# STAT679 Computing for Data Science and Statistics

Lecture 2: Conditionals, Recursion and Iteration

Boolean expressions evaluate the truth/falsity of a statement

Python supplies a special Boolean type, bool variable of type bool can be either True or False

1 type(True)

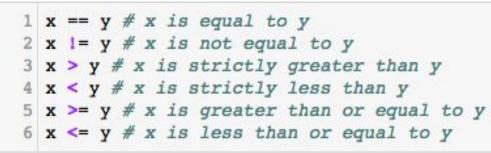
bool

1 type(False)

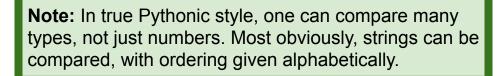
bool

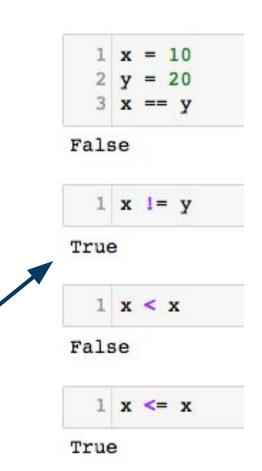
"Boolean" refers to George Boole, an English logician and philosopher. <u>https://en.wikipedia.org/wiki/George\_Boole</u>

Comparison operators available in Python:



Expressions involving comparison operators evaluate to a Boolean.





Can combine Boolean expressions into larger expressions via logical operators

In Python: and, or and not



1 X > 100 Or X

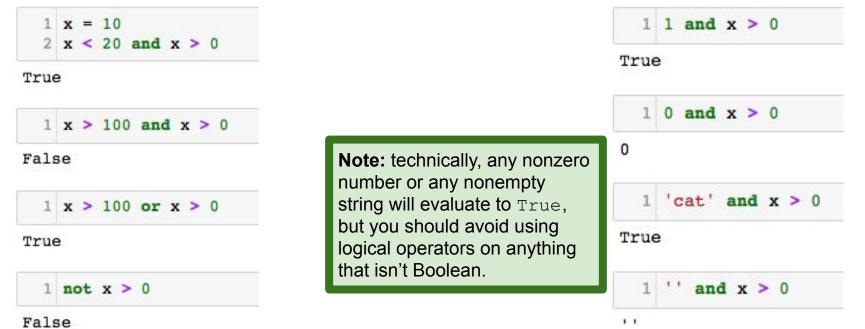
True

1 not x > 0

False

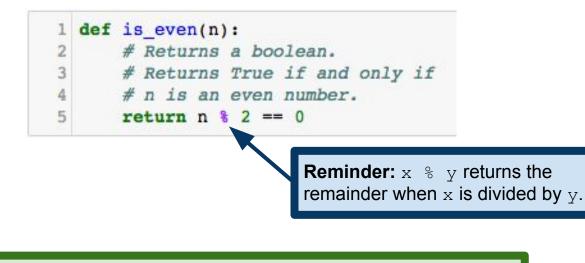
Can combine Boolean expressions into larger expressions via logical operators

In Python: and, or and not

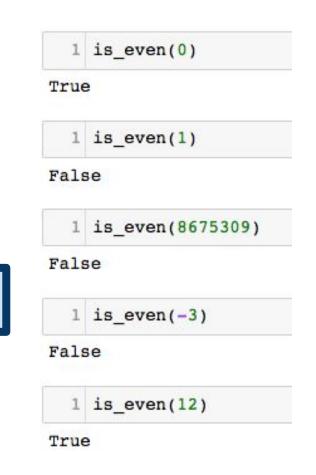


## **Boolean Expressions: Example**

Let's see Boolean expressions in action



**Note:** in practice, we would want to include some extra code to check that n is actually a number, and to "fail gracefully" if it isn't, e.g., by throwing an error with a useful error message. More about this in future lectures.

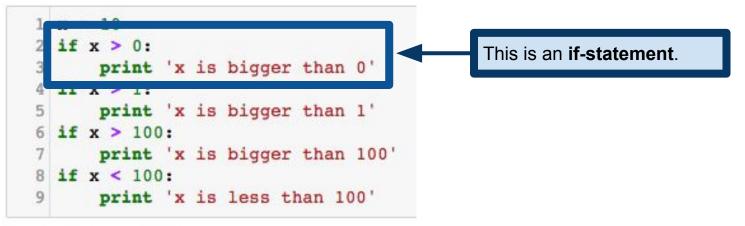


Sometimes we want to do different things depending on certain conditions

```
1 x = 10
2 if x > 0:
3     print 'x is bigger than 0'
4 if x > 1:
5     print 'x is bigger than 1'
6 if x > 100:
7     print 'x is bigger than 100'
8 if x < 100:
9     print 'x is less than 100'</pre>
```

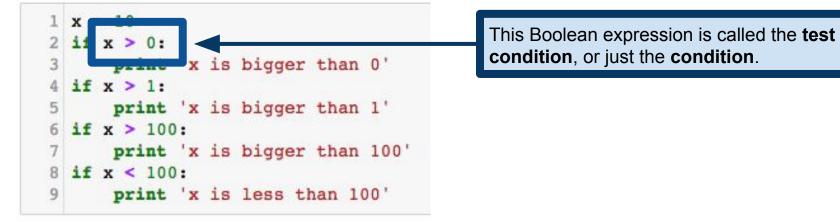
x is bigger than 0
x is bigger than 1
x is less than 100

Sometimes we want to do different things depending on certain conditions



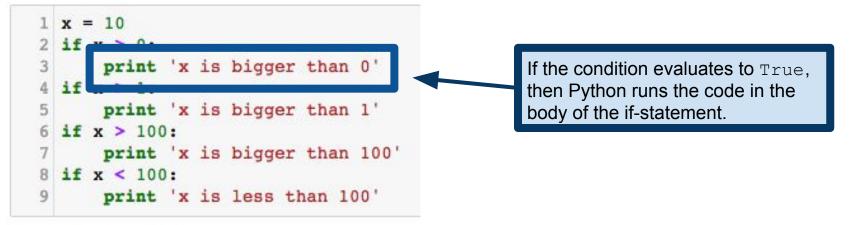
x is bigger than 0
x is bigger than 1
x is less than 100

Sometimes we want to do different things depending on certain conditions



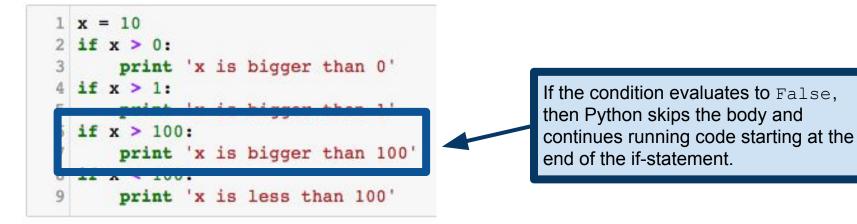
x is bigger than 0 x is bigger than 1 x is less than 100

Sometimes we want to do different things depending on certain conditions



x is bigger than 0
x is bigger than 1
x is less than 100

Sometimes we want to do different things depending on certain conditions



x is bigger than 0
x is bigger than 1
x is less than 100

Sometimes we want to do different things depending on certain conditions

```
1 \mathbf{x} = 10
  2 if x > 0:
     print 'x is bigger than 0'
  3
  4 if x > 1:
  5
      print 'x is bigger than 1'
  6 if x > 100:
      print 'x is bigger than 100'
  8 if x < 100:
        print 'x is less than 100'
  9
x is bigger than 0
                                            1 y = 20
x is bigger than 1
x is less than 100
```

**Note:** the body of a conditional statement can have any number of lines in it, but it must have at least one line. To do nothing, use the pass keyword.

```
1 y = 20
2 if y > 0:
3     pass # TODO: handle positive numbers!
4 if y < 100:
5     print 'y is less than 100'
```

y is less than 100

More complicated logic can be handled with chained conditionals

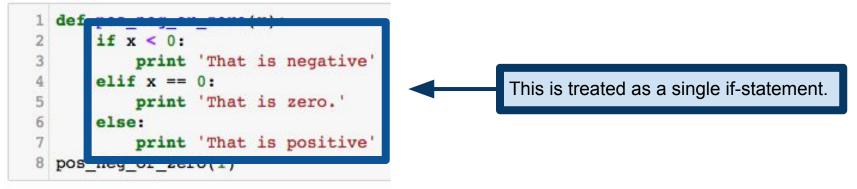
```
1 def pos_neg_or_zero(x):
2     if x < 0:
3        print 'That is negative'
4     elif x == 0:
5        print 'That is zero.'
6     else:
7        print 'That is positive'
8 pos_neg_or_zero(1)
```

That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero. That is negative That is positive

More complicated logic can be handled with chained conditionals

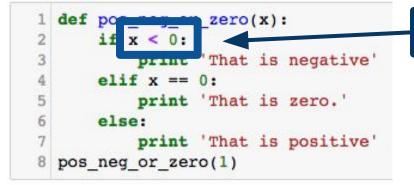


That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero. That is negative That is positive

More complicated logic can be handled with chained conditionals



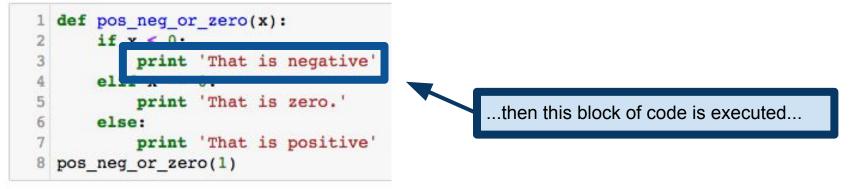
That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero. That is negative That is positive **Note:** Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

If this expression evaluates to True...

More complicated logic can be handled with chained conditionals

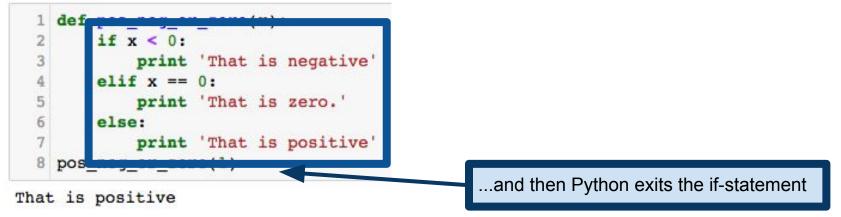


That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero. That is negative That is positive

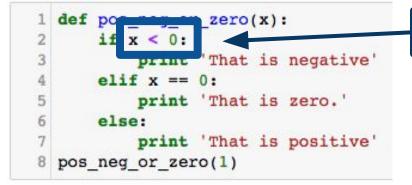
More complicated logic can be handled with chained conditionals



```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero. That is negative That is positive

More complicated logic can be handled with chained conditionals



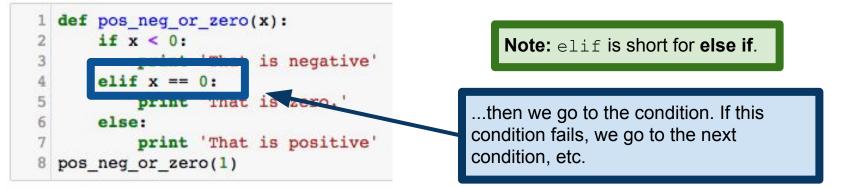
That is positive

```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero. That is negative That is positive **Note:** Oops! Still Python 2! You need to change the print statements to functions to get this to run in Python 3.

If this expression evaluates to False...

More complicated logic can be handled with chained conditionals

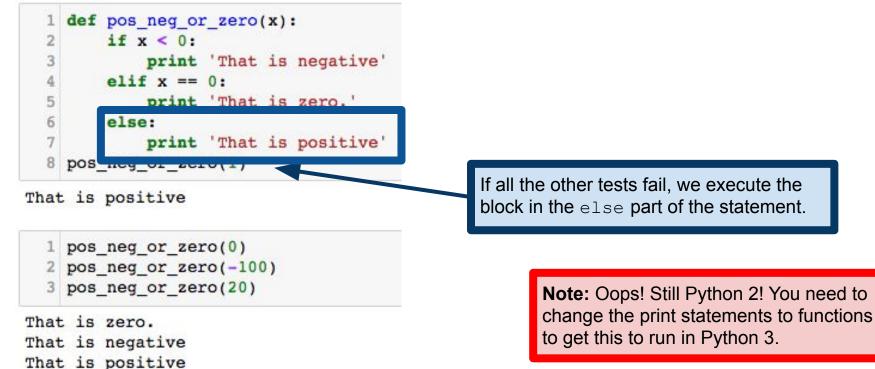


That is positive

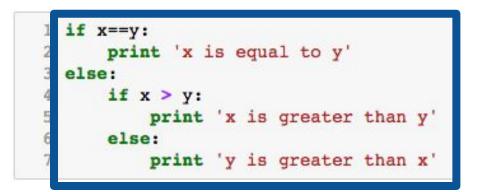
```
1 pos_neg_or_zero(0)
2 pos_neg_or_zero(-100)
3 pos_neg_or_zero(20)
```

That is zero. That is negative That is positive

More complicated logic can be handled with chained conditionals

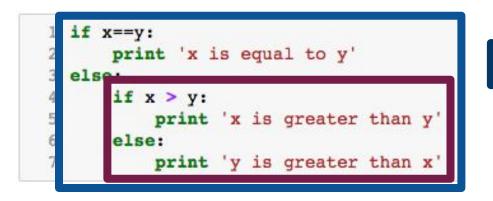


Conditionals can also be nested



This if-statement...

Conditionals can also be nested



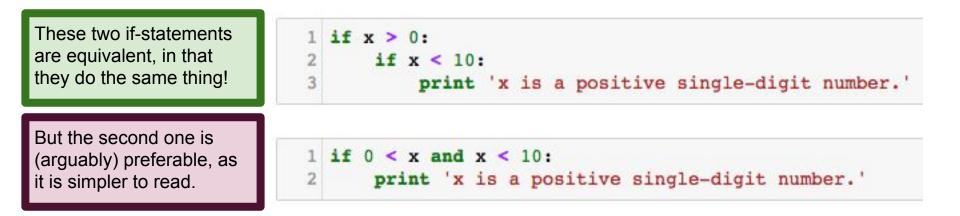
This if-statement...

...contains another if-statement.

Often, a nested conditional can be simplified

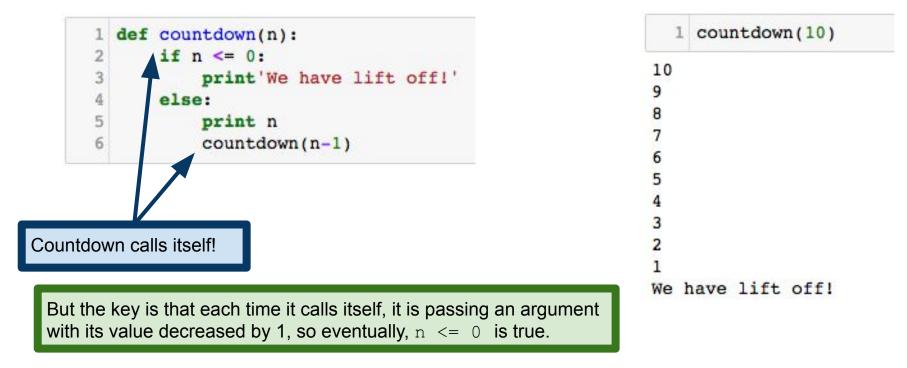
When this is possible, I recommend it for the sake of your sanity,

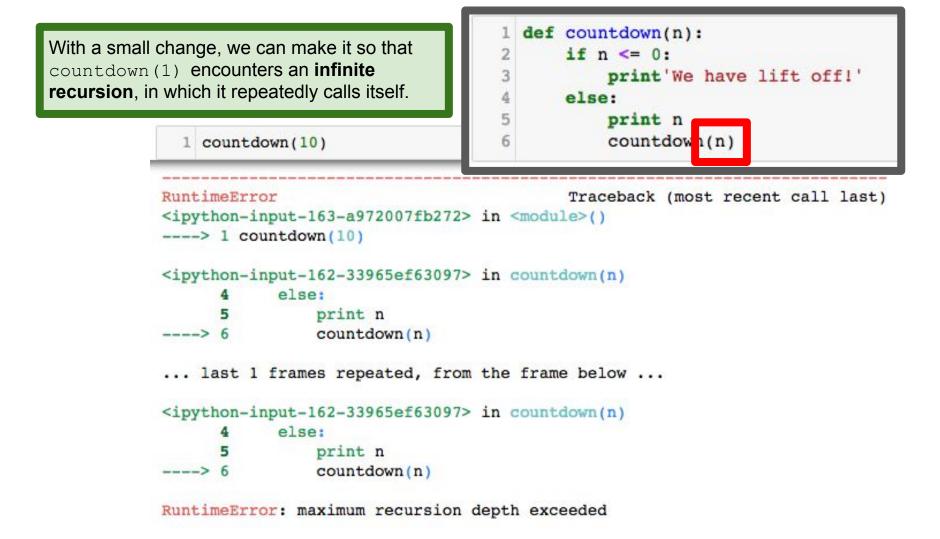
because debugging complicated nested conditionals is tricky!



#### Recursion

A function is a allowed to call itself, in what is termed **recursion** 



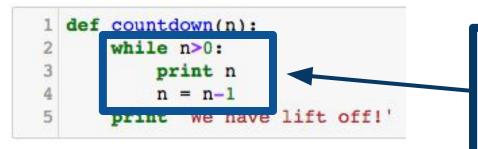


Recursion is the first tool we've seen for performing repeated operations But there are better tools for the job: while and for loops.

1	def countdown(n):					
2		while	n>0			
3		pı	rint	n		
4		n	= n-	-1		
5		print	'We	have	lift	off!'

1	L	cou	ntdow	n(10)
10				
9				
8				
7				
6				
5				
4				
3				
2				
1				
We	h	ave	lift	off!

Recursion is the first tool we've seen for performing repeated operations But there are better tools for the job: while and for loops.



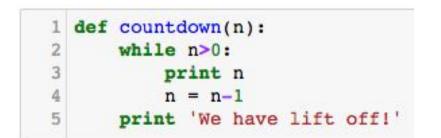
This block specifies a while-loop. So long as the condition is true, Python will run the code in the body of the loop, checking the condition again at the end of each time through.

Recursion is the first tool we've seen for performing repeated operations But there are better tools for the job: while and for loops.

1	def countdown(n):					
2		while	n>0			
3		pı	rint	n		
4		n	= n-	-1		
5		print	'We	have	lift	off!'

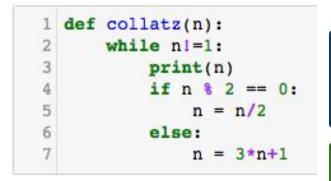
1	L	cou	ntdow	n(10)
10				
9				
8				
7				
6				
5				
4				
3				
2				
1				
We	h	ave	lift	off!

Recursion is the first tool we've seen for performing repeated operations But there are better tools for the job: while and for loops.



**Warning:** Once again, there is a danger of creating an **infinite loop**. If, for example, n never gets updated, then when we call countdown(10), the condition n>0 will always evaluate to True, and we will never exit the while-loop.





One always wants to try and ensure that a while loop will (eventually) terminate, but it's not always so easy to know! <u>https://en.wikipedia.org/wiki/Collatz\_conjecture</u>

"Mathematics may not be ready for such problems." Paul Erdős

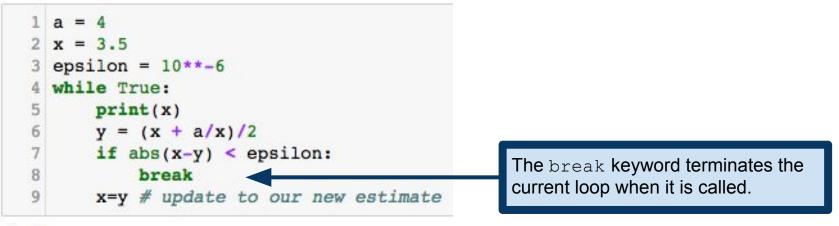
1	collatz(20)
20	
10	

5 16

8

4

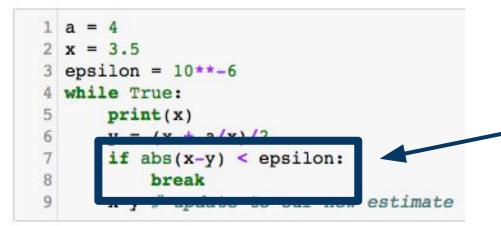
We can also terminate a while-loop using the break keyword



3.5 2.32142857143 2.02225274725 2.00012243394 2.00000000375

Newton-Raphson method: https://en.wikipedia.org/wiki/Newton's\_method

We can also terminate a while-loop using the break keyword



Notice that we're not testing for equality here. That's because testing for equality between pairs of floats is dangerous. When I write x=1/3, for example, the value of x is actually only an approximation to the number 1/3.

3.5 2.32142857143 2.02225274725 2.00012243394 2.00000000375

Newton-Raphson method: https://en.wikipedia.org/wiki/Newton's\_method

## Aside: floating-point arithmetic

The number 1/3 has no finite decimal representation, i.e., 1/3 = 0.33333333... ...but our computer uses only finitely many bits to represent a float...

One solution: symbolic computation <u>https://en.wikipedia.org/wiki/Computer\_algebra</u>

More practical: floating-point arithmetic <u>https://en.wikipedia.org/wiki/Floating-point\_arithmetic</u> **Idea:** essentially, pick out a finite set of decimal numbers... ...round variables, solutions, etc to the nearest such number

Drawback: introduces rounding errors e.g., 0.3333333333... becomes 0.333332.

For more, see Goldberg (1991), *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, **23**(1).