STATS 507
Data Analysis in Python

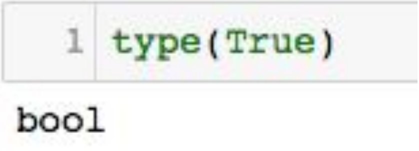
Lecture 2: Conditionals, Recursion, Iteration and Strings

Boolean Expressions

Boolean expressions evaluate the truth/falsity of a statement

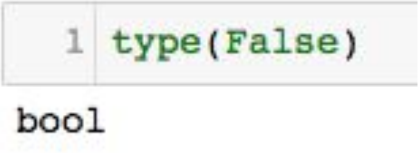
Python supplies a special Boolean type, `bool`

variable of type `bool` can be either `True` or `False`



```
type(True)
```

```
bool
```

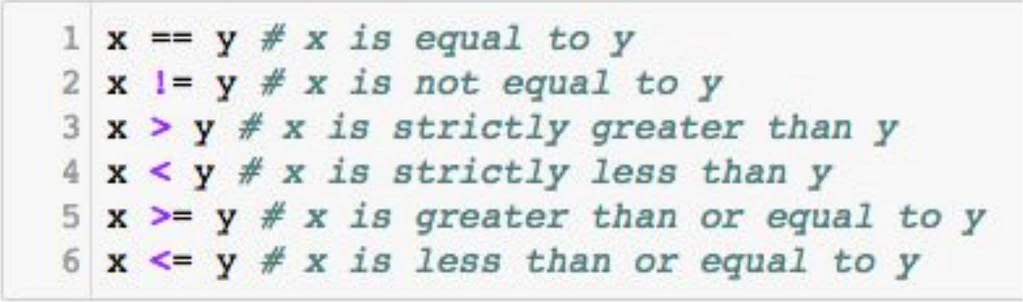


```
type(False)
```

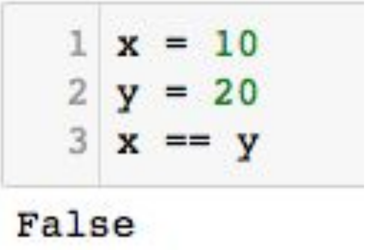
```
bool
```

Boolean Expressions

Comparison operators available in Python:

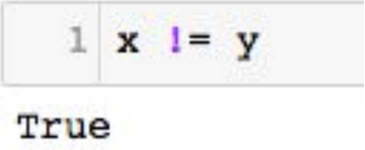


```python
x == y # x is equal to y
x != y # x is not equal to y
x > y # x is strictly greater than y
x < y # x is strictly less than y
x >= y # x is greater than or equal to y
x <= y # x is less than or equal to y
```



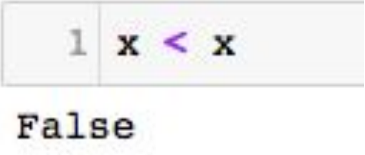
```python
x = 10
y = 20
x == y
```

```
False
```



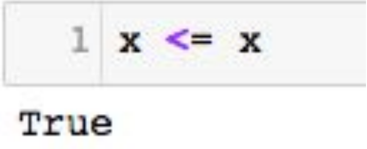
```python
x != y
```

```
True
```



```python
x < x
```

```
False
```



```python
x <= x
```

```
True
```

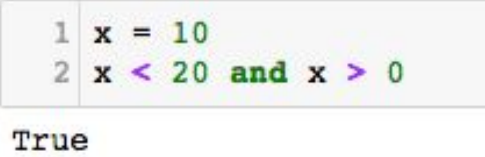
Expressions involving comparison operators evaluate to a Boolean.

Note: In true Pythonic style, one can compare many types, not just numbers. Most obviously, strings can be compared, with ordering given alphabetically.

Boolean Expressions

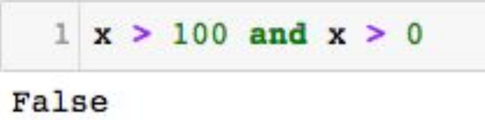
Can combine Boolean expressions into larger expressions via **logical operators**

In Python: `and`, `or` and `not`



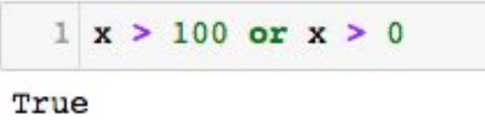
```
x = 10
x < 20 and x > 0
```

```
True
```



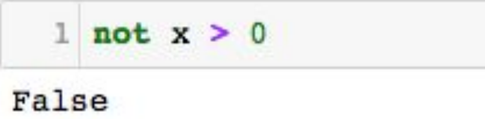
```
x > 100 and x > 0
```

```
False
```



```
x > 100 or x > 0
```

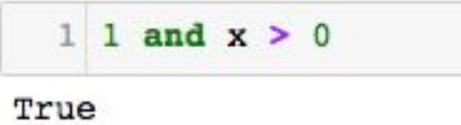
```
True
```



```
not x > 0
```

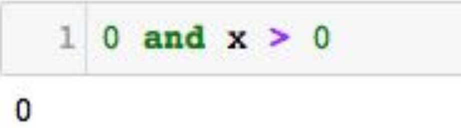
```
False
```

Note: technically, any nonzero number or any nonempty string will evaluate to `True`, but you should avoid comparing anything that isn't Boolean.



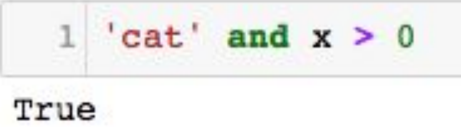
```
1 and x > 0
```

```
True
```



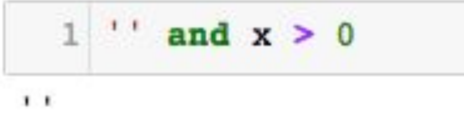
```
0 and x > 0
```

```
0
```



```
'cat' and x > 0
```

```
True
```

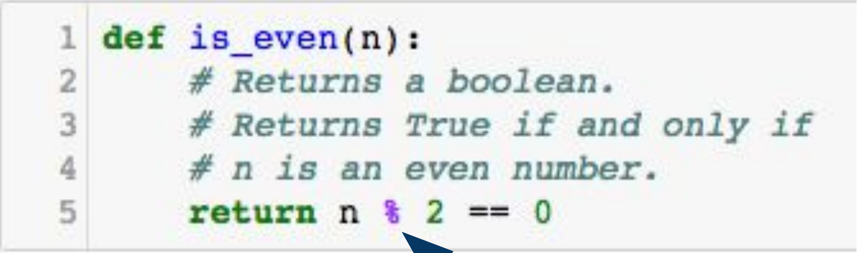


```
'' and x > 0
```

```
''
```

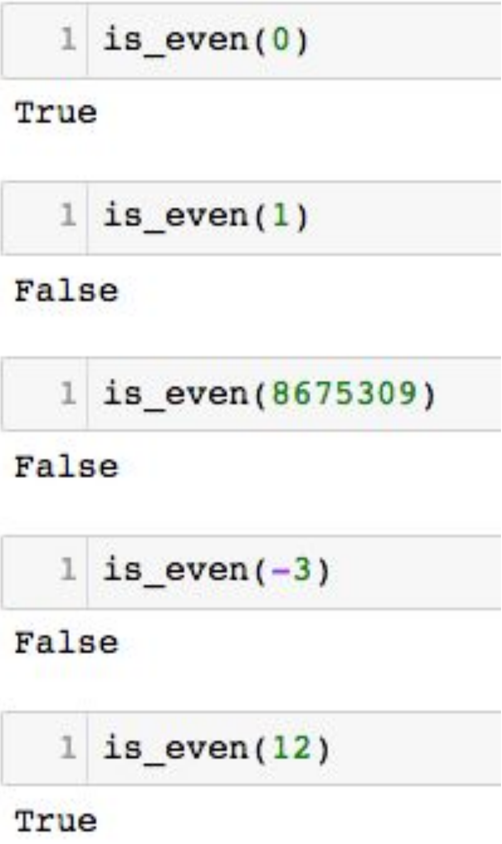
Boolean Expressions: Example

Let’s see Boolean expressions in action



```
def is_even(n):
    # Returns a boolean.
    # Returns True if and only if
    # n is an even number.
    return n % 2 == 0
```

Reminder: `x % y` returns the remainder when `x` is divided by `y`.



```
is_even(0)
```

```
True
```

```
is_even(1)
```

```
False
```

```
is_even(8675309)
```

```
False
```

```
is_even(-3)
```

```
False
```

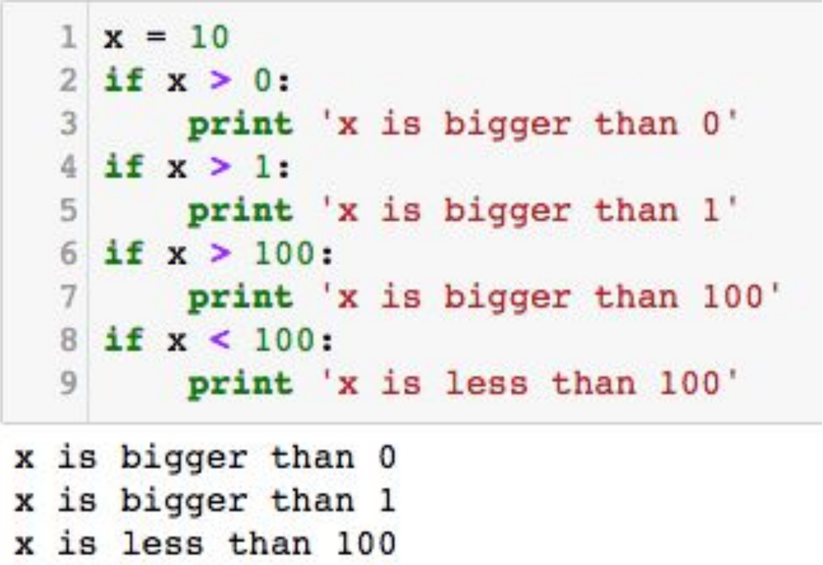
```
is_even(12)
```

```
True
```

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.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions

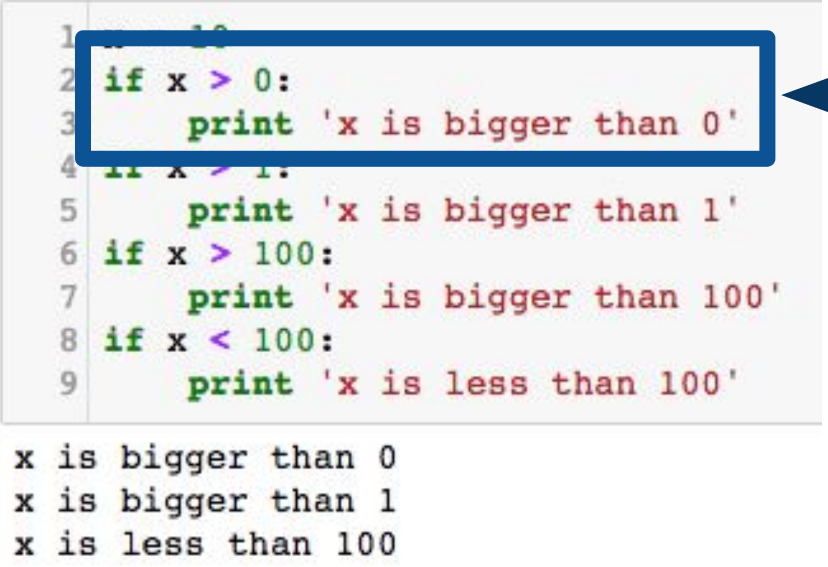


```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

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

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



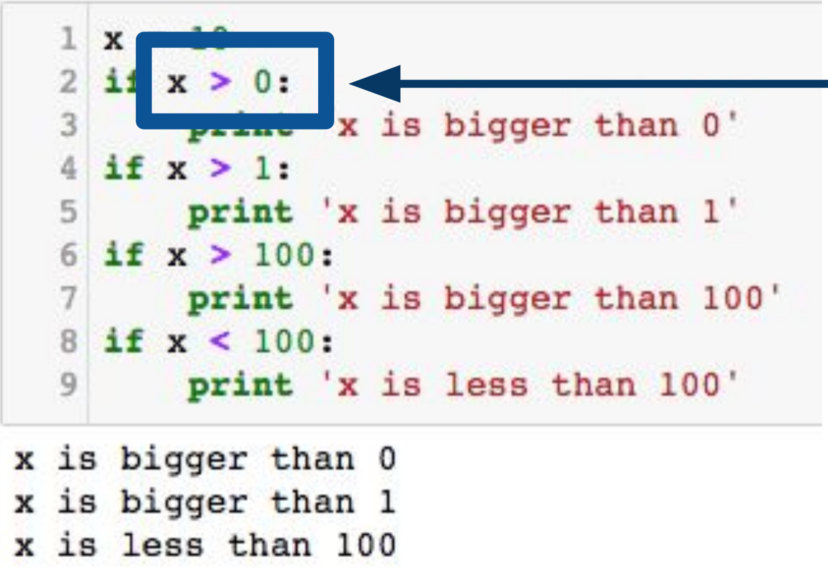
```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

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

This is an **if-statement**.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



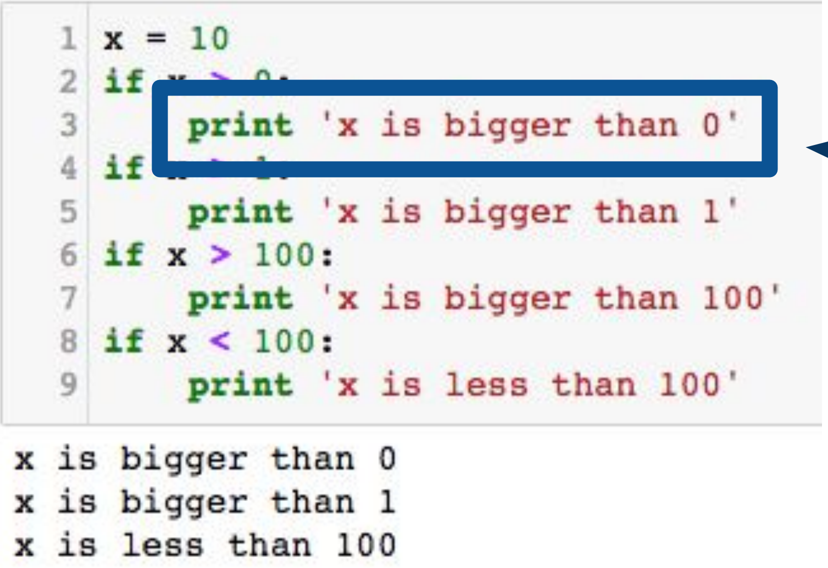
```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

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

This Boolean expression is called the **test condition**, or just the **condition**.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



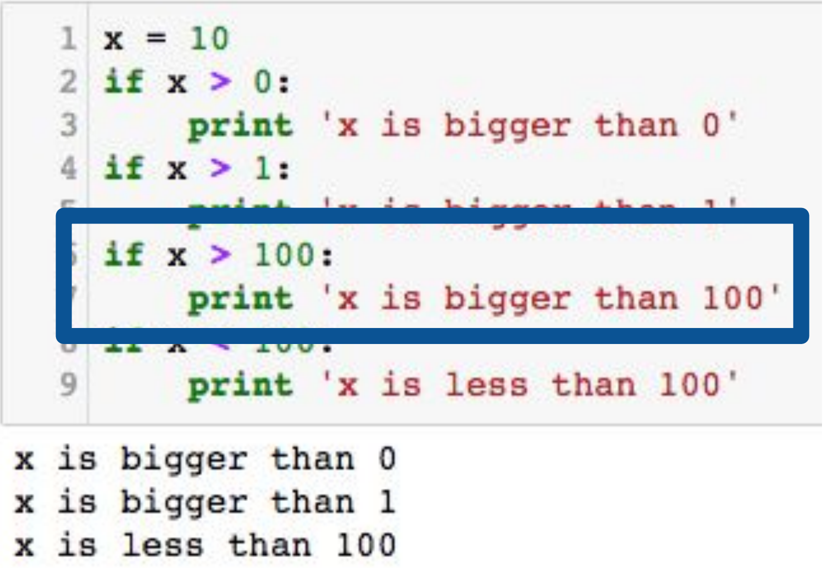
```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

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

If the condition evaluates to `True`, then Python runs the code in the body of the if-statement.

Conditional Expressions

Sometimes we want to do different things depending on certain conditions



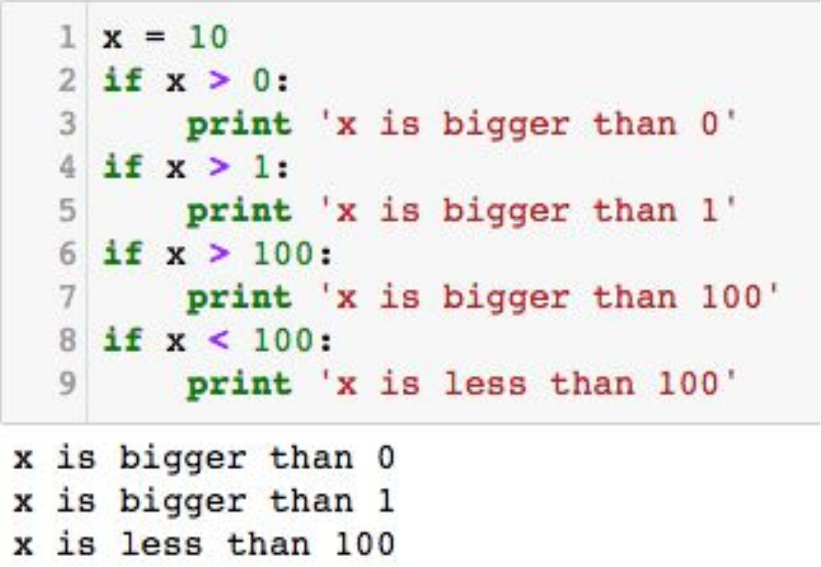
```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

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

If the condition evaluates to `False`, then Python skips the body and continues running code starting at the end of the if-statement.

Conditional Expressions

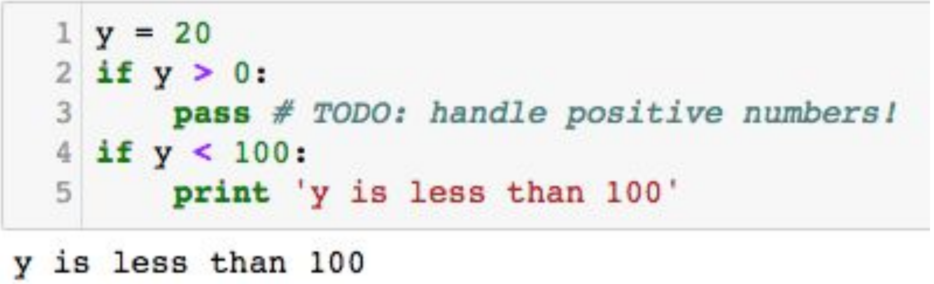
Sometimes we want to do different things depending on certain conditions



```
x = 10
if x > 0:
    print 'x is bigger than 0'
if x > 1:
    print 'x is bigger than 1'
if x > 100:
    print 'x is bigger than 100'
if x < 100:
    print 'x is less than 100'
```

```
x is bigger than 0
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.

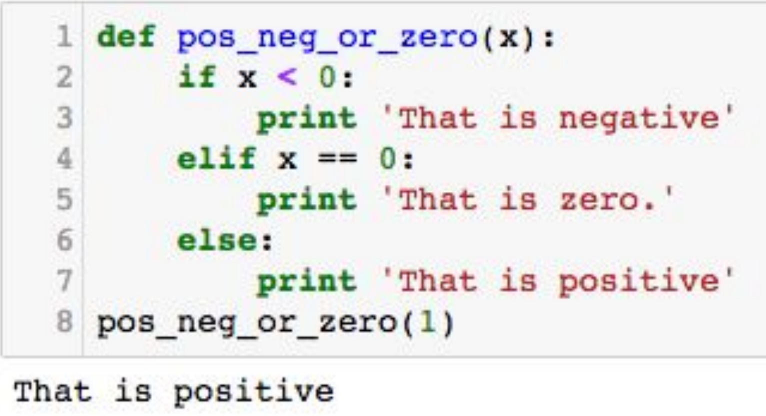


```
y = 20
if y > 0:
    pass # TODO: handle positive numbers!
if y < 100:
    print 'y is less than 100'
```

```
y is less than 100
```

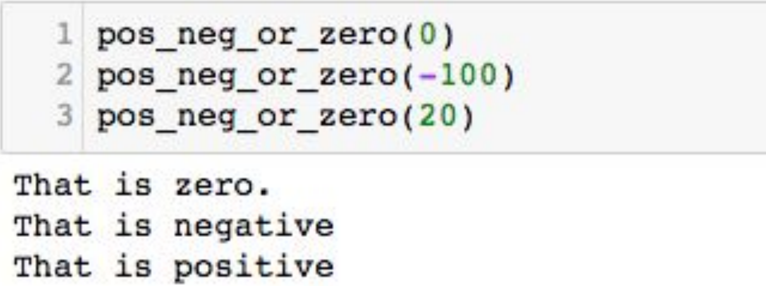
Conditional Expressions

More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

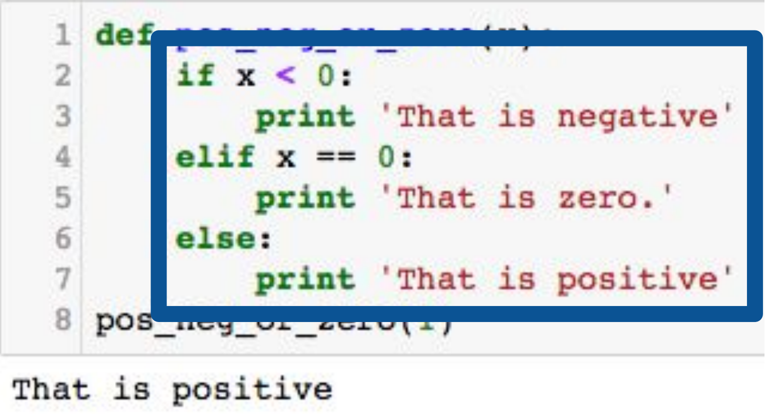


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

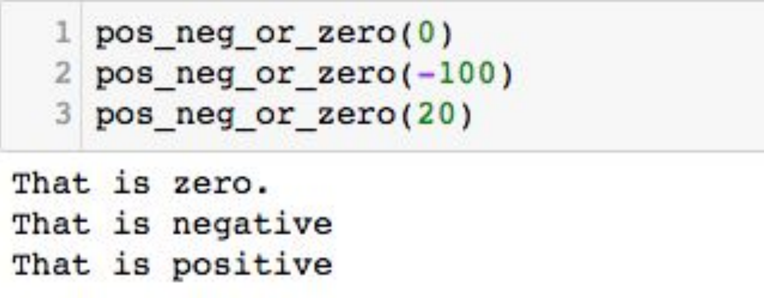
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

This is treated as a single if-statement.

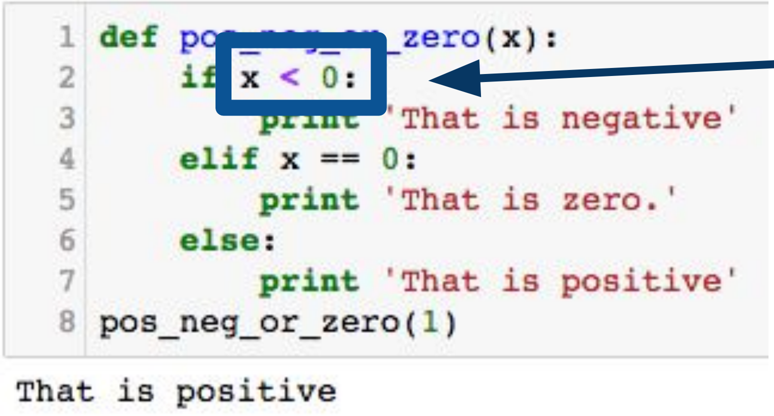


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

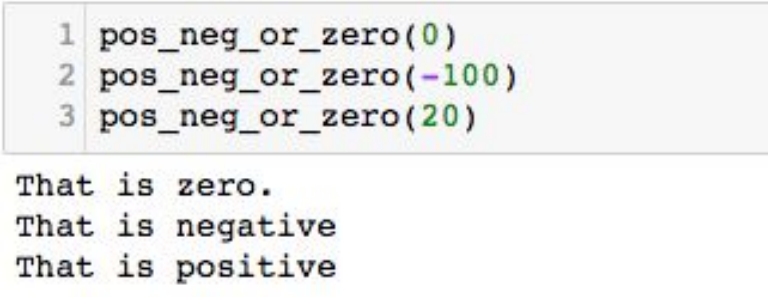
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

If this expression evaluates to `True`...

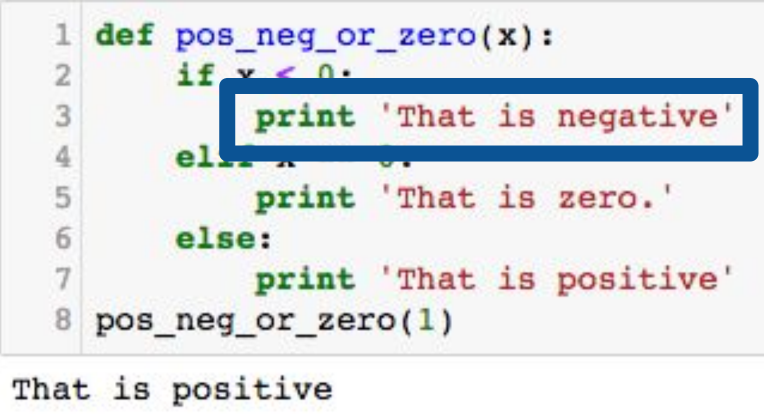


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

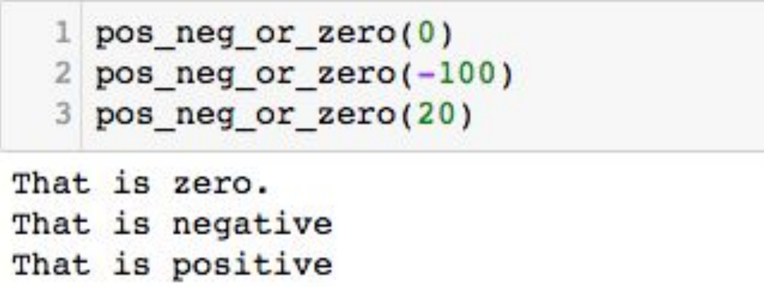
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

...then this block of code is executed...

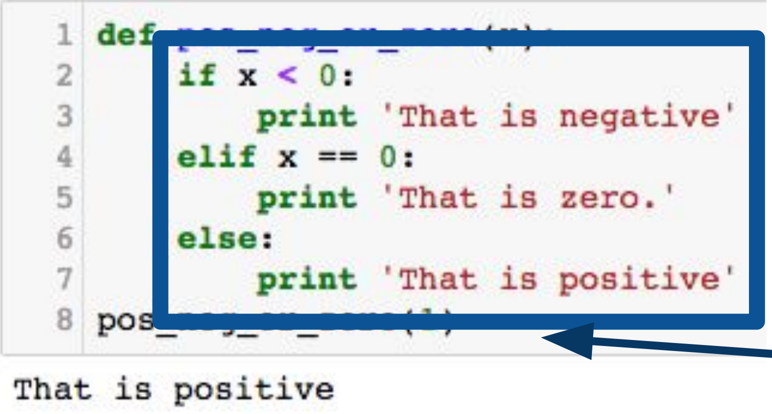


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

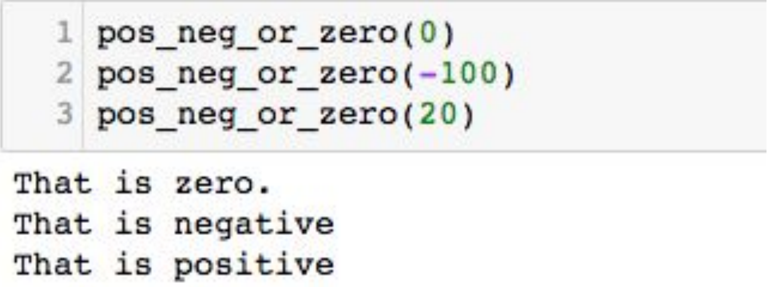
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

...and then Python exits the if-statement

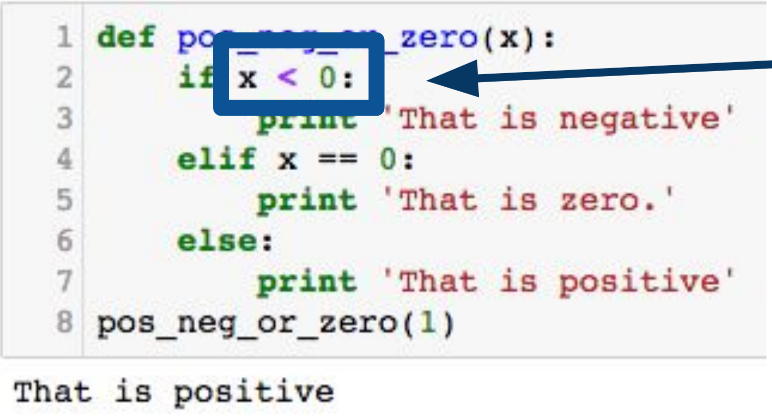


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

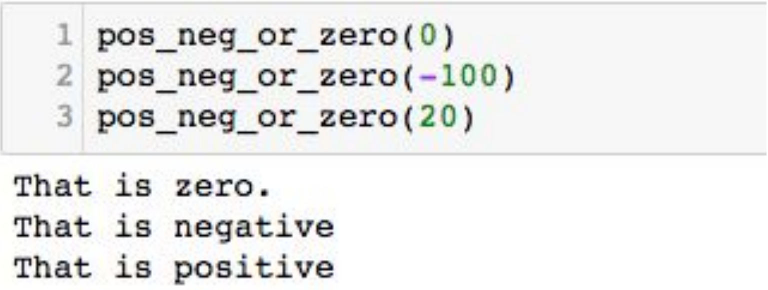
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

If this expression evaluates to `False`...

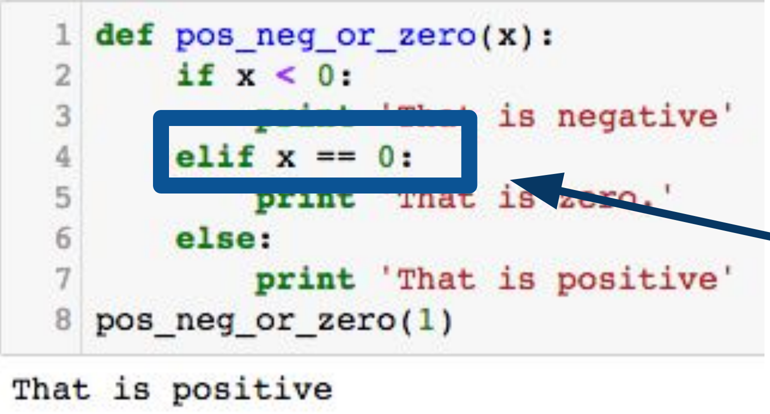


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

More complicated logic can be handled with **chained conditionals**

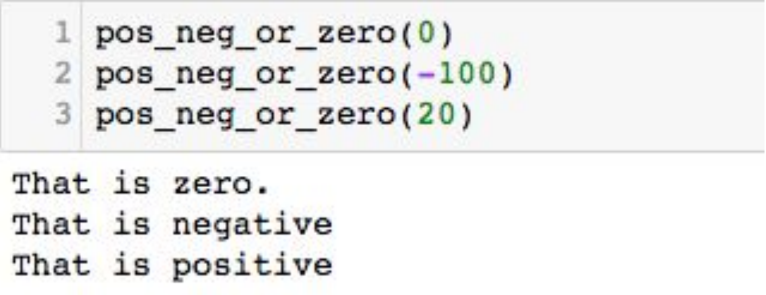


```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

Note: `elif` is short for **else if**.

...then we go to the condition. If this condition fails, we go to the next condition, etc.

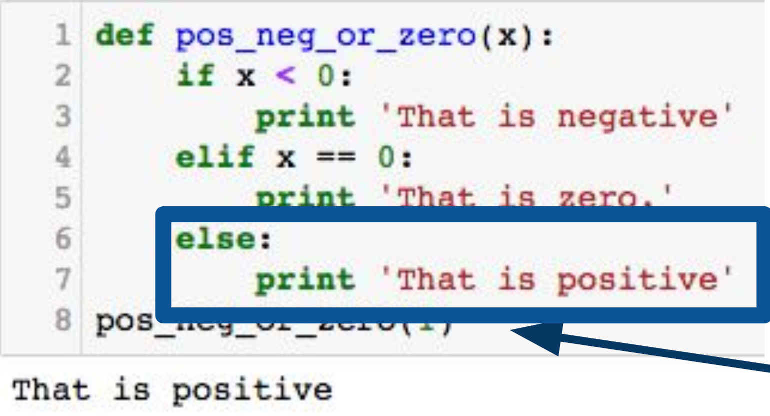


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

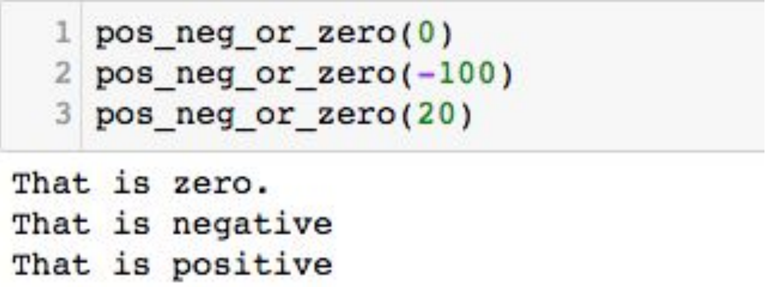
More complicated logic can be handled with **chained conditionals**



```
def pos_neg_or_zero(x):
    if x < 0:
        print 'That is negative'
    elif x == 0:
        print 'That is zero.'
    else:
        print 'That is positive'
pos_neg_or_zero(1)
```

```
That is positive
```

If all the other tests fail, we execute the block in the `else` part of the statement.

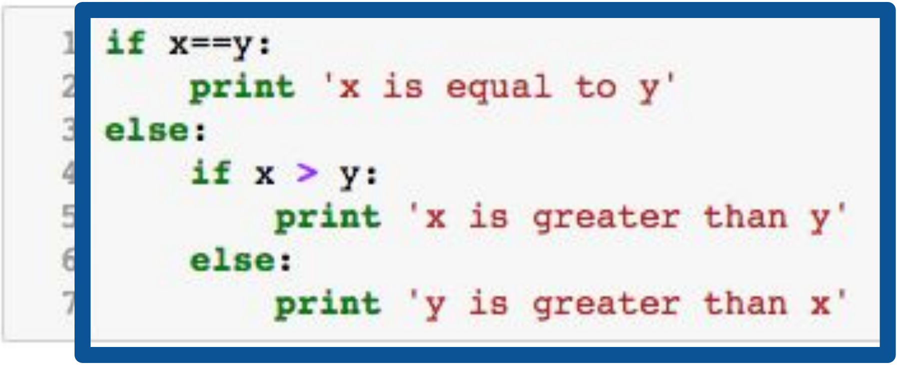


```
pos_neg_or_zero(0)
pos_neg_or_zero(-100)
pos_neg_or_zero(20)
```

```
That is zero.
That is negative
That is positive
```

Conditional Expressions

Conditionals can also be nested

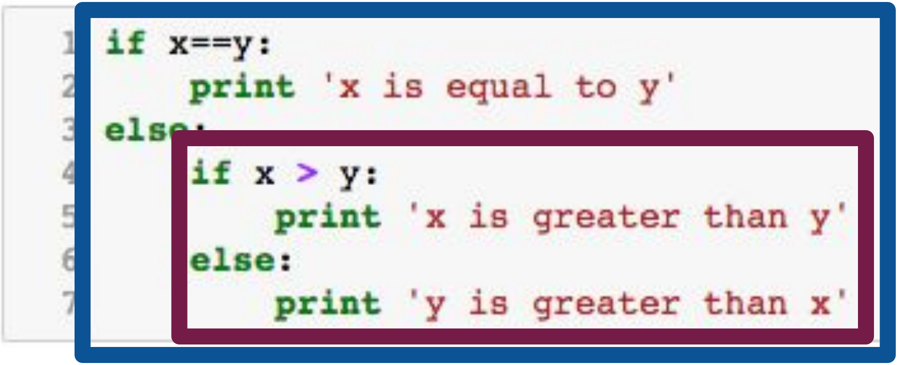


```
if x==y:
    print 'x is equal to y'
else:
    if x > y:
        print 'x is greater than y'
    else:
        print 'y is greater than x'
```

This if-statement...

Conditional Expressions

Conditionals can also be nested



```
if x==y:
    print 'x is equal to y'
else:
    if x > y:
        print 'x is greater than y'
    else:
        print 'y is greater than x'
```

This if-statement...

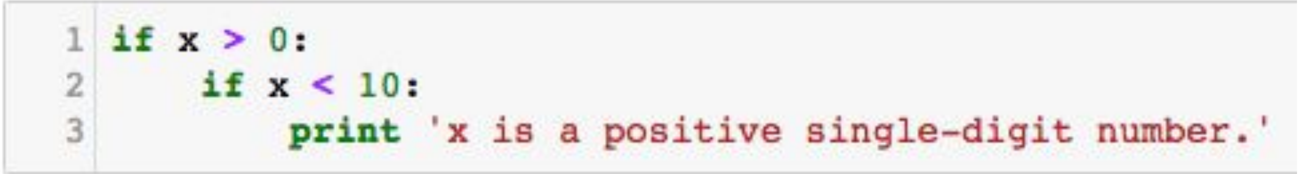
...contains another if-statement.

Conditional Expressions

Often, a nested conditional can be simplified

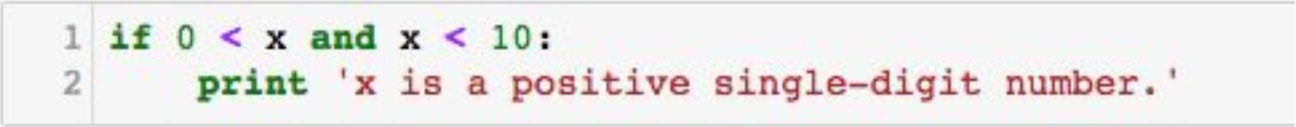
When this is possible, I recommend it for the sake of your sanity, because debugging complicated nested conditionals is tricky!

These two if-statements are equivalent, in that they do the same thing!



```
if x > 0:
    if x < 10:
        print 'x is a positive single-digit number.'
```

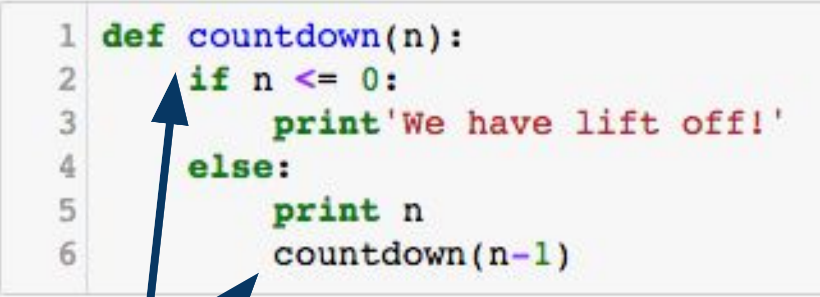
But the second one is (arguably) preferable, as it is simpler to read.



```
if 0 < x and x < 10:
    print 'x is a positive single-digit number.'
```

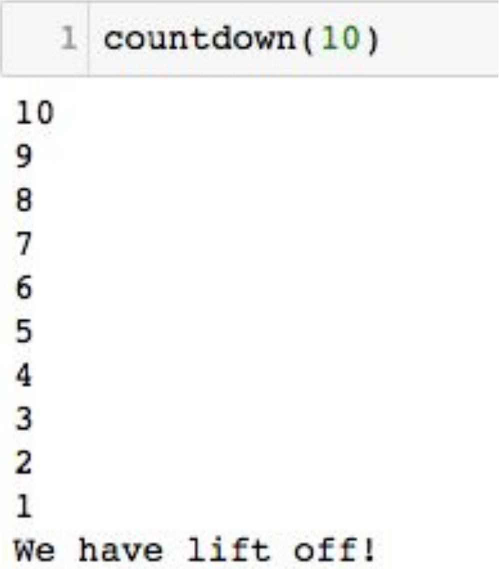
Recursion

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



```python
def countdown(n):
    if n <= 0:
        print'We have lift off!'
    else:
        print n
        countdown(n-1)
```

Countdown calls itself!

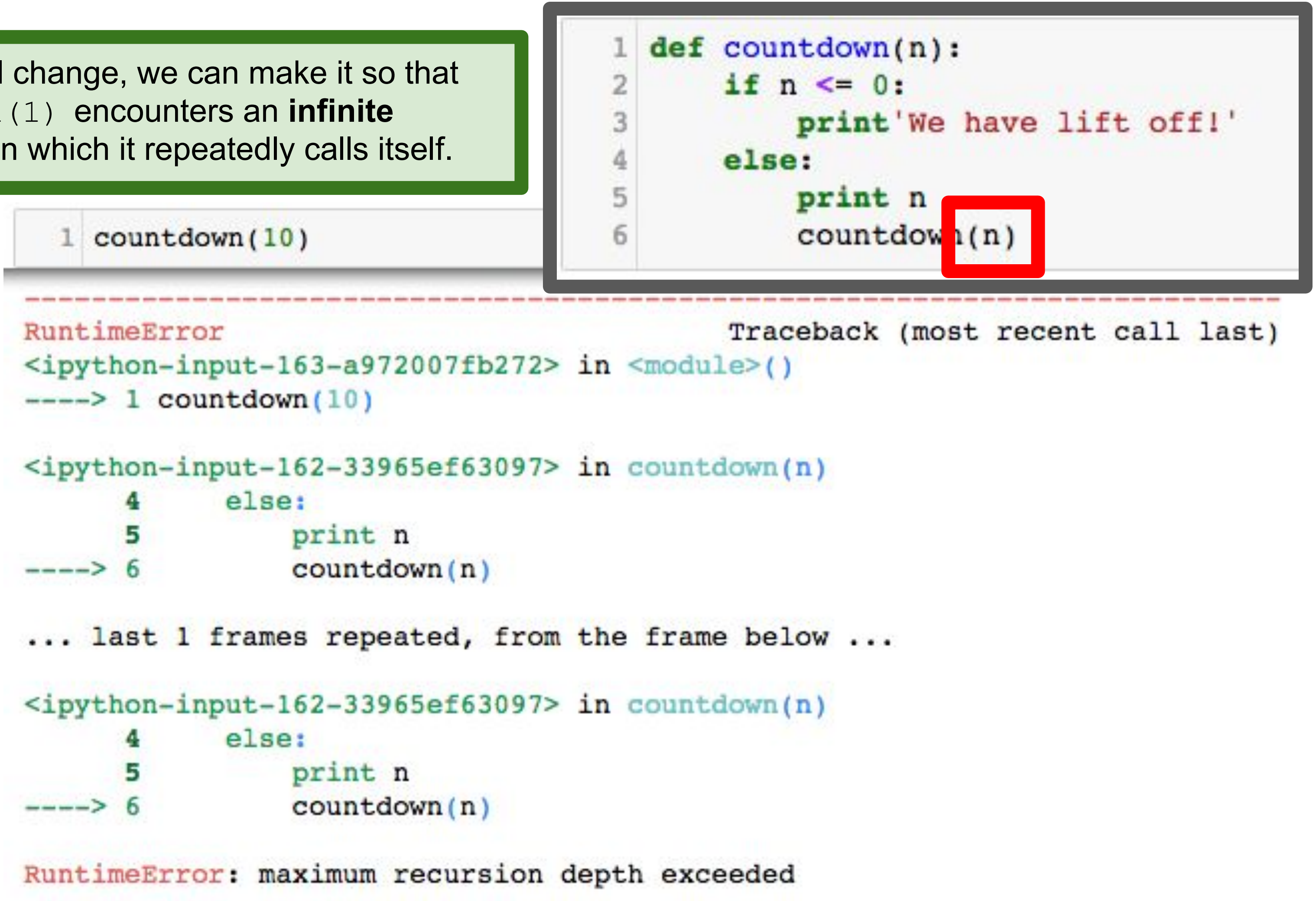


```
countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

But the key is that each time it calls itself, it is passing an argument with its value decreased by 1, so eventually, `n <= 0` is true.

With a small change, we can make it so that `countdown(1)` encounters an **infinite recursion**, in which it repeatedly calls itself.



```
def countdown(n):
    if n <= 0:
        print'We have lift off!'
    else:
        print n
        countdown(n)
```

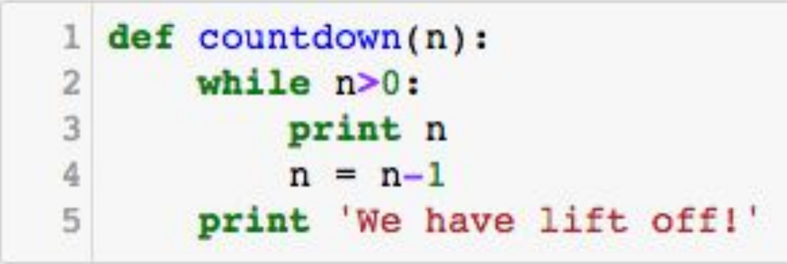
```
countdown(10)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-163-a972007fb272> in <module>()
----> 1 countdown(10)

<ipython-input-162-33965ef63097> in countdown(n)
      4     else:
      5         print n
----> 6         countdown(n)

... last 1 frames repeated, from the frame below ...

<ipython-input-162-33965ef63097> in countdown(n)
      4     else:
      5         print n
----> 6         countdown(n)

RuntimeError: maximum recursion depth exceeded
```

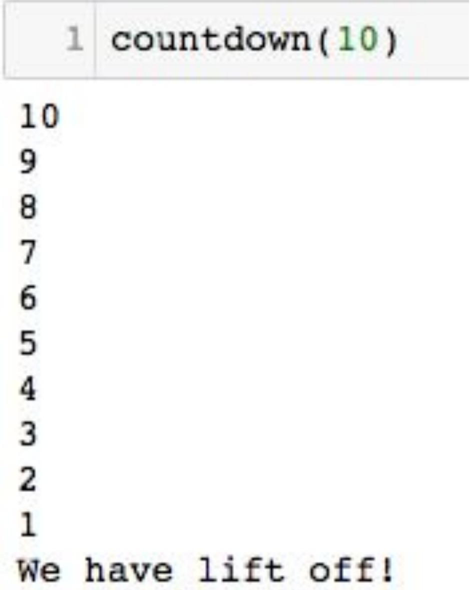
Repeated actions: Iteration

Recursion is the first tool we’ve seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.



```
def countdown(n):
    while n>0:
        print n
        n = n-1
    print 'We have lift off!'
```



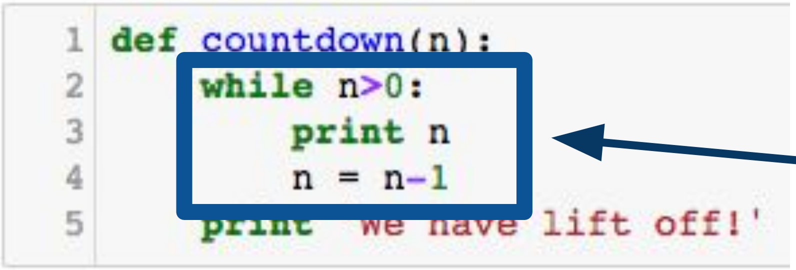
```
countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

Repeated actions: Iteration

Recursion is the first tool we’ve seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.



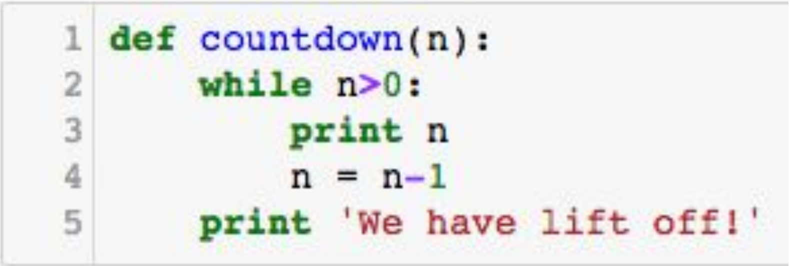
```
def countdown(n):
    while n>0:
        print n
        n = n-1
    print 'We have lift off!'
```

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.

Repeated actions: Iteration

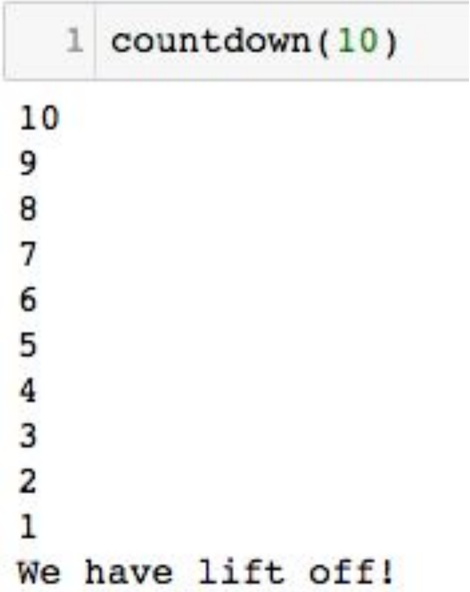
Recursion is the first tool we've seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.



```
def countdown(n):
    while n>0:
        print n
        n = n-1
    print 'We have lift off!'
```

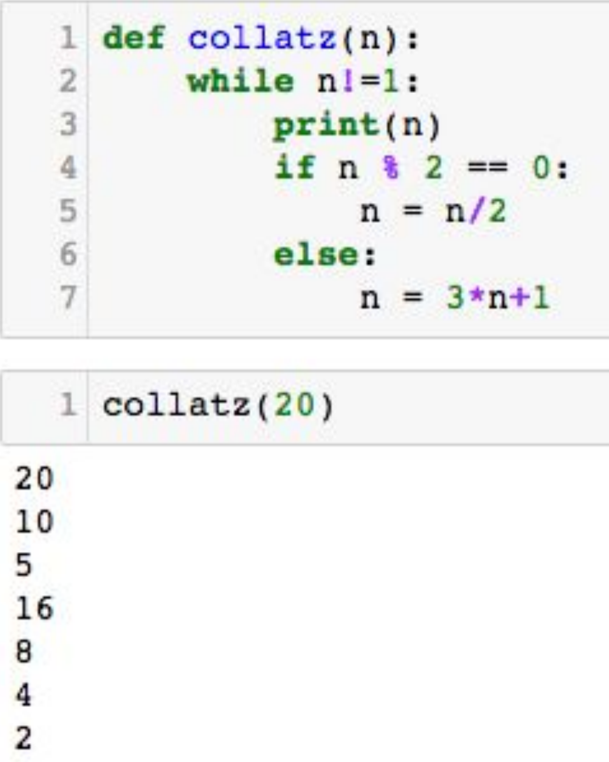
Warning: Once again, there is a danger of creating an **infinite loop**. If, for example, `n` never gets updated, then when we call `countdown(10)` , the condition `n>0` will always evaluate to `True`, and we will never exit the while-loop.



```
countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

Repeated actions: Iteration



```python
def collatz(n):
    while n!=1:
        print(n)
        if n % 2 == 0:
            n = n/2
        else:
            n = 3*n+1
```

```python
collatz(20)
```

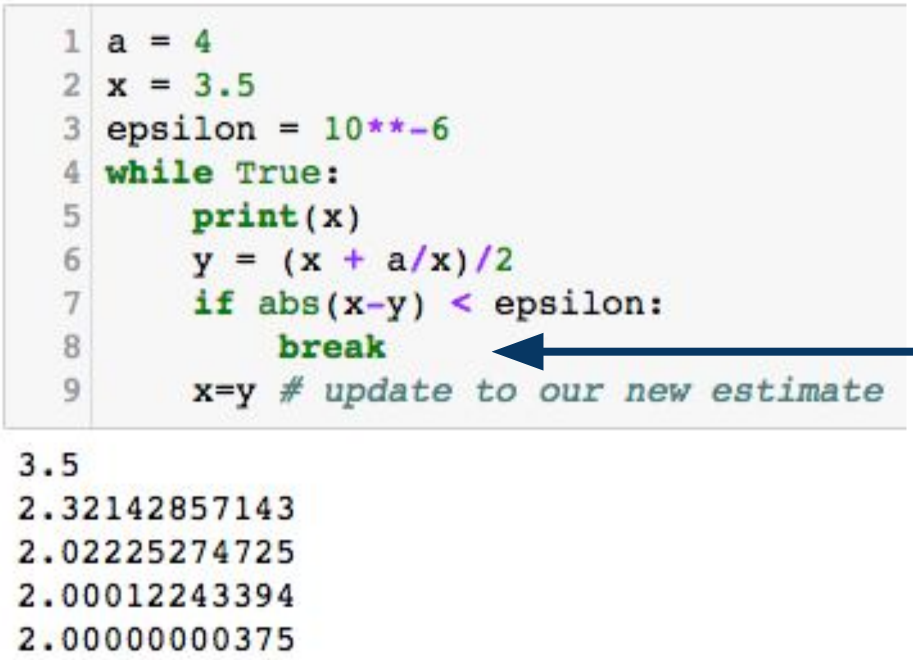
```
20
10
5
16
8
4
2
```

One always wants to try and ensure that a while loop will (eventually) terminate, but it’s not always so easy to know! https://en.wikipedia.org/wiki/Collatz_conjecture

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

Repeated actions: Iteration

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



```
a = 4
x = 3.5
epsilon = 10**-6
while True:
    print(x)
    y = (x + a/x)/2
    if abs(x-y) < epsilon:
        break
    x=y # update to our new estimate
```

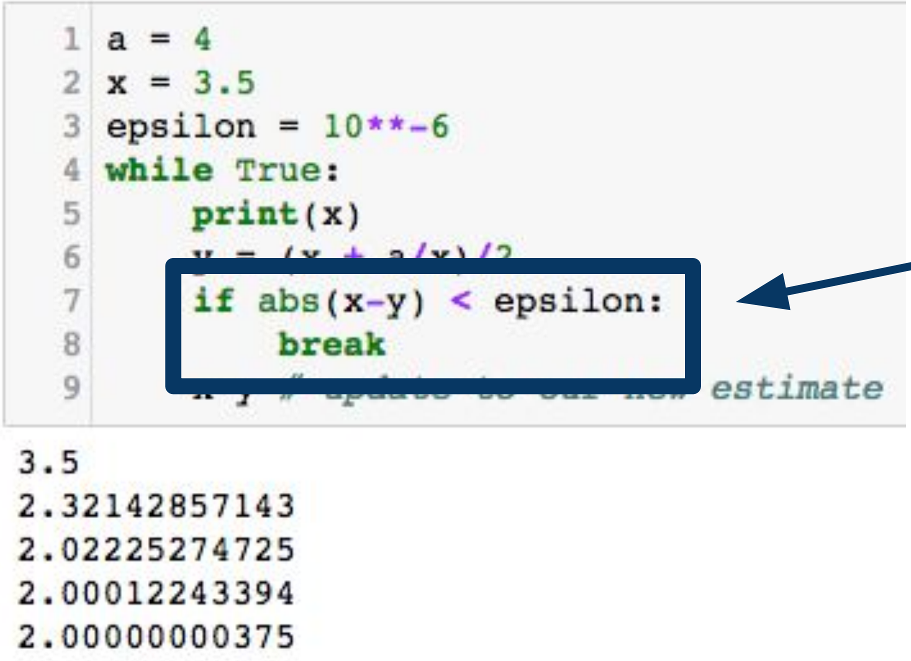
```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

The `break` keyword terminates the current loop when it is called.

Newton-Raphson method: https://en.wikipedia.org/wiki/Newton's_method

Repeated actions: Iteration

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



```
a = 4
x = 3.5
epsilon = 10**-6
while True:
    print(x)
    y = (x + a/x)/2
    if abs(x-y) < epsilon:
        break
    x = y # update to our new estimate
```

```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

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.

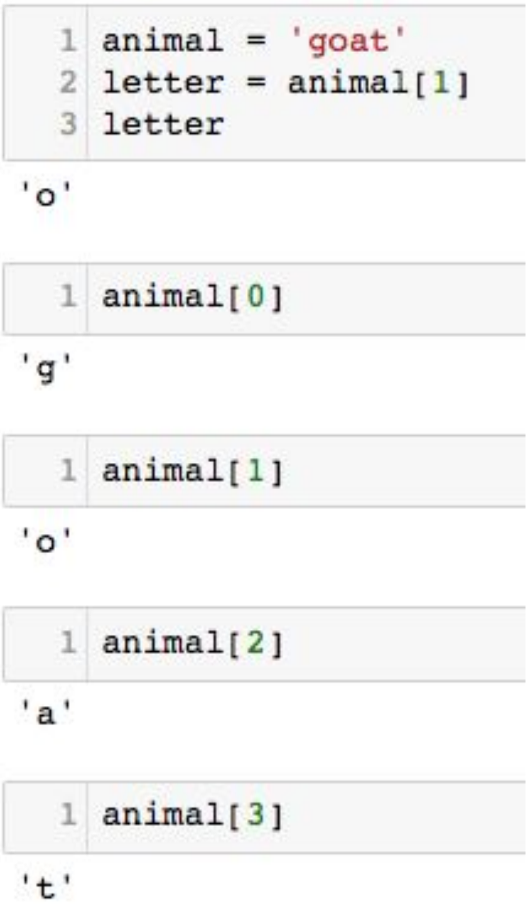
Newton-Raphson method: https://en.wikipedia.org/wiki/Newton's_method

Strings in Python

Strings are sequences of characters

Python sequences are 0-indexed. The index counts the offset from the beginning of the sequence. So the first letter is the 0-th character of the string.

Note: in some languages, there’s a difference between a character and a string of length 1. That is, the character ‘g’ and the string “g” are different data types. In Python, no such difference exists. A character is just a one-character string.



```
animal = 'goat'
letter = animal[1]
letter
```

```
'o'
```

```
animal[0]
```

```
'g'
```

```
animal[1]
```

```
'o'
```

```
animal[2]
```

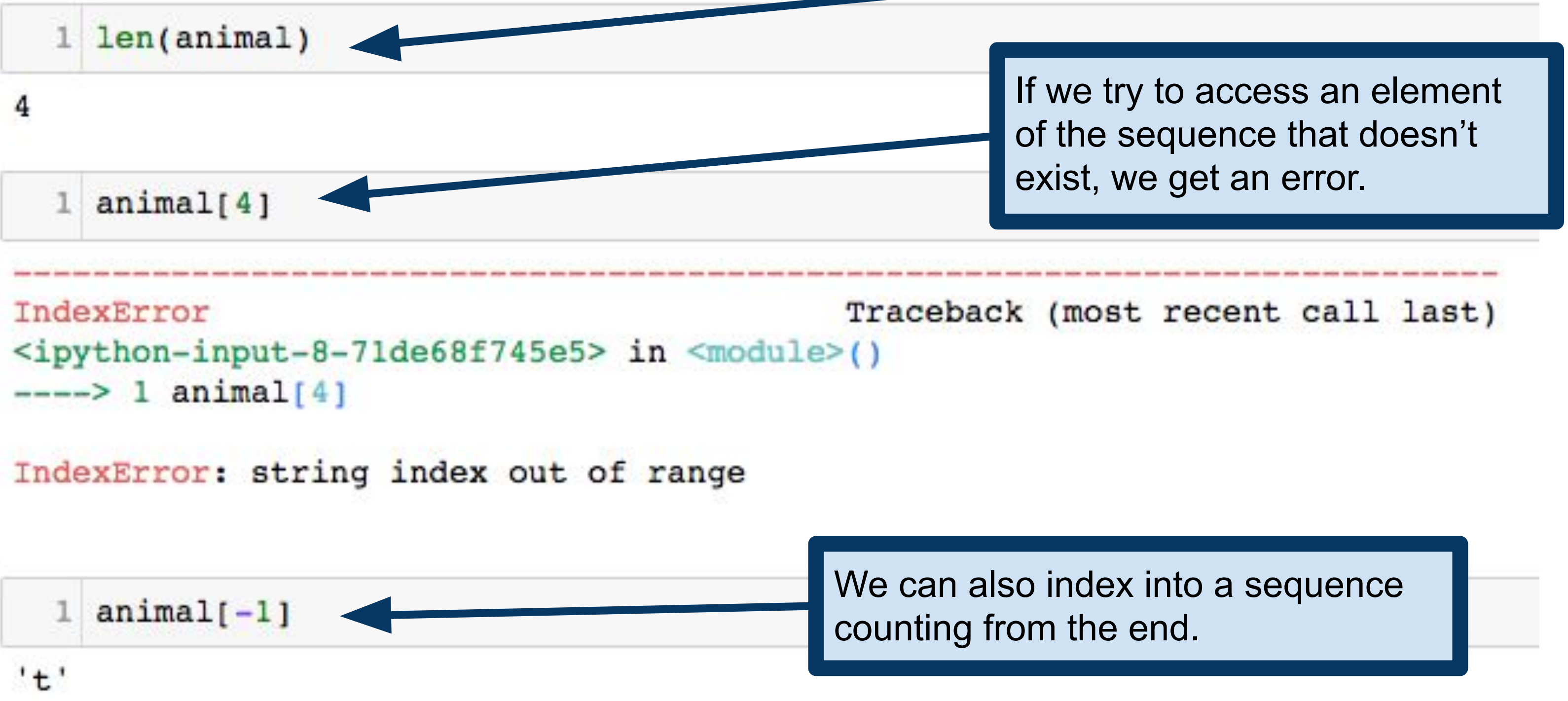
```
'a'
```

```
animal[3]
```

```
't'
```

Strings in Python

Strings are **sequences** of characters



```
len(animal)
```

```
4
```

```
animal[4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-8-71de68f745e5> in <module>()
----> 1 animal[4]

IndexError: string index out of range
```

```
animal[-1]
```

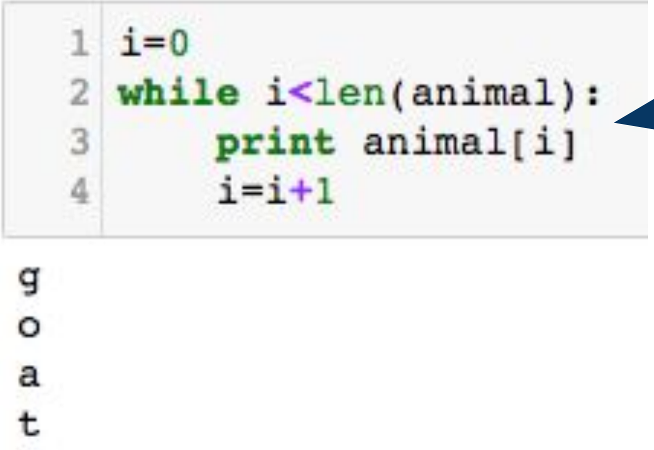
```
't'
```

All Python sequences include a **length** attribute, which is the number of elements in the sequence.

If we try to access an element of the sequence that doesn’t exist, we get an error.

We can also index into a sequence counting from the end.

Strings in Python



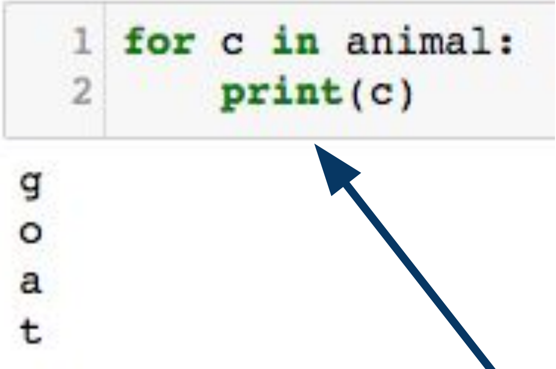
```
i=0
while i<len(animal):
    print animal[i]
    i=i+1
```

```
g
o
a
t
```

We can index into a sequence using an index variable.

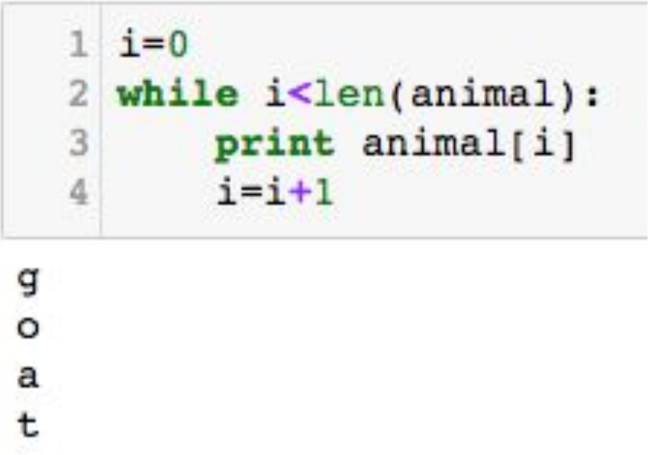
...but there’s a better way to perform this operation...

Iterations and traversals: for-loops



```
for c in animal:
    print(c)
```

```
g
o
a
t
```



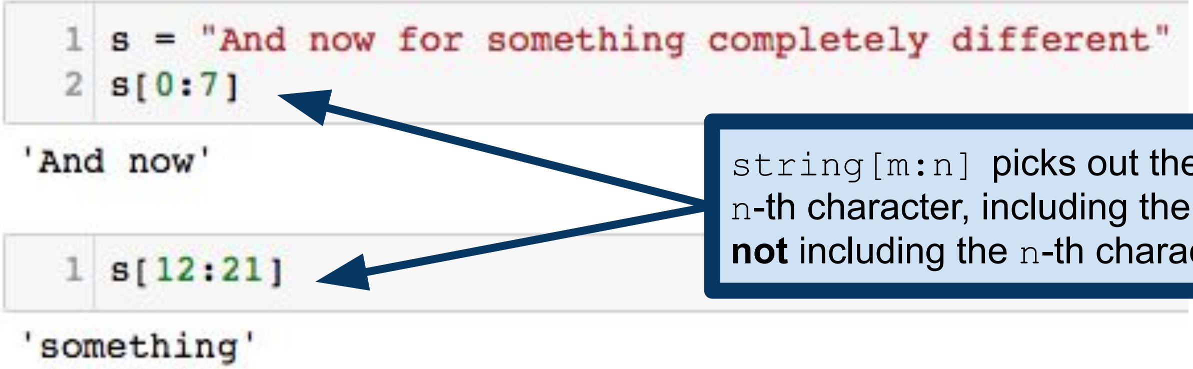
```
i=0
while i<len(animal):
    print animal[i]
    i=i+1
```

```
g
o
a
t
```

For-loop provides a more concise way to express the pattern on the right.

Selecting subsequences: slices

A segment of a Python sequence is called a **slice**



```
s = "And now for something completely different"
s[0:7]
```

```
'And now'
```

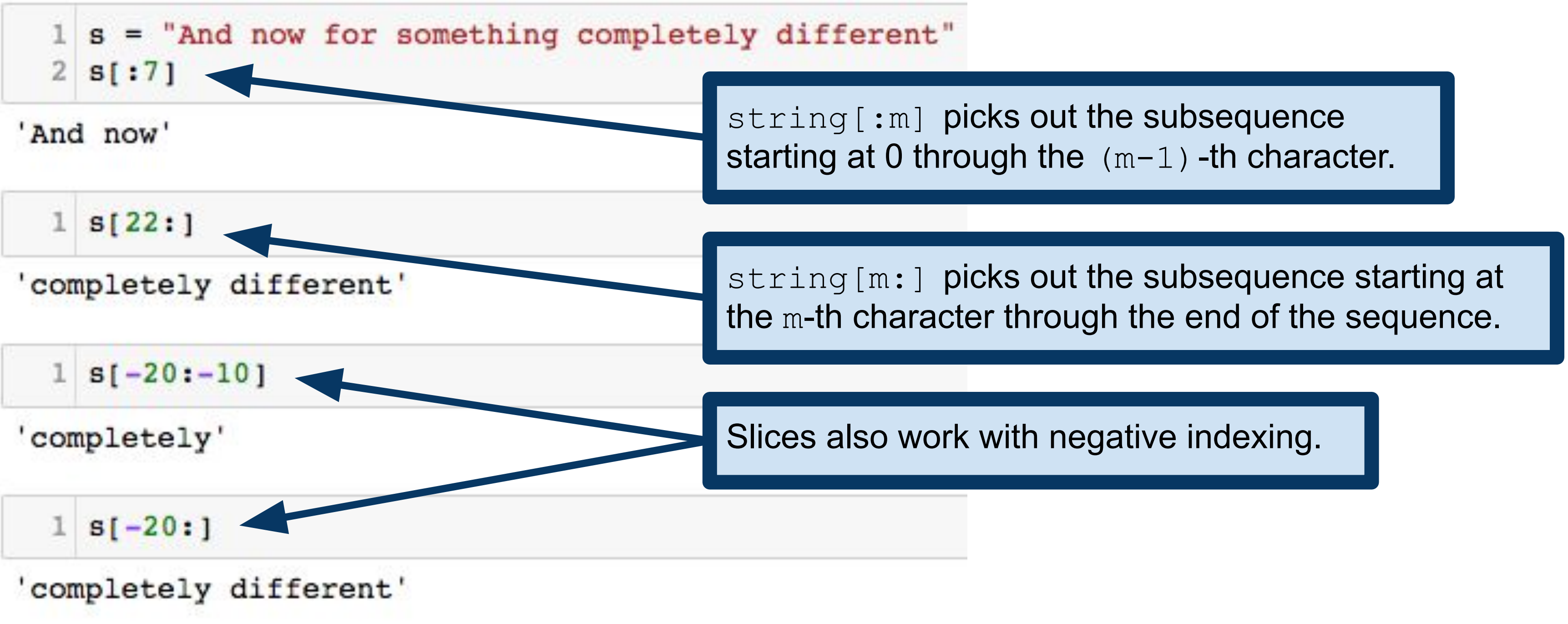
```
s[12:21]
```

```
'something'
```

`string[m:n]` picks out the `m`-th character to the `n`-th character, including the `m`-th character, but **not** including the `n`-th character.

Selecting subsequences: slices

A segment of a Python sequence is called a **slice**



```
s = "And now for something completely different"
s[:7]
```

```
'And now'
```

```
s[22:]
```

```
'completely different'
```

```
s[-20:-10]
```

```
'completely'
```

```
s[-20:]
```

```
'completely different'
```

`string[:m]` picks out the subsequence starting at 0 through the `(m−1)` -th character.

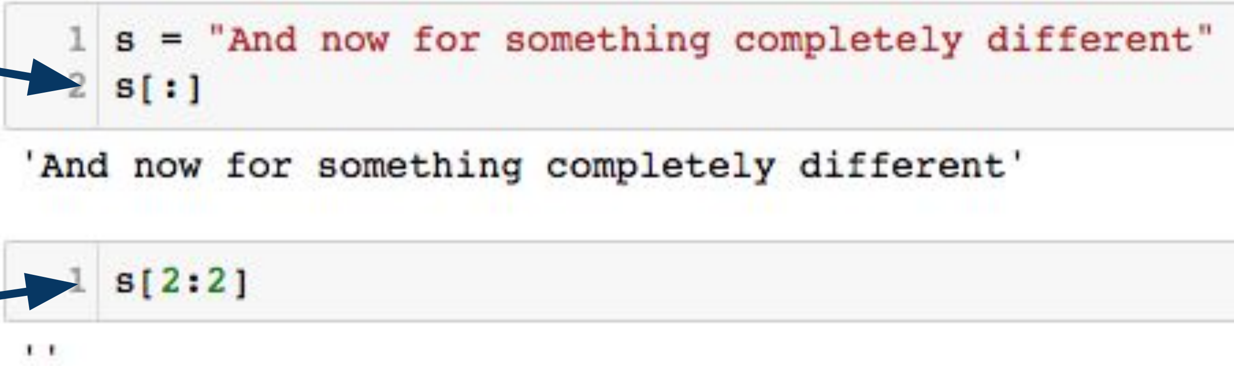
`string[m:]` picks out the subsequence starting at the `m`-th character through the end of the sequence.

Slices also work with negative indexing.

Selecting subsequences: slices

`string[:]` picks out the entire string.

`string[x:x]` picks out the `x`-th through `x`-th letters, not including the `x`-th, so this gets the **empty string.**



```
s = "And now for something completely different"
s[:]
```

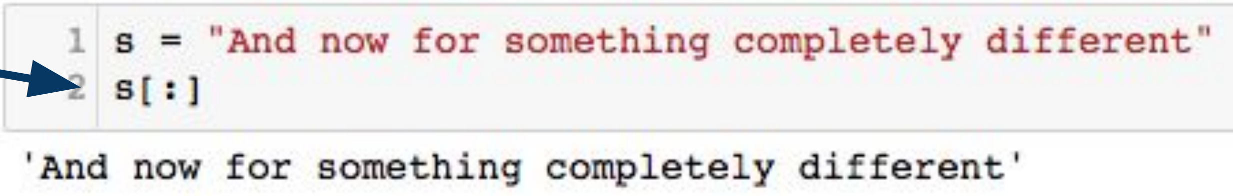
```
'And now for something completely different'
```

```
s[2:2]
```

```
''
```

Selecting subsequences: slices

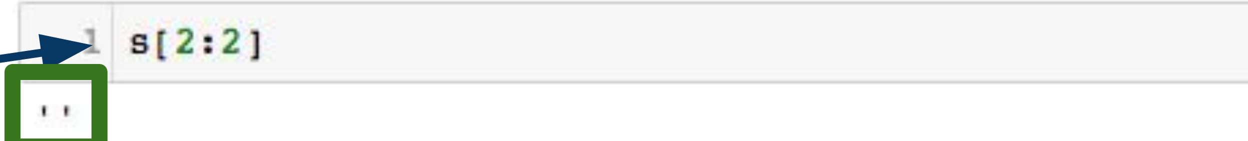
`string[:]` picks out the entire string.



```
s = "And now for something completely different"
s[:]
```

```
'And now for something completely different'
```

`string[x:x]` picks out the x-th through x-th letters, not including the x-th, so this gets the **empty string.**



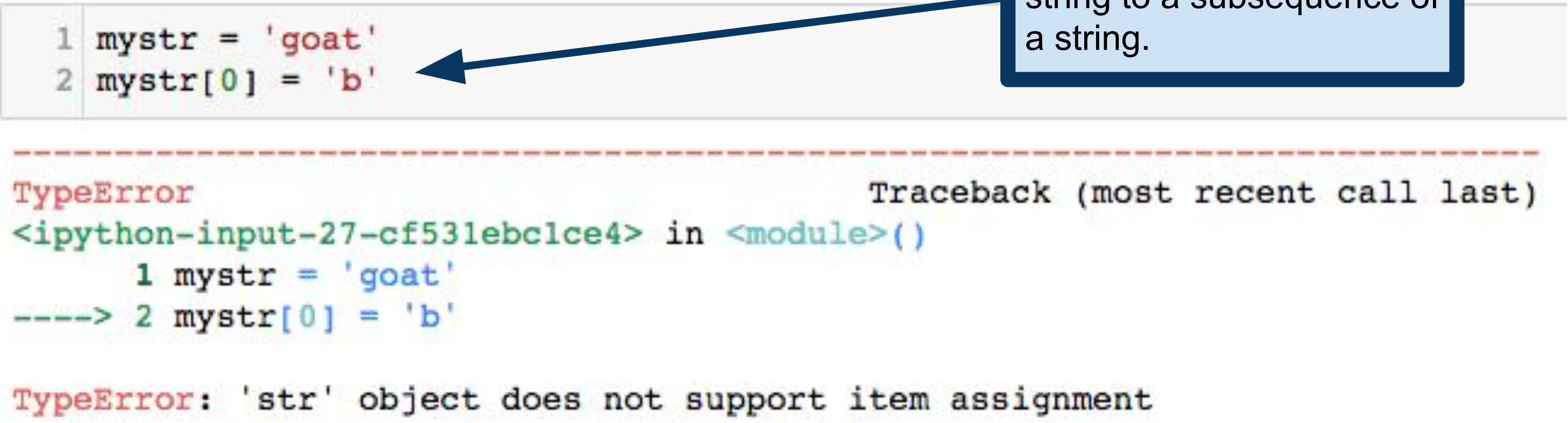
```
s[2:2]
```

```
''
```

The empty string is a string just like any other, but it contains **no letters** and has length 0.

Important concept: immutability

What if I want to change a letter in my string?



```
mystr = 'goat'
mystr[0] = 'b'
```

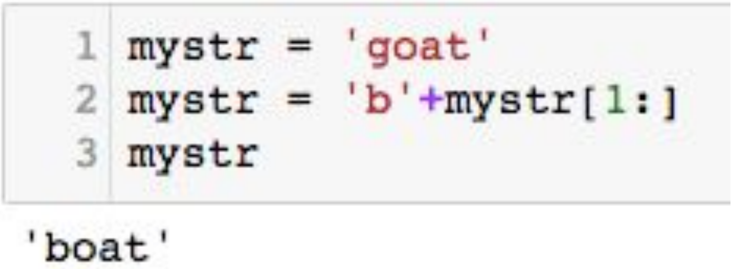
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-cf531ebc1ce4> in <module>()
      1 mystr = 'goat'
----> 2 mystr[0] = 'b'

TypeError: 'str' object does not support item assignment
```

Try and assign a different string to a subsequence of a string.

We get an error because strings are **immutable**. We can’t change the value of an *existing* string.

Important concept: immutability

What if I want to change a letter in my string?

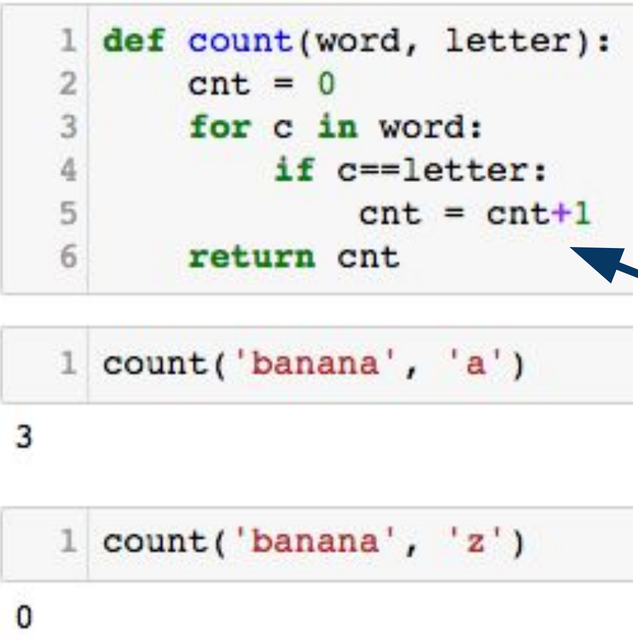


```python
mystr = 'goat'
mystr = 'b'+mystr[1:]
mystr
```

```
'boat'
```

This avoids the error we saw before because it changes the value of the variable `mystr`, rather than trying to change the contents of a string.

Example: string traversal



```python
def count(word, letter):
    cnt = 0
    for c in word:
        if c==letter:
            cnt = cnt+1
    return cnt
```

```python
count('banana', 'a')
```

```
3
```

```python
count('banana', 'z')
```

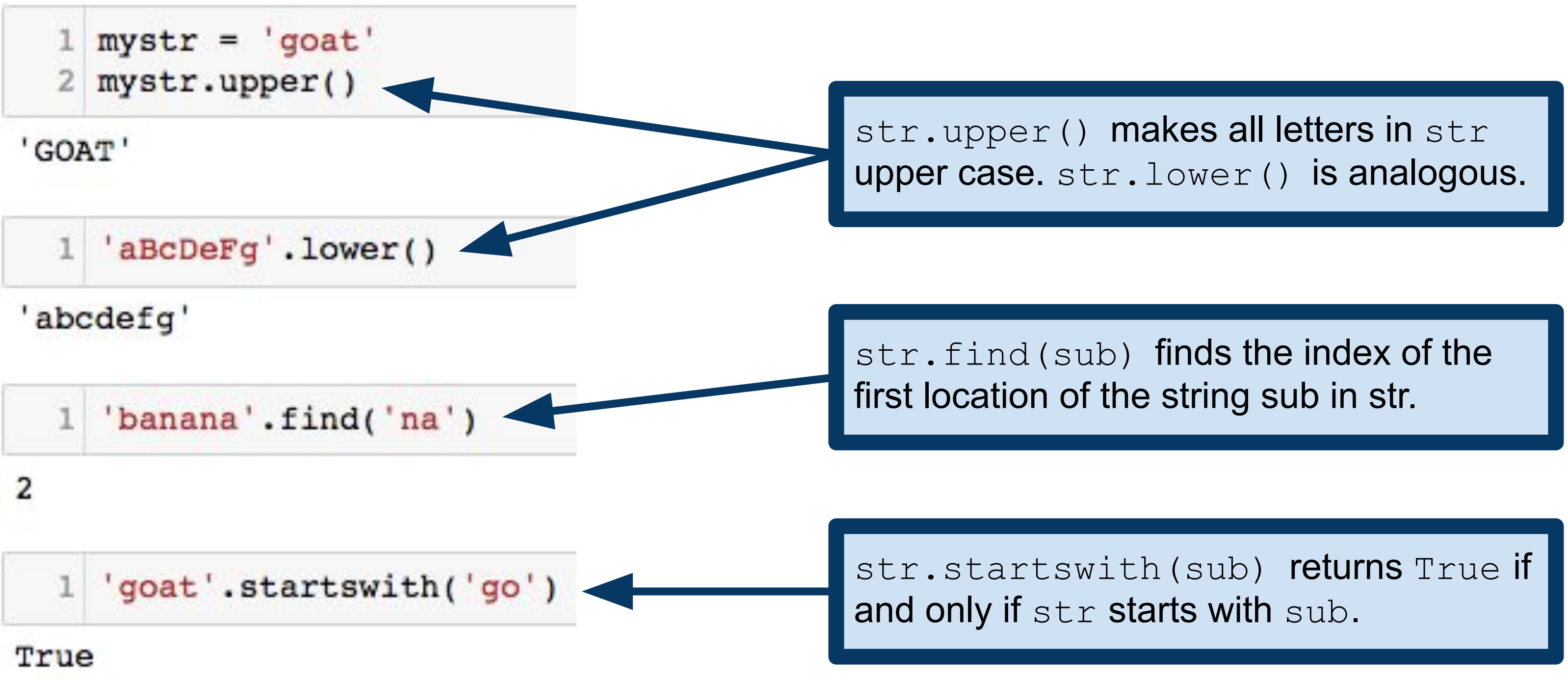
```
0
```

The function `count` makes use of a common pattern, often called a **traversal**. We examine each element of a sequence (i.e., a string), taking some action for each element.

The variable `cnt` keeps a tally of how many times we have seen letter in the string word, so far. We call such a variable a **counter** or an **accumulator**.

Python string methods

Python strings provide a number of built-in operations, called **methods**



```
mystr = 'goat'
mystr.upper()
```

```
'GOAT'
```

```
'aBcDeFg'.lower()
```

```
'abcdefg'
```

```
'banana'.find('na')
```

```
2
```

```
'goat'.startswith('go')
```

```
True
```

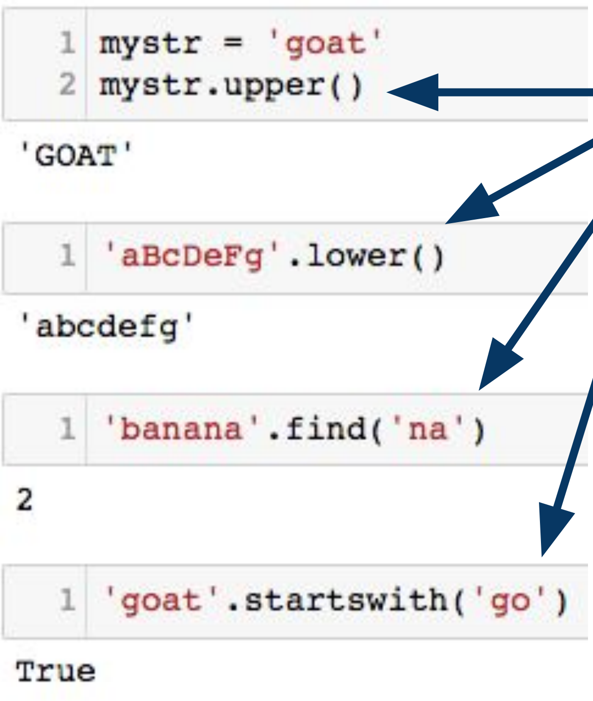
`str.upper()` makes all letters in `str` upper case. `str.lower()` is analogous.

`str.find(sub)` finds the index of the first location of the string sub in str.

`str.startswith(sub)` returns `True` if and only if `str` starts with `sub`.

Python string methods

Python strings provide a number of built-in operations, called **methods**



```
mystr = 'goat'
mystr.upper()
```

```
'GOAT'
```

```
'aBcDeFg'.lower()
```

```
'abcdefg'
```

```
'banana'.find('na')
```

```
2
```

```
'goat'.startswith('go')
```

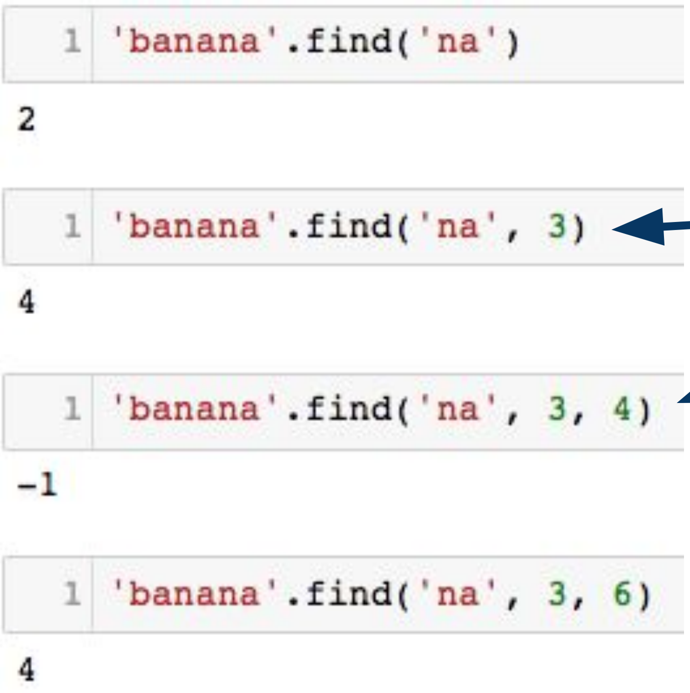
```
True
```

This `variable.method()` notation is called **dot notation**, and it is ubiquitous in Python (and many other languages).

A **method** is like a function, but it is provided by an **object**. We'll learn much more about this later in the semester, but for now, it suffices to know that some data types provide what *look* like functions (they take arguments and return values), and we call these function-like things **methods**.

Many more Python string methods:
https://docs.python.org/3/library/stdtypes.html#string-methods

Optional arguments: `str.find()`



```
'banana'.find('na')
```

```
2
```

```
'banana'.find('na', 3)
```

```
4
```

```
'banana'.find('na', 3, 4)
```

```
-1
```

```
'banana'.find('na', 3, 6)
```

```
4
```

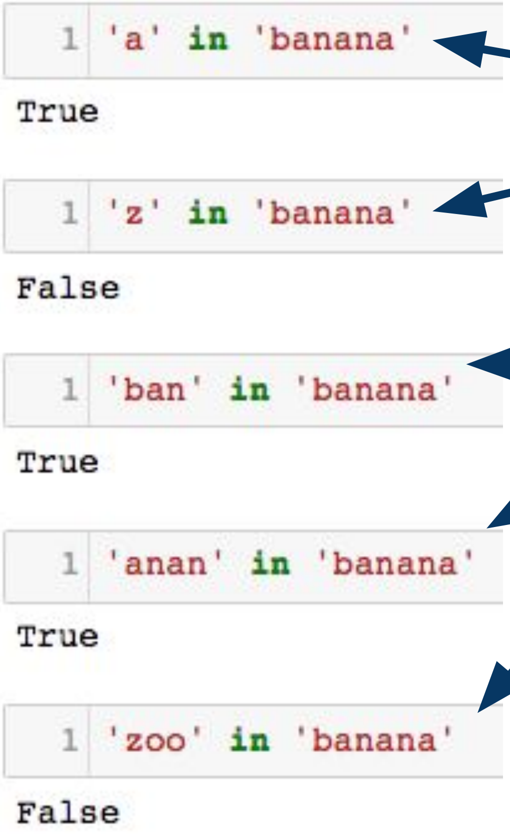
The `str.find()` method takes **optional arguments**, which specify where in the string to start looking for a match, and the last index to consider for a match.

Find first occurrence of `‘na’`, starting from index `3`.

Find first occurrence of `‘na’`, starting from index `3`, and nowhere past `4`.

The documentation writes this method as `str.find(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`. Square brackets indicate optional arguments. In this case, brackets also indicate that with two arguments, the second one will be interpreted as the `start` argument. https://docs.python.org/3/library/stdtypes.html#string-methods

Searching sequences: the `in` keyword



```
'a' in 'banana'
```

```
True
```

```
'z' in 'banana'
```

```
False
```

```
'ban' in 'banana'
```

```
True
```

```
'anan' in 'banana'
```

```
True
```

```
'zoo' in 'banana'
```

```
False
```

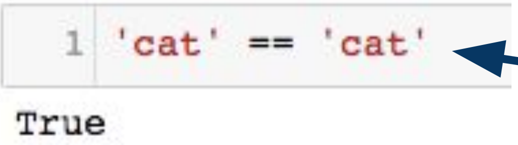
`x in y` returns `True` if `x` occurs as a substring of `y`, and `False` otherwise.

Importantly, we can check for a whole substring, making this very similar to `str.find()`.

The `in` keyword applies more generally to check whether an object is contained in a sequence. We'll see more examples of this in the future, but for now, we only need to worry about strings.

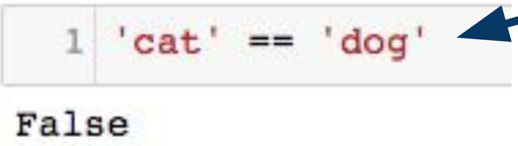
String Comparison

Sometimes we want to check if two strings are equal



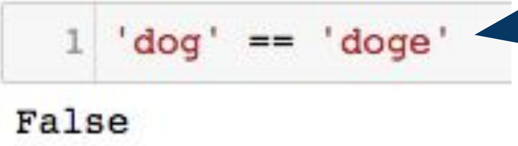
```
'cat' == 'cat'
```

```
True
```



```
'cat' == 'dog'
```

```
False
```



```
'dog' == 'doge'
```

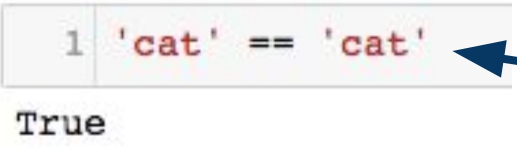
```
False
```

Use the equality operator (==), just like for comparing numbers.

Strings have to match exactly. Substring is not enough!

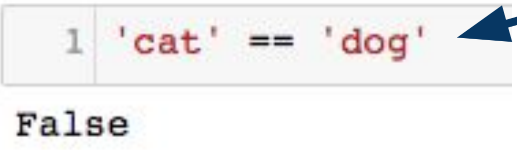
String Comparison

Sometimes we want to check if two strings are equal



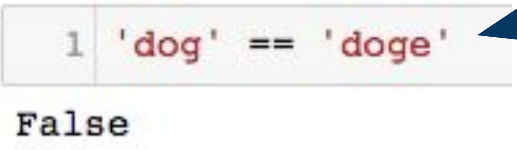
```
'cat' == 'cat'
```

```
True
```



```
'cat' == 'dog'
```

```
False
```



```
'dog' == 'doge'
```

```
False
```

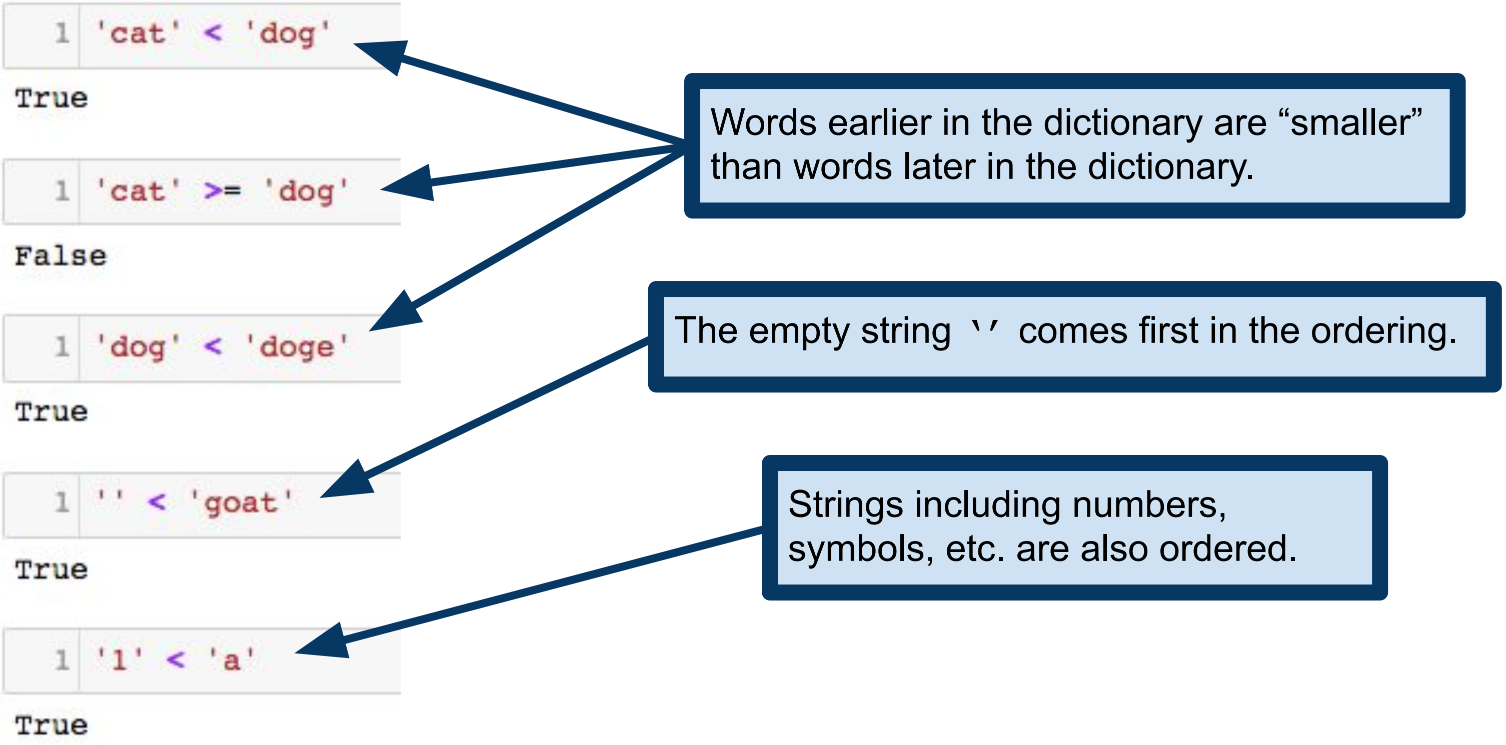
Use the equality operator (==), just like for comparing numbers.

Strings have to match exactly. Substring is not enough!

If we can compare strings with equality, we should be able to compare them with inequalities, too...

String Comparison

We can also compare words under alphabetical ordering



```
'cat' < 'dog'
```

```
True
```

```
'cat' >= 'dog'
```

```
False
```

```
'dog' < 'doge'
```

```
True
```

```
'' < 'goat'
```

```
True
```

```
'1' < 'a'
```

```
True
```

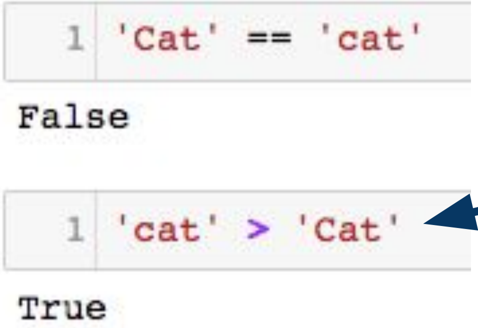
Words earlier in the dictionary are “smaller” than words later in the dictionary.

The empty string ‘’ comes first in the ordering.

Strings including numbers, symbols, etc. are also ordered.

String Comparison

Important: upper case and lower case letters ordered differently!



```
'Cat' == 'cat'
```

```
False
```

```
'cat' > 'Cat'
```

```
True
```

Upper case letters are ordered before lower case letters.

For more information:
https://docs.python.org/3/library/stdtypes.html#comparisons

For **much** more information:
https://docs.python.org/3/library/operator.html?highlight=equallity

Python Lists

Strings in Python are “sequences of characters”

But what if I want a sequence of something else?
A vector would be naturally represented as a sequence of numbers
A class roster might be represented as a sequence of strings

Python lists are sequences whose values can be of any data type
We call these list entries the **elements** of the list