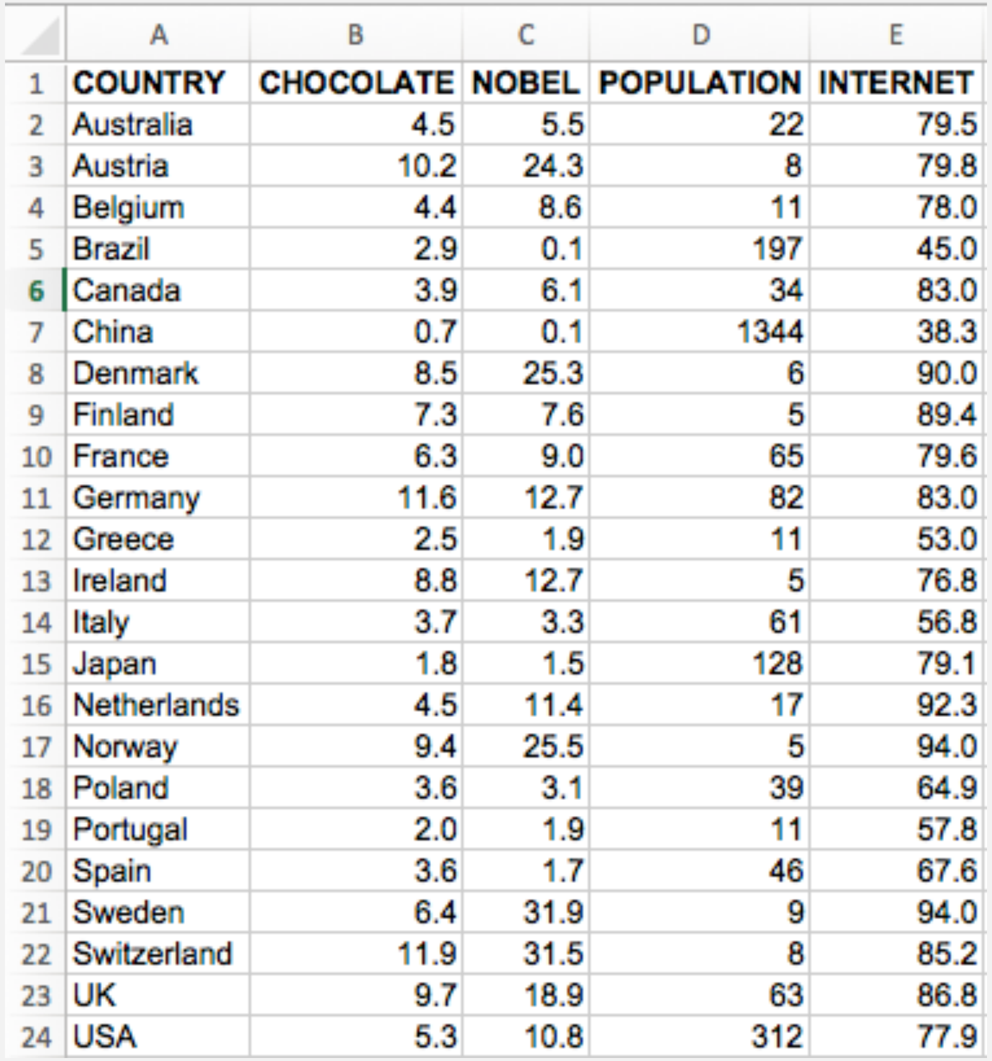
Lists and List Operations

The World Is Not Flat

The World Is Not Flat



	A	B	C	D	E
1	COUNTRY	CHOCOLATE	NOBEL	POPULATION	INTERNET
2	Australia	4.5	5.5	22	79.5
3	Austria	10.2	24.3	8	79.8
4	Belgium	4.4	8.6	11	78.0
5	Brazil	2.9	0.1	197	45.0
6	Canada	3.9	6.1	34	83.0
7	China	0.7	0.1	1344	38.3
8	Denmark	8.5	25.3	6	90.0
9	Finland	7.3	7.6	5	89.4
10	France	6.3	9.0	65	79.6
11	Germany	11.6	12.7	82	83.0
12	Greece	2.5	1.9	11	53.0
13	Ireland	8.8	12.7	5	76.8
14	Italy	3.7	3.3	61	56.8
15	Japan	1.8	1.5	128	79.1
16	Netherlands	4.5	11.4	17	92.3
17	Norway	9.4	25.5	5	94.0
18	Poland	3.6	3.1	39	64.9
19	Portugal	2.0	1.9	11	57.8
20	Spain	3.6	1.7	46	67.6
21	Sweden	6.4	31.9	9	94.0
22	Switzerland	11.9	31.5	8	85.2
23	UK	9.7	18.9	63	86.8
24	USA	5.3	10.8	312	77.9

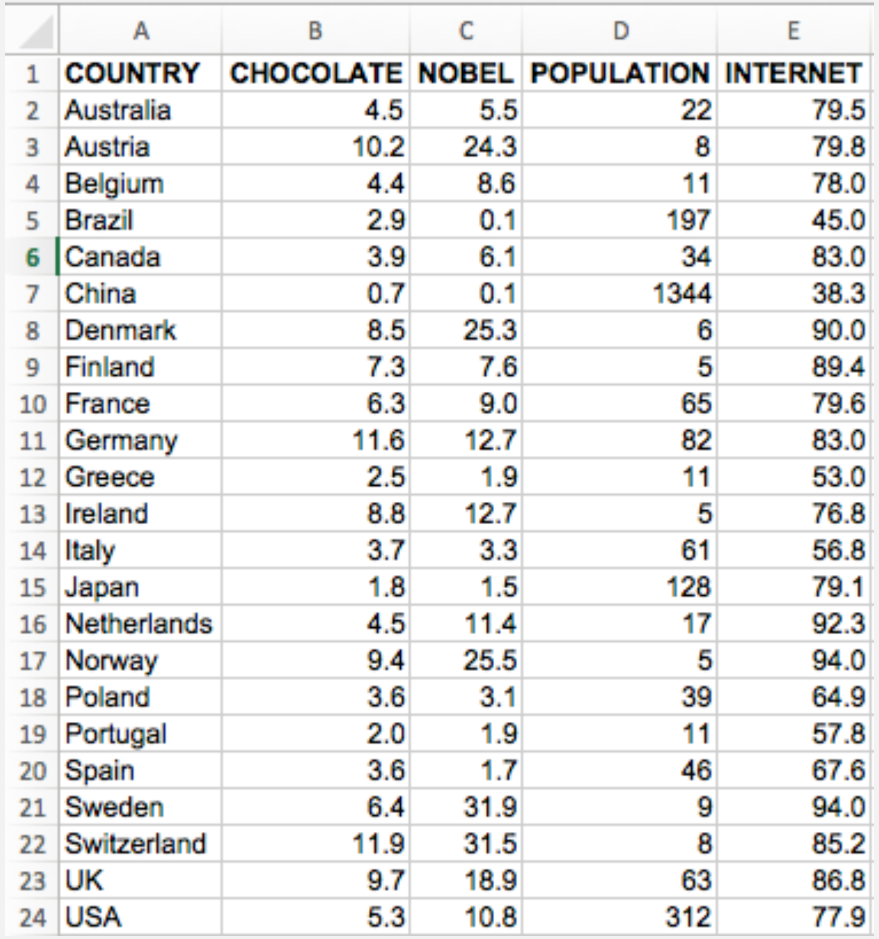
Many of us started in the world of flat files.

The world of data is much more diverse.

The backbone of most data structures is the list.

The World Is Not Flat

If this were a list,



	A	B	C	D	E
1	COUNTRY	CHOCOLATE	NOBEL	POPULATION	INTERNET
2	Australia	4.5	5.5	22	79.5
3	Austria	10.2	24.3	8	79.8
4	Belgium	4.4	8.6	11	78.0
5	Brazil	2.9	0.1	197	45.0
6	Canada	3.9	6.1	34	83.0
7	China	0.7	0.1	1344	38.3
8	Denmark	8.5	25.3	6	90.0
9	Finland	7.3	7.6	5	89.4
10	France	6.3	9.0	65	79.6
11	Germany	11.6	12.7	82	83.0
12	Greece	2.5	1.9	11	53.0
13	Ireland	8.8	12.7	5	76.8
14	Italy	3.7	3.3	61	56.8
15	Japan	1.8	1.5	128	79.1
16	Netherlands	4.5	11.4	17	92.3
17	Norway	9.4	25.5	5	94.0
18	Poland	3.6	3.1	39	64.9
19	Portugal	2.0	1.9	11	57.8
20	Spain	3.6	1.7	46	67.6
21	Sweden	6.4	31.9	9	94.0
22	Switzerland	11.9	31.5	8	85.2
23	UK	9.7	18.9	63	86.8
24	USA	5.3	10.8	312	77.9

It would look like this:

```
[
[“Australia”, 4.5, 5.5, 22, 79.5],
[“Austria”, 10.2, 24.3, 8, 79.8],
[“Belgium”, 4.4, 8.6, 11, 78.0],
[“Brazil”, 2.9, 0.1, 197, 45.0]
…
]
```

Lists of Lists

The World Is Really Not Flat

Lists of Lists

Oftentimes, we group things together in sub lists (lists of lists)

[[“Australia”, 4.5, 5.5, 22, 79.5] , [“Austria”, 10.2, 24.3, 8, 79.8],
[“Belgium”, 4.4, 8.6, 11, 78.0] , [“Brazil”, 2.9, 0.1, 197, 45.0] …]

Advantages:

- helps keep data organized
- easy to store in databases

Accessing Elements in Lists of Lists

```
country_list =  [  [“Australia”, 4.5, 5.5, 22, 79.5],
                   [“Austria”, 10.2, 24.3, 8, 79.8],
                   [“Belgium”, 4.4, 8.6, 11, 78.0],
                   [“Brazil”, 2.9, 0.1, 197, 45.0]  …  ]
```

country_list[0] = gives us our first sublist

country_list[0] = [“Australia”, 4.5, 5.5, 22, 79.5]

Accessing Elements in Lists of Lists

```
country_list =  [  [“Australia”, 4.5, 5.5, 22, 79.5],
                   [“Austria”, 10.2, 24.3, 8, 79.8],
                   [“Belgium”, 4.4, 8.6, 11, 78.0],
                   [“Brazil”, 2.9, 0.1, 197, 45.0]  …  ]
```

```
country_list[3] =     [“Brazil”, 2.9, 0.1, 197, 45.0]
```

Accessing Elements in Lists of Lists

```
country_list =  [  [“Australia”, 4.5, 5.5, 22, 79.5],
                   [“Austria”, 10.2, 24.3, 8, 79.8],
                   [“Belgium”, 4.4, 8.6, 11, 78.0],
                   [“Brazil”, 2.9, 0.1, 197, 45.0]  …  ]
```

country_list[0][0] = gives us the first element of our first sublist

country_list[0][0] = “Australia”

Accessing Elements in Lists of Lists

```
country_list =  [  [“Australia”, 4.5, 5.5, 22, 79.5],
                   [“Austria”, 10.2, 24.3, 8, 79.8],
                   [“Belgium”, 4.4, 8.6, 11, 78.0],
                   [“Brazil”, 2.9, 0.1, 197, 45.0]  …  ]
```

```
country_list[1][3] =       8
```

Accessing Elements in Lists of Lists

```
country_list =  [  [“Australia”, 4.5, 5.5, 22, 79.5],
                   [“Austria”, 10.2, 24.3, 8, 79.8],
                   [“Belgium”, 4.4, 8.6, 11, 78.0],
                   [“Brazil”, 2.9, 0.1, 197, 45.0]  …  ]
```

- -

```
country_list[0:2] =      gives us our first two sublists

country_list[0:2] =      [“Australia”, 4.5, 5.5, 22, 79.5],
                         [“Austria”, 10.2, 24.3, 8, 79.8]
```

Accessing Elements in Lists of Lists

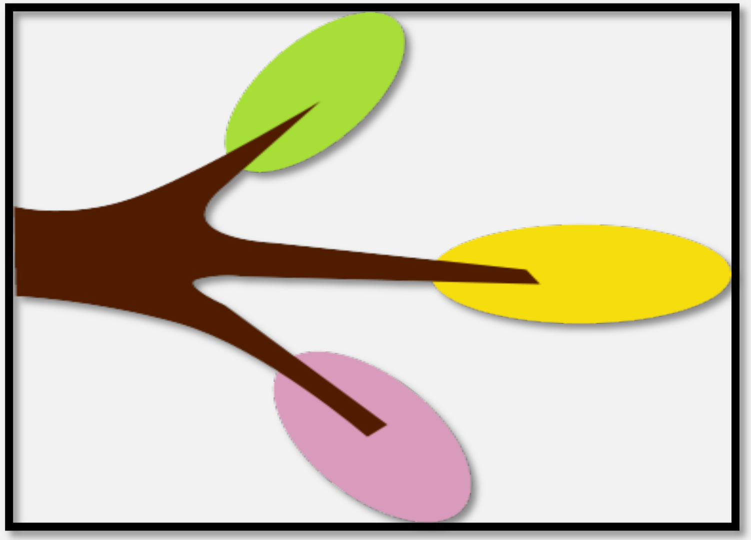
```
country_list =  [  [“Australia”, 4.5, 5.5, 22, 79.5],
                   [“Austria”, 10.2, 24.3, 8, 79.8],
                   [“Belgium”, 4.4, 8.6, 11, 78.0],
                   [“Brazil”, 2.9, 0.1, 197, 45.0]  …  ]
```

```
country_list[1:3] =       [“Austria”, 10.2, 24.3, 8, 79.8],
                          [“Belgium”, 4.4, 8.6, 11, 78.0]
```

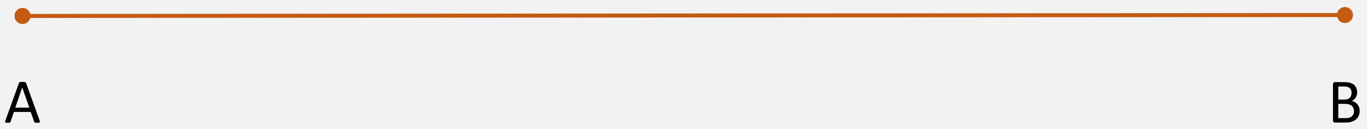
Menti!

Conditional Statements

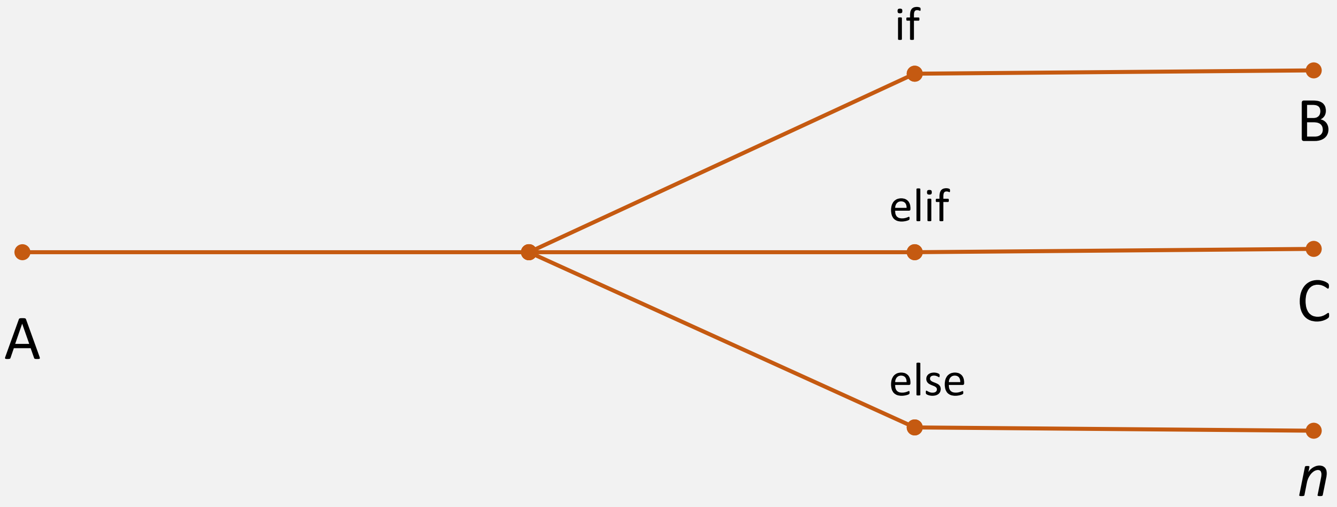
Branching Out



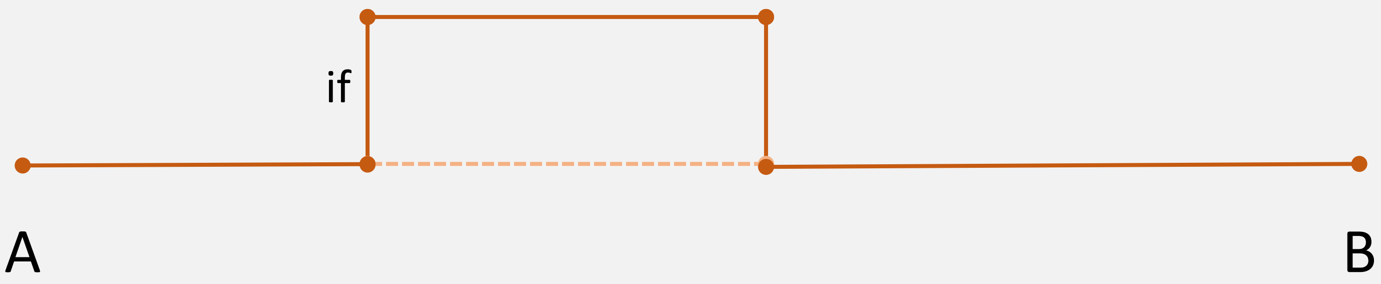
Current Programming Approach



Conditionals Change This



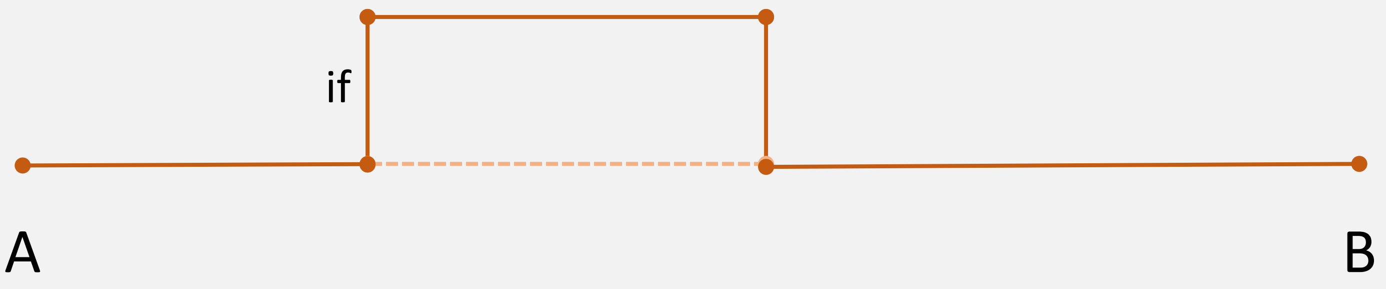
if



Runs only if a condition is met.

If the condition is not met, the code skips the if statement and continues running.

if



```
people = 100
chairs = 50

if people > chairs:
    print(”We need more chairs!”)
```

This indentation is necessary for the code to run properly.

if

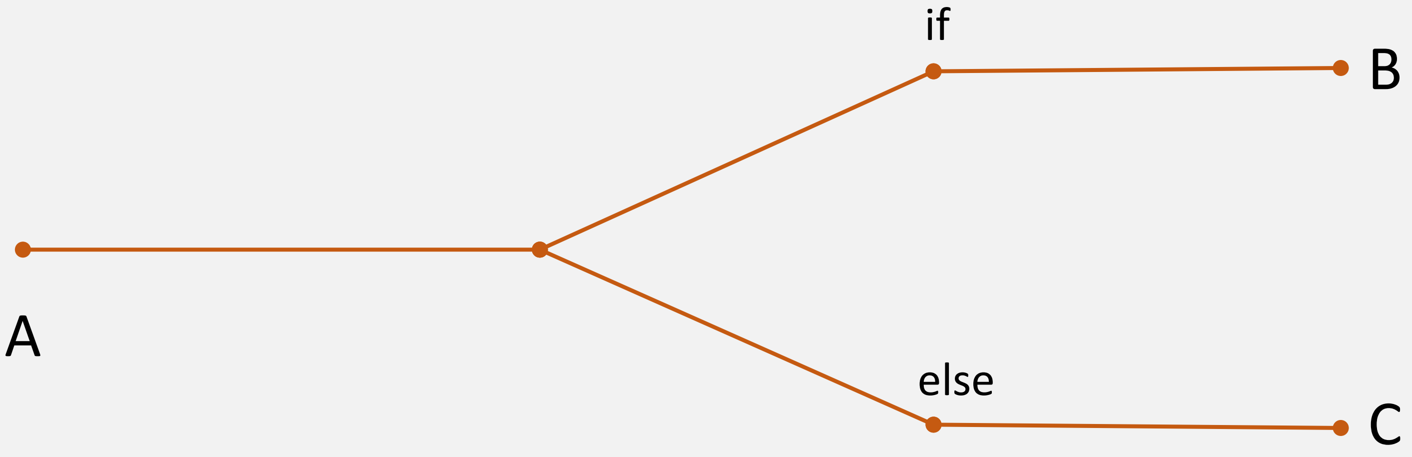
```
people = 100
chairs = 50

if people > chairs:
      print(”We need more chairs!”)
```

Condition is met.
The print statement will run.

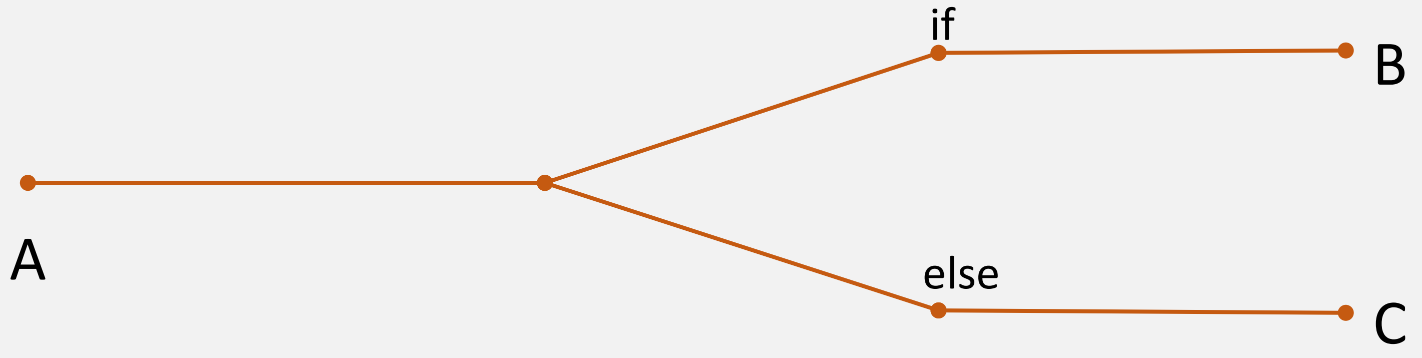
```
people = 100
chairs = 101

if people > chairs:
      print(”We need more chairs!”)
```

Condition is **not** met.
The print statement will **not** run.

if - else



if - else



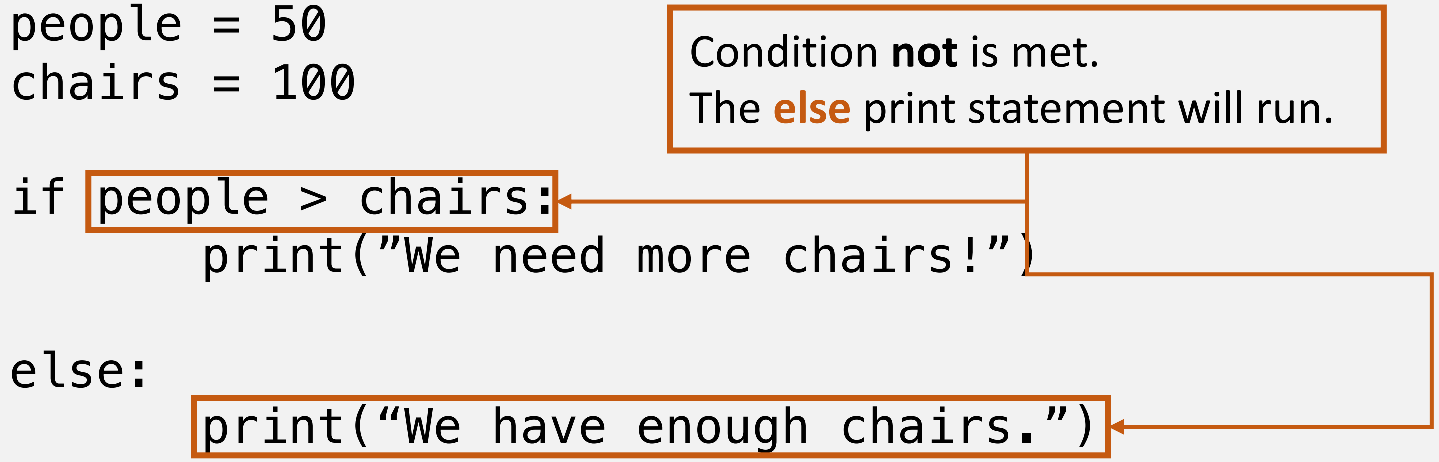
```
people = 100
chairs = 50

if people > chairs:
	print(”We need more chairs!”)

else:
	print(“We have enough chairs.”)
```

if - else

```
people = 100
chairs = 50

if people > chairs:
        print(”We need more chairs!”)

else:
        print(“We have enough chairs.”)
```

Condition is met.
The **if** print statement will run.

if - else



```
people = 50
chairs = 100

if people > chairs:
        print(”We need more chairs!”)

else:
        print(“We have enough chairs.”)
```

Condition **not** is met.
The **else** print statement will run.

if - else

```
goldfish = input(“Do you like goldfish? (Y/N)”)

> Y

if goldfish == ‘Y’:
        print(”That’s like the dirtiest fish!”)

else:
        print(“Good. Did you know that goldfish tend
to be very dirty?”)
```

```
That’s like the dirtiest fish!
```

if - else

```
cheese = input(“””
If you’re eating a cheese that isn’t yours, what
kind of cheese is it?
”””)

> Nacho Cheese


if cheese == ‘Nacho Cheese’:
       print(”That’s a great joke!”)

else:
       print(“Nacho cheese!”)
```

```
That’s a great joke!
```

if - elif - else

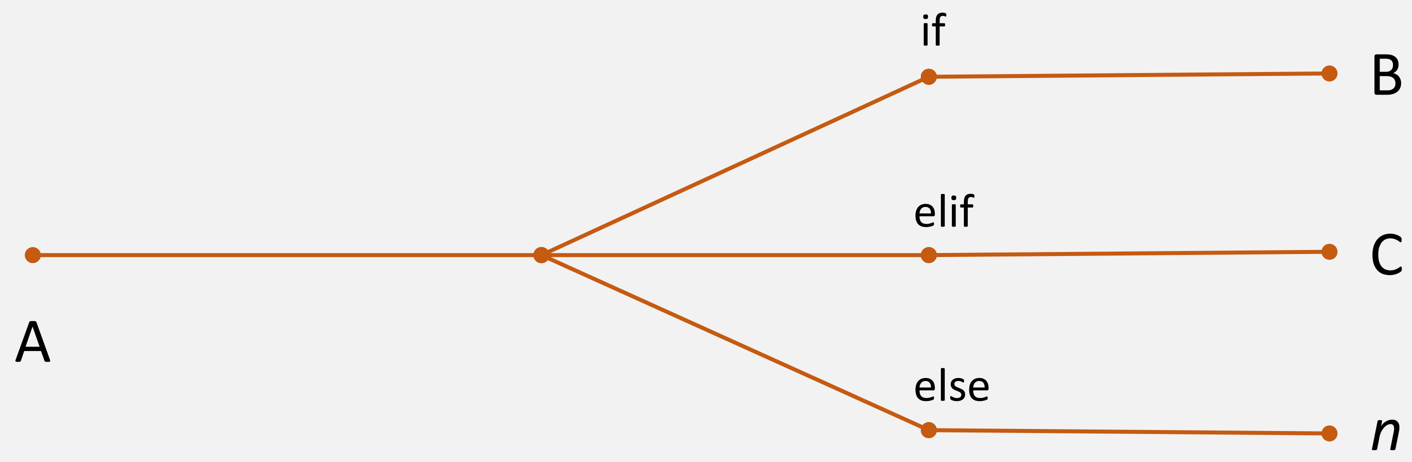
In Excel…

```
=IF(condition, output if true, output if false)
```

```
=IF(people > chairs, ”We need more chairs”,
IF(people == chairs, “Perfect! Good job
everyone!”, ”We have enough chairs.”), ”We
have enough chairs.”)
```

elif is an elegant solution for nested IF statements.

if - elif - else



if - elif - else

```
people = 100
chairs = 50

if people > chairs:
        print(”We need more chairs!”)


elif people == chairs:
        print(“Perfect! Great job everyone!”)


else:
        print(“We have enough chairs.”)
```

Condition is met.
The **if** print statement will run.

if - elif - else

```
people = 100
chairs   = 100

if people > chairs:
       print(”We need more chairs!”)


elif people == chairs:
       print(“Perfect! Great job everyone!”)


else:
       print(“We have enough chairs.”)
```

Condition is met.

The **elif** print statement will run.

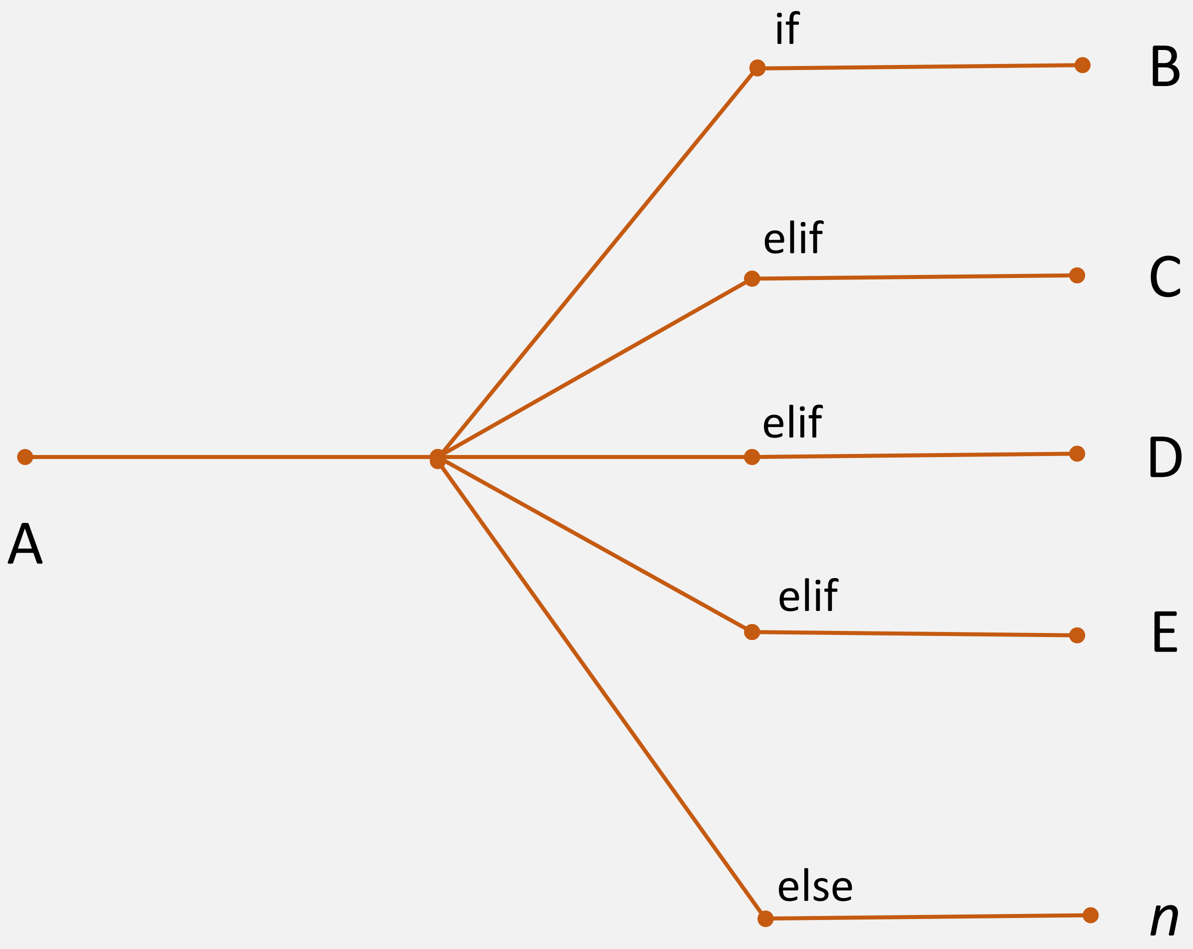
if - elif - else

```
people = 50
chairs   = 100

if people > chairs:
       print(”We need more chairs!”)


elif people == chairs:
       print(“Perfect! Great job everyone!”)


else:
       print(“We have enough chairs.”)
```

No condition is met.
The **else** print statement will run.

if - elif - else

This can be extended as many times as needed.



if - elif - else

If two separate conditions are met, only the first one will execute.

```
people = 100

if people > 5:
      print(”Some people are coming.”)


elif people > 50:
      print(“Many people are coming.”)


else:
      print(“Too many or too few people are coming.”)
```

Both conditions are met
The **first** condition’s print statement will run.

if - elif - else

```
answer = 5

guess = input(“Guess a number between 1 and 10:  ”)


if guess == answer:
       print(”Great job!”)


elif guess != answer:
       print(“Sorry, that’s incorrect.”)


else:
       print(“Something went wrong, please try again.”)
```

if - elif - else

```
answer = 5

guess = input(“Guess a number between 1 and 10:  ”)


if guess == answer:
      print(”Great job!”)


elif guess != answer:
      print(“Sorry, that’s incorrect.”)


else:
      print(“Something went wrong, please try again.”)
```

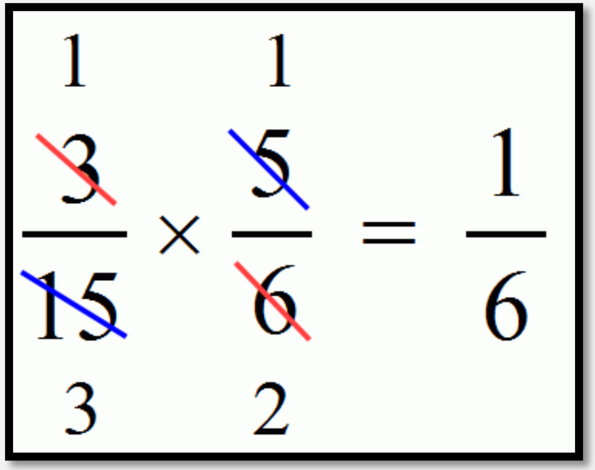
In this case, else will ALWAYS run.
Why?

if - elif - else

```
answer = 5

guess = input(“Guess a number between 1 and 10:  ”)

print(type(answer))

- - - - - - - - - - - - - - - - - - - - - - - - - - -

 < class ‘int’>

 print(type(guess))
- - - - - - - - - - - - - - - - - - - - - - - - - - -

< class ‘str’>
```

Logical and Boolean Operators

Learning to Simplify



$$\frac{\overset{1}{\cancel{3}}}{\underset{3}{\cancel{15}}} \times \frac{\overset{1}{\cancel{5}}}{\underset{2}{\cancel{6}}} = \frac{1}{6}$$

Logic Table

Command	Description
and	checks if two or more conditions are all True
or	checks if one of many conditions is True
not	checks if a condition is not True
!= (not equal)	checks if two objects are not the same
== (equal)	checks if two objects are the same
>=	checks if an object greater than or equal to something
<=	checks if an object less than or equal to something
True	checks to see if a condition is True
False	checks to see if a condition is False

and | not

Learning to Simplify

!||||!

~ ~

^

&

v

Example: and

checks if one of many conditions is True

```
# not using ‘and’
x = 10

if x > 5:
    if x < 15:
        print(“x is between 5 and 15”)

    else:
        print(“x is NOT between 5 and 15”)

else:
    print(“x is NOT between 5 and 15”)
```

Example: and

checks if two or more conditions are all True

```
# using ‘and’
x = 10

if x > 5 and x < 15:
    print(“x is between 5 and 15”)

else:
    print(“x is NOT between 5 and 15”)
```

Two full expressions must be written.

Example: and

checks if two or more conditions are all True

Two expressions must be written.
THIS WILL PRODUCE AN ERROR

```
# using ‘and’
x = 10

if x > 5 and < 15:
        print(“x is between 3 and 15”)

else:
        print(“x is NOT between 3 and 15”)
```

Example: and

checks if two or more conditions are all True

We do not need to use the same variable in each expression.

```
# using ‘and’
x = 10
y = 11

if x > 5 and y < 15:
        print(“Both conditions are met”)

else:
        print(”At least one condition is not met”)
```

Example: or

checks if at least one of many conditions is True

```
# not using ‘or’
x = ‘USA’

if x == ’Canada’:
       print(“North America”)

elif x == ‘USA’:
       print(“North America”)

elif x == ‘Mexico’
       print(”North America”)

else:
       print(“Not North America.”)
```

Example: or

checks if at least one of many conditions is True

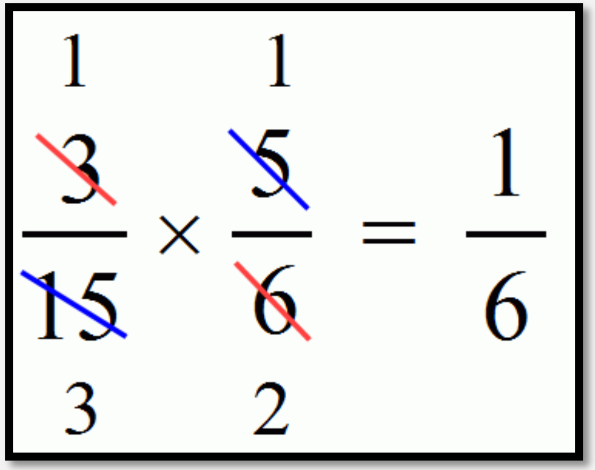
```
# using ‘or’
x = ‘USA’

if x == ’Canada’ or x == ‘USA’ or x ==
‘Mexico’:
        print(“North America”)

else:
        print(“Not North America.”)
```

Notice that we are able to use more than one or operator in the same line of code.

Menti!

Loops

for _____ in _____:



$$\frac{\overset{1}{\cancel{3}}}{\underset{3}{\cancel{15}}} \times \frac{\overset{1}{\cancel{5}}}{\underset{2}{\cancel{6}}} = \frac{1}{6}$$

Purpose

Loops are extremely useful in programming to simplify our code.

They are an alternative to copy/paste programming.

Uses the syntax “in”

adding in

in is a powerful keyword that can help make your code more user-friendly

It is used to detect if something is present inside of an iterable

- the letter “a” in the string “Chase”
- a number in the following list: [0, 1, 2, 3, 4, 5]

if - else

```
name = ‘xiong’

‘g’ in name
```

熊 (xióng) means bear in Mandarin

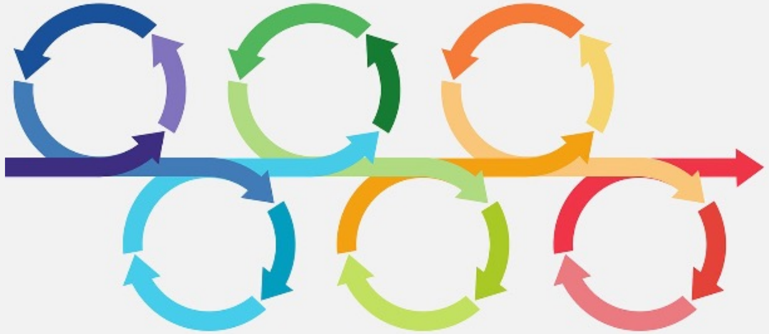
Will return True because there is a ‘g’ in ‘xiong’

```
name = ‘xiong’

‘t’ in name
```

Will return False because there is no ‘t’ in ‘xiong’

For Loops

Iteration



Purpose

for loops

used to iterate over a set (i.e. lists, arrays, columns of data, etc.)

Example: printing all items in a list one-by-one

Example: List Looping

If we create and print a list, we get the following output.

```
lst = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]

print(lst)
```

```
# Output:
['a', 'b', 'c', 'd', 'e']
```

Example: List Looping

To print each element of the list, we could do the following:

```
lst = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]

print(lst[0])           # a
print(lst[1])           # b
print(lst[2])           # c
print(lst[3])           # d
print(lst[4])           # e
```

What if we had hundreds of elements?

What if the size of our list changes?

Example: List Looping

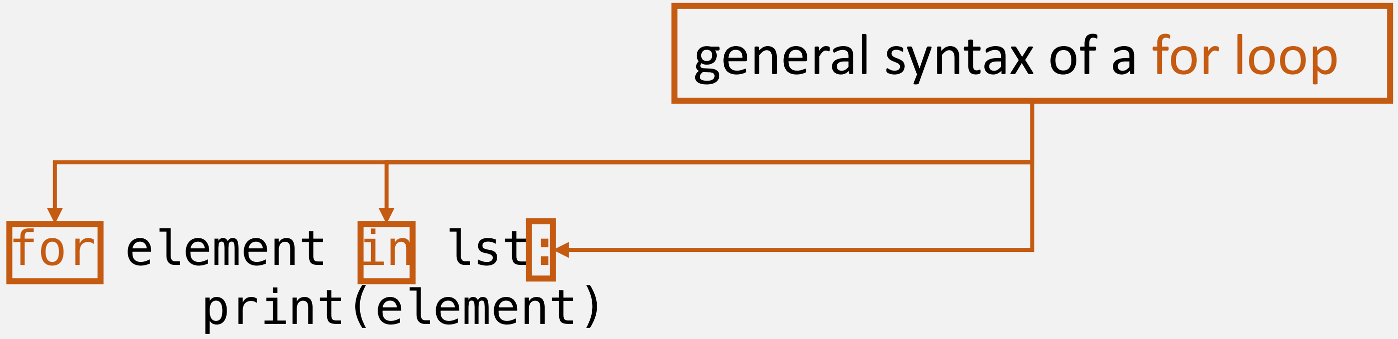
To print each element of the list, we could do the following:

```
lst = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]

for element in lst:
      print(element)
```

```
# Output:
a
b
c
d
e
```

For Loops

```
lst = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
```



general syntax of a for loop

```
for element in lst:
       print(element)
```

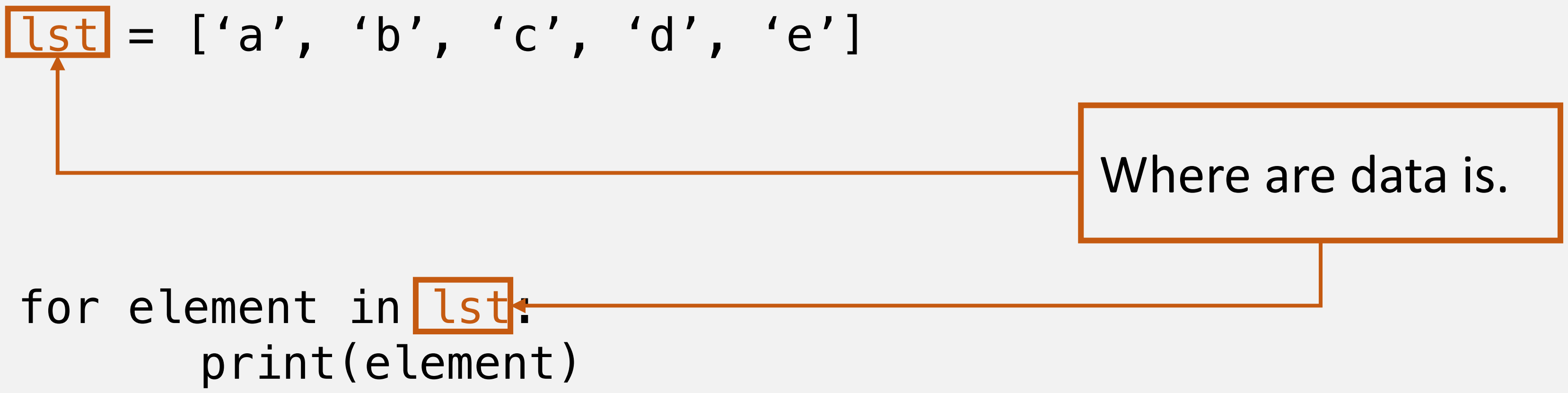
For Loops

```
lst = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
```

This name is made up on the spot.
It represents the things we are iterating over.

```
for element in lst:
     print(element)
```

For Loops



```
lst = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]


for element in lst:
        print(element)
```

Where are data is.

For Loops with Breaks

A break stops a loop if a condition is met.

```
lst = [1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,
1, 1]

for element in lst:
      print(element)
      if element == 0:
            break
```

```
# Output:
1
1
1
1
0
```

Question

Which country has the best national football team?

Menti!

Applying for loops