# 第三章 列表

## 1.list 的增删改

**增**：list.append() 在末尾增加元素

list.insert(0,x) 在位置0添加元素x，添加完成后所有元素向后移位。

**删**：list.pop() 弹出末尾元素

List.pop(x) 弹出位置x的元素

del list[0] 删除位置0的元素

**何时用pop何时用del**

如果你要从列表中删除一个元素， 且不再以任何方式使用它， 就使用del 语句； 如果你要在删除元素后还能继续使用它， 就使用方法pop() 。

**根据值来删除元素：**

List.remove(x) 删除值为x的元素

## 2.组织list

### 1.永久性的排序：list.sort（）

### 2. 使用函数sorted() 对列表进行临时排序

要保留列表元素原来的排列顺序， 同时以特定的顺序呈现它们， 可使用函数sorted() 。 函数sorted() 让你能够按特定顺序显示列表元素， 同时不影响它们在列表中的原始排

列顺序

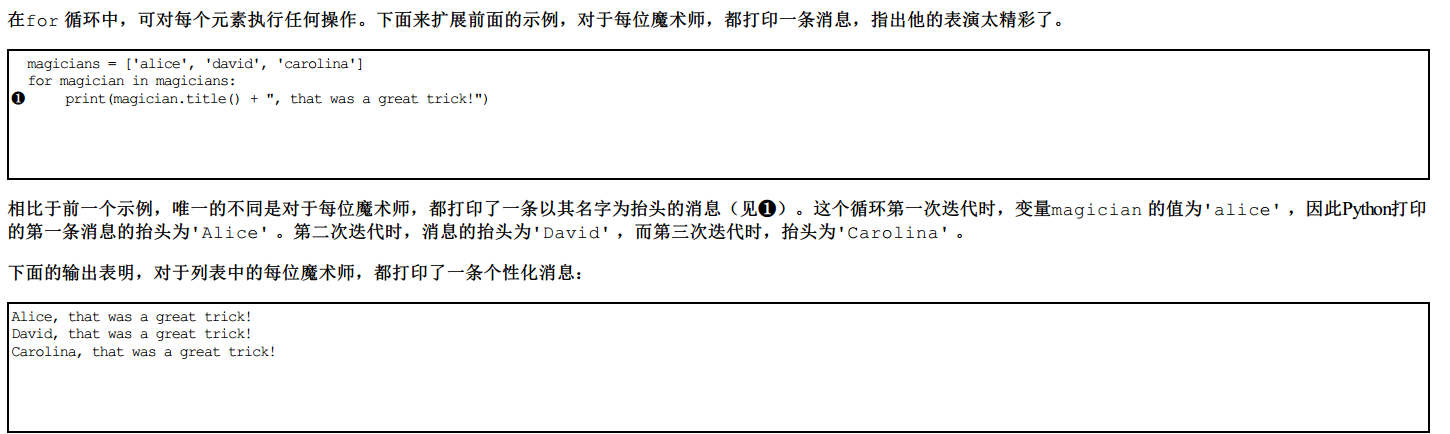
### 3.列表翻转：list.reverse()

### 4.列表长度：len(list)

# 第四章：操作列表

## 遍历：

### 在for 循环中执行更多的操作：title() 方法返回"标题化"的字符串,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())。



## 数值列表：

### Range：

Range(a,b) 从a开始一直到b前一个数结束

Range(a,b,c) 步长为c

### squares = [value\*\*2 for value in range(1,11)]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

## List的复制与关联

my\_foods = ['pizza', 'falafel', 'carrot cake']

复制：

friend\_foods = my\_foods[:]

此时两个list是不相关的list，改变my\_foods不改变friend\_foods

关联：

friend\_foods = my\_foods

这种语法实际上是让Python将新变量friend\_foods 关联到包含在my\_foods 中的列表， 因此这两个变量都指向同一个列表。

# 第五章 if语句

## 1.检查特定值是否在list中

If xxx in list:

Or

If xxx not in list:

# 第六章 字典

## 1.增删查改

创建空字典：

Alian\_0 = {}

添加：

Alian\_0[‘color’] = ‘green’

修改：

Alian\_0[‘color’] = ‘yellow’

查：>>>print(alian[‘color’])

>>>yellow

删除：

Del alian[‘color’]

## 2遍历

# 创建字典

user\_0 = {

'username': 'efermi',

'first': 'enrico',

'last': 'fermi',

}

# 遍历：

for key, value in user\_0.items():

print("\nKey: " + key)

print("Value: " + value)

#输出：

Key: last

Value: fermi

Key: first

Value: enrico

Key: username

Value: efermi

## 遍历字典中的所有key

favorite\_languages = {

'jen': 'python',

'sarah': 'c',

'edward': 'ruby',

'phil': 'python',

}

#使用方法.key()

for name in favorite\_languages.keys():

print(name.title())

#输出：

Jen

Sarah

Phil

Edward

方法keys() 并非只能用于遍历； 实际上， 它返回一个列表， 其中包含字典中的所有键，

favorite\_languages = {

'jen': 'python',

'sarah': 'c',

'edward': 'ruby',

'phil': 'python',

}

if 'erin' not in favorite\_languages.keys():

print("Erin, please take our poll!")

输出：

Erin, please take our poll！

## 遍历字典中的所有值

可以使用.value()方法

# 第七章 用户输入和while循环

## 1.input()

Input() 只接收一个参数：就是提示信息

如：

Age = input(How old are you?)

将会打印how old are you？

另外用户输入的内容会被保存为字符串。需要转换类型。

Age = int(age) #转换为int类型

## 2.while循环

### 1.用while循环遍历一个list

While list:

举例：

unconfirmed\_users = ['alice', 'brian', 'candace']

confirmed\_users = []

# 验证每个用户， 直到没有未验证用户为止

# 将每个经过验证的列表都移到已验证用户列表中

while unconfirmed\_users:

current\_user = unconfirmed\_users.pop()

print("Verifying user: " + current\_user.title())

confirmed\_users.append(current\_user)

# 显示所有已验证的用户

print("\nThe following users have been confirmed:")

for confirmed\_user in confirmed\_users:

print(confirmed\_user.title())

### 2.用while删除list中的指定元素

pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']

#删除所有cat元素

While ‘cat’ in pets:

Pets.remove(‘cat’)

# 第八章 函数

## 1.传递任意数量的形参：

例如， 来看一个制作比萨的函数， 它需要接受很多配料， 但你无法预先确定顾客要多少种配料。 下面的函数只有一个形参\*toppings ， 但不管调用语句提供了多少实参， 这个

形参都将它们统统收入囊中：

def make\_pizza(\*toppings):

"""打印顾客点的所有配料"""

print(toppings)

make\_pizza('pepperoni')

make\_pizza('mushrooms', 'green peppers', 'extra cheese')

#输出：

('pepperoni',)

('mushrooms', 'green peppers', 'extra cheese')

形参名\*toppings 中的星号让Python创建一个名为toppings 的空元组， 并将收到的所有值都封装到这个元组中。 函数体内的print 语句通过生成输出来证明Python能够处理

使用一个值调用函数的情形， 也能处理使用三个值来调用函数的情形。 它以类似的方式处理不同的调用， 注意， Python将实参封装到一个元组中， 即便函数只收到一个值也如此

## 2. 结合使用位置实参和任意数量实参

如果要让函数接受不同类型的实参， 必须在函数定义中将接纳任意数量实参的形参放在最后。 Python先匹配位置实参和关键字实参， 再将余下的实参都收集到最后一个形参中。

## 3. 使用任意数量的关键字实参

def build\_profile(first, last, \*\*user\_info):

"""创建一个字典， 其中包含我们知道的有关用户的一切"""

profile = {}

❶ profile['first\_name'] = first

profile['last\_name'] = last

❷ for key, value in user\_info.items():

profile[key] = value

return profile

user\_profile = build\_profile('albert', 'einstein',

location='princeton',

field='physics')

print(user\_profile)

函数build\_profile() 的定义要求提供名和姓， 同时允许用户根据需要提供任意数量的名称—值对。 形参\*\*user\_info 中的两个星号让Python创建一个名为user\_info 的

空字典， 并将收到的所有名称—值对都封装到这个字典中。 在这个函数中， 可以像访问其他字典那样访问user\_info 中的名称—值对。

在build\_profile() 的函数体内， 我们创建了一个名为profile 的空字典， 用于存储用户简介。 在❶处， 我们将名和姓加入到这个字典中， 因为我们总是会从用户那里收到

这两项信息。 在❷处， 我们遍历字典user\_info 中的键—值对， 并将每个键—值对都加入到字典profile 中。 最后， 我们将字典profile 返回给函数调用行。

我们调用build\_profile() ， 向它传递名（'albert' ） 、 姓（'einstein' ） 和两个键—值对（location='princeton' 和field='physics' ） ， 并将返回

的profile 存储在变量user\_profile 中， 再打印这个变量：

## 把函数储存在模块中

模块 是扩展名为.py的文件， 包含要导入到程序中的代码。下面来创建一个包含函数make\_pizza() 的模块。

#pizza.py:

def make\_pizza(size, \*toppings):

"""概述要制作的比萨"""

print("\nMaking a " + str(size) +

"-inch pizza with the following toppings:")

for topping in toppings:

print("- " + topping)

在另一个代码中调用make\_pizza函数：

Import pizza

pizza.make\_pizza(16, 'pepperoni')

pizza.make\_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')

# 第九章 类

类中的函数称为方法

## 创建和使用类：

### 1.\_\_init\_\_()方法：

class Dog():

❷ """一次模拟小狗的简单尝试"""

❸ def \_\_init\_\_(self, name, age):

"""初始化属性name和age"""

❹ self.name = name

self.age = age

❺ def sit(self):

"""模拟小狗被命令时蹲下"""

print(self.name.title() + " is now sitting.")

def roll\_over(self):

"""模拟小狗被命令时打滚"""

print(self.name.title() + " rolled over!")

\_\_init\_\_() 是一个特殊的方法每当使用类创建实例时 python都会自动运行它

形参self 必不可少， 还必须位于其他形参的前面。每个与类相关联的方法调用都自动传递实参self ， 它是一个指向实例本身的引用， 让实例能够访问类中的属性和方法。

### 根据类创建实例

my\_dog = Dog('willie', 6)

print("My dog's name is " + my\_dog.name.title() + ".")

print("My dog is " + str(my\_dog.age) + " years old.")

#输出：

My dog's name is Willie.

My dog is 6 years old.

调用方法：

my\_dog.sit()

my\_dog.roll\_over()

#输出：

Willie is now sitting.

Willie rolled over!

## 2.继承

class Car():

"""一次模拟汽车的简单尝试"""

def \_\_init\_\_(self, make, model, year):

self.make = make

self.model = model

self.year = year

self.odometer\_reading = 0

def get\_descriptive\_name(self):

long\_name = str(self.year) + ' ' + self.make + ' ' + self.modelreturn long\_name.title()

def read\_odometer(self):

print("This car has " + str(self.odometer\_reading) + " miles on it.")

def update\_odometer(self, mileage):

if mileage >= self.odometer\_reading:

self.odometer\_reading = mileage

else:

print("You can't roll back an odometer!")

def increment\_odometer(self, miles):

self.odometer\_reading += miles

class ElectricCar(Car):

"""电动汽车的独特之处"""

❸ def \_\_init\_\_(self, make, model, year):

"""初始化父类的属性"""

super().\_\_init\_\_(make, model, year)

‘’’

super() 是一个特殊函数， 帮助Python将父类和子类关联起来。 这行代码让Python调用ElectricCar 的父类的方法\_\_init\_\_() ， 让ElectricCar 实例包含父类的所有属性。 父类也称为超类 （superclass） ， 名称super因此而得名。

’’’

my\_tesla = ElectricCar('tesla', 'model s', 2016)

print(my\_tesla.get\_descriptive\_name())

创建子类时， 父类必须包含在当前文件中， 且位于子类前面。

定义子类时， 必须在括号内指定父类的名称 如：class ElectricCar(Car):。 方法\_\_init\_\_() 接受创建Car 实例所需的信息

### 给子类定义属性和方法:

class ElectricCar(Car):

"""Represent aspects of a car, specific to electric vehicles."""

def \_\_init\_\_(self, make, model, year):

"""

电动汽车的独特之处

初始化父类的属性， 再初始化电动汽车特有的属性

"""

super().\_\_init\_\_(make, model, year)

❶self.battery\_size = 70

❷def describe\_battery(self):

"""打印一条描述电瓶容量的消息"""

print("This car has a " + str(self.battery\_size) + "-kWh battery.")

在❶处， 我们添加了新属性self.battery\_size ， 并设置其初始值（如70 ） 。 根据ElectricCar 类创建的所有实例都将包含这个属性， 但所有Car 实例都不包含它。 在❷

处， 我们还添加了一个名为describe\_battery() 的方法， 它打印有关电瓶的信息。 我们调用这个方法时， 将看到一条电动汽车特有的描述

### 重写父类的方法:

在子类中重写方法可以覆盖父类中的方法

### 将实例用作属性：

class Car():

--snip--

❶ class Battery():

"""一次模拟电动汽车电瓶的简单尝试"""

❷ def \_\_init\_\_(self, battery\_size=70):

"""初始化电瓶的属性"""

self.battery\_size = battery\_size

❸ def describe\_battery(self):

"""打印一条描述电瓶容量的消息"""

print("This car has a " + str(self.battery\_size) + "-kWh battery.")

class ElectricCar(Car):

"""电动汽车的独特之处"""

def \_\_init\_\_(self, make, model, year):

"""

初始化父类的属性， 再初始化电动汽车特有的属性

"""

super().\_\_init\_\_(make, model, year)

❹ self.battery = Battery() #将一个Battery 实例用作ElectricCar 类的一个属性

my\_tesla = ElectricCar('tesla', 'model s', 2016)

print(my\_tesla.get\_descriptive\_name())

my\_tesla.battery.describe\_battery()