



香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Introduction to Computer Science: Programming Methodology

Lecture 5 List

Prof. Junhua Zhao

School of Science and Engineering

List is kind of a collection

- A collection allows us to put **many values** in a **single “variable”**
- A collection is nice because we can carry all many variables around in one convenient package



What is not a collection

- Most of our variables have only **one value** in them – when we put a new value in the variable, the old value will be **over-written**

```
>>> x=2  
>>> 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([])
[]
```

List and definite loop – best pal

```
friends = ['Tom', 'Jerry', 'Bat']
for friend in friends:
    print('Happy new year', friend)
print('Done')
```

```
Happy new year Tom  
Happy new year Jerry  
Happy new year Bat  
Done
```

Looking inside lists

- Just like strings, we can access any **single element** in a list using an **index** specified in square bracket

Joseph	Glenn	Sally
0	1	2

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

Lists are mutable

- Strings are “immutable” – we **cannot** change the contents of a string unless we make a **new string**
- Lists are “mutable” – we can change an element of a list using **index** operator

```
>>> fruit = 'Banana'
>>> fruit[0] = 'b'
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fruit[0] = 'b'
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 input and returns the **number of elements** in that list
- Actually `len()` tells us the number of elements in **any sequence** (e.g. strings)

```
>>> greet = 'Hello Bob'  
>>> print(len(greet))  
9  
>>> x=[1, 2, 'joe', 99]  
>>> print(len(x))  
4
```

Range() function

- The `range()` function returns a **list** of numbers
- We can construct an **index loop** using `for` and an integer iterator

```
>>> x=range(4)
>>> x
range(0, 4)
>>> x[0]
0
>>> x[1]
1
>>> x[2]
2
>>> x[3]
3
>>> x=range(2, 10, 2)
>>> x[0]
2
>>> x[3]
8
>>> x[4]
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    x[4]
IndexError: range object index out of range
```

2 3 4 6 7 8 9 · 0

A tale of two loops

Example

```
friends = ['Tom', 'Jerry', 'Bat']

for friend in friends:
    print('Happy new year, ', friend)

for i in range(len(friends)):
    friend = friends[i]
    print('Happy new year, ', friend)
```

Output

```
Happy new year, Tom
Happy new year, Jerry
Happy new year, Bat
Happy new year, Tom
Happy new year, Jerry
Happy new year, Bat
>>> |
```

Concatenating lists using +

- Similar to strings, we can **add** two existing lists together to create a **new list**

```
>>> 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: similar to strings, the second number is “**up to but no 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]
```

List methods

```
>>> x=list()
>>> type(x)
<class 'list'>
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

<https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

Building a list from scratch

- We can **create** an empty list using `list()`, and then **add elements** using `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** to check whether 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
```

A list is an ordered sequence

- A list can hold many items and keeps them **in the order** until we do something to change the order
- A list can be **sorted** (i.e. change the order)
- The **sort()** method means “sort yourself”

```
>>> friend = ['Tom', 'Jerry', 'Bat']
>>> friends.sort()
>>> print(friends)
['Bat', 'Jerry', 'Tom']
>>> print(friends[1])
Jerry
>>>
>>> numbers = [1, 2, 5, 100, 32, 7, 97, 1001]
>>> numbers.sort()
>>> print(numbers)
[1, 2, 5, 7, 32, 97, 100, 1001]
```

Built-in functions and lists

- There are a number of **functions** built into Python that take lists **as inputs**

- Remember the loops we built? These are much simpler

```
>>> numbers = [3, 41, 12, 9, 74, 15]
>>> print(len(numbers))
6
>>> print(max(numbers))
74
>>> print(min(numbers))
3
>>> print(sum(numbers))
154
>>> print(sum(numbers)/len(numbers))
25.666666666666668
```

Averaging with a list

```
total = 0
count = 0
while True:
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1

average = total/count
print('The average is:', average)
```

Practice

- Write a program to instruct the user to input several numbers and calculate their average using list methods

Answer

```
numlist = list()
while True:
    inp = input('Enter a number:')
    if inp == 'done': break
    value = float(inp)
    numlist.append(value)

average = sum(numlist)/len(numlist)
print('The average is:', average)
```

Best friends: strings and lists

- Use the `split()` method to break up a string into **a list of strings**
- We think of these as **words**
- We can access a particular word or loop through all the words

```
>>> myStr = 'Catch me if you can'  
>>> words = myStr.split()  
>>> print(words)  
['Catch', 'me', 'if', 'you', 'can']  
>>> print(len(words))  
5  
>>> print(words[0])  
Catch  
  
>>> for w in words: print(w)  
  
Catch  
me  
if  
you  
can
```

- When you do not specify a delimiter, multiple spaces are treated like “one” delimiter
- You can specify what delimiter character to use in splitting

```
>>> 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']  
>>> len(thing)  
1  
>>>  
>>> thing = line.split(';')  
>>> print(thing)  
['first', 'second', 'third']  
>>> print(len(thing))  
3
```

Practice

- The header of an email takes the following format:

From professor.xman@uct.edu Sat Jan 5 09:14:16 2008

For a given email header, write a program to find out the domain of email address, and the month in which this email is sent

Answer

```
header = 'From professor.xman@uct.edu Sat Jan 5 09:14:16 2008'
words = header.split()
address = words[1].split('@')

print('The email domain is:', address[1])
print('The month when the email is sent:', words[3])
```

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



Dictionary



https://en.wikipedia.org/wiki/Associative_array

Dictionary

- Dictionaries are Python's most powerful data collection
- Dictionaries allow us to do fast **database-like operations** in Python
- Dictionaries have different names in different languages
 - **Associative arrays** – Perl/PHP
 - **Properties or Map or HashMap** – Java
 - **Property Bag** – C#/.Net

Dictionary

Key: can be anything
recommended: string

- Lists **index** their entries based on the position in the list
- **Dictionaries** are like bags – no order
- We **index** the elements 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}  
  
>>> purse[3] = 77  
>>> print(purse)  
{3: 77, 'money': 12, 'tissues': 75, 'candy': 5}
```

Dictionary

```
>>> 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}
```



List v.s. dictionary

- Dictionaries are similar to **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)  
{'age': 21, 'course': 182}  
>>> ddd['age']=23  
>>> print(ddd)  
{'age': 23, 'course': 182}
```

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

List	
Key	Value
[0]	21
[1]	183

Dictionary	
Key	Value
[course]	183
[age]	21

Dictionary literals (constants)

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

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

Most common names

A 4x4 grid of names on a black background. The names are: zhen, zhen, marquard, cwen; csev, marquard, marquard, zhen; zhen, csev, cwen, csev; and cwen, zhen, zhen, zhen.

zhen	zhen	marquard	cwen
csev			csev
marquard	marquard	csev	cwen
		csev	cwen

Counting with a dictionary

- A common use of dictionary is counting how often we “see” something

```
>>> ccc=dict()  
>>> ccc['csev']=1  
>>> ccc['owen']=1  
>>> print(ccc)  
{'csev': 1, 'owen': 1}  
>>> ccc['owen']=ccc['owen']+1  
>>> print(ccc['owen'])  
2
```

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

```
>>> ccc=dict()
>>> print(ccc['csev'])
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    print(ccc['csev'])
KeyError: 'csev'
>>> 'csev' in ccc
False
```

Practice

- Write a program to instruct the user to continuously input some words, and use dictionary to count how many times a word has been inputted before.

Answer

```
wordDict = dict()
while True:
    word = input('Enter a word:')
    if word in wordDict:
        wordDict[word]=wordDict[word]+1
    elif word=='done':
        break
    else:
        wordDict[word]=1
print('The result of word count:')
print(wordDict)
```

The get() method

- 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

```
>>> counts = {'aaa':1, 'bbb':2, 'ccc':5}
>>> print(counts.get('eee', 0))
0
```

Practice

- Write a program to instruct the user to input a line of texts, and use dictionary to count how many times a word has been seen in this line. You should use the `get()` method in this program.

Answer

```
wordDict = dict()

textLine = input('Enter a line:')
words = textLine.split()

print('Counting...')
for word in words:
    wordDict[word] = wordDict.get(word, 0) + 1

print('The result of word count:')
print(wordDict)
```

Definite loops and dictionaries

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

```
counts = {'chuck':1, 'fred':42, 'jan':100}
```

```
for key in counts:  
    print(key, counts[key])
```

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

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', 'fred', 'chuck']

>>> print(list(jjj.keys()))
['jan', 'fred', 'chuck']

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

ving lists of

et a list of
es or items
n a

```
[{'a': 100, 'b': 200, 'c': 300}
>>> purse.get('a', True)
100
>>> jjj >>> purse.get('b', True)
200
>>> prin >>> purse.get('ddd', True)
['jan', True
      >>> purse.get('ddd', -1)
>>> prin -1
['jan', >>> purse
      {'a': 100, 'b': 200, 'c': 300}
>>> a=purse.keys()
>>> prin >>> type(a)
[100, 42 <class 'dict_keys'>
>>> prin >>> list(a)
[('jan', ['a', 'b', 'c']
>>>
```

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 second variable is the **corresponding value** for the key

```
counts = {'chuck':1, 'fred':42, 'jan':100}
for key, value in counts.items():
    print(key, value)
```

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

Tuples

- Tuples are another type of sequence that function more like a list – they have elements which are indexed starting from 0

But, tuples are “immutable”

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

```
>>> x=[9, 8, 7] >>> y='abc'  
>>> x[2]=6      >>> y[2]='e'  
>>> print(x)   Traceback (most recent call last)  
[9, 8, 6]          :  
                  File "<pyshell#23>", line 1, in  
<module>  
                  y[2]='e'  
TypeError: 'str' object does not  
support item assignment  
...
```

```
>>> z=(5, 4, 3)  
>>> z[2]  
3  
>>> z[2]=0  
Traceback (most recent call last)  
:  
      File "<pyshell#28>", line 1, in  
<module>  
      z[2]=0  
TypeError: 'tuple' object does no  
t support item assignment
```

Some things that you cannot do with tuples

```
>>> x=(1, 2, 3)
>>> x.sort()
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    x.sort()
AttributeError: 'tuple' object has no attribute 'sort'
>>> x.append(5)
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    x.append(5)
AttributeError: 'tuple' object has no attribute 'append'
>>> x.reverse()
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    x.reverse()
AttributeError: 'tuple' object has no attribute 'reverse'
```

A tale of two sequences

```
>>> l = list()
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
 '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
 '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
>>> t = tuple()
>>> dir(t)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
 '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '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
- In our program when we are making “temporary variables” we prefer tuples over lists

Tuples and dictionaries

- The `item()` method in dictionaries returns a list of **(key, value)** tuples

```
>>> d=dict()
>>> d['csev']=2
>>> d['cwen']=4
>>> for (k, v) in d.items():
    print(k, v)

csev 2
cwen 4
>>> tups = d.items()
>>> print(tups)
dict_items([('csev', 2), ('cwen', 4)])
>>> print(list(tups))
[('csev', 2), ('cwen', 4)]

>>> tups = list(tups)
>>> tups[1]
('cwen', 4)
```

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, until it finds the elements which are different

```
>>> (0, 1, 2) < (5, 1, 2)
True
>>> (0, 1, 200000) < (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()
>>> t=list(t)
>>> t
[('c', 22), ('b', 1), ('a', 10)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

Using sorted()

- We can do this even more efficiently using a built-in function `sorted()` which takes a sequence as a parameter and returns a sorted sequence

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



```
>>> for k, v in t:
    print(k, v)
```



```
a 10
b 1
c 22
```

Practice

- Write a program, which sorts the elements of a dictionary by the value of each element

Sort by values instead of key

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

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

Example: Finding the 10 most common words in a file

```
fhand = open('myhost.txt', 'r')
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)
```