Python Bootcamp 3

This is a Jupyter Notebook. To run a gray code cell, click in the cell and either click on the "Run" arrow, or type shift+enter (or shift+return on a Mac).

Lists - Compound data type

allow us to work with multiple items at once

A list is a collection of objects kept in a *strict* order. A list is designated by square brackets. Items in the list are separated by commas.

Other compound data types include: tuples, strings, dictionaries

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

A list can contain any type of object:

```
grades = [93, 75, 80, 98, 100, 64, 88]

students = ["Casey", "Taylor", "Lee", "Luca", "Quinn", "Ruka"]
```

A list can also contain only one item:

```
final_grade = [90]
```

A list can be empty:

```
new_grades = []

new_grades
```

```
[]
```

Order matters: list_a and list_b are **not** the same list

```
list_a = [1, 2, 3]
list_b = [3, 2, 1]
```

A list can contain mixed data types:

```
mixed_list = [1, 2, 3, "Hello", 2.3, True]

mixed_list
```

```
[1, 2, 3, 'Hello', 2.3, True]
```

You can even have a list of lists, or a list within a list, called a **nested list**:

```
gradebook = ["Hello", 3214, ["Casey", "Taylor", "Lee"], [93, 75, 80]]

gradebook
```

```
['Hello', 3214, ['Casey', 'Taylor', 'Lee'], [93, 75, 80]]
```

Access Nested List Items by Index

You can access individual items in a nested list using multiple indexes.

The indexes for the items in a nested list are illustrated as below:

```
gradebook_nested = ["Hello", 3214, ["Casey", "Taylor", "Lee"], [93,
75, 80]]
# how to output "Hello", "Taylor", 80?

print(gradebook_nested[2])
print(gradebook_nested[2][1])
print(gradebook_nested[3][2])
```

```
['Casey', 'Taylor', 'Lee']
Taylor
80
```

Try to create some lists on your own:

```

```

Difference between List and Array in Python

List: A list in Python is a collection of items which can contain elements of multiple data types, which may be either numeric, character logical values, etc. It is an ordered collection supporting negative indexing. A list can be created using [] containing data values. Contents of lists can be easily merged and copied using python's inbuilt functions.

```
# creating a list containing elements
# belonging to different data types
sample_list = [1,"Yash",['a','e']]
print(sample_list)
```

```
[1, 'Yash', ['a', 'e']]
```

The first element is an integer, the second a string and the third is an list of characters.

Array: An array is a vector containing homogeneous elements i.e. belonging to the same data type. Elements are allocated with contiguous memory locations allowing easy modification, that is, addition, deletion, accessing of elements. In Python, we have to use the array module to declare arrays. If the elements of an array belong to different data types, an error "Incompatible data types" is thrown.

```
# creating an array containing same data type elements
# arrayName = arr.array(code for data type, [array and its items])
# https://docs.python.org/3/library/array.html
import array as arr

sample_array = arr.array('u', "[1, [2, 3, 4], [5, 6]")
print(sample_array)
# how to change it to str?

# method 1
print((''.join(str(x) for x in sample_array[5:12])).replace(', ',''))
print(str_array, type(str_array))

# method 2
str_array2 =''.join(str(x) for x in sample_array[1:8:3])
print(str_array2, type(str_array2))
```

```
array('u', '[1, [2, 3, 4], [5, 6]')
234

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
Cell In[2], line 12
      8 # how to change it to str?
      9
     10 # method 1
     11 print((''.join(str(x) for x in sample_array[5:12])).replace(',
','',''))
---> 12 print(str_array, type(str_array))
     14 # method 2
     15 str_array2 =''.join(str(x) for x in sample_array[1:8:3])

NameError: name 'str_array' is not defined
```

```
# accessing elements of array
for i in sample_array:
     print(i)
```

```
[
1
,
```

```
2
,

3
]
```

```
import numpy as np

sample_array2 = np.array([1, 2, 3, 4])
print(sample_array2)
```

```
[1 2 3 4]
```

Here are the differences between List and Array in Python :

List	Array
Can consist of elements belonging to different data types	Only consists of elements belonging to the same data type
No need to explicitly import a module for declaration	Need to explicitly import a module for declaration
Cannot directly handle arithmetic operations	Can directly handle arithmetic operations
Can be nested to contain different type of elements	Must contain either all nested elements of same size
Preferred for shorter sequence of data items	Preferred for longer sequence of data items
Greater flexibility allows easy modification (addition, deletion) of data	Less flexibility since addition, deletion has to be done element wise
The entire list can be printed without any explicit looping	A loop has to be formed to print or access the components of array
Consume larger memory for easy addition of elements	Comparatively more compact in memory size

List functions

Let's learn some list functions. While we're learning list functions, we'll also learn a bit about how the list object works.

As a review, there are two types of functions:

1. those that take a list as an argument and do something *with* the list
2. method functions that follow the list and do something *to* the list

```
grades = [93, 75, 80, 98, 100, 64, 88]
```

The `len()` function that we learned with strings also works with lists:

```
len(grades)
```

```
7
```

```
len("93")
```

```
2
```

```
len([93, 98])
```

```
2
```

The `sort()` method:

```
grades.sort()
```

Nothing was returned.

```
print(grades)
```

```
[64, 75, 80, 88, 93, 98, 100]
```

Our list has changed order. **Unlike strings, list methods can change the list object even if you didn't save the changed object as a new variable.**

We can also pass a **keyword argument** to the `sort()` function to reverse the sort order of the list:

```
grades.sort(reverse=True)
print(grades)
```

```
[100, 98, 93, 88, 80, 75, 64]
```

A **keyword argument** is an argument that has a default value. If you want to use something other than the default value, you have to pass the function the keyword with the new value. So for `sort()` the default of the `reverse` argument is `False`. When we want to reverse it, we need to change `reverse` to `True` by passing it as an argument.

Be careful with list methods, as they might change your object!

The `append()` method will add a new item to the end of a list:

```
students = ["Casey", "Taylor", "Lee", "Luca", "Quinn", "Ruka"]
students
```

```
['Casey', 'Taylor', 'Lee', 'Luca', 'Quinn', 'Ruka']
```

```
students.append("Jo")
```

Just like all objects, you can return the variable value by either just running the name of the variable:

```
students
```

```
['Casey', 'Taylor', 'Lee', 'Luca', 'Quinn', 'Ruka', 'Jo']
```

Or by printing the variable:

```
print(students)
```

```
['Casey', 'Taylor', 'Lee', 'Luca', 'Quinn', 'Ruka', 'Jo']
```

The `sum()` function:

```
grades = [93, 75, 80, 98, 100, 64, 88]

sum(grades)
```

```
598
```

But you cannot sum a list of strings

```
#sum(students)
```

We can use comparison operators with lists, just like other objects:

```
students = ["Casey", "Taylor", "Lee", "Luca", "Quinn", "Ruka"]
grades = [93, 75, 80, 98, 100, 64, 88]

len(students) == len(grades)
```

```
False
```

```
len(students) < len(grades)
```

```
True
```

We can use some arithmetic operators with lists:

```
grades = [93, 75, 80, 98, 100, 64, 88]
more_grades = [70, 93]

grades = grades + more_grades
print(grades)
```

```
[93, 75, 80, 98, 100, 64, 88, 70, 93]
```

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

```
print('The list is ' + series)
```

You can't subtract lists!

```
#grades = grades - more_grades
#print(grades)

x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
y = [1, 3, 5, 7, 9]

#x-y

print(grades * 4)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[7], line 1
----> 1 print(grades / 4)

TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

```
grades
```

```
[93, 75, 80, 98, 100, 64, 88, 70, 93]
```

Min and max functions

```
min(grades)
```

```
64
```

```
max(grades)
```

```
100
```

We are going to use some functions in this notebook that aren't automatically loaded with Python. They are in a module that is included with Python, called `statistics`.

You should import any modules at the top of your notebook, even if you don't use them until much later in the notebook. This is so that people can see which modules they need as soon as they open the notebook, in case they need to **install** the module on their computer. For the sake of this exercise, we will import it here instead.

Run this cell to import the `statistics` module:

```
import statistics as s
```

To use a function/method from an imported module, you type the name of the module followed by `.` followed by the name of the function. Let's use `statistics.mean()`:

```
mean_grade = s.mean(grades)
print(mean_grade)
```

```
85.42857142857143
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[9], line 3
      1 mean_grade = s.mean(grades)
      2 print(mean_grade)
----> 3 grades.mean()

AttributeError: 'list' object has no attribute 'mean'
```

`statistics.median()`:

```
median_grade = stats.median(grades)
print(median_grade)
```

```
88
```

Use `?` to see documentation about the package

```
?stats
```

```
Type:        module
String form: <module 'statistics' from 'C:\\Program Files\\Python312\\Lib\\statistics.py'>
File:        c:\program files\python312\lib\statistics.py
Docstring:
Basic statistics module.

This module provides functions for calculating statistics of data, including
averages, variance, and standard deviation.

Calculating averages
--------------------

==================  ==================================================
Function            Description
==================  ==================================================
mean                Arithmetic mean (average) of data.
fmean               Fast, floating point arithmetic mean.
geometric_mean      Geometric mean of data.
harmonic_mean       Harmonic mean of data.
median              Median (middle value) of data.
median_low          Low median of data.
```

```
median_high          High median of data.
median_grouped       Median, or 50th percentile, of grouped data.
mode                 Mode (most common value) of data.
multimode            List of modes (most common values of data).
quantiles            Divide data into intervals with equal probability.
==================   ================================================

Calculate the arithmetic mean ("the average") of data:

>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625


Calculate the standard median of discrete data:

>>> median([2, 3, 4, 5])
3.5


Calculate the median, or 50th percentile, of data grouped into class
intervals
centred on the data values provided. E.g. if your data points are
rounded to
the nearest whole number:

>>> median_grouped([2, 2, 3, 3, 3, 4])  #doctest: +ELLIPSIS
2.8333333333...

This should be interpreted in this way: you have two data points in
the class
interval 1.5-2.5, three data points in the class interval 2.5-3.5, and
one in
the class interval 3.5-4.5. The median of these data points is
2.8333...


Calculating variability or spread
---------------------------------

==================   ============================================
Function             Description
==================   ============================================
pvariance            Population variance of data.
variance             Sample variance of data.
pstdev               Population standard deviation of data.
stdev                Sample standard deviation of data.
==================   ============================================

Calculate the standard deviation of sample data:

>>> stdev([2.5, 3.25, 5.5, 11.25, 11.75])  #doctest: +ELLIPSIS
```

```
4.38961843444...

If you have previously calculated the mean, you can pass it as the
optional
second argument to the four "spread" functions to avoid recalculating
it:

>>> data = [1, 2, 2, 4, 4, 4, 5, 6]
>>> mu = mean(data)
>>> pvariance(data, mu)
2.5


Statistics for relations between two inputs
-------------------------------------------

==================
=====================================================
Function            Description
==================
=====================================================
covariance          Sample covariance for two variables.
correlation         Pearson's correlation coefficient for two
variables.
linear_regression   Intercept and slope for simple linear regression.
==================
=====================================================

Calculate covariance, Pearson's correlation, and simple linear
regression
for two inputs:

>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> covariance(x, y)
0.75
>>> correlation(x, y)  #doctest: +ELLIPSIS
0.31622776601...
>>> linear_regression(x, y)  #doctest:
LinearRegression(slope=0.1, intercept=1.5)


Exceptions
----------

A single exception is defined: StatisticsError is a subclass of
ValueError.
```

Joining items in a list into one string

Occasionally, you will need to join the items in a list together into a string.

```
full_name_list = ["Bartholomew", "JoJo", "Simpson"]
```

The `join()` function is actually a string method. The string you start with is the string that you want to use to connect all the items in the list. Then you pass the list to the function as an argument:

```
",".join(full_name_list)
```

```
'Bartholomew,JoJo,Simpson'
```

```
"-".join(full_name_list)
```

```
'Bartholomew-JoJo-Simpson'
```

```
"AAA".join(full_name_list)
```

```
'BartholomewAAAJoJoAAASimpson'
```

You can use an empty string to connect all the items in the list with nothing in between:

```
"".join(full_name_list)
```

```
'BartholomewJoJoSimpson'
```

Changing a string into a list

Another common task is splitting a string into a list. This is also a string method, so let's load a string to practice with:

```
csv_line = "sample 1,5,24,864,NA,.4556,,,65"
```

You must pass the `split()` function an argument - what string do you want to use to separate the items for your list. Here we'll use a comma:

```
csv_line_list = csv_line.split(",")
print(csv_line_list)
```

```
['sample 1', '5', '24', '864', 'NA', '.4556', '', '', '65']
```

```
csv_line
```

```
'sample 1,5,24,864,NA,.4556,,,65'
```

```
csv_line2 = ","
```

```
csv_line_list2 = csv_line2.split(",")
print(csv_line_list2)
```

```
['', '']
```

List indexing

Indexing works just like it does in strings.

```
students = ["Casey", "Taylor", "Lee", "Luca", "Quinn", "Ruka"]
```

```
students[0]
```

```
'Casey'
```

```
students[-1]
```

```
'Ruka'
```

```
students[1:] # again, index 4 is excluded
```

```
['Taylor', 'Lee', 'Luca', 'Quinn', 'Ruka']
```

We can change the value of individual items in a list:

```
students
```

```
['Casey', 'Taylor', 'Lee', 'Luca', 'Quinn', 'Ruka']
```

```
students[1] = "Tyler"
```

```
students
```

```
['Casey', 'Tyler', 'Lee', 'Luca', 'Quinn', 'Ruka']
```

Another example:

```
days = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Wednesday", "Friday", "Saturday"]
```

```
days[-1]
```

```
'Saturday'
```

```
days[3]
```

```
'Wednesday'
```

```
days[:4]
```

```
['Sunday', 'Monday', 'Tuesday', 'Wednesday']
```

```
print("Today is " + days[1])
```

```
Today is Monday
```

Indexing multiple levels

You can index characters inside a string inside a list. Like this:

```
name = ["Abc", "Def", "Ghi"]
print("My first name starts with " + name[0][0] + ".")
```

```
My first name starts with A.
```

```
name[2][0:2]
```

```
'Gh'
```

You would use the same style of indexing to index items inside a **nested list**:

```
gradebook = [["Casey", "Maya", "Toni", "Mae"], [85, 95, 100, 88]]
```

```
gradebook[0][-1]
```

```
'Mae'
```

```
gradebook[1][3]
```

```
88
```

```
gradebook[0][1][2]
```

```
'y'
```

for loops

We're going to work with a list of student grades.

```
grades = [73, 64, 89, 93, 59, 100, 79]

for grade in grades:
    print(grade)
```

```
73
64
89
93
59
100
79
```

Notice that the items are printed in the same order as the list.

We can change the name of our temporary variable:

```
for g in grades:
    print(g)
```

```
73
64
89
93
59
100
79
```

```
for sandwich in grades:
    print(sandwich)
```

```
73
64
89
93
59
100
79
```

```
sandwich
```

```
79
```

We should **not** use the name of the list as the name of the temp variable:

```
for grades in grades:
    print(grades)
```

```
73
64
89
93
59
100
79
```

```
grades
```

```
79
```

```
grades = [73, 64, 89, 93, 59, 100, 79]
```

Let's say we want to give each student an extra 5 points:

```
for g in grades:
    print(g + 5)
```

```
78
69
94
98
64
105
84
```

Adding items to an empty list

Let's save a new list of grades with the 5 extra points:

```
new_grades = []

for g in grades:
    new_grades.append(g + 5)

print(new_grades)
```

```
[78, 69, 94, 98, 64, 105, 84]
```

If you are working with more complicated calculations, you will want to be more **explicit** by defining a new variable inside the loop instead of doing the calculations inside the `append()` function:

```
new_grades = []

for g in grades:
    new_g = round((g + 5) / 100, 2)
    new_grades.append(new_g)

print(new_grades)
```

```
[0.78, 0.69, 0.94, 0.98, 0.64, 1.05, 0.84]
```

```
for g in grades:
    print(g)
```

```
73
64
89
93
59
100
79
```

```
for g in grades:
    print(g)
```

```
73
64
```

```
89
93
59
100
79

for g in grades:
    print(g)

73
64
89
93
59
100
79
```

Example - Doing something with each item in a list

First, let's store a list of the high temperatures in some city over the past week:

```
c_temps = [25, 25, 25, 27, 27, 26, 24]
```

The equation to convert from Celsius to Farenheit is F = C x 9/5 + 32. We can write a for loop to convert our `c_temps` to Farenheit:

```
c_temps = [25, 25, 25, 27, 27, 26, 24]
for temp in c_temps[1:7]:
    f = temp * (9/5) + 32
    print(f)

77.0
77.0
80.6
80.6
78.80000000000001
75.2
```

What is the problem with the above for loop?

Let's improve this output by rounding the temperatures to one digit past the decimal, and by writing a complete sentence:

```
for temp in c_temps:
    f = temp * (9/5) + 32
    f2 = round(f, 1)
    print("The high temperature was " + str(f2) + " degrees F (" +
str(temp) + " C).")
```

```
The high temperature was 77.0 degrees F (25 C).
The high temperature was 77.0 degrees F (25 C).
The high temperature was 77.0 degrees F (25 C).
The high temperature was 80.6 degrees F (27 C).
The high temperature was 80.6 degrees F (27 C).
The high temperature was 78.8 degrees F (26 C).
The high temperature was 75.2 degrees F (24 C).
```

if statements

```
grades = [73, 64, 89, 93, 59, 100, 79]
```

We can use a series of if/elif/else statements to perform different actions on grades in different ranges:

```
for g in grades:
    if g >= 90:
        print(g)
        print("grade is A")
    elif g >= 80:
        print(g)
        print("grade is B")
    elif g >= 70:
        print(g)
        print("grade is C")
    elif g >= 60:
        print(g)
        print("grade is D")
    else:
        print(g)
        print("grade is Fail")
```

```
73
grade is C
64
grade is D
89
grade is B
93
grade is A
59
grade is Fail
100
grade is A
79
grade is C
```

Here we are going to save a new list of grades that only includes grades 60 or higher:

```
grades = [73, 64, 89, 93, 59, 100, 79]

passing_grades = []
for g in grades:
    if g >= 60:
        passing_grades.append(g)
    else:
        continue
print(passing_grades) # notice the indentation here
```

```
[73, 64, 89, 93, 100, 79]
```

We don't have to explicitly pass if the condition isn't met:

```
passing_grades = []
for g in grades:
    if g >= 60:
        passing_grades.append(g)
print(passing_grades)
```

```
[73, 64, 89, 93, 100, 79]
```

We can also stop the loop if a condition is or isn't met:

```
for g in grades:
    if g >= 70:
        print("This student is doing ok.")
    else:
        break
        print("I give up. I quit.")
```

```
This student is doing ok.
```

Here I'm switching the order of the last two lines. Take a minute to try and predict what will happen before running the code.

```
for g in grades:
    if g >= 70:
        print("This student is doing ok.")
    else:
        break
        print("I give up. I quit.")
```

```
This student is doing ok.
```

Example - for loop with if/elif/else statements

```
students = ["Eleven", "Dustin", "Mike", "Will", "Lucas", "Max"]
```

```
for student in students:
    if student == "Will":
        print(student + " should repeat 7th grade because of
absences.")
    elif student == "Eleven":
        print("I'm not sure who this student is, they just showed up
one day.")
    else:
        print(student + " can move up to 8th grade.")
```

```
I'm not sure who this student is, they just showed up one day.
Dustin can move up to 8th grade.
Mike can move up to 8th grade.
Will should repeat 7th grade because of absences.
Lucas can move up to 8th grade.
Max can move up to 8th grade.
```

We can add more code under each if/elif/else statement. Let's add the students to the correct empty list:

```
grade_8 = []
grade_7 = []
unsure = []
for student in students:
    if student == "Will":
        print(student + " should repeat 7th grade because of
absences.")
        grade_7.append(student)
    elif student == "Eleven":
        print("I'm not sure who this student is, they just showed up
one day.")
        unsure.append(student)
    else:
        print(student + " can move up to 8th grade.")
        grade_8.append(student)
```

```
I'm not sure who this student is, they just showed up one day.
Dustin can move up to 8th grade.
Mike can move up to 8th grade.
Will should repeat 7th grade because of absences.
Lucas can move up to 8th grade.
Max can move up to 8th grade.
```

```
print("Grade 7:")
print(grade_7)
print("Grade 8:")
print(grade_8)
print("Unsure where to place:")
print(unsure)
```

```
Grade 7:
['Will']
Grade 8:
['Dustin', 'Mike', 'Lucas', 'Max']
Unsure where to place:
['Eleven']
```

BACK TO THE SLIDES

try/except

We often need to use try/except if we aren't sure whether all the items in our list are of the correct data type.

```
test_scores = [90, 95, "absent", 100, 76]

bonus_scores = []
for score in test_scores:
    bonus_scores.append(score + 2)
print(bonus_scores)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[110], line 3
      1 bonus_scores = []
      2 for score in test_scores:
----> 3     bonus_scores.append(score + 2)
      4 print(bonus_scores)

TypeError: can only concatenate str (not "int") to str
```

This gave us an error because you can't add the number 2 to the word "absent", so we can rewrite it:

```
test_scores = [90, 95, "absent", 100, 76]
bonus_scores = []
for score in test_scores:
    try:
        bonus_scores.append(score + 2)
    except TypeError:
        bonus_scores.append(score)
print(bonus_scores)
```

```
[92, 97, 'absent', 102, 78]
```

You need to specify the type of error in your except statement. Sometimes this means that you need to run the code without try/except first to get the name of the error you should expect. If you don't include the error type, it will perform the except code for all exceptions, which might seem great, but it can actually create more trouble for you down the road.

```
test_scores2 = [90, 95, "absent", 100, "76"]
```

Here we have the same list, except you accidentally entered the last score as a string instead of an integer. Let's try this list with the same code we just wrote:

```
bonus_scores2 = []
for score in test_scores2:
    try:
        bonus_scores2.append(score + 2)
    except TypeError:
        bonus_scores2.append(score)
print(bonus_scores2)
```

```
[92, 97, 'absent', 102, '76']
```

Let's try without adding any error type:

```
bonus_scores2 = []
for score in test_scores2:
    try:
        bonus_scores2.append(score + 2)
    except:
        bonus_scores2.append(score)
print(bonus_scores2)
```

```
[92, 97, 'absent', 102, '76']
```

It didn't add bonus points to the last score, even though you need that score included. Let's try this to account for the last score:

```
bonus_scores2 = []
for score in test_scores2:
    try:
        bonus_scores2.append(score + 2)
    except TypeError:
        bonus_scores2.append(int(score) + 2)
print(bonus_scores2)
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
Cell In[115], line 4
      3 try:
```

```
----> 4     bonus_scores2.append(score + 2)
      5 except TypeError:

TypeError: can only concatenate str (not "int") to str

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call
last)
Cell In[115], line 6
      4         bonus_scores2.append(score + 2)
      5     except TypeError:
----> 6         bonus_scores2.append(int(score) + 2)
      7 print(bonus_scores2)

ValueError: invalid literal for int() with base 10: 'absent'
```

Now we can see that the second error is a `ValueError`, so we can add a second try/except to our code:

```
test_scores2 = [90, 95, "absent", 100, "76"]

bonus_scores2 = []
for score in test_scores2:
    try:
        bonus_scores2.append(score + 2)
    except TypeError:
        try:
            bonus_scores2.append(int(score) + 2)
        except ValueError:
            bonus_scores2.append(score)
print(bonus_scores2)
```

```
[92, 97, 'absent', 102, 78]
```

This was a complicated example, but it shows how it can take some logic to get the results you want.

New boolean operator

So far we've learned `<` `>` `<=` `>=` `==` and `!=`.

`in` is another Boolean operator. It works for both lists and strings.

```
"pie" in "pizza pie"
```

```
True
```

```
"Oreo" in ["Chips Ahoy", "Oreo", "Oatmeal raisin"]
```

```
True
```

```
cookies = ["Chips Ahoy", "Oreo", "Oatmeal raisin"]
"Oreo" in cookies
```

```
True
```

```
"oreo" in ["Chips Ahoy", "Oreo", "Oatmeal raisin"]
```

```
False
```

```
"Ahoy" in "Chips Ahoy"
```

```
True
```

Note the difference between the following two cells:

```
"Ahoy" in ["Chips Ahoy", "Oreo", "Oatmeal raisin"]
```

```
False
```

```
cookies = ["Chips Ahoy", "Oreo", "Oatmeal raisin"]
for c in cookies:
    if "Ahoy" in c:
        print(c)
```

```
Chips Ahoy
```

Let's say we want to give one extra point to anyone who is right on the verge of getting a better grade:

```
grades = [73, 64, 89, 93, 59, 100, 79]
grades_to_round = [59, 69, 79, 89]
rounded_grades = []

for g in grades:
    if g in grades_to_round:
        rounded_grades.append(g + 1)
    else:
        rounded_grades.append(g)

print(rounded_grades)
```

```
[73, 64, 90, 93, 60, 100, 80]
```

There's also `not in`:

```
grades = [73, 64, 89, 93, 59, 100, 79]
grades_to_round = [59, 69, 79, 89]
rounded_grades = []
```

```
for g in grades:
    if g not in grades_to_round:
        rounded_grades.append(g)
    else:
        rounded_grades.append(g + 1)

print(rounded_grades)
```

```
[73, 64, 90, 93, 60, 100, 80]
```

BACK TO THE SLIDES