# Single line comments start with a number symbol.

""" Multiline strings can be written

using three "s, and are often used

as comments

"""

**####################################################**

**## 1. Primitive Datatypes and Operators*原始数据类型和运算符***

**####################################################**

# You have numbers

3 # => 3

# Math is what you would expect

1 + 1 # => 2

8 - 1 # => 7

10 \* 2 # => 20

# Except division which returns floats, real numbers, by default默认情况下， 除法返回浮点数，实数

35 / 5 # => 7.0

# Result of integer division truncated down both for positive and negative. 整除的结果被截断为正数和负数

5 // 3 # => 1

5.0 // 3.0 # => 1.0 # works on floats too

-5 // 3 # => -2

-5.0 // 3.0 # => -2.0

# When you use a float, results are floats

3 \* 2.0 # => 6.0

# Modulo operation取模

7 % 3 # => 1

# Exponentiation (x\*\*y, x to the yth power)求幂

2\*\*4 # => 16

# Enforce precedence with parentheses括号优先

(1 + 3) \* 2 # => 8

# Boolean values are primitives (Note: the capitalization)布尔值也是基本数据类型

True

False

# negate with not使用 not 来取反

not True # => False

not False # => True

# Boolean Operators

# Note "and" and "or" are case-sensitive

True and False # => False

False or True # => True

# Note using Bool operators with ints注意使用 整数的 Bool运算符

0 and 2 # => 0

-5 or 0 # => -5

0 == False # => True

2 == True # => False

1 == True # => True

# Equality is == 等式判断用 ==

1 == 1 # => True

2 == 1 # => False

# Inequality is !=

1 != 1 # => False

2 != 1 # => True

# More comparisons

1 < 10 # => True

1 > 10 # => False

2 <= 2 # => True

2 >= 2 # => True

# Comparisons can be chained!居然可以把比较运算串连起来！

1 < 2 < 3 # => True

2 < 3 < 2 # => False

# (is vs. ==) **is** checks if two variables refer to the same object **is**判断指向相同否

# but **==** checks,if the objects pointed to have the same values. **==**判断值

a = [1, 2, 3, 4] # Point a at a new list, [1, 2, 3, 4]

b = a # Point b at what a is pointing to

b is a # => True, a and b refer to the same object

b == a # => True, a's and b's objects are equal

b = [1, 2, 3, 4] # Point b at a new list, [1, 2, 3, 4]

b is a # => False, a and b do not refer to the same object

b == a # => True, a's and b's objects are equal

# Strings are created with " or ' 使用标点 " 或 ' 来创建字符串

"This is a string."

'This is also a string.'

# Strings can be added too! But try not to do this.字符串也可以相加

"Hello " + "world!" # => "Hello world!"

# Strings can be added without using '+'

"Hello " "world!" # => "Hello world!"

# A string can be treated like a list of characters字符串可视为字符的列表

"This is a string"[0] # => 'T'

# You can find the length of a string

len("This is a string") # => 16

# .format can be used to format strings, like this: .format函数用于格式化字符串

"**{}** can be **{}**".format("Strings", "interpolated") # => "Strings can be interpolated"

# You can repeat the formatting arguments to save some typing. 格式化参数

"{**0**} be nimble, {**0**} be quick, {**0**} jump over the {**1**}".format("Jack", "candle stick")

# => "Jack be nimble, Jack be quick, Jack jump over the candle stick"

# You can use keywords if you don't want to count. 格式化密码

"{**name**} wants to eat {**food**}".format(**name=**"Bob", **food=**"lasagna") # => "Bob wants to eat lasagna"

# If your Python 3 code also needs to run on Python 2.5 and below, you can also

# still use the old style of formatting: 通用格式化

"**%s** can be **%s** the **%s** way" **%** ("Strings", "interpolated", "old") # => "Strings can be interpolated the old way"

# None is an object None 是一个对象

None # => None

# Don't use the equality "==" symbol to compare objects to None

# Use "is" instead. This checks for equality of object identity.

#不要使用相等符号 `==` 来把对象和 None 进行比较,而要用 `is`

"etc" is None # => False

None is None # => True

# None, 0, and empty strings/lists/dicts all evaluate to False None、0 以及空字符串和空列表都等于 False

# All other values are True除此以外的所有值都等于 True

bool(0) # => False

bool("") # => False

bool([]) # => False

bool({}) # => False

**####################################################**

**## 2. Variables and Collections*变量和集合***

**####################################################**

# Python has a print function

print("I'm Python. Nice to meet you!") # => I'm Python. Nice to meet you!

# By default the print function also prints out a newline at the end. 默认情况，打印功能也打印出来在最后一个换行符

# Use the optional argument end to change the end character. 使用可选参数end更改结束字符

print("Hello, World", end="!") # => Hello, World!

# Simple way to get input data from console从控制台获取输入数据的简单方法: input()

input\_string\_var = **input(**"Enter some data: "**)** # Returns the data as a **string**

# Note: In earlier versions of Python, input() method was named as raw\_input()

# No need to declare variables before assigning to them.在赋值给变量之前不需要声明(不同于c)

# Convention is to use lower\_case\_with\_underscores变量名的约定是使用下划线分隔的小写单词

some\_var = 5

some\_var # => 5

# Accessing a previously unassigned variable is an exception. 访问一个未赋值的变量会产生一个异常

# See Control Flow to learn more about exception handling. 进一步了解异常处理，可参见下一节《控制流》

some\_unknown\_var # Raises a NameError

# if can be used as an expression if 可以作为表达式来使用

# Equivalent of C's '?:' ternary operator 等价于C中的' ？： '运算符

"yahoo!" if 3 > 2 else "ops" # => "yahoo!"

# Lists store sequences列表用于存储序列

li = [] #初始化列表

# You can start with a prefilled list我们先尝试一个预先填充好的列表

other\_li = [4, 5, 6]

# Add stuff to the end of a list with **append** 使用 append 方法把元素添加到列表的尾部

li.append(1) # li is now [1]

li.append(2) # li is now [1, 2]

li.append(4) # li is now [1, 2, 4]

li.append(3) # li is now [1, 2, 4, 3]

# Remove from the end with **pop** 使用 pop 来移除最后一个元素

li.pop() # => 3 and li is now [1, 2, 4]

# Let's put it back

li.append(3) # li is now [1, 2, 4, 3] again.

# Access a list like you would any array像访问其它语言的数组那样访问列表

li[0] # => 1

# Look at the last element: -1 查询最后一个元素

li[**-1**] # => 3

# Looking out of bounds is an IndexError越界查询会产生一个索引错误(有冒号时貌似不会

li[4] # Raises an IndexError抛出一个索引错误

# You can look at ranges with slice syntax. 你可以使用**切片**语法来查询列表的一个范围

# (It's a closed/open range for you mathy types.) （这个范围相当于数学中的**左闭右开**区间。）

li[1:3] # => [2, 4]

# Omit the beginning省略开头

li[2:] # => [4, 3]

# Omit the end省略结尾

li[:3] # => [1, 2, 4]

# Select every second entry步长为2

li[::2] # =>[1, 4]

# Return a reversed copy of the list反转

li[::-1] # => [3, 4, 2, 1]

# Use any combination of these to make advanced slices

# **li[start:end:step]**

# Make a one layer deep copy using slices使用切片进行一层深层复制

li2 = li[:] # => li2 = [1, 2, 4, 3] but (li2 is li) will result in false.

# Remove arbitrary elements from a list with "**del**"使用 del 来删除列表中的任意元素

del li[2] # li is now [1, 2, 3]

# Remove first occurrence of a value: **remove** 删除**第一次**出现的值

li.remove(2) # li is now [1, 3]

li.remove(2) # Raises a ValueError as 2 is not in the list引发一个 ValueError， 因为2不在列表中

# Insert an element at a specific index: **insert** 在特定**下标/索引**处插入元素

li.insert(1, 2) # li is now [1, 2, 3] again

# Get the index of the first item found matching the argument：**index** 获取与参数匹配的**第一**项的索引/下标

li.index(2) # => 1

li.index(4) # Raises a ValueError as 4 is not in the list

# You can add lists: + 列表相加

# Note: values for li and for other\_li are not modified. li和 other\_li的值不会被修改

li **+** other\_li # => [1, 2, 3, 4, 5, 6]

# Concatenate lists with "**extend()**" 使用 extend 来合并列表

li.extend(other\_li) # Now li is [1, 2, 3, 4, 5, 6]

# Check for existence in a list with "**in**" 用 in 来检查是否存在于某个列表中

1 in li # => True

# Examine the length with "**len()**"用 len 来求列表的长度

len(li) # => 6

# **Tuple**s are like lists but are immutable. **元组**很像列表，但它是“不可变”的

tup = (1, 2, 3)

tup[0] # => 1

tup[0] = 3 # Raises a TypeError

# Note that a tuple of length one has to have a comma after the last element but

# tuples of other lengths, even zero, do not.

#长度为1的元组必须在最后一个元素之后有逗号但是其他长度的元组 ，包括零，也不用加

type((1)) # => <class 'int'>

type((1,)) # => <class 'tuple'>

type(()) # => <class 'tuple'>

# You can do most of the list operations on tuples too可以对元组进行大部分列表的操作

len(tup) # => 3

tup + (4, 5, 6) # => (1, 2, 3, 4, 5, 6)

tup[:2] # => (1, 2)

2 in tup # => True

# You can unpack tuples (or lists) into variables你可以把元组（或列表）中的元素解包赋值给多个变量

a, b, c = (1, 2, 3) # a is now 1, b is now 2 and c is now 3

# You can also do extended unpacking还可以进行扩展解包

a, **\*b**, c = (1, 2, 3, 4) # a is now 1, b is now [2, 3] and c is now 4

# Tuples are created by default if you leave out the parentheses如果你省去了小括号，那么元组会被自动创建

d, e, f = 4, 5, 6

# Now look how easy it is to swap two values再来看看交换两个值是多么简单

e, d = d, e # d is now 5 and e is now 4

# **Dictionaries** store mappings**字典**用于存储映射关系

empty\_dict = {} #初始化

# Here is a prefilled dictionary

filled\_dict = {"one": 1, "two": 2, "three": 3} # **dic={“key”: value}**

# Note **keys** for dictionaries have to be immutable types. This is to ensure that

# the key can be converted to a constant hash value for quick look-ups.

# Immutable types include ints, floats, strings, tuples.

#字典的key必须是不可变类型, 不可变类型包括 整数 ，浮点数，字符串， 元组;

invalid\_dict = {[1,2,3]: "123"} # => Raises a TypeError: unhashable type: 'list' #列表不能作为key

valid\_dict = {(1,2,3):[1,2,3]} # Values can be of any type, however.

# Look up values with **[]** ：使用 [] 来查询键key的值value

filled\_dict["one"] # => 1

# Get all keys as an iterable with "**keys()**". We need to wrap the call in **list()**

# to turn it into a **list**. We'll talk about those later. Note - Dictionary key

# ordering is not guaranteed. Your results might not match this exactly.

＃ 使用“ keys（ ）、values（）” 将 所有key作为可迭代对象获取 ，还要用 list（）包装调用将其变为列表。

#我们稍后会谈到这些。 注 - 字典key顺序不保证。 您的结果可能与此完全不符。

**list(filled\_dict.keys())** # => ["three", "two", "one"]

# Get all values as an iterable with "**values()**". Once again we need to wrap it

# in **list()** to get it out of the iterable. Note - Same as above regarding key ordering.

list(filled\_dict**.values()**) # => [3, 2, 1]

# Check for existence of **key**s in a dictionary with "**in**"

"one" in filled\_dict # => True

1 in filled\_dict # => False

# Looking up a non-existing key is a KeyError查询一个不存在的键名会产生一个键名错误

filled\_dict["four"] # KeyError

# Use "**get()**" method to avoid the KeyError

filled\_dict.get("one") # => 1

filled\_dict.get("four") # => None

# The get method supports a default argument when the value is missing

#get 方法支持传入一个默认值参数，将在取不到值时返回

filled\_dict.get("one", 4) # => 1

filled\_dict.get("four", 4) # => 4

# "**setdefault()**" inserts into a dictionary only if the given key isn't present

#“ setdefault （ ）”仅在**给定键key不存在时**才插入字典

filled\_dict.setdefault("five", 5) # filled\_dict["five"] is set to 5

filled\_dict.setdefault("five", 6) # filled\_dict["five"] is **still** 5

# Adding to a dictionary: **update** 添加键值对到字典的两种方法

filled\_dict**.update({"four":4})** # => {"one": 1, "two": 2, "three": 3, "four": 4}

#filled\_dict**["four"] = 4** #another way to add to dict

# Remove keys from a dictionary with **del** ：删除key

del filled\_dict["one"] # Removes the key "one" from filled dict

# From Python 3.5 you can also use the additional unpacking options Python 3.5中，您还可以使用其他解包选项

{'a': 1, \*\*{'b': 2}} # => {'a': 1, 'b': 2}

{'a': 1, \*\*{'a': 2}} # => {'a': 2}

# Sets store ... well sets set 用于保存**集合**

empty\_set = set()

# Initialize a set with a bunch of values. Yeah, it looks a bit like a dict. Sorry.

some\_set = {1, 1, 2, 2, 3, 4} # some\_set is now {1, 2, 3, 4}

#集合是无序**不重复**的元素集；{} 不会创建一个空集合，只会创建一个空字典

# Similar to keys of a dictionary, elements of a set have to be immutable.

invalid\_set = {[1], 1} # => Raises a TypeError: unhashable type: 'list' #列表不能作为元素

valid\_set = {(1,), 1}

# Can set new variables to a set 可以将新变量设置为集合

filled\_set = some\_set

# Add one more item to the set：**add** 添加元素

filled\_set.add(5) # filled\_set is now {1, 2, 3, 4, 5}

# Do set intersection with & 使用 & 来获取交集

other\_set = {3, 4, 5, 6}

filled\_set & other\_set # => {3, 4, 5}

# Do set union with | 使用 | 来获取并集

filled\_set | other\_set # => {1, 2, 3, 4, 5, 6}

# Do set difference with - 使用 - 来获取补集

{1, 2, 3, 4} - {2, 3, 5} # => {1, 4}

# Do set symmetric difference with ^ 对称差，A并B-A交B

{1, 2, 3, 4} ^ {2, 3, 5} # => {1, 4, 5}

# Check if set on the left is a superset of set on the right: **>=** 检查左侧的设置是否是右侧设置的超集

{1, 2} >= {1, 2, 3} # => False

# Check if set on the left is a subset of set on the right:**<=** 检查左侧的set是否是右侧set的子集

{1, 2} <= {1, 2, 3} # => True

# Check for existence in a set with **in**

2 in filled\_set # => True

10 in filled\_set # => False

**####################################################**

**## 3. Control Flow and Iterables*控制流和可迭代对象***

**####################################################**

# Let's just make a variable

some\_var = 5

# Here is an if statement. Indentation is significant in python! 这是 一个if 语句。 缩进在python中很重要

# prints "some\_var is smaller than 10"

**if** some\_var > 10**:**

print("some\_var is totally bigger than 10.")

**elif** some\_var < 10: # This elif clause is optional. 这个 elif 子句是可选的

print("some\_var is smaller than 10.")

**else:** # This is optional too.

print("some\_var is indeed 10.")

"""

For loops iterate over lists for循环迭代列表

prints:

dog is a mammal

cat is a mammal

mouse is a mammal

"""

**for** animal in **["dog",** "cat", "mouse"]**:**

# You can use format() to interpolate formatted strings您可以使用 format（ ）来插入格式化的字符串

print("{} is a mammal".format(animal))

"""

"range(number)" returns an iterable of numbers

from zero to the given number从 **零**到给定的数字**-1**

prints:

0

1

2

3

"""

for i in **range(4)**:

print(i)

"""

"**range(lower, upper)**" returns an iterable of numbers **左闭右开**

from the lower number to the upper number

prints:

4

5

6

7

"""

for i in range(4, 8):

print(i)

"""

"**range(lower, upper, step)**" returns an iterable of numbers

from the lower number to the upper number, while incrementing

by step. If step is not indicated, the default value is 1.  如果未指示step，则默认值为1

prints:

4

6

"""

for i in range(4, 8, 2):

print(i)

"""

While loops go until a condition is no longer met.

prints:

0

1

2

3

"""

x = 0

**while** x < 4:

print(x)

x += 1 # Shorthand for x = x + 1

# Handle exceptions with a try/except block使用try / except块处理异常

**try:**

# Use "raise" to raise an error使用 raise 来抛出一个错误

**raise** IndexError("This is an index error")

**except** IndexError as e:

pass # Pass is just a no-op. Usually you would do recovery here.

#pass 只是一个空操作。通常你应该在这里做一些恢复工作

except (TypeError, NameError):

pass # Multiple exceptions can be handled together, if required. 可以一起处理多个异常

**else:** # Optional clause to the try/except block. Must follow all except blocks

#Try / except块的可选子句。 必须跟着所有except块之后

print("All good!") # Runs only if the code in try raises no exceptions

**finally:** # Execute under all circumstances在所有情况下都执行

print("We can clean up resources here")

# Instead of try/finally to cleanup resources you can use a with statement

**with** open("myfile.txt") as f:

for line in f:

print(line)

# Python offers a fundamental abstraction called the Iterable.

# An iterable is an object that can be treated as a sequence. 一个 iterable 是一个可以被视为序列的对象

# The object returned the range function, is an iterable.  该 对象返回了range函数，是一个 可迭代的

filled\_dict = {"one": 1, "two": 2, "three": 3}

our\_iterable = filled\_dict.keys()

print(our\_iterable) # => dict\_keys(['one', 'two', 'three']).

#This is an object that implements our Iterable interface.

# We can loop over it. 可以循环

for i in our\_iterable:

print(i) # Prints one, two, three

# However we cannot address elements by index. 不能通过索引来求元素

our\_iterable[1] # Raises a TypeError

# An iterable is an object that knows how to create an iterator. 一个 iterable 可以创建为迭代器

our\_iterator = iter(our\_iterable)#iter用于生成迭代器

# Our iterator is an object that can remember the state as we traverse through it.

# 迭代器是一个在我们遍历它时可以记住状态的对象

# We get the next object with "**next()**".

next(our\_iterator) # => "one"

# It maintains state as we iterate. 在我们迭代时保持状态

next(our\_iterator) # => "two"

next(our\_iterator) # => "three"

# After the iterator has returned all of its data, it gives you a StopIterator Exception

#迭代后所有的数据，它给你一个 StopIterator 异常

next(our\_iterator) # Raises StopIteration

# You can grab all the elements of an iterator by calling **list()** on it. 见前，通过调用list（ ）来获取迭代器的所有元素

list(filled\_dict.keys()) # => Returns ["one", "two", "three"]

**####################################################**

**## 4. Functions函数**

**####################################################**

# Use "def" to create new functions

**def** add(x, y)**:**

print("x is {} and y is {}".format(x, y))

**return** x + y # Return values with a return statement

# Calling functions with parameters

add(5, 6) # => prints out "x is 5 and y is 6" and returns 11

# Another way to call functions is with keyword arguments调用函数的另一种方式是传入关键字参数

add(y=6, x=5) # Keyword arguments can arrive in any order.关键字参数可以以任意顺序传入

# You can define functions that take a variable number of positional arguments

#你可以定义一个函数，并让它接受**可变数量**的**\*定位参数**（调用函数时根据函数定义的**参数位置**来传递参数）

def varargs(**\*args**):

return args

varargs(1, 2, 3) # => (1, 2, 3)

# You can define functions that take a variable number of keyword arguments, as well

#你也可以定义一个函数，并让它接受**可变数量**的**\*\*关键字参数**（函数调用时通过**“键-值”**形式加以指定）

def keyword\_args(**\*\*kwargs**):

return kwargs

# Let's call it to see what happens

keyword\_args(big="foot", loch="ness") # => {"big": "foot", "loch": "ness"}

# You can do both at once, if you like定位参数必须在关键字参数的前面，但关键字参数之间不存在先后顺序的

def all\_the\_args(\*args, \*\*kwargs):

print(args)

print(kwargs)

"""

all\_the\_args(1, 2, a=3, b=4) prints:

(1, 2)

{"a": 3, "b": 4}

"""

# When calling functions, you can do the opposite of args/kwargs!

# Use \* to expand tuples and use \*\* to expand kwargs.

# 在调用函数时，定位参数和关键字参数还可以反过来用。

# 使用 \* 来展开元组，使用 \*\* 来展开关键字参数。

args = (1, 2, 3, 4)

**kwargs = {"a": 3, "b": 4}**

all\_the\_args(\*args) # equivalent to foo(1, 2, 3, 4)

all\_the\_args(**\*\*kwargs**) # equivalent to foo(**a=3, b=4**)

all\_the\_args(\*args, \*\*kwargs) # equivalent to foo(1, 2, 3, 4, a=3, b=4)

# Returning multiple values (with tuple assignments) 将多个值作为**元组**返回

def swap(x, y):

return **y, x** # Return multiple values as a tuple without the parenthesis.将多个值作为元组返回，不带括号

# (Note: parenthesis have been excluded but can be included)

x = 1

y = 2

x, y = swap(x, y) # => x = 2, y = 1

# (x, y) = swap(x,y) # Again parenthesis have been excluded but can be included.

# Function Scope

x = 5

def set\_x(num):

# Local var x not the same as global variable x

x = num # => 43

print (x) # => 43

def set\_global\_x(num):

**global x #在函数内修改全局变量**

print (x) # => 5

x = num # global var x is now set to 6

print (x) # => 6

set\_x(43)

set\_global\_x(6)

# Python has first class functions

#返回函数的函数

def create\_adder(x):

def adder(y):

return x + y

return adder

add\_10 = create\_adder(10)

add\_10(3) # => 13

# There are also anonymous functions匿名函数：**lambda (参数:运算式)(参数的值)**

(lambda x: x > 2)(3) # => True

(lambda x, y: x \*\* 2 + y \*\* 2)(2, 1) # => 5 #

# There are built-in higher order functions内建的高阶函数**list(fun(某函数，对应参数（列表）))**

**#fun：如map(将参数传递给某函数)、 filter(返回True对应的参数)**

list(map(add\_10, [1, 2, 3])) # => [11, 12, 13]

list(map(max, [1, 2, 3], [4, 2, 1])) # => [4, 2, 3]

list(filter(lambda x: x > 5, [3, 4, 5, 6, 7])) # => [6, 7]

# We can use list comprehensions for nice maps and filters我们可以使用列表推导式来代替 map 和 filter

# List comprehension stores the output as a list which can itself be a nested list

#List comprehension将输出存储为列表，该列表本身可以是嵌套列表

[add\_10(i) for i in [1, 2, 3]] # => [11, 12, 13]

[x for x in [3, 4, 5, 6, 7] if x > 5] # => [6, 7]

# You can construct set and dict comprehensions as well. 也可以构建集合和 字典

{x for x in 'abcddeef' if x not in 'abc'} # => {'d', 'e', 'f'}

{x**:** x\*\*2 for x in range(5)} # => {0**:** 0, 1: 1, 2: 4, 3: 9, 4: 16}

**####################################################**

**## 5. Modules模块**

**####################################################**

# You can import modules

import math

print(math.sqrt(16)) # => 4.0

# You can get specific functions from a module

from math import ceil, floor

print(ceil(3.7)) # => 4.0

print(floor(3.7)) # => 3.0

# You can import **all** functions from a module.

# Warning: this is not recommended

from math **import \***

# You can **shorten** module names

import math **as** m

math.sqrt(16) == m.sqrt(16) # => True

# Python modules are just ordinary python files. You

# can write your own, and import them. The name of the

# module is the same as the name of the file.

# You can find out which functions and attributes

# defines a module.你可以查出一个模块里有哪些函数和属性

import math

**dir**(math)

# If you have a Python script named math.py in the same

# folder as your current script, the file math.py will

# be loaded instead of the built-in Python module.

# This happens because the local folder has priority

# over Python's built-in libraries.

# 如果 你有一个名为math.py的Python脚本作为当前的脚本文件夹 ，

#该文件math.py会加载

＃而不是内置的Python模块。

＃ 这 是因为本地文件夹具有优先级

＃ 在 Python的内置库。

**####################################################**

**## 6. Classes类：属性与方法**

**####################################################**

# We use the "class" operator to get a class

**class Human:**

# A class attribute. It is shared by all instances of this class

# 下面是一个类属性。它将被这个类的所有实例共享

species = "H. sapiens"（放在函数之外）

# Basic initializer, this is called when this class is instantiated.构造函数，在实例化此类时调用它

# Note that the double leading and trailing underscores denote objects双前导和尾随下划线表示

# or attributes that are used by python but that live in user-controlled namespaces.

#python使用但属于用户控制的命名空间的对象或属性

#Methods(or objects or attributes) like: \_\_init\_\_, \_\_str\_\_,

# \_\_repr\_\_ etc. are called magic methods (or sometimes called dunder methods)

# You should not invent such names on your own.

**def \_\_init\_\_(self, name):**

# Assign the argument to the instance's name attribute把参数赋值给实例的 name 属性

self.name = name

# Initialize property

self.age = 0

# An instance method. All methods take "self" as the first argument

#下面是一个实例方法。所有方法都以 self 作为第一个参数

def say(self, msg):

print ("{name}: {message}".format(name=self.name, message=msg))

#equal to：return "%s: %s" % (self.name, msg)

# Another instance method

def sing(self):

return 'yo... yo... microphone check... one two... one two...'

# A class **method** is shared among all instances**类方法**会被所有实例共享

# They are called with the calling class as the first argument类方法在调用时，会将类本身作为第一个函数传入

@classmethod

def get\_species**(cls)**:

return cls.species

# A static method is called without a class or instance reference**静态方法**在调用时，不会传入类或实例的引用

@staticmethod

def grunt**():**

return "\*grunt\*"

# A property is just like a getter.

# It turns the method age() into an read-only attribute一个**属性**将方法 age（ ）转换为同名只读属性

# of the same name.

@property

def age(self):

return self.\_age

# This allows the property to be set这允许**设置**属性

@age.setter

def age(self, age):

self.\_age = age

# This allows the property to be deleted这允许删除属性

@age.deleter

def age(self):

del self.\_age

# When a Python interpreter reads a source file it executes all its code.

# This \_\_name\_\_ check makes sure this code block is only executed when this

# module is the main program. \_\_name\_\_检查确保此代码块仅在主程序中执行。

**if \_\_name\_\_ == '\_\_main\_\_':**

# Instantiate a class

i = Human(name="Ian")

i.say(**"**hi**"**) # "Ian: hi"

j = Human("Joel")

j.say("hello") # "Joel: hello"

# i and j are instances of type Human, or in other words: they are Human objects

# Call our class method

i.say(i.get\_species()) # "Ian: H. sapiens"

# Change the shared attribute更改共享属性

Human.species = "H. neanderthalensis"

i.say(i.get\_species()) # => "Ian: H. neanderthalensis"

j.say(j.get\_species()) # => "Joel: H. neanderthalensis"

# Call the static method调用静态方法

print(Human.grunt()) # => "\*grunt\*"

print(i.grunt()) # => "\*grunt\*"

# Update the property for this instance更新此实例的属性

i.age = 42

# Get the property

i.say(i.age) # => 42

j.say(j.age) # => 0

# Delete the property

del i.age

# i.age # => this would raise an AttributeError

**####################################################**

**## 6.1 Multiple Inheritance多重继承（未）**

**####################################################**

# Another class definition

class Bat:

species = 'Baty'

def \_\_init\_\_(self, can\_fly=True):

self.fly = can\_fly

# This class also has a say method

def say(self, msg):

msg = '... ... ...' # msg是什么都会变成省略号

return msg

# And its own method as well

def sonar(self):

return '))) ... ((('

if \_\_name\_\_ == '\_\_main\_\_':

**b = Bat()**

print(b.say('hello'))

print(b.fly)

# from "filename-without-extension" import "function-or-class"

from human import Human

from bat import Bat

# Batman inherits from both Human and Bat Batman继承了Human和Bat

class Batman(**Human, Bat**):

# Batman has its own value for the species class attribute

species = 'Superhero'

**def \_\_init\_\_(self, \*args, \*\*kwargs):**

# Typically to inherit attributes you have to call super: 通常要继承属性，你必须调用super

#super(Batman, self).\_\_init\_\_(\*args, \*\*kwargs)

# However we are dealing with multiple inheritance here, and super() 这里处理多重继承，而 super（）

# only works with the next base class in the MRO list. 仅适用于方法解析顺序MRO列表中的下一个基类

# So instead we explicitly call \_\_init\_\_ for all ancestors. 所以我们明确地 为所有祖先 调用\_\_ init \_\_

# The use of \*args and \*\*kwargs allows for a clean way to pass arguments,

# with each parent "peeling a layer of the onion". 与每个父母“剥一层洋葱”

Human.\_\_init\_\_(self, **'anonymous'**, \*args, \*\*kwargs)

Bat.\_\_init\_\_(self, \*args, **can\_fly=False**, \*\*kwargs)

# override the value for the name attribute覆盖name属性的值

self.name = 'Sad Affleck'

def sing(self):

return 'nan nan nan nan nan batman!'

if \_\_name\_\_ == '\_\_main\_\_':

sup = Batman()

# Instance type checks实例类型检查

if isinstance(sup, Human):

print('I am human')

if isinstance(sup, Bat):

print('I am bat')

if type(sup) is Batman:

print('I am Batman')

# Get the Method Resolution search Order used by both getattr() and super().

＃获取方法解析搜索 getattr （ ）和super（） 使用的顺序

# This attribute is dynamic and can be updated此属性是动态的，可以更新

print(Batman.\_\_mro\_\_) # => (<class '\_\_main\_\_.Batman'>, <class 'human.Human'>, <class 'bat.Bat'>, <class 'object'>)

# Calls parent method but uses its own class attribute调用父方法但使用自己的类属性

print(sup.get\_species()) # => Superhero

# Calls overloaded method调用重载方法，**优先**调用**自己**的

print(sup.sing()) # => nan nan nan nan nan batman!

# Calls method from Human, because inheritance order matters调用say会用Human的，因为**继承顺序先H后B**

sup.say('I agree') # => Sad Affleck: I agree

# Call method that exists only in 2nd ancestor

print(sup.sonar()) # => ))) ... (((

# Inherited class attribute继承的 class属性

sup.age = 100

print(sup.age)

# Inherited attribute from 2nd ancestor whose default value was overridden.

#从第二个祖先得到的的继承 属性，其默认值True被覆盖

print('Can I fly? ' + str(sup.fly))

**####################################################**

**## 7. Advanced高级**

**####################################################**

# Generators help you make lazy code.

def double\_numbers(iterable):

for i in iterable:

yield i + i

# Generators are memory-efficient because they only load the data needed to

# process the next value in the iterable. This allows them to perform

# operations on otherwise prohibitively large value ranges.

# NOTE: `range` replaces `xrange` in Python 3.

for i in double\_numbers(range(1, 900000000)): # `range` is a generator.

print(i)

if i >= 30:

break

# Just as you can create a list comprehension, you can create generator

# comprehensions as well.

values = (-x for x in [1,2,3,4,5])

for x in values:

print(x) # prints -1 -2 -3 -4 -5 to console/terminal

# You can also cast a generator comprehension directly to a list.

values = (-x for x in [1,2,3,4,5])

gen\_to\_list = list(values)

print(gen\_to\_list) # => [-1, -2, -3, -4, -5]

# Decorators

# In this example `beg` wraps `say`. If say\_please is True then it

# will change the returned message.

from functools import wraps

def beg(target\_function):

#@wraps(target\_function)

def wrapper(\*args, \*\*kwargs):

msg, say\_please = target\_function(\*args, \*\*kwargs)

if say\_please:

return "{} {}".format(msg, "Please! I am poor :(")

return msg

return wrapper

@beg #相当于把say传入beg,say=beg(say)

def say(say\_please=False):

msg = "Can you buy me a beer?"

return msg, say\_please

print(say()) # Can you buy me a beer?

print(say(say\_please=True)) # Can you buy me a beer? Please! I am poor :(

**Ready For More?**