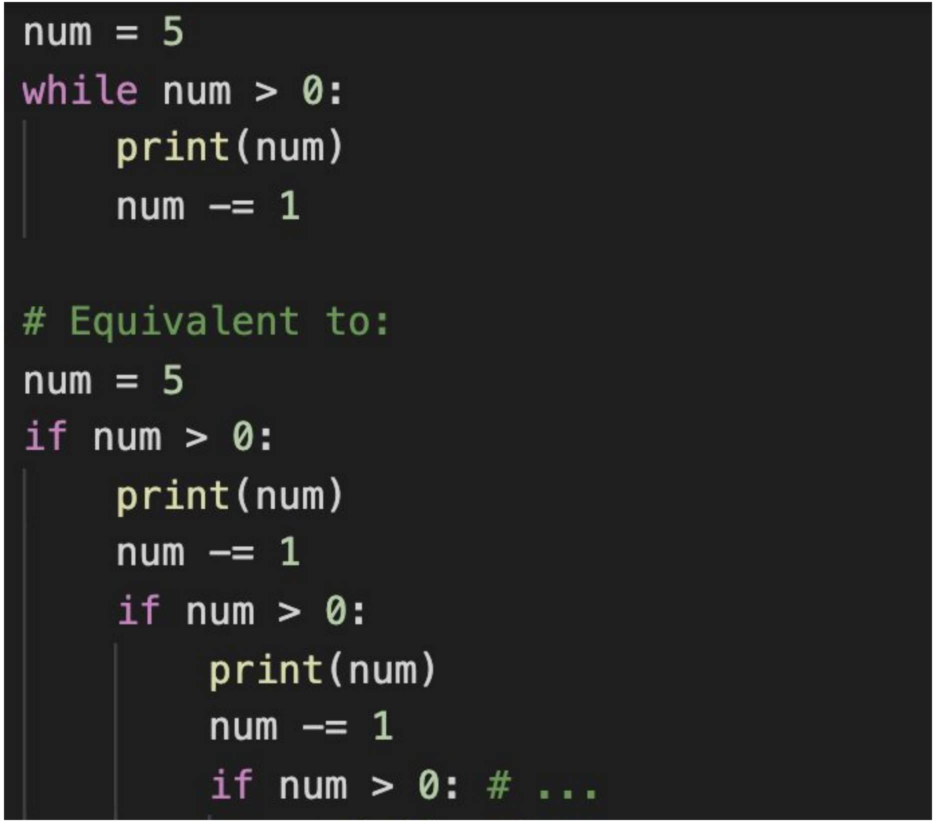
Example

Starting at the number 5, print all the numbers from 5 down to 1



```
>>> num = 5
>>> while num > 0:
...     print(num)
...     num -= 1
...
5
4
3
2
1
```

Unravelling the `while` Loop



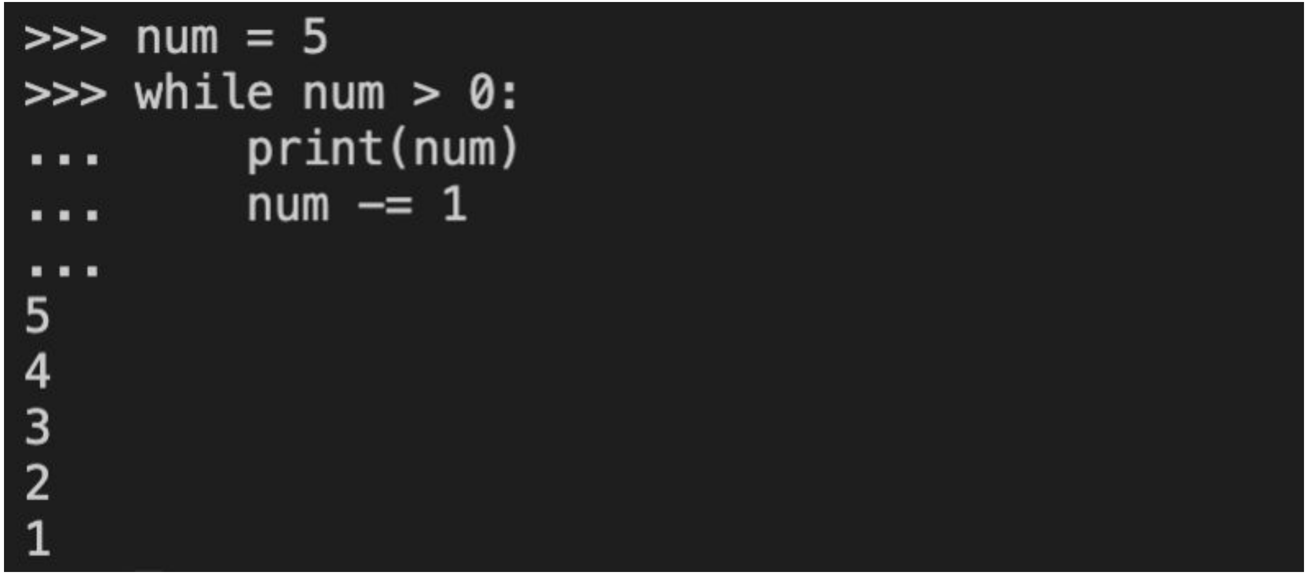
```
num = 5
while num > 0:
    print(num)
    num -= 1

# Equivalent to:
num = 5
if num > 0:
    print(num)
    num -= 1
    if num > 0:
        print(num)
        num -= 1
        if num > 0: # ...
```

The `while` loop is *much* cleaner!

`while` loop vs. for `loop`

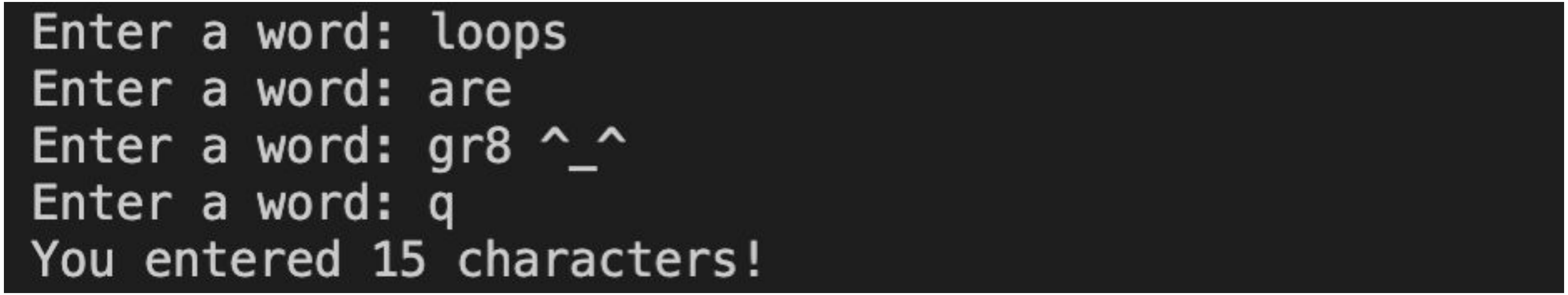
We could also write this with a `for` loop. How?



```
>>> num = 5
>>> while num > 0:
...     print(num)
...     num -= 1
...
5
4
3
2
1
```

Another Example

Use **`input`** to read words and print them, stopping when “Q” or “q” is read (for “Quit”)

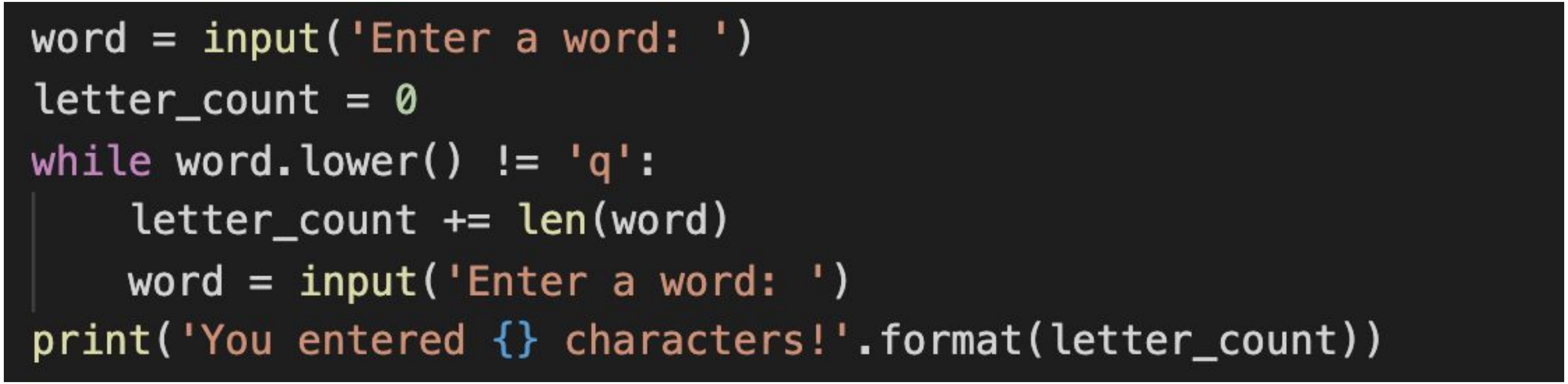


```
Enter a word: loops
Enter a word: are
Enter a word: gr8 ^_^
Enter a word: q
You entered 15 characters!
```

In this case, we **cannot know** how many times we will have to go through the loop

This is a much more natural situation in which to use a **`while`** loop...

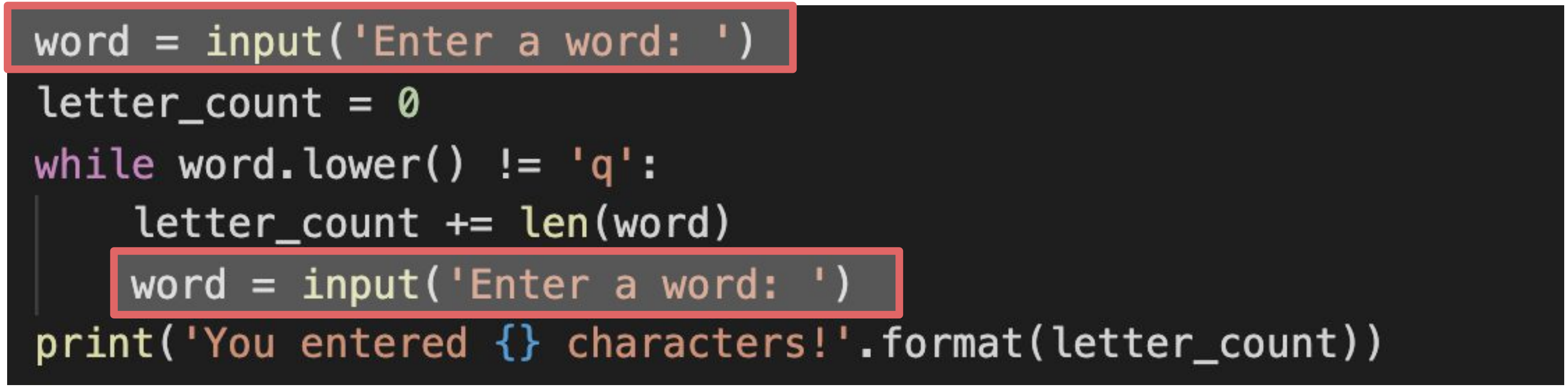
Not the best solution...



```
word = input('Enter a word: ')
letter_count = 0
while word.lower() != 'q':
    letter_count += len(word)
    word = input('Enter a word: ')
print('You entered {} characters!'.format(letter_count))
```

What’s a code quality issue here?

Not the best solution...



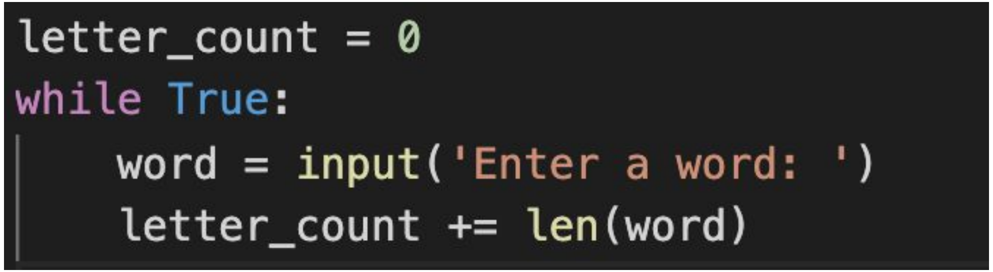
```python
word = input('Enter a word: ')
letter_count = 0
while word.lower() != 'q':
    letter_count += len(word)
    word = input('Enter a word: ')
print('You entered {} characters!'.format(letter_count))
```

Redundancy!

Programming principle: “DRY” (Don’t Repeat Yourself)

Repeated code usually means there’s a better way to write the code (just like we’ve seen with functions)

A First Attempt

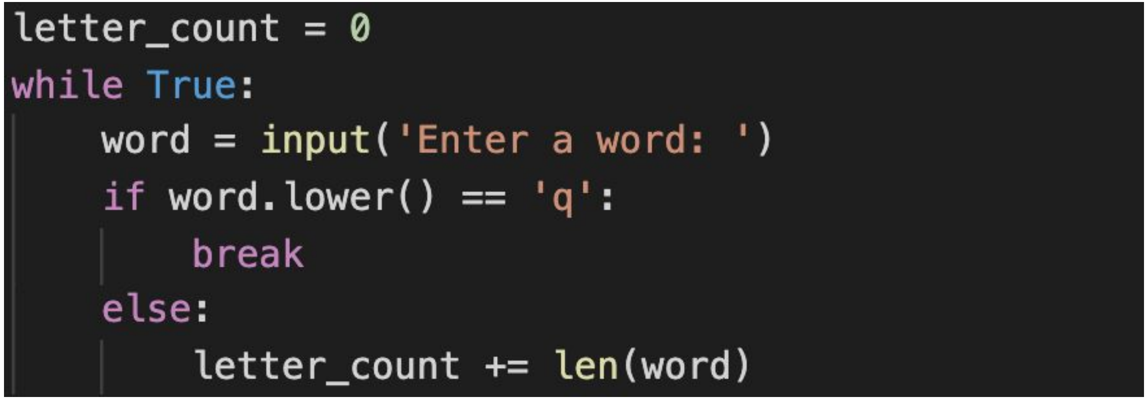


```
letter_count = 0
while True:
    word = input('Enter a word: ')
    letter_count += len(word)
```

No more redundancy!

But what’s an issue with this solution?

A Better Attempt with break



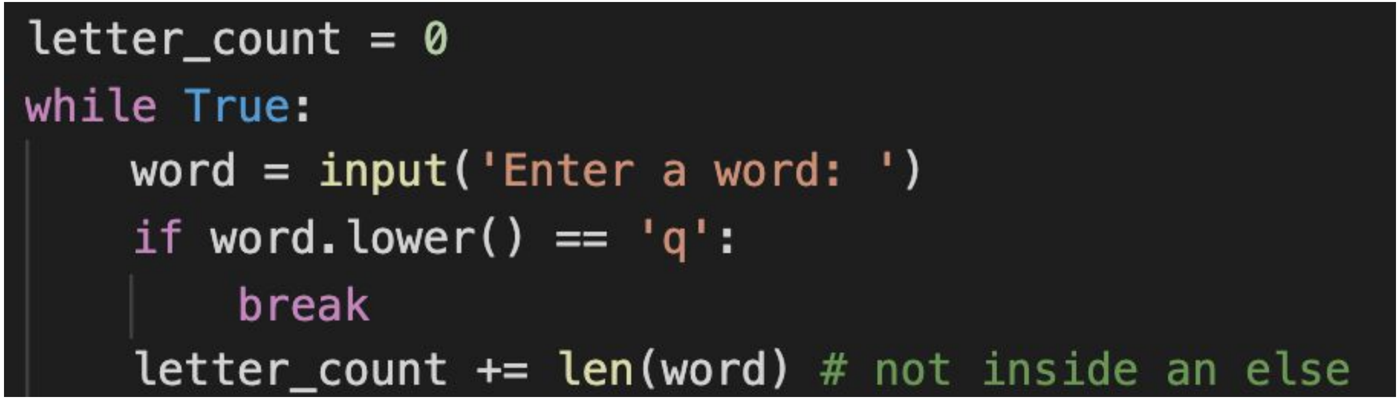
```
letter_count = 0
while True:
    word = input('Enter a word: ')
    if word.lower() == 'q':
        break
    else:
        letter_count += len(word)
```

A **break** statement directs Python to stop and exit the loop, continuing the rest of the program

Helps follow the DRY principle, both reducing redundancy and ensuring the program halts

An Even Better Attempt

Can also write:



```
letter_count = 0
while True:
    word = input('Enter a word: ')
    if word.lower() == 'q':
        break
    letter_count += len(word) # not inside an else
```

Why is this ok?

This code is preferable to the previous version, as the **`else`** is unnecessary

We don’t always need a `break`...

`break` statements are not needed very often, and are never “necessary” (can always re-write without using **`break`**)

But when a test *naturally* fails in the middle of a loop body, **`break`** can make code much cleaner

A good rule of thumb is to write a solution without **`break`** until it's motivated enough to reduce redundancy (without being "a hack")

Practice Problems

while_mystery_3

for_to_while

dice_sum

flip_coin_three_heads

guessing_game

guess_2d

Coming Attractions

On Friday, we will introduce file processing:

- Types of files and applications
- Processing files as **lists** of **strings** (lines ending with '\n')
  - Using **loops**!
- Overview of common pitfalls of file IO
  - Edge cases: empty files, first line, last line
  - What does it mean to inefficiently read a file?
  - How do we write a new file? Update an existing one without wiping the original contents?
  - How do we work with multiple files at once?
  - Why is it important to read line-by-line vs. loading all the lines at once?