

# TUPLES, LISTS, ALIASING, MUTABILITY, CLONING

(download slides and follow along on [repl.it](https://repl.it)!)

---

COMP100 LECTURE 5

# LAST TIME

---

- functions
- decomposition – create structure
- abstraction – suppress details
- from now on will be using functions a lot

# TODAY

---

- have seen variable types: `int`, `float`, `bool`, `string`
- introduce new **compound data types**
  - tuples
  - lists
- idea of aliasing
- idea of mutability
- idea of cloning

# TUPLES

---

- an ordered sequence of elements, can mix element types

# TUPLES

---

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

*remember  
strings?*

# TUPLES

---

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

te = **()** *empty tuple*

*remember strings?*

# TUPLES

---

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

`te = ()` *empty tuple*

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

`t[0]`

→ evaluates to 2

`(2, "mit", 3) + (5, 6)`

→ evaluates to `(2, "mit", 3, 5, 6)`

*remember strings?*

# TUPLES

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

*remember  
strings?*

`te = ()` *empty  
tuple*

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

`t[0]` → evaluates to 2

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

`t[1:2]` → slice tuple, evaluates to `("mit", )`

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

`len(t)` → evaluates to 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

# TUPLES

---

- conveniently used to **swap** variable values

x = y

y = x



# TUPLES

---

- conveniently used to **swap** variable values

`x = y`

`y = x`



`temp = x`

`x = y`

`y = temp`



# TUPLES

---

- conveniently used to **swap** variable values

`x = y`

`y = x`



`temp = x`

`x = y`

`y = temp`



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



# 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)
```

*integer  
division*







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

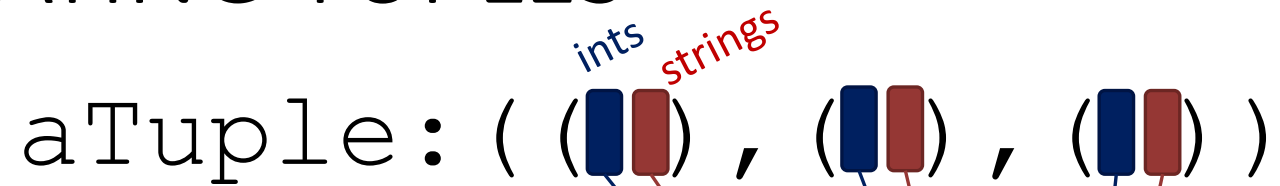
# MANIPULATING TUPLES

aTuple: ( (   ) , (   ) , (   ) )

---

# MANIPULATING TUPLES

aTuple: ( ( <sup>ints</sup>   ) , (   ) , (   ) )









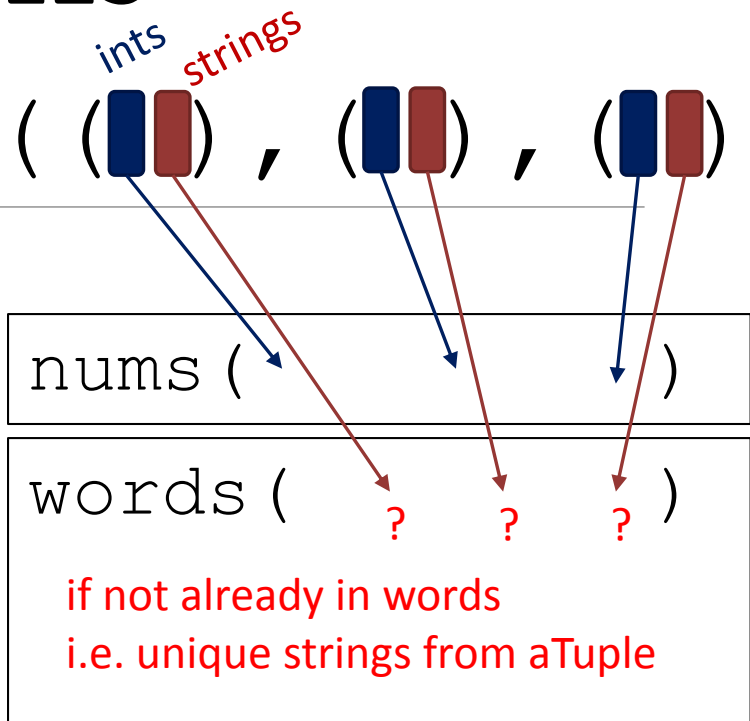
nums (    )

words (    )

if not already in words  
i.e. unique strings from aTuple

# MANIPULATING TUPLES

aTuple: ( (   ), (   ), (   ) )



- 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_n = min(nums)  
    max_n = max(nums)  
    unique_words = len(words)  
    return (min_n, max_n, unique_words)
```

nums (







)

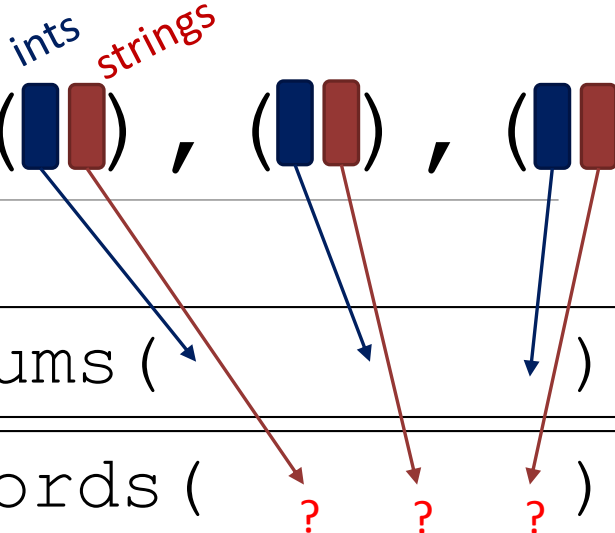
words ( ? ? ? )

if not already in words  
i.e. unique strings from aTuple




# MANIPULATING TUPLES

aTuple: ( (   ) , (   ) , (   ) )



- 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_n = min(nums)  
    max_n = max(nums)  
    unique_words = len(words)  
    return (min_n, max_n, unique_words)
```

*empty tuple*







nums (

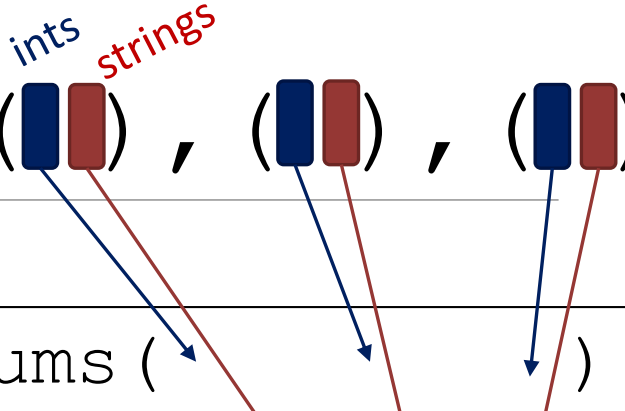
)

words ( ? ? ? )


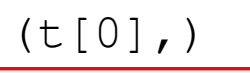
if not already in words  
i.e. unique strings from aTuple

# MANIPULATING TUPLES

aTuple: ( (   ) , (   ) , (   ) )



- can **iterate** over tuples

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

*empty tuple*







*singleton tuple*

nums (       )


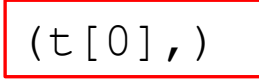
words ( ? ? ? )

if not already in words  
i.e. unique strings from aTuple

# MANIPULATING TUPLES

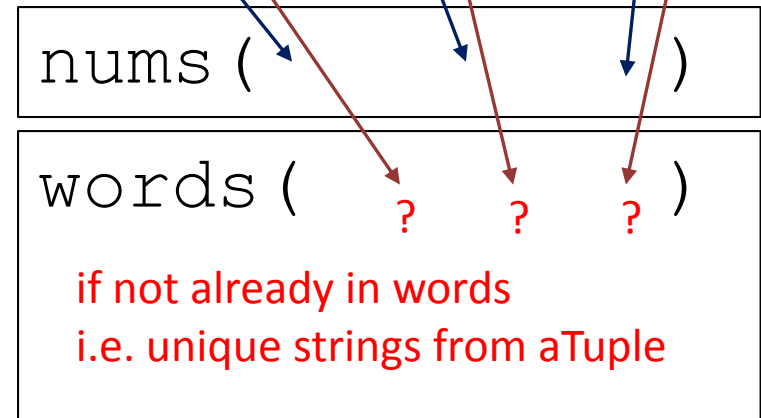
aTuple: ( ( <sup>ints</sup>   ) , (   ) , (   ) )

- can **iterate** over tuples

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

empty tuple

singleton tuple



# LISTS

---

- **ordered sequence** of information, accessible by index
- a list is denoted by **square brackets**, [ ]
- a list contains **elements**
  - usually homogeneous (ie, all integers)
  - can contain mixed types (not common)
- list elements can be changed so a list is **mutable**

# INDICES AND ORDERING

---

`a_list = []` *empty list*

# INDICES AND ORDERING

---

`a_list = []` *empty list*

`L = [2, 'a', 4, [1, 2]]`

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

# INDICES AND ORDERING

---

`a_list = []` *empty list*

`L = [2, 'a', 4, [1, 2]]`

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1, 2]`, another list!

`L[4]` → gives an error

# INDICES AND ORDERING

---

`a_list = []` *empty list*

`L = [2, 'a', 4, [1, 2]]`

`len(L)` → evaluates to 4

`L[0]` → evaluates to 2

`L[2]+1` → evaluates to 5

`L[3]` → evaluates to `[1, 2]`, another list!

`L[4]` → gives an error

`i = 2`

`L[i-1]` → evaluates to 'a' since `L[1] = 'a'` above



# CHANGING ELEMENTS

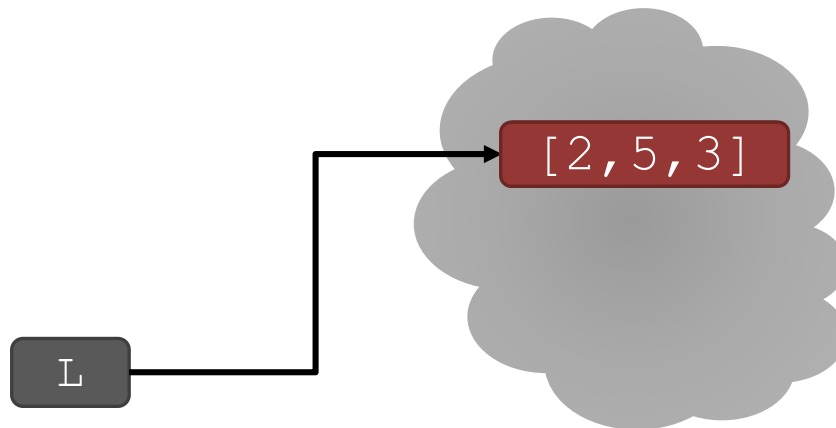
---

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

`L = [2, 1, 3]`

`L[1] = 5`

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



# ITERATING OVER A LIST

---

- compute the **sum of elements** of a list
- common pattern, iterate over list elements

```
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  $\text{len}(L) - 1$
  - `range(n)` goes from 0 to  $n - 1$

# OPERATIONS ON LISTS - ADD

---

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

`L = [2, 1, 3]`

`L.append(5)`       $\rightarrow$  L is now `[2, 1, 3, 5]`

# OPERATIONS ON LISTS - ADD

---

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

- **mutates** the list!

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

```
L.append(5)      → L is now [2, 1, 3, 5]
```



- 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

# OPERATIONS ON LISTS - ADD

---

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

`L1 = [2, 1, 3]`

`L2 = [4, 5, 6]`

`L3 = L1 + L2`

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

`L1.extend([0, 6])`

→ mutated `L1` to `[2, 1, 3, 0, 6]`

# 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

# 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` as 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']`

`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/3/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]`

# MUTATION, ALIASING, CLONING

---



IMPORTANT  
and  
TRICKY!

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

# AN ANALOGY

---



- attributes of a person
  - singer, rich

Justin Bieber

singer

rich

# AN ANALOGY

---



- attributes of a person
  - singer, rich
- he is known by many names
- all nicknames point to the **same person**

Justin Bieber

singer

rich

The Bieb

singer

rich

JBeebs

singer

rich

# AN ANALOGY



- 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

singer

rich

troublemaker

JBeebs

singer

rich

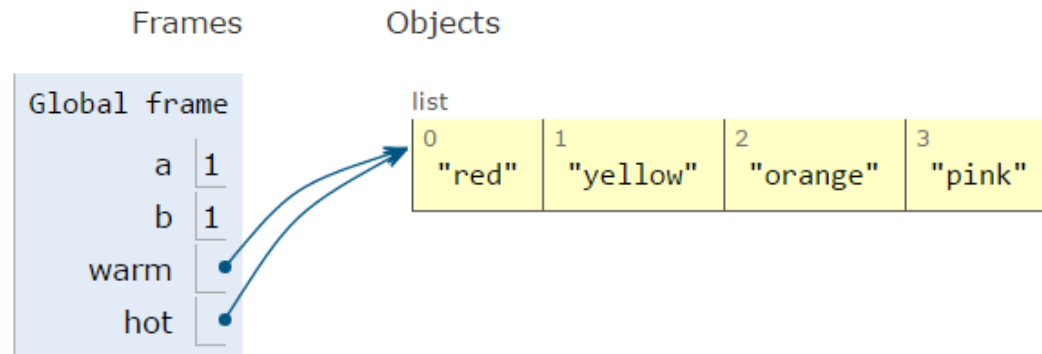
troublemaker

# ALIASES

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

```
1 a = 1
2 b = a
3 print(a)
4 print(b)
5
6 warm = ['red', 'yellow', 'orange']
7 hot = warm
8 hot.append('pink')
9 print(hot)
10 print(warm)
```

```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```

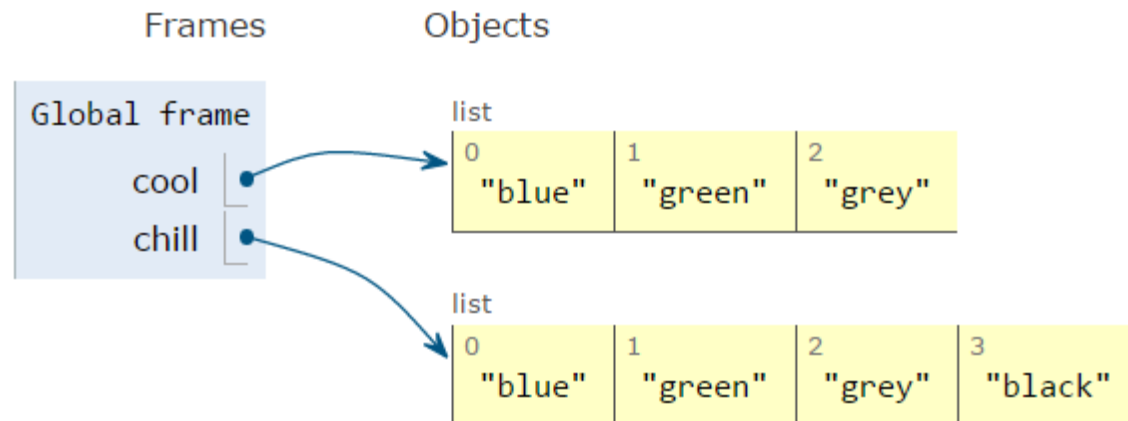


# CLONING A LIST

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

```
1 cool = ['blue', 'green', 'grey']  
2 chill = cool[:]  
3 chill.append('black')  
4 print(chill)  
5 print(cool)
```

```
['blue', 'green', 'grey', 'black']  
['blue', 'green', 'grey']
```



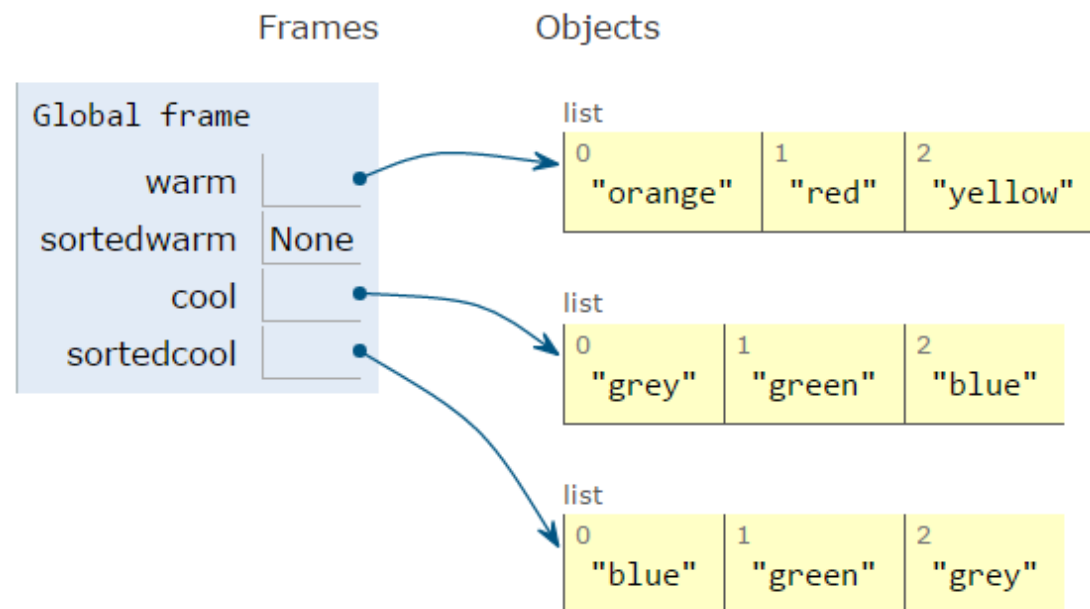


# SORTING LISTS

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

```
['orange', 'red', 'yellow']  
None  
['grey', 'green', 'blue']  
['blue', 'green', 'grey']
```

```
1 warm = ['red', 'yellow', 'orange']  
2 sortedwarm = warm.sort()  
3 print(warm)  
4 print(sortedwarm)  
5  
6 cool = ['grey', 'green', 'blue']  
7 sortedcool = sorted(cool)  
8 print(cool)  
9 print(sortedcool)
```

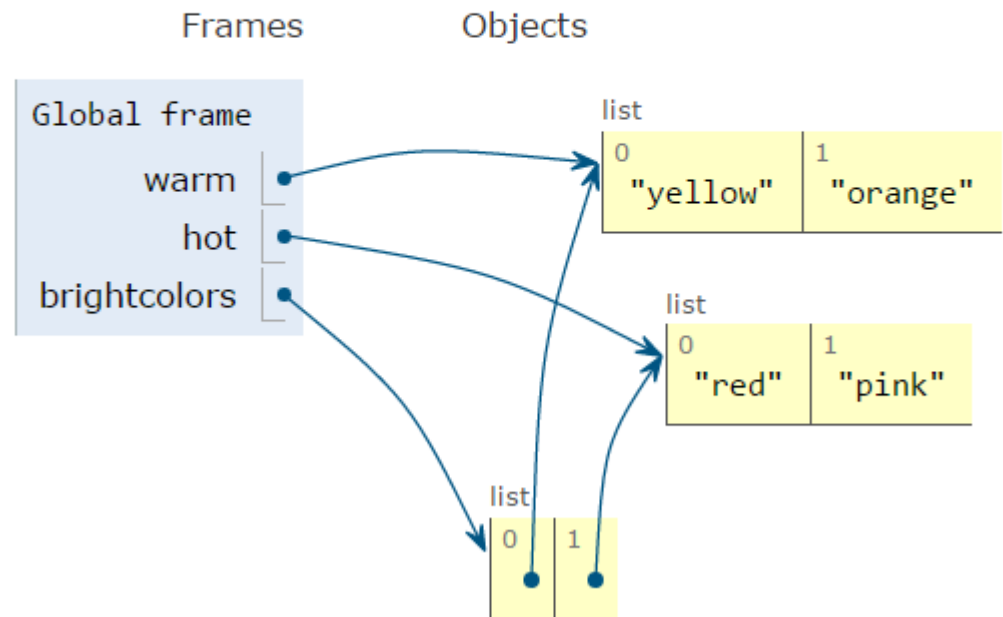


# LISTS OF LISTS OF LISTS OF....

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

```
[['yellow', 'orange'], ['red']]  
['red', 'pink']  
[['yellow', 'orange'], ['red', 'pink']]
```

```
1 warm = ['yellow', 'orange']  
2 hot = ['red']  
3 brightcolors = [warm]  
4 brightcolors.append(hot)  
5 print(brightcolors)  
6 hot.append('pink')  
7 print(hot)  
8 print(brightcolors)
```



# MUTATION AND ITERATION

## Try this in Python Tutor!

---

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

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

# MUTATION AND ITERATION

## Try this in Python Tutor!

---

- **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)
```




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

# MUTATION AND ITERATION

## Try this in Python Tutor!


---

- **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)
```

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




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

# MUTATION AND ITERATION

## Try this in Python Tutor!


---

- **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)
```

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




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

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

# MUTATION AND ITERATION

## Try this in Python Tutor!

- **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)
```

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

- 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



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

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