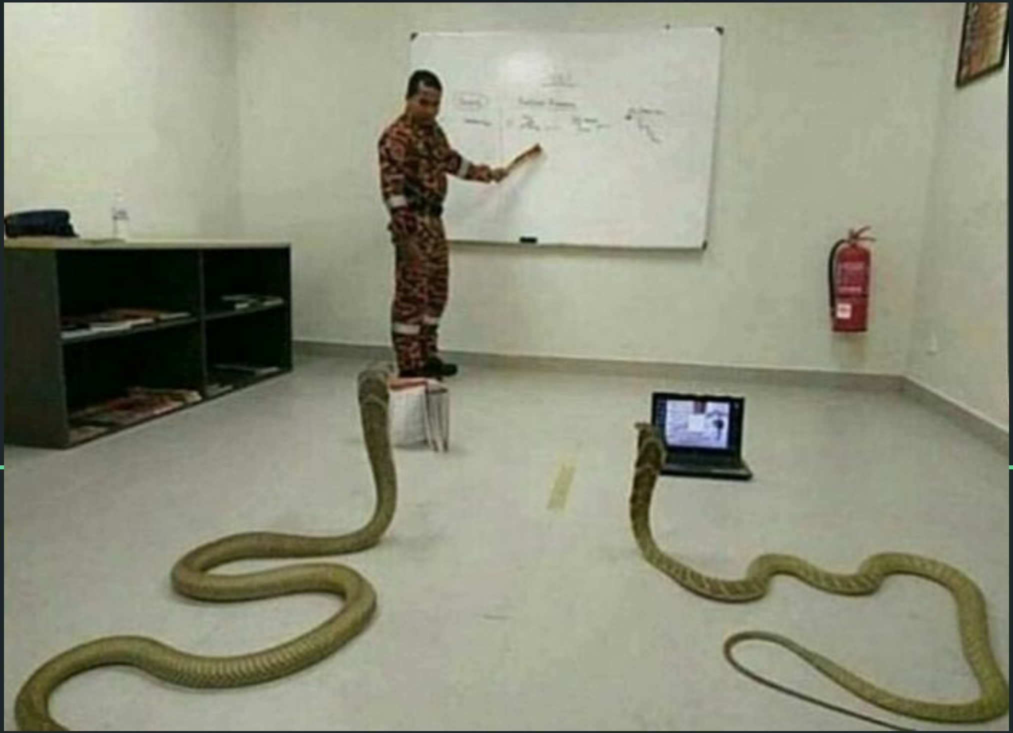
python!

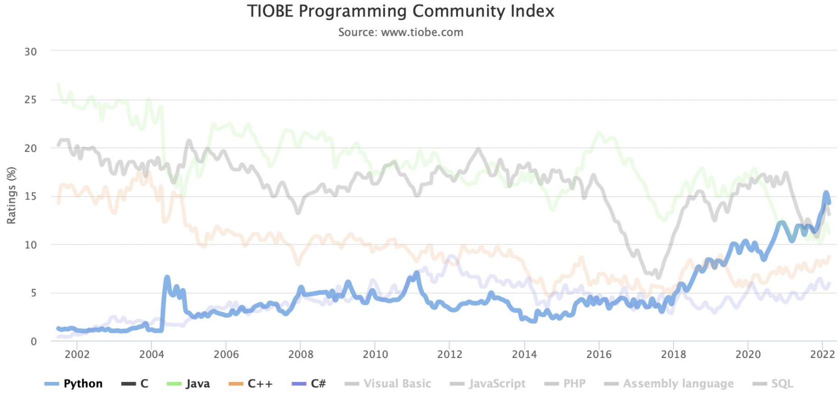


Final Stretch Time line

- Week 8 (Apr 4- Apr 10) Transition to Python
- Week 9 (Apr 11 - Apr 17) Python continued
- Week 10 (Apr 18 - Apr 25) Specialization:
- Week 11-Week 13 : FINAL PROJECT!!

Brief History

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning
- Considered a scripting language, but is much more
- Scalable, object oriented and functional from the beginning
- Used by Google from the beginning
- Increasingly popular



TIOBE Programming Community Index

Source: www.tiobe.com

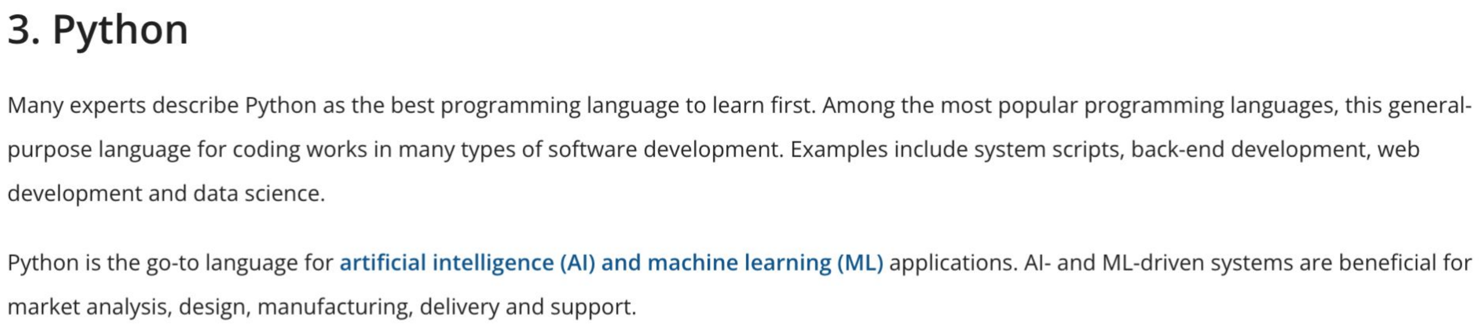
How to install it?

- If you are on a mac/unix based system, python comes pre installed.
- If you are on a windows you can get it at https://www.python.org/

For practicality we will be using colab:

https://colab.research.google.com/

Why python?



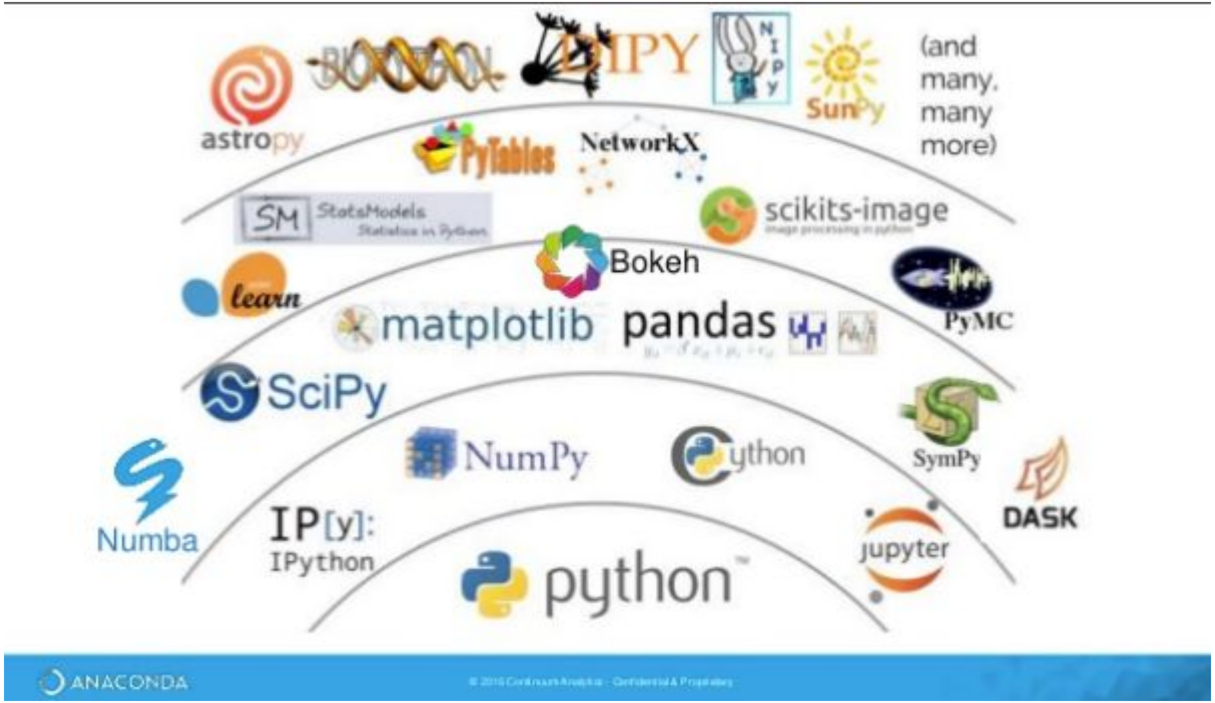
3. Python

Many experts describe Python as the best programming language to learn first. Among the most popular programming languages, this general-purpose language for coding works in many types of software development. Examples include system scripts, back-end development, web development and data science.

Python is the go-to language for **artificial intelligence (AI) and machine learning (ML)** applications. AI- and ML-driven systems are beneficial for market analysis, design, manufacturing, delivery and support.

https://www.exitcertified.com/blog/the-top-10-programming-languages-in-2021?

Great support and libraries



Characteristics

- Interpreted language: work with an evaluator for language expressions (like DrJava, but more flexible)
- Dynamically typed: variables do not have a predefined type
- Rich, built-in collection types:
  - Lists
  - Tuples
  - Dictionaries (maps)
  - Sets
- Concise

Some examples converting from matlab to python

- disp —-> print
- ends —-> Indentation
- % comments —> #for comments

-
  ```
  [y ]= function (var1,var2 ,..)    —> def function(var1,var2, ..)

  end                                        return y
  ```
- A = [0,1,2,3]
- disp(A(1)) —> print(A[0])

-
  ```
  for v = 10:-2:0           for i in range(10,0,-2):
     disp(v)          --- >             print(i)
  End
  ```
- no more semicolons!

operators

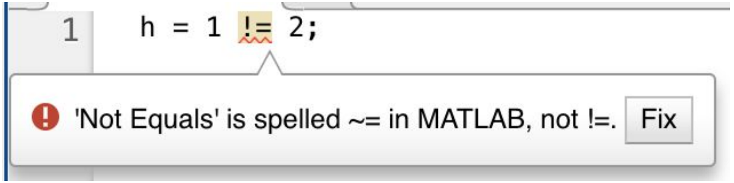
most math is the same!

we can use special commands “and” “or” “not” in python as a shortcut

some packages in python (like pandas) still use & so it’s good we learned them

(haley’s) favorite error in matlab is:

in python we will use !=



```
h = 1 != 2;
```

'Not Equals' is spelled ~= in MATLAB, not !=.

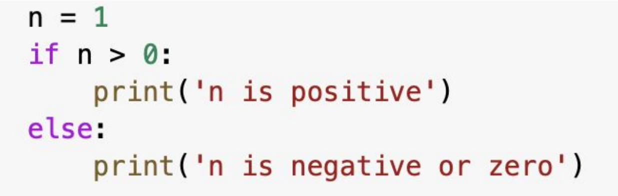
power is ** in python, and modulo is %

some math can be shortened: in matlab n = n+1 in python becomes n+=1

control flow and indentation

in matlab, indentation was optional and just for readability of the user. in python, indentations mean something!

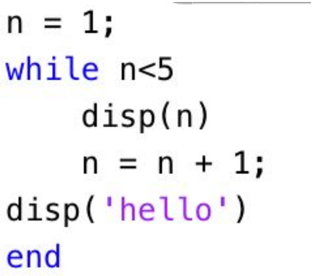
let’s look at this if-else statement



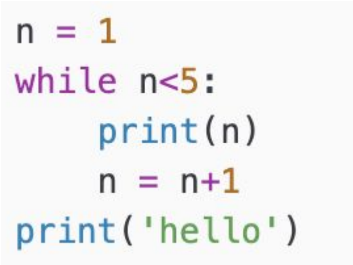
```python
n = 1
if n > 0:
    print('n is positive')
else:
    print('n is negative or zero')
```

first, notice the : which are used to tell python the statement is over

then, notice that what we do in each case is **indented** but that we don’t have to use the **end** command to finish our control statement!



```matlab
n = 1;
while n<5
    disp(n)
    n = n + 1;
disp('hello')
end
```



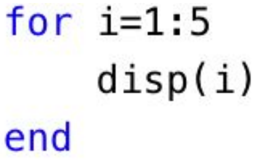
```python
n = 1
while n<5:
    print(n)
    n = n+1
print('hello')
```

how many times will hello print in matlab? in python?

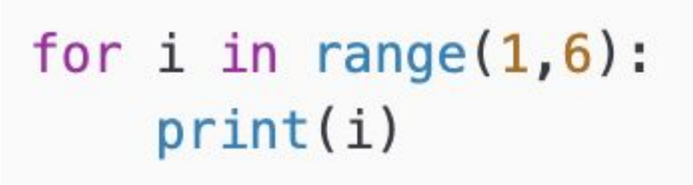
control flow and indentation

if / else still works the same **logically** in python and matlab. just the syntax changes! for example, **elseif** is spelled **elif** in python. for loops still define variables and while loops still need pre-defined ones. while loops have similar syntax (just need the colon).

for loops are a little different:



```
for i=1:5
    disp(i)
end
```

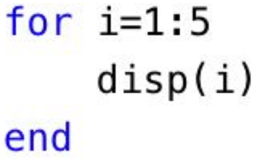


```
for i in range(1,6):
    print(i)
```

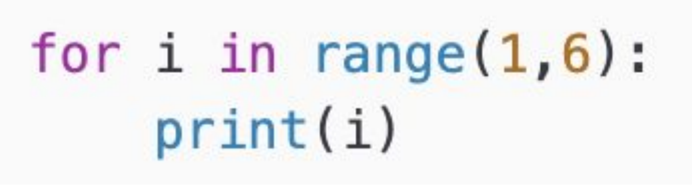
control flow and indentation

if / else still works the same **logically** in python and matlab. just the syntax changes! for example, **elseif** is spelled **elif** in python. for loops still define variables and while loops still need pre-defined ones. while loops have similar syntax (just need the colon).

for loops are a little different:



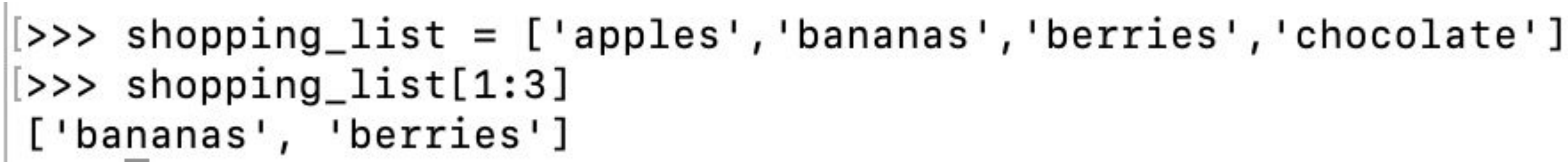
```
for i=1:5
    disp(i)
end
```



```
for i in range(1,6):
    print(i)
```

you may have heard of “zero” counting before in other programming languages. now we will use it!

the **end value** is **non-inclusive** in python. this will be the same in lists:

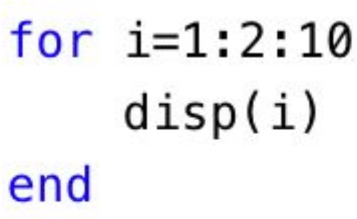


```
[>>> shopping_list = ['apples','bananas','berries','chocolate']
[>>> shopping_list[1:3]
 ['bananas', 'berries']
```

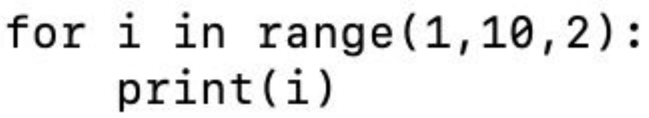
control flow and indentation

range() function: **range(**start**,** stop**,** step**)**

if you don’t input a start or a step, will assume start=0 and step=1.



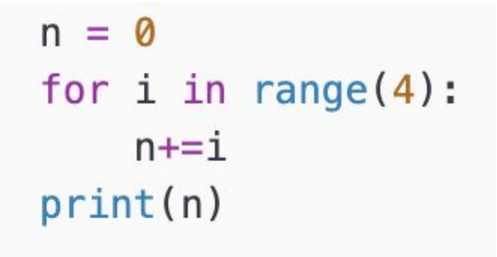
```
for i=1:2:10
    disp(i)
end
```



```
for i in range(1,10,2):
    print(i)
```

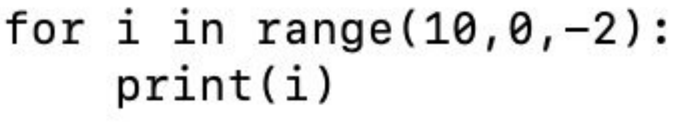
note! range can’t take decimals, so no range(1,3,.1). when we get into numpy, we can see how to do that.

what will this print?



```
n = 0
for i in range(4):
    n+=i
print(n)
```

what will this print?

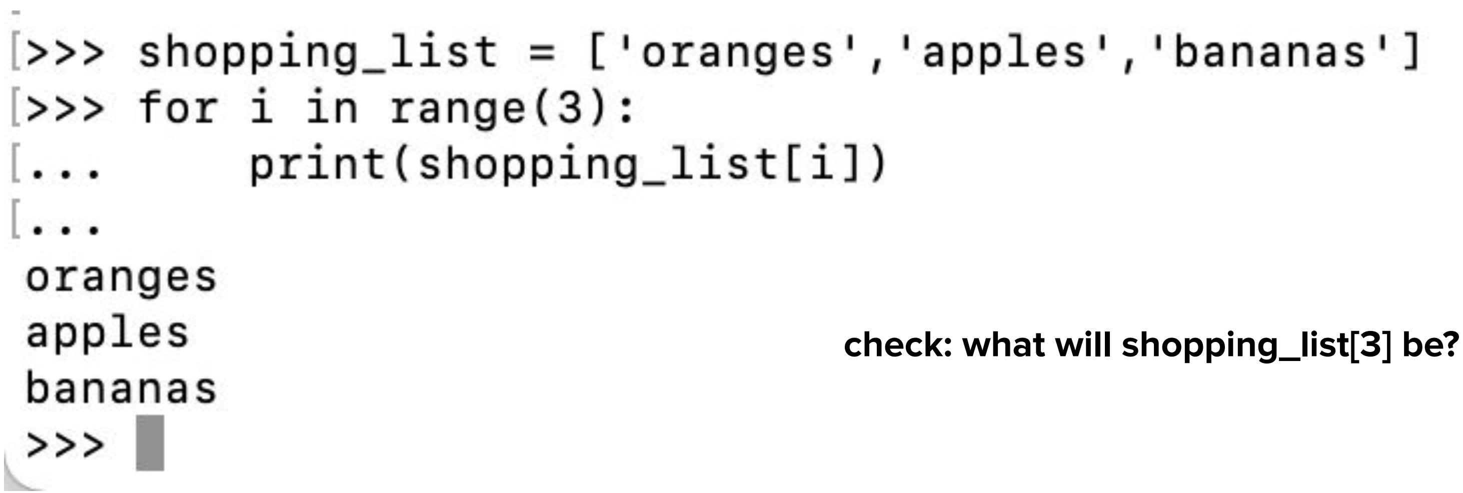


```
for i in range(10,0,-2):
    print(i)
```

(brief) info about lists – more wednesday

it might seem weird to have the range() function automatically start at 0, but everything in python is zero-indexed so it will be very helpful!

for example, I want to print out everything in a list. the first element of the list will be the “zero-th” element!



```
[>>> shopping_list = ['oranges','apples','bananas']
[>>> for i in range(3):
[...     print(shopping_list[i])
[...
oranges
apples
bananas
>>> 
```

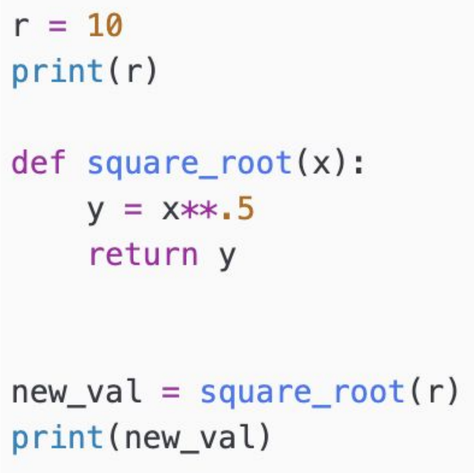
check: what will shopping_list[3] be?

functions

biggest differences are:

- **function -> def**
- **use return statement for variables you keep**
- **need to have indentations**
- **no need for end, yes need for :**
- **no need to create its own file / name your file with a function**

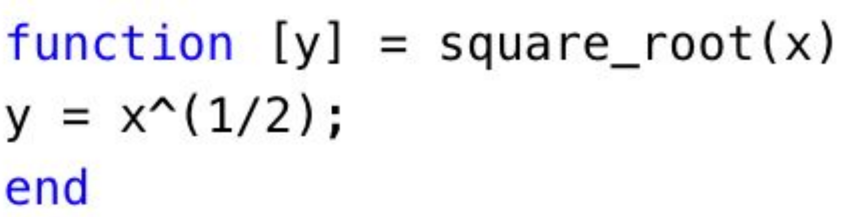
global / local function information still applies!



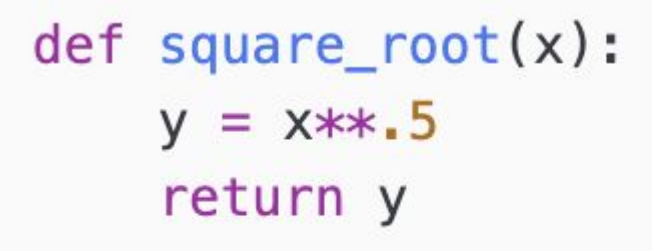
```
r = 10
print(r)

def square_root(x):
    y = x**.5
    return y


new_val = square_root(r)
print(new_val)
```



```
function [y] = square_root(x)
y = x^(1/2);
end
```



```
def square_root(x):
    y = x**.5
    return y
```

python is very flexible

with great power comes great responsibility... it will not throw an error if you try to make a list of

p = [3, ’3’, ”three”, [3,3,3]]

but if you try to do p+1 you will have problems!

Let’s get our hands dirty:

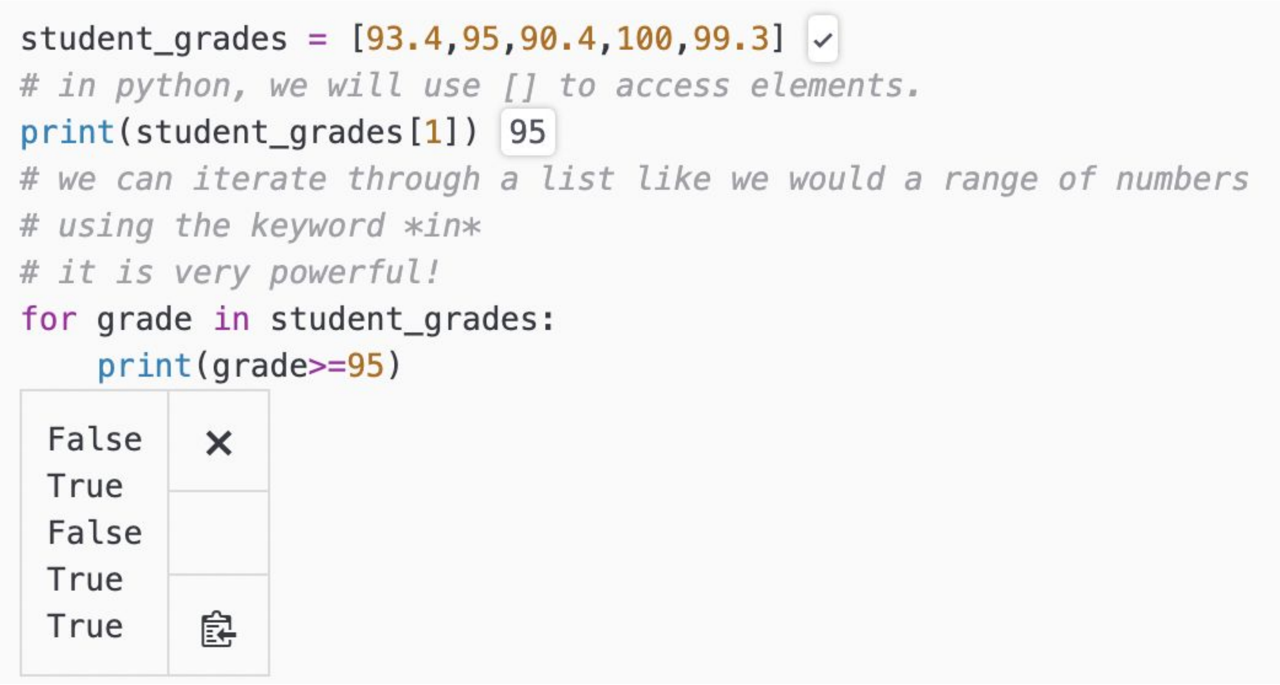
visit :

bit.ly/clps950_lect44

and MAKE A COPY!

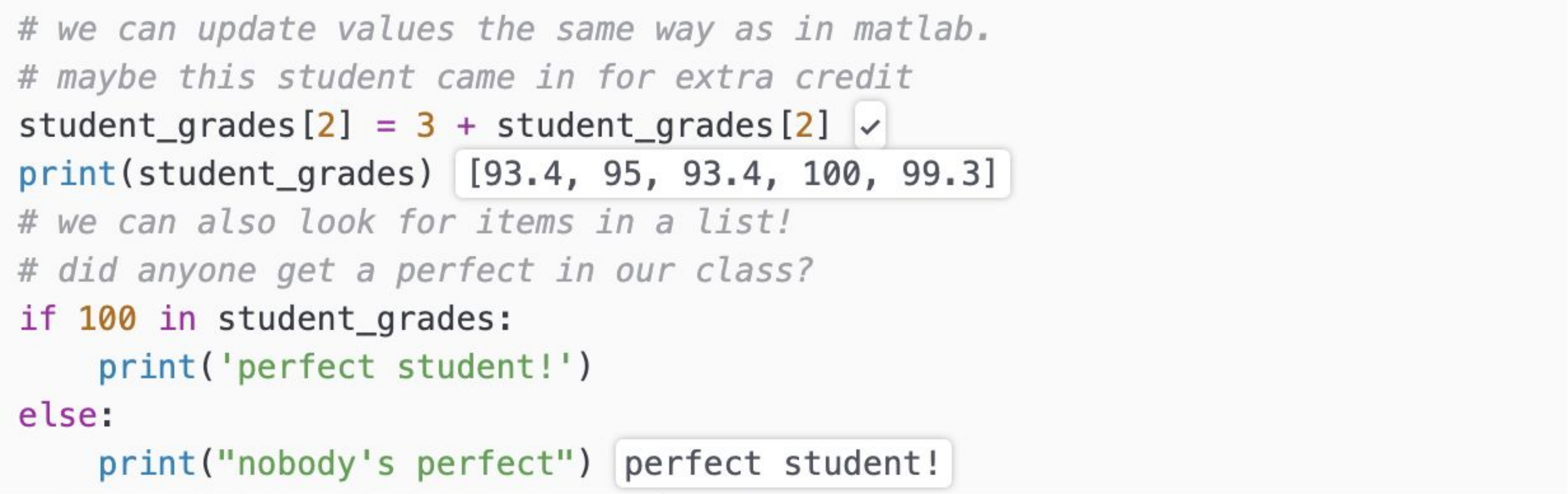
lists - like less structured matlab arrays!

we’ll work with numbers for this example – the example in the tutorial is strings!

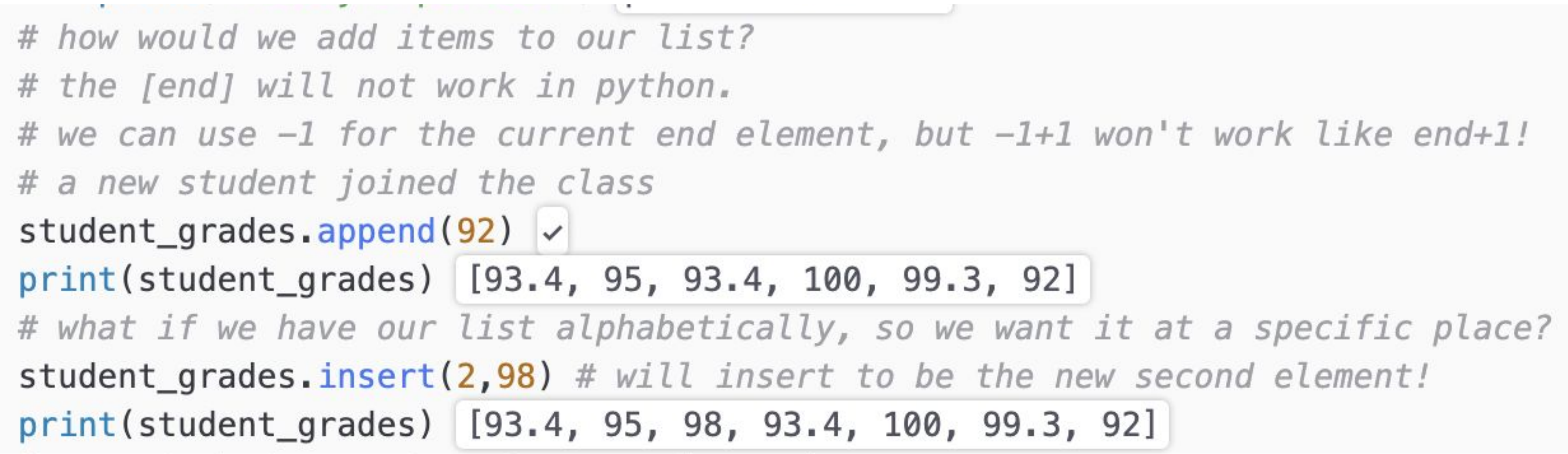


```
student_grades = [93.4,95,90.4,100,99.3] ✓
# in python, we will use [] to access elements.
print(student_grades[1]) 95
# we can iterate through a list like we would a range of numbers
# using the keyword *in*
# it is very powerful!
for grade in student_grades:
    print(grade>=95)
```

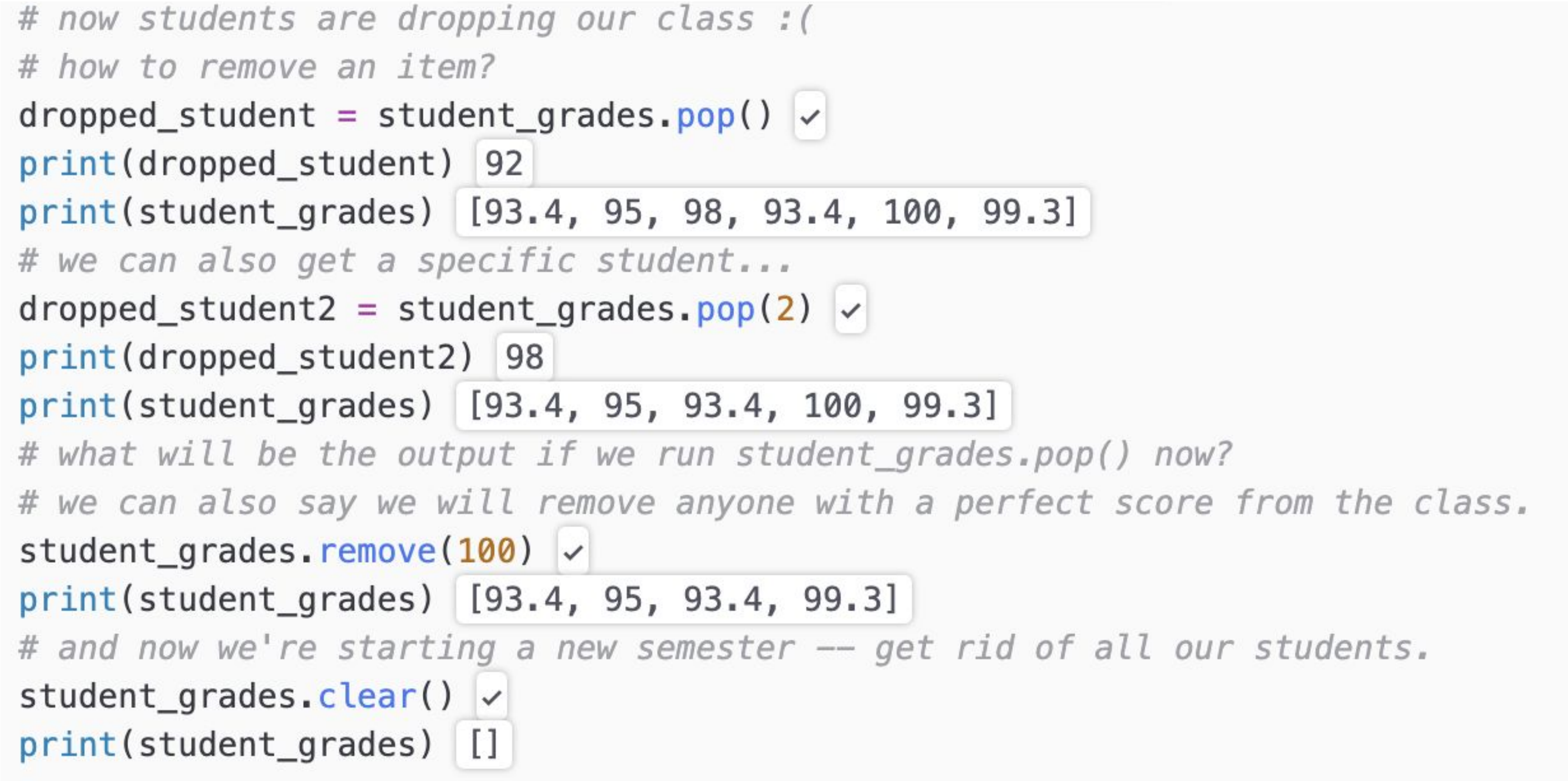
```
False
True
False
True
True
```



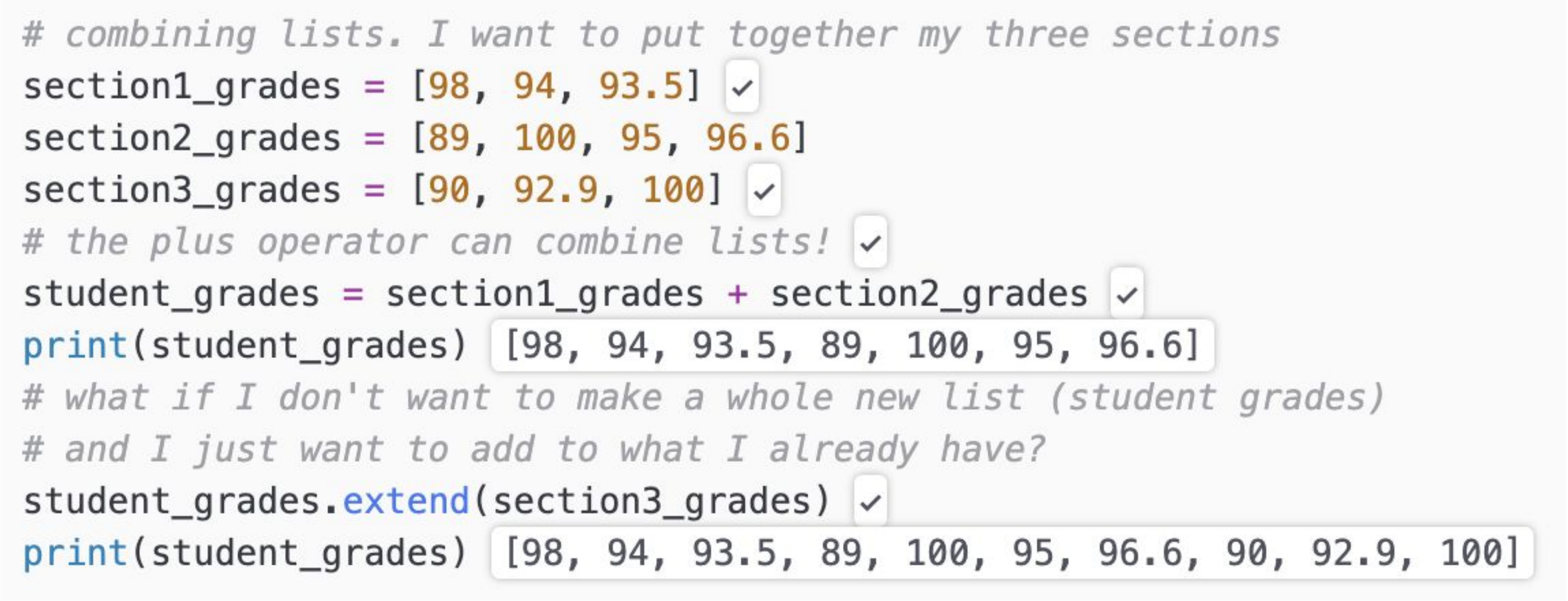
```
# we can update values the same way as in matlab.
# maybe this student came in for extra credit
student_grades[2] = 3 + student_grades[2] ✓
print(student_grades) [93.4, 95, 93.4, 100, 99.3]
# we can also look for items in a list!
# did anyone get a perfect in our class?
if 100 in student_grades:
    print('perfect student!')
else:
    print("nobody's perfect") perfect student!
```



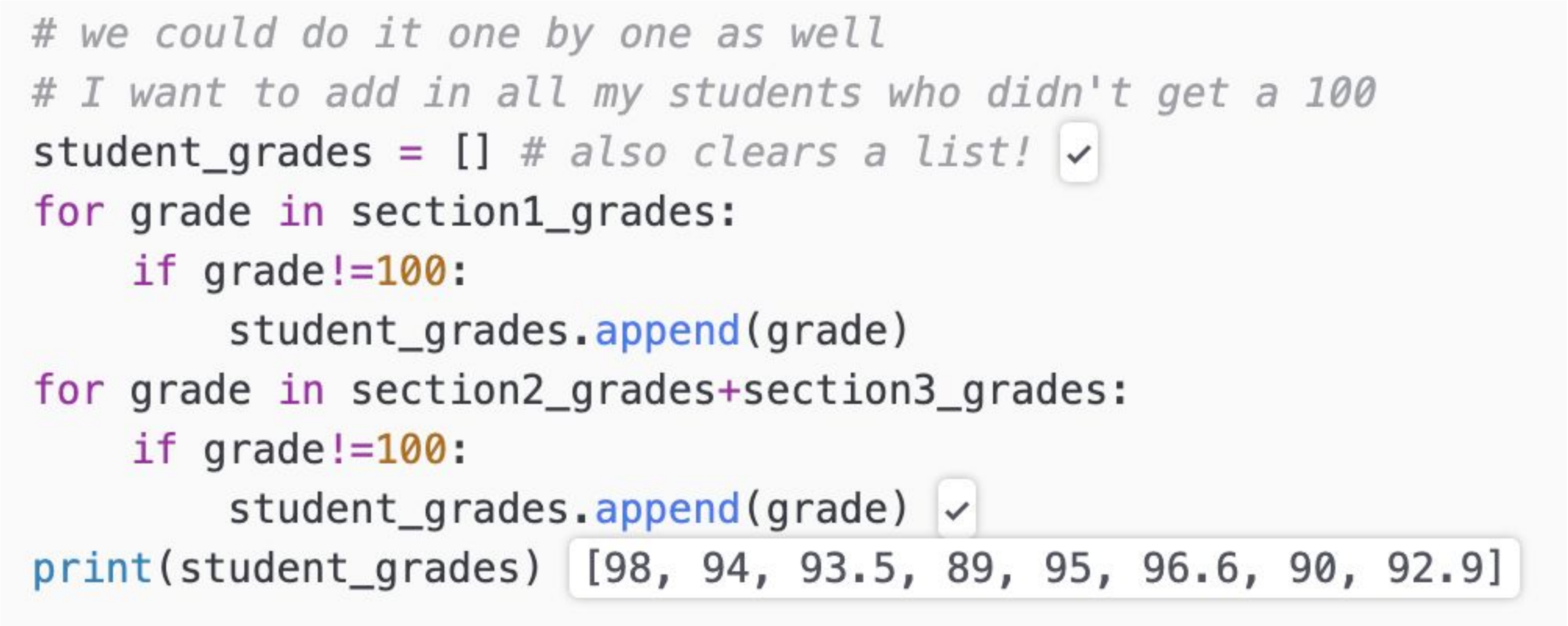
```
# how would we add items to our list?
# the [end] will not work in python.
# we can use -1 for the current end element, but -1+1 won't work like end+1!
# a new student joined the class
student_grades.append(92) ✓
print(student_grades) [93.4, 95, 93.4, 100, 99.3, 92]
# what if we have our list alphabetically, so we want it at a specific place?
student_grades.insert(2,98) # will insert to be the new second element!
print(student_grades) [93.4, 95, 98, 93.4, 100, 99.3, 92]
```



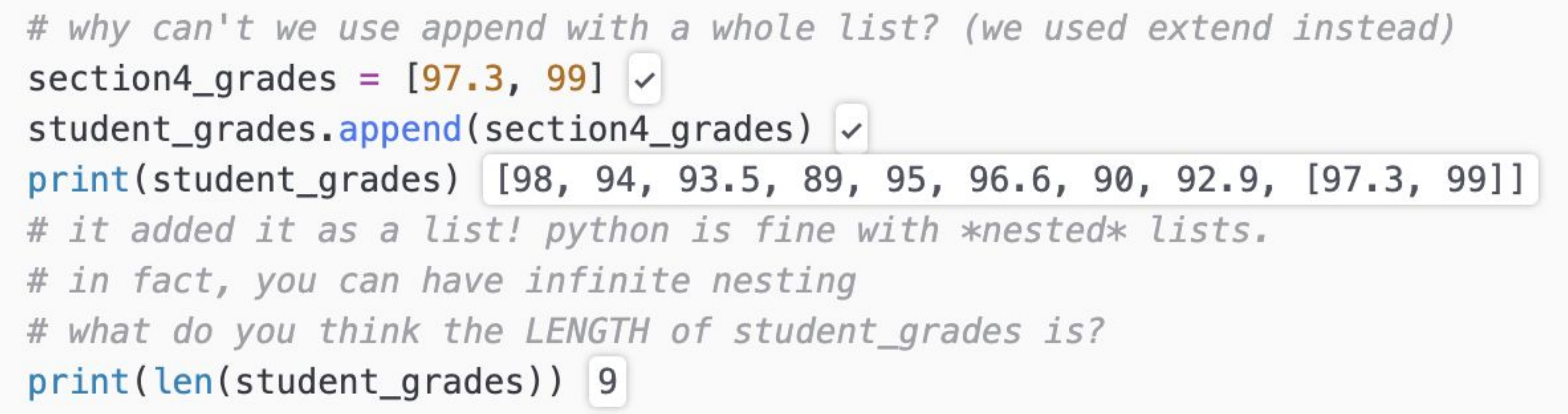
```
# now students are dropping our class :(
# how to remove an item?
dropped_student = student_grades.pop() ✓
print(dropped_student) 92
print(student_grades) [93.4, 95, 98, 93.4, 100, 99.3]
# we can also get a specific student...
dropped_student2 = student_grades.pop(2) ✓
print(dropped_student2) 98
print(student_grades) [93.4, 95, 93.4, 100, 99.3]
# what will be the output if we run student_grades.pop() now?
# we can also say we will remove anyone with a perfect score from the class.
student_grades.remove(100) ✓
print(student_grades) [93.4, 95, 93.4, 99.3]
# and now we're starting a new semester -- get rid of all our students.
student_grades.clear() ✓
print(student_grades) []
```



```
# combining lists. I want to put together my three sections
section1_grades = [98, 94, 93.5] ✓
section2_grades = [89, 100, 95, 96.6]
section3_grades = [90, 92.9, 100] ✓
# the plus operator can combine lists! ✓
student_grades = section1_grades + section2_grades ✓
print(student_grades) [98, 94, 93.5, 89, 100, 95, 96.6]
# what if I don't want to make a whole new list (student grades)
# and I just want to add to what I already have?
student_grades.extend(section3_grades) ✓
print(student_grades) [98, 94, 93.5, 89, 100, 95, 96.6, 90, 92.9, 100]
```



```
# we could do it one by one as well
# I want to add in all my students who didn't get a 100
student_grades = [] # also clears a list! ✓
for grade in section1_grades:
    if grade!=100:
        student_grades.append(grade)
for grade in section2_grades+section3_grades:
    if grade!=100:
        student_grades.append(grade) ✓
print(student_grades) [98, 94, 93.5, 89, 95, 96.6, 90, 92.9]
```



```
# why can't we use append with a whole list? (we used extend instead)
section4_grades = [97.3, 99] ✓
student_grades.append(section4_grades) ✓
print(student_grades) [98, 94, 93.5, 89, 95, 96.6, 90, 92.9, [97.3, 99]]
# it added it as a list! python is fine with *nested* lists.
# in fact, you can have infinite nesting
# what do you think the LENGTH of student_grades is?
print(len(student_grades)) 9
```