Topic 5

Tuples and Lists

Murdoch
UNIVERSITY

Last Topic

- structuring programs and hiding details (abstraction)
- divide-and-conquer strategy for solving large problems
- defining a function
- calling or invoking a function
- `return` statement
- scope of a function
- global variable and its scope
- local variable and its scope
- nested function
- modules

Today

- have seen variable types: `int`, `float`, `bool`, `string`, `NoneType`
- introduce new **compound data types:** tuple and list
- Access elements of a tuple or list, slicing a tuple and list
  Concatenation of two lists or two tuples
- List comprehension
- idea of mutability
- idea of aliasing and cloning
- command line arguments `sys.argv`

Tuples and Lists

- Tuple and list are two frequently used data types in Python.
- They are part of a more general data types known as "collection".
- Tuple and list are language built-in data types.
- A tuple is an ordered sequence of elements.
  - tuples are defined using parentheses, eg, `(1, "two", 3)`
- A list is also an ordered sequence of elements.
  - lists are defined using square brackets, eg, `[1, "two", 3]`
- Elements in tuples and lists are accessible using indexes

Tuples and Lists

- Tuples and lists each has its own set of built-in methods
- Some of the functions and notations apply to both, such as `len` function and syntax for indexing
- Elements in tuples and lists are heterogenous, these elements do not have to have the same type
- The main difference between the two:
  - a tuple is immutable – once created, you cannot add elements to it or remove elements from it
  - while a list is mutable, you can add elements to it, and remove elements from it

Tuples

Murdoch UNIVERSITY

- a tuple is an ordered sequence of elements, can mix element types
- cannot change element values, **immutable (like a string)**
- represented with parentheses

`t1 = ()` -> an empty tuple

`t2 = (2,"mit",3)` -> `t2` is a tuple with 3 elements

`len(t2)` -> evaluate to `3`, similar to strings

`t3 = ("One",)` -> `t3` is a tuple with a single element, note the comma at the end!

`t4 = ("One")` -> `t4` is just a string "One", not a tuple!!!

`t5 = t2 + t3` -> `t5` is a new tuple `(2,"mit",3, "One")`

`t2[1] = 4` -> gives error, because tuples cannot be modified

Indexing a Tuple

- As with strings, we can use index to get an individual element from a tuple

  `t = (1, 2, "mit", 4, True, 6.6, "7", 8, 9, "ten")`

  `x1 = t[0]` -> `t[0]` evaluate to `1`

  `x2 = t[2]` -> `t[2]` evaluate to `"mit"`

  `x3 = t[9]` -> `t[9]` evaluate to `"ten"`

  `x4 = t[10]` -> error: "`IndexError: tuple index out of range`"

- Elements in a tuple can also be tuples (nested tuple)

  `tn = (1, 2, (3.1, 3.2), 4, (5, 6, 7))`

  `tn[1]` -> evaluate to `2`

  `tn[2]` -> evaluate to `(3.1, 3.2)`

  `tn[2][1]` -> evaluate to `3.2`

Slicing a Tuple

- We can also create a new tuple out of an existing tuple by slicing the tupe. Note: no change to the existing tuple
- The syntax is similar to the syntax used to get a new string out of an existing string.

```
t = (1, 2, "mit", 4, True, 6.6, "7", 8, 9, "ten")
```

`t[1:2]` -> evaluate to a new tuple `(2,).` Note the extra comma at the end

`t[1:5]` -> evaluate to a new tuple `(2,"mit",4,True)`

`t[1:5:2]` -> evaluate to a new tuple `(2,4)`

`t[1::2]` -> evaluate to a new tuple `(2,4,6.6,8,"ten")`

`t[-1::2]` -> evaluate to a new tuple `("ten",)`

`t[-1::-2]` -> evaluate to a new tuple `("ten",8,6.6,4,2)`

-> note the elements are reversed due to negative step -2

Example: Swap Two Values

- 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)
```

integer division

Using Tuple In For Loops

```
fruits = ("apple", "banana", "orange", "mango")
vegetables = ("carrot", "broccoli", "pea", "cabbage", "spinach")
grains = ("wheat", "rice", "barley")

def count_food (food):
    fruit_c = vegetable_c=grain_c=0
    for f in food:
        if f in fruits:
            fruit_c += 1
        elif f in vegetables:
            vegetable_c += 1
        else:
            grain_c += 1
    return (fruit_c, vegetable_c, grain_c)

myfood = ("rice", "broccoli", "pea", "banana", "barley", "spinach")
print(count_food(myfood))
```

Lists

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

List Index and List Slicing

`a_list = []`	-> `a_list` is an empty list
`L1 = [1]`	-> `L1` is a list with one element,
	note: no comma at the end
`L = [2, 'a', 4, True]`	
`len(L)`	-> evaluate to 4
`L[0]`	-> evaluate to 2
`L[2]+1`	-> evaluate to 5
`L[3]`	-> evaluate to `True`
`L[1:3]`	-> evaluate to a new list `['a', 4]`
`L[-1:1:-1]`	-> evaluate to a new list `[True, 4]`
`L[4]`	-> gives an error, index out of range
`i=2`	
`L[i-1]`	-> evaluate to `'a'` since the 2nd element is `'a'`

Lists Can Be Nested

`L = [[1,2,3], [4,5], [6,7,8,9]]`

`len(L)` -> evaluate to `3`

`L[-1]` -> evaluate to `[6,7,8,9]`

`L[-1][1]` -> evaluate to `7`

Concatenate Tuples and Lists

- Two tuples can be concatenated, or merged, into a single tuple using the concatenation operator +

```
t1 = (1, 2, "mit", 4, True)
t2 = ('a','b')
t3 = t1 + t2

      t3 is (1,2,"mit",4,True,'a','b')
```

- The same operation, +, also applies to lists.

```
L = [[1,2,3], [4,5], [6,7,8,9]]
L2 = L + ['a','b']
```

-> merge two lists

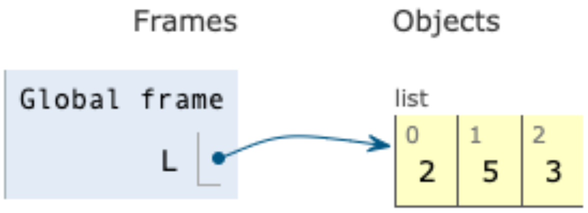
```
      L2 is [[1,2,3],[4,5],[6,7,8,9],'a','b']
```

Change List 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`



- It is important to note that the list is still the original list, with the same id, only its content has changed

Iterating Over a List

- compute the **sum of elements** of a list
- common pattern, iterate over list elements
- assuming `L` is a list of numbers:

```
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 from `0` to `len(L)-1`
- `range(n)` goes from `0` to `n-1`

List method – Add an Element to the end of the List

- **add** elements to the end of list with `append` method

```
L.append(element)
```

- this would **mutate** the list!

```
 L = [2,1,3]
 L.append(5)     -> L is now [2,1,3,5]
```

- what is the dot?
  - a list is a Python object
  - an object has a set of data
  - an object also has a set of methods
  - access a method of an object using dot .
    `object_name.do_something()`
  - will learn more about these later

List method – Extend the List

- to combine lists together use concatenation operator `+`, to give you a new list
- alternatively, you may mutate the 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]
```

List method – Remove List Elements

- delete an element at a specific index with `del(L[index])`
- remove the element at end of list with the `pop` method, as in `L.pop()`, and returns the removed element
- remove a specific element with the `remove` method, as in `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

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

Convert List to String and Back

- convert string `s` to list with `L=list(s)`, returns a list with every character from `s` an element in `L`
- use `s.split(c)`, to return a list containing two substrings split on a character parameter `c`, 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

- `sorted` function
- `sort` method
- `reverse` method
- and many more! https://docs.python.org/3/tutorial/datastructures.html
- Examples:

```
L = [9, 6, 0, 3]
L1 = sorted(L)       -> returns sorted list, does not mutate L
L.sort()             -> mutates L to [0,3,6,9]
L.reverse()          -> mutates L to [9,6,3,0]
```

List Comprehension

- List comprehension allows us to create lists with shorter syntax:

```
list = [ expr1 for item in iterable if expr2 ]
```

the above notation is equivalent to

```
list = []
for item in iterable:
    if expr2:
        list.append(expr1)
```

- The if-clause is optional:

```
list = [ expr1 for item in iterable ]
```

which is equivalent to

```
list = []
for item in iterable:
    list.append(expr1)
```

List Comprehension

- Examples:

```
 l1=[x for x in range(1,20,3)]
                     # [1,4,7,10,13,16,19]
 l2=[x*2 for x in range(1,20,3) if x>7]
                     # [20,26,32,38]
 l3=[ x*x for x in l1 if x%2==0]
                     # [16,100,256]

 fruits=["apple", "orange", "banana", "cherry" ]
 fa = [ f.upper() for f in fruits if f.endswith('e')
]
                     # [ "APPLE", "ORANGE" ]

 circleArea = [ 3.14*x**2 for x in range(1,5) ]
                     # [ 3.14, 12.56, 28.26, 50.24 ]
```

Lists in Memory

- lists are **mutable**
- behave differently than immutable types
- is an object in memory
- variable contains the memory address pointing to the object
- changes to the object affect any variable pointing to that object
- key phrase to keep in mind when working with lists is **side effects**

Aliases

- When you assign an object to a variable, the memory address at which the object is stored is copied to that variable.
- Example: `warm`:

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

- what is stored in the variable `warm` is the start address of the object `[1,2,3]`.
- The object `[1,2,3]` is actually stored somewhere else, not in variable `warm`.

Aliases

- When assigning a *variable* to another variable, such as `hot`,

  ```
  hot = warm
  ```

- what is copied to variable `hot` is not object `[1,2,3]`, rather, it is the start address of object `[1,2,3]`.

- This means both `warm` and `hot` contain the same address, which point to the same object `[1,2,3]`. We call `hot` an alias of `warm`, because they point to the same object.

- change the object pointed to by `hot` also changing the object pointed by `warm`. This is called "side effect".

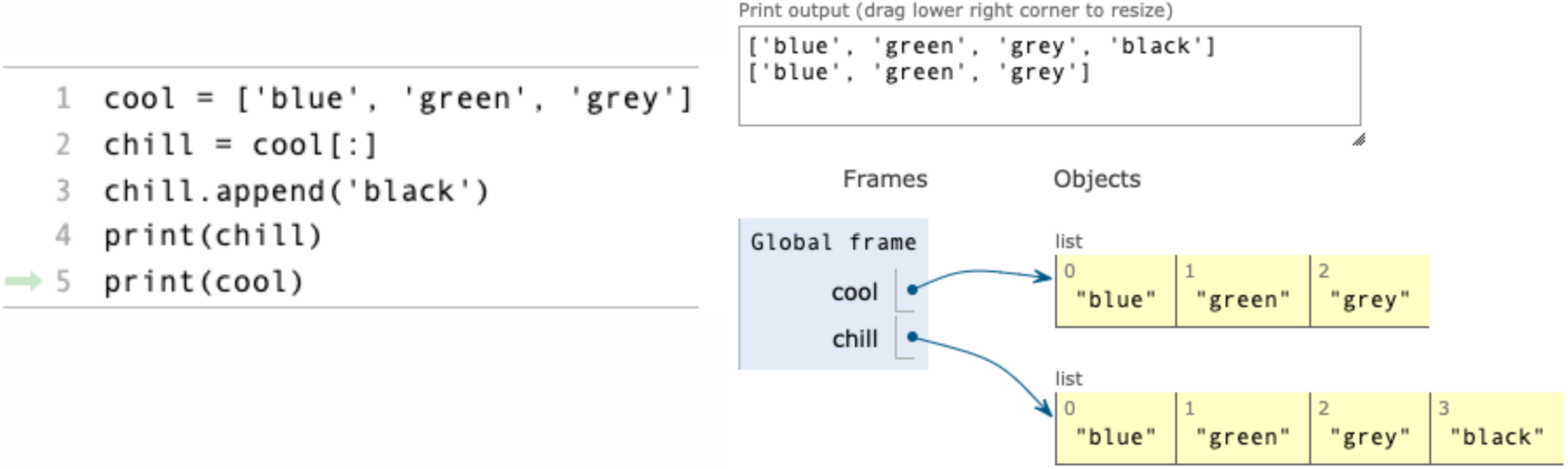
Cloning Lists

- create a new list and **copy every element** using

```
chill = cool[:].     # list slicing
```

  - `cool[:]` is short for `cool[0:len(cool)]` which returns a copy of `cool`.

- After copying, the two variables are independent of each other: `cool` still points to the original list, while `chill` points to the new list.



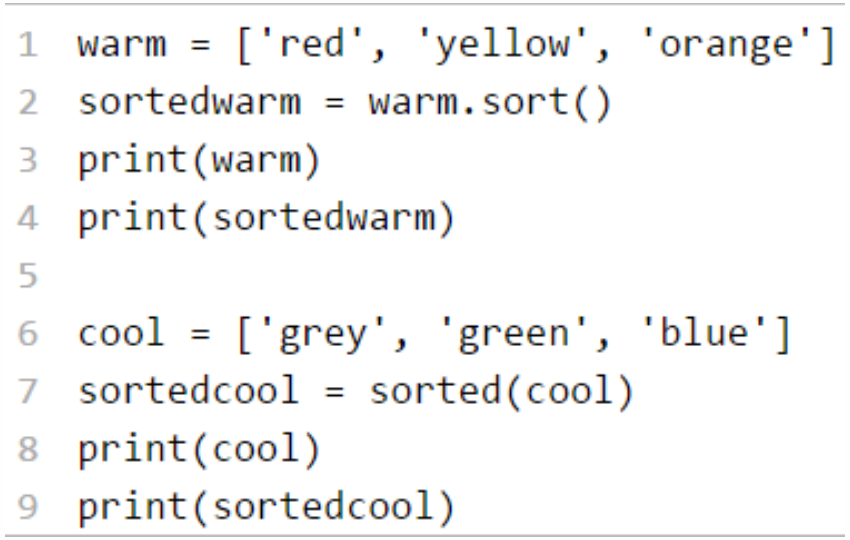
```
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```

Print output (drag lower right corner to resize)

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

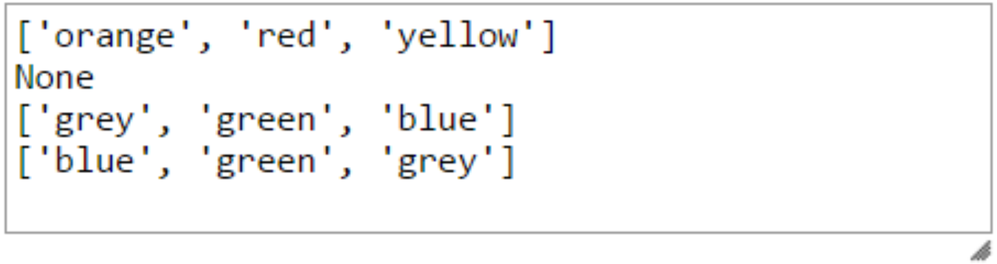
Example: Sorting Lists

- calling method `sort()` **mutates** the list, returns nothing (None)
- calling function `sorted()` **does not mutate** the list, must assign the result to a variable

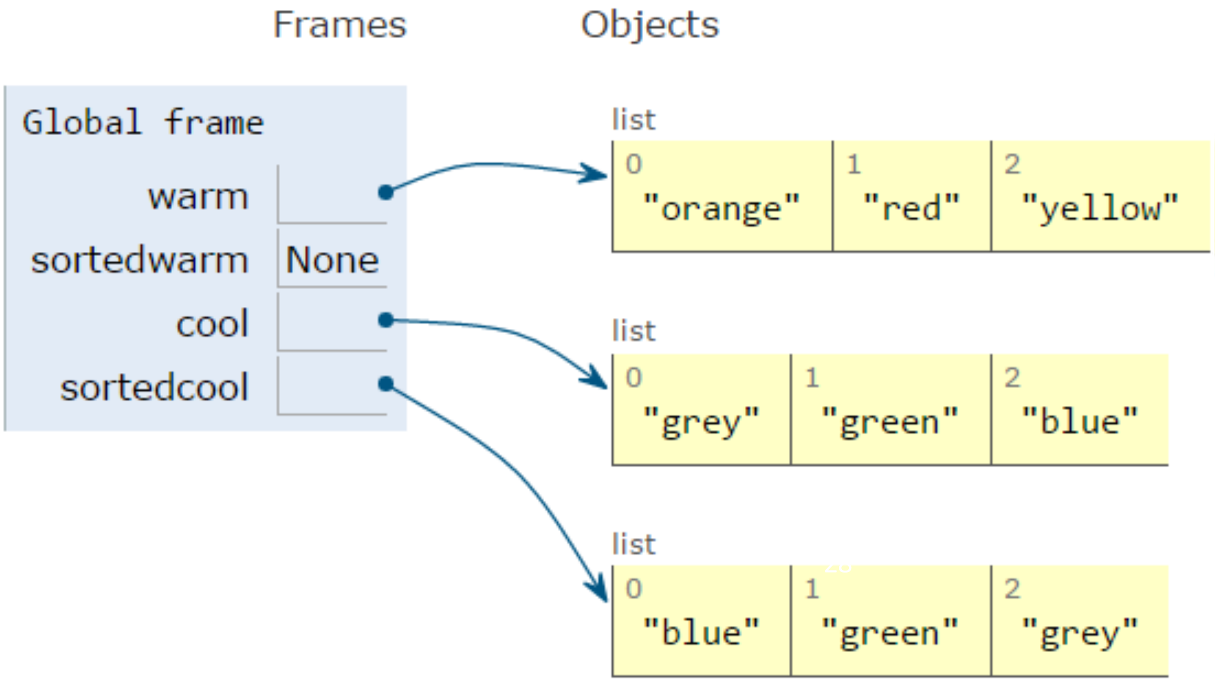


```
warm = ['red', 'yellow', 'orange']
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)

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

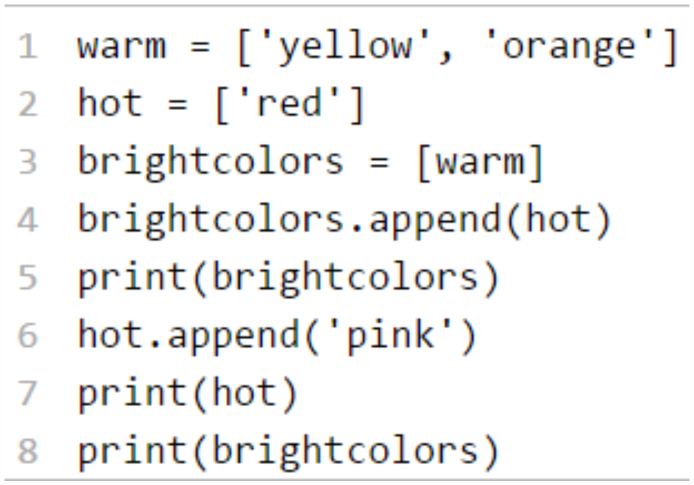


```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```

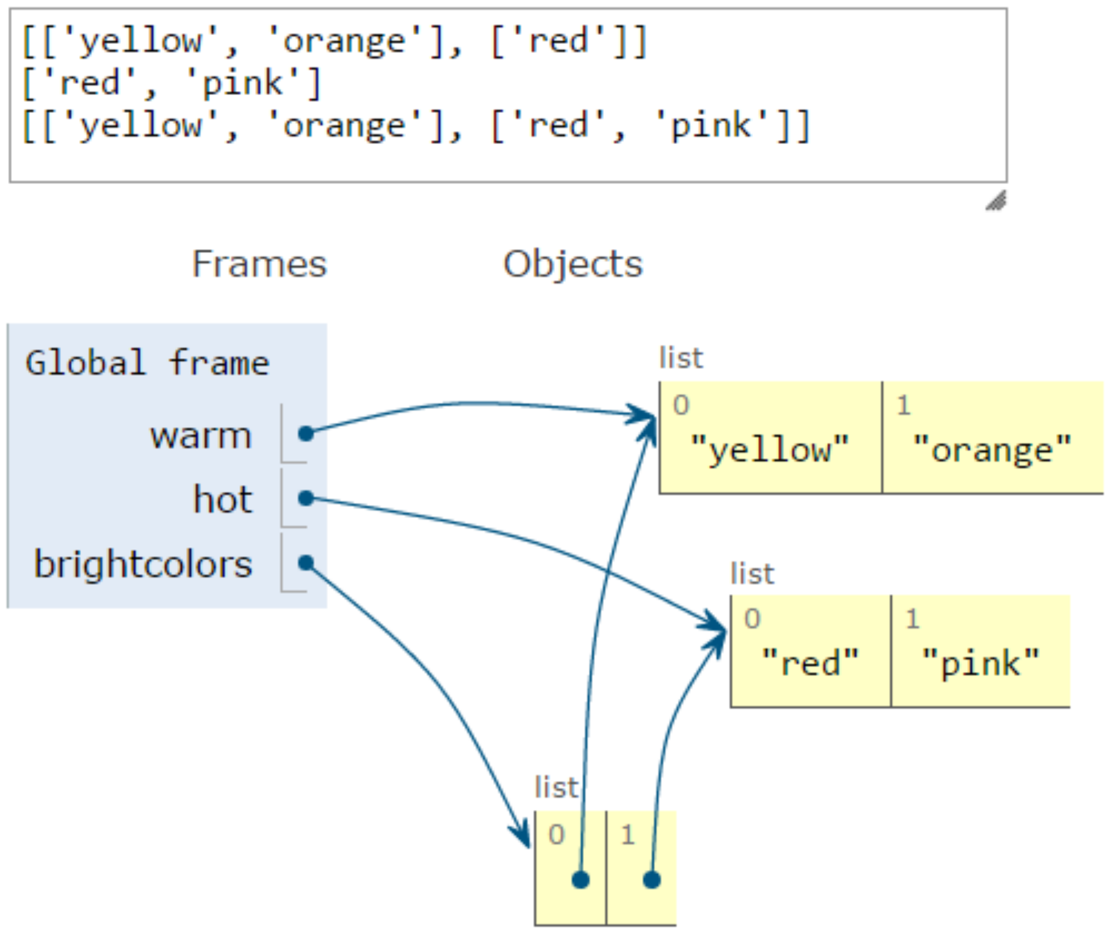


List of Lists of Lists of ….

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



```
warm = ['yellow', 'orange']
hot = ['red']
brightcolors = [warm]
brightcolors.append(hot)
print(brightcolors)
hot.append('pink')
print(hot)
print(brightcolors)
```



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

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

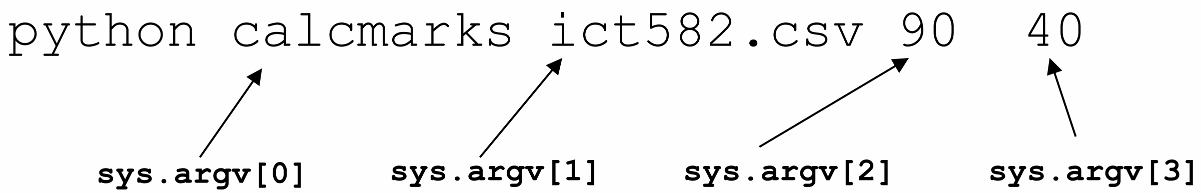
clone list first, note that `L1_copy = L1` does NOT clone

```
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`

Command line arguments

- Command line arguments are values that are passed to a Python script when it is executed from the command line.
- They provide a way to input data to your scripts at runtime.
- Command line arguments are stored in the `sys.argv` list, which is part of the `sys` module.
- `sys.argv[0]` contains the script's name, and subsequent elements contain the remaining arguments.
- Example



```
python calcmarks ict582.csv 90  40
```

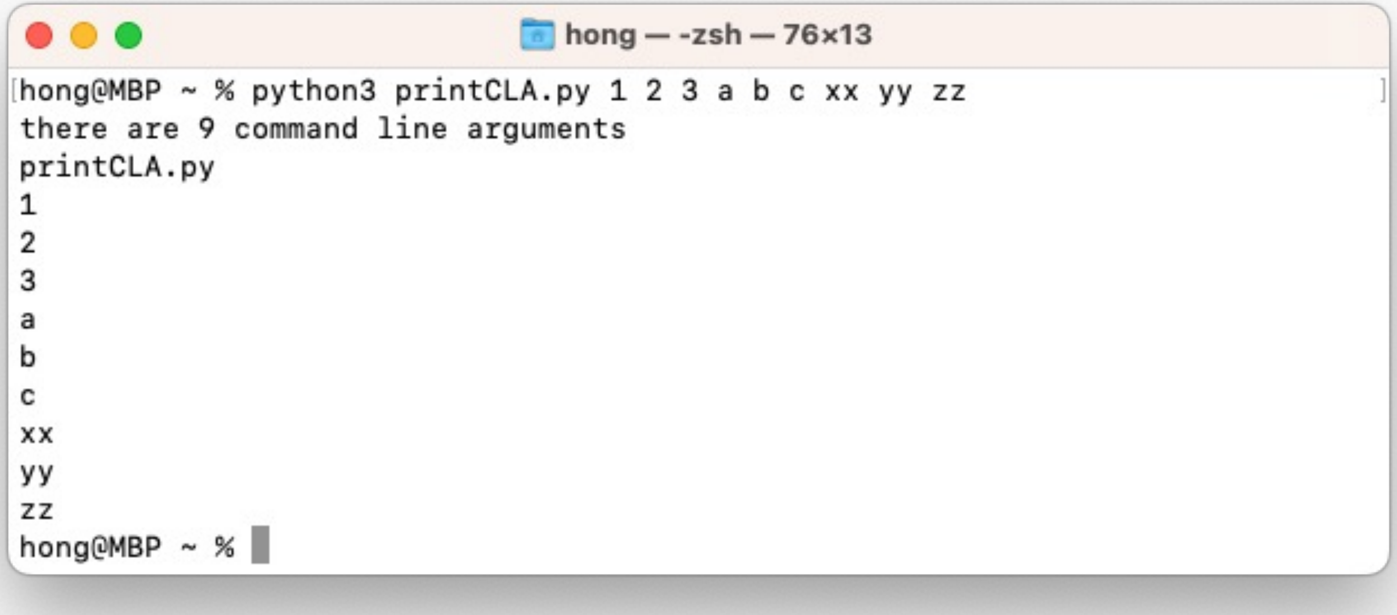
sys.argv[0] sys.argv[1] sys.argv[2] sys.argv[3]

Accessing command line arguments

- In the list `sys.argv`, the first element at index 0 is the script name. The remaining elements form the command line arguments for the program. Example: `printCLA.py`

```
import sys

print("there are %d cmd line arguments" % (len(sys.argv)-1))
for a in sys.argv:
    print(a)
```



```
hong@MBP ~ % python3 printCLA.py 1 2 3 a b c xx yy zz
there are 9 command line arguments
printCLA.py
1
2
3
a
b
c
xx
yy
zz
hong@MBP ~ %
```

Summary

- introduced new **compound data types**
  - tuples
  - lists
- discussed indexing and slicing of tuples and lists
- discussed iteration over elements of a tuple or list in a for loop
- discussed idea of mutability
- discussed concept of aliasing and cloning
- command line arguments `sys.argv`

Acknowledgement

- Python Tutor http://www.pythontutor.com/
- Programiz
- MIT OCW