



# 第4课 数据结构2



# Python序列概述

- 列表
- 元组
- 字典
- 集合
- 高级数据结构

# 字典 dictionary

Mutable map from hashable values to (arbitrary) objects

- 字典 (Dictionaries)，属于映射类型(key-value pairs)，它是通过键(key) 实现元素 (value) 存取，具有无序、可变长度、异构、嵌套和可变类型容器等特点。
- 字典中的键可以为任意不可变数据，比如整数、实数、复数、字符串、元组等等。
- 什么时候使用dictionary?
  - 当需要以key-value对的形式创建关联时。
  - 当需要基于自定义键快速查找数据时。
  - 当需要修改或添加到key-value对时。

# 创建字典

- 小括号（元组），中括号（列表），大括号（字典）
- 使用空大括号或dict()构造函数创建空字典
- 通过指定每个key:value对来初始化字典

使用 = 将一个字典赋值给一个变量

```
>>> d1 = {}          #空字典
>>> d2 = {"Name": "Susan", "Age": 19, "Major": "CS"}
#{'Major': 'CS', 'Age': 19, 'Name': 'Susan'}
```

使用dict利用已有数据创建字典:

```
>>> d3 = dict()      #空字典
>>> d4 = dict(zip(['Name', 'Age', 'Major'], ["Susan", 19, "CS"]))
#{'Major': 'CS', 'Age': 19, 'Name': 'Susan'}
```

# 创建字典

使用dict根据给定的键、值创建字典

```
>>> d5 = dict(Name="Susan", Age=19, Major="CS")  
#{'Major': 'CS', 'Age': 19, 'Name': 'Susan'}
```

```
>>> d6 = dict([('Age', 19), ('Name', "Susan"), ('Major', "CS")])  
#{'Major': 'CS', 'Age': 19, 'Name': 'Susan'}
```

以给定内容为键，创建值为空的字典

```
>>> adict = dict.fromkeys(['name', 'age', 'sex'])  
#{'age': None, 'name': None, 'sex': None}
```

可以使用del删除整个字典

# 访问字典

- 要访问字典，只需按key对字典进行索引即可获得value。
- 如果key不在字典中，将引发异常。

```
a = {"one": 1, "two": 2, "three": 3}
b = dict(one=1, two=2, three=3)
c = dict([('one', 1), ('two', 2), ('three', 3)])
```

```
a == b == c # => True
```

```
c[one]      #NameError: name 'one' is not defined
```

```
c['one']    # => 1
```

```
c.get('one') # => 1; 使用get()方法
```

# 访问字典

- 查看字典的结构
- 使用字典对象的`items()`方法可以返回字典的键、值对列表
- 使用字典对象的`keys()`方法可以返回字典的键列表
- 使用字典对象的`values()`方法可以返回字典的值列表

```
d = {"one": 1, "two": 2, "three": 3}
d.keys() # => dict_keys(['one', 'two', 'three'])
d.values() # => dict_values([1, 2, 3])
d.items()
# => dict_items([('one', 1), ('two', 2), ('three', 3)])

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

# 访问字典

- 注意：字典的索引方式和列表、元组不一样！

```
squares = {1: 1, 2: 4, 3: 9, 4: 16}
```

```
squares[3] + squares[3]    #18
```

```
squares[3 + 3]  
#KeyError
```

```
squares[2] + squares[2]    #8
```

```
squares[2 + 2]             #16
```



# 字典元素的添加与修改

- 当以指定key为下标为字典赋值时：
  - ✓ 若key存在，则可以修改该key的value；
  - ✓ 若不存在，则表示添加一个key: value对（不报错！）。

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
```

```
>>> d1['Age'] = 21
```

```
>>> d1['Year'] = "Junior"
```

```
>>> d1
```

```
{ 'Age': 21, 'Name': 'Susan', 'Major': 'CS',  
  'Year': 'Junior' }
```

# 字典元素的添加与修改

- 使用字典对象的update方法
  - ✓ 将另一个字典的键、值对添加到当前字典对象
  - ✓ 类似列表的extend方法

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS"}
```

```
>>> d1
```

```
{'Age': 19, 'Name': 'Susan', 'Major': 'CS'}
```

```
>>> d1.update({'Year':'Junior','Height':160})
```

```
>>> d1
```

```
{'Major': 'CS', 'Age': 19, 'Year': 'Junior', 'Name':  
  'Susan', 'Height': 160}
```

# 字典元素的添加与修改

- 使用del删除字典中指定键的元素
- 使用字典对象的pop()方法删除并返回指定key的元素
- 使用字典对象的clear()方法来删除字典中所有元素

```
>>> d1 = {'Age':19, 'Name':"Susan", 'Major':"CS",'height':160}
```

```
>>> del d1['height']
```

```
>>> d1
```

```
{'Age': 19, 'Name': 'Susan', 'Major': 'CS'}
```

```
>>> d1.pop('Age')
```

```
19
```

```
>>> d1
```

```
{'Name': 'Susan'}
```

# 字典推导式

*# Dictionary comprehensions*

```
{key_fun(x):val_fun(x) for x in iterable}
```

```
>>> {i:str(i) for i in range(1, 5)}  
{1: '1', 2: '2', 3: '3', 4: '4'}
```

```
>>> x = ['A', 'B', 'C', 'D']  
>>> y = ['a', 'b', 'b', 'd']
```

```
>>> {i:j for i,j in zip(x,y)}  
{ 'A': 'a', 'C': 'b', 'B': 'b', 'D': 'd' }
```

# 字典推导式

# 练习

```
fav_animals = {'tom': 'cat', 'michael': 'elephant',  
'zheng': 'dog', 'sam': 'cow', 'nick': 'goat'}
```

```
fav_humans = {val:key for key, val in fav_animals.items() }
```

```
>>> fav_humans
```

输出什么?

# 字典推导式

# 练习


```
fav_animals = {'tom': 'cat', 'michael': 'elephant',  
               'zheng': 'dog', 'sam': 'cow', 'nick': 'goat'}
```

```
fav_humans = {val:key for key, val in fav_animals.items() }
```

```
>>> fav_humans
```

```
{'cat': 'tom', 'elephant': 'michael', 'dog': 'zheng',  
 'cow': 'sam', 'goat': 'nick'}
```

Key-Value Pairs → Value-Key Pairs



# 字典的应用

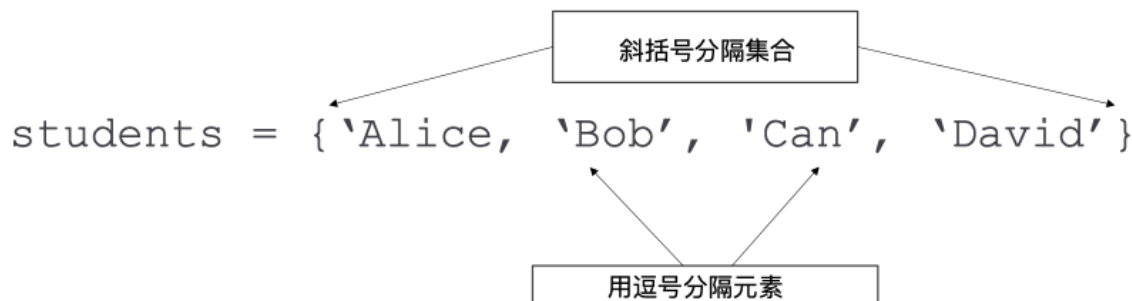
**实验：统计文本里单词出现的频率**



# 集合 set

Unordered, finite collection of distinct, hashable elements.

- 集合，可以理解为数学课里学习的集合。
- 集合中只能包含数字、字符串、元组等不可变类型（或者说可散列）的数据，而不能包含列表、字典、集合等可变类型的数据。
- 什么时候使用set？
  - 当元素必须是**唯一**的
  - 当需要数学集合运算的支持时





# 创建集合

- 使用set构造函数或{}符号来初始化一个集合
- 不要使用空的大括号{ }来创建空的集合；得到的是一个空的字典

```
>>> a = {3, 5}
>>> a.add(7)
>>> a
{3, 5, 7}
```

# a = {3:5}是什么?  
#向集合中添加元素

- 使用set构造函数创建空集。

```
myset = set(sequence)
myset2 = {expression for variable in sequence}
```

```
empty_set = set() #空集合
set_from_list = set([1, 2, 1, 4, 3]) # => {1, 2, 3, 4}
```

```
basket = {'orange', 'banana', 'pear', 'apple'}
```

# 创建集合

- 集合中只能包含数字、字符串、元组等不可变类型（或者说可散列）的数据，而不能包含列表、字典、集合等可变类型的数据。

```
>>> a=[1,2]
>>> b=[3,4]
>>> c={a,b}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

注意：集合是无序的，无法通过数字进行索引

```
>>> a={3,5}
>>> a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

# 集合的创建与删除

- 使用del命令删除整个集合
- pop()方法弹出并删除一个元素
- remove()方法直接删除指定元素
- clear()方法清空集合

```
>>> myset = {x for x in 'abracadabra'}  
>>> myset  
{ 'b', 'a', 'c', 'd', 'r' }
```

```
>>> myset.add('y')  
>>> myset  
{ 'b', 'y', 'a', 'c', 'd', 'r' }
```

```
>>> myset.remove('a')  
>>> myset  
{ 'b', 'y', 'c', 'd', 'r' }
```

```
>>> myset.pop()  
'b'  
>>> myset  
{ 'y', 'c', 'd', 'r' }
```

# 集合操作

- 集合支持交集、并集、差集等运算

```
a = set('abracadabra') # {'a', 'r', 'b', 'c', 'd'}  
b = set('alacazam') # {'a', 'm', 'c', 'l', 'z'}
```

```
# Set union (并集)
```

```
a | b # => {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

```
# Set intersection (交集)
```

```
a & b # => {'a', 'c'}
```

```
# Set difference (在集合a但不在集合b的元素)
```

```
a - b # => {'r', 'd', 'b'}
```

```
# Symmetric difference (在集合a或b但不同时在两个集合的元素)
```

```
a ^ b # => {'r', 'd', 'b', 'm', 'z', 'l'}
```

# 集合操作

- 集合支持交集、并集、差集等运算

```
a = set('abracadabra') # {'a', 'r', 'b', 'c', 'd'}  
b = set('alacazam') # {'a', 'm', 'c', 'l', 'z'}
```

```
# Set union (并集)
```

```
a.union(b) # => {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

```
# Set intersection (交集)
```

```
a.intersection(b) # => {'a', 'c'}
```

```
# Set difference (在集合a但不在集合b的元素)
```

```
a.difference(b) # => {'r', 'd', 'b'}
```

```
# Symmetric difference (在集合a或b但不同时在两个集合的元素)
```

```
a.symmetric_difference(b) # => {'r', 'd', 'b', 'm', 'z', 'l'}
```

# 集合操作

- 集合包含关系测试
- 比较运算符 $\geq$ ,  $\leq$ 测试一个集是另一个集的超集还是子集。 $>$ 和 $<$ 运算符检查是否有真超集/真子集。

```
>>> s1 = set('abracadabra')  
>>> s1  
{'b', 'a', 'c', 'd', 'r'}
```

```
>>> s2 = set('bard')  
>>> s2  
{'b', 'd', 'a', 'r'}
```

```
>>> s1 >= s2
```

```
True
```

```
>>> s1 > s2
```

```
True
```

```
>>> s1 <= s2
```

```
False
```

```
>>> s2.issubset(s1)
```

```
True
```

# 集合推导式

```
# Set comprehensions  
{fun(x) for x in iterable}
```

练习:

```
>>> s = {len(x) for x in ('he', 'she', 'I')}
```

```
>>> s
```

输出什么?

# 集合推导式

```
# Set comprehensions  
{fun(x) for x in iterable}
```

练习:

```
>>> s = {len(x) for x in ('he', 'she', 'I')}
```

```
>>> s
```

```
{1, 2, 3}
```



# 集合的优势

检查一个单词是否有特定字母组成？

不用集合.....

```
EFFICIENT_LETTERS = 'BCDGIJLMNOPSUVWZ'

def is_efficient(word):
    for letter in word:
        if letter not in EFFICIENT_LETTERS:
            return False
    return True
```

# 集合的优势

检查一个单词是否由特定字母组成？

不用集合.....

```
EFFICIENT_LETTERS = 'BCDGIJLMNOPSUVWZ'

def is_efficient(word):
    for letter in word:
        if letter not in EFFICIENT_LETTERS:
            return False
    return True
```

使用集合!!!

```
EFFICIENT_LETTERS = set('BCDGIJLMNOPSUVWZ')

def is_efficient(word):
    return set(word) <= EFFICIENT_LETTERS
```

# 固定集合 - frozenset

- 创建固定集合 - 使用构造函数(**frozenset**)  
**frozenset(iterable)**: 使用可迭代对象创建固定集合
- 拥有集合的全部方法, 但去掉能够修改集合的方法

```
>>> s=frozenset([1,3,4,5])
>>> type(s)
<class 'frozenset'>
>>> s|frozenset([2,3,4])
frozenset({1, 2, 3, 4, 5})
```

```
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'pop'
>>> s.remove(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'remove'
>>> s.add(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

# 排序

- `sorted()` 支持`key`参数实现对字典进行排序。

```
>>> persons = [{'name': 'Dong', 'age': 37}, {'name': 'Zhang', 'age': 40}, {'name': 'Li', 'age': 50},  
{'name': 'Dong', 'age': 43}]
```

```
>>> print(sorted(persons))
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

**TypeError: '<' not supported between instances of 'dict' and 'dict'**

#使用`key`参数来指定排序依据，先按姓名升序排序，姓名相同的按年龄降序排序

```
>>> print(sorted(persons, key=lambda x:(x['name'], -x['age'])))
```

```
[{'name': 'Dong', 'age': 43}, {'name': 'Dong', 'age': 37}, {'name': 'Li', 'age': 50}, {'name':  
'Zhang', 'age': 40}]
```

# Lambda 函数

- 使用关键字lambda而不是def
- 仅限于一个表达式
- 通常与函数式编程一起使用
- 以后课程学习

```
>>> def f(x):  
...     return x**2
```

```
...  
>>> print f(8)
```

64

```
>>> g = lambda x: x**2
```

```
>>> print g(8)
```

64

## 排序: itemgetter

```
>>> phonebook = {'Linda': '7750', 'Bob': '9345', 'Carol': '5834'}
```

```
>>> from operator import itemgetter
```

```
>>> sorted(phonebook.items(), key=itemgetter(1)) #按字典中元素值进行排序
```

```
[('Carol', '5834'), ('Linda', '7750'), ('Bob', '9345')]
```

```
>>> sorted(phonebook.items(), key=itemgetter(0)) #按字典中元素的键进行排序
```

```
[('Bob', '9345'), ('Carol', '5834'), ('Linda', '7750')]
```

# 排序: itemgetter

```
>>> gameresult = [['Bob', 95.0, 'A'], ['Alan', 86.0, 'C'], ['Mandy', 83.5, 'A'], ['Rob', 89.3, 'E']]
```

```
>>> sorted(gameresult, key=itemgetter(0, 1)) #按姓名升序, 姓名相同按分数升序排序
```

```
[['Alan', 86.0, 'C'], ['Bob', 95.0, 'A'], ['Mandy', 83.5, 'A'], ['Rob', 89.3, 'E']]
```

```
>>> sorted(gameresult, key=itemgetter(1, 0)) #按分数升序, 分数相同的按姓名升序排序
```

```
[['Mandy', 83.5, 'A'], ['Alan', 86.0, 'C'], ['Rob', 89.3, 'E'], ['Bob', 95.0, 'A']]
```

```
>>> sorted(gameresult, key=itemgetter(2, 0)) #按等级升序, 等级相同的按姓名升序排序
```

```
[['Bob', 95.0, 'A'], ['Mandy', 83.5, 'A'], ['Alan', 86.0, 'C'], ['Rob', 89.3, 'E']]
```

# 排序: itemgetter

## 练习

```
>>> pythonresult = [{'name':'Tom', 'exam':80, 'lab':70, 'grade':75.0},  
                      {'name':'David', 'exam':70, 'lab':80, 'grade':75.0},  
                      {'name':'Lucy', 'exam':50, 'lab':62, 'grade':56.0},  
                      {'name':'Bill', 'exam':80, 'lab':80, 'grade':80}]  
  
>>> sorted(pythonresult , key=itemgetter('grade', 'exam'))
```

输出什么?



# 排序: itemgetter

## 练习

```
>>> pythonresult = [{ 'name': 'Tom', 'exam': 80, 'lab': 70, 'grade': 75.0},  
                     { 'name': 'David', 'exam': 70, 'lab': 80, 'grade': 75.0},  
                     { 'name': 'Lucy', 'exam': 50, 'lab': 62, 'grade': 56.0},  
                     { 'name': 'Bill', 'exam': 80, 'lab': 80, 'grade': 80}]
```

```
>>> sorted(pythonresult , key=itemgetter('grade', 'exam'))
```

**#按'grade'升序，该值相同的按'exam'升序排序**

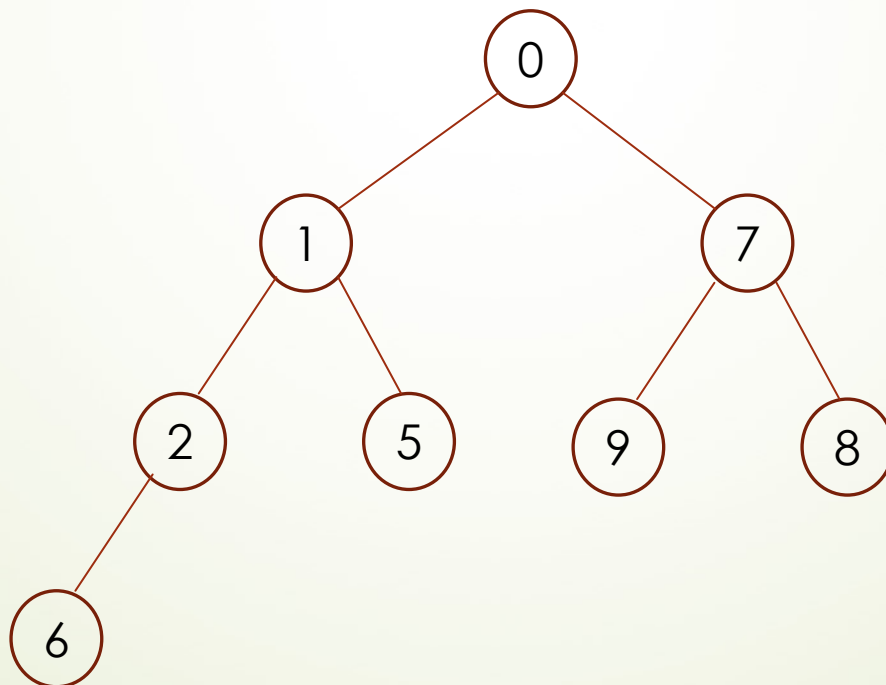
```
[{ 'name': 'Lucy', 'exam': 50, 'lab': 62, 'grade': 56.0},  
 { 'name': 'David', 'exam': 70, 'lab': 80, 'grade': 75.0},  
 { 'name': 'Tom', 'exam': 80, 'lab': 70, 'grade': 75.0},  
 { 'name': 'Bill', 'exam': 80, 'lab': 80, 'grade': 80}]
```

# 高级数据结构

- 堆、栈、队列、树、图
- 有些结构Python已经提供，而有些则需要自己利用基本数据结构来实现。

# 堆

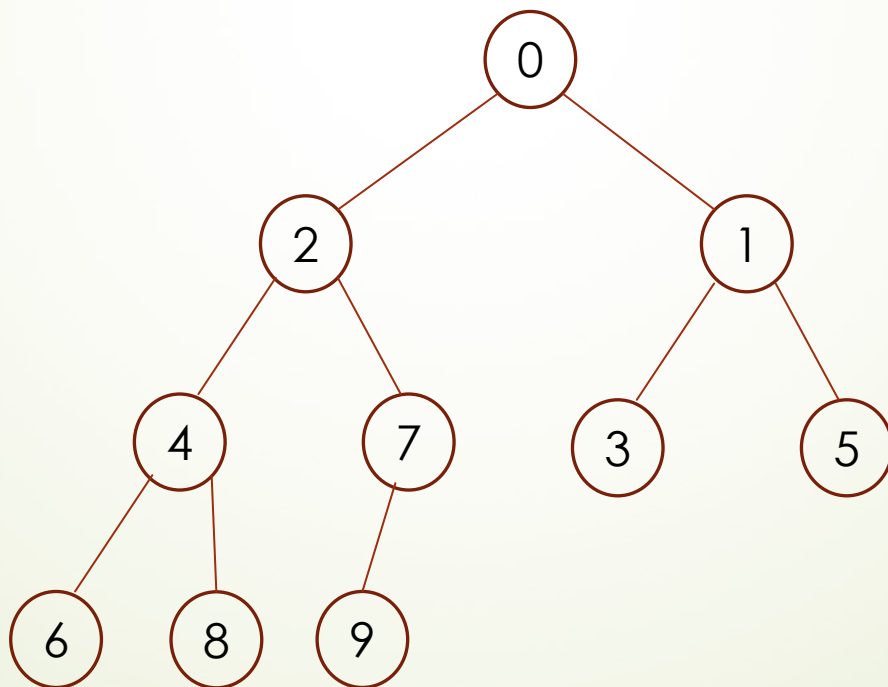
- 以最小堆为例子：
  - 父节点的值比每一个子节点的值都要小。
  - “堆属性”：对堆中的每一个节点都成立。
- 插入：首先加入到二叉树最后的一个节点，依据最小堆的定义，自底向上，递归调整。
- `import heapq`



# 堆

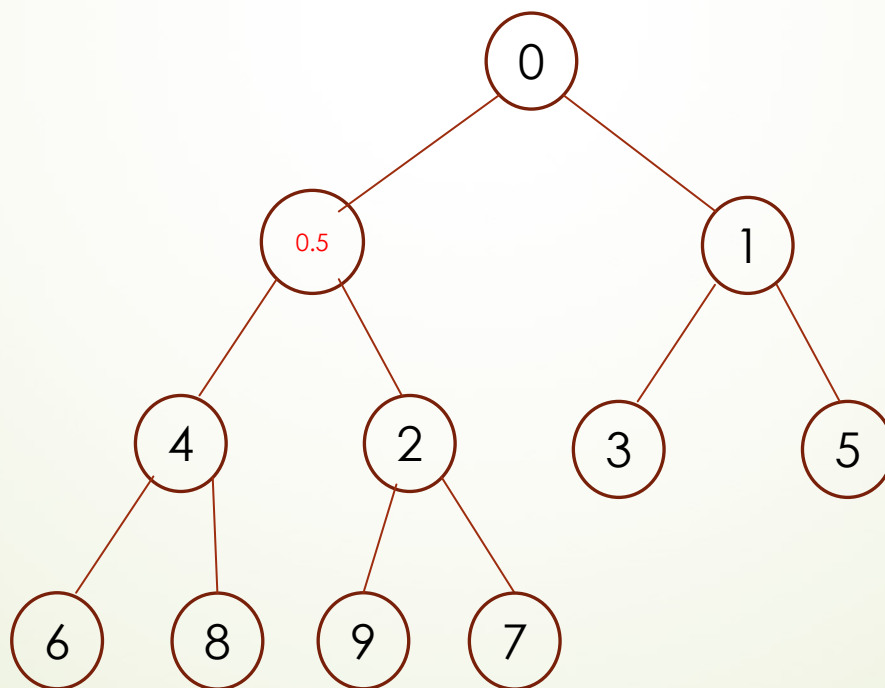
```
>>> import heapq
>>> data=[6, 1, 3, 4, 9, 0, 5, 2, 8, 7]
>>> heap=[]
>>> for n in data:
>>>     heapq.heappush(heap,n)

>>> heap
[0, 2, 1, 4, 7, 3, 5, 6, 8, 9]
```



# 堆

```
>>> heapq.heappush(heap, 0.5)           #入堆, 自动重建
>>> heap
[0, 0.5, 1, 4, 2, 3, 5, 6, 8, 9, 7]
>>> heapq.heappop(heap)                   #出堆, 自动重建
0
>>> heap      ##????
```



# 堆

```
>>> heapq.heappush(heap, 0.5)
```

#入堆，自动重建

```
>>> heap
```

```
[0, 0.5, 1, 4, 2, 3, 5, 6, 8, 9, 7]
```

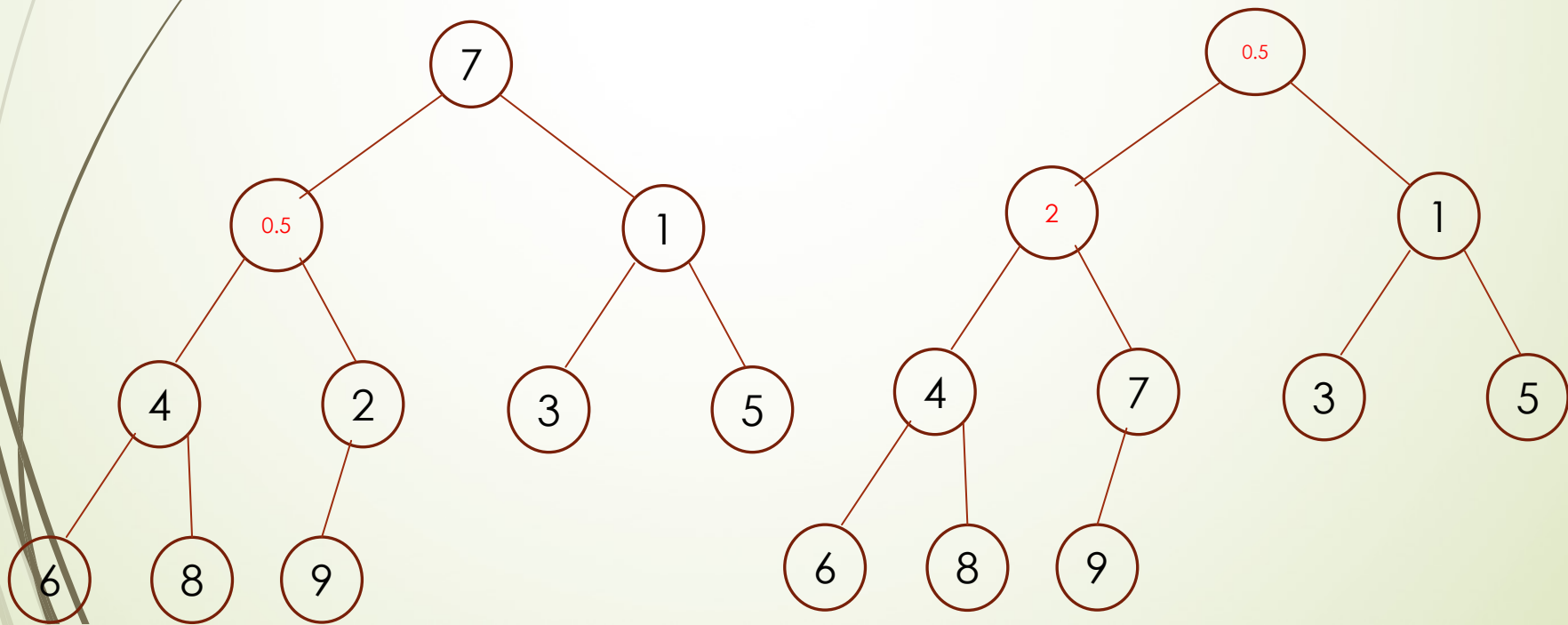
```
>>> heapq.heappop(heap)
```

#出堆，自动重建

```
0
```

```
>>> heap
```

```
[0.5, 2, 1, 4, 7, 3, 5, 6, 8, 9]
```



# 堆

```
>>> myheap=[1,2,3,5,7,8,9,4,10,333]
```

```
>>> heapq.heapify(myheap)
```

#建堆

```
>>> myheap
```

```
[1, 2, 3, 4, 7, 8, 9, 5, 10, 333]
```

```
>>> heapq.heapreplace(myheap,6)
```

#弹出最小元素，同时插入新元素

```
1
```

*实验：画出这个过程*

```
>>> myheap
```

```
[2, 4, 3, 5, 7, 8, 9, 6, 10, 333]
```

```
>>> heapq.nlargest(3, myheap)
```

#返回前3个最大的元素

```
[333, 10, 9]
```

```
>>> heapq.nsmallest(3, myheap)
```

#返回前3个最小的元素

```
[2, 3, 4]
```

# 队列

- First-In-First-Out队列（银行服务、排队吃饭等）

```
>>> import queue                #queue是Python标准库
>>> q=queue.Queue()
>>> q.put(0)                     #入队
>>> q.put(1)
>>> q.put(2)
>>> q.queue
deque([0, 1, 2])

>>> q.get()                     #出队
0

>>> q.queue
deque([1, 2])

>>> q.get()
1
>>> q.queue
deque([2])
```



# 队列

- Python标准库collections提供了双端队列deque

```
>>> from collections import deque
```

```
>>> q = deque(maxlen=5)          #创建双端队列
```

```
>>> for item in [3, 5, 7, 9, 11]:
```

```
    q.append(item)
```

```
>>> q.append(13)
```

```
>>> q.append(15)
```

```
>>> q
```

```
deque([7, 9, 11, 13, 15], maxlen=5)
```

```
>>> q.appendleft(5)              #从左侧添加元素，右侧自动溢出
```

```
>>> q
```

```
deque([5, 7, 9, 11, 13], maxlen=5)
```

```
>>> q.popleft()                  #输出什么？
```

```
>>> q
```

# 队列

- Python标准库collections提供了双端队列deque

```
>>> from collections import deque
```

```
>>> q = deque(maxlen=5)           #创建双端队列
```

```
>>> for item in [3, 5, 7, 9, 11]:
```

```
q.append(item)
```

```
>>> q.append(13)
```

```
>>> q.append(15)
```

>>> q

```
deque([7, 9, 11, 13, 15], maxlen=5)
```

```
>>> q.appendleft(5) #从左侧添加元素，右侧自动溢出
```

>>> q

```
deque([5, 7, 9, 11, 13], maxlen=5)
```

```
>>> q.popleft() #5
```

```
>>> q #deque([7, 9, 11, 13],maxlen=5)
```

# 栈

- 栈是一种后进先出Last-In-First-Out (LIFO)的数据结构。
- Python列表本身就可以实现栈结构的基本操作。
  - 列表的append方法是在列表尾部追加元素，类似于入栈操作；pop方法默认是弹出并返回列表的最后一个元素，类似于出栈操作。

```
>>> stack = [3, 2, 5, 6, 9]
```

```
>>> stack.append(4)
```

```
>>> stack
```

```
[3, 2, 5, 6, 9, 4]
```

```
>>> stack.pop()
```

```
4
```

```
>>> stack
```

```
[3, 2, 5, 6, 9]
```

```
>>> stack.pop(2) #合法，但不符合栈的定义
```

# 栈

- 直接使用列表模拟栈操作问题
  - 当列表为空时再执行pop()出栈操作时则会抛出异常
  - 无法限制栈的大小

```
>>> myStack = [3, 5, 7]
>>> myStack.pop()
7
>>> myStack.pop()
5
>>> myStack.pop()
3
>>> myStack.pop()
出错
```

解决办法：自定义栈模块，然后导入

# 面向对象程序设计

```
class Stack:
    def __init__(self, size = 10):
        self._content = []           #使用列表存放栈的元素
        self._size = size            #初始栈大小
        self._current = 0            #栈中元素个数初始化为0

    def empty(self):
    def isEmpty(self):
    def isFull(self):

    def setSize(self, size) :
        #如果缩小栈空间，则删除指定大小之后的已有元素

    def push(self, v):
    def pop(self):
    def show(self):
```

解决办法： `import Stack`

# 面向对象程序设计

```
class Stack: #class关键字定义类 (class)
```

```
    def __init__(self, size = 10): #初始化
        self._content = []         #使用列表存放栈的元素
        self._size = size          #初始栈大小
        self._current = 0          #栈中元素个数初始化为0
```

```
# Stack类里面的函数，self参数代表对象本身，必须是第一个参数
```

```
    def empty(self):
    def isEmpty(self):
    def isFull(self):
    def push(self, v):
```

```
# 使用
```

```
import Stack
s=Stack.Stack()
s.isEmpty()
...
```

# 其他数据结构

- 图 networkx Graph Library

```
import networkx as nx
```

```
g = nx.Graph()# Creates a graph  
# Adds edge from node 1 to node 2  
g.add_edge(1, 2)  
g.add_edge(1, 3)  
g.add_node(4)
```

```
print("Edges:", g.edges())  
print("Nodes:", g.nodes())  
print("Neighbors of node 1:", list(g.neighbors(1)))
```

Edges: [(1, 2), (1, 3)]

Nodes: [1, 2, 3, 4]

Neighbors of node 1: [2, 3]

# 其他数据结构

- 自定义以下模块，导入使用
  - ✓ 链表
  - ✓ 二叉树
- 课后练习