STATS 507
Data Analysis in Python

Lecture 3: Lists

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

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

This is a list of four strings.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

This is a list of nine integers

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

The elements of a list need not be of the same type. Here is a list with a string, an integer and a float.

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

A list can even contain more lists! This is a list of three lists, each of which is a list of three integers.

Constructing Lists

It is possible to construct a list with no elements, the empty list.

Two equivalent ways of creating an empty list.

```
x = []
x
```

```
[]
```

Straight-forwardly create a list using square brackets notation, but supply no elements. So `x` is empty.

```
x = list()
x
```

```
[]
```

Use the reserved keyword `list`, which casts to a list. Given no arguments, it creates an empty list.

Accessing List Elements

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fruits[0]
```

'apple'

```
fruits[1]
```

'orange'

```
fruits[2]
```

'banana'

```
fruits[-1]
```

'kiwi'

We can access individual elements of a list just like a string. This is because both strings and lists are examples of Python **sequences.**

Indexing from the end of the list, just like with strings.

Accessing List Elements

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fruits
```

```
['apple', 'orange', 'banana', 'kiwi']
```

Unlike strings, lists are **mutable**. We can change individual elements after creating the list.

```
fruits[-1] = 'mango'
fruits
```

```
['apple', 'orange', 'banana', 'mango']
```

Reminder of what happens if we try to do this with a string. This error is because string are **immutable**. Once they're created, they can't be altered.

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-b526da741b9a> in <module>()
      1 mystring = 'goat'
----> 2 mystring[0]='b'

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

Lists are sequences, so they have a length

```python
fruits = ['apple', 'orange', 'banana', 'kiwi']
len(fruits)
```

4

```python
len([])
```

0

The empty list has length 0, just like the empty string.

```python
pythagoras = [[3, 4, 5], [5, 12, 13], [8, 15, 17]]
len(pythagoras)
```

3

One might be tempted to say that `pythagoras` should have length 9, but each element of a list counts only once, even if it is itself a more complicated object!

Lists are sequences, so they support the `in` operator

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
'apple' in fruits
```

```
True
```

Just like with strings, `x in y` returns `True` if and only if `x` is an element of `y`.

```
'grape' in fruits
```

```
False
```

```
['apple', 'orange'] in fruits
```

```
False
```

Warning: This contrasts with the string case. Recall that `'ap' in 'apple'` evaluates to `True`. By analogy, this line of code should also evaluate to `True`, but it doesn't, because for lists, the `in` operator only checks elements, not subsequences.

```
['cat','dog'] in [['cat','dog'], ['bird','goat']]
```

```
True
```

Common pattern: list traversal

For each element of a list, do something with that element

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
for f in fruits:
    print(f)
```

```
apple
orange
banana
kiwi
```

```
numbers = range(5)
for n in numbers:
    print(2**n)
```

```
1
2
4
8
16
```

`range(x)` produces a list of the integers 0 to `x-1`.
For more information:
https://docs.python.org/3/library/stdtypes.html#ranges

Common pattern: list traversal

For each element of a list, do something with that element

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
for i in range(len(fruits)):
    fruits[i] = fruits[i].upper()

for f in fruits:
    print(f)
```

```
APPLE
ORANGE
BANANA
KIWI
```

> Sometimes, we need to be able to index into the list itself, in which case we use a slightly different traversal pattern, in which we iterate an **index variable**, `i` in this example.

Common pattern: list traversal

For each element of a list, do something with that element

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
for i in range(len(fruits)):
    fruits[i] = fruits[i].upper()

for f in fruits:
    print(f)
```

```
APPLE
ORANGE
BANANA
KIWI
```

Sometimes, we need to be able to index into the list itself, in which case we use a slightly different traversal pattern, in which we iterate an **index variable**, `i` in this example.

Note: this operation is possible because lists are mutable!

List operations: concatenation

List concatenation is similar to strings.

```
fibonacci = [0,1,1,2,3,5,8]
primes = [2,3,5,7,11,13]
fibonacci + primes
```

```
[0, 1, 1, 2, 3, 5, 8, 2, 3, 5, 7, 11, 13]
```

```
3*['cat','dog']
```

```
['cat', 'dog', 'cat', 'dog', 'cat', 'dog']
```

These operations are precisely analogous to the corresponding string operations. This makes sense, since both strings and lists are **sequences**.
https://docs.python.org/3/library/stdtypes.html#typesseq

List operations: slices

Also like strings, it is possible to select **slices** of a list

```
animals = ['cat', 'dog', 'goat', 'bird', 'llama']
animals[1:3]
```

```
['dog', 'goat']
```

```
animals[3:]
```

```
['bird', 'llama']
```

```
animals[:2]
```

```
['cat', 'dog']
```

```
animals[:]
```

```
['cat', 'dog', 'goat', 'bird', 'llama']
```

Again, analogously to the corresponding string operations.
https://docs.python.org/3/library/stdtypes.html#typesseq

List Methods

Lists supply a certain set of methods:

`list.append(x)`: adds `x` to the end of the list

`list.extend(L2)`: adds list `L2` to the end of another list (like concatenation)

`list.sort()`: sort the elements of the list

`list.remove(x)`: removes from the list the first element equal to `x`.

`list.pop()`: removes the last element of the list and returns that element.

`list.append()` and `list.extend()`

```
animals = ['cat', 'dog', 'goat', 'bird']
animals.append('unicorn')
animals
```

```
['cat', 'dog', 'goat', 'bird', 'unicorn']
```

We call list methods with dot notation. These are **methods** supported by certain **objects**.

```
fibonacci = [0,1,1,2,3,5,8]
fibonacci.append([13,21])
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, [13, 21]]
```

Warning: `list.append()` adds its argument as the last element of a list! Use `list.extend()` to concatenate to the end of the list!

```
fibonacci = [0,1,1,2,3,5,8]
fibonacci.extend([13, 21])
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Note: all of these list methods act upon the list that calls the method. These methods don't return the new list, they alter the list on which we call them.

`list.sort()` and `sorted()`

```
animals = ['cat', 'dog', 'goat', 'bird']
animals.sort()
animals
```

```
['bird', 'cat', 'dog', 'goat']
```

`list.sort()` sorts the list **in place**. See documentation for how Python sorts data of different types.

```
mixed = [1, 'two', 3.0, [4,5]]
mixed.sort()
mixed
```

```
[1, 3.0, [4, 5], 'two']
```

```
animals = ['cat', 'dog', 'goat', 'bird']
sorted_animals = sorted(animals)
sorted_animals
```

```
['bird', 'cat', 'dog', 'goat']
```

If I don't want to sort a list in place, the `sorted()` command returns a sorted version of the list, leaving its argument unchanged.

```
animals
```

```
['cat', 'dog', 'goat', 'bird']
```

Removing elements: `list.pop()`

```
animals = ['cat', 'dog', 'goat', 'bird']
animals.pop()
```

'bird'

```
animals
```

['cat', 'dog', 'goat']

`list.pop()` removes the last element from the list and returns that element.

```
fibonacci = [0,1,1,2,3,5,8]
fibonacci.pop(3)
```

2

`list.pop()` takes an **optional argument**, which indexes into the list and removes and returns the indexed element

```
fibonacci
```

[0, 1, 1, 3, 5, 8]

Again, this method alters the list itself, rather than returning an altered list.

Removing elements: `list.remove()`

```
animals = ['cat', 'dog', 'goat', 'bird']
animals.remove('cat')
animals
```

```
['dog', 'goat', 'bird']
```

```
numbers = [0,1,2,3,1,2,3,2,3]
numbers.remove(2)
numbers
```

```
[0, 1, 3, 1, 2, 3, 2, 3]
```

`list.remove(x)` removes the first instance of `x` in the list.

```
numbers.remove(4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-160-6d289ee6c03d> in <module>()
----> 1 numbers.remove(4)

ValueError: list.remove(x): x not in list
```

Raises a `ValueError` if no such element exists.

Map, filter and reduce

Example: suppose I want to square every element of a list.

```
def square_all(t):
    res = []
    for elmt in t:
        res.append(elmt**2)
    return res

square_all(range(10))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
fibonacci = [0,1,1,2,3,5,8,13,21]
square_all(fibonacci)
```

```
[0, 1, 1, 4, 9, 25, 64, 169, 441]
```

```
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

This function takes a list `t`, and creates a new list `res`, which consists of the squares of the elements of `t`.

This kind of operation, in which we apply a function to each element of a list, is called a **map** operation.

Note: unlike the list methods in the previous slides, this function creates a new list, and doesn't alter the argument.

Map, filter and reduce

Example: I want to remove all even numbers from a list.

```
def remove_even(t):
    res = []
    for elmt in t:
        if elmt % 2 == 0:
            continue
        else: # elmt is odd.
            res.append(elmt)
    return res

remove_even(range(10))
```

```
[1, 3, 5, 7, 9]
```

This function takes a list `t`, and creates a new list `res`, which contains only the odd elements of `t`.

This kind of operation, in which we keep only the elements of a list that satisfy some condition, is called a **filter** operation.

```
fibonacci = [0,1,1,2,3,5,8,13,21]
remove_even(fibonacci)
```

```
[1, 1, 3, 5, 13, 21]
```

```
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Note: again, this function creates a new list, and doesn't alter the argument.

Map, filter and reduce

Example: compute the sum of a list of numbers

```
def my_sum(t):
    res = 0
    for elmt in t:
        res += elmt
    return res

my_sum(range(10))
```

```
45
```

```
fibonacci = [0,1,1,2,3,5,8,13,21]
my_sum(fibonacci)
```

```
54
```

```
my_sum([])
```

```
0
```

This function takes a list `t`, sums the elements of `t`, and returns the sum.

This notation may be familiar to you already. It is called **augmented assignment**. It is short for `res = res + elmt`.

The variable `res` holds a running sum. We call a variable like this an **accumulator**.

This kind of operation, in which we combine the elements of a list to obtain a single element, is called a **reduce** operation.

Map, filter and reduce

We'll see lots more of these operations later in the course

- They're fundamental to functional programming
- MapReduce and related frameworks are built on this paradigm

Note: all examples were on lists of numbers...

- ...but can write similar functions for strings or other more complicated data

Some of these operations can be expressed with Python **list comprehensions**

Map with list comprehensions

```
zero2nine = range(10)
[x**2 for x in zero2nine]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Basic pattern: `[f(x) for x in mylist]` creates a new list, whose elements are the elements of `mylist`, each with function `f` applied.

```
animals = ['cat', 'dog', 'goat', 'bird']
[s.upper() for s in animals]
```

```
['CAT', 'DOG', 'GOAT', 'BIRD']
```

Note: the function `f` must actually return something!

List comprehensions are a special pattern supplied by Python. They're one of the features of Python that makes it appealing. Very expressive way to write operations!

Filter with list comprehensions

```
fibonacci = [0,1,1,2,3,5,8,13,21]
[x for x in fibonacci if x % 2 ==1]
```

```
[1, 1, 3, 5, 13, 21]
```

Basic pattern:
`[x for x in mylist if boolean_expr]`
creates a new list of all and only the elements of `mylist` that satisfy `boolean_expr`.

```
animals = ['cat', 'dog', 'goat', 'bird']
[x.upper() for x in animals if 'o' in x[1]]
```

```
['DOG', 'GOAT']
```

Can combine filter and map to apply a function to only the elements that pass the filter.

```
[x for x in animals if len(x)==5]
```

```
[]
```

Lists and strings

Lists and strings are both sequences, but they aren't quite the same...

```python
goatstr = 'goat'
goatlist = list(goatstr)
goatlist
```

```
['g', 'o', 'a', 't']
```

`str.split()` turns a string into a list of strings, splitting the string on its argument, called the **delimiter**.

```python
wittgenstein = 'Die Welt ist alles was der Fall ist.'
t = wittgenstein.split(' ')
t
```

```
['Die', 'Welt', 'ist', 'alles', 'was', 'der', 'Fall', 'ist.']
```

`str.join()` is like the inverse of `str.split()`. It takes a list of strings and joins them into a single string.

```python
delim = ' '
delim.join(t)
```

```
'Die Welt ist alles was der Fall ist.'
```

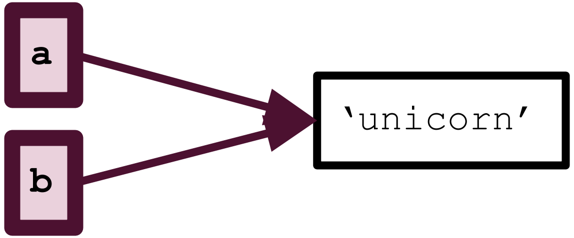
Equivalent vs identical objects

```
a = 'unicorn'
b = 'unicorn'
```

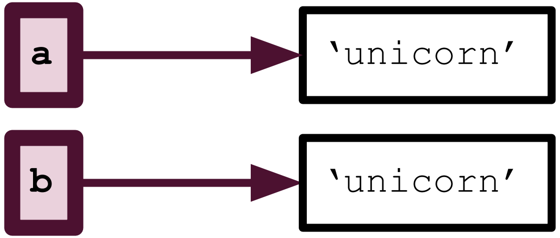
Question: are `a` and `b` the same?

Well, what do we mean by "the same"?

Possibility 1:
`a` and `b` both 'point to' the *same* object.



Possibility 2:
`a` and `b` 'point to' different objects, both objects have *same value*.



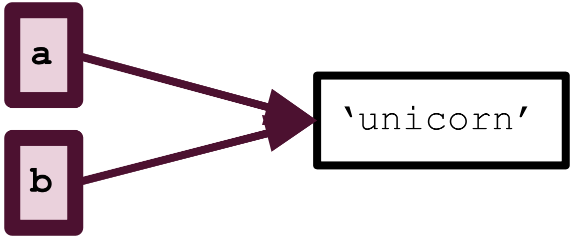
Equivalent vs identical objects

```
a = 'unicorn'
b = 'unicorn'
```

Question: are `a` and `b` the same?

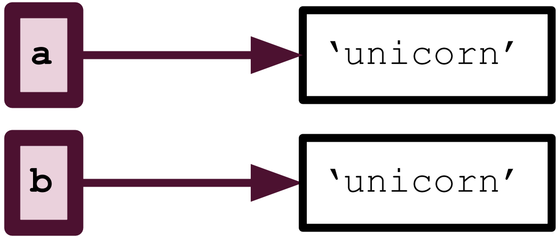
Well, what do we mean by “the same”?

Possibility 1:
`a` and `b` both ‘point to’ the *same* object.



In this case, we say that `a` and `b` are **identical**

Possibility 2:
`a` and `b` ‘point to’ different objects, both objects have *same value*.



In this case, we say that `a` and `b` are **equivalent**

Equivalent vs identical objects

```
a = 'unicorn'
b = 'unicorn'
```

```
a == b
```

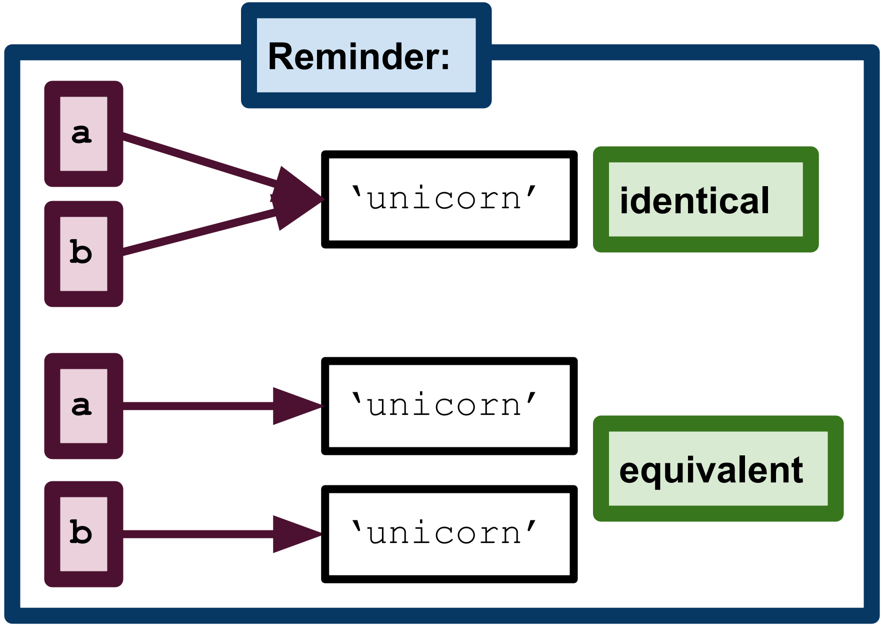
True

```
a is b
```

True

`==` tests if two variables are **equivalent**.
`is` tests if two variables are **identical**.

Strings are immutable, so Python only creates one copy of the string `'unicorn'`, and both `a` and `b` point to it. So they are equivalent **and** identical.



Equivalent vs identical objects

```
a = [1,2,3]
b = [1,2,3]
a == b
```

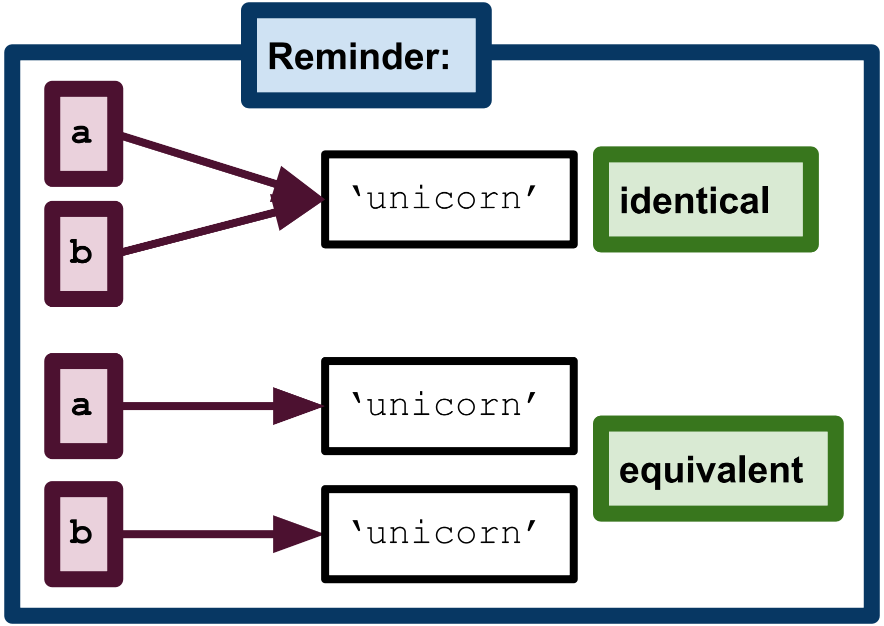
```
True
```

```
a is b
```

```
False
```

`==` tests if two variables are **equivalent**.
`is` tests if two variables are **identical**.

Lists are mutable, so Python creates different copies for `a` and `b`. So they are equivalent but **not** identical.

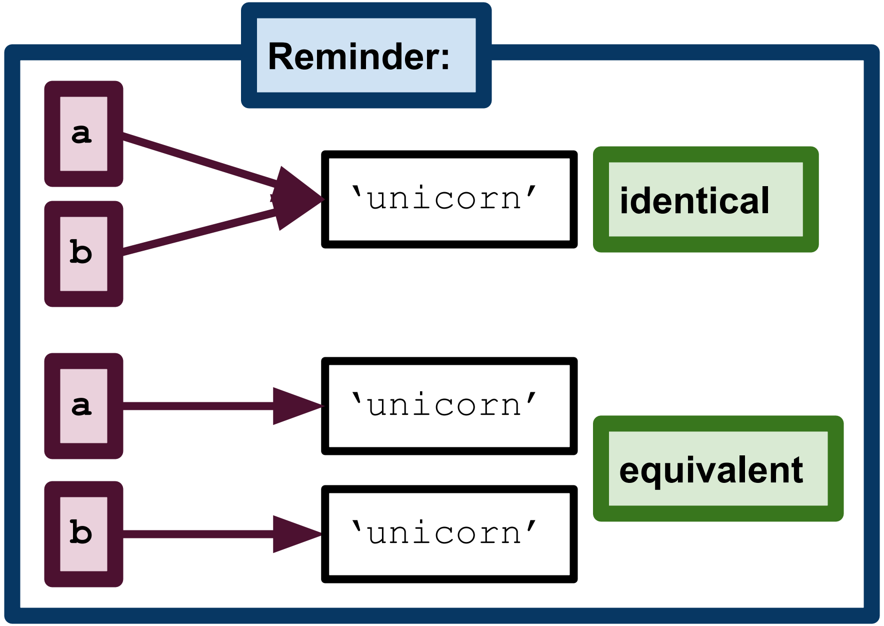


Equivalent vs identical objects: reference

```
a = [1,2,3]
b = a
a is b
```

`==` tests if two variables are **equivalent**.
`is` tests if two variables are **identical**.

Question: will this evaluate to `True` or `False`?



Equivalent vs identical objects: reference

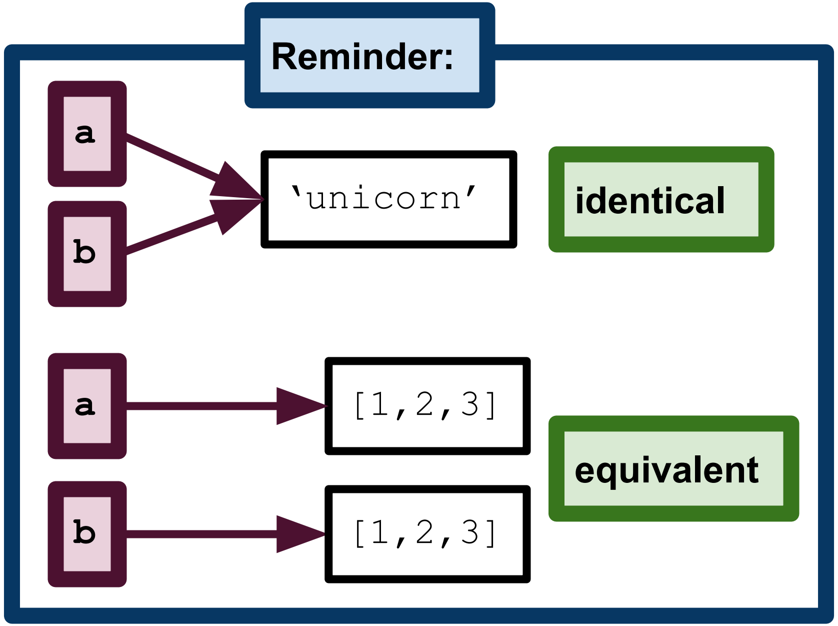
```
a = [1,2,3]
b = a
a is b
```

```
True
```

`==` tests if two variables are **equivalent**.
`is` tests if two variables are **identical**.

Answer: evaluates to `True`, because assignment changes the **reference** of a variable.

Reference of a variable is the value to which it “points”, like on the right.



An object that has more than one reference (i.e., more than one “name”) is called **aliased**. So, on the right, `'unicorn'` is aliased.

Equivalent vs identical objects: reference

```
a = [1,2,3]
b = a
b[-1] = 42
b
```

```
[1, 2, 42]
```

Warning: Aliased mutable objects can sometimes cause unexpected behavior.

```
a[-1]
```

Question: what should this evaluate to?

Equivalent vs identical objects: reference

```
a = [1,2,3]
b = a
b[-1] = 42
b
```

```
[1, 2, 42]
```

```
a[-1]
```

```
42
```

Warning: Aliased mutable objects can sometimes cause unexpected behavior.

Question: what should this evaluate to?

Answer: when we changed the last element of `b`, we changed the object referenced by both `a` and `b`.

Pass-by-reference vs pass-by-value

```
def make_end_42(t):
    # Change the last element of
    # list t to be 42.
    t[-1] = 42

a = [1,2,3]
make_end_42(a)
a
```

```
[1, 2, 42]
```

When you pass an object to a function, the function gets a reference to that object. So changes that we make inside the function are also true outside. This is called **pass-by-reference**, because the function gets a reference to its argument.

Note: strictly speaking, what Python does is not pass-by-reference in the same way as what is normally meant by the term. This is because Python does not use pointers per se in the way that, e.g., C/C++ does.

Pass-by-reference vs pass-by-value

```python
def wrong_make_end_42(t):
    # Change the last element of
    # list t to be 42, incorrectly.
    t = t[:-1] # delete the last element.
    t.append(42)

a = [1,2,3]
wrong_make_end_42(a)
a
```

```
[1, 2, 3]
```

When we make the assignment to `t`, we create a new list, and the reference of `t` is changed, so it no longer points to the list that we passed to the function!

Moral of the story: be careful when working with mutable objects, especially when you are trying to modify objects in place. Often, it's better to just write a function that modifies a list and returns the modified list!