TUPLES, LISTS, MUTABILITY, CLONING

TUPLES

sequence itself has an order so that I
can get to different parts of the sequence
by simply indexing

can include any different kind of element within

- an ordered sequence of elements, can mix element types
- **immutable**, cannot change element values
- represented with parentheses

remember strings?

`te = ()` empty tuple

`t = (2,"one",3)`

`t[0]` → evaluates to 2 -> index into tuples

-> concatenate tuples

`(2,"one",3) + (5,6)` → evaluates to `(2,"one",3,5,6)`

-> slice tuples

`t[1:2]` → slice tuple, evaluates to `("one",)` extra comma is telling me that this is, in fact, a tuple

`t[1:3]` → slice tuple, evaluates to `("one",3)`

`t[1] = 4` → gives error, can't modify object

extra comma means a tuple with one element

TUPLES

- conveniently used to **swap** variable values

```
x = y
y = x
```

✗

```
temp = x
x = y
y = temp
```

✓

```
(x, y) = (y, x)
```

✓

- used to **return more than one value** from a function

```
def quotient_and_remainder(x, y):
    q = x//y
    r = x%y
    return (q, r)
```

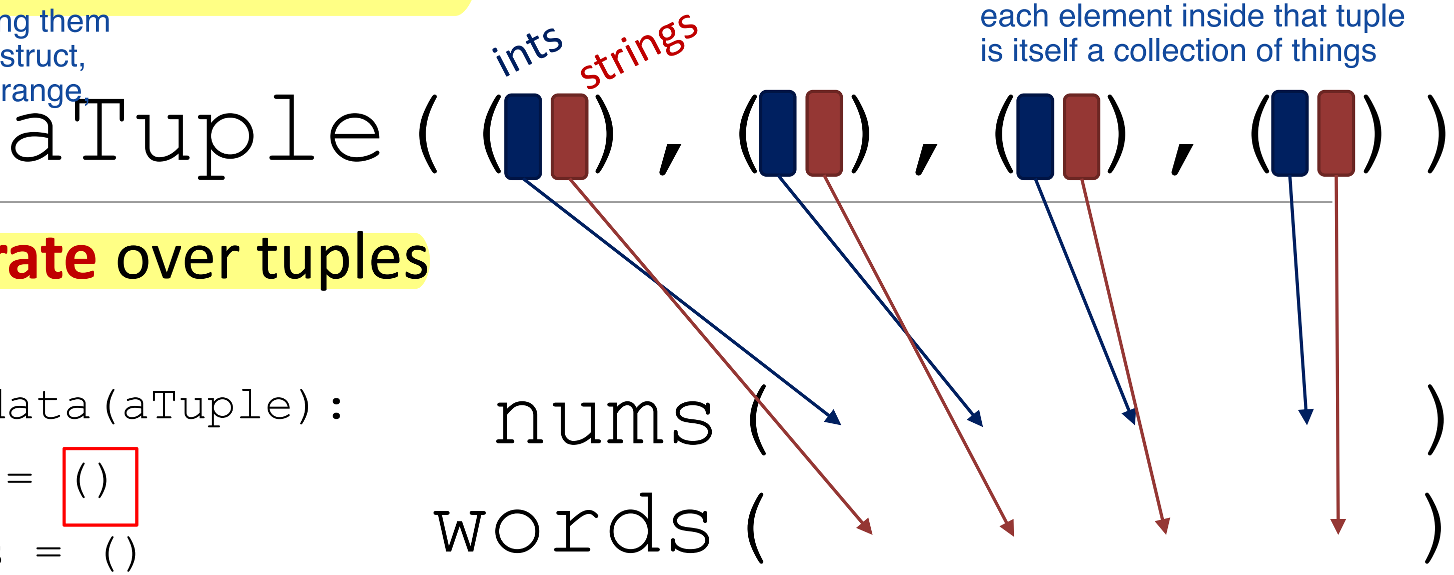
```
(quot, rem) = quotient_and_remainder(4,5)
```

a tuple of names.
That will give Quot the value that q holds.
It will give Rem the value that r holds

MANIPULATING TUPLES

iterate over the tuples, treating them as if they're just a single construct, just the same I would with a range, or I would with a string.

tuple itself consists of tuples each element inside that tuple is itself a collection of things



- can **iterate over tuples**

```
def get_data(aTuple):
    nums = ()
    words = ()
    for t in aTuple:
        nums = nums + (t[0],)
        if t[1] not in words:
            words = words + (t[1],)
    min_nums = min(nums)
    max_nums = max(nums)
    unique_words = len(words)
    return (min_nums, max_nums, unique_words)
```

empty tuple

singleton tuple

iterate over tuple

paren treats it as if it was a single expression
+ to concatenate

return the smallest number, the largest number, and the number of unique words

```
>>> tup1 = (5)
>>> print(tup1)
5
>>> type(tup1)
<type 'int'>
```

When a tuple has only one element, you must specify it as follows: (elt,)

```
>>> tup2 = (5,)
>>> print(tup2)
(5,)
>>> type(tup2)
<type 'tuple'>
```

LISTS

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, [ ]

lists <-> tuples with ()

- a list contains **elements**
  - usually homogeneous (i.e., all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

lists <-> different from a tuple or a string.immutable (not change portions inside of them)

INDICES AND ORDERING

- an element of a list is at a position (aka **index**) in list, indices start at 0

variable name

```
a_list = []
```

empty list

```
b_list = [2, 'a', 4, True]
L = [2, 1, 3]
     ↑  ↑  ↑
```

index: 0 1 2

```
len(L)
```
→ evaluates to 3

```
L[0]
```
→ evaluates to 2

```
L[2]+1
```
→ evaluates to 4

```
L[3]
```
→ gives an error

- index can be a **variable or expression**, must evaluate to an `int`

```
i = 2
L[i-1]
```
→ evaluates to 1 since `L[1] = 1` from above

CHANGING ELEMENTS

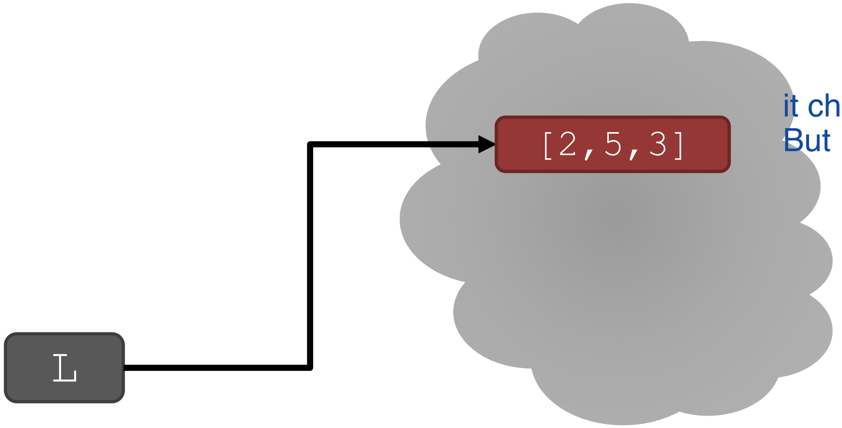
- lists are **mutable**!
- assigning to an element at an index changes the value

```
L = [2, 1, 3]
L[1] = 5
```

change L[1] itself.
It does not produce a new list.
It changes the same object

different from strings and tuples! (immutable)

- `L` is now `[2, 5, 3]`, note this is the **same object** `L`



it changes that middle element.
But L is still pointing to the same expression/list

ITERATING OVER A LIST

- compute the **sum of elements** of a list
- common pattern

```
total = 0
 for i in range(len(L)):
     total += L[i]
 print(total)
```

```
total = 0
 for i in L:
     total += i
 print(total)
```

like strings, can iterate over list elements directly

- notice
  - list elements are **indexed** `0` to `len(L)-1`
  - `range(n)` goes from `0` to `n-1`

function without parenthesis returns function
function () calls the function
if function has no return statement, function returns None !!!

OPERATIONS ON LISTS - ADD

- **add** elements to end of list with `L.append(element)`
- **mutates** the list!

actually changed L itself
So anything that was depending on L being three elements long
is now in trouble because it's now four elements long

```
L = [2,1,3]
L.append(5)
```

→ L is now `[2,1,3,5]`

↑ what is this dot?

this dot is saying, look at L, what is it, it's a list.
So get the append method, or function, associated with lists and apply it

- what is the dot?
  - lists are Python objects, everything in Python is an object
  - objects have data
  - objects have methods and functions
  - access this information by `object_name.do_something()`
  - will learn more about these later

get out some method/function, associated with that kind of object, and then call it ()

OPERATIONS ON LISTS - ADD

- to combine lists together use **concatenation**, + operator
- **mutate** list with `L.extend(some_list)`

So concatenation does not mutate, extension does.

`L1 = [2,1,3]`

`L2 = [4,5,6]`

L1 has not changed, nor has L2.
L3 is a new list that has been formed by concatenating together copies of l1 and l2.

`L3 = L1 + L2` → `L3` is `[2,1,3,4,5,6]`

`L1.extend([0,6])` → mutated `L1` to `[2,1,3,0,6]`

L1 changed, it mutated.

OPERATIONS ON LISTS - REMOVE

- delete element at a **specific index** with `del(L[index])`
- remove element at **end of list** with `L.pop()`, returns the removed element
- remove a **specific element** with `L.remove(element)`
    - looks for the element and removes it
    - if element occurs multiple times, removes first occurrence
    - if element not in list, gives an error

all these operations mutate the list

```
L = [2,1,3,6,3,7,0] # do below in order
L.remove(2)     → mutates L = [1,3,6,3,7,0]
L.remove(3)     → mutates L = [1,6,3,7,0]
del(L[1])       → mutates L = [1,3,7,0]
L.pop()         → returns 0 and mutates L = [1,3,7]
```

CONVERT LISTS TO STRINGS AND BACK

- convert **string to list** with `list(s)`, returns a list with every character from `s` an element in `L`

- can use `s.split()`, to **split a string on a character** parameter, splits on spaces if called without a parameter

- use `''.join(L)` to turn a **list of characters into a string**, can give a character in quotes to add char between every element

```
s = "I <3 cs"              → s is a string
list(s)          → returns ['I',' ','<','3',' ','c','s']
s.split('<')     → returns ['I ', '3 cs']  broken it up into
                                           two parts
L = ['a', 'b', 'c']        → L is a list
''.join(L)       → returns "abc"
'_'.join(L)      → returns "a_b_c"
```

OTHER LIST OPERATIONS

- `sort()` and `sorted()`
- `reverse()`
- and many more!
https://docs.python.org/2/tutorial/datastructures.html

`L=[9,6,0,3]`

`sorted(L)` → returns sorted list, does **not mutate** `L`

`L.sort()` → **mutates** `L=[0,3,6,9]`

`L.reverse()` → **mutates** `L=[9,6,3,0]`

BRINGING TOGETHER LOOPS, FUNCTIONS, range, and LISTS

- range is a special procedure
  - **returns something that behaves like a tuple!**
  - doesn't generate the elements at once, rather it generates the first element, and provides an iteration method by which subsequent elements can be generated

method that says, when you want the next one, we'll give it to you -> nice if I don't want to have to generate a huge range before I start doing some computation

range(5) → equivalent to tuple [0,1,2,3,4]
range(2,6) → equivalent to tuple [2,3,4,5]
range(5,2,-1) → equivalent to tuple [5,4,3]

- when use range in a for loop, what the loop variable iterates over behaves like a list!

```
for var in range(5):
   <expressions>
```

behind the scenes, gets converted to something that will behave like:

```
for var in (0,1,2,3,4):
    <expressions>
```

MUTATION, ALIASING, CLONING

IMPORTANT
and
TRICKY!

Python Tutor is your best friend to help sort this out!

http://www.pythontutor.com/

LISTS IN MEMORY

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable name points to object

I could have multiple variables / aliases pointing to the same list

- any variable pointing to that object is affected
- key phrase to keep in mind when working with lists is **side effects**

if I go in and change an element of a list under one name,
it also changes under the version as I reference it from the other name
because it points to the same place

-> you share information
because you have aliases or nicknames
for the same structure.

AN ANALOGY

Justin Drew Bieber
Justin Bieber
JB
Bieber
The Bieb
JBeebs

- attributes of a person
  - singer, rich
- he is known by many names
- all nicknames point to the same person
  - add new attribute to **one nickname** …

    Justin Bieber: singer, rich **, troublemaker**
  - … **all his nicknames** refer to old attributes AND all new ones

    The Bieb is: singer, rich, **troublemaker**
    JBeebs is: singer, rich, **troublemaker**
    etc…

ALIASES

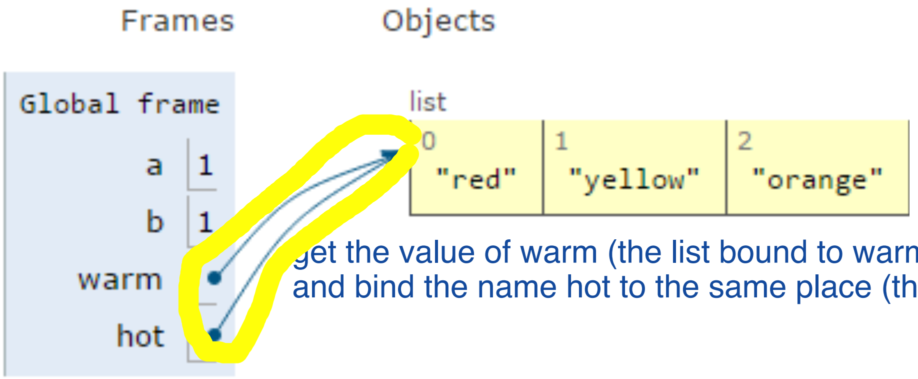
- `hot` is an **alias** for `warm` – changing one changes the other!
- `append()` has a side effect

```
a = 1
b = a
print(a)
print(b)

warm = ['red', 'yellow', 'orange']
hot = warm
```

aliases in global frame are assigned
to the same object (the same list)

"nicknames"

"attributes"



3
pink

get the value of warm (the list bound to warm)
and bind the name hot to the same place (the same list)

```
hot.append('pink')
```

list ["red", "yellow", "orange", "pink"]

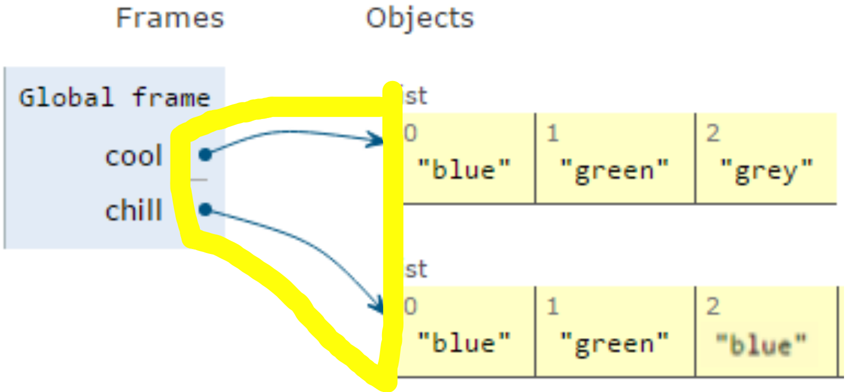
```
print(hot)
print(warm)
```

if I print warm, I'm going to get that list.
If I print hot, I'm going to get that list

PRINT IS NOT ==

- if two lists print the same thing, does not mean they are the same structure
- can test by mutating one, and checking

```
cool = [‘blue’, ‘green’, ‘grey’]
chill = [‘blue’, ‘green’, ‘grey’]
print(cool)
print(chill)

chill[2] = ‘blue’
print(chill)
print(cool)
```



variables in global frame point to different objects, even though they might print out the same

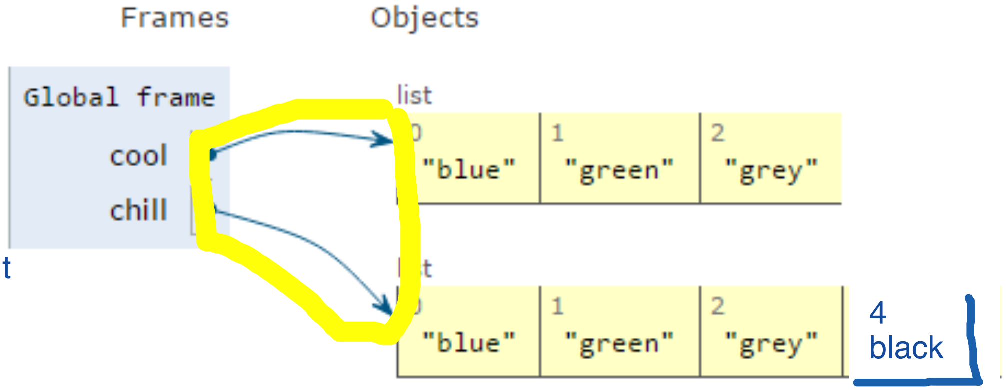
CLONING A LIST

- create a new list and **copy every element** using
  `chill = cool[:]`

-> useful when
I want to do something with a list that involves mutation,
but I don't want to, in fact, change the original list

```
cool = ['blue', 'green', 'grey']
chill = cool[:]

chill.append('black')
print(chill)
print(cool)
```

mutating chill will not affect cool,
bescause they are pointing to different
objects



pointing to different objects, one
object is copy of other object

SORTING LISTS

- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable

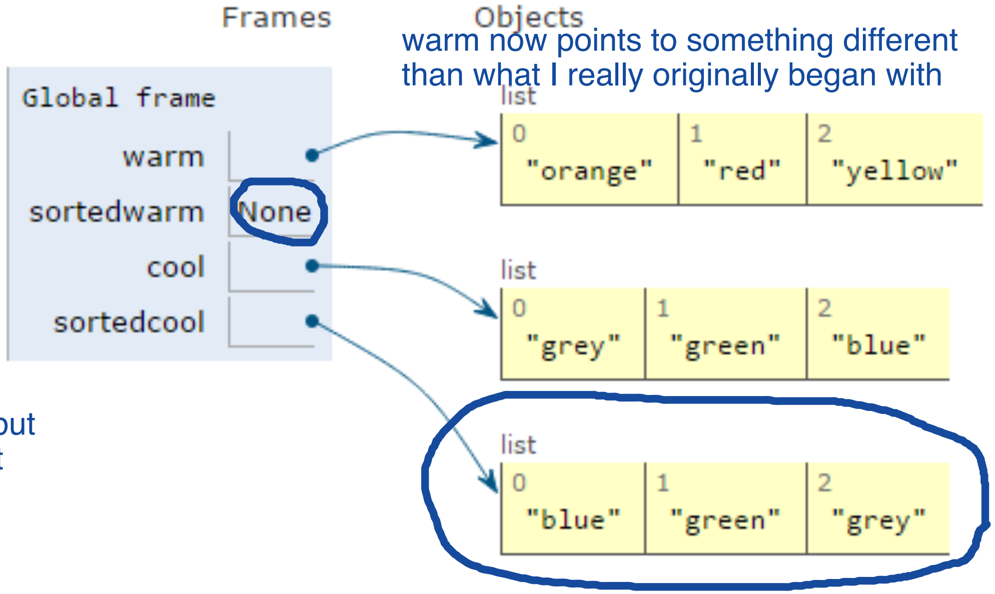
returns list in sorted order

```
warm = [‘red’, ‘yellow’, ‘orange’]
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)
```

functiom returns nothing

```
cool = [‘grey’, ‘green’, ‘blue’]
sortedcool = sorted(cool)
print(cool)
print(sortedcool)
```

function returns sorted list, but does not change original list



warm now points to something different than what I really originally began with

Frames

Global frame

warm	●
sortedwarm	None
cool	●
sortedcool	●

Objects

list

0	1	2
"orange"	"red"	"yellow"

list

0	1	2
"grey"	"green"	"blue"

list

0	1	2
"blue"	"green"	"grey"

LISTS OF LISTS OF LISTS OF….

- can have **nested** lists
- side effects still possible after mutation

```
warm = [‘yellow’, ‘orange’]
hot = [‘red’]
brightcolors = [warm]
```

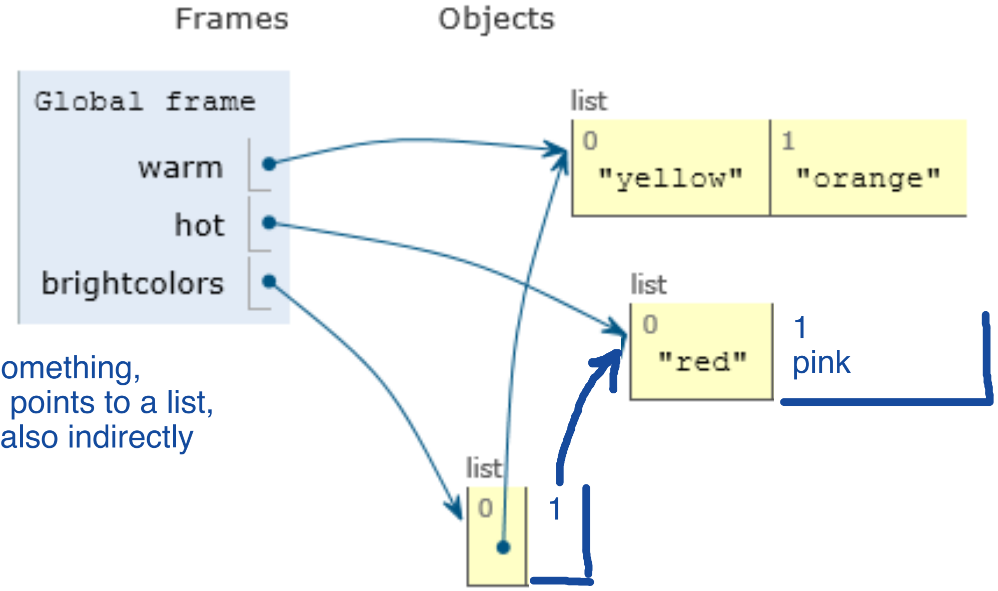
aliasing!

```
brightcolors.append(hot)
print(brightcolors)
```

bright colors points to something, one of whose elements points to a list, when I change hot I've also indirectly changed bright colors

```
hot.append(‘pink’)
print(hot)
print(brightcolors)

print(hot + warm)
print(hot)
```

not mutated, just concatenated



MUTATION AND ITERATION

- **avoid** mutating a list as you are iterating over it

✗

```
def remove_dups(L1, L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
```

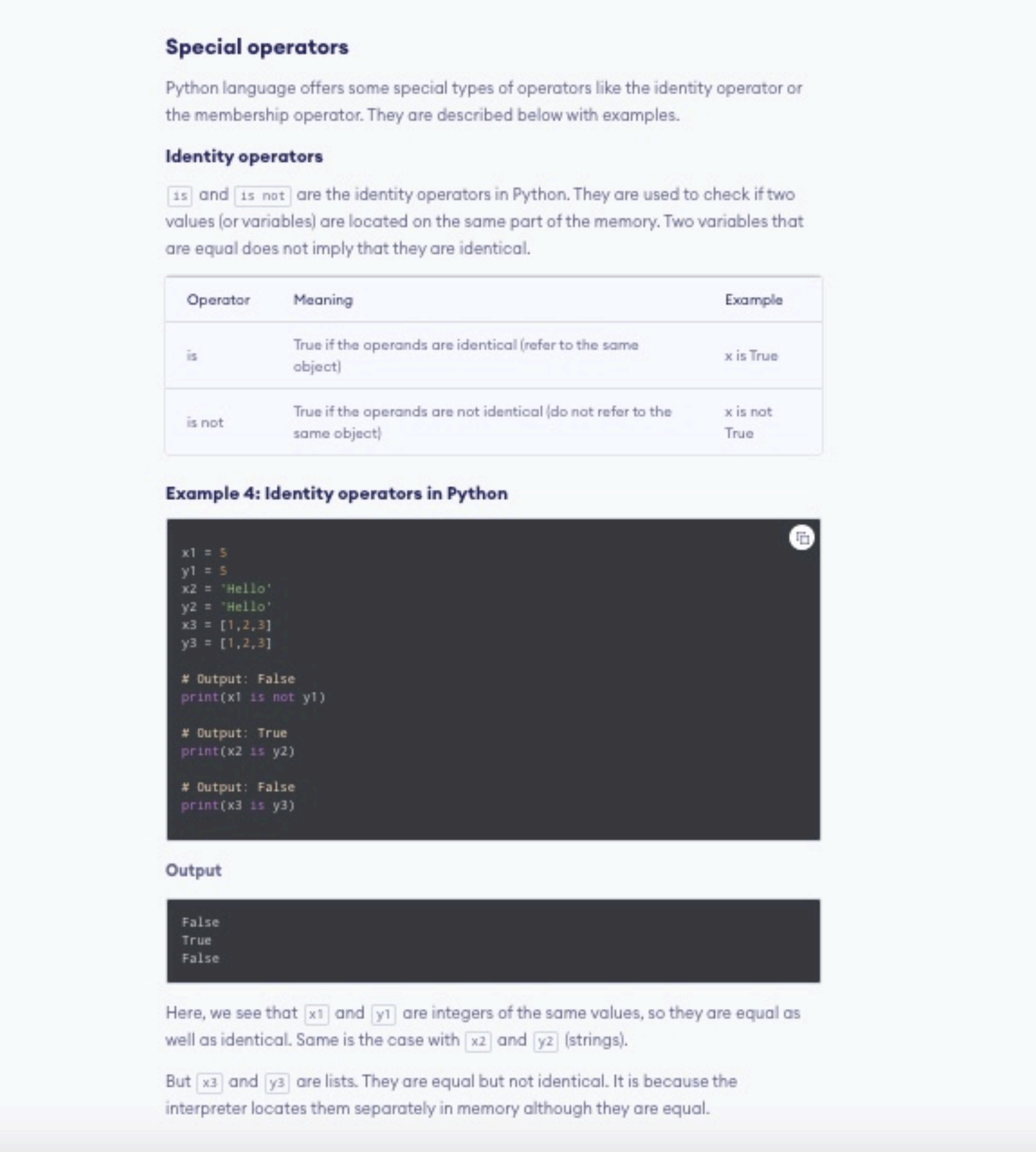
✓

```
def remove_dups_new(L1, L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

```
L1 = [1, 2, 3, 4]
L2 = [1, 2, 5, 6]
remove_dups(L1, L2)
```

clone list first, note that `L1_copy = L1` does NOT clone

- `L1` is `[2,3,4]` not `[3,4]` Why?
  - Python uses an internal counter to keep track of index it is in the loop
  - mutating changes the list length but Python doesn't update the counter
  - loop never sees element 2



Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

Identity operators

`is` and `is not` are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example 4: Identity operators in Python

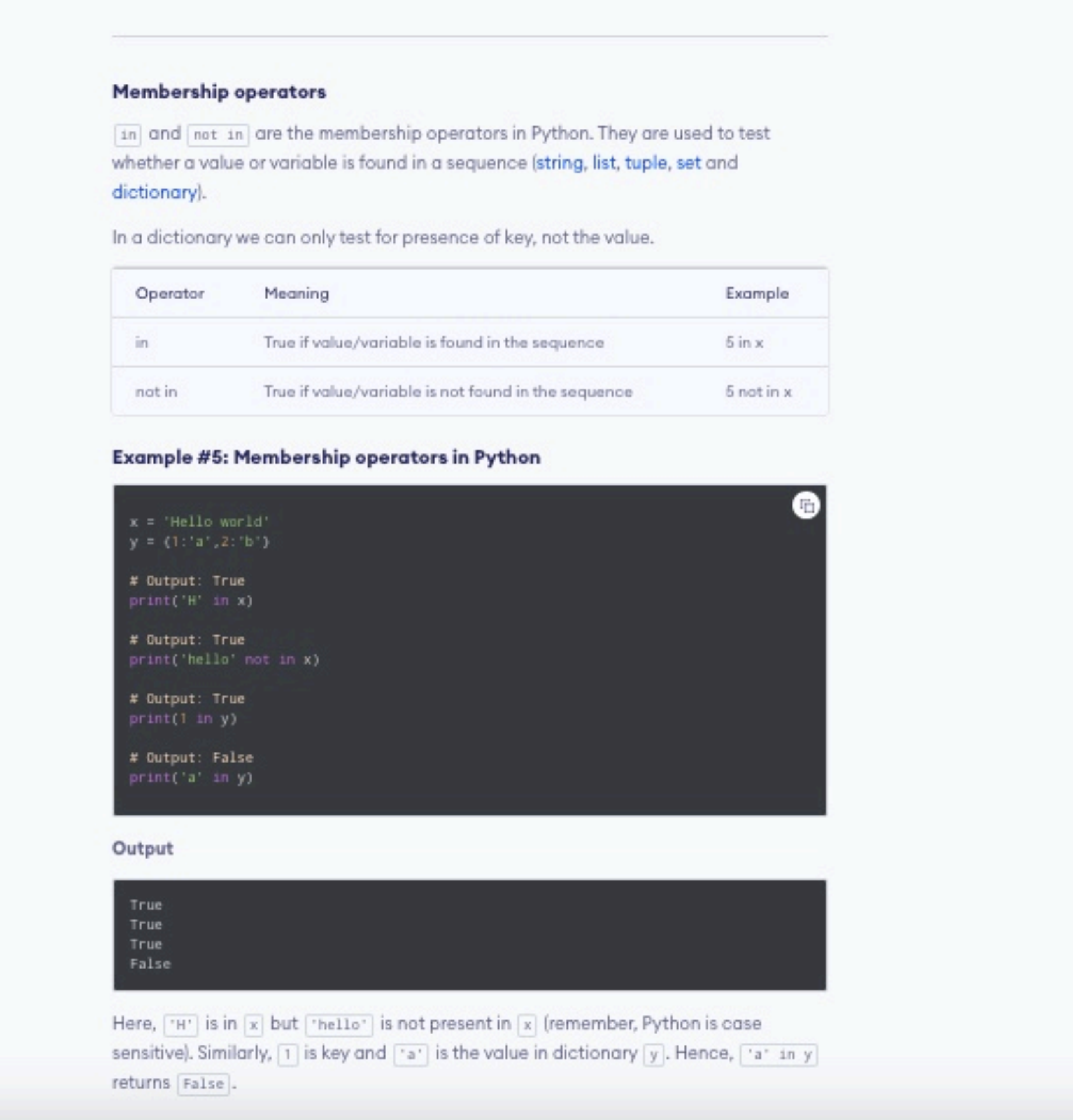
```
x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
y3 = [1,2,3]

# Output: False
print(x1 is not y1)

# Output: True
print(x2 is y2)

# Output: False
print(x3 is y3)
```

Output

```
False
True
False
```

Here, we see that `x1` and `y1` are integers of the same values, so they are equal as well as identical. Same is the case with `x2` and `y2` (strings).

But `x3` and `y3` are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.



Membership operators

`in` and `not in` are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example #5: Membership operators in Python

```
x = 'Hello world'
y = {1:'a',2:'b'}

# Output: True
print('H' in x)

# Output: True
print('hello' not in x)

# Output: True
print(1 in y)

# Output: False
print('a' in y)
```

Output

```
True
True
True
False
```

Here, `'H'` is in `x` but `'hello'` is not present in `x` (remember, Python is case sensitive). Similarly, `1` is key and `'a'` is the value in dictionary `y`. Hence, `'a' in y` returns `False`.