Chapter 5: Lists, sets, dictionaries and tuples

Contents

- Lists
- MapReduce and List Comprehension
- Sets
- Dictionaries
- Memoized recursion
- Tuples

Lists

A List is a kind of Collection

- A collection allows us to put many values in a single "variable"
- A collection is nice because we can carry many values around in one convenient package.

```
friends = [ 'Joseph', 'Glenn', 'Sally' ]

carryon = [ 'socks', 'shirt', 'perfume'
```

What is not a “Collection”

- Most of our variables have one value in them - when we put a new value in the variable - the old value is over written

```
>>> x = 2
>>> x = “Hello”
>>> x = 4
>>> print(x)
4
```

List Constants

- List constants are surrounded by square brackets and the elements in the list are separated by commas.
- A list element can be any Python object - even another list
- A list can be empty

```
>>> print([1, 24, 76])
[1, 24, 76]
>>> print(['red', 'yellow', 'blue'])
['red', 'yellow', 'blue']
>>> print(['red', 24, 98.6])
['red', 24, 98.6]
>>> print([ 1, [5, 6], 7])
[1, [5, 6], 7]
>>> print([])
[]
```

Using a List : an example

```
for i in [5, 4, 3, 2, 1] :
    print(i)
```

```
5
4
3
2
1
```

Lists and definite loops

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends :
    print('Happy New Year: ',  friend)
print('Done!')
```

Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!

Looking Inside Lists

- Just like strings, we can get at any single element in a list using an index specified in square brackets


```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> print(friends[1])
Glenn
```

Lists are Mutable

- Strings are "immutable" - we cannot change the contents of a string - we must make a new string to make any change
- Lists are "mutable" - we can change an element of a list using the index operator

```
>>> fruit = 'Banana'
>>> fruit[0] = 'b'
Traceback
TypeError: 'str' object does not

support item assignment
>>> x = fruit.lower()
>>> print(x)
banana

>>> lotto = [2, 14, 26, 41, 63]
>>> print(lotto)
[2, 14, 26, 41, 63]
>>> lotto[2] = 28
>>> print(lotto)
[2, 14, 28, 41, 63]
```

How Long is a List?

- The len() function takes a list as a parameter and returns the number of elements in the list
- Actually len() tells us the number of elements of any set or sequence (i.e. such as a string...)

```
>>> greet = 'Hello Bob'
>>> print(len(greet))
9

>>> x = [ 1, 2, 'joe', 99]
>>> print(len(x))
4
```

Using the range function

- The range function returns a list of numbers that range from zero to one less than the parameter

```
>>> print(range(4))
[0, 1, 2, 3]
```

Using range in a for loop

```
friends = ['Joseph', 'Glenn', 'Sally']
for i in range(len(friends)) :
    friend = friends[i]
    print('Happy New Year: ',  friend)
```

Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally

Iterating over a loop

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

like strings, can iterate over list elements directly

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

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

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

Concatenating lists using +

- We can create a new list by adding two existing lists together

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print(c)
[1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3]
```

Lists can be sliced using :

- Remember: Just like in strings, the second number is "up to but not including"

```
>>> t = [9, 41, 12, 3, 74, 15]
>>> t[1:3]
[41,12]
>>> t[:4]
[9, 41, 12, 3]
>>> t[3:]
[3, 74, 15]
>>> t[:]
[9, 41, 12, 3, 74, 15]
```

Negative indexing

- Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing in lists
my_list = ['p','r','o','b','e']

print(my_list[-1])
# e

print(my_list[-5])
# p
```

List Methods

```
>>> x = [1, 2, 3]
>>> type(x)
<type 'list'>
>>> dir(x)
['append', 'count', 'extend', 'index', 'insert',
 'pop', 'remove', 'reverse', 'sort']
```

http://docs.python.org/tutorial/datastructures.html

Building a list from scratch

- We can create an empty list and then add elements using the append method
- The list stays in order and new elements are added at the end of the list

```
>>> stuff = list()
>>> stuff.append('book')
>>> stuff.append(99)
>>> print(stuff)
['book', 99]

>>> stuff.append('cookie')
>>> print(stuff)
['book', 99, 'cookie']
```

Is Something in a List?

- Python provides two operators that let you check if an item is in a list
- These are logical operators that return True or False
- They do not modify the list

```
>>> some = [1, 9, 21, 10, 16]
>>> 9 in some
True
>>> 15 in some
False
>>> 20 not in some
True
```

Built in Functions and Lists

- There are a number of functions built into Python that take lists as parameters
- Remember the loops we built? These are much simpler

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums) / len(nums))
25
```

http://docs.python.org/lib/built-in-funcs.html

A List is an Ordered Sequence

- A list can hold many items and keeps those items in the order until we do something to change the order
- A list can be sorted (i.e. we can change its order)

```
>>> friends = [ 'Joseph', 'Glenn', 'Sally' ]
>>> friends.sort()
>>> print(friends)
['Glenn', 'Joseph', 'Sally']
>>> print(friends[1])
Joseph
```

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]
```

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]`

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']
L = ['a','b','c']     → L is a list
''.join(L)            → returns "abc"
'_'.join(L)           → returns "a_b_c"
```

Best Friends: Strings and Lists

- Split breaks a string into parts produces a list of strings. We think of these as words. We can access a particular word.

```
>>> abc = 'With three words'
>>> stuff = abc.split()
>>> print(stuff)
['With', 'three', 'words']
>>> print(len(stuff))
3
>>> print(stuff[0])
With
```

```
>>> line = 'A lot                    of spaces'
>>> etc = line.split()
>>> print(etc)
['A', 'lot', 'of', 'spaces']

>>> line = 'first;second;third'
>>> thing = line.split()
>>> print(thing)
['first;second;third']
>>> print(len(thing))
1

>>> thing = line.split(';')
>>> print(thing['first', 'second', 'third'])
>>> print(len(thing))
3
```

You can specify what delimiter character to use in the splitting.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

```
fhand = open('mbox-short.txt')
for line in fhand:
     line = line.rstrip()
     if not line.startswith('From '):
          continue
     words = line.split()
print(words[2])
```

Sat

Fri

Fri

Fri

…

The Double Split Pattern

- Sometimes we split a line one way and then grab one of the pieces of the line and split that piece again

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

```
words = line.split()
email = words[1]                    stephen.marquard@uct.ac.za
pieces = email.split('@')           ['stephen.marquard', 'uct.ac.za']

print(pieces[1])                    'uct.ac.za'
```

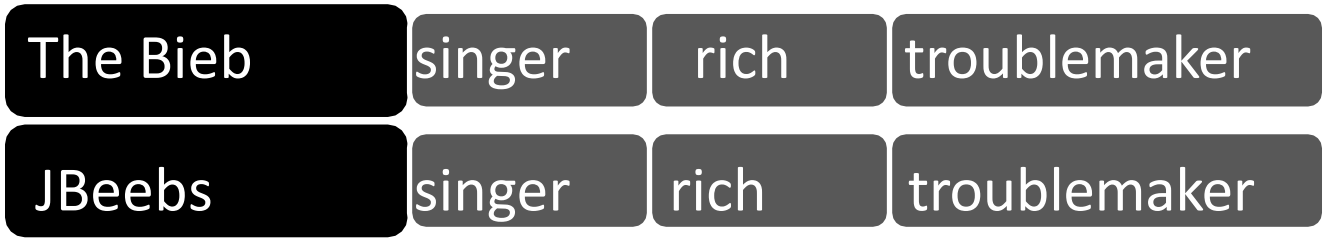
LISTS IN MEMORY

- 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
- he is known by many names
- all nicknames point to the **same person**
  - ✓ add new attribute to **one nickname** …

- … **all his nicknames** refer to old attributes AND all new ones

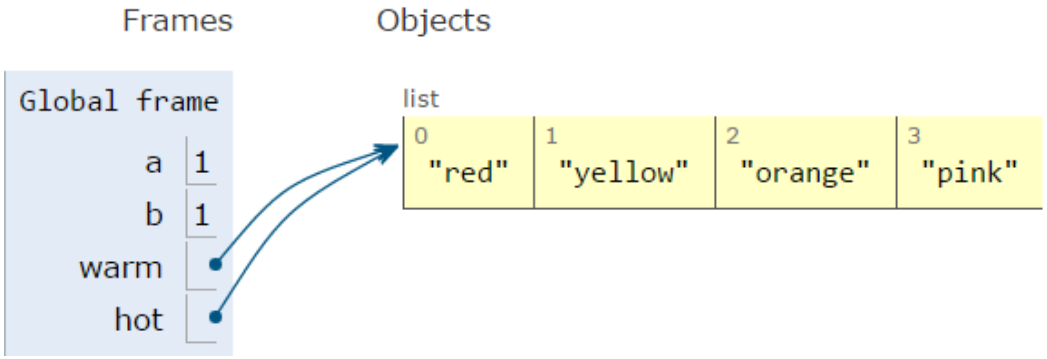


ALIASES

- `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']
```



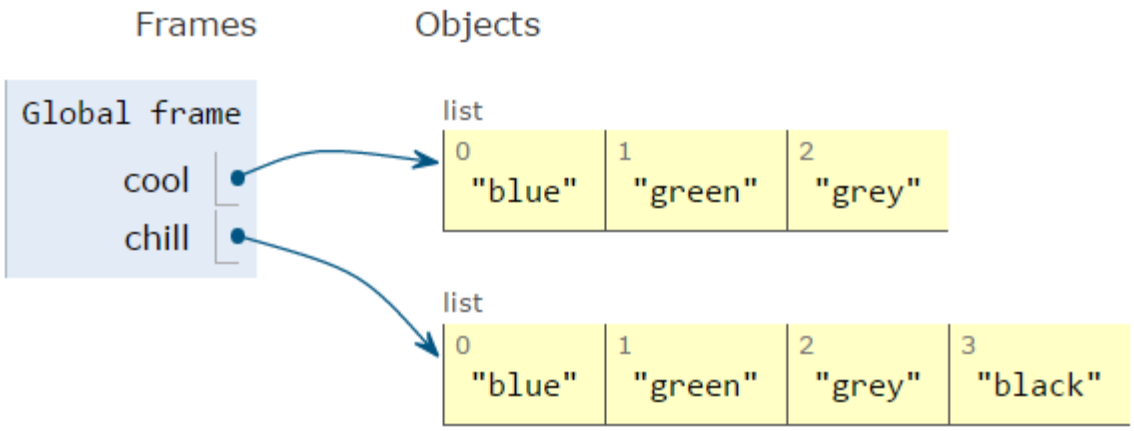
CLONING A LIST

- 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']
```

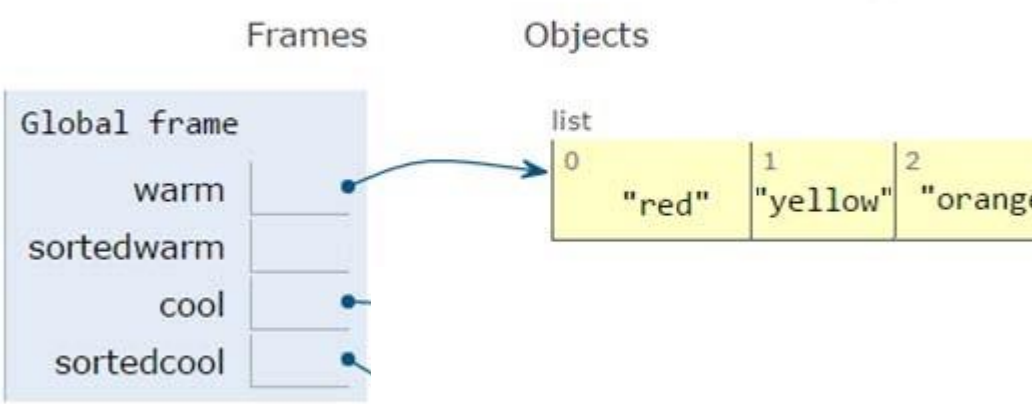


SORTING LISTS

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

```
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)
```

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

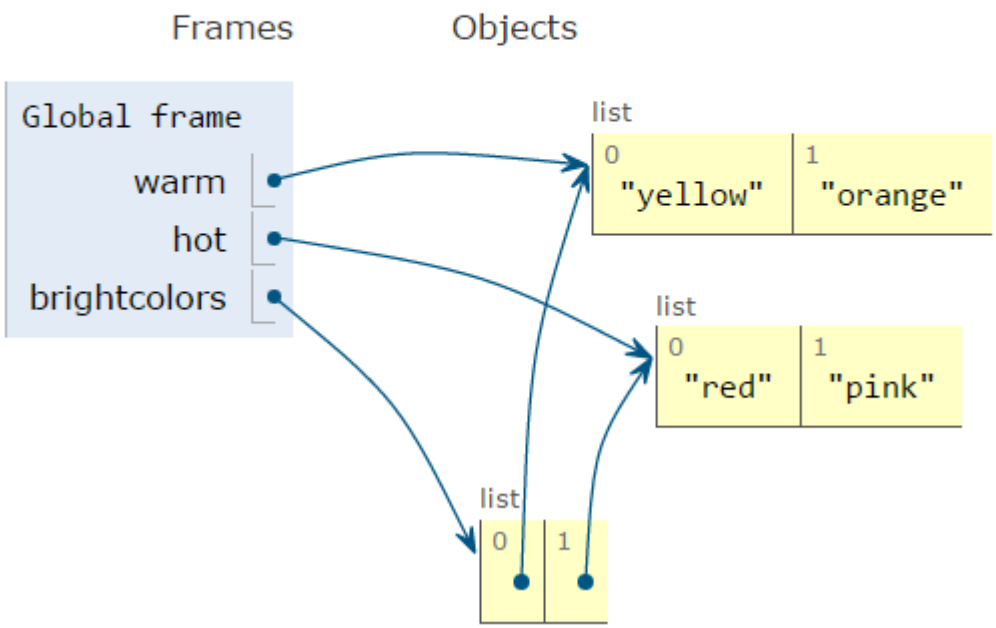


LISTS OF LISTS OF LISTS OF….

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

```
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)
```

```
[['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)
```

```
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

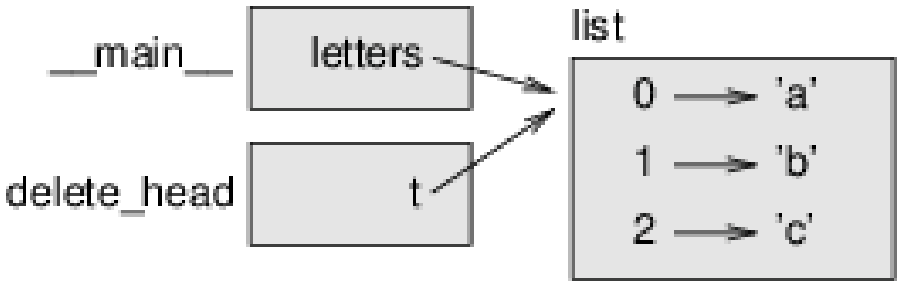
List arguments

- When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, delete_head removes the first element from a list:

```
def delete_head(t):
    del t[0]
```

- Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```



List arguments

- This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list:

```
def bad_delete_head(t):
    t = t[1:]                    # WRONG!
```

- The slice operator creates a new list and the assignment makes t refer to it, but that doesn't affect the caller.

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

MapReduce and List Comprehension

map

```
map(function, iterable, ...)
```

- Map applies function to each element of iterable and creates a list of the results
- You can optionally provide more iterables as parameters to map and it will place tuples in the result list
- Map returns an iterator which can be cast to list

map

```
nums = [0, 4, 7, 2, 1, 0 , 9 , 3, 5, 6, 8, 0, 3]
nums = list(map(lambda x : x % 5, nums))
print(nums)
#[0, 4, 2, 2, 1, 0, 4, 3, 0, 1, 3, 0, 3]
def even (x):
    if (x % 2 == 0):
        return "even"
    else:
        return "odd"
list (map(even, nums))
#['even', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'odd', 'odd', 'even', 'even', 'even', 'odd']
```

reduce

```
reduce(function, iterable[,initializer])
```

- Reduce will apply function to each element in iterable along with the sum so far and create a cumulative sum of the results
- function must take two parameters
- If initializer is provided, initializer will stand as the first argument in the sum
- Unfortunately in python 3 reduce() requires an import statement
  - `from functools import reduce`

reduce

```
nums = [9, 2, 0, -4, 0, 0, 7, 5, 3, 8]

reduce(lambda x, y: x+y, nums)
# 30

foo = ['once', 'upon', 'a', 'time', 'in', 'a'
, 'far', 'away']

reduce(lambda x, y : x + y, foo)
# 'onceuponatimeinafaraway'
```

reduce

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

reduce(lambda a, b: a + b, numlists, [])

# [1, 2, 3, 4, 5, 6, 7, 8, 9]

nums = [1, 2, 3, 4, 5, 6, 7, 8]

nums = list(reduce(lambda x, y : (x, y), nums))

print(nums)
#(((((((1, 2), 3), 4), 5), 6), 7), 8)
```

A reduce problem

- Goal: given a list of numbers I want to find the average of those numbers in a few lines using reduce()

- For Loop Method:
  - sum up every element of the list
  - divide the sum by the length of the list

A reduce problem

```
nums = [92, 27, 63, 43, 88, 8, 38, 91, 47, 74, 18
, 16, 29, 21, 60, 27, 62, 59, 86, 56]

sum = reduce(lambda x, y : x + y, nums)/len(nums)
```

MapReduce

- Framework for processing huge datasets on certain kinds of distributable problems
- **Map Step:**

  - master node takes the input, chops it up into smaller sub-problems, and distributes those to worker nodes.

  - worker node may chop its work into yet small pieces and redistribute again

- **Reduce Step:**

  - master node then takes the answers to all the subproblems and combines them in a way to get the output

MapReduce

- **Problem:** Given an email how do you tell if it is spam?
- **Solution:** Count occurrences of certain words. If they occur too frequently the email is spam.

```
email = ['the', 'this', 'annoy', 'the', 'the', 'annoy']
>>> def inEmail (x):
        if (x == "the"):
            return 1;
        else:
            return 0;
 >>> map(inEmail, email)
[1, 0, 0, 0, 1, 1, 0]

>>> reduce ((lambda x, xs: x + xs), map(inEmail, email))
3
```

List comprehensions

```
[expression for element in list]
```

- Applies the expression to each element in the list
- You can have 0 or more for or if statements
- If the expression evaluates to a tuple it must be in parenthesis

List comprehensions

- You can do most things that you can do with map, filter and reduce more nicely with list comprehensions
- The email spam program from earlier using list comprehensions:

```
>>> email = ['once', 'upon', 'a', 'time', 'in
', 'a', 'far', 'away']

>>> len( [1 for x in email if x == 'a'] )

>>> 2
```

List comprehensions

- You can do most things that you can do with map, filter and reduce more nicely with list comprehensions
- The email spam program from earlier using list comprehensions:

```
>>> email = ['once', 'upon', 'a', 'time', 'in
', 'a', 'far', 'away']

>>> len( [1 for x in email if x == 'a'] )

>>> 2
```

List Comprehensions vs For Loop

```
h_letters = []

for letter in 'human':
    h_letters.append(letter)

print(h_letters)
#['h', 'u', 'm', 'a', 'n']
```

- List comprehension is an elegant way to define and create lists based on existing lists.

```
h_letters = [ letter for letter in 'human' ]
print( h_letters)
#['h', 'u', 'm', 'a', 'n']
```

List Comprehensions vs Lambda functions

```
h_letters = [ letter for letter in 'human' ]
print( h_letters)
#['h', 'u', 'm', 'a', 'n’]

letters = list(map(lambda x: x, 'human'))
print(letters)
#['h', 'u', 'm', 'a', 'n’]
```

- List comprehensions are usually more human readable than lambda functions. It is easier to understand what the programmer was trying to accomplish when list comprehensions are used.

Conditionals in List Comprehensions

```
number_list = [ x for x in range(20) if x % 2 == 0]
print(number_list)
#[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

num_list = [y for y in range(100) if y % 2 == 0 if y %
5 == 0]
print(num_list)
#[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

obj = ["Even" if i%2==0 else "Odd" for i in range(10)]
print(obj)
#['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even',
'Odd', 'Even', 'Odd']
```

Nested Loops in List Comprehensions

```
transposed = []
matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]

for i in range(len(matrix[0])):
    transposed_row = []

    for row in matrix:
        transposed_row.append(row[i])
    transposed.append(transposed_row)

print(transposed)
```

Nested Loops in List Comprehensions

```
transposed = []
matrix = [[1, 2, 3, 4], [4, 5, 6, 8]]
for i in range(len(matrix[0])):
    transposed_row = []

    for row in matrix:
        transposed_row.append(row[i])
    transposed.append(transposed_row)
print(transposed)
#[[1, 4], [2, 5], [3, 6], [4, 8]]

matrix = [[1, 2], [3,4], [5,6], [7,8]]
transpose = [[row[i] for row in matrix] for i in range(2)]
print (transpose)
#[[1, 3, 5, 7], [2, 4, 6, 8]]
```

Sets

Sets

- A set is an unordered collection with no duplicate elements.

```
>>> basket = {'apple', 'orange', 'apple', 'pe
ar', 'orange', 'banana'}
>>> print(basket)
# show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket
# fast membership testing
True
>>> 'crabgrass' in basket
False
```

Sets

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b              # letters in both a and b
{'a', 'c'}
>>> a ^ b              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Set comprehensions

```
>>> a = {x for x in 'abracadabra' if x not in
 'abc'}
>>> a
{'r', 'd'}
```

Dictionaries

What is a Collection?

- A collection is nice because we can put more than one value in them and carry them all around in one convenient package.
- We have a bunch of values in a single “variable”
- We do this by having more than one place “in” the variable.
- We have ways of finding the different places in the variable

What is not a “Collection”

- Most of our variables have one value in them - when we put a new value in the variable - the old value is over written

```
>>> x = 2
>>> x = 4
>>> print(x)
4
```

A Story of Two Collections..

- List
  - A linear collection of values that stay in order
- Dictionary
  - A “bag” of values, each with its own label (key)
  - In other words, a collection of Key/Value pairs

Dictionaries (Associative Arrays)

- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries have different names in different languages
  - Dictionaries – Python, Objective-C, Smalltalk, REALbasic
  - Hashes – Ruby, Perl,
  - Maps – C++, Java, Go, Clojure, Scala, OCaml, Haskell
  - Property Bag - C#

Dictionaries

- Lists index their entries based on the position in the list
- Dictionaries are like bags - no order
- So we index the things we put in the dictionary with a "lookup tag"

```
>>> purse = dict()
>>> purse['money'] = 12
>>> purse['candy'] = 3
>>> purse['tissues'] = 75
>>> print(purse)
{'money': 12, 'tissues': 75, 'candy': 3}
>>> print(purse['candy'])
3
>>> purse['candy'] = purse['candy'] + 2
>>> print(purse)
{'money': 12, 'tissues': 75, 'candy': 5}
```

Comparing Lists and Dictionaries

- Dictionaries are like Lists except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print(lst)
[21, 183]
>>> lst[0] = 23
>>> print(lst)
[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print(ddd
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print(ddd
{'course': 182, 'age': 23}
```

Dictionary Literals (Constants)

- Dictionary literals use curly braces and have a list of key : value pairs
- You can make an empty dictionary using empty curly braces

```
>>> j = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print(j)
{'jan': 100, 'chuck': 1, 'fred': 42}
>>> o = { }
>>> print(o)
{}
```

Dictionary Tracebacks

- It is an error to reference a key which is not in the dictionary
- We can use the in operator to see if a key is in the dictionary

```
>>> c = dict()
>>> print(c['csev'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'csev'
>>> print('csev' in c)
False
```

When we see a new name

- When we encounter a new name, we need to add a new entry in the dictionary and if this the second or later time we have seen the name, we simply add one to the count in the dictionary under that name

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names :
    if name not in counts:
        counts[name] = 1
    else :
        counts[name] = counts[name] + 1
print(counts)

{'csev': 2, 'zqian': 1, 'cwen': 2}
```

The get method for dictionary

- This pattern of checking to see if a key is already in a dictionary and assuming a default value if the key is not there is so common, that there is a method called get() that does this for us
- Default value if key does not exist (and no Traceback).

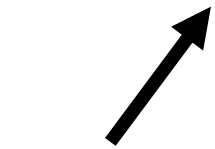
```
if name in counts:
    print(counts[name])
else:
    print(0)
print(counts.get(name, 0))
```

```
{'csev': 2, 'zqian': 1, 'cwen': 2}
```

Simplified counting with get()

- We can use get() and provide a default value of zero when the key is not yet in the dictionary - and then just add one

```
counts = dict()
names = ['csev', 'cwen', 'csev', 'zqian', 'cwen']
for name in names :
      counts[name] = counts.get(name, 0) + 1
print(counts)
```




```
{'csev': 2, 'zqian': 1, 'cwen': 2}
```

Counting Pattern

- The general pattern to count the words in a line of text is to split the line into words, then loop through the words and use a dictionary to track the count of each word independently.

```
counts = dict()
print('Enter a line of text:')
line = input('')
words = line.split()
print('Words:', words)
print('Counting...')
for word in words:
    counts[word] = counts.get(word,0) + 1
print('Counts', counts)
```

Counting Words

```
python wordcount.py
Enter a line of text:
the clown ran after the car and the car ran into the tent
 and the tent fell down on the clown and the car
Words: ['the', 'clown', 'ran', 'after', 'the', 'car', 'an
d', 'the', 'car', 'ran', 'into', 'the', 'tent', 'and', 't
he', 'tent', 'fell', 'down', 'on', 'the', 'clown', 'and',
 'the', 'car']

Counting...
Counts {'and': 3, 'on': 1, 'ran': 2, 'car': 3, 'into': 1,
 'after': 1, 'clown': 2, 'down': 1, 'fell': 1, 'the': 7,
'tent': 2}
```

Definite Loops and Dictionaries

- Even though dictionaries are not stored in order, we can write a for loop that goes through all the entries in a dictionary - actually it goes through all of the keys in the dictionary and looks up the values

```
>>> counts = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for key in counts:
    print(key, counts[key])
jan 100
chuck 1
fred 42
>>>
```

Retrieving lists of Keys and Values

- You can get a list of keys, values or items (both) from a dictionary

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> print(list(jjj))
['jan', 'chuck', 'fred']
>>> print(jjj.keys())
['jan', 'chuck', 'fred']
>>> print(jjj.values())
[100, 1, 42]
>>> print(jjj.items())
[('jan', 100), ('chuck', 1), ('fred', 42)]
>>>
```

Bonus: Two Iteration Variables!

- We loop through the key-value pairs in a dictionary using *two* iteration variables
- Each iteration, the first variable is the key and the the second variable is the corresponding value for the key

```
>>> jjj = { 'chuck' : 1 , 'fred' : 42, 'jan': 100}
>>> for aaa,bbb in jjj.items():
        print(aaa, bbb)

jan 100
chuck 1
fred 42
>>>
```

Dictionary comprehensions

- Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.

```
# Dictionary Comprehension
squares = {x: x*x for x in range(6)}

print(squares)
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Dictionary Comprehension with if conditional
odd_squares = {x: x*x for x in range(11) if x % 2 == 1}

print(odd_squares)
# {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

Memos

Memoized recursion

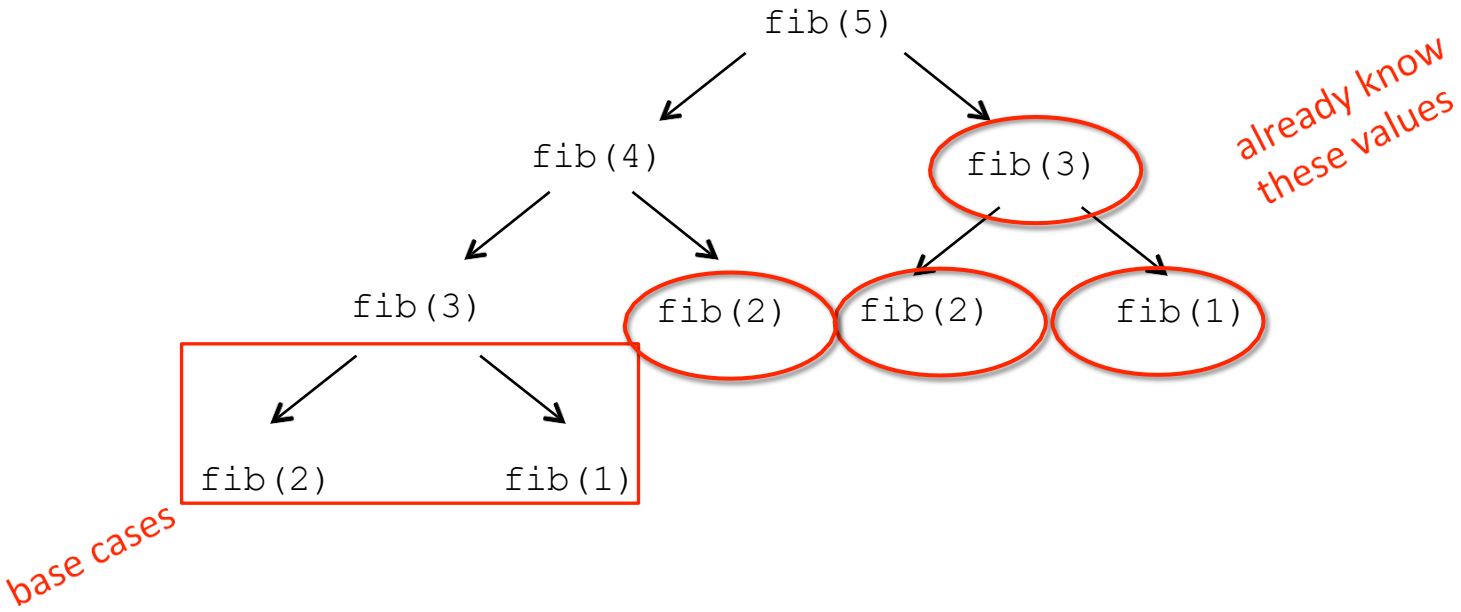
FIBONACCI RECURSIVE CODE

```
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 times!
- could keep **track** of already calculated values

FIBONACCI WITH A DICTIONARY

```
def fib_fast(n, d):
    if n in d:
        return d[n]
    else:
        ans = fib_fast(n-1, d) + fib_fast(n-2, d)
        d[n] = ans
    return  ans

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

Method sometimes called " memoization"

Initialize dictionary with base cases

- do a lookup first in case already calculated the value
- modify dictionary as progress through function calls

EFFICIENCY GAINS

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

Longest Increasing Subsequence (LIS)

- The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.
- For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 and LIS is {10, 22, 33, 50, 60, 80}.
- Let a[0..n-1] be the input sequence and L(i) be the length of the LIS ending at index i such that a[i] is the last element of the LIS.
- Then, L(i) can be recursively written as:

```
L(i) = 1 + max(L(j)) where 0 < j < i and a[j] < a[i];
L(i) = 1, if no such j exists.
```

Longest Increasing Subsequence (LIS)

```
def lis(arr, i, d):
    if i in d:
        return d[i]
    else:
        res = 1
        for j in range(0, i):
            if arr[i] > arr[j]:
                res = max(res, 1 + lis(arr, j, d))
        d[i] = res
        return res
```

Longest Increasing Subsequence (LIS)

```
def lis_ans(arr):
    ans = 0
    d = {0:1}
    for i in range(len(arr)):
        ans = max(ans, lis(arr, i, d))
    return ans


print(lis_ans([10, 22, 9, 33, 21, 50, 90, 80]))
print(lis_ans([3, 10, 2, 1, 20]))
print(lis_ans([50, 3, 10, 7, 40, 80]))
```

Tuples

Tuples are like lists

- Tuples are another kind of sequence that function much like a list - they have elements which are indexed starting at 0

```
>>> x = ('Glenn', 'Sally', 'Joseph')
>>> print(x[2])
Joseph
>>> y = (1, 9, 2)
>>> print(y)
(1, 9, 2)
>>> print(max(y))
9
>>> for iter in y:
        print(iter, end = ' ')
1 9 2
```

..but.. Tuples are "immutable"

- Unlike a list, once you create a tuple, you cannot alter its contents - similar to a string

```
>>> x = [9, 8, 7]
>>> x[2] = 6
>>> print(x)
[9, 8, 6]
>>>
```

```
>>> y = 'ABC'
>>> y[2] ='D'
Traceback:'str' object does
 not support
item assignment
>>>
```

```
>>> z = (5, 4, 3)
>>> z[2] = 0
Traceback:'tuple'
object does
not support item
assignment
>>>
```

Things not to do with tuples

```
>>> x = (3, 2, 1)
>>> x.sort()
Traceback:AttributeError:
 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback:AttributeError:
 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback:AttributeError:
 'tuple' object has no attribute 'reverse'
>>>
```

A Tale of Two Sequences

```
>>> l = list()
>>> dir(l)
['append', 'count', 'extend', 'index', 'in
sert', 'pop', 'remove', 'reverse', 'sort']

>>> t = tuple()
>>> dir(t)
['count', 'index']
```

Tuples are more efficient

- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- So in our program when we are making "temporary variables", we prefer tuples over lists.

Tuples and Assignment

- We can also put a tuple on the left hand side of an assignment statement
- We can even omit the parenthesis

```
>>> (x, y) = (4, 'fred')
>>> print(y)
fred
>>> (a, b) = (99, 98)
>>> print(a)
99
```

Tuples as return values

- A function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values
- For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x//y and then x%y. It is better to compute them both at the same time.
- The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder.

```
>>> t = divmod(7, 3)
>>> t
(2, 1)

>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

Tuples as return values

- Here is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

- max and min are built-in functions that find the largest and smallest elements of a sequence.
- min_max computes both and returns a tuple of two values.

Lists and tuples

- **zip** is a built-in function that takes two or more sequences and interleaves them:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
<zip object at 0x7f7d0a9e7c48>
```

- The result is a **zip object** that knows how to iterate through the pairs. The most common use of zip is in a for loop:

```
>>> for pair in zip(s, t):
        print(pair)

('a', 0)
('b', 1)
('c', 2)
```

Lists and tuples

- If you want to use list operators and methods, you can use a zip object to make a list:

```
>>> list(zip(s, t))
[('a', 0), ('b', 1), ('c', 2)]
```

- If the sequences are not the same length, the result has the length of the shorter one:

```
>>> list(zip('Anne', 'Elk'))
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

- You can use tuple assignment in a for loop to traverse a list of tuples:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

Lists and tuples

- If you need to traverse the elements of a sequence and their indices, you can use the built-in function enumerate:
- The result from enumerate is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence.

```
for index, element in enumerate('abc'):
    print(index, element)
```

```
0 a
1 b
2 c
```

Dictionaries and tuples

- Dictionaries have a method called items that returns a sequence of tuples, where each tuple is a key-value pair.

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])

>>> for key, value in d.items():
        print(key, value)

c 2
a 0
b 1
```

- As you should expect from a dictionary, the items are in no particular order.

Dictionaries and tuples

- You can use a list of tuples to initialize a new dictionary:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

- Combining dict with zip yields a concise way to create a dictionary:

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

Tuples are Comparable

- The comparison operators work with tuples and other sequences if the first item is equal, Python goes on to the next element, and so on, until it finds elements that differ.

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
>>> ( 'Jones', 'Sally' ) < ('Jones', 'Fred')
False
>>> ( 'Jones', 'Sally') > ('Adams', 'Sam')
True
```

Sorting Lists of Tuples

- We can take advantage of the ability to sort a list of tuples to get a sorted version of a dictionary
- First we sort the dictionary by the key using the items() method

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = d.items()
print(t)
[('a', 10), ('c', 22), ('b', 1)]
>>> t.sort()
print(t)
[('a', 10), ('b', 1), ('c', 22)]
```

Using sorted()

- We can do this even more directly using the built-in function sorted that takes a sequence as a parameter and returns a sorted sequence

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> d.items()
[('a', 10), ('c', 22), ('b', 1)]

>>> t = sorted(d.items())
print(t)
[('a', 10), ('b', 1), ('c', 22)]

>>> for k, v in sorted(d.items()):
        print(k, v,)

 a 10 b 1 c 22
```

Sort by values instead of key

- If we could construct a list of tuples of the form (value, key) we could sort by value
- We do this with a for loop that creates a list of tuples

```
>>> c = {'a':10, 'b':1, 'c':22}
>>> tmp = list()
>>> for k, v in c.items() :
        tmp.append( (v, k) )
>>> print(tmp)
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> tmp.sort(reverse=True)
>>> print(tmp)
[(22, 'c'), (10, 'a'), (1, 'b')]
```

```
fhand = open('romeo.txt')
counts = dict()
for line in fhand:
    words = line.split()
        for word in words:
            counts[word] = counts.get(word, 0 ) + 1

lst = list()
for key, val in counts.items():
    lst.append( (val, key) )
lst.sort(reverse=True)
for val, key in lst[:10] :
    print(key, val)
```

The top 10 most common words.

Even Shorter Version (adv)

- List comprehension creates a dynamic list. In this case, we make a list of reversed tuples and then sort it.

```
>>> c = {'a':10, 'b':1, 'c':22}

>>> print(sorted([ (v,k) for k,v in c.items()]))

[(1, 'b'), (10, 'a'), (22, 'c')]
```

References

1. MIT Introduction to Computer Science and Programming in Python
2. Think Python: How to Think Like a Computer Scientist: https://greenteapress.com/thinkpython2/html/index.html

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!

soict.hust.edu.vn/ fb.com/groups/soict