CCPS109
Computer Science I
L5

Lecturer: Nhan Tran

nhantran@Ryerson.ca

Agenda

Announcement

- **Lab tomorrow: due Satureday @23:00**

Lecture:

- Mutation, Alias, Cloning
- Dictionary

Last time

- List
- Module
- File i/o

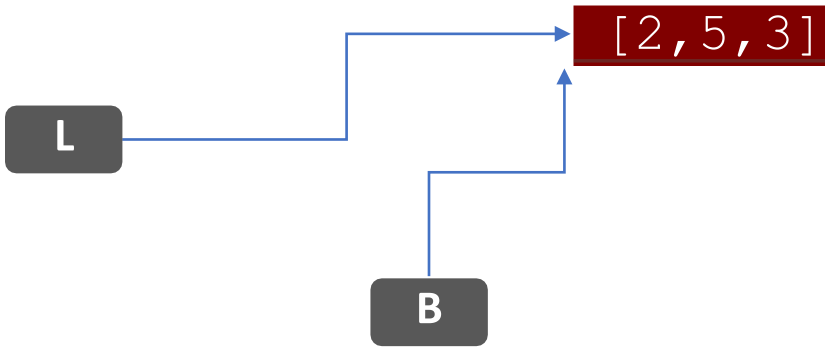
Mutation, Aliasing, Cloning

IMPORTANT
and TRICKY!

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

http://www.pythontutor.com/

List 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

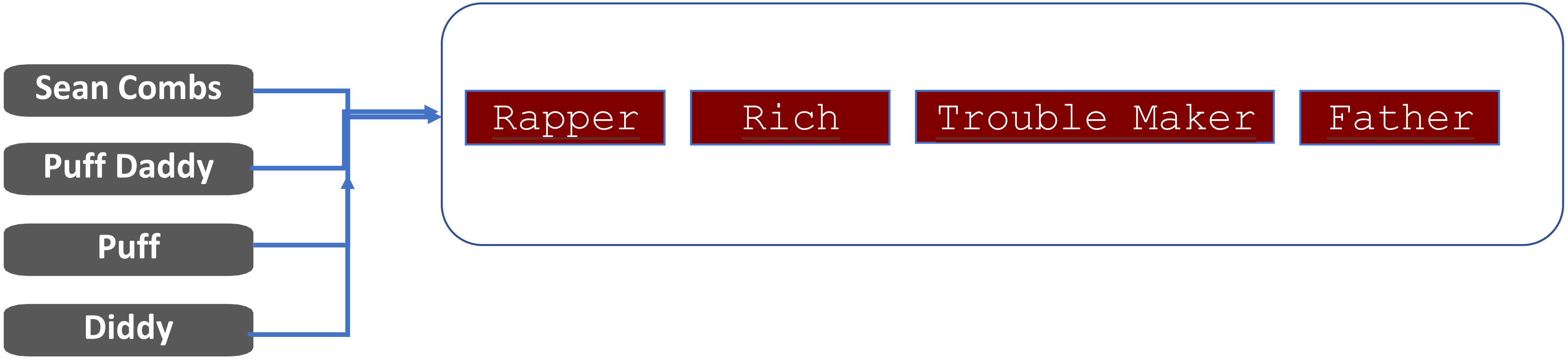
- Attributes of famous person
  - Rapper, rich
- Known by many names
- All nicknames point to the **same person**
  - add new attribute to **one nickname**

Sean Combs	Rapper	Rich	Trouble Maker

All his nicknames refer to the old attributes and all new ones too

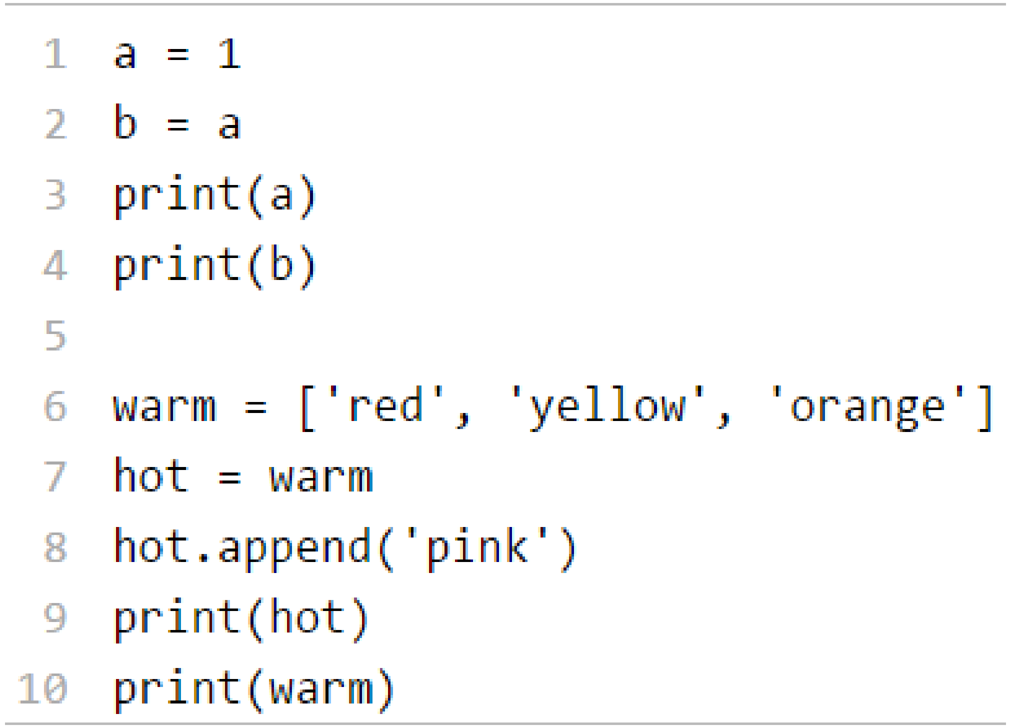
Puff Daddy	Rapper	Rich	Trouble Maker
Puff	Rapper	Rich	Trouble Maker
Diddy	Rapper	Rich	Trouble Maker

An Analogy

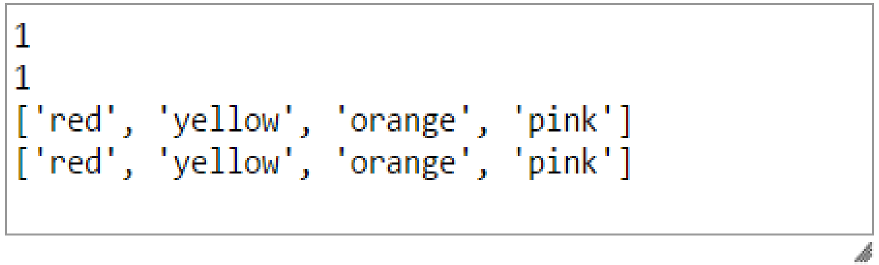

Sean Combs
Puff Daddy
Puff
Diddy
Rapper
Rich
Trouble Maker
Father

Alias

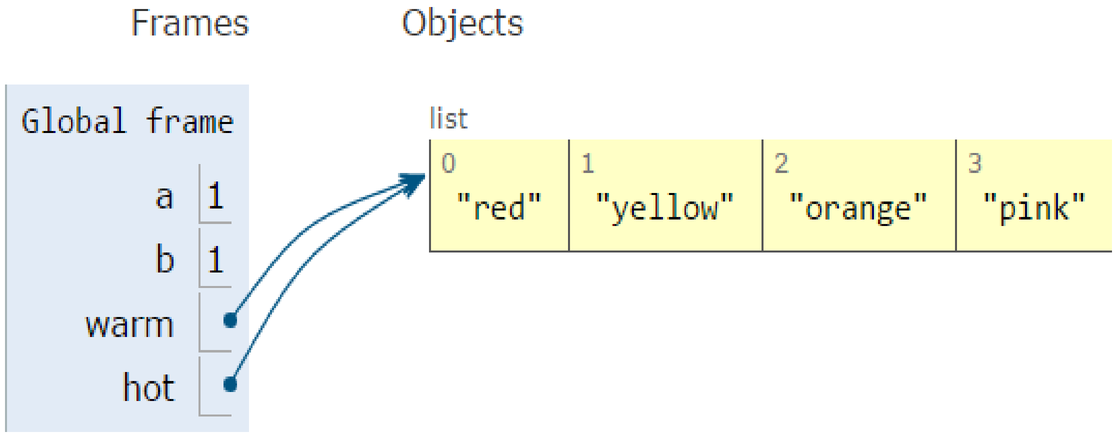
- `hot` is an **alias** for `warm` – changing one changes the other!



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

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



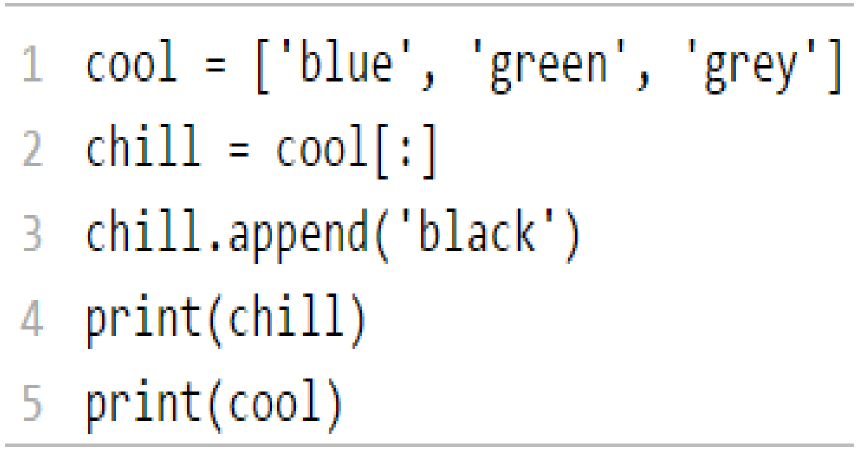
```
1
1
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```



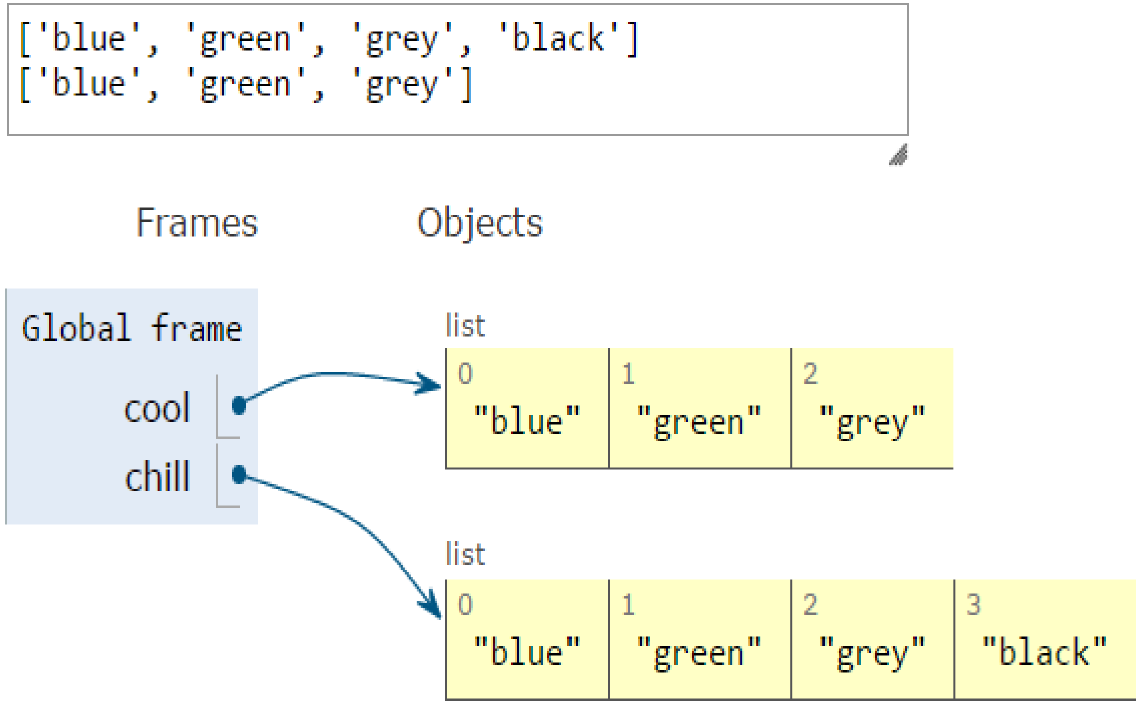
- `append()` has a side effect

Cloning

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



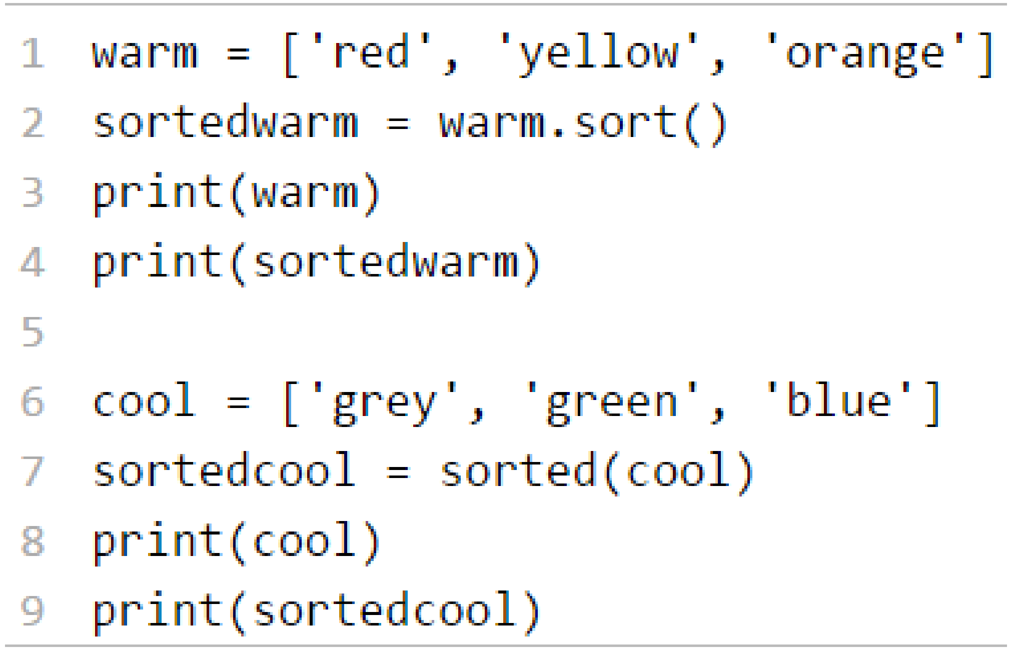
```python
cool = ['blue', 'green', 'grey']
chill = cool[:]
chill.append('black')
print(chill)
print(cool)
```



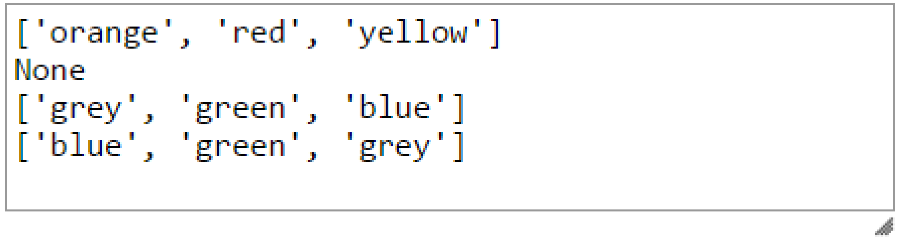
```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

Sorting Lists

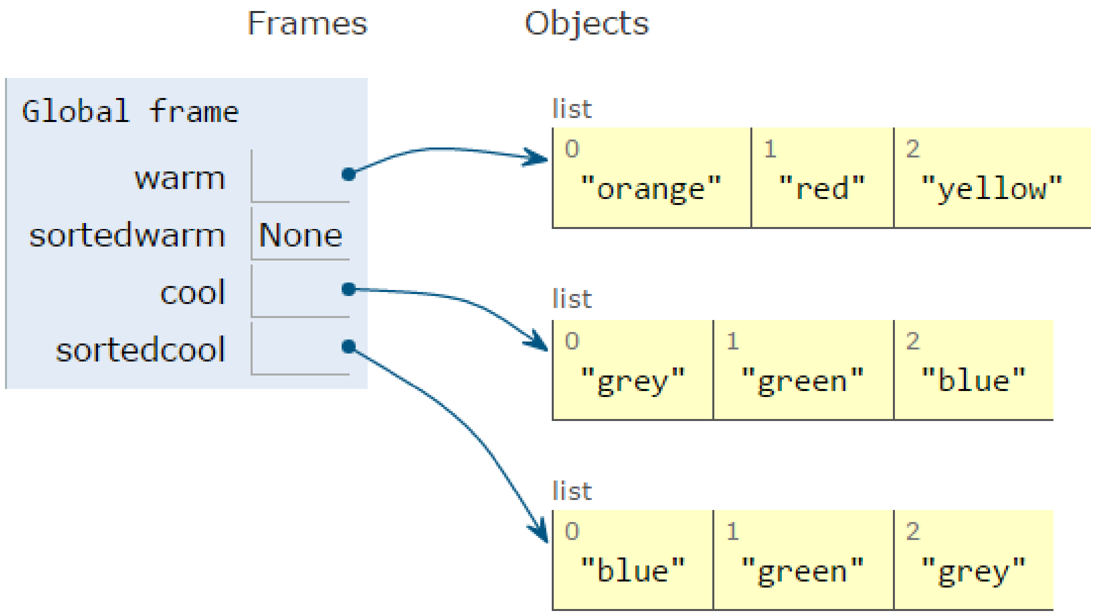
- calling `sort()` **mutates** the list, returns nothing
- calling `sorted()` **does not mutate** list, must assign result to a variable



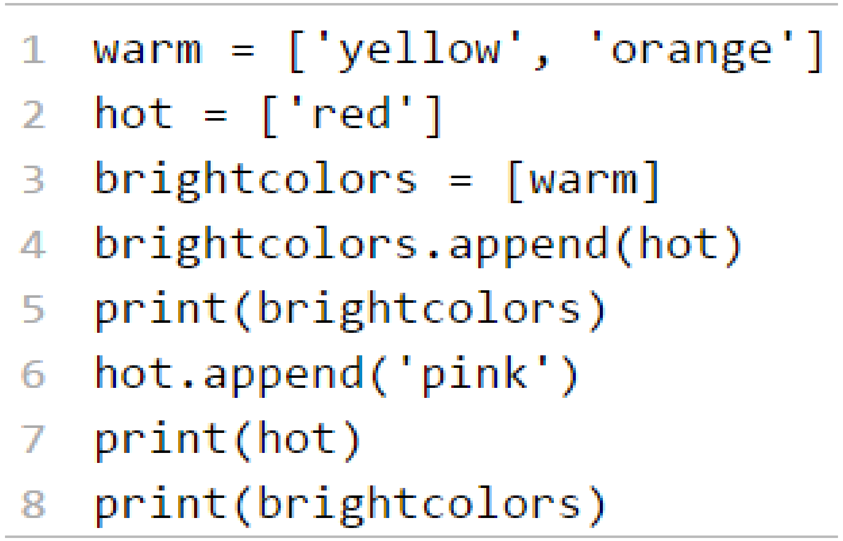
```python
warm = ['red', 'yellow', 'orange']
sortedwarm = warm.sort()
print(warm)
print(sortedwarm)

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



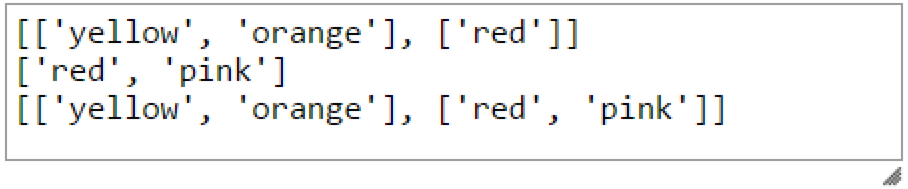
```
['orange', 'red', 'yellow']
None
['grey', 'green', 'blue']
['blue', 'green', 'grey']
```



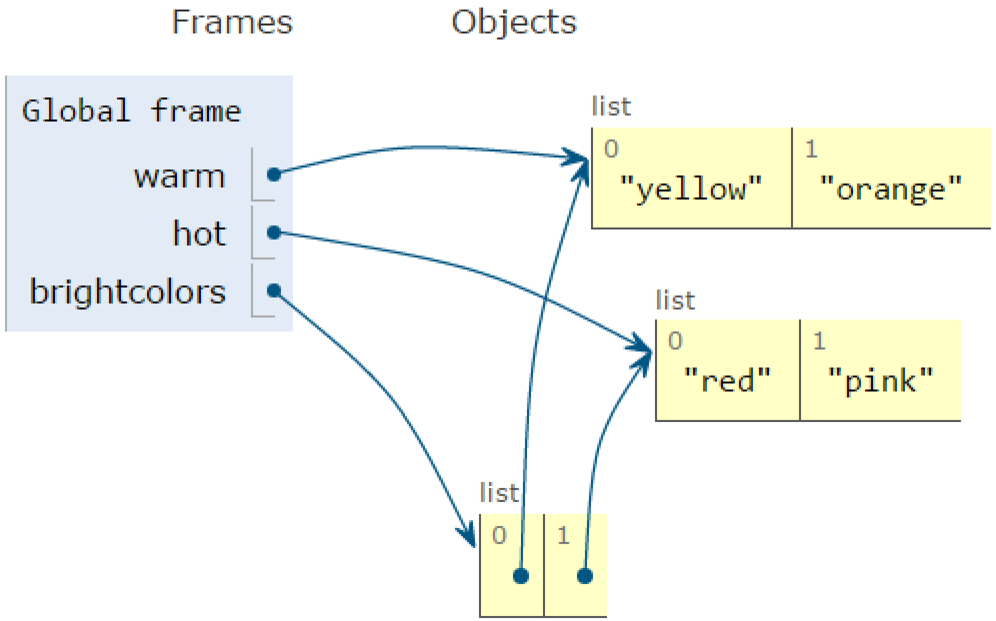
Lists of Lists of List



```
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']]
```



- side effects of Nested list still possible after mutation

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)
```

✗

```
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)
```

✓

Mutable vs Immutable

Immutable	Mutable
integers	lists
floats	dictionaries
booleans	sets
strings	numpy arrays
tuples	

Student Info Using Lists

- so far, can store using separate lists for every info

```
names =  ['Ana', 'John', 'Denise', 'Katy']
grade =  ['B', 'A+', 'A', 'A']
course = [2.00, 6.0001, 20.002, 9.01]
```

- a **separate list** for each item
- each list must have the **same length**

Student Info Using Lists

- so far, can store using separate lists for every info

```
names =  ['Ana',  'John', 'Denise', 'Katy']
grade =  ['B’,    'A+’,   'A’,       'A']
course = [2.00,   6.0001,  20.002,   9.01]
```

- a **separate list** for each item
- each list must have the **same length**
- info stored across lists at **same index**, each index refers to info for a different person

Update/Retrieve Student Info?

```
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return  (course, grade)
```

- **messy** if have a lot of different info to keep track of
- must maintain **many lists** and pass them as arguments
- must **always index** using integers
- must remember to change multiple lists

Dictionary

- nice to **index item of interest directly** (not always int)
- nice to use **one data structure**, no separate lists

A list

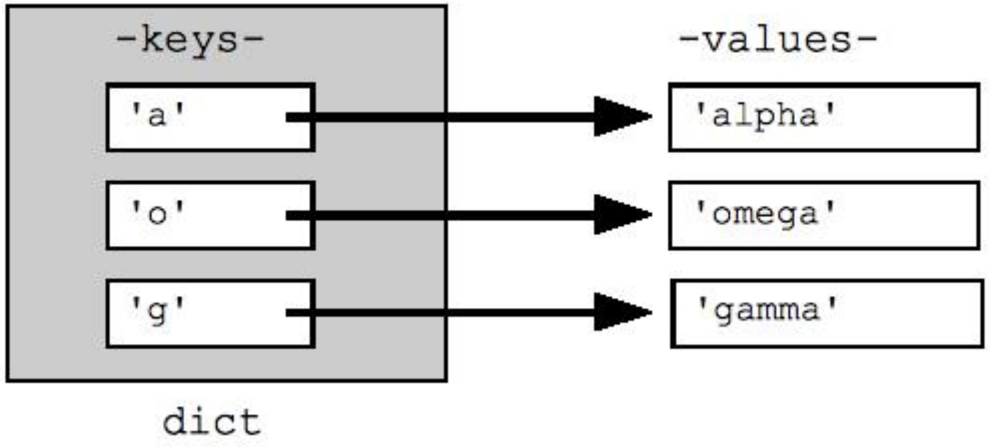
0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

index element

A dictionary

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

custom index by label element



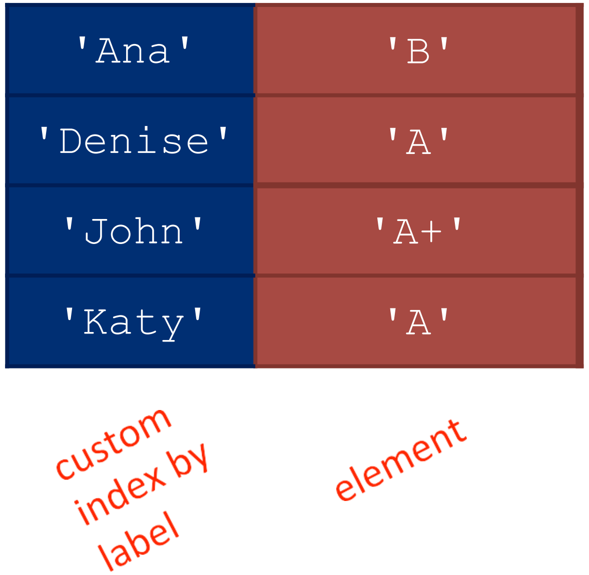
Dictionary

- Collection of unordered data values in a key:value pair structure.
- uses { }
- ‘comma’ for item separation
- Keys are unique and immutable
- Data values may be of any datatype and can be duplicate
  - Can be lists, another dictionary

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

Dictionary

- store pairs of data
  - key
  - value



'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

empty dictionary

```
my_dict = {}

grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

Dictionary Lookup

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn’t found, get an error

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

`grades['John']` evaluates to `'A+'`

`grades['Sylvan']` gives a `KeyError`

Dictionary Operations

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

- **test** if key in dictionary

`'John' in grades` returns `True`

`'Daniel' in grades` returns `False`

- **delete** entry

```
del(grades['Ana'])
```

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

Dictionary Operations

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **iterable that acts like a tuple of all keys**

  `grades.keys()` – returns `['Denise','Katy','John','Ana']`

  no guaranteed order

- **iterable that acts like a tuple of all values**

  `grades.values()` – returns `['A', 'A', 'A+', 'B’]`

  no guaranteed order

- **`Get method: return the value with the asociated key`**

```
grades.get('Katy')  -returns 'A'
```

Dictionary Operations

'Ana'	'B'
'Denise'	'A-'
'John'	'A+'
'Arthur'	'100'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A’}
```

Change Value of a key

```
grades['Denise’]=‘A-’
```

Add entry

```
grades[‘Arthur’]=‘100’
```

Pop key/value pair: ‘Katy’

```
grades.pop[‘Katy’]
```

Dictionary Operations

Tuple can serve as keys for dictionary:

```
d= {(5, 1): 'abb', (1, 2): 'bcd', (2, 1): 'cat', (4, 2): 'dog'}
d[(2,1)]   -return 'cat'
```

Try using list as keys and see what happen:

d= {[5, 1]: ‘abb', [1, 2]: ‘bcd', [2, 1]: ‘cat’, [4, 2]: ‘dog’}

Nested Dictionary on board

Dictionary Operations

Constructor : dict()

```
newdictionary=dict([(1, 4139), ('guido', 4127), ('jack', 4098)])
```

When the key are strings it may be easier using:

```
newdict=dict(sape=4139, guido=4127, jack=4098)
```

Iterating through dictionaries for key/value pairs using item()

```
for k, v in newdict.items():
    print(k, v)
```

Printing all values in dict:

```
for x in newdict.values():
      print(x)
```

Dictionary Methods

Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(*seq*[, *v*])	Return a new dictionary with keys from *seq* and value equal to *v* (defaults to None).
get(*key*[,*d*])	Return the value of *key*. If *key* doesnot exit, return *d* (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
values()	Return a new view of the dictionary's values

Dictionary Methods

Method	Description
pop(*key*[,*d*])	Remove the item with *key* and return its value or *d* if *key* is not found. If *d* is not provided and *key* is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(*key*[,*d*])	If *key* is in the dictionary, return its value. If not, insert *key* with a value of *d* and return *d* (defaults to None).
update([*other*])	Update the dictionary with the key/value pairs from *other*, overwriting existing keys.

Additional Reading: **https://docs.python.org/3/tutorial/datastructures.html**

`list` vs `dict`

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

- **matches** “keys” to “values”
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

EXAMPLE: 3 FUNCTIONS TO ANALYZE SONG LYRICS

1)create a **frequency dictionary** mapping `str:int`

2)find **word that occurs the most** and how many times

- use a list, in case there is more than one word
- return a tuple `(list,int)` for (words_list, highest_freq)

3)find the **words that occur at least X times**

- let user choose “at least Xtimes”, so allow as parameter
- return a list of tuples, each tuple is a `(list, int)` containing the list of words ordered by their frequency

- IDEA: From song dictionary, find most frequent word. Delete most common word. Repeat. It works because you are mutating the song dicSonary.

Example 3: Functions to analyze song lyrics

- Let's do the translation!
- Input: song lyrics
- Output: the most frequent words
- Phrase 1 – creating the dictionary: have the vocabulary associated with frequency

```
1)Initialize an empty dictionary: myDict = {}
2)What are the pairs in the dictionary? word:frequency
3)How to have these pairs?
   MyDict[custom index] = element
```

Example 3: Functions to analyze song lyrics

- Let's do the translation!
- Input: song lyrics
- Output: the most frequent words
- Phrase 1 – creating the dictionary: have the vocabulary associated with frequency
  1) Initialize an empty dictionary: myDict = {}
  2) What are the pairs in the dictionary? word:frequency
  3) How to have these pairs?

     MyDict[custom index] = element
- Phrase 2 – using the dictionary: find the most frequent words (maximum frequency)
  1) Find the biggest number of the frequency = the maximum value of the list of elements
  2) More than one words associated with the maximum frequency? Iteratively find them!

CREATING A DICTIONARY

```
def lyrics_to_frequencies(lyrics):
    myDict = {}   # Initialize an empty dictionary
    for word in lyrics:                  # custom index: word
        if word in myDict:               # element: frequency
            myDict[word] += 1  # pairs: word:frequency
        else:
            myDict[word] = 1
    return myDict
```

can iterate over list

can iterate over keys in dictionary

update value associated with key

USING THE DICTIONARY

```
def most_common_words(freqs):
    values = freqs.values()
                  # the maximum value of the list of elements
    best = max(values)
    words = []
    for k in freqs: # Iteratively find them!
        if freqs[k] == best:
            words.append(k)
    return (words, best)
```

this is an iterable, so can
apply built-in function

can iterate over keys
in dictionary

LEVERAGING DICTIONARY PROPERTIES

```
def words_often(freqs, minTimes):
    result = []
    done = False
    while not done:
        temp = most_common_words(freqs)
        if temp[1] >= minTimes:
            result.append(temp)
            for w in temp[0]:
                del(freqs[w])
        else:
            done = True
    return result


print(words_often(beatles, 5))
```

can directly mutate dictionary; makes it easier to iterate

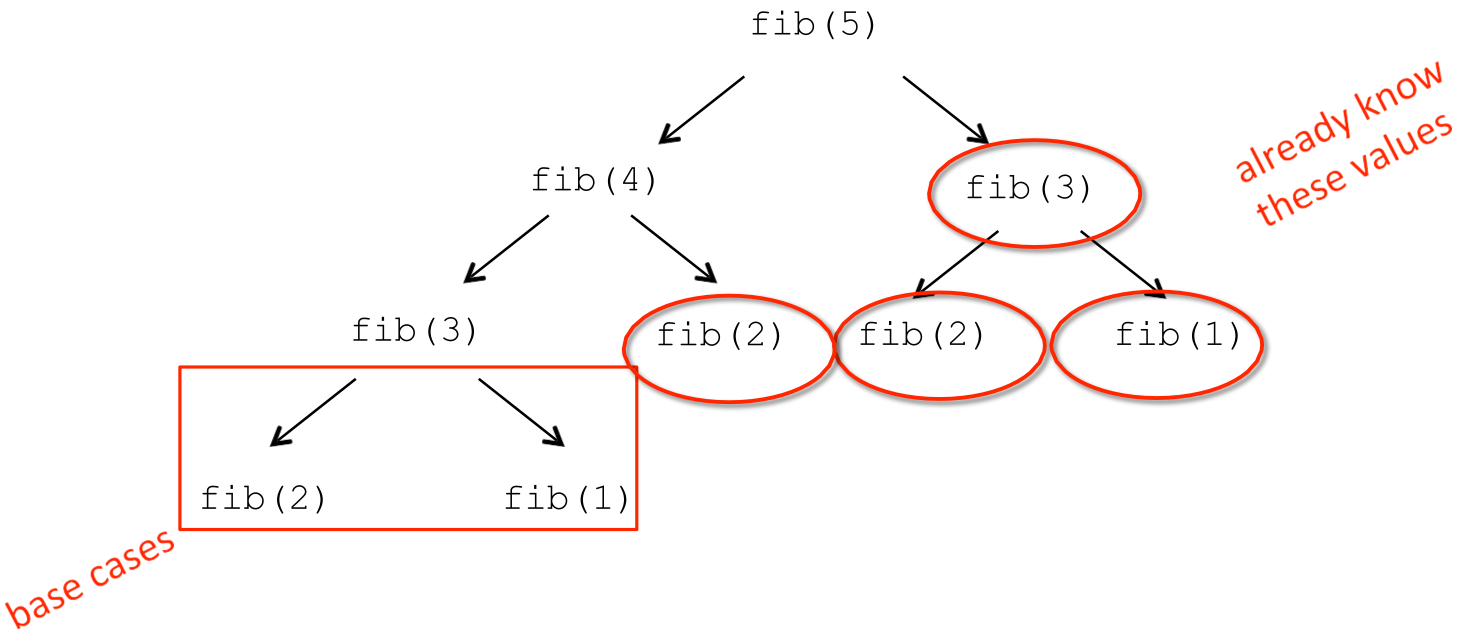
FIBONACCI RECURSIVE CODE

```
def fib(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1) + fib(n-2)
```

- two base cases
- calls itself twice
- this code is inefficient

INEFFICIENT FIBONACCI

`fib(n) = fib(n-1) + fib(n-2)`



- **recalculating** the same values many Smes!
- could keep **track** of already calculated values

FIBONACCI WITH A DICTIONARY

```
def fib_efficient(n,d):
   if n in d:

   return d[n]  else:
    ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
    d[n] = ans
    return  ans
```

Method sometimes called " memoization"

Initialize dictionary with base cases

```
d = {1:1, 2:2}
print(fib_efficient(6, d))
```

- do a **lookup first** in case already calculated the value
- **modify dic0onary** as progress through funcSon calls

EFFICIENCY GAINS

- Calling fib(34) results in 11,405,773 recursive calls to the procedure

- Calling fib_efficient(34) results in 65 recursive calls to the procedure

- Using dictionaries to capture intermediate results can be very efficient
- But note that this only works for procedures without side effects (i.e., the procedure will always produce the same result for a specific argument independent of any other computations between calls)