*第一行* 规则以 [UNIX "shebang"](https://docs.python.org/zh-cn/3/tutorial/appendix.html#tut-scripts) 开头。第二行设置使用编码，其中 *encoding* 可以是 Python 支持的任意一种 [codecs](https://docs.python.org/zh-cn/3/library/codecs.html#module-codecs)。

*#!/usr/bin/env python3*

*# -\*- coding: utf-8 -\*-*

## 标识符（变量）命名规则

变量是标识符的一个例子。标识符(*Identifiers*)遵守以下规则:

* 第一个字符必须是字母表中的字母(大写 ASCII 字符或小写 ASCII 字符或 Unicode 字 符)或下划线( \_ )。
* 标识符的其它部分可以由字符(大写 ASCII 字符或小写 ASCII 字符或 Unicode 字符)、 下划线( \_ )、数字(0~9)组成。
* 标识符名称区分大小写。

Module常用内置函数/变量：

\_\_version\_\_ = '0.1'

\_\_name\_\_：当前模块名称

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

Main()

else:

print('I am being imported from another module')

**海象运算符:=**

Python 与 C 不同，在表达式内部赋值必须显式地使用 [海象运算符](https://docs.python.org/zh-cn/3/faq/design.html#why-can-t-i-use-an-assignment-in-an-expression) := 来完成。 这避免了 C 程序中常见的一种问题：想要在表达式中写 == 时却写成了 =。

**while** chunk := fp.read(200):

print(chunk)

my\_list = [1,2,3]

count = len(my\_list)

if count > 3:

print(f"Error, {count} is too many items")

# 当转换为海象运算符时...

**if (count := len(my\_list)) > 3:**

print(f"Error, {count} is too many items")

import

**>>> from** **fibo** **import** \*

**>>>** fib(500)

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

Import \*会调入所有非以下划线（\_）开头的名称。 在多数情况下，Python程序员都不会使用这个功能，因为它在解释器中引入了一组未知的名称，而它们很可能会覆盖一些你已经定义过的东西，通常还会导致代码的可读性很差。不过，在交互式编译器中为了节省打字可以这么用。

几种格式化输出的方法：

* 在字符串的开始引号或三引号之前加上一个 f 或 F 。在此字符串中，你可以在 { 和 } 字符之间写可以引用的变量或字面值的 Python表达式。

**>>>** f'Results of the *{*year*}* *{*event*}*'

'Results of the 2016 Referendum'

* 字符串的 [str.format()](https://docs.python.org/zh-cn/3/library/stdtypes.html" \l "str.format" \o "str.format) 方法，仍需使用 { 和 } 来标记变量将被替换的位置，并且可以提供详细的格式化指令，但还需要提供要格式化的信息。

**>>>** percentage = yes\_votes / (yes\_votes + no\_votes)

**>>>** '*{:-9}* YES votes *{:2.2%}*'.format(yes\_votes, percentage)

' 42572654 YES votes 49.67%'

* 最后，还可以使用字符串切片和连接操作自己完成所有的字符串处理，以创建你可以想象的任何布局。字符串类型有一些方法可以执行将字符串填充到给定列宽的有用操作。

## 私有变量

那种仅限从一个对象内部访问的“私有”实例变量在 Python 中并不存在。 但Python 代码都遵循这样一个约定：带有一个下划线的名称 (例如 \_spam) 应该被当作是 API 的非公有部分 (无论它是函数、方法或是数据成员)。

## 系统与交互

系统函数调用：os.system()

def Async(f):

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

thr = Thread(target=f, args=args, kwargs=kwargs)

thr.start()

return wrapper

@Async

def Sound():

os.system('afplay ../study/xiaofang.mp3')

命令参数传递：

import sys

print('The command line arguments are:')

for i in sys.argv:

print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')

dir(sys)

用户输入：

something = input("Enter text: ")

if is\_palindrome(something):

print("Yes, it is a palindrome")

else:

print("No, it is not a palindrome")

## 字符串：

DocStrings：

\_\_doc\_\_:，其中第一行以某一大写字母开始，以句号结束。 第二行为空行，后跟的第三行开始是任何详细的解释说明。强烈建议所有文档字符串中都遵循这一约定。

print(print.\_\_doc\_\_)

字符串格式化方法

有时候我们会想要从其他信息中构建字符串。这正是 format() 方法大有用武之地的地方。

age = 20

name = 'Swaroop'

print('{0} was {1} years old when he wrote this book'.format(name, age))

print('Why is {0} playing with that python?'.format(name))

一个字符串可以使用某些特定的格式(Specification)，随后， format 方法将被调用，使用 这一方法中与之相应的参数替换这些格式。

同时还应注意数字（index）只是一个可选选项（除非打乱顺序），所以你同样可以写成:

print('{} was {} years old when he wrote this book'.format(name, age))

# 对于浮点数 '0.333' 保留小数点(.)后三位

print('{0:.3f}'.format(1.0/3))

# 使用下划线填充文本，并保持文字处于中间位置

# 使用 (^) 定义 '\_\_\_hello\_\_\_'字符串长度为 11

print('{0:\_^11}'.format('hey'))

# 基于关键词（变量名）输出

print('{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python'))

字符串对象 str

name = 'Swaroop'

if name.startswith('Swa'):

print('Yes, the string starts with "Swa"')

if 'a' in name:

print('Yes, it contains the string "a"')

if name.find('war') != -1:

print('Yes, it contains the string "war"')

delimiter = '\_\*\_'

mylist = ['Brazil', 'Russia', 'India', 'China']

print(delimiter.join(mylist))

**Howto（supper helper）: 进入python模式后，**

**dir(str)**

**help(str.supper)**

## 三种控制流语句

if ， while 和 for及其相关的 break 与 continue 语句是 Python 中最常用的部分。

**控制流 if-elif-else**

number = 23

guess = int(input('Enter an integer : '))

if guess == number: # 新块从这里开始

print('Congratulations, you guessed it.') print('(but you do not win any prizes!)') # 新块在这里结束

elif guess < number: # 另一代码块

print('No, it is a little higher than that')

# 你可以在此做任何你希望在该代码块内进行的事情

else:

print('No, it is a little lower than that') # 你必须通过猜测一个大于(>)设置数的数字来到达这里。

print('Done')  
# 这最后一句语句将在if 语句执行完毕后执行。

**while 语句**

while 语句同样可以拥有 else 子句作为可选项。

number = 23

running = True

while running:

guess = int(input('Enter an integer : '))

if guess== 'quit':

break

if len(guess) < 2:

print('Too small')

continue

if guess == number:

print('Congratulations, you guessed it.') # 这将导致 while 循环中止  
running = False

elif guess < number:

print('No, it is a little higher than that.')

else:

print('No, it is a little lower than that.')

else:

print('The while loop is over.') # 在这里你可以做你想做的任何事

print('Done')

**for 循环**

else子语句为可选项

for i in range(1, 5):

print(i)

else:

print('The for loop is over')

**break 语句同样适用于 for 循环。**

**continue 语句同样用于 for 循环。**

## 函数

函数可以通过关键字 def 来定义。这一关键字后跟一个函数的标识符名称，再跟一对圆括号，其中可以包括一些变量的名称，再以冒号结尾，结束这一行。随后而来的语句块是函数的一部分:

Count = 0

def say\_hello(name，age=20):

# 该块属于这一函数

global Count

Count+=1

print('hello world,{}, you are {}, there are {} persons'.format(name,age,Count) # 函数结束

say\_hello() # 调用函数

say\_hello() # 再次调用函数

* global 语句完成全局变量声明。
* 默认参数，通过赋值运算符( = )来为参数指定默认参数值。 带默认值参数应放在最后。
* 关键字参数(*Keyword Arguments*)，支持使用命名(关键字)而非位置(一直以来我们所使用的方式)来指定函数中的参数值

def func(a, b=5, c=10):

print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7) # a is 3 and b is 7 and c is 10

func(25, c=24) # a is 25 and b is 5 and c is 24

func(c=50, a=100) # a is 100 and b is 5 and c is 50

* 可变参数

声明\*param 的星号参数，从此处开始直到结束的所有位置参数 (Positional Arguments)都将被收集并汇集成一个称为“param”的元组(Tuple) 。

声明诸如 \*\*param 的双星号参数时，从此处开始直至结束的所有关键字参数都将被收集并汇集成一个名为 param 的字典(Dictionary) ：

def total(a=5, \*numbers, \*\*phonebook):

print('a', a)

#遍历元组中的所有项目  
for single\_item in numbers:

print('single\_item', single\_item)

#遍历字典中的所有项目  
for first\_part, second\_part in phonebook.items():

print(first\_part,second\_part)

print(total(10,1,2,3,Jack=1123,John=2231,Inge=1560))

* return 语句用于从函数中返回（任何对象），也就是中断函数。
* pass 语句用于指示一个没有内容的语句块。

def some\_function():

'''这里就是著名的DocStrings。

可以多行的吆'''

pass

print(some\_function.\_\_doc\_\_)

## 四种数据结构

Python中有四种内置的数据结构——列表(*List*)、元组(*Tuple*)、字典(*Dictionary*)和集合(*Set*)

项目的列表应该用方括号括起来(a[0])，这样 Python 才能理解到你正在指定一张列表。列表是一种可变的(*Mutable*)数据类型，可以添加、移除或搜索列表中的项目。（append，del，len）

# This is my shopping list

shoplist = ['apple', 'mango', 'carrot', 'banana']

for item in shoplist:

print(item, end=' ')

del shoplist[0]

print(len(shoplist))

元组（tuple）可以近似地看作列表，不如列表功能广泛。元组的一大特征类似于字符串，它们是不可变的，也就是说，你不能编辑或更改元组。

元组通过括号组织元素(非必需)，并在括号内部用逗号进行分隔。 （a,b,c）

元组通常用于保证某一语句或某一用户定义的函数可以安全地采用一组数值，意即元组内的数值不会改变。

# 我会推荐你总是使用括号，尽管括号是一个可选选项。  
# 明了胜过晦涩，显式优于隐式。  
zoo = ('python', 'elephant', 'penguin')

new\_zoo = 'monkey', 'camel', zoo

print('Number of cages in the new zoo is', len(new\_zoo))

print('All animals in new zoo are', new\_zoo)

print('Animals brought from old zoo are', new\_zoo[2])

print('Last animal brought from old zoo is', new\_zoo[2][2])

字典就像一本地址簿，有键值(*Keys*)(即姓名)与值(*Values*)，键值必须是唯一。定义为： d = {key : value1 , key2 : value2}。字典中的数据不会以任何方式进行排序。如果你希望为它们安排一个特别的次序，只能在使用它们之前自行进行排序。

**for key, val in dic.items():**

**if key in dic:**

ab = {

'Swaroop': 'swaroop@swaroopch.com', 'Larry': 'larry@wall.org', 'Matsumoto': 'matz@ruby-lang.org', 'Spammer': 'spammer@hotmail.com'

}

print("Swaroop's address is", ab['Swaroop'])

# 删除一对键值—值配对

del ab['Spammer']

print('\nThere are {} contacts in the address-book\n'.format(len(ab)))

for name, address in ab.items():

print('Contact {} at {}'.format(name, address))

# 添加一对键值—值配对

ab['Guido'] = 'guido@python.org'

if 'Guido' in ab:

print("\nGuido's address is", ab['Guido'])

序列 ，列表、元组和字符串可以看作序列(Sequence)的某种表现形式。

序列的主要功能是资格测试(*Membership Test*)(也就是 in 与 not in 表达式)和索引操作(*Indexing Operations*)，它们能够允许我们直接获取序列中的特定项目。

序列的三种形态——列表、元组与字符串，同样拥有一种切片(*Slicing*)运算 符，允许我们在序列中的某段切片——也就是序列之中的一部分。

Slice切片，索引操作也可以使用负数，表示位置计数将从队列的末尾开始。shoplist[-1] 指的是序列的最后一个项目， shoplist[-2] 将获取序列中倒数第二个项目。

在切片操作中，第一个数字(冒号前面的那位)指的是切片开始的位置，第二个数字(冒号 后面的那位)指的是切片结束的位置。

可以在切片操作中提供第三个参数，这一参数将被视为切片的步长(*Step*)(在默认情况下，步长大小为 1 。

如果第一位数字没有指定，Python 将会从序列的起始处开始操作。如果第二个数字留空，Python 将会在序列的末尾结束操作。要注意的是切片操 作会在开始处返回 *start*，并在 *end* 前面的位置结束工作。也就是说，序列切片将包括起始位置，但不包括结束位置。

shoplist = ['apple', 'mango', 'carrot', 'banana']

name = 'swaroop'

# Indexing or 'Subscription' operation # # 索引或“下标(Subscription)”操作符

# print('Item 0 is', shoplist[0])

print('Item -2 is', shoplist[-2])

# Slicing on a list #

print('Item 1 to 3 is', shoplist[1:3])

print('Item 2 to end is', shoplist[2:])

# 从某一字符串中切片 #  
print('characters 1 to 3 is', name[1:3])

print('characters 2 to end is', name[2:])

集合(Set)是简单对象的无序集合(Collection)。当集合中的项目存在与否比起次序或其出现次数更加重要时，我们就会使用集合。

bri = set(['brazil', 'russia', 'india'])

print('india' in bri)

bric = bri.copy()

bric.add('china')

bric.issuperset(bri)

bri.remove('russia')

## Class

从属于对象或类的变量叫作字段 (**Field**)。对象还可以使用属于类的函数来实现某些功能，这种函数叫作类的方法 (**Method**)。这两个术语很重要，它有助于我们区分函数与变量，哪些是独立的，哪些又是属于类或对象的。总之，字段与方法通称类的属性(**Attribute**)。

字段有两种类型——它们属于某一类的各个实例或对象，或是从属于某一类本身。它们被分别称作实例变量(**Instance Variables**)与类变量(**Class Variables**)。

**self**

类方法与普通函数只有一种特定的区别——前者必须多加一个参数在参数列表开头，这个名 字必须添加到参数列表的开头，但是你不用在你调用这个功能时为这个参数赋值，Python 会为它提供。这种特定的变量引用的是对象本身，按照惯例，它被赋予 self 这一名称。

class Person:

def say\_hi(self):

print('Hello, how are you?')

当调用一个对象的方法，如myobject.method(arg1, arg2) 时，Python 将会自动将其转换成 MyClass.method(myobject, arg1, arg2) ——这就是self 的全部特殊之处所在。这同时意味着，如果你有一个没有参数的方法，你依旧必须拥有一个参数—— self 。

Class的\_\_init\_\_ 方法

\_\_init\_\_ 方法会在类的对象被实例化(Instantiated)时立即运行（可以理解为C++中的类构造函数，但python类不支持重载）。这一方法可以对任何你想进行操作的目标对象进行初始化(*Initialization*)操作。这里你要注意在 init 前后加上的双下划线。

类变量(**Class Variable**)是共享的(Shared)——它们可以被属于该类的所有实例访问。 下例：population = 0,通过@classmethod访问。

对象变量(**Object variable**)由类的每一个独立的对象或实例所拥有。

下例：self.name = name。

class Robot:

"""表示有一个带有名字的机器人。"""

# 一个类变量，用来计数机器人的数量

population = 0

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

"""初始化数据"""

self.name = name  
print("(Initializing {})".format(self.name))

# 当有人被创建时，机器人 # 将会增加人口数量 Robot.population += 1

@classmethod

def how\_many(cls):  
"""类方法，打印出当前的人口数量"""

print("We have {:d} robots.".format(cls.population))

droid1 = Robot("R2-D2")

Robot.how\_many()

**@staticmethod or @classmethod**

一般来说，要使用某个类的方法，需要先实例化一个对象再调用方法。而使用@staticmethod或@classmethod，就可以不需要实例化，直接类名.方法名()来调用。

他们有什么区别呢?

@staticmethod不需要表示自身对象的self和自身类的cls参数，就跟使用函数一样。

@classmethod也不需要self参数，但第一个参数需要是表示自身类的cls参数。

如果在@staticmethod中要调用到这个类的一些属性方法，只能直接类名.属性名或类名.方法名。

而@classmethod因为持有cls参数，可以来调用类的属性，类的方法，实例化对象等，避免硬编码。

class AA(object):

bar = 1

def foo(self):

print('foo')

@staticmethod

def static\_foo():

print('static\_foo')

print(AA.bar)

AA.class\_foo() # can't AA.foo()

"""

AA.foo()

TypeError: foo() missing 1 required positional argument: 'self'

foo()

NameError: name 'foo' is not defined

"""

@classmethod

def class\_foo(cls):

print('class\_foo')

print(cls.bar)

cls().foo()

继承，重用的一种实现方法就是通过继承 (**Inheritance**)机制。Class A（B）：

class SchoolMember: '''代表任何学校里的成员。'''

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

self.name = name

print('(Initialized SchoolMember: {})'.format(self.name))

class Teacher(SchoolMember): '''代表一位老师。'''

def \_\_init\_\_(self, name, age, salary):

SchoolMember.\_\_init\_\_(self, name, age)

self.salary = salary

print('(Initialized Teacher: {})'.format(self.name))

members = [t, s]

for member in members:

member.DoSomething()

类对象toString：重写 \_\_str\_\_

class AA(object):

bar = 1

def foo(self):

print('foo')

def \_\_str\_\_(self):

return "this is a toString"

a = AA()

print(a) #. This is a toString

## 文件操作：

可以通过创建一个file 类的对象，并使用它的open， read 、 readline 、 write 方法来打开或使用文件。最后需要调用 close 方法关闭。

open 函数打开一个文件。打开模式可以是阅读模式( 'r' )，写入模式( 'w' )和追加模式( 'a' )。可以选择是通过文本模式( 't' )还是二进制模式( 'b' )来读取、写入或追加文本。

实际上还有其它更多的模式可用， help(open) 会给你有关它们的更多细节。

# 打开文件以编辑

f = open('poem.txt', 'w', encoding="utf-8")

# 向文件中编写文本

f.write(poem)

# 关闭文件

f.close()

# 如果没有特别指定,将假定启用默认的阅读('r'ead)模式

f = open('poem.txt', encoding="utf-8")

while True:

line = f.readline()

# 零长度指示 EOF  
if len(line) == 0:

Break

# 每行(`line`)的末尾  
# 都已经有了换行符 #因为它是从一个文件中进行读取的

print(line, end='')

# 关闭文件

f.close()

Python 提供了一个叫作 Pickle 的标准模块，通过它你可以将任何纯 Python 对象存储到一 个文件中，并在稍后将其取回。这叫作持久地(*Persistently*)存储对象。

import pickle

# 我们存储相关对象的文件的名称

shoplistfile = 'shoplist.data'

# 需要购买的物品清单

shoplist = ['apple', 'mango', 'carrot']

# 准备写入文件

f = open(shoplistfile, 'wb', encoding="utf-8")

# 转储对象至文件

pickle.dump(shoplist, f)

f.close()

# 重新打开存储文件

f = open(shoplistfile, 'rb', encoding="utf-8") # 从文件中载入对象

storedlist = pickle.load(f)

print(storedlist)

## 异常：except

class ShortInputException(Exception): '''一个由用户定义的异常类'''

def \_\_init\_\_(self, length, atleast):

Exception.\_\_init\_\_(self)

self.length = length

self.atleast = atleast

try:

text = input('Enter something --> ')

if len(text) < 3:

raise ShortInputException(len(text), 3)

except EOFError:

print('Why did you do an EOF on me?')

except KeyboardInterrupt:

print('You cancelled the operation.')

else:

print('You entered {}'.format(text))

finally:

print('Cleaning up: close all file')

## 日志模块

import logging

logging\_file =‘test.log'

logging.basicConfig(level=logging.DEBUG,format='%(asctime)s : %(levelname)s : %(message)s', filename=logging\_file,

filemode='w',

)

logging.debug("Start of the program")

logging.info("Doing something")

logging.warning("Dying now")

Third Lib： loguru

from loguru import logger

logger.add("file\_X.log",retention="1 days",colorize=True, format="<green>{time}</green> <level>{message}</level>")

logger.debug("That's it, beautiful and simple logging!")

logger.info("this is a test")

logger.warning('warn message')

logger.error('error message')

logger.critical('critical message')

## 包

Packages:包是指一个包含模块与一个特殊的 \_\_init\_\_.py 文件的文件夹，后者向 Python 表明这一文件夹是特别的，因为其包含了 Python 模块。

如下结构:

- <some folder present in the sys.path>/

- world/

- \_\_init\_\_.py

- asia/

- \_\_init\_\_.py

- india/

- \_\_init\_\_.py

- foo.py

- africa/

- \_\_init\_\_.py

- madagascar/

- \_\_init\_\_.py

- bar.py

包是一种能够方便地分层组织模块的方式。

## 重要知识点：

特殊方法：

\_\_init\_\_(self, ...) ，这一方法在新创建的对象被返回准备使用时被调用。

\_\_del\_\_(self)，这一方法在对象被删除之前调用(它的使用时机不可预测，所以避免使用它)

\_\_str\_\_(self)，当我们使用 print 函数时，或 str() 被使用时就会被调用。

\_\_lt\_\_(self, other) ，当小于运算符(<)被使用时被调用。类似地，使用其它所有运算符(+、> 等等) 时都会有特殊方法被调用。

\_\_getitem\_\_(self, key) ，使用 x[key] 索引操作时会被调用。

\_\_len\_\_(self) ，当针对序列对象使用内置 len() 函数时会被调用

**Lambda** 表格

发音：['læmdə]，匿名函数（没有函数名），Lambda表达式可以表示[闭包](https://baike.baidu.com/item/%E9%97%AD%E5%8C%85/10908873" \t "_blank)。Python的Lambda表达式的函数体只能有单独的一条语句，也就是返回值表达式语句。其语法如下：**lambda 形参列表 : 函数返回值表达式语句**。从本质上说，lambda 需要一个参数，后跟一个表达式作为函数体，这一表达式执行的值将作为这个新函数的返回值。

points = [{'x': 2, 'y': 3}, {'x': 4, 'y': 1}]

points.sort(key=lambda i: i['y'])

print(points)

assert 语句

assert 语句用以断言(Assert)某事是真的。当语句断言失败时，将会抛出 AssertionError 。

>>> mylist = ['item']

>>> assert len(mylist) >= 1

>>> mylist.pop()

'item'

>>> assert len(mylist) >= 1 Traceback (most recent call last):

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

装饰器:

Decorators是应用包装函数的快捷方式。这有助于将某一功能与一些代码一遍又一遍地“包装”。有点类似java中的切面应用，应用场景可以是打印log等。

from functools import wraps

import logging logging.basicConfig()  
log = logging.getLogger("retry")

def retry(f):

@wraps(f)

def wrapped\_f(\*args, \*\*kwargs):

MAX\_ATTEMPTS = 5

for attempt in range(1, MAX\_ATTEMPTS + 1):

try:

return f(\*args, \*\*kwargs)

except:

log.exception("Attempt %s/%s failed : %s",

attempt,

MAX\_ATTEMPTS,

(args, kwargs))

sleep(10 \* attempt)

log.critical("All %s attempts failed : %s",

return wrapped\_f

counter = 0

@retry

def save\_to\_database(arg):

print("Write to a database or make a network call or etc.")

print("This will be automatically retried if exception is thrown.")

global counter  
counter += 1  
# 这将在第一次调用时抛出异常  
# 在第二次运行时将正常工作(也就是重试)  
if counter < 2:

raise ValueError(arg)

**functool.update\_wrapper**

对象没有**name**和**doc**,对于装饰器函数非常难以debug。update\_wrapper()可以把被封装函数的**name**、module、**doc**和 **dict**都复制到封装函数去(模块级别常量WRAPPER\_ASSIGNMENTS, WRAPPER\_UPDATES)

说白了就是没有加@functools.wraps的函数，装饰器@返回的是函数名是原本的但是其函数签名等内部参数说明却是装饰原函数的那个函数的。

**functool.wraps**对于*update\_wrapper*再次简化，：相当于@functools.wraps(func)这句话代替了update\_wrapper(call\_it, func)，其他的输出结果相同。

def wrap(func):

**@functools.wraps(func)**

def call\_it(\*args, \*\*kwargs):

"""wrap func: call\_it2"""

print('before call')

return func(\*args, \*\*kwargs)

return call\_it

**from..import** 语句:

为了避免每次都要输入module，可以通过使用 from module import function 语句来实现这一点。

from mymodule import \*

from math import sqrt

print("Square root of 16 is", sqrt(16))

## 测试：unittest

import unittest

class TestHello(unittest.TestCase):

@classmethod

def setUpClass(cls):

print("run setUp only once when class load")

@classmethod

def tearDownClass(cls):

print("run tearDown only once when class load")

def setUp(self):

print("setUp 0")

def tearDown(self):

print("tearDown 0")

def test\_hello(self):

a = 1

self.assertEqual(a, 1)

@unittest.skip("I don't want to run this case.")

def test\_world(self):

a = 1

self.assertEqual(a, 2)

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

unittest.main()

unittest中有4个重要的概念：test fixture, test case, test suite, test runner

Testcase：

一个TestCase的实例就是一个测试用例。什么是测试用例呢？就是一个完整的测试流程，包括测试前准备环境的搭建(setUp)，执行测试代码 (run)，以及测试后环境的还原(tearDown)。元测试(unit test)的本质也就在这里，一个测试用例是一个完整的测试单元，通过运行这个测试单元，可以对某一个问题进行验证。测试函数**必须小写test开头**！

Test suite：

多个测试用例集合在一起，就是TestSuite，而且TestSuite也可以嵌套TestSuite。

Test runner：

是来执行测试用例的，其中的run(test)会执行TestSuite/TestCase中的run(result)方法。

TestLoader：

是用来加载TestCase到TestSuite中的，其中有几个loadTestsFrom\_\_()方法，就是从各个地方寻找TestCase，创建它们的实例，然后add到TestSuite中，再返回一个TestSuite实例。

Test fixture：

对一个测试用例环境的搭建和销毁，是一个fixture，通过覆盖 TestCase的setUp()和tearDown()方法来实现。这个有什么用呢？比如说在这个测试用例中需要访问数据库，那么可以在setUp() 中建立数据库连接以及进行一些初始化，在tearDown()中清除在数据库中产生的数据，然后关闭连接。注意tearDown的过程很重要，要为以后的 TestCase留下一个干净的环境。关于fixture，还有一个专门的库函数叫做fixtures，功能更加强大。

unittest.main()调用Testloader加载testcase到Testsuite，再用Testrunner运行testsuite，生成Testresult。从output中可以看出，执行每个testcase之前都会先setUp()初始化，执行完毕之后tearDown()清理环境。

skip装饰器一共有三个:

unittest.skip(reason)、

unittest.skipIf(condition, reason)

unittest.skipUnless(condition, reason)

import unittest

from testhello import TestHello

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

suite = unittest.TestSuite()

tests = [TestHello("test\_hello"), TestHello("test\_world")]

suite.addTests(tests)

runner = unittest.TextTestRunner(verbosity=2)

runner.run(suite)

############### Report.txt

suite = unittest.TestSuite()

suite.addTests(unittest.TestLoader().loadTestsFromTestCase(TestHello) )

with open('UnittestTextReport.txt', 'a') as f:

runner = unittest.TextTestRunner(stream=f, verbosity=2)

runner.run(suite)

命令行界面

unittest 模块可以通过命令行运行模块、类和独立测试方法的测试:

python -m unittest test\_module1 test\_module2

python -m unittest test\_module.TestClass

python -m unittest test\_module.TestClass.test\_method

你可以传入模块名、类或方法名或他们的任意组合。

同样的，测试模块可以通过文件路径指定:

python -m unittest tests/test\_something.py

Sqlite3

import sqlite3

conn = sqlite3.connect('example.db')

c = conn.cursor()

# Create table

c.execute('''CREATE TABLE IF NOT EXISTS stocks(date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data

c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes

conn.commit()

# We can also close the connection if we are done with it.

# Just be sure any changes have been committed or they will be lost.

symbol = 'RHAT'

c.execute("SELECT \* FROM stocks WHERE symbol = '%s'" % symbol)

print("fetchall \n", c.fetchall())

# Do this instead

t = ('RHAT',)

c.execute('SELECT \* FROM stocks WHERE symbol=?', t)

print("fetmany \n",c.fetchmany(3))

# Larger example that inserts many records at a time

purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),

('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),

('2006-04-06', 'SELL', 'IBM', 500, 53.00),

]

c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)

c.execute('SELECT \* FROM stocks')

print("fetchalln",c.fetchall())

conn.close()

## With/Try finally

with 语句适用于对资源进行访问的场合，确保不管使用过程中是否发生异常都会执行必要的“清理”操作，释放资源，比如文件使用后自动关闭、线程中锁的自动获取和释放等。

try/finally 方式操作文件对象

|  |  |
| --- | --- |
|  | somefile = open(r'somefileName')  try:      for line in somefile:          print line          # ...more code  finally:      somefile.close() |

使用 with 语句操作文件对象

|  |  |
| --- | --- |
|  | with open(r'somefileName') as somefile:  for line in somefile:           print line           # ...more code |

这里使用了 with 语句，不管在处理文件过程中是否发生异常，都能保证 with 语句执行完毕后已经关闭了打开的文件句柄。

已经加入对上下文管理协议支持的还有模块 threading、decimal 等。

Web template

## 模板语法

一、变量输出  
{{ ... }}  
可以直接输出render时传过来的变量

二、表达式输出  
输出python表达式，通过AutoEscape设置插入和输出{% %}.

三、注释一个部分，防止他被输出  
{# ... #}.

这些标签可以被转移为{{!, {%!, and {#!  
如果需要包含文字{{, {%, or {# 在输出中使用

四、  
{% apply \*function\* %}...{% end %}  
将函数应用于所有模板之间的输出 apply  
和 end::

{% apply linkify %}{{name}} said: {{message}}{% end %}

Note that as an implementation detail apply blocks are implemented

as nested functions and thus may interact strangely with variables

通过设置 ``{% set %}``, or the use of ``{% break %}`` or ``{% continue %}``

within loops.

五、  
{% autoescape \*function\* %}  
设置当前文件的AutoEscape模式。这不影响  
其他文件，甚至那些引用{% include %}. 请注意，  
autoescaping也能设置全局生效, 或在 .Application  
or Loader.::

{% autoescape xhtml\_escape %}

{% autoescape None %}

六、模板替换  
{% block \*name\* %}...{% end %}  
指定一个可被替换的块 {% extends %}.  
父块的块可以被字块所替换，例如::

<!-- base.html -->

<title>{% block title %}Default title{% end %}</title>

<!-- mypage.html -->

{% extends "base.html" %}

{% block title %}My page title{% end %}

七、模板  
{% comment ... %}  
将模板输出中的注释去除. 当遇到 {% end %} 标签时会结束; 在 comment 到%} 标签之间写参数.

八、模板继承  
{% extends \*filename\* %}  
从另一个模板那里继承过来. extends包含一个或多个标签以从父模块那继承过来 ，不包含在块中的子模板及时存在标签页也会被忽略 , 详见 {% block %} 标签，列 如:

九、for循环  
{% for \*var\* in \*expr\* %}...{% end %}  
这和 python 的for 是一样的。 {% break %} 和  
{% continue %} 语句是可以用于循环体之中的。

十、from引入包  
{% from \*x\* import \*y\* %}  
这和python的 import语法是一样的。

十一、if分支  
{% if \*condition\* %}...{% elif \*condition\* %}...{% else %}...{% end %}  
表达式为真时，第一个条件语句会被输出 (在 elif 和 else之间都是可选的)

十二、import引入包  
{% import \*module\* %}  
和python代码一样的声明 import

十三、引入模板文件  
{% include \*filename\* %}  
包含另一个模板文件，所包含的模板文件可以使用所有的局部变量，如果是直接被 include进来的话(其中 {% autoescape %} 是个例外).  
另外, {% module Template(filename, \*\*kwargs) %} 可以将两个模板的命名空间隔离.

十四、渲染UI模块  
{% module \*expr\* %}  
渲染一个 ~tornado.web.UIModule. The output of the UIModule is  
not escaped::

{% module Template("foo.html", arg=42) %}

``UIModules`` are a feature of the `tornado.web.RequestHandler`

class (and specifically its ``render`` method) and will not work

when the template system is used on its own in other contexts.

十五、不转义输出  
{% raw \*expr\* %}  
输出的结果表达式没有autoescaping

十六、定义变量  
{% set \*x\* = \*y\* %}  
设置局部变量.  
十七、异常处理  
{% try %}...{% except %}...{% else %}...{% finally %}...{% end %}  
这和python try 陈述相同.  
十八、while语句  
{% while \*condition\* %}... {% end %}  
和python语句一样 while 。 {% break %} 和  
{% continue %} 可以在while循环中使用。

{% whitespace \*mode\* %}  
设置当前文件的剩余空白模式

## 性能测量

Python提供了一种性能测量工具。[timeit](https://docs.python.org/zh-cn/3/library/timeit.html" \l "module-timeit" \o "timeit: Measure the execution time of small code snippets.) 模块可以快速演示在运行效率方面一定的优势:

**>>> from** **timeit** **import** Timer

**>>>** Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()

0.57535828626024577

**>>>** Timer('a,b = b,a', 'a=1; b=2').timeit()

0.54962537085770791

与 [timeit](https://docs.python.org/zh-cn/3/library/timeit.html" \l "module-timeit" \o "timeit: Measure the execution time of small code snippets.) 的精细粒度级别相反， [profile](https://docs.python.org/zh-cn/3/library/profile.html#module-profile) 和 [pstats](https://docs.python.org/zh-cn/3/library/profile.html" \l "module-pstats" \o "pstats: Statistics object for use with the profiler.) 模块提供了用于在较大的代码块中识别时间关键部分的工具。

## 十进制浮点运算

[decimal](https://docs.python.org/zh-cn/3/library/decimal.html#module-decimal) 模块提供了一种 [Decimal](https://docs.python.org/zh-cn/3/library/decimal.html#decimal.Decimal) 数据类型用于十进制浮点运算。 相比内置的 [float](https://docs.python.org/zh-cn/3/library/functions.html#float) 二进制浮点实现，该类特别适用于

* 财务应用和其他需要精确十进制表示的用途，
* 控制精度，
* 控制四舍五入以满足法律或监管要求，
* 跟踪有效小数位，或
* 用户期望结果与手工完成的计算相匹配的应用程序。

例如，使用十进制浮点和二进制浮点数计算70美分手机和5％税的总费用，会产生的不同结果。如果结果四舍五入到最接近的分数差异会更大:

**>>> from** **decimal** **import** \*

**>>>** round(Decimal('0.70') \* Decimal('1.05'), 2)

Decimal('0.74')

**>>>** round(.70 \* 1.05, 2)

0.73

精确表示特性使得 [Decimal](https://docs.python.org/zh-cn/3/library/decimal.html#decimal.Decimal) 类能够执行对于二进制浮点数来说不适用的模运算和相等性检测:

**>>>** Decimal('1.00') % Decimal('.10')

Decimal('0.00')

**>>>** 1.00 % 0.10

0.09999999999999995

**>>>** sum([Decimal('0.1')]\*10) == Decimal('1.0')

True

**>>>** sum([0.1]\*10) == 1.0

False

### 行拼接

两个或更多个物理行可使用反斜杠字符 (\) 拼接为一个逻辑行，例如:

**if** 1900 < year < 2100 **and** 1 <= month <= 12 \

**and** 1 <= day <= 31 **and** 0 <= hour < 24 \

**and** 0 <= minute < 60 **and** 0 <= second < 60: *# Looks like a valid date*

**return** 1

以反斜杠结束的行不能带有注释。

圆括号、方括号或花括号以内的表达式允许分成多个物理行，无需使用反斜杠。例如:

month\_names = ['Januari', 'Februari', 'Maart', *# These are the*

'April', 'Mei', 'Juni', *# Dutch names*

'Juli', 'Augustus', 'September', *# for the months*

'Oktober', 'November', 'December'] *# of the year*

隐式的行拼接可以带有注释。后续行的缩进不影响程序结构。后续行也允许为空白行。

Yield

 生成器是可以迭代的，但只可以读取它一次。因为用的时候才生成。

 生成器(generator)能够迭代的关键是它有一个next()方法，工作原理就是通过重复调用next()方法，直到捕获一个异常。

 带有 yield 的函数不再是一个普通函数，而是一个生成器generator。

 yield 是一个类似 return 的关键字，迭代一次遇到yield时就返回yield后面(右边)的值。重点是：下一次迭代时，从上一次迭代遇到的yield后面的代码(下一行)开始执行。

 简要理解：yield就是 return 返回一个值，并且记住这个返回的位置，下次迭代就从这个位置后(下一行)开始。

 send(msg)与next()的区别在于send可以传递参数给yield表达式，就是send可以强行修改上一个yield表达式值。

 第一次调用时必须先next()或send(None)，否则会报错，next()等同于send(None)。

def foo():

print("starting...")

while True:

res = yield 4

print("res:",res)

g = foo()

print(next(g)) # 4

print(next(g)) # res:None\n 4

print("\*"\*20) # \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

print(next(g)) # res:None\n 4

print(g.send(7)) # res:7\n 4

print(next(g)) # res:None\n 4

----------------- run Result ---------------

4

res: None

4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

res: None

4

res: 7

4

res: None

4

## 多线程[¶](https://docs.python.org/zh-cn/3/tutorial/stdlib2.html" \l "multi-threading" \o "永久链接至标题)

线程是一种对于非顺序依赖的多个任务进行解耦的技术。多线程可以提高应用的响应效率，当接收用户输入的同时，保持其他任务在后台运行。一个有关的应用场景是，将 I/O 和计算运行在两个并行的线程中。

**import** **threading**, **zipfile**

**class** **AsyncZip**(threading.Thread):

**def** \_\_init\_\_(self, infile, outfile):

threading.Thread.\_\_init\_\_(self)

self.infile = infile

self.outfile = outfile

**def** run(self):

f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP\_DEFLATED)

f.write(self.infile)

f.close()

print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')

background.start()

print('The main program continues to run in foreground.')

background.join() *# Wait for the background task to finish*

print('Main program waited until background was done.')

通过装饰器实现：

def Async(f):

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

thr = Thread(target=f, args=args, kwargs=kwargs)

thr.start()

return wrapper

@Async

def Sound(filename):

cmd = ""

cmd = "afplay {}".format(filename)

os.system(cmd)

由于GIL（全局解释器锁）的存在，导致Python多线程性能甚至比单线程更糟。于是出现了协程（Coroutine）。

协程

协程: 协程，又称微线程，纤程，英文名Coroutine。协程的作用，是在执行函数A时，可以随时中断，去执行函数B，然后中断继续执行函数A（可以自由切换）。但这一过程并不是函数调用（没有调用语句），这一整个过程看似像多线程，然而协程只有一个线程执行.

协程由于由程序主动控制切换，没有线程切换的开销，所以执行效率极高。对于IO密集型任务非常适用，如果是cpu密集型，推荐多进程+协程的方式。

而Python对协程的支持，是通过Generator实现的，协程是遵循某些规则的生成器。

async def sub\_worker():

print('sub\_worker(): before do sub work')

try:

# Wait for 1 hour

for i in range(200):

print("sub me {}".format(i))

await asyncio.sleep(1)

except asyncio.CancelledError:

print('sub\_worker(): cancel work')

raise

finally:

print('sub\_worker(): after sub\_worker')

async def mainm():

# Create a "cancel\_me" Task

task = asyncio.create\_task(sub\_worker())

# Wait for 1 second

for i in range(20):

await asyncio.sleep(1)

print("mainm {}".format(i))

task.cancel()

try:

await task

except asyncio.CancelledError:

print("main(): sub\_worker is cancelled now")

asyncio.run(mainm())