PythonCheatSheet

综述

变量和字符串（Variable and String）

> “变量用来存储具体的值，“字符串”由一连串的字符组成，并用单引号或者双引号括起来。

打印 Hello World

```
print("Hello World!")
```

Hello World 存储在变量里

```
msg = "Hello World!"
print(msg)
```

字符串连接

```
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

列表（Lists）

> 列表将元素按照一定顺序存储起来，用户可以使用索引 index 或者循环 loop 来对元素进行操作。

创建一个列表

```
bikes = ['trek', 'redline', 'giant']
```

获取列表的第一个元素

```
first_bike = bikes[0]
```

获取列表的最后一个元素

```
first_bike = bikes[-1]
```

遍历列表（相当于把列表中所有元素撸一遍）

```
for bike in bikes:
    print(bike)
```

向列表中增加元素

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

创建数值列表

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

列表的推导

```
squares = [x**2 for x in range(1, 11)]
```

列表的切片

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

列表的复制

```
copy_of_bikes = bikes[:]
```

元组（Tuples）

> 元组类似于列表，只不过元组中的元素不能够被操作。

创建一个元组

```
dimensions = (1920, 1080)
```

If语句

> If语句用来判定在不同的情况下执行相应的语句。

条件判断

```
equals x == 42
not equal x != 42
greater than x > 42
  or equal to x >= 42
less than x < 42
  or equal to x <= 42
```

列表条件判断

```
'trek' in bikes
'surly' not in bikes
```

赋予布尔值（Boolean）

```
game_active = True
can_edit = False
```

一个简单的if判定

```
if age >= 18:
    print("You can vote!")
```

If-elif-else语句

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

字典

> 字典中存储有对应关系的成对信息，其中的每一个元素被称为一个“键值”对（key-value）。

一个简单的字典

```
alien = {'color': 'green', 'points': 5}
```

取得该字典的一个值（color的值）

```
print("The alien's color is " + alien['color'])
```

加入一个新的键值（key为x_position，value为0）

```
alien['x_position'] = 0
```

遍历所有的键值（key-value）

```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```

遍历所有的“键”（key）

```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```

遍历所有的“值”（value）

```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite')
```

用户输入

> 程序提示用户需要输入，所有的输入都被存储在字符串中。

提示输入值

```
name = input("What's your name? ")
print("Hello, " + name + "!")
```

while循环

> 当while后的条件为真时，会一直循环执行某段程序。

一个简单的while循环

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

用户决定什么时候退出

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

函数

> 函数式用来实现特定功能的代码段，其中，提供给函数的参数叫做“实参”（argument），函数定义的参数叫做“形参”（parameter）。

一个简单的函数

```
def greet_user():
    """Display a personalized greeting."""
    print("Hello!")
greet_user()
```

传递一个实参（argument）

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")
greet_user('Jesse')
```

形参（parameter）的默认值

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")
make_pizza()
make_pizza('pepperoni')
```

返回一个值sum

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y
sum = add_numbers(3, 5)
print(sum)
```

类

> 类定义了具有相同属性和方法的对象的集合，类的参数信息存储在属性（attribute）中，类中定义的函数叫做方法（method），子类继承了父类中所有的参数和方法。

创建一个名为dog的类

```
class Dog():
    """Represent a dog."""

    def __init__(self,name):
        """Initialize dog object."""
        self.name = name
    def sit(self):
        """Simulate sitting."""
        print(slef.name + " is sitting.")

my_dog = Dog('Peso')

print(my_dog.name + " is a great dog.")
my_dog.sit()
```

继承Inheritance

```
class SARDog(Dog):
    """Represent a search a dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)
    def search (self):
        """Simulate searching."""
        print(self.name + " is searching.")

my_dog = SARDog('Willie')

print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

学以致用

> 如果你有牛叉的编程技能，你会做点什么呢？

当你学习编程时，你一定想结合实际生活来做点学以致用的东西，有些同学平时就有一个好习惯，一旦想起来一些有意思的idea，就记在本子上。如果你暂时还没这个习惯，可以现在就想出几个想做的东西。

文件files操作

> 程序可以读写文件，默认状态下，文件以“r”只读模式打开，但也能以“w”写模式和“a”追加模式打开。

读取文件并按顺序存储

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

向文件中写（覆盖掉之前的内容）

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

向文件中添加（不覆盖之前的内容）

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love programming.")
```

异常处理 Exceptions

> 异常处理帮助你对错误做出适当的响应：首先将需要执行的程序放在try语句后，若程序出现异常，则执行except语句后的代码；若try语句后的代码没有异常，则执行else语句后的代码。

捕获一个异常

```
prompt = "How many tickets do you need?"
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Python 的佛性

> 告别繁文缛节，回归佛性极简主义

如果你有两个方案的代码：简单的和复杂的，并且都能解决问题，那么毫无疑问立马选择简单的。简洁的代码也更方便后期的维护和修改。

列表List

什么是列表?

一系列的元素按照顺序存储在列表中，且没有数量限制。列表是python中对初学者来说最有用的元素类型，而且和程序中的很多概念都紧密相关。

定义一个列表

> 使用方括号[]来定义列表，列表中存储的各个元素用逗号，隔开。最好用通俗易懂的名称来命名列表，让程序更容易阅读。

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

访问列表

> 列表中的每一个元素的位置都会按顺序被分配一个值，这个值叫索引index，第一个元素的index是0，第二个元素的index是1，以此类推。如果index是负数，则表示从列表的尾部倒数的。要获取特定的元素，只需写出列表的名称，然后标出钙元素的索引。

获取第一个元素

```
first_user = users[0]
```

获取第二个元素

```
second_user = users[1]
```

获取最后一个元素

```
last_user = user[-1]
```

更新列表

> 一旦创建了列表，就可以通过个元素项的位置，来改变元素项的值。

更新元素

```
users[0] = 'valerie'
users[-2] = 'ronald'
```

增加元素

> 可以再列表中的任意为止增加元素。

列表尾部添加元素。

```
users.append('amy')
```

像一个空的列表添加元素。

```
users = []
users.append('val')
users.append('bob')
users.append('mia')
```

在列表中指定的位置插入元素。

```
users.insert(0, 'joe')
users.insert(3, 'bea')
```

删除元素

> 可以删除指定位置的元素，或直接指出元素的值来进行删除，若该值在列表中有多个，则删除第一个是该值的元素。

通过位置删除

```
del users[-1]
```

通过具体的值删除

```
users.remove('mia')
```

弹出元素

> 如果想将某个元素从列表中弹出，并返回该元素的值，可以使用pop()函数。可以将列表想象成一叠堆起来的元素，最上面的是原来列表中最后面一个元素。默认情况下，pop函数删除的是列表中最后一个元素，当然你也可以让pop函数删除指定位置的元素，并返回该元素的值。

pop删除列表的最后一个元素

```
most_recent_user = users.pop()
print(most_recent_user)
```

pop删除列表的第一个元素

```
first_user = user.pop(0)
print(first_user)
```

列表长度

> len()函数返回列表的长度(元素个数)。

求列表的长度

```
num_users = len(users)
print("We have " + str(num_users) + " users.")
```

列表排序

> sort()函数会永久改变列表中元素的顺序，sorted()函数会返回一个已经改变顺序的列表副本，原列表顺序依然保存。该函数可以让元素按照字母的顺序进行正向/反向排序。reverse()函数可以颠倒原列表的排列顺序。需要注意的是，小写字母和大写字母会影响排列顺序。

列表排序（按照元素字母的正向顺序排序）

```
users.sort()
```

按照元素字母的反向顺序排序

```
users.sort(reverse=True)
```

返回重新排序的列表

```
print(sorted(users))
print(sorted(users,  reverse=True))
```

反向列表元素

```
users.reverse()
```

遍历列表

> 遍历，就是整个列表撸一遍，循环访问所有的元素。列表中有很多元素，Python提供了高效的循环遍历的方法。当你建立一个循环时，Python一次一个地从列表中拉出每个元素，并将其存储在一个由用户命名的临时变量中，该变量名称应该是列表名称的单数形式。
> 缩进的代码段构成循环的主体，用户可以在循环中处理每一个元素。循环完成后，后面没有在缩进的代码会继续运行。

打印列表元素，user是用户命名的临时变量

```
for user in users:
    print(user)
```

为每一个元素打印一条信息，循环完成后在打印一条单独的信息

```
for user in users:
    print("Welcome, " + user + "!")

print("Welcome, we're glad to see you all!")
```

range()函数

> 你可以使用range()函数有效地处理一组数据。range()函数默认从0开始，在给定数字的前一个位置停止。使用list()函数可以方便的生成数量庞大的列表。

打印 0 到 1000 的数字

```
for number in range(1001):
    print(number)
```

打印 1 到 1000 的数字

```
for number in range(1, 1001):
    print(number)
```

创建一个列表，其元素的值为 1 到 100 万

```
numbers = list(range(1, 1000001))
```

简单的数据统计

> 你可以在包含数字元素的列表上进行一些简单的数据统计。

找出列表中元素的最小值

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
youngest = min(ages)
```

找出最大值

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
youngest = max(ages)
```

计算所有元素的和

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
total_years = sum(ages)
```

列表截取

> 你可以得到列表中的任何元素的集合。列表的一个部分成为一个切片。

截取前三个元素

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
first_three = finishers[:3]
```

截取中间的三个元素

```
middle_three = finishers[1:4]
```

截取最后三个元素

```
last_three = finishers[-3:]
```

复制列表

> 赋值列表其实就是截取整个列表，除此之外还真没有其他更高效的方法。

创建一个列表的副本

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
copy_of_finishers = finishers[:]
```

列表生成式

> 使用循环，可以给予一个列表或一组数字生成一个新的列表。除此之外，还有一种更高效的方法：列表推导式。
> 要使用列表推导式，首先要为存储在列表中的值定义一个表格式，然后编写 for 循环来生成列表所需要的输入值。

使用 for 循环，生成一个平方数的列表

```
squares = []
    for x in range(1, 11):
        square = x ** 2
        squares.append(square)
```

使用列表推导式，生成一个平方数的列表

```
squares = [x**2 for x in range(1, 11)]
```

使用 for 循环，把列表中的小写字母转换成大写字母

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = []
for name in names:
    upper_names = [name.append(name.upper())]
```

使用列表推导式，把列表中的小写字母转换成大写字母

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = [name.upper() for name in names]
```

元组 Tuples

> 元组类似于列表，只是一旦被定义后就不能改变元组中的值。所以在程序中，元组适合存储始终不变的信息。元组由括号而不是方括号来表示。（可以覆盖正噶元祖，但不能更改元组中的单个元素）

定义元组

```
dimensions = (800, 600)
```

元组 for 循环

```
for dimension in dimensions:
    print(dimensions)
```

重写覆盖元组

```
dimensions = (800, 600)
print(dimensions)
dimensions = (1200, 900)
```

代码运行可视化

> 当你第一次学习列表等数据结构时，他可以帮助你想象 Python 如何处理程序中的信息。pythontutor.com 是查看Python 如何跟踪列表中信息的好工具。尝试在 pythontutor.com 上运行以下代码，然后运行你自己的代码。

创建一个列表，打印出列表中的元素。

```
dogs = []
dogs.append('willie')
dogs.append('hootz')
dogs.append('peso')
dogs.append('goblin')

for dog in dogs:
    print("Hello " + dog + "!")
print("I love these dogs!")

print("\nThese were my first two dogs:")
old_dogs = dogs[:2]
for old_dog in old_dogs:
    print(old_dog)

del dogs[0]
dogs.remove('peso')
print(dogs)
```

代码风格

- 一个缩进使用四个空格
- 每行最多79个字符，或者更少
- 使用一个空白行把程序分段，视觉上更易阅读

字典Dictionaries

什么是字典

Python的字典可以让你存储一对有关联的信息。字典中的每条信息都存储为一个键值对key-value。当你提供键（key）时，Python会返回与该键匹配的值（value）。你可以遍历所有的键值对，或所有键，或所有值。

定义字典

> 使用大括号{}来定义字典。使用冒号来连接键和值，并使用逗号分隔开键值对。

创建一个字典

```
alien_0 = {'color': 'green', 'points': 5}
```

访问值

> 要想访问字典中的一个值value，首先给出与值匹配的键的名称，然后把该键放在[]中，如果你想放置的键不在这个字典中，那就会出现错误。
> 也可以使用 get()函数来访问值，如果是不存在的键，该函数会返回一个空值None，或返回一个事先设定的值。

使用 key 来访问value

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

使用 get() 访问 value

```
alien_0 = {'color': 'green'}

alien_color = alien_0.get('color')
alien_points = alien_0.get('points')

print(alien_color)
print(alien_points)
```

添加新的键值对

> 只要电脑内存足够，就可以在字典里存储尽可能多的键值对。如何在字典中添加新的键值对？首先写出字典的名称，然后把新的键 key 放在方括号里，并将其等于新的值 value。
> 如果字典是空的，也可以这样来添加新的键值对。

增加一个键值对

```
alien_0 = {'color': 'green', 'point': 5}
alien_0['x'] = 0
alien_0['y'] = 25
alien_0['speed'] = 1.5
```

向空字典中增加键值对

```
alien_0 = {}
alien_0['color'] = 'green'
alien_0['point'] = 5
```

修改值

> 字典中的任何值都可以进行修改，只需要知道与该值配对的键的名称，然后把这个键放在方括号中，最后赋予新的值。

修改字典里的值

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

# Change the alien's color and point value.
alien_0['color'] = 'yellow'
alien_0['points'] = 10
print(alien_0)
```

删除键值对

> 字典中的任何键值对都可以被删除，使用 del 关键字和字典名称，最后把键的名称放在方括号中，这样就可以删除这个键值对。

删除一个键值对

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)
del alien_0['points']
print(alien_0)
```

字典可视化

登录pythontutor.com，再看看字典如何在程序中运行的。

遍历字典

> 有三种遍历字典的方法：遍历所有键值对，遍历所有键，遍历所有值。
> 字典只能标记处键值对中键与值的匹配关系，而不能标记字典中的键值对顺序。如果想按顺序处理元素，需要在循环中对键进行排序。

遍历所有的键值对

```
# Store people's favorite languages.
fav_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python'
    }

# Show each person's favorite language.
for name, language in fav_languages.items():
    print(name + ": " + language)
```

遍历所有的键

```
# Show everyone who's taken the survey.
for name in fav_languages.key():
    print(name)
```

遍历所有的值

```
# Show all the languages thathave been chosen.
for language in fav_languages.values():
    print(language)
```

按照顺序遍历所有的键

```
# Show each person's favorite language,
# in order by the person's name.
for name in stored(fav_languages.keys()):
    print(name + ": " + language)
```

字典长度

> 使用 len() 函数可以得出字典的长度（键值对的个数）。

求字典长度

```
num_responses = len(fav_languages)
```

嵌套——字典嵌套在列表中

> 在列表中存储一系列字典，成为嵌套 nesting

列表中存储字典

```
# Start with an empty list.
users = []
# Make a new user, and add them to the list.
new_user = {
    'last': 'fermi',
    'first': 'enrico',
    'username': 'efermi',
    }
users.append(new_user)
# Make another new user, and add them as well.
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
    }
users.append(new_user)
# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    priint("\n")
```

不使用 append()，直接定义含字典的列表

```
# Define a list of users, where each user
# is represented by a dictionary.
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]
# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
```

```
    print("\n")
```

嵌套——列表嵌套在字典中

> 在字典中存储一系列列表，可以让一个键与多个值配对。

字典中存储列表

```
# Store multiple languages for each person.
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# Show all responses for each person.
for name, langs in fav_languages.items():
    print(name + ": ")
    for lang in langs:
        print("- " + lang)
```

嵌套——字典嵌套在字典中

> 可以在字典中存储字典，那么一个键就和一个字典匹配。

字典中存储字典

```
users = {
    'aeinstein': {
        'first': 'albert','last': 'einstein',
        'location': 'princeton',},
    'mcurie': {
        'first': 'marie',  'last': 'curie',
        'location': 'paris',},
    }
for username, user_dict in users.items():
    print("\nUsername: " + username)
    full_name = user_dict['first'] + " "
    full_name += user_dict['last']
    location = user_dict['location']
print("\tFull name: " + full_name.title())
print("\tLocation: " + location.title())
```

> 嵌套的使用情况下非常有用，不过嵌套会让程序看起来有点复杂，如果你定义的嵌套比刚才介绍的还要复杂，那你得注意了，有可能有更简单的方法来定义你的数据，比如类(class)。

有序字典 OrderedDict

> 往字典中增加键值时，键值在字典中的位置是随机无序的，字典只保留键值本身的配对关系。如果想保留添加减值的顺序，需要用到 OrderedDict()函数。

保留添加键值的顺序

```
from collections import OrderedDict

# Store each person's languages, keeping
# track of who respoded first.
fav_languages = OrderedDict()

fav_languages['jen'] = ['python', 'ruby']
```

```
fav_languages['sarah'] = ['c']
fav_languages['edward'] = ['ruby', 'go']
fav_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they
# were entered.
for name, langs in fav_languages.items():
    print(name + ":")
    for lang in langs:
        print("- " + lang)
```

创建大量字典

> 如果字典的起始数据类似，可以使用循环很快的创建大量的字典。

大量外星人

```
aliens = []

# Make a million green aliens, worth 5 points
# each. Have them all start in one row.
for alien_num in range(1000000):
    new_alien = {}
    new_alien['color'] = 'green'
    new_alien['points'] = 5
    new_alien['x'] = 20 * alien_num
    new_alien['y'] = 0
    aliens.append(new_alien)

# Prove the list contains a million aliens.
num_aliens = len(aliens)
print("Number of aliens created:")
print(num_aliens)
```

if语句和while循环

什么是if语句，while循环?

if 语句允许你检查程序的当前状态并适当地响应该状态。你可以写一个简单的 if 语句来检查一个条件，或者你可以创建一系列复杂的 if 语句来确定你正在寻找的条件。

只要某些条件保持正确，循环就会运行。若用户需要，可以使用 while 循环让程序一直运行。

条件测试

> 条件测试就是判断是 True 或 False 的表达式。 Python 使用值True 和 False 来决定是否应执行 if 语句中的代码。

判断是否相等

一个=符号可以给变量赋值，双==符号则用来判读两个值是否相等

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

在进行比较时忽略大小写

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

判断是否不相等

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

数值比较

> 既可比较字符串的值是否相等，也可比较数字的值是否相等。

测试是否相等

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

对比符号

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

同时判断多个条件

> 你可以同时判断多个条件。如果列出的所有条件均为真，则 and 运算符将返回 True。如果任何条件为真，则 or 运算符返回 True。

使用 and 判断多个条件

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

使用 or 判断多个条件

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

布尔值

> 布尔值只有两种： True 和 False，带布尔值的变量通常用于跟踪程序中的某些条件。

简单的布尔值

```
game_active = True
can_edit = False
```

If语句

> If 语句有好几种，如何选择呢？首先要看判断条件的数量，条件多的情况下可以反复使用 elif 语句（elif 是 else if 的缩写），有些情况下 else 是不需要的。

一个简单的 If 语句

```
age = 19
if age >= 18:
    print("You're old enough to vote!")
```

if-else 语句

```
age = 17
if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

if-elif-else 语句链

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
else:
    price = 10
print("Your cost is $" + str(price) + ".")
```

列表的条件测试

> 可以测试列表中是否含有某个值，在遍历列表前，也可以测试列表是否是空的。

测试某个值是否在列表中

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```

测试某个值是否在列表中

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'
if user not in banned_users:
    print("You can play!")
```

判断列表是否为空

```
players = []
if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

接收输入

> 在 python3 中， input()函数将所有的输入按照字符串进行处理，并返回一个字符串。

简单的输入

```
name = input("What's your name? ")
print("Hello, " + name + ".")
```

接受数字输入

```
age = input("How old are you? ")
age = int(age)
if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

接受输入（ python2.7）

Python2.7 中，使用 raw_input()函数，同 python3 中的 input() 函数

```
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

While循环

> 只要条件为 True， while 循环中的代码会反复执行。

从 1 算到 5

```
current_number = 1
while current_number <= 5:
    print(current_number)
    current_number += 1
```

用户决定何时退出

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

使用 flag

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)
    if message == 'quit':
        active = False
    else:
        print(message)
```

使用 break 退出当前循环

```
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done. "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

退出循环

在任何 python 循环中，都可以使用 break 语句和 continue 语句，使用 continue 跳出本次循环,使用 break 跳出整个循环。continue 语句用来告诉 Python 跳过当前循环的剩余语句，然后继续进行下一轮循环。

Continue

continue 用在循环中

```
banned_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "

players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

避免无限循环

> while 循环后的条件若为 Ture，程序会一直循环执行不会停止，直到条件为 False，循环才会停止运行。

一个无限的循环

```
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

删除列表中某个值的所有实例

> remove()方法从列表中删除特定值，但它只会删除这个值的第一个实例，所以可以用 while 循环来删除该值的所有实例。

删除 pets 列表中的所有 cat

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

函数Functions

什么是函数?

函数是用于执行一项特定工作的代码块。函数允许你编写一次代码，然后在需要完成相同任务时再次运行。函数可以接收它需要的信息，并返回它生成的信息。使用函数会让你的程序更高效，易于编写，读取，测试和修复。

定义一个函数

> 函数的第一行是它的定义，用关键字 def 标记。该函数的名称后面跟着一组括号和一个冒号。文档字符串描述函数的功能，前后有三个双引号。函数的主体缩进一级。
> 要调用一个函数，给出函数的名字和后面的一组括号。

建立一个函数

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

给函数传递信息

> 传递调用给函数的信息称为实参，实参可以是常量、变量、表达式、函数等. 由函数接收或定义的信息称为形参。实参包含在函数名称后面的圆括号中，形参列在函数定义的圆括号中。

传递单独的实参

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

位置和关键字实参

> 两种主要实参： 位置和关键字实参。当你使用位置实参时，Python 将调用函数中的第一个实参与函数定义中的第一个形参相匹配，以此类推。
> 使用关键字实参，可以在函数调用时，将指定的实参匹配给指定的形参。当你使用关键字实参时，实参的顺序无关紧要。

使用位置实参

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

使用关键字实参

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

默认参数值

> 你可以为形参提供默认值。当函数调用忽略此实参时，将使用此默认值。具有默认值的形参位置必须在函数中没有默认值的形参之后，这样位置参数仍然可以正常工作。

使用默认参数值

```
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

使用 None 值时实参可有可无

```
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")
describe_pet('hamster', 'harry')
describe_pet('snake')
```

函数返回值

> 一个函数可以返回一个值或一组值。 当函数返回一个值时，调用行必须提供一个变量来存储返回值。 函数在到达 return 语句时停止运行。

返回一个值

其中 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

```
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

返回一个字典

```
def build_person(first, last):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

返回一个字典，其中有些值为空

```
def build_person(first, last, age=None):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

传递列表给函数

> 将列表作为参数传递给函数，并且函数可以使用列表中的值。函数对列表所做的任何更改都会影响原始列表。你可以通过传递列表副本作为参数来防止函数修改列表。

将列表作为参数传递

```
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

允许函数修改列表

以下示例将 unprinted 和 printed 列表发送给函数打印。unprinted 列表被清空， printed 列表被填充。

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
# and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

防止函数修改列表
以下示例与前一个示例相同，只是在调用 print_models()后，unprinted 列表未更改。

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs, and print each of them.
original = ['phone case', 'pendant', 'ring'] printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

传递任意数量的实参

> 如果不知道函数要接受多少实参，那么在形参命名时，可以在形参名前加上*符号，表示该形参可以接受多个实参，带有*符号的形参必须放在其他形参的后面。
> ** 操作符允许形参接受多个关键字实参

传递任意数量的实参

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
    'onions', 'extra cheese')
```

传递任意数量的关键字实参

```
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}
    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
# of information.
user_0 = build_profile('albert', 'einstein',
    location='princeton')
user_1 = build_profile('marie', 'curie',
    location='paris', field='chemistry')

print(user_0)
print(user_1)
```

哪种方法写函数最好?

> 正如你所看到的，有很多方法来编写和调用一个函数。当你开始时，瞄准一些简单的工作。随着你获得经验，你将了解不同结构（如位置和关键字参数）以及导入函数的各种方法等更细微的优点。现在，如果你的函数满足你所需要的功能，说明你已经很不错了。

模块化

> 你可以将函数存储在一个单独的文件中，该文件称为模块，若需要使用该函数，那就在主程序中导入该文件。这样会使得程序看起来更干净（确保你的模块和你的主程序存放在同一个目录中）。

在模块 File:pizza.py 中存储程序

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

导入 File:pizza.py

模块中的每个函数在整个程序中都可以起作用

```
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

只导入特定的函数

只有导入的函数才在整个程序中起作用

格式为 from 文件名 inmport 文件中的函数名

```
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

给模块起个别名

as 后是别名，一般别名都是缩写

```
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

导入模块中的所有函数

最好不要这样做，但是当你在别人的代码中看到它时你就会想到，这可能会导致命名冲突，进而产生错误。

```
from pizza import *

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

类Class

什么是类?

> 类是面向对象编程的基础。类代表你想要在程序中建模的真实世界的东西：例如狗，汽车和机器人。你使用一个类来制作对象，这些对象是狗，汽车和机器人的特定实例（比如牧羊犬，大众 polo，和变形金刚）。 类定义了整个类别的对象可以拥有的共同行为以及可以与这些对象相关联的信息。类可以相互继承：你可以编写一个扩展现有类功能的类。这样就可以针对多种情况进行高效的编程。

创建并使用类

Car 类

```
class Car():
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year
```

```
        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

创建一个类的对象

```
my_car = Car('audi', 'a4', 2016)
```

访问属性值

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

调用方法

```
my_car.fill_tank()
my_car.drive()
```

创建多个对象

```
my_car = Car('audi', 'a4', 2016)
my_old_car = Car('subaru', 'outback', 2013)
my_truck = Car('toyota', 'tacoma', 2010)
```

修改属性

> 可以直接修改属性的值，或者编写方法更好的管理属性的值。

直接修改属性的值

```
my_new_car = Car('audi', 'a4', 2016)
my_new_car.fuel_level = 5
```

写一个方法更新属性值

```
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

写一个方法增加属性值

```
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
        <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

命名规则

> 在 Python 中，类名称是用 CamelCase 写的，而对象名称用小写字母和下划线写的。包含类的模块仍应以小写字和下划线命名。

类的继承

> 如果你要写的类是另一个类的制定版本，则可以使用继承。当一个类从另一个类继承时，它会自动接受父类的所有属性和方法。子类可以自由引入新的属性和方法，并覆盖父类的属性和方法。定义新类时，要从另一个类继承，请在括号中写上父类的名称。

子类的_init _()方法

```
class ElectricCar(Car):
    """A simple model of an electric car."""

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attributes specific to electric cars.
        # Battery capacity in kWh.
        self.battery_size = 70
        # Charge level in %.
        self.charge_level = 0
```

向子类中添加新的方法

```
class ElectricCar(Car):
    --snip--
    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

使用子类和父类方法

```
my_ecar = ElectricCar('tesla', 'model s', 2016)
my_ecar.charge()
my_ecar.drive()
```

找到自己的方法

> 有许多方法可以用代码模拟真实世界的对象和情况，有时候方法太多也会让人感到压力。 那就选择一种方法并尝试 - 如果你的第一次尝试不起作用，请尝试不同的方法。

类的继承

覆盖父类

```
class ElectricCar(Car):
    --snip--
    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

实例为属性

Battery 类

```
class Battery():
    """A battery for an electric car."""

    def __init__(self, size=70):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 70:
            return 240
        elif self.size == 85:
            return 270
```

使用实例作为属性

```
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

使用实例

```
my_ecar = ElectricCar('tesla', 'model x', 2016)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

导入类

> 在添加详细信息和函数时，类文件可能会变得很长。 为了保持程序文件的整洁，可以将类先存储在模块中，然后将模块导入到主程序中。

在文件 car.py 中存储类

```
"""Represent gas and electric cars."""

class Car():
    """A simple attempt to model a car."""
    --snip--

class Battery():
    """A battery for an electric car."""
    --snip-

class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

从模块 car.py 中导入各个类

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

导入整个模块

```
import car

my_beetle = car.Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar('tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

导入模块的所有类
（不要这样做，看到的时候知道是什么就可以了）

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2016)
```

Python2.7类

类是从对象继承来的

```
class ClassName(object):
```

Car class

```
class Car(object):
```

子类的*init*()方法是不一样的

```
class ChildClassName(ParentClass):
    def __init__(self):
        super(ClassName, self).__init__()
```

ElectricCar class

```
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(
            make, model, year)
```

列表中存储对象

> 一个列表可以容纳任意数量的元素，因此你可以从一个类中创建大量对象并将它们存储在一个列表中。
> 下面是一个示例，展示如何创建一系列 rental cars，并确保所有car 都已准备好开车。

一系列 rental cars

```
from car import Car, ElectricCar

# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []

# Make 500 gas cars and 250 electric cars.
for _ in range(500):
    car = Car('ford', 'focus', 2016)
    gas_fleet.append(car)
for _ in range(250):
    ecar = ElectricCar('nissan', 'leaf', 2016)
    electric_fleet.append(ecar)

# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print("Gas cars:", len(gas_fleet))
print("Electric cars:", len(electric_fleet))
```

文件和异常

什么是文件？什么是异常？

你的程序可以从文件中读取信息，也可以将数据写入文件。读取文件时，可以访问各种信息，把文本写入文件时，也可以把Python 结构体（如列表）存储在文件中。

异常是帮助程序以适当方式响应错误的特殊对策。例如，如果你的程序尝试打开不存在的文件时，可以使用异常来显示带有提示性的错误消息，而不是让程序崩溃。

读取文件

> 要读取文件，程序需要先打开文件，然后读取文件的内容。你可以一次读取文件的全部内容，或者逐行读取文件。 with 语句确保在程序访问完文件后该文件已正确关闭。

一次读取文件的全部内容

```
filename = 'siddhartha.txt'
with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

逐行读取

从文件中读取的每一行在行尾都有一个换行符，并且打印函数会添加它自己的换行符。 rstrip()方法消除了打印到终端时会产生的额外空白行。

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

把每行内容存储在列表中

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

写文件

> 将'w'参数传递给 open()可以告诉 Python 你想写入文件。注意了，如果文件已经存在，这将删除文件的内容。传递'a'参数告诉 Python 你想添加内容到现有文件的末尾。

向空文件写入

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

向空文件中写入多行内容

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

添加到文件末尾

```
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

文件路径

> 当 Python 运行 open()函数时，执行的程序位于哪个文件夹下， open()函数就在哪个文件夹下查找文件。你可以使用相对路径从子文件夹打开文件。你也可以使用绝对路径打开系统上的任何文件。

从子文件夹中打开文件

```
f_path = "text_files/alice.txt"
with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

使用绝对路径打卡文件

```
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

在 windows 中打开文件

Windows 有时会错误地编译正斜杠。如果遇到这种情况，请在文件路径中使用反斜杠。

```
f_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

try-except 异常处理

> 当你认为可能发生错误时，你可以编写 try-except 程序来处理可能引发的异常。 try 程序告诉 Python 尝试运行的代码，而except 程序告诉 Python 如果代码导致某种特定类型的错误该怎么做。

处理 ZeroDivisionError 异常

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

处理 FileNotFoundError 异常

```
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {0}.".format(f_name)
    print(msg)
```

知道要处理的是什么异常

> 写程序时很难知道处理什么样的异常。先试试在没有 try 的情况下编写代码，并使其生成错误。根据反馈的错误来找到需要处理什么样的异常。