第2章 数据类型

李宇 讲师

东北大学-计算机学院-智慧系统实验室

liyu@cse.neu.edu.cn

Python常用数据结构

- **序列(Sequence)**：
  - 序列中每个元素都有自己的编号(位置/索引)
  - 索引从0开始
  - Python内置序列类型：
    - **列表(list)、元组(tuple)、字符串(string)**、 unicode字符串、buffer对象和xrange对象
- **映射(map)：**
  - **字典(dict)**是Python中唯一内建的映射类型
  - 键-值对 （键值无重复）
- **集合(set)**
  - 无序排列、无法使用索引、元素不重复

Python高级数据结构

- Collections模块
  - Counter（计数器）：
    - 统计一个list中元素出现的次数
  - Deque （双端队列）：
    - 在队列两端添加或删除
  - Defaultdict（默认字典）：
    - dict的子类， 如果键不在于字典中，会添加新的键并将值设为默认值
  - Nametuple（可命名元组）：
    - 给元祖的而每个元素起一个名字，这样就可以通过名字访问元素，有点类似字典
  - OrderedDict（ 有序字典）：
    - 将无序的字典变为序有

不详细介绍

列表

- 列表(List) ∈ 序列(Sequence)
  - 序列中每个元素都有自己的编号(位置/**索引**)
  - 索引**从0开始**
  - 用[ ]来表示
- 定义后**可修改**
- 每个元素的**类型不必一致**
- 几乎在所有情况下都可使用列表来代替元组

```
edward = ['Edward Gumby', 42]
john = ['John Smith', 50]
database = [edward, john]
```

列表(list)的基本操作（一）

Northeastern University

- 函数 list(object)：
  - 创建列表，创建空列表： list()
  - 将字符串、字典、元组、集合等**可迭代对象**转换为列表
  - **例：list**(‘Hello’) `# ['H', 'e', 'l', 'l', 'o’]`
  - 转换回字符串： ''.join(['H', 'e', 'l', 'l', 'o’])
- 修改列表：给某个元素赋值

  x = [1,1,1]

  x[1] = 2

  x `# [1,2,1]`
- 删除元素
  - names = ['Alice', 'Beth', 'Cecil', 'Dee-Dee', 'Earl']
  - **del** names[2:4]
  - names `# ['Alice', 'Beth', 'Earl']`

列表(list)的基本操作（二）

- 切片赋值 （批量赋值）

```
name = list('Perl')
name # ['P', 'e', 'r', 'l']
name[2:] =list('ar')
name  # ['P', 'e', 'a', 'r']
```

将切片替换为长度与其不同的序列

```
name = list('Perl')
name[1:] = list('ython')
name #['P', 'y', 't', 'h', 'o', 'n']
```

- 在某位置插入新元素

```
numbers = [1, 5]
numbers[1:1] = [2, 3, 4]
numbers  # [1, 2, 3, 4, 5]
```

- 用切片赋值操作来删除元素

```
numbers = [1, 2, 3, 4, 5]
numbers[1:4] = [ ]
numbers  # [1, 5]
```

列表的常用方法（一）

- 方法的调用：object.method(arguments)

- append, clear
  - lst = [1, 2, 3]

    lst.append(4) → 将对象附加到队列末尾

    lst　# [1,2,3,4]

    lst.clear() → 清空列表内容 类似于 lst [:] = [ ]

    lst　# []

列表的常用方法（二）

- copy

```
a = [1, 2, 3]
b = a
b[1] = 4
a # [1,4,3 ]
```

```
a = [1, 2, 3]
b = a.copy()
b[1] = 4
a # [1,2,3 ]
```

→ a[:]或list(a)也都复制a

- count：统计列表中某元素出现的次数
  - x = [[1, 2], 1, 1, [2, 1, [1, 2]]]
  - x.count(1) # 2
- extend 将**多个**值添加到列表队尾

```
a = [1, 2, 3]
b = [4, 5, 6]
a.extend(b)
a   # [1, 2, 3, 4, 5, 6]
```

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a+b
c  # [1, 2, 3, 4, 5, 6]
a  # [1, 2, 3]
```

→ 拼接时会创建a和b的副本
a=a+b 结果等同于 a.extend(b)，但效率低

列表的常用方法（三）

- index： 在列表中**查找**指定值第一次出现的**索引**

```
knights = ['We', 'are', 'the', 'knights', 'who', 'say', 'ni']
knights.index('who')      # 4
knights.index('herring')  # Error
```

- insert： 在索引处**插入**对象到列表

```
numbers = [1, 2, 3, 5, 6, 7]
numbers.insert(3, 'four')
numbers  # [1, 2, 3, 'four', 5, 6, 7]
```

→ 在索引3处，插入'four'，与切片赋值方法效果一样

- pop： 从列表中**删除**一个元素，并**返回**此元素

```
numbers.pop() # [1, 2, 3, 'four', 5, 6]
```

→ 默认弹出最后一个元素

```
numbers.pop(0) # [2, 3, 'four', 5, 6]
```

→ 指定位置弹出

  - 既修改列表，又返回非none值

列表的常用方法（四）

- remove： 删除第一个指定的元素

```
x = ['to', 'be', 'or', 'not', 'to', 'be']
x.remove('be')
x      # ['to', 'or', 'not', 'to', 'be']
```

→ 与pop不同的是，remove修改列表，但不返回值

- reverse： 反序排列列表中元素
  - 修改列表，但不返回值
- sort： **就地排序**，修改列表，**无返回值**
  - y = x.sort() # x已排完序，y为none
  - 函数sorted：非就地排序

```
x = [4, 6, 2, 1, 7, 9]
y = sorted(x)
x # [4, 6, 2, 1, 7, 9]
y # [1, 2, 4, 6, 7, 9]
```

→ 有返回值，不改变x，可获得排序后的副本

列表的常用方法（五）

- 高级sort
  - 方法sort接受两个可选参数：key和reverse
  - key=len：根据元素长度排序

```
x = ['aardvark', 'abalone', 'acme', 'add', 'aerate']
x.sort(key=len)
x # ['add', 'acme', 'aerate', 'abalone', 'aardvark']


# 获取elem的第二个元素
def takeSecond(elem):
    return elem[1]
random = [(2, 2), (3, 4), (4, 1), (1, 3)]
# 指定第二个元素排序
random.sort(key=takeSecond)
# 输出
print(random)
#[(4, 1), (2, 2), (1, 3), (3, 4)]
```

列表的常用方法（五）

- reverse参数可以指定为True/False，表示是否要反序排列

```
x = [4, 6, 2, 1, 7, 9]
x.sort(reverse=True)
x # [9, 7, 6, 4, 2, 1]
```

排序后再反序

元组（一）

- **不可修改**的序列
  - 与列表唯一的区别
- 用逗号分隔，可用( )括起来

```
x = 1, 2, 3
x = (1, 2, 3)
( ) # 空元组
x = 42,  # (42,) 只有一个值的元组，最后需有一个逗号
3 * (40 + 2)  # 126
3 * (40 + 2,) # (42, 42, 42)
```

- tuple(iterable) 方法
  - 传入序列，创建元组
  - 与list函数类似

```
tuple([1, 2, 3])  # (1, 2, 3)

tuple('abc')  # ('a', 'b', 'c')

tuple((1, 2, 3))  # (1, 2, 3)
```

元组（二）

- 元组取值操作和列表一样

```
x = 1, 2, 3
x[1] # 2
x[0:2] # (1, 2)
```

- 因为不能对元组进行修改，除了创建和访问元素外，可做的操作不多。

- 一般使用列表足以满足对序列的要求，存在意义？
  - 元组可用作映射中的**键**（以及集合的成员），而列表不行。
  - 有些内置函数和方法返回元组

字符串

- 所有标准序列操作都适用于字符串
  - 索引，切片，乘法，加法，成员资格检查，长度，最小最大值
- 字符串**不可变**
  - 元素赋值和切片赋值都非法，报错

```
x = ‘abcdef'
x[-3:] = ‘aaa'
```

✗

- 字符串格式设置
  - 转换说明符——百分号(%)。

```
format = "Hello, %s. %s enough for ya?"
values = ('world', 'Hot')
format % values
'Hello, world. Hot enough for ya?'
```

```
print(“My name is %s and weight is %.2f kg!" % ('Zara', 21.332))
```

字符串格式化符号

符 号	描述
%c	格式化字符及其ASCII码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f和%e的简写
%G	%F 和 %E 的简写
%p	用十六进制数格式化变量的地址

字符串

- 字符串格式设置
  - 转换说明符——百分号(%)。
  - 模版字符串

```
from string import Template
tmpl = Template("Hello, $who! $what enough for ya?")
tmpl.substitute(who="Mars", what="Dusty") # 'Hello, Mars! Dusty enough for ya?'
```

使用Template()创建模版

  - 字符串方法format（推荐）
    - "{}, {} and {}".format("first", "second", "third") `# 'first, second and third'`
    - "{0}, {1} and {2}".format("first", "second", "third") → 使用索引
    - "{3} {0} {2} {1} {3} {0}".format("be", "not", "or", "to") `# 'to be or not to be'`
    - from math import pi

```
"{name} is approximately {value:.2f}.".format(value=pi, name="π")
结果:  'π is approximately 3.14.'
```

使用关键字参数

字符串

- 字符串格式设置
  - 转换说明符——百分号(%)。
  - 模版字符串
  - 字符串方法format （推荐）
  - f字符串
    - 如果变量与替换字段同名，可使用f字符串简写。在字符串前加上f

```
from math import e
f"Euler's constant is roughly {e}."
输出："Euler's constant is roughly 2.718281828459045."
```

使用f字符串，则.format(e=e)可以省略

```
"Euler's constant is roughly {e}.".format(e=e)
输出："Euler's constant is roughly 2.718281828459045."
```

字符串基本转换

- 设置花括号内字段的格式
  - **转换**标志： !
  - 三种标志: !s （与原值一样），!r（原值加引号），!a（带引号的ascii编码）

```
print("{pi!s} {pi!r} {pi!a}".format(pi="π"))
# π 'π' '\u03c0'
```

  - **格式**说明符（冒号：）

```
"The number is {num:f}".format(num=42)
# 'The number is 42.000000'

"The number is {num:b}".format(num=42)
'The number is 101010'
```

类型说明符表格

类型	含　义
b	将整数表示为二进制数
c	将整数解读为Unicode码点
d	将整数视为十进制数进行处理，这是整数默认使用的说明符
e	使用科学表示法来表示小数（用e来表示指数）
E	与e相同，但使用E来表示指数
f	将小数表示为定点数
F	与f相同，但对于特殊值（nan和inf），使用大写表示
g	自动在定点表示法和科学表示法之间做出选择。这是默认用于小数的说明符，但在默认情况下至少有1位小数
G	与g相同，但使用大写来表示指数和特殊值
n	与g相同，但插入随区域而异的数字分隔符
o	将整数表示为八进制数
s	保持字符串的格式不变，这是默认用于字符串的说明符
x	将整数表示为十六进制数并使用小写字母
X	与x相同，但使用大写字母
%	将数表示为百分比值（乘以100，按说明符f设置格式，再在后面加上%）

宽度、精度、千位分隔符

- 宽度使用整数指定
  - 数和字符串对齐方式不同

```
"{num:10}".format(num=3)
# ‘         3'
"{name:10}".format(name="Bob")
# ‘Bob      ‘
```

- 精度也使用整数指定
  - 可同时指定宽度和精度

```
"{pi:10.2f}".format(pi=pi)
# ‘       3.14'
```

- 千分位分隔符：逗号

```
‘One googol is { :, }'.format(10**10)
# 'One googol is 10,000,000,000'
```

符号、对齐、用0填充

- 在指定宽度和精度的**数前面**，可添加一个标志
  - 可以是零、加号减号或空格
  - 零表示使用**0来填充**

```
'{:010.2f}'.format(pi)
# '0000003.14'
# 010.2f中，第一个0表示用0填充，10表示宽度
```

- 左对齐(<)、右对齐(>)、居中(^)

```
print('{0:<10.2f}\n{0:^10.2f}\n{0:>10.2f}'.format(pi))
3.14
    3.14
        3.14
```

- 填充其他符号

```
"{:$^15}".format(" WIN BIG ")
# '$$$ WIN BIG $$$'
```

→ 一共15位，居中，不足的用$号补齐

符号、对齐、用0填充

- 说明符 “=”，将**填充**字符放在**符号和数字之间**

```
print('{0:10.2f}\n{1:=10.2f}'.format(pi, -pi))
#       3.14
#-      3.14
```

此处没有指定填充字符，默认为空格
如果想使用$号填充，则：

```
print(‘{0:10.2f}\n{1:$=10.2f}’.format(pi, -pi))
```

- 加号、减号：显示正负号

```
print('{0:-.2}\n{1:-.2}'.format(pi, -pi))
3.1
-3.1
#显示负号是默认的，不写也行
```

```
print('{0:+.2}\n{1:+.2}'.format(pi, -pi))
+3.1
-3.1
#当数字是正数时，显示正号
```

字符串的方法（一）

- center方法：通过填充，让字符居中

"The Middle by Jimmy Eat World".center(39, "*")
'*****The Middle by Jimmy Eat World*****'

- find方法：查找子串，返回第一个字符索引，或-1
  - str.find(str, start=0, end=len(string))

'With a moo-moo here, and a moo-moo there'.find('moo’)
7
区别于成员资格检查：in（返回True/False）

str='$$$ Get rich now!!! $$$'
str.find(‘!!!’, 0, 16) # 指定了查找的起点和终点

字符串的方法（二）

- join方法：与split相反，用于合并序列元素

```
sep = '+'
seq = ['1', '2', '3', '4', '5']
sep.join(seq) # 用加号合并一个字符串列表
# '1+2+3+4+5'
```

- lower,upper方法：返回字符串的**小写/大写**版本
- replace方法：将指定子串替换为另一个字符串
  - replace(old, new, count=-1, /)

```
'This is a test'.replace('is', 'eez')
# 'Theez eez a test'
```

- strip方法：把开头和末尾的空格删除，并返回
  - str.lstrip()、str.rstrip()

字符串的方法（三）

- split方法：将字符串**拆分为序列**，默认按空格和换行
  - split(sep=None, maxsplit=-1)

```
'1+2+3+4+5'.split('+')
#['1', '2', '3', '4', '5']
```

- translate方法：单字符替换，可换多个

```
table = str.maketrans('cs', 'kz')
'this is an incredible test'.translate(table)
# 'thiz iz an inkredible tezt'
```

→ 使用translate前先要创建转换表

- 判断字符串是否满足特定条件的方法：**is开头**，返回True/False
  - isalnum、isdigit、isidentifier、islower、isnumeric、isprintable、isspace、isupper

字符串is开头的方法

方　法	描　述
string.isnumeric()	检查字符串中的所有字符是否都是数字字符
string.isprintable()	检查字符串中的字符是否都是可打印的
string.isspace()	检查字符串中的字符是否都是空白字符
string.istitle()	检查字符串中位于非字母后面的字母都是大写的，且其他所有字母都是小写的
string.isupper()	检查字符串中的字母是否都是大写的
string.isalnum()	检查字符串中的字符是否都是字母或数
string.isalpha()	检查字符串中的字符是否都是字母
string.isdecimal()	检查字符串中的字符是否都是十进制数
string.isdigit()	检查字符串中的字符是否都是数字
string.isidentifier()	检查字符串是否可用作Python标识符
string.islower()	检查字符串中的所有字母都是小写的

字符串is开头的方法

- str.isdecimal()
  - 如果str中只有十进制字符，返回true，否则false
- str.isdigit()
  - 如果str中所有字符都是数字，则返回true
- str.isnumeric()
  - 如果str中只有数字字符，返回true

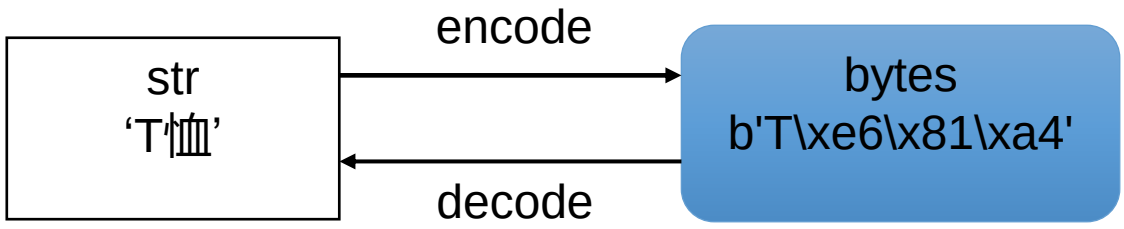
```
s='123'
print(s.isdigit())
print(s.isdecimal())
print(s.isnumeric())
#true
#true
#true
```

```
s=b'123'
print(s.isdigit())
print(s.isdecimal())
print(s.isnumeric())
#true
#error
#error
```

```
s='123.0'
print(s.isdigit())
print(s.isdecimal())
print(s.isnumeric())
#false
#false
#false
```

```
s='三叁'
print(s.isdigit())
print(s.isdecimal())
print(s.isnumeric())
#false
#false
#true
```

str和bytes



- str和bytes的相互转换
- **str.encode(‘encoding’) -> bytes**
- **bytes.decode(‘encoding’) -> str**
  - encoding 指的是具体的编码规则的名称，对于中文来说，它可以是这些值： ‘utf-8’, ‘gb2312’, ‘gbk’, ‘big5’ 等等

```
"AAAbbb".encode()
# b'AAAbbb’

b'AAAbbb'.decode()
#'AAAbbb’
```

```
"T恤".encode()
# b'T\xe6\x81\xa4’

b = b'T\xe6\x81\xa4'
b.decode()
#'T恤'
```

```
b = "T恤".encode('gbk’)
# b'T\xd0\xf4’

b.decode('gbk’)
#'T恤'
```

字符串is开头的方法

- string.isidentifier()： 检查字符串是否可用作标识符
  - 以下变量名中，不符合 Python 语言变量命名规则的是 （ ）
    - A. keyword_33
    - B. keyword33_
    - C. 33_keyword
    - D. _33keyword
- Python中标识符的命名规则：
  - 标识符是由字符（A~Z 和 a~z）、下划线和数字组成，但第一个字符不能是数字。
  - 标识符不能和 Python 中的保留字相同。有关保留字，后续章节会详细介绍。
  - Python中的标识符中，不能包含空格、@、% 以及 $ 等特殊字符。

字符串is开头的方法

- string.isprintable(): 检查字符串中的字符是否都可打印
  - 占用屏幕打印空间的字符称为可打印字符
    - 字母和符号（包括空字符）
    - 数字
    - 标点
    - 空格

```
s = '\nNew Line is printable'
print(s.isprintable())
#false

s = '' #空字符串
s.isprintable())
#true
```

- Unicode字符集中“Other” “Separator”类别的字符**为不可打印的字符**（但不包括ASCII码中的空格（0x20））
  - \t, \n, \b, \v, \r, \f,\b,\a,\v 等
  - 以上转义字符都含有分割打印格式的语义，所以不可打印

补充

- str.strip(self, **chars=None**, /)
  - Return a copy of the string with leading and trailing whitespace remove.(无参数时，去掉两端空白)
  - If chars is given and not None, remove characters in chars instead.（有参数时，去掉字符串中的字符）
  - 字符串 s="To Be or Not to Be"，执行 s.strip("To Be")语句结果是 （ ）
    - A. "To Be or Not to Be"
    - B. " Be or Not to "
    - C. "r N"
    - D. "r Not t"

序列的通用操作（一）

- 索引(indexing)：访问单个元素

```
>>> x = [1,2,3,4]
>>> x[1]  # 2
>>> x[-1] # 4
```

```
>>> greeting = 'Hello'
>>> greeting[0] # 'H'
>>> greeting[-2]# 'l'
```

- 切片(slicing)： 访问特定范围内的元素

```
x[1:3] #[2,3]
#第一个索引指定的元素包含在切片内，第二个索引不包含
x[1:]  #[2,3, 4]
x[:2]  #[1,2]
x[1:4] #[2,3, 4]
```

```
x[-3:0]  #[]
x[0:3:2] # [1,3] 2是步长
x[3:0:-1]  # [4,3,2]
```

序列的通用操作（二）

- 连接(Concatination)
  - 序列相加
    - [1, 2, 3] + [4, 5, 6] `# [1, 2, 3, 4, 5, 6]`
    - [1, 2, 3] + ‘world!‘ `# Error,` 不能拼接不同类型的序列
  - 序列相乘
    - ‘python’ * 5 `# 'pythonpythonpythonpythonpython'`
    - sequence = [None] * 5 `# [None, None, None, None, None]`
- 成员资格检查(membership check)
  - 检查特定的值是否包含在序列中 运算符：in，返回True/False

```
subject = '$$$ Get rich now!!! $$$’
'$$$' in subject      # True
```

序列的通用操作（三）

- 长度、最小值、最大值
  - 内置函数len、min、max

```
numbers = [100, 34, 678]
len(numbers)    # 3
max(numbers)    # 678
min(numbers)    # 34
```

- 以上介绍的操作适用于所有序列类型 （列表、元组、字符串）
  - 索引、切片、连接（相加、相乘）、成员资格检查、长度、最小值、最大值

字典(python唯一内置映射)

- 字典：
  - 通过键（**不重复**），访问值（键-值对）
  - **不按顺序**排列
  - 键可以是 **数、字符、元组 （不可变的类型）**

- 创建和使用字典
  - phonebook = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258’}
  - 空字典 : {}

键-值对称为**项(item)**

字典

- 函数dict()：从其他字典或者键-值序列**创建字典**

传入序列　　键　　值　　键　　值

```
items = [('name', 'Gumby'), ('age', 42)]
d = dict(items)
d          #{'age': 42, 'name': 'Gumby'}
```

使用关键字参数

```
d = dict(name='Gumby', age=42)
d       #{'age': 42, 'name': 'Gumby'}
```

```
d['name'] #'Gumby'
```

→ 访问键为'name'的值

- 字典的基本操作
  - len(d)： 返回字典的**项数**
  - d[key] : 返回键key的**值**
  - d[key] = v : 将值v关联到键key
  - del d[key]： 删除键为k的**项**
  - key in d ： 检查字典d中是否包含键是k的项

字典方法

- clear：删除所有字典项，无返回值

```
x = {}
y = x
x['key'] = 'value‘
y   #{'key': 'value’}
x = {} # 并不能清空，而是在内存中开辟
       一个空字典，并贴上名为x的标签
y   #{'key': 'value’}
```

```
x = {}
y = x
x['key'] = 'value‘
y #{'key': 'value’}
x.clear()  #清空x中的内容

y  #{} y指向的内容也被清空
```

- copy：浅复制（无法复制嵌套的值）

```
x = {'username': 'admin', 'machines': ['foo', 'bar', 'baz’]}
y = x.copy()
y['username'] = 'mlh’
y['machines'].remove('bar‘)
y #{'username': 'mlh', 'machines': ['foo', 'baz’]}
x #{'username': 'admin', 'machines': ['foo', 'baz’]}
```

['foo', 'bar', 'baz’]并没有被copy到y里在y中，键‘machines’中的值仍指向原件

字典方法

- deepcopy（深复制）
  - 同时复制值及其包含的所有值
  - 需使用 模块copy中的函数deepcopy
  - from copy import deepcopy
    - deepcopy的用法：y = deepcopy(x)，**不是**y= x.deepcopy
- fromkeys：只使用键创建新字典，**值为none**

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

- get方法：访问字典项
  - 传入键，返回值

```
d = {}
print(d['name'])   # 报错

#使用get()不会报错：
print(d.get('name'))  # None
```

字典方法

- items()：
  - 返回包含所有**项的列表**，每个元素为(key，value)的形式
  - 返回值属于**字典视图类型**，可用于迭代（**不产生副本**，可以理解为原字典的别称，可直接通过视图访问和操作原字典）
- pop()：获取key的值，并删除键-值对
  - dict.pop('key') # return value
- popitem()：随机弹出字典项
  - 字典项的顺序不确定，只能随机pop
  - 可高效的逐个删除并处理所有字典项

字典方法

- update方法：使用一个字典中的项更新另一个字典
  - d.update(x)
  - 用字典x的项更新字典d
  - x中若没有d中的键，则把x添加进d；若有，则替换

```
d = {'title': 'Python', 'url': 'http://www.python.org', 'time': 'Tuesday'}
x = {'title': 'Java'}
d.update(x)
d #{'url': 'http://www.python.org', 'changed': 'Tuesday', 'title': 'Java'}
```

- keys方法：
  - 返回字典视图，包含所有键。
- values方法：
  - 返回所有值（可重复）

集合

- 集合(set)是一个无序、不重复元素序列
  - 使用{ }表示集合(字典也是{ })
  - 也可使用**set()**函数创建集合
  - 创建**空集合**需使用set()，而不是{ } （空字典）

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)
#{'orange', 'banana', 'pear', 'apple'}

a = set('abracadabra') # 用字符串创建集合
print(a)
# {'a', 'r', 'b', 'c', 'd'}
```

  - 集合间运算符：-（差集），|（并集），&(交集)

集合方法

add()	为集合添加元素
clear()	移除集合中的所有元素
copy()	拷贝一个集合
difference()	返回多个集合的差集
difference_update()	移除集合中的元素，该元素在指定的集合也存在。
discard()	删除集合中指定的元素
intersection()	返回集合的交集
intersection_update()	返回集合的交集。
isdisjoint()	判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。
issubset()	判断指定集合是否为该方法参数集合的子集。
issuperset()	判断该方法的参数集合是否为指定集合的子集

集合

- 由内置类set实现
  - 老版本需导入模块sets
  - 集合是**可变的**，不可用做字典中的键
  - 集合只能包含**不可变的值**，因此不能包含其他集合

```
>>> a = set()
>>> b = set()
>>> a.add(b)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: set objects are unhashable
```

  - 使用fronzenset类型可以嵌套集合

```
>>> a.add(frozenset(b))
```

使用构造函数frozenset创建一个不可变的集合

字符串小练习

- "{foo} {} {bar} {}".format(1, 2, bar=4, foo=3)
- "{foo} {1} {bar} {0}".format(1, 2, bar=4, foo=3)
- ```
  fullname = ["Alfred", "Smoketoomuch"]
  "Mr {name[1]}".format(name=fullname)
  ```
- ```
  import math
  tmpl = "The {mod.__name__} module defines the value {mod.pi} for π"
  tmpl.format(mod=math)
  ```

Python中的内置数字类型

- **int类型**
  - 包括正整数，0和负整整，不能包含小数点
  - int类型默认为10进制的，我们也可以在程序中使用二进制、八进制和十六进制的整型数字
- **float类型**
  - float类型是含小数点的数字。包括正负浮点数
  - 也可以使用“e”或“E”来定义科学计数法的浮点数

```
x = 1.2E3
print(x) #输出1200.0

y = 12.34e3
print(y)#输出12340.0
```

- **complex类型**
  - 复数类型有两部分组成：实部和虚部，复数的虚部在Python中使用j来做后缀。

```
x = complex(1, 2)
print(x)   #(1+2j)

y = complex(1)
print(y)   #(1+0j)
```

```
x = complex("-2 + 3j")
print(x)   #报错，arg is a
           malformed string
```

```
z = complex(-1, -2.4)
print(z)   #(-1-2.4j)

x = complex("-2+3j")
print(x)   #(-2+3j)
```

堆(heap)

- 一种优先队列
  - 能够以任意顺序添加对象
  - 随时找出（并删除）最小的元素
    - 比列表的min方法要高效
- 模块heapq(q表示queue)
  - Python中没有独立的堆类型，需要使用模块

表10-5　模块heapq中一些重要的函数

函　　数	描　　述
heappush(heap, x)	将x压入堆中
heappop(heap)	从堆中弹出最小的元素
heapify(heap)	让列表具备堆特征
heapreplace(heap, x)	弹出最小的元素，并将x压入堆中
nlargest(n, iter)	返回iter中n个最大的元素
nsmallest(n, iter)	返回iter中n个最小的元素

heappush：往堆中添加元素


```
>>> from heapq import *
>>> from random import shuffle
>>> data = list(range(10))
>>> shuffle(data)
>>> heap = []
>>> for n in data:
...     heappush(heap, n)
...
>>> heap
[0, 1, 2, 3, 4, 6, 8, 9, 7, 5]
>>> heappush(heap, 0.5)
>>> heap
[0, 0.5, 2, 4, 1, 7, 3, 8, 9, 6, 5]
```

创建一个含有数字0-9的list

打乱顺序

此时heap只是一个列表

使用heappush后，变成了堆

注意顺序

注意顺序的变化

堆特征(heap property)

```
>>> heap
[0, 1, 2, 3, 4, 6, 8, 9, 7, 5]
>>> heappush(heap, 0.5)
>>> heap
[0, 0.5, 2, 4, 1, 7, 3, 8, 9, 6, 5]
```

- 位置i处的元素总是大于位置i // 2处的元素（反过来说就是小于位置2 * i和2 * i + 1处的元素）。这是底层堆算法的基础
- heappop弹出最小元素(总是位于索引0处)

```
>>> heappop(heap)
0
>>> heappop(heap)
0.5
>>> heappop(heap)
1
>>> heap
[2, 3, 4, 7, 9, 6, 5, 8]
```

heapify方法

- heapify将列表变成合法的堆
  - 如果堆不是用heappush创建的，则应该在使用heappush和pop之前使用heapify

```
>>> heap = [5, 8, 0, 3, 6, 7, 9, 1, 4, 2] #此时只是普通列表
>>> heapify(heap) #变成堆
>>> heap
[0, 1, 5, 3, 2, 7, 9, 8, 4, 6]   #排序改变、具备堆的排序特征
```

- heapreplace方法： 弹出**最小**元素、同时压入一个新元素

```
>>> heapreplace(heap, 0.5)
0
>>> heap
[0.5, 1, 5, 3, 2, 7, 9, 8, 4, 6]
>>> heapreplace(heap, 10)
0.5
>>> heap
[1, 2, 5, 3, 6, 7, 9, 8, 4, 10]
```

弹出0，同时压入0.5

列表推导

- 从其他列表创建列表的一种方式
  - 类似for循环
  - 需要用方括号[ ]括起来。（因为是列表）
  - 例：由range(10)内每个值的平方组成的列表：

```
>>> [x * x for x in range(10)]
#[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

  - 可以在推导中添加一条if语句

```
>>> [x * x for x in range(10) if x % 3 == 0]
#[0, 9, 36, 81]
# for前面的部分（x*x），代表所生成列表中的元素
```

列表推导

- 添加更多for的部分

```
>>> [(x, y) for x in range(3) for y in range(3)]
#[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

- 以上代码结果等价于使用2次for循环

```
result = []
for x in range(3):
        for y in range(3):
                result.append((x, y))
```

- 使用多个for时，也可以添加if子句

```
>>> girls = ['alice', 'bernice', 'clarice']
>>> boys = ['chris', 'arnold', 'bob']
>>> [b+'+'+g for b in boys for g in girls if b[0] == g[0]]
#['chris+clarice', 'arnold+alice', 'bob+bernice']
```

字典推导

- 可使用花括号来执行字典推导。

```
squares = {i:"{} squared is {}".format(i, i**2) for i in range(10)}
squares[8]
#'8 squared is 64'
```

- 在列表推导中，for前面只有一个表达式，而在字典推导中，for前面有两个用冒号分隔的表达式。这两个表达式分别为键及其对应的值。
- 来个简单点的

```
squares = {i: i*i for i in range(10)}
print(squares)

#{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

生成器(generator)

- 生成器是一种使用普通函数语法定义的迭代器

```
nested = [[1, 2], [3, 4], [5]]

def flatten(nested):
        for sublist in nested:
                for element in sublist:
                        yield element
```

包含yield语句的函数都被称为生成器

- 生成器**不使用return**返回值，而是可以**生成**多个值，每次一个。每次使用yield生成一个值后，函数都将冻结，即在此停止执行，等待被重新唤醒。被重新唤醒后，函数将从停止的地方开始继续执行。使用next()函数逐个取出

生成器

- 若想使用生成器的所有的值，可对**生成器**进行**迭代。**

```
nested = [[1, 2], [3, 4], [5]]
for num in flatten(nested):
        print(num)
1
2
3
4
5
```

- 也可以

```
x = list(flatten(nested))
x
[1, 2, 3, 4, 5]
```

生成器推导

- 使用圆括号代替方括号不能实现元组推导，而是创建**生成器**
- 生成器推导（也叫生成器表达式），可以**逐步计算**
  - 工作原理与列表推导相同，但不是创建一个列表（即不立即执行循环），而是返回一个生成器

```
>>> g = ((i + 2) ** 2 for i in range(2, 27))
>>> next(g)
#16
>>> next(g)
#25
```

此时，g是一个生成器

- 优点：
  - 迭代量大时，占内存少
  - 代码优雅

```
sum(i ** 2 for i in range(10))
```

在一对既有的圆括号内（如在函数调用中）使用生成器推导时，无需再添加一对圆括号

迭代器(Iterator)

- 生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值
  - 可以被next()函数调用并不断返回下一个值的对象称为**迭代器**：**Iterator**

```
>>> from collections import Iterator, Generator

>>> isinstance((x for x in range(10)), Generator)
True
>>> isinstance((x for x in range(10)), Iterator)
True
>>> isinstance([], Iterator)
False
>>> isinstance({}, Iterator)
False
>>> isinstance('abc', Iterator)
False
```

生成器也是Iterator对象

list、dict、str不是Iterator实例

迭代器 (Iterator)

- 任何实现了__iter__() 和 __next__()方法的对象都是迭代器
  - __iter__() 返回迭代器自身
  - __next__() 返回下一个值
  - 如果迭代时没有更多元素了，则抛出StopIteration异常
  - 例：itertools这个模块内含有多种类型的迭代器

```
from itertools import count
counter = count(start=13)
next(counter)
#13
next(counter)
#14
```

  - count方法用于生成连续整数，从13开始的无限整数序列，是一种迭代器

自定义迭代器

```
class Fib:
    def __init__(self):
        self.prev = 0
        self.curr = 1

    def __iter__(self):
        return self

    def __next__(self):
        value = self.curr
        self.curr += self.prev
        self.prev = value
        return value

f = Fib()
for x in f:
    print(x)
    if(x>100):
        break
```

因为Fib实现了__iter__方法和__next__方法，所以Fib的实例f是一个迭代器

可迭代对象(Iterable)

- Iterable：可以直接作用于for循环的数据类型，有：
  - 内置数据类型，如list、tuple、dict、set、str
  - 生成器，包括生成器推导和带yield的生成器函数
  - 迭代器
  - 以上都称为可迭代对象：Iterable

- 使用isinstance()查看是否是可迭代对象

```
>>> from collections.abc import Iterable #需导入模块
>>> isinstance([], Iterable)
True
>>> isinstance({}, Iterable)
True
>>> isinstance('abc', Iterable)
True
>>> isinstance((x for x in range(10)), Iterable)
True
```

迭代器与生成器区别

- 生成器只能遍历一次，而迭代器可以迭代多次。
- 生成器是一类特殊的迭代器， 它的返回值不是通过return而是用yield 。

lambda表达式

- 也叫做匿名函数
  - 能够创建内嵌的简单函数: lambda arg1,arg2,arg3… :<表达式>

```
f = lambda x: x.isalnum()
f("aa12")
#true
g = lambda x, y: x + y #多个参数
g(1,2)
#3
```

- lambda与def的区别:
  - def创建的方法是有名称的，而lambda没有
  - lambda表达式 ” : “ 后面，只能有**一个**表达式，def则可以有多个
  - 像 **if** 或 **for** 等语句**不能**用于lambda中
  - lambda表达式不能共享给别的程序调用(写单独脚本时候用)

结合map、filter、reduce函数

- map(function, sequence)会根据提供的**函数**对指定**序列的每个元素**做映射

```
lst = ['1','2','3','4']
list(map(int,lst))
#[1, 2, 3, 4]
```

```
str = "1234"
list(map(int,str))
#[1, 2, 3, 4]
```

```
foo = [2, 18, 9, 22]
list(map(lambda x: x * 2 + 10, foo))
#[14, 46, 28, 54]
```

- filter(function, sequence)函数——筛选函数
  - 按照 function函数的规则在列表 sequence 中筛选数据

```
lst =[1,2,3,4]
list(filter(lambda x: x>2, lst))
#[3, 4]
```

结合map、filter、reduce函数

Northeastern University

- reduce(function, sequence)——求积累运算
  - 将sequence 中数据，按照 function 函数操作，如将列表第一个数与第二个数进行 function 操作，得到的结果和列表中下一个数据进行 function 操作，一直循环，返回一个值

```
from functools import reduce
lst =[1, 2, 3, 4]
reduce(lambda x,y: x+y, lst)
#10

from functools import reduce
lst =[1, 2, 3, 4]
reduce(lambda x,y: x*y, lst)
#24
```

```
def myadd(x,y):
        return x+y
lst =[1, 2, 3, 4]
reduce(myadd, lst)
#10
```

- 感谢