Python 程序设计

Xia Tian
Email: xiat(at)ruc.edu.cn

Renmin University of China

May 24, 2016

整体内容

- 简介
- 数据类型
- 控制流
- 函数
- 模块
- 标准库
- 面向对象编程
- 异常、调试与测试
- 输入输出
- 应用 (Web，DB，etc.)

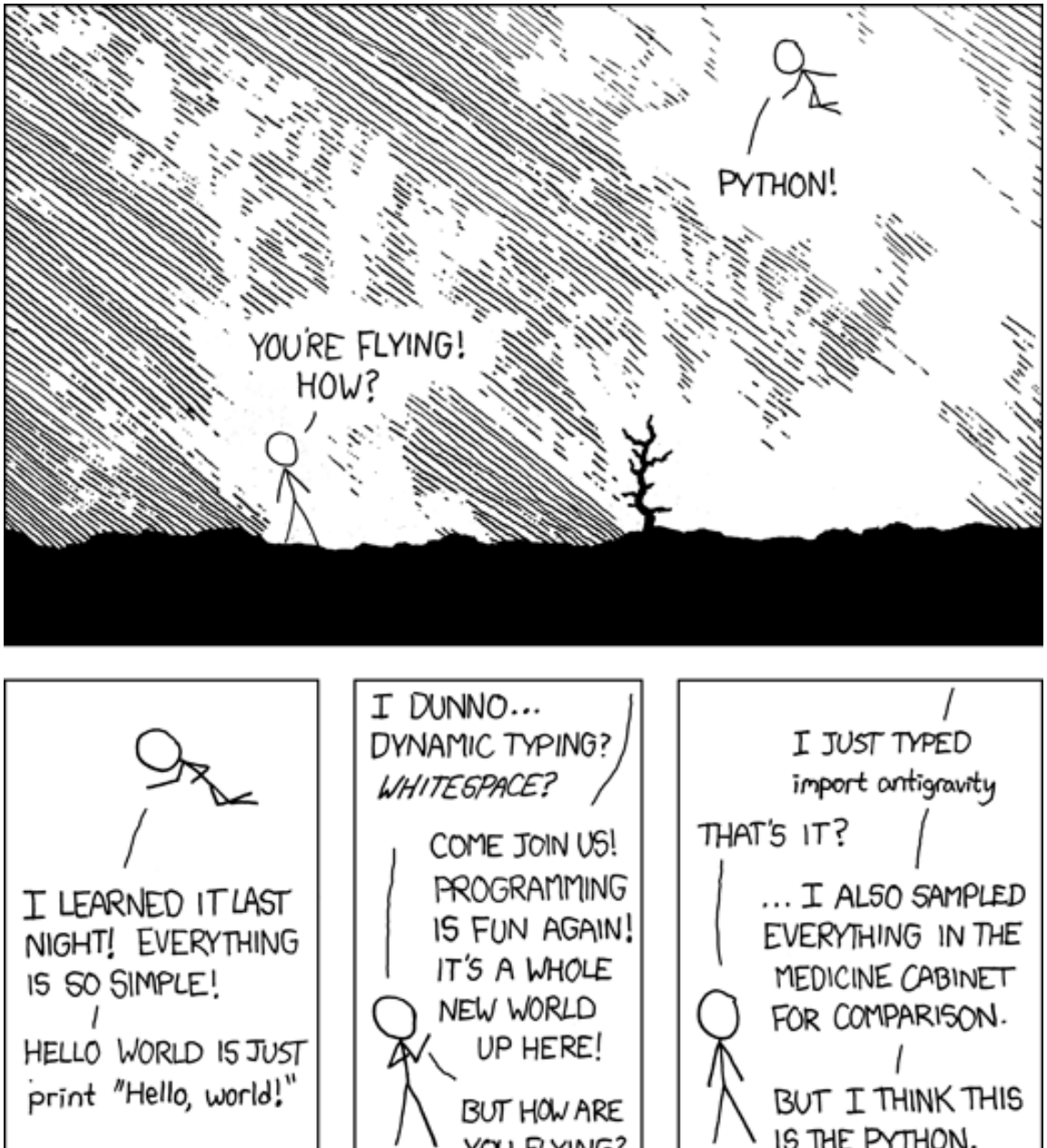
致谢

- 本课件的制作参考了部分图书和第三方网络资源，在此对原作者表达致谢和敬意，如有侵权需要从本课件中移除，请留言或联系 xiat(at)ruc.edu.cn。参考资料包括但不限于以下：
  - 廖雪峰, Python 教程
  - Introducing Python by Bill Lubanovic
  - Dive Into Python 3 by Mark Pilgrim
  - Magnus Lie Hetland, 司维等翻译，Python 基础教程 (2ed), 人民邮电出版社
  - NumPy Cookbook
  - Python for Data Analysis

CH1 简介

- 1.1 什么是 Python
- 1.2 Python 可以做什么
- 1.3 如何安装 Python
- 1.4 Python 开发环境
- 1.5 如何运行 Python 程序
- 1.6 若干例子

You're flying!



用 Python，飞一般的感觉！

- Friend：“你在飞！怎么做到的?”
- Cueball：“Python！我昨晚刚刚学会了 Python。一切都变得如此简单！写一个 Hello World 程序只要一行代码 print ”Hello World!” 就搞定了！”
- Friend：“什么情况？呃……动态类型？泛空格符?”
- Cueball：“来加入我们吧，有了 Python，编程再次变得有趣。这是一个全新的世界！”
- Friend：“但是你到底是怎么飞在天上的?”
- Cueball：“我只是输入了“import antigravity”命令而已。”
- Friend：“就这样?”
- Cueball：“我还把药柜中的药嗑了个遍……但我觉得还是 Python 的原因。”

1.1 什么是 Python

- Python 是一种既简单又强大的编程语言
- 注重如何解决问题，而不是编程语言的语法和结构
- 拥有高效的高级数据结构，简单有效地实现面向对象编程
- 语法简洁、动态解释、适用于快速应用开发和脚本编程
- 在数据科学中大有用武之地

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.

Python 之禅 by Tim Peters

优美胜于丑陋

Python 以编写优美的代码为目标

明了胜于晦涩

优美的代码应当是明了的，命名规范，风格相似

简洁胜于复杂

优美的代码应当是简洁的，不要有复杂的内部实现

复杂胜于凌乱

如果复杂不可避免，那代码间也不能有难懂的关系，要保持接口简洁

扁平胜于嵌套

优美的代码应当是扁平的，不能有太多的嵌套

间隔胜于紧凑

优美的代码有适当的间隔，不要奢望一行代码解决问题

可读性很重要

优美的代码是可读的

即便假借特例的实用性之名，也不可违背这些规则

编程语言排名 @Feb, 2016

- Java
- C
- C++
- C#
- Python ⋆
- PHP
- Visual Basic
- Perl
- JavaScript
- Delphi/Object Pascal

From: `http://www.tiobe.com/tiobe_index?page=index`

1.2 Python 可以做什么

- Almost anything
  - From system management, security, web, to data mining, machine learning ...
- 数据科学界华山论剑：R 与 Python 巅峰对决
  `http://chuansong.me/n/1458679`
- 为什么很多人喜欢 Python?
  `https://www.zhihu.com/question/28676107`

一段简单的 Python 代码

```python
#Python 3: Fibonacci series up to n
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=', ')
        a, b = b, a+b
    print()
fib(100)
```

Result: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,

1.3 如何安装 Python

- 在线直接编写运行
  - Host, run, and code Python in the cloud!
  - https://www.python.org/
- python shell 及增强版本
- 编辑器编辑代码文件，利用 python 运行

Python shell

```
1 $python3
2 Python 3.4.3+ (default, Oct 14 2015, 16:03:50)
3 [GCC 5.2.1 20151010] on linux
4 Type "help", "copyright", "credits" or "license" for more
      information.
5 >>>
```

课堂练习

- Windows 环境下 Python 程序的安装和运行测试
  - https://www.python.org/downloads/
- 注意：本课程后续讲解均以 Linux(Ubuntu) 作为操作系统环境

安装 EasyInstall

- wget␣http://peak.telecommunity.com/dist/ez_setup.py
- python␣ez_setup.py␣setuptools
- ez_install␣pip
- ez_install␣pip3

Python2 vs Python3

- Python3 是 Python2 的重大变更版本
- 有许多正在运行的程序使用了 Python2
- 系统中可以同时安装 python2 和 python3
- 与 Python2 相比，Python3 相关的工具多以 3 结尾，如 ipython3
- 本课程讲解以 Python3 为主，穿插 Python2 的语法

1.4 Python 开发环境

- Python shell
  - 自带的命令解释器
  - ipython
  - jupyter notebook
  - bpython
- Editor
  - pycharm
  - Sublime text
  - Emacs
  - Vim

ipython

- sudo␣apt-get␣install␣ipython3
- 展示

bpython

- sudo␣apt-get␣instal␣bpython3
- 展示

善用 help

利用 Python 自带的 help 函数获取帮助

```
>>> help(max)
```

```
>>> help()
```

增强命令行

- tmux
- zsh

1.5 如何运行 Python 程序

- 直接在 Python Shell 中输入，解释执行
- 保存到文件中，以.py 结尾，通过 python 运行
  - 如把前面的斐波那契数列的代码保存到文本文件中，并把文件名命名为 fib.py
  - python3 fib.py

1.5 如何运行 Python 程序

- 直接在 Python Shell 中输入，解释执行
- 保存到文件中，以.py 结尾，通过 python 运行
  - 如把前面的斐波那契数列的代码保存到文本文件中，并把文件名命名为 fib.py
  - python3␣fib.py
- 可执行的 Python 脚本
  - 在 Python 脚本文件的开头加上一行声明
    - #!/usr/bin/python3
    - chmod u+x filename.py

代码清单：fib.py

```
1  #!/usr/bin/python3
2
3
4  def fib(n):
5      '''
6      Fibonacci series up to n
7      '''
8      a, b = 0, 1
9      while a < n:
10         print(a, end=', ')
11         a, b = b, a+b
12     print()
13
14 if __name__=='__main__':
15     fib(100)
```

1.6 一个完整的 Python 程序 I

```
1  '''Convert file sizes to human-readable form.
2
3  Available functions:
4  approximate_size(size, kb_is_1024_bytes)
5      takes a file size and returns a human-readable string
6
7  Examples:
8  >>> approximate_size(1024)
9  '1.0 KiB'
10 >>> approximate_size(1000, False)
11 '1.0 KB'
12 '''
13
```

1.6 一个完整的 Python 程序 II

```
14 SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB',
       'YB'],
15             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB',
       'YiB']}
16
17
18 def approximate_size(size, kb_is_1024_bytes=True):
19     '''Convert a file size to human-readable form.
20
21     Keyword arguments:
22     size -- file size in bytes
23     kb_is_1024_bytes -- if True (default), use multiples of 1024
24                         if False, use multiples of 1000
25
```

1.6 一个完整的 Python 程序 III

```
26      Returns: string
27
28      '''
29      if size < 0:
30          raise ValueError('number must be non-negative')
31
32      multiple = 1024 if kb_is_1024_bytes else 1000
33      for suffix in SUFFIXES[multiple]:
34          size /= multiple
35          if size < multiple:
36              return '{0:.1f} {1}'.format(size, suffix)
37
38      raise ValueError('number too large')
39
```

1.6 一个完整的 Python 程序 IV

```
40 if __name__ == '__main__':
41     print(approximate_size(1000000000000, False))
42     print(approximate_size(1000000000000))
```

Code From: “Dive into Python 3”

例子解释

- 如何声明函数：def
- 可选的和命名的参数
- 文档字符串
- 代码缩进
- 异常
- 大小写

Python 基本用法预览

- 把 Python 当做计算器
- 字符串的表示
- 字符串切片

把 Python 当做计算器

- Python 解释器可以当做简单的计算器，输入表达式，即可对表达式求值
- 练习
  - 打开 Python 解释器，输入以下表达式求值
  - 2+2
  - 8/5 # 结果是 1.6？还是 1？
  - (3+5)/2
  - 8//5

字符串的表示

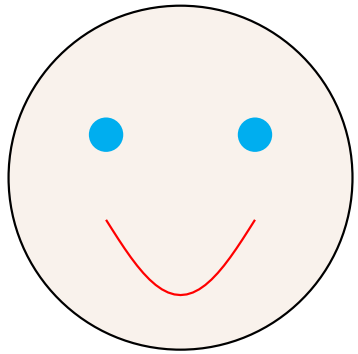
- 单引号
- 双引号
- 三引号

```
1 msg = 'hello "Python"'
2 print(msg)
3 msg = "hello 'Python'"
4 print(msg)
5 msg = '''
6     hello
7     world!
8     '''
9 print(msg)
```

字符串切片

```
>>> msg = '中国人民大学信息资源管理学院'

>>> msg[0]
'中'

>>> msg[-8:-1]
'信息资源管理学'

>>> ms g[-8:]
??

>>> msg[:6]

>>> msg[:]
```

—END—



CH2 数据类型

- 2.1 标识符与关键字
- 2.2 基本数据类型
  - 2.2.1 整数
  - 2.2.2 布尔
  - 2.2.3 浮点
  - 2.2.4 字符
- 2.3 组合数据类型
  - 2.3.1 序列类型：元组与列表
  - 2.3.2 集合
  - 2.3.3 字典（映射）

2.1 标识符与关键字

- 标识符：任意长度的非空字符序列
  - 要求
    - 引导字符：Unicode 编码字母、ASCII 字符、_，不能是数字！
    - 不能是关键字

and	as	assert	break	class
continue	def	del	eif	else
except	False	finally	for	from
global	if	import	in	is
lambda	None	nonlocal	not	or
pass	raise	return	True	try
while	with	yield		

Table: Python 关键字列表

标识符命名约定

- 尽量不要使用 Python 内置函数名与异常名, 如 int, float...
- 避免使用名称的开始和结尾都是下划线
  - 在 Python 解释器中，输入如下语句，观察输出：

    >>> max.__doc__

    注意：前后各是两个下划线”_”

测试

```
>>> hello-world = 1
Error!

>>> 5miles = 5
Error!

>>> int = 5
正常运行，但不建议

>>> hello_world = 'Hello world!'
OK
```

2.2 基本数据类型

- 基本数据类型
  - 整数
  - 布尔
  - 浮点
  - 字符

2.2.1 整数

- 默认为 10 进制形式

  >>> 12345
- 二进制形式：0bxxxx

  >>> 0b1111 (15)
- 八进制形式：0oxxxx

  >>> 0o10 (8)
- 十六进制形式：0xABCD

  >>> 0xFF (255)
- 前导字符大小写均可以，如 0XFF

数值型操作符与函数

语法	描述
$x//y$	整除
$x\%y$	取模
$x**y$	x 的 y 次幂
abs(x)	取绝对值
pow(x, y)	x 的 y 次幂
round(x, n)	四舍五入，n 指小数位保留几位

整数转换函数

语法	描述
bin(i)	返回整数 i 的二进制形式
hex(i)	返回整数 i 的十六进制形式
int(x)	将 x 转换为整数
oct(i)	返回 i 的八进制形式

整数位移操作符

语法	描述
$i \mid j$	逻辑 OR 运算
$i \hat{} j$	XOR
$i \& j$	AND
$i << j$	i 左移 j 位，类似于 $i * (2 * *j)$，但不带溢出检查
$i >> j$	i 右移 j 位，类似于 $i//(2 * *j)$，但不带溢出检查
$\sim i$	反转 i 的每一位

位逻辑操作练习

```
>>> i = 0b1010
>>> j = 0b11000
>>> print(i, j, i|j, i&j, i >> 2, i << 2, ~i)
```

手工计算一下结果是多少？并利用 Python 解释器进行验证，注意，可以利用 bin() 函数，查看结果的二进制形式。

位逻辑操作练习

```
>>> i = 0b1010
>>> j = 0b11000
>>> print(i, j, i|j, i&j, i >> 2, i << 2, ~i)
```

手工计算一下结果是多少？并利用 Python 解释器进行验证，注意，可以利用 bin() 函数，查看结果的二进制形式。

输出结果：10 24 26 8 2 40 -11

位逻辑操作练习

```
>>> i = 0b1010
>>> j = 0b11000
>>> print(i, j, i|j, i&j, i >> 2, i << 2, ~i)
```

手工计算一下结果是多少？并利用 Python 解释器进行验证，注意，可以利用 bin() 函数，查看结果的二进制形式。

输出结果：10 24 26 8 2 40 -11

- 计算机内部通常采用补码表示整数，补码对于正数就是本身，负数补码等于源码的反码加一，正数取反后在数值上体现为 $-|x+1|$，负数的是 $|x|-1$。
- 如果要表示多个布尔值，可以利用一个整数进行表示，如文件的属性有“读、写和执行”三种，可以用一个三位的二进制数字表示。

原码 → 反码 → 补码 I

- 无符号数字：只有正数，没有负数的概念。

十进制	二进制
0	0000
1	0001
2	0010
3	0011
4	0100
⋮	⋮

原码 → 反码 → 补码 II

- 神说，要有光，就有了光，要有正数和负数，就有了正数和负数
  - 为表示正数和负数，人们发明了“原码”：
    - 左边第一位存放符号，正用 0 来表示，负用 1 来表示

	正数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

	负数
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

原码 $\rightarrow$ 反码 $\rightarrow$ 补码 III

- 易于理解，但不利于计算机处理
  - $1 + (-1) = 0001_b + 1001_b = 1010_b = -2 =$
  - 存在正 0(0000_b) 和负 0(1000_b)
  - 正负相加不等于 0

原码 → 反码 → 补码 IV

- 反码
  - 用来处理负数的，符号位置不变，其余位置相反

正数：

	正数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

反码：

	负数
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000

原码：

	负数
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

原码 $\rightarrow$ 反码 $\rightarrow$ 补码 V

- 原码变成反码后，解决了正负相加等于 0 的问题
- 过去的 $+1$ 和 -1 相加，变成了 $0001_b + 1101_b = 1111_b$，刚好反码表示方式中，1111_b 象征 -0
- 新问题：存在两个零，$+0$ 和 -0
- 从原来”反码”的基础上，补充一个新的代码，负数加 1，因此，-0 由 1111_b 变为 10000_b，舍去溢出的高位，变为 0000_b

原码 → 反码 → 补码 VI

正数：

	正数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

补码：

	负数
-0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

反码：

	负数
-0	1111
-1	1110
-2	1101
-3	1100
-4	1011
-5	1010
-6	1001
-7	1000

原码：

	负数
-0	1000
-1	1001
-2	1010
-3	1011
-4	1100
-5	1101
-6	1110
-7	1111

原码 → 反码 → 补码 VII

- 解决了 $+0$ 和 -0 同时存在的问题
- 另外” 正负数相加等于 0” 的问题
- 多了一个 -8

2.2.2 布尔类型

基本用法：

```
>>> t=True
>>> f = False
>>> t and f
False
>>> t and True
True
```

2.2.2 布尔类型

基本用法：

```
>>> t=True
>>> f = False
>>> t and f
False
>>> t and True
True
```

布尔类型可以看作是一种特殊的整数类型，False 为 0，True 为 1, 0 为 False, 非 0 值为 True，例如：

```
>>> False + 1
1
>>> True - 2
-1
```

2.2.3 浮点类型

float: 计算机以二进制表示浮点数，是近似表示

```
>>> 0.0, 5.4, 1e-2
(0.0, 5.4, 0.01)

>>> x = 5.59
>>> int(x)
5
>>> s=x.hex()
'0x1.65c28f5c28f5cp+2', 此处指数以 p 表示，而不是 e，why?
>>> y=float.fromhex(s)
```

其他数字类型及操作

- 复数

  >>> z = 1 + 2j

- 扩展包
  - math

    >>> import math

    >>> math.pi

    >>> math._____

  - decimal: 精度可控

```
1 import decimal
2 a = decimal.Decimal(1234)
3 b = decimal.Decimal("1234567890000.1234567")
4 a + b
```

Result: Decimal('1234567891234.1234567')

2.2.4 字符串

- 可以使用单引号或双引号，但两端必须相同
- 三个引号包含的字符串
  - 可以直接在里面使用换行，而不需要进行转义处理
- 转义符
  - 用于处理特殊字符，如回车、换行、引号等
    - \r, \t, \n, \', \", ...
    - \xhh: 对应十六进制数字的字符
  - 例子

```
>>> s = 'hello\x20world\n\tI\'m fine.'
>>> print(s)
```

2.2.4 字符串

- 可以使用单引号或双引号，但两端必须相同
- 三个引号包含的字符串
  - 可以直接在里面使用换行，而不需要进行转义处理
- 转义符
  - 用于处理特殊字符，如回车、换行、引号等
    - \r, \t, \n, \', \", ...
    - \xhh: 对应十六进制数字的字符
  - 例子

```
>>> s = 'hello\x20world\n\tI\'m fine.'
>>> print(s)
```

输出结果:

```
hello world
    I'm fine.
```

常见函数

- Python 通过 ord 和 chr 两个内置函数, 用于字符与 ASCII 码之间的转换
  - ord() 函数
    求特定字符的 Unicode 编码，以十进制表示返回结果
  - chr() 函数
    返回整数所对应的字符

```
>>> ord('A')
65
>>> hex(ord('A'))
'0x41'
>>> hex(ord('中'))
'0x4e2d'
>>> chr(0x4e2d)
'中'
```

字符串比较

内部安装字符串的 ASCII 码值进行比较

```
>>> "RUC" > "IRM"
True

>>> "China" > "Canda"
True
```

字符串分片与步距

s[-9]	s[-8]	s[-7]	s[-6]	s[-5]	s[-4]	s[-3]	s[-2]	s[-1]
W	h	o		a	m		I	?
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]

- 语法
  - s[start]
  - s[start:end]
  - s[start:end:step]

```
>>> s = 'Who am I?'
>>> s[0:6:2], s[0:3]
'Woa', 'Who'
```

字符串步距

步距默认为 1，但可以为负

```
>>> s[-1:]
'?'

>>> s[-1::-1]
'?I ma ohW'
```

字符串操作方法 I

语法	描述
s.capitalize()	返回 s 的副本，并将首字母变为大小
s.center(width,char)	返回 s 中间部分的一个子字符串，长度为 width，并使用空格或可选的 char（长度为 1 的字符串）进行填充。参考 str.ljust()、str.rjust() 与 str.format()
s.count(t,start,end)	返回字符串 s 中（或在 s 的 start:end 分片中）子字符串 t 出现的次数
s.encode(encoding,err)	返回一个 bytes 对象，该对象使用默认的编码格式或制定的编码格式来表示该字符串
s.endswith(x,start,end)	如果 s（或在 s 的 start:end 分片中）一字符串 x（或元组中的任意字符串）结尾，就返回 true，否则返回 False，参考 str.find()
s.expandtabs(size)	返回 s 的一个副本，其中的制表符使用 8 个货指定数量的空格替换
s.find(t,star,end)	返回 t 在 s 中（或在 s 的 start：end 分片中）的最左位置，如果没有找到，就返回-1；使用 str.rfind() 则可以发现相应的最右边位置：参考 str.index()

字符串操作方法 II

s.formal(...)	返回按给定参数进行格式化后的字符串副本，这一方法及其参数将在下一小节进行讲解
s.index(t,start,end)	返回 t 在 s 中的最左边位置（或在 s 的 start:end 分片中），如果没有找到，就产生 ValuError 异常。使用 sr.rindex() 可以从右面开始搜索。参见 str.find()
s.isalnum()	如果 s 非空，并且其中的每个字符串都是字母数字的，就返回 True
s.isalpha()	如果 s 非空，并且其中每个字符都是字母的，就返回 Ture
s.isdecimal()	如果 s 非空，并且其中的每个字符都是 Uicode 的基数为 10 的数字，就返回 True，
s.isdigit()	如果 s 非空，并且每个字符都是一个 ASCII 数字，就返回 Ture
s.isidentifier()	如果 s 非空，并且是一个邮箱的标识符，就返回 Ture
s.islower()	如果 s 至少有一个可小写的字符，并且所有可小写的字符都是小写的，就返回 True，参见 str.isupper()
s.isnumeric()	如果 s 非空，并且其中的每个字符都是数值型的 Uicode 字符，比如数字或小数，就返回 True
s.isprintable()	如果 s 非空，并且其中的每一个字符被认为死可打印的，包括空格，但不包括换行，就返回 True

字符串操作方法 III

s.isspace()	如果 s 非空，并且其中的每一个字符都是空白字符，就返回 Ture
s.istitle()	如果 s 是一个非空的首字母大写的字符串，就返回 True，参见 str.title()
s.isupper()	如果 s 至少有一个可大写的字符，并且所有可大写的字符都是大写的，就返回 True，参见 str.islower()
s.join(seq)	返回序列 seq 中每个项连接起来的后果，并以 s（可以为空）在每两项之间分隔
s.ljust(width,char)	返回长度为 width 的字符串（使用空格或可选的 char（长度为 1 的字符串）进行填充）中左对齐的字符串 s 的一个副本，使用 str.rjust() 可以右对齐，str.center() 可以中间对齐，参考 str.format()
s.lower()	将 s 中的字符变为小写，参见 str.upper()
s.maketrans()	与 str.translate() 类似，参见正文谅解详细资料
s.partition(t)	返回包含 3 个字符串的元组————字符串 s 在 t 的最左边之前的部分、t、字符串 s 在 t 之后的部分。如果 t 不在 s 内，则返回 s 与两个空字符串。使用 str.rpartition() 可以在 t 最右边部分进行分区

字符串操作方法 IV

s.replace(t,u,n)	返回 s 的一个副本，其中每个（或最多 n 个，如果给定）字符串 t 使用 u 替换
s.split(t,n)	返回一个字符串列表。要求在字符串 t 处至多分割 n 次，如果没有给定 n，就分割尽可能多次，如果 t 没有给定，就在空白处分割。使用 str.rsplit() 可以从右边进行分割——只有在给定 n 并且 n 小于可能分割的最大次数时才能起作用
s.splitlines(f)	返回在行终结符处进行分割产生的列表，并剥离行终结符（除非 f 为 True）
s.starswith(x,start,end)	如果 s（或 s 的 start:end 分片）以字符串 x 开始（或以元组 x 中的任意字符串开始），就返回 True，否则返回 False，参考 str.endswith()
s.strip(chars)	返回 s 的一个副本，并将开始处与结尾处的空白字符（或字符串 chars 中的字符）移除，str.lstrip() 仅剥离起始处的相应字符，str.rstrip() 只剥离结尾处的相应字符
s.swapcase()	返回 s 的辅副本，并将其中大写字符变为小写，小写字符变大写，参考 str.lower() 与 str.upper()
s.title()	返回 s 的副本，并将每个单词的首字母变为大写，其他字母都变为小写，参考 str.istitle()

字符串操作方法 V

s.translate()	与 str.maketrans() 类似，参考正文了解详细资料
s.upper()	返回 s 的大写化版本，参考 str.lower()
s.zfill(w)	返回 s 的副本，如果比 w 短，就在开始处添加 0，使其长度为 w

字符串操作方法 VI

join 函数

```
>>> countries=["China", "USA", "Japan"]

>>> " ".join(countries)
'China USA Japan'

>>> " <-> ".join(countries)
'China <-> USA <-> Japan'
```

字符串复制 *

```
>>> s = 'ab' * 3
'ababab'

>>> s *= 2
'abababababab'
```

format() 函数

```
>>> '{0} is {1} years old.'.format("Tom", 5)
'Tom is 5 years old.'
```

2.3 组合数据类型

- 列表
- 元组
- 字典
- 集合

列表

- list 是一种有序的集合，可以随时添加和删除其中的元素

```
>>> classmates = ['Michael', 'Bob', 'Tracy']

>>> classmates
['Michael', 'Bob', 'Tracy']
```

列表——索引

- 用索引来访问 list 中每一个位置的元素，索引是从 0 开始的，最后一个元素的索引是 −1

```
>>> classmates[0]
'Michael'

>>> classmates[1]
'Bob'

>>> classmates[-1]
'Tracy'

>>> classmates[-2]
'Bob'
```

列表——追加

- list 是一个可变的有序表，可以往 list 中追加元素到末尾

```
>>> classmates.append('Adam')

>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
```

- 可以把元素插入到指定的位置，比如索引号为 1 的位置

```
>>> classmates.insert(1, 'Jack')

>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
```

列表——删除

- 用 pop() 方法删除 list 末尾的元素

```
>>> classmates.pop()
'Adam'
```

```
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
```

- 用 pop(i) 方法删除指定位置的元素

```
>>> classmates.pop(1)
'Jack'
```

```
>>> classmates
['Michael', 'Bob', 'Tracy']
```

列表——替换

- 赋值给对应的索引位置把某个元素替换成别的元素

```
>>> classmates[1] = 'Sarah'

>>> classmates
['Michael', 'Sarah', 'Tracy']
```

列表——数据类型

- list 里面的元素的数据类型可以不同

```
>>> L = ['Apple', 123, True]
```

- list 元素也可以是另一个 list

```
>>> p = ['asp', 'php']
>>> s = ['python', 'java', p, 'scheme']
>>> s[2][1]
'php'
```

元组

- 元组 tuple 是一种有序的集合，一旦初始化就不能修改，使代码更安全

```
>>> classmates = ('Michael', 'Bob', 'Tracy')
```

元组——属性

- tuple 不能改变，不可以追加、删除、替换

```
>>> t = (1, 2)

>>> t
(1, 2)
```

tuple 的“可变性”

```
>>> t = ('a', 'b', ['A', 'B'])

>>> t[2][0] = 'X'

>>> t[2][1] = 'Y'

>>> t
('a', 'b', ['X', 'Y'])
```

字典

- 字典是一种可变的无序的数据组合类型，使用键-值（key-value）储存，进行极快的查找

- 键必须是唯一的，必须是可哈希的

```
>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
>>> d['Michael']
95
```

字典——原理

- 给定一个名字，比如'Michael'，dict 在内部就可以直接计算出 Michael 对应的存放成绩的“页码”
- 也就是 95 这个数字存放的内存地址，直接取出来，所以速度非常快
- 在放进去的时候，必须根据 key 算出 value 的存放位置，取的时候才能根据 key 直接拿到 value

字典——对应性

- 一个 key 只能对应一个 value，多次对一个 key 放入 value，后面的值会把前面的值冲掉

```
>>> >>> d['Jack'] = 90

>>> d['Jack']
90

>>> >>> d['Jack'] = 88

>>> d['Jack']
88
```

字典——更新和删除

- 字典是可变的，所以可以用 update 进行更新

```
>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
>>> d1 = {'Jack': 80}
>>> d.updata(d1)
>>> d {'Jack' 80,' Michael' : 95,' Bob' : 75,' Tracy' : 85}
```

- 如果要删除一个 key，用 pop(key) 方法

```
>>> d.pop('Bob')
75
>>> d
{'Jack' 80,' Michael' : 95,' Tracy' : 85}
```

集合

- 集合是一个无序的、不重复的元素集，包含两种类型：可变集合（set）和不可变集合（frozen set）

(1) 可变集合（set）：一组 key 的集合，不储存 value，可添加和删除元素，非可哈希的，不能用作字典的键，也不能做其他集合的元素

```
>>> s = set([1, 1, 2, 2, 3, 3])

>>> s
1, 2, 3
```

(2) 不可变集合（frozenset）：不可添加和删除元素，可哈希的，能用作字典的键，能做其他集合的元素

集合——添加和删除

- set 集合是可变的，可以用 add(key) 进行添加

```
>>> s.add(4)

>>> s
```

1, 2, 3, 4

- 如果要删除一个 key，用 remove(key) 方法

```
>>> s.remove(4)

>>> s
```

1, 2, 3

集合——交集和并集

- 两个 set 可以做数学意义上的交集、并集等操作

```
>>> s1 = set([1, 2, 3])
>>> s2 = set([2, 3, 4])
>>> s1 & s2
2, 3
>>> s1 | s2
1, 2, 3, 4
```

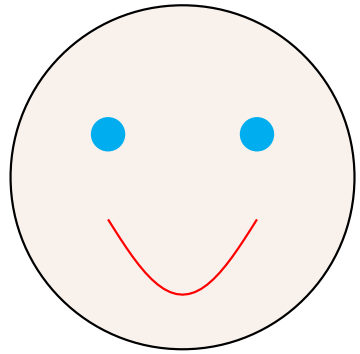
练习

- 给定一个字符串变量 s，编写 Python 代码统计字符串中每个字符的出现频度。
  - 例如：

    $s =' hello'$，则输出：

    e:1, h:1, l:2, o:1

```
1 s = 'hello world'
2 d = {}
3 for ch in s:
4     if d.get(ch)==None:
5         d[ch] = 1
6     else:
7         d[ch] += 1
8
```



—END—

CH3 控制流

- if
- while
- for
- break
- continue

补充：函数的简单定义

- 函数可通过 def 来定义，语法：

```
def function_name(param1, param2, ...):
    function_suite

```

- 例子

```
def test(name, age):
    print("{0} is {1} years old".format(name, age))

test('Mike', 20) #test function

```

- 函数的详细内容下次课讲解

if

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
elif boolean_expression3:
    suite...
else:
    suite...

```

- 可以有 0 到多个 elif 语句
- 可以有 0 到 1 个 else 语句

pass 的应用

- 如果需要考虑某个特定的情况，但该情况出现时又不需要做什么，可以用 pass

```
1 if 5>2:
2     pass
3
```

  - pass 也可以用在 while，函数定义 def 等地方，共同的作用是保持语法缩进的正确性

if 实现的条件表达式

- 对于如下语句

```
speed = 200 #any value for test
if speed > 150:
    train = 'Harmony'
else:
    train = 'Normal'

```

- 可以缩写为一句：

```
train = 'Harmony' if speed > 150 else 'Normal'

```

循环 – while

- 语法

```
while boolean_expression:
    while_suite
else:
    else_suite

```

- else 分支
  - 本身是可选的组成部分
  - 只要循环是正常终止，else 分支总会执行
  - 由于 break, 返回语句或异常导致跳出循环，则 else 不会执行
  - else 的上述特点对于 for 循环，及 try...except 都是一样的

while 示例

```
1  def list_find(lst, target):
2      index = 0
3      while index<len(lst):
4          if lst[index] == target:
5              break
6          index += 1
7      else:
8          index = -1
9      return index
10
11 list_find(['ruc', 'irm'], 'irm') #??
12 list_find(['ruc', 'irm'], 'irm2') #??
13
```

循环 – for

- 语法

```
for expression in iterable:
    for_suite
else:
    else_suite

```

  - expression 或者是一个单独的变量，或者是一个变量序列，一般以元组形式给出
  - 如果将元组或列表用于 expression，则其中的每一数据项都会拆分到表达式的项

for 示例

```
1  def list_find(lst, target):
2      for index, x in enumerate(lst):
3          if lst[index] == target:
4              break
5      else:
6          index = -1
7      return index
8
9  list_find(['ruc', 'irm'], 'irm') #??
10 list_find(['ruc', 'irm'], 'irm2') #??
11
```

- 考虑以上代码中 x 的含义是什么

break

- 提前终止循环语句的执行，如前面的例子

continue

- 终止当前循环体内的后续语句执行，重新执行下一轮次的循环
- 例如：统计一个数字列表中大于 10 的元素之和

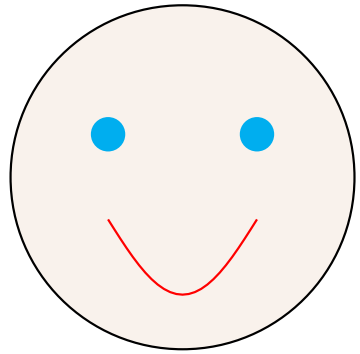
```
1  def countall(lst):
2      count = 0
3      for n in lst:
4          if n<=10:
5              continue
6          count += n
7      return count
8
9  countall([1,3,12,15]) # Result?
10
```

课堂练习

- 复习以前内容
- 利用蒙特卡洛方法计算圆周率
- 打印杨辉三角

圆周率计算

```
1  import random
2
3  def pi(count):
4      total = 0
5      in_circle = 0
6      for i in range(count):
7          total += 1
8          x = random.uniform(-1,1)
9          y = random.uniform(-1,1)
10         distance = x**2 + y**2
11         if(distance<=1):
12             in_circle +=1
13     return in_circle/total * 4
14
15 pi(1000000)
16
```



—END—

CH4 函数

1. 函数的创建
2. 函数参数
3. 作用域
4. 名称与 docstring
5. 递归

函数介绍

- 函数的类型
  - 全局函数
  - 局部函数
  - lambda 函数：特殊表达式，比普通函数有更多限制
  - 方法
- 根据功能的实现者不同
  - 内置函数
  - 第三方函数
  - 自己编写的函数

函数的创建

```
def functionName(parameters):
    suite
```

- parameters 是可选的，如果有多个，则用英文逗号分隔
- 函数参数可以有默认值
- 每个函数都有返回值，如果未明确指定返回，则返回 None

函数创建示例

计算前 top_n 个元素的平均值

```
1 def avg_top(lst, top_n):
2     total = 0
3     for e in lst[:top_n]:
4         total += e
5     print(total/top_n)
6
7 avg_top([1,3,9,7,6], 4)
```

函数参数

- 参数可以有默认值

```
1 def avg_top(lst, top_n=4):
2     total = 0
3     for e in lst[:top_n]:
4         total += e
5     print(total/top_n)
6
7 avg_top([1,3,9,7,6])
```

形式参数与实际参数

- 写在 def 语句中函数名称后面的变量叫函数的形式参数
- 函数调用时提供的值称为实际参数
- 字符串、数字等普通类型默认为传值调用
- 列表、元组等默认为传递引用方式

传值调用示例

```
1 def change(name):
2     name = 'Summer'
3
4 name = 'Spring'
5 change(name)
6 print(name)
```

此时，输出的 name 为 Spring 还是 Summer?

传递引用调用示例

```
1 def change_list(lst):
2     lst.append('Summer')
3
4 x = ['Spring']
5 change_list(x)
6 print(x)
```

此时，输出的 x 结果是 ['Spring', 'Summer']，还是 ['Spring']?

位置参数与关键字参数

- 函数默认按照声明位置依次传入参数，即位置参数
- 当参数顺序难以记忆时，可以使用关键字参数方式，即指定参数名称和参数值的方式传递参数

```
1 def hello(name, greeting):
2     print("%s, %s!" % (name, greeting))
3
4 hello("悟空", "欢迎你")
5 hello(greeting="欢迎你", name="悟空" )
6 hello(name="悟空", greeting="欢迎你")
```

后面三条语句的输出结果分别是什么？

收集参数

- 有时候允许用户提供任意数量的参数非常有用，如实现任意数量的数字的加法
- Python 允许在一个形式参数名称前面加上星号 *，表示收集该位置开始的任意数量的参数
- 任意多个关键字参数的收集：** 修饰符 (自学)

```
1 def add(*numbers):
2     total = 0
3     for n in numbers:
4         total += n
5     return total
6
7 add(1,2,3)
```

Lambda 函数

- 一种快速定义单行的最小函数
- 按照如下语法创建的匿名函数
  - lambda parameters: expression
- 特点
  - parameters 是可选的
  - epxression 不能包含分支或循环，但可以使用条件表达式
  - 不能包含 return, yield 语句
  - 结果为一个匿名函数

Lambda 函数示例

```
1 def f(x,y):
2     return x*y
3
4 g = lambda x,y: x*y
5 print(f(3,4))
6 print(g(3,4))
7
```

- 上例中函数 f 和 g 的效果相同，但 g 的定义更快捷
- 区别：def 是语句而 lambda 是表达式

作用域

- 全局变量
  - 函数内部可以直接调用之前声明的全局变量，但不建议使用这种方式
- 局部变量
  - 在函数 (类) 内部定义的变量

```
1 def f():
2     x = 10
3
4 x = 1
5 f()
6 print(x)
```

结果为：？

在函数内部绑定全局变量

- 通过 global 关键字在函数内部将变量绑定为全局变量

```
def f():
    global x
    x = 10

x = 1
f()
print(x)
```

结果为：10

参数的命名建议

- 对函数及其参数合理命名有助于代码理解
  - 使用命名框架，保持一致性
  - 所有名称避免使用缩略 (除非是标准化且广泛使用的)
  - 合理地使用变量与参数名
    - x: 适合作为坐标参数
    - i: 适合用于简单的循环计数
  - 函数名与方法名应可以表现其行为或返回值
    - E.g. 查找在列表中的位置

```
1 def find(l, s, i=0)
2 def linear_search(l, s, i = 0)
3 def first_index_of(sorted_name_list, name, start=0) #GOOD!
```

docstring

- 对于上面的函数 first_index_of，如果没有找到对应的 name 该怎么处理，是返回 -1，还是产生异常？这类信息可以通过 docstring 提供给函数使用者
- docstring 第一行是对函数的一个简短的描述，紧接着一个空白行，然后是完整描述，如果是交互式输入再执行的程序，还会给出一些示例
- 例子

docstring 示例

```
1  # coding: utf-8
2
3  def first_index_of(sorted_name_list, name, start=0):
4      '''获取排序列表sorted_name_list中指定名称元素的位置
5
6      如果元素name在sorted_name_list中存在，则返回其下标，
7      下标从0开始计数；如果不存在，则返回-1.
8
9      >>> first_index_of([1,2,3], 1)
10     0
11     >>> first_index_of([1,2,3], 5)
12     -1
13
14     '''
15     try:
16         return sorted_name_list.index(name)
17     except ValueError as e:
18         return -1
```

递归

- 函数不仅可以调用其他函数，还可以调用自身
- 递归的典型特点是一个函数不断调用自身，当到达指定条件时，再逐级返回
- 普通的幂函数实现

```
1 def power(x, n):
2     result = 1
3     for i in range(n):
4         result = result*x
5     return result
6
7 power(2,3) #应输出8
8
```

幂函数的递归实现

- 对于任意数字 x，power(x,0) = 1
- 对于任意大于 0 的数 n 来说，power(x,n) 是 x 乘以 power(x, n-1) 的结果

```
def power(x, n):
    if(n==0):
        return 1
    else:
        return power(x, n-1)*x

power(2,3)#应输出8

```

课堂练习

- 利用递归算法解决汉诺塔 (Hanoi Tower) 问题

Hanoi Tower

```
1  def hanoi(a, b, c, n):
2      if n==1:
3          print("move {0} from {1} to {2}".format(n, a, c))
4      else:
5          hanoi(a, c, b, n-1)
6          print("move {0} from {1} to {2}".format(n, a, c))
7          hanoi(b, a, c, n-1)
8
9  hanoi('a', 'b', 'c', 3)
10
```

求组合 $\binom{n}{m}$

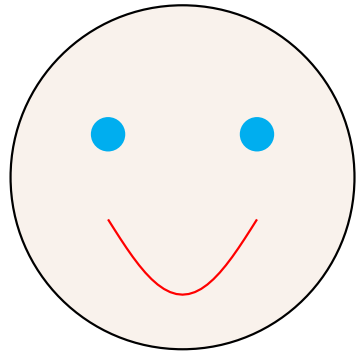
c(n,m)=c(n-1,m-1)+c(n-1,m) 等式左边表示从 n 个元素中选取 m 个元素，而等式右边表示这一个过程的另一种实现方法：任意选择 n 中的某个备选元素为特殊元素，从 n 中选 m 个元素可以由此特殊元素的分成两类情况，即 m 个被选择元素包含了特殊元素和 m 个被选择元素不包含该特殊元素。

函数式编程

- 面向过程的程序设计
  - 把复杂任务分解成简单的任务，简单任务通常以函数表示
- 函数式编程 (Functional Programming)
  - 也可以归结到面向过程的程序设计，但其思想更接近数学计算。
  - 就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如 C 语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如 Lisp 语言。
  - 函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量。任意一个函数，只要输入是确定的，输出就是确定的，这种没有副作用的函数称为纯函数。
  - 允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

高阶函数：Higher-order function

- 变量可以指向函数
- 函数名也是变量
- 传入函数
-



—END—

CH5 模块

1. 模块介绍
2. 模块的导入方式
3. 模块的作用域
4. 模块的测试
5. 模块导入的路径搜索
6. 包

模块介绍

- 随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。
- 为了编写可维护的代码，把函数分组放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。
- 在 Python 中，一个.py 文件就称之为一个模块（Module）。
  - 模块：把多个函数组织到一起，方便其他程序调用
  - 提高了代码的可维护性
  - 编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。
- 之前我们编写的程序也保存在.py 文件中，程序和模块的区别在于：
  - 程序的设计目标是运行
  - 模块的设计目标是由其他程序导入并使用

模块的导入方式

- import importable
- import importable1, importable2, ..., importableN
- import importable as preferred_name
- from importable import *

标准库中的模块使用示例

```
1 import sys
2 from pprint import pprint
3 pprint(sys.path)
4
```

```
['',
'/usr/lib/python3.4',
'/usr/lib/python3.4/plat-x86_64-linux-gnu',
'/usr/lib/python3.4/lib-dynload',
'/usr/local/lib/python3.4/dist-packages',
'/usr/lib/python3/dist-packages']
```

自定义模块 I

```
1  #!/usr/bin/env python3
2  # −*− coding: utf−8 −*−
3
4  ’’’
5  Hello world!
6  ’’’
7
8  import sys
9
10 __author__ = ’Tian Xia’
11
12
13 def sayHi():
14     args = sys.argv
```

自定义模块 II

```
15      if len(args) == 1:
16              print('Hello, world!')
17      elif len(args) == 2:
18          print('Hello, %s!' % args[1])
19      else:
20          print('Too many arguments!')
21
22  if __name__ == '__main__':
23      sayHi()
```

自定义模块 III

- 行 1 的注释可以让 hello.py 文件直接在 Unix/Linux/Mac 上运行
- 行 2 的注释表示.py 文件本身使用标准 UTF-8 编码
- 第 4 到 6 行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释
- 第 8 行导入了引用的模块
- 第 10 行使用 __author__ 变量把作者写进去

如何运行

- 方式 1:
  - 保存到 hello.py 文件中
  - 进入命令行，通过 cd 命令进入 hello.py 文件所在的目录
    - python3 hello.py
    - python3 hello.py Tom
- 方式 2:
  - 启动 python 交互环境
    - `>>> import hello`
    - `>>> hello.sayHi()`
    - `>>> help(hello)`
- 观察 sys.argv 是否包含了模块对应的文件名称

模块的作用域

- 在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。
  - 通过 _ 前缀定义的函数和变量只能在模块内部访问
  - 其他函数和变量则是公开可访问的
  - __xxx__ 这样的变量可以被直接引用，但通常有特殊含义
  - 如 __name__, __author__
- private 函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为 Python 并没有一种方法可以完全限制访问 private 函数或变量

作用域示例: hello2.py I

```
1  #!/usr/bin/env python3
2  # −*− coding: utf−8 −*−
3
4  '''
5  Hello world!
6  '''
7
8  import sys
9
10 ___author___ = 'Tian Xia'
11
12
13 def _sayInChinese():
14     return '你好'
```

作用域示例: hello2.py II

```
15
16
17 def _sayInEnglish():
18     return 'hello'
19
20
21 def sayHi():
22     args = sys.argv
23     if len(args) == 1:
24             print(_sayInChinese())
25     elif len(args) == 2:
26         print('%s, %s!' % (_sayInEnglish(), args[1]))
27     else:
28         print('Too many arguments!')
```

作用域示例: hello2.py III

```
29
30 if __name__ == '__main__':
31     sayHi()
```

- 请分别用命令行和 Python 交互环境进行测试
- 问题：能够在交互环境中通过 hello2._sayInChinese() 访问私有方法?
- 实验：添加代码，使得程序能够根据命令行传入的参数，决定源代码 26 行处是调用 _sayInChinese() 还是 _sayInEnglish()
  - 假设命令行传入的第一个有效参数用于指定语言，中文对应为 zh，英文对应为 en

hello_lang.py I

```
1  #!/usr/bin/env python3
2  # −*− coding: utf−8 −*−
3
4  '''
5  Hello world!
6  '''
7
8  import sys
9
10 ___author___ = 'Tian Xia'
11
12
13 def _sayInChinese():
14     return '你好'
```

hello_lang.py II

```
15
16
17 def _sayInEnglish():
18     return 'hello'
19
20
21 def sayHi():
22     args = sys.argv
23     if len(args) == 1:
24         print(_sayInChinese())
25     elif len(args) == 2:
26         if(args[1] == 'en'):
27             print(_sayInEnglish())
28         else:
```

hello_lang.py III

```
29              print(_sayInChinese())
30      elif len(args) == 3:
31          msg = _sayInEnglish() if args[1] == 'en' else
          _sayInChinese()
32          print('%s, %s!' % (msg, args[2]))
33      else:
34          print('Too many arguments!')
35
36  if __name__ == '__main__':
37      sayHi()
```

hello_lang.py IV

- python3 hello_lang.py zh
- python3 hello_lang.py en
- python3 hello_lang.py zh Tom
- python3 hello_lang.py en Tom

模块的测试

- 模块本身用于定义函数、类及其他一些内容
- 在模块中添加一些检查模块本身是否正常工作的测试代码非常有用

```
1 # coding: utf-8
2 # hello3.py
3
4 def hello():
5     print("Hello world!")
6
7 # 测试代码
8 hello()
```

- 打开 Python 交互环境测试
  - `>>> import hello3`
  - `>>> hello3.hello()`

__name__

- >>> hello3.___name___
- >>> ___name___
- 因此，在测试模块时，可以通过如下方式：

```
1 if ___name___ == '___main___':
2     test_suite...
```

  * 此时，如果将模块作为独立的程序，条件判断将会满足，继续执行测试代码
  * 如果是 import 引入模块，则条件表达式不成立，测试代码被忽略

如何让 Python 找到自定义的模块

1. 在源代码目录下执行 python
2. 设置 sys.path

```
1  mport sys
2  from pprint import pprint
3
4  import hello
5  # ImportError: No module named 'hello'
6
7  pprint(sys.path)
8  sys.path.append('/home/xiatian/Documents/git/teaching/python/sli
9
10 import hello
11 hello.sayHi()
12 # Hello, world!
```

包的处理

- 包是一个有层次的文件目录结构，由模块和子包组成。
  - 为平坦的名称空间加入了有层次的组织结构
  - 允许程序员把有联系的模块组织到一起
  - 允许分发者使用目录结构而非一大堆文件
  - 有助于解决模块名称冲突问题

__init__.py 文件

- python 的每个模块的包中，都有一个 __init__.py 文件，有了这个文件，我们才能导入这个目录下的 module
  - 该文件可以为空
  - 我们在导入一个包时，实际上导入了它的 __init__.py 文件

包的示例

```
graphics/
    __init__.py
    primitive/
        __init__.py
        line.py
        fill.py
        text.py
    formats/
        __init__.py
        png.py
        jpg.py
```

以上包结构的引用方式

```
1 import graphics.primitive.line
2 from graphics.primitive import line
3 from graphics.primitive import *
4 import graphics.formats.jpg as jpg
```

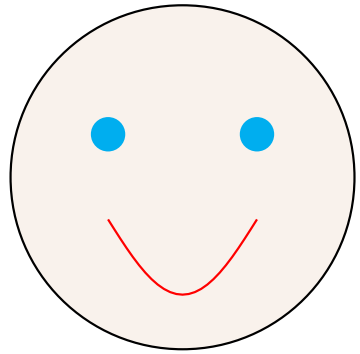
- 第 1 行在使用 line 中的方法时，只能用全名称引用：
  - graphics.primitive.line.xxxx
- 第 2 行和第 3 行的方式，则可以直接使用如下方式
  - line.xxxx, fill.xxxx
  - jpg.xxxx

练习

- 实现以上包结构
  - line.py, fill.py 和 text.py 中分别写一个 draw() 方法，在该方法中输入一个简单的字符串
  - 在 png.py 和 jpg.py 中添加 open() 方法和 close() 方法，方法本身只输出一个提示字符串即可

__init__.py 中的 __all__

- 通过 from p1.p2 import *，可以一次性引入包里面的所有子模块
- 如果想限定默认引入的子模块集合，可以通过设置 __init__.py，例如：
  - 在 __init__.py 中添加如下内容：
    __all__ = ['line', 'fill']
  - from graphics.primitive import *
  - 请完善代码并测试能够直接调用 line.xxx() 和 text.xxxx()，并分析原因



—END—

CH6 Python 标准库

- 内置电池：Battery Included　用于形容 Python 标准库
  - 标准模块
    - string
    - io & sys
    - optparse
    - math
    - random

字符串标准模块：string

- 提供了字符串相关的一些常量

```
>>> string.ascii_letters
>>> string.ascii_lowercase
>>> string.ascii_uppercase
>>> string.digits
>>> string.hexdigits
>>> string.punctuation
```

IO 输出相关标准模块

- 标准输出
  - sys.stdout
- 字符串输出
  - io.StringIO
- 以下输出方式是等价的

```
1 import sys,io
2 print("hello")
3 print("hello", file=sys.stdout)
4 sys.stdout.write("hello\n") #python3 中会同时输出字符串的数量
```

StringIO

- 如果要把输出重定向到字符串中，在需要时再获取，可以 StringIO
  - 注意：Python 2.x 和 3.x 在输出时的行为不同

```
1 import sys, io
2 out = io.StringIO()
3 sys.stdout = out
4 out.write('how are you')
5 print('hello')
6 print('guys')
7 sys.stdout = sys.__stdout__
8 print(out.getvalue())
```

命令行参数模块：optparse I

- 如何指定脚本运行的命令行参数
  - 例如，Linux 命令 ls -l
  - Python 提供的 optparse 模块对命令行参数的解析处理提供了良好的支持

```
1  # coding: utf-8
2
3  from optparse import OptionParser
4  parser = OptionParser()
5
6  parser.add_option("-f", "--file", dest="filename",
7                    help="write report to FILE", metavar="FILE")
8  parser.add_option("-q", "--quiet",
9                    action="store_false", dest="verbose", default=True,
10                   help="don't print status messages to stdout")
11
```

命令行参数模块：optparse II

```
12 (options, args) = parser.parse_args()
13 print(options)
14
15 if not options.filename:
16     parser.error('未指定-f参数')
17
18 f = open(options.filename, 'r')
19 lineno = 1
20 for line in f:
21     print(lineno, '\t', line, end='')
22     lineno += 1
23 print('---FINISH---')
```

分析

- 说明
  - 长短参数名称
  - 参数对应的变量名称及获取方式
  - metavar 参数：提醒用户该命令行参数所期待的参数，参数中的字符串会自动变为大写
  - parser.parse_args()
    - 解析后得到的 options 拥有两个属性：filename 和 verbose
- 假设文件保存到 cmdopt.py，执行:
  - python3 cmdopt.py -f cmdopt.py，观察输出结果
  - python3 cmdopt.py -h

数学标准模块：math

- math.exp(x)

$$e^x$$

- math.sqrt(x)

$$\sqrt{x}$$

- math.pi
- math.e
- math.log
  - >>> math.log(math.e)
  - >>> math.log2(2)
  - >>> math.log10(10)
  - >>> math.log1p(math.e − 1)

随机数标准模块：random

- 生成 $[0, 1)$ 之间的随机数
  - `>>> import random`
  - `>>> random.random()`
- random.gauss(mu, sigma) 生成一个均值为 mu, 标准值为 sigma 的符合高斯分布的随机数
  - `>>> random.gauss(0, 1)`

高斯分布模拟

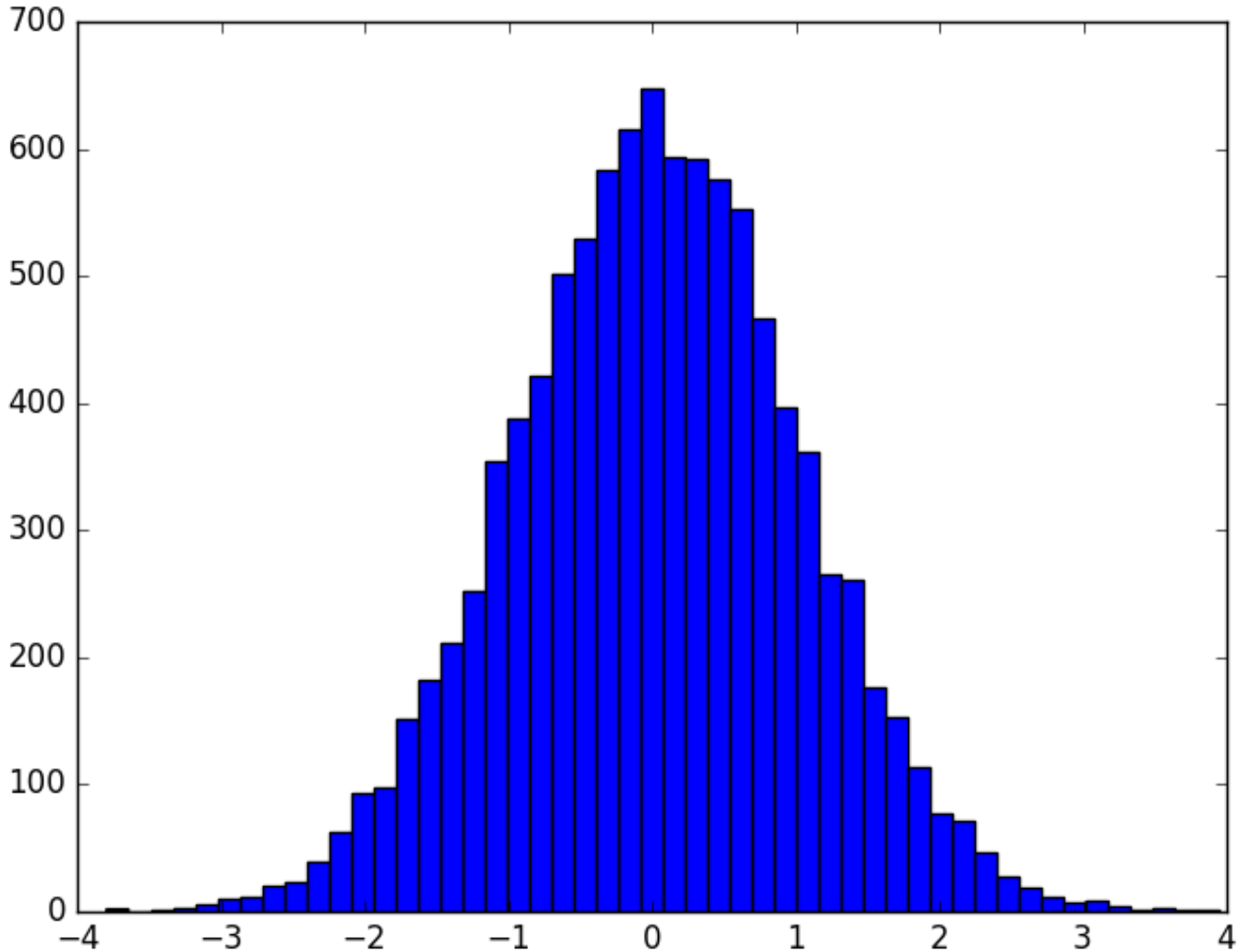
- 通过 random.gauss() 生成一组随机数
- 通过柱状图显示结果

高斯分布模拟

- 通过 random.gauss() 生成一组随机数
- 通过柱状图显示结果

```
1 import matplotlib.pyplot as plt
2 import random
3
4 data = [random.gauss(0,1) for x in range(10000)]
5 plt.hist(data, bins = 50)
6 plt.show()
```

Result



高斯分布模拟

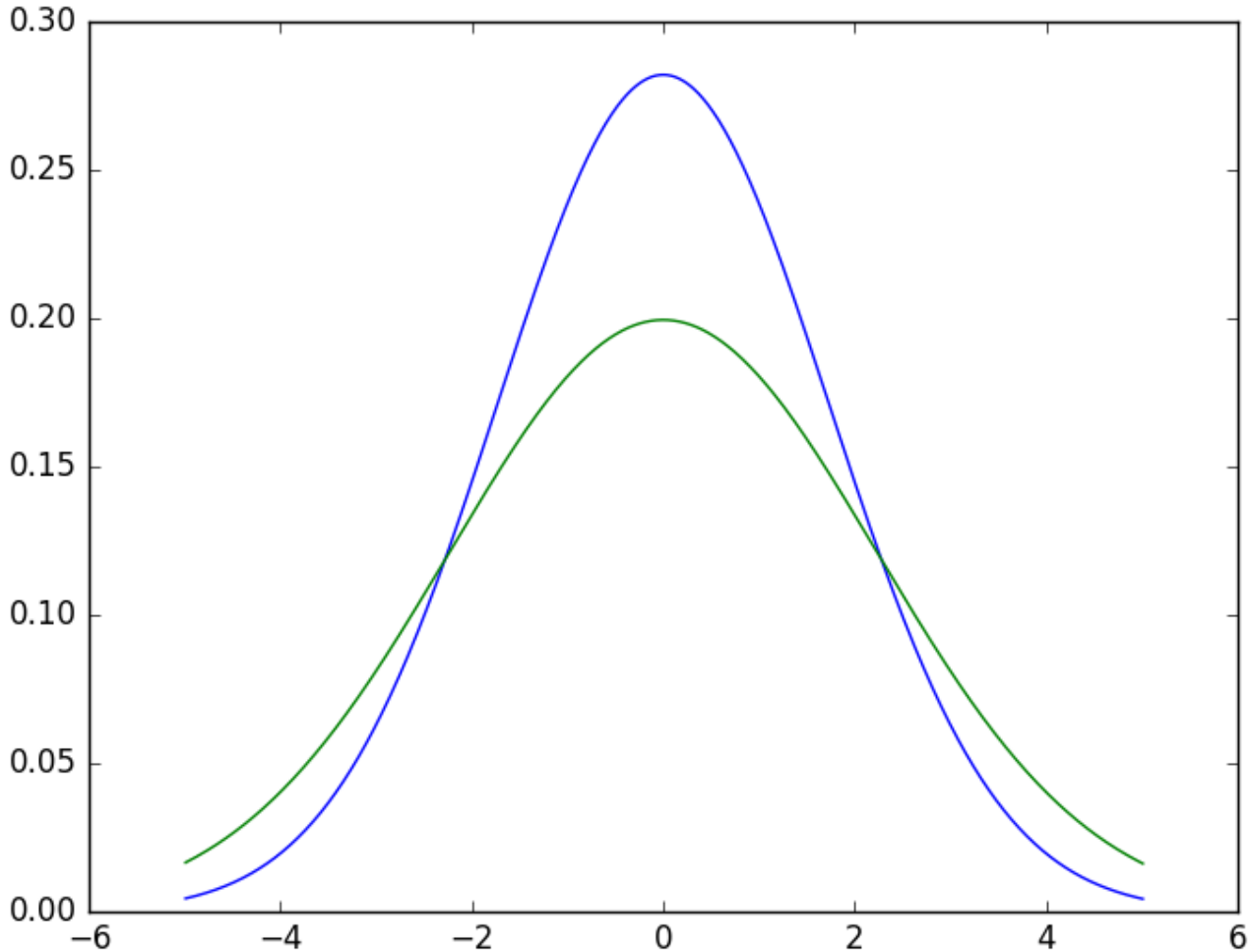
- 通过高斯分布的概率密度函数生成 (x, y) 对，利用散点图显示结果

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

高斯分布模拟代码

```
1  import math
2  import matplotlib.pyplot as plt
3
4  sigma = 2
5  sigma2 = 4
6  X = []
7  Y = []
8  Y2 = []
9  x = -5
10 while x < 5:
11     x = x+0.01
12     y = 1/math.sqrt(sigma*2*math.pi)*math.exp(-x*x/(2*sigma^2))
13     y2 = 1/math.sqrt(sigma2*2*math.pi)*math.exp(-x*x/(2*sigma2^2))
14     X.append(x)
15     Y.append(y)
16     Y2.append(y2)
17
18 plt.plot(X, Y)
19 plt.plot(X,Y2)
```

Result



日志处理标准库：logging

- 日志的作用
- 日志的级别
  - DEBUG：详细的信息，通常只出现在诊断问题上
  - INFO：确认一切按预期运行
  - WARNING：一个迹象表明，一些意想不到的事情发生了，或表明一些问题在不久的将来（例如。磁盘空间低”）。这个软件还能按预期工作。
  - ERROR：个更严重的问题，软件没能执行一些功能
  - CRITICAL：一个严重的错误，这表明程序本身可能无法继续运行
- 优先级关系
  - CRITICAL › ERROR › WARNING › INFO › DEBUG

日志示例

```
1 import logging
2
3 logging.debug('debug message')
4 logging.info('info message')
5 logging.warning('warning message')
6 logging.error('error message')
7 logging.critical('critical message')
8
```

- 默认输出到控制台，级别在 warning 及以上的信息会输出，
- 可以通过 basicConfig 设置输出方式和记录级别

日志示例：输出到文件

```
1  import os
2  import logging
3
4
5  FILE = os.getcwd()
6  logging.basicConfig(filename=os.path.join(FILE, 'log.txt'),
7                      level=logging.DEBUG)
8  logging.debug('some debug messages...')
9  logging.info('some info messages...')
10 logging.warning('some warning messages...')
11
```

第 3 方扩展：Requests

- 通过 Requests 包可以更方便地实现 Web 数据的采集
  - Install: pip install requests

```
1 import requests
2
3 url = 'http://download.pchome.net/wallpaper/zhiwu/'
4 response = requests.get(url)
5 print(response.text)
6
```

BeautifulSoup

- 怎么解析抓取到的网页，例如，如何抽取网页中的图片?
- BeautifulSoup – Html Parser
  - Install: pip install beautifulsoup4

Example

```
1  import requests
2
3  url = 'http://download.pchome.net/wallpaper/zhiwu/'
4  response = requests.get(url)
5  print(response.text)
6
7  from bs4 import BeautifulSoup as soup
8  doc = soup(response.text, 'html')
9  images = [element.get('src') for element in doc.find_all('img')]
10
```

课堂练习

- 编写程序，实现对任意指定网页，下载该网页包含的所有图片到指定的文件夹当中

Thanks

CH7 面向对象编程

- 简介
- 类和实例
- 数据封装
- 继承和多态
- Duck Typing
- 获取对象信息
- 实例属性和类属性
- 高级特性
- 小结

⇒ 面向对象编程简介

- 面向对象编程 OOP(Object Oriented Programming)
  - 一种程序设计思想，OOP 把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。
  - 面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行
  - OOP 通过把大块函数切割为小块函数来降低系统的复杂度
  - OOP 把计算机程序看做是一组对象的集合，程序执行就是一些列消息在各个对象之间传递，给对象发消息实际上就是调用对象对应的关联函数
- OOP 特点
  - 封装、继承、多态
- 类 (Class) 与实例 (Instance)

举例: 面向过程方式

处理学生成绩并打印输出，面向过程设计方式：

```
1 std1 = {'name': '韩寒', 'score':85}
2 std2 = {'name': '黄锤', 'score':90}
3
4 def print_score(std):
5     print('%s: %s' % (std['name'], std['score']))
6
7 print_score(std1)
8 print_score(std2)
9
```

举例: 面向对象方式

```
1  class Student(object):
2      def ___init___(self, name, score):
3          self.name = name
4          self.score = score
5
6      def print_score(self):
7          print('%s: %s' % (self.name, self.score))
8
9  if ___name___ == '___main___':
10     han = Student('韩寒', 85)
11     huang = Student('黄锤', 90)
12     han.print_score()
13     huang.print_score()
14
```

例子解释

- class 关键字
- object
- self

⇒ 类和实例

- 类是抽象的模板，比如 Student 类
- 实例是根据类创建出来的一个个具体的“对象”
- 每个对象都拥有相同的方法，但各自的数据可能不同。

类的定义

通过 class 关键字定义类

```
1 class Student(object):
2     pass
3
```

- class 后面紧接着是类名
- 类名通常是大写开头的单词
- 类名后紧接着是 (object)，表示该类是从哪个类继承下来的
  - object 是所有类最终都会继承的类

类的实例化

类名 +() 实现, 例如:

```
1 xiaoming = Student('小明', 85)
2 print(xiaoming)
3 print(Student)
```

<__main__.Student object at 0x7f16e99ee710>
<class '__main__.Student'>

实例的变量绑定

可以自由地给一个实例变量绑定属性和方法，比如，给实例 xiaoming 绑定一个 name 属性，和 speak() 方法：

```
1 def speak(something):
2     print('speak:' , something)
3
4 xiaoming.name = 'xiaoming'
5 xiaoming.speak = speak
6 print(xiaoming.name)
7 xiaoming.speak('hello')
```

实例的初始化

通过定义特殊的 __init__ 方法，在创建实例的时候，可以绑定期望的属性

```
1 class Student(object):
2
3   def ___init___(self, name, score):
4       self.name = name
5       self.score = score
```

- __init__ 方法的第一个参数永远是 self，表示创建的实例本身
- 在 __init__ 方法内部，可以把各种属性绑定到 self，因为 self 就指向创建的实例本身。

__init__() 解释

- 有了 __init__ 方法，在创建实例的时候，就不能传入空的参数了，必须传入与该方法匹配的参数，但 self 不需要传，Python 解释器自己会把实例变量传进去

```
>>> xiaoli = Student('小李', 80)
```

- 和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量 self，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

⇒ 数据封装

- 实例中拥有的属性、方法，被封装到实例中
- 在实例中使用本身的属性或方法，用 self.xxx 方式
- 在类中定义方法时，第一个参数必须为 self
- 在外部调用方法时，无需指定 self，类会自动把 self 传入
- 封装的优点是调用简单，无需关心内部实现细节

⇒ 继承和多态

在 OOP 程序设计中定义一个 class 的时候，可以从某个现有的 class 继承，新的 class 称为子类（Subclass），而被继承的 class 称为基类、父类或超类（Base class、Super class）。

```
1 class Postgraduate(Student):
2     '''post graduate student'''
3     pass
```

继承可以把父类的所有功能都直接拿过来，这样就不必从零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写(多态)。

多态

- 多态即多种状态
- 同一个实体同时具有多种形式, 对基类的引用指向子类的对象
  - 例子中，子类和父类都存在相同的 sayHi() 方法，此时，子类的 sayHi() 覆盖了父类的 sayHi()，在代码运行的时候，总是会调用子类的 sayHi()

多态示例 I

```
1  # coding: utf-8
2
3
4  class Student(object):
5      ”” 学生的基类 ””
6
7      def __init__(self, name, score):
8          self.name = name
9          self.score = score
10
11     def print_score(self):
12         print('%s: %s' % (self.name, self.score))
13
14     def sayHi(self):
15         print('我是学生%s' % self.name)
16
17
```

多态示例 II

```
18 class Postgraduate(Student):
19     '''研究生'''
20     def sayHi(self):
21         print('我是研究生%s' % self.name)
22
23
24 def meet(student):
25     student.sayHi()
26
27
28 if __name__ == '__main__':
29     han = Student('韩寒', 85)
30     wang = Postgraduate('王小二', 80)
31     meet(han)
32     meet(wang)
```

isinstance

- 可以用 isinstance 判断实例的类型

```
1 wang = Postgraduate('王二小', 80)
2 isinstance(wang, Postgraduate) #True or False?
3 isinstance(wang, Student) #True or False?
```

⇒Duck Typing[1]

- 对于静态语言 (如 Java)，如果需要传入 Student 类型，则传入的对象必须是 Student 类型或者它的子类
- 对于 Python 这样的动态语言，只需要保证传入的对象有 sayHi() 方法就可以
- 这种设计方法称之为鸭子类型:
  - 一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。

```
1 class Robot():
2     def sayHi(self):
3         print("I'm robot")
4
5 meet(Robot())
```

[1] https://en.wikipedia.org/wiki/Duck_typing

Duck Typing

Duck Test

”When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

– Indiana poet James Whitcomb Riley (1849– 1916)

Duck Typing 示例 I

```
1  class Duck:
2      def quack(self):
3          print("Quaaaaaack!")
4      def feathers(self):
5          print("The duck has white and gray feathers.")
6
7  class Person:
8      def quack(self):
9          print("The person imitates a duck.")
10     def feathers(self):
11         print("The person takes a feather from the ground and shows it.")
12     def name(self):
13         print("John Smith")
14
15 def in_the_forest(duck):
16     duck.quack()
17     duck.feathers()
```

Duck Typing 示例 II

```
18
19 def game():
20     donald = Duck()
21     john = Person()
22     in_the_forest(donald)
23     in_the_forest(john)
24
25 game()
```

⇒ 获取对象信息

- 当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢?
  - isinstance(): 判断对象是否为某个类的实例
  - type(): 判断对象的类型
  - dir(): 获得对象的所有属性和方法

type()

```
1  type(123) == type(456)
2  type(123) == int
3  type('hello') == str
4
5  import types
6  type(abs) == types.BuiltinFunctionType
7  type(lambda x:x) == types.LambdaType
8  type((x for x in range(10)))==types.GeneratorType
9
10 def hi():
11     pass
12 type(hi) == types.FunctionType
```

dir()

- 获得一个对象的所有属性和方法，可以使用 dir() 函数，它返回一个包含字符串的 list

```
1 wang = Postgraduate('王小二', 80)
2 dir(wang)
3
4 ['__class__', '__delattr__', '__dict__', '__dir__',
      '__doc__', '__eq__', '__format__', '__ge__',
      '__getattribute__', '__gt__', '__hash__', '__init__',
      '__le__', '__lt__', '__module__', '__ne__',
      '__new__', '__reduce__', '__reduce_ex__', '__repr__',
      '__setattr__', '__sizeof__', '__str__',
      '__subclasshook__', '__weakref__', 'name', 'print_score',
      'sayHi', 'score']
```

⇒ 实例属性和类属性

- 实例属性隶属于实例
  - 通过 self 或实例绑定属性
- 类属性隶属于类, 其所有实例均可以访问到
  - 直接在 class 中定义的属性

实例属性和类属性示例 I

```
1 #实例属性示例
2 class Student(object):
3     def ___init___(self, name):
4         self.name = name #通过self绑定属性
5
6 s = Student('Bob')
7 s.score = 90 #通过实例绑定属性
8 print("%s : %s" % (s.name,  s.score))
```

实例属性和类属性示例 II

```
1 #类属性示例
2 class Student(object):
3     name = 'Student'
```

>>> s = Student() # 创建实例 s

>>> print(s.name) # 打印 name 属性，因为实例并没有 name 属性，所以会继续查找 class 的 name 属性

Student

>>> print(Student.name) # 打印类的 name 属性

Student

>>> s.name = 'Michael' # 给实例绑定 name 属性

实例属性和类属性示例 III

>>> print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的 name 属性

Michael

>>> print(Student.name) # 但是类属性并未消失，用 Student.name 仍然可以访问

Student

>>> del s.name # 如果删除实例的 name 属性

>>> print(s.name) # 再次调用 s.name，由于实例的 name 属性没有找到，类的 name 属性就显示出来了

Student

⇒ 面向对象编程小结

- 类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；
- 方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；
- 通过在实例上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。
- 和静态语言不同，Python 允许对实例变量绑定任何数据
  - 也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同
- Duck Typing
- 实例属性与类属性

练习

- 假设磁盘中存在一个学生成绩的文本文件, 每行格式如下:
  - 123, 成工, 高等数学,80
  - 123, 成工，线性代数,85
  - 124, 王小二, 线性代数,80
  - ...
- 利用 OOP 设计思想, 实现以下功能:
  - 按照学号或姓名查找满足条件的学生的所有成绩信息，并输出其平均分
  - 根据课程名称输出所有学生的成绩，并输出平均成绩、最高分和最低分

Test I

```
1  class Query():
2      def ___init___(self, filename):
3          slef.filename = filename
4
5      def _parse_line(self, line):
6          return 123, 'zhang', 'math', 80
7
8      def find_by_number(self, num):
9          scores = {}
10         f = open(self.filename, 'r')
11         for line in f:
12             n, name, course, score = self._parse_line(line)
13             if n == num:
14                 scores[course] = score
```

Test II

```
15
16          total = 0
17          for course, score in scores.items():
18              print(course, '==>', score)
19              total += score
20          print('avg', total/len(scores))
21
22      if __name__ == '__main__':
23          q = Query('some_file.txt')
24          q.find_by_number(123)
25
```

Python 面向对象的高级特性

- __slots__
- @property
- 多重继承
- 定制类
- 枚举类
- 元类

__slots__ 的引入 I

实例绑定与类的绑定示例

```
1  class Student(object):
2      pass
3
4  s = Student()
5  s.name = 'Lucy'
6  print(s.name) # 输出Lucy
7
8  def set_age(self, age):
9      self.age = age
10
11 from types import MethodType
12 s.set_age = MethodType(set_age, s) # 给实例绑定方法
```

__slots__ 的引入 II

```
13 s.set_age(25) # 调用实例方法
14 print(s.age) # 测试结果
15
16 s2 = Student() # 创建新的实例
17 s2.set_age(25) # 尝试调用方法
18
```

调用 s2.set_age(25) 会给出错误提示：对象 Student 没有属性 set_age，此时，为了给所有实例都绑定方法，可以给 class 绑定方法，如下：

__slots__ 的引入 III

```
def set_score(self, score):
    self.score = score

Student.set_score = set_score # 在类级别上绑定方法

s.set_score(100)
print(s.score)
s2.set_score(99)
print(s2.score)
```

- 新问题：
  - 如果要限制对实例的属性随意赋值，该怎么处理?
  - __slots__

__slots__

__slots__ 规定了 class 能添加的属性

```
1 class Student(object):
2     ___slots___ = ('name', 'age')
3
4 s = Student()
5 s.age = 25
6 s.score = 90 # Error!
```

AttributeError: 'Student' object has no attribute 'score'

注意：__slots__ 定义的属性仅对当前类实例起作用，对继承的子类是不起作用的

@property 的引入 I

- 在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数
  - 例如，成绩 score，我们希望把成绩的范围限制到 0-100 之间
  - 思路：增加 get_score() 和 set_score() 对进行读取和修改

```
1 class Student(object):
2     def get_score(self):
3         return self._score
4 
5     def set_score(self, value):
6         if not isinstance(value, int):
7             raise ValueError('score必须是整数!')
8         if value < 0 or value > 100:
9             raise ValueError('score必须是一个0到100之间的数字')
```

@property 的引入 II

```
10            self._score = value
11
```

@property 的引入 III

```
s = Student()
s.set_score(60)
print(s.get_score())
s.set_score(999) # Error!

```

- 上面的调用方法又略显复杂
- 能否仍然使用 s.score = 某个数字, 同时还能判断赋予的数值是否满足要求?
- Python 的 @property 装饰器负责把一个方法变成属性调用

@property 示例 I

```
1  class Student(object):
2      @property
3      def score(self):
4          return self._score
5
6      @score.setter
7      def score(self, value):
8          if not isinstance(value, int):
9              raise ValueError('score必须是整数!')
10         if value < 0 or value > 100:
11             raise ValueError('score必须是一个0到100之间的数字')
12         self._score = value
```

@property 示例 II

```
1 s = Student()
2 s.score = 60
3 print(s.score)
4 s.score = 999 # Error!
```

多重继承

- 继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。
- 假设我们要实现以下 4 种动物：
  - Dog - 狗
  - Bat - 蝙蝠
  - Parrot - 鹦鹉
  - Ostrich - 鸵鸟

设计思路

按照哺乳动物和鸟类归类:

按照“能跑”和“能飞”来归类

设计思路

```
class Animal(object):
    pass

# 大类:
class Mammal(Animal):
    pass

class Bird(Animal):
    pass

# 各种动物:
class Dog(Mammal):
    pass

class Bat(Mammal):
    pass
```

定义 Runnable 和 Flyable

```
1 class Runnable(object):
2     def run(self):
3         print('Running...')
4
5 class Flyable(object):
6     def fly(self):
7         print('Flying...')
8
```

多重继承示例

```
1 class Dog(Mammal, Runnable):
2     pass
3
4 class Bat(Mammal, Flyable):
5     pass
6
```

- 通过多重继承，一个子类就可以同时获得多个父类的所有功能。

MixIn

- 在设计类的继承关系时，通常，主线都是单一继承下来的
  - 如：Ostrich 继承自 Bird
- 如果需要“混入”额外的功能，通过多重继承就可以实现
  - 比如，让 Ostrich 除了继承自 Bird 外，再同时继承 Runnable
- 这种设计方法通常称之为 MixIn
  - 为了更好地看出继承关系，通常采用 MixIn 作为功能性的父类名称的后缀

MixIn 实例

```
1  class RunnableMixIn(object):
2      def run(self):
3          print('Running...')
4
5  class FlyableMixIn(object):
6      def fly(self):
7          print('Flying...')
8
9  class Dog(Mammal, RunnableMixIn):
10     pass
11
```

对类进行定制

- __len__
- __str__
- __repr__
- __iter__ 与 __next__
- __getitem__
- __getattr__
- __call__

__len__

```
1 class Panda(object):
2 def ___init___(self):
3     pass
4
5 def ___len___(self):
6     return 10
7
8 a = Panda()
9 panda)
```

__str__ 与 __repr__

```
1  class Panda(object):
2      def ___init___(self):
3          pass
4
5      def ___str___(self):
6          return '熊猫'
7
8  panda = Panda()
9  print(panda) # 熊猫
10 panda    # <___main___.Panda at 0x7f20e414db38>
11
```

- 如何解决非 print 时输出地址的问题?

__str__ 与 __repr__

```
1  class Panda(object):
2      def ___init___(self):
3          pass
4
5      def ___str___(self):
6          return '熊猫'
7
8      ___repr___ = ___str___
9  panda = Panda()
10 print(panda) # 熊猫
11 panda    # 熊猫
12
```

__iter__ 与 __next__

- 方便通过 for 循环对对象维持的数据进行遍历
  - 例如：假设拥有一个斐波那契数列的类 Fib，通过以下语句循环输出：

```
for n in Fib(10):
    print(n)

```

  - 我们希望输出 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
  - 怎么实现 Fib?

Fib I

```
1  # coding: utf-8
2
3
4  class Fib(object):
5      '''
6      生成指定数量并满足斐波那契数列的数字序列
7      '''
8      def __init__(self, total):
9          self.a, self.b = 0, 1  # 初始化两个计数器a，b
10         self.count, self.total = 0, total  #
           初始化已经输出的数量和需要输出的总数量
11
12     def __iter__(self):
13         return self  # 实例本身就是迭代对象，故返回自己
14
15     def __next__(self):
16         self.a, self.b = self.b, self.a + self.b  # 计算下一个值
```

Fib II

```
17          self.count += 1
18          if self.count > self.total:
19              raise StopIteration()
20          return self.a
21
22  if __name__ == '__main__':
23      for n in Fib(10):
24          print(n)
```

扩展练习

- 用 Python 编写程序，对本机指定目录下的文件进行扫描，把所有指定后缀名的文档名称统一输出到一个文本文件中。

END

CH8 异常、调试与测试

- 异常
  - Python 的异常处理使用方法
  - Python 的异常继承关系
  - 利用 raise 抛出异常
- 调试
  - print 调试法
  - logging 调试法
  - assert
  - pdb
- 单元测试

异常处理

- 程序在编写过程中，有大量情况需要考虑
  - 除数是否为 0
  - 打开文件时，需要判断文件是否存在，有无权限
  - ...
- 异常可以简化这一处理过程
- Python 的错误处理机制
  - try...except...finally...

try

```
1  try:
2      print('try...')
3      r = 10 / 0
4      print('result:', r)
5  except ZeroDivisionError as e:
6      print('except:', e)
7  finally:
8      print('finally...')
9  print('END')
10
```

try...
except: division by zero
finally...
END

try

```
1  try:
2      print('try...')
3      r = 10 / 2
4      print('result:', r)
5  except ZeroDivisionError as e:
6      print('except:', e)
7  finally:
8      print('finally...')
9  print('END')
10
```

```
try...
result: 5.0
finally...
END
```

Python 异常处理的规则

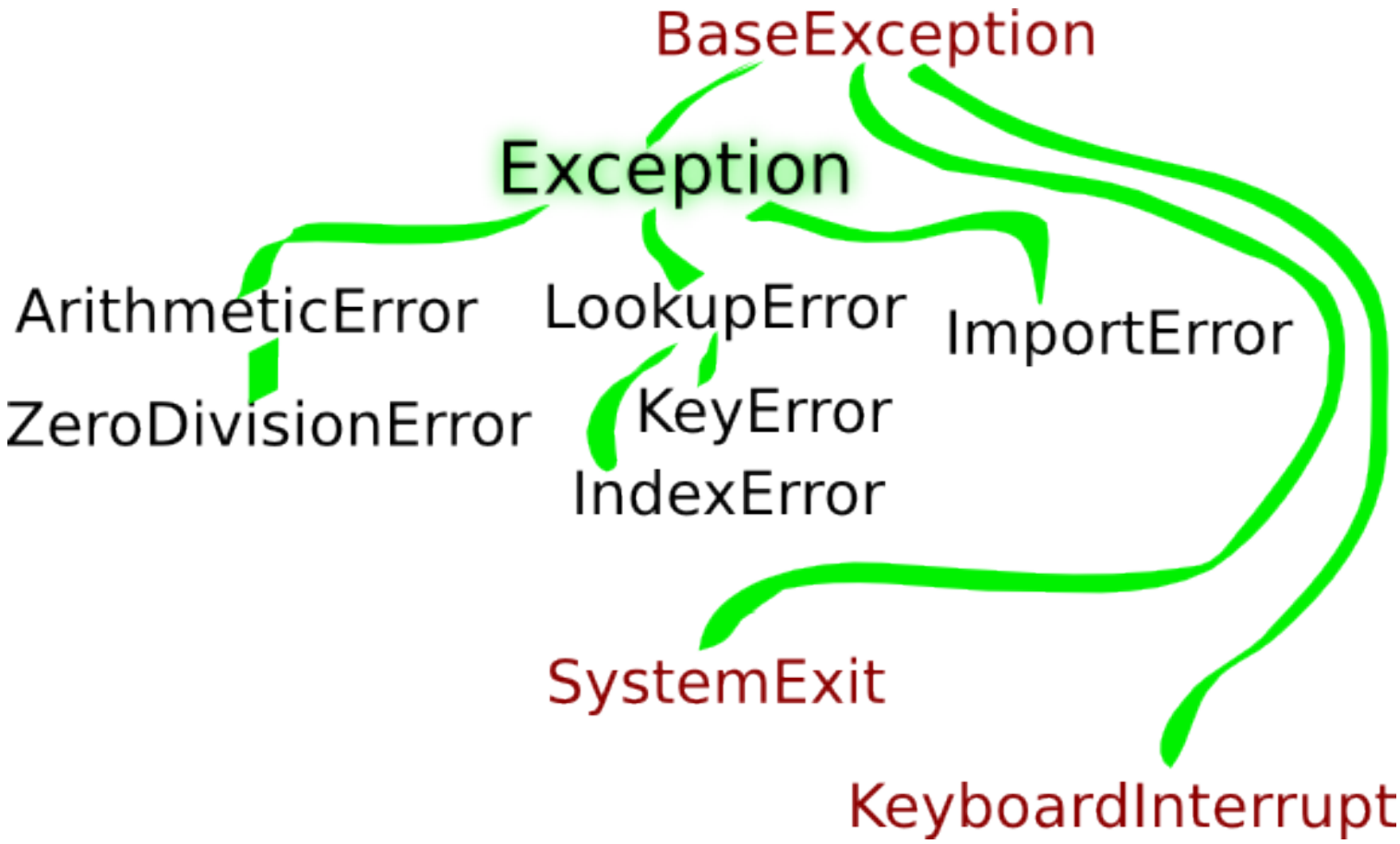
- 遇到第一个满足条件的异常，执行该异常下的语句，忽略后续的其它异常
- finally 永远会被执行

Python 异常的继承关系

- Python 的错误也是 class，都继承自 BaseException
  - 在使用 except 时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。

```
lst = [x for x in range(10)]
try:
    n = lst[15]
except LookupError as e:
    print('LookupError ', e)
except IndexError as e:
    print('IndexError ', e)

```

Python 异常继承关系示例



https://docs.python.org/3/library/exceptions.html#exception-hierarchy

抛出异常

- 可以用 raise 语句来引发一个异常。异常/错误对象必须有一个名字，且它们应是 Error 或 Exception 类的子类。

raise 示例 I

```
1  class Student(object):
2      @property
3      def score(self):
4          return self._score
5
6      @score.setter
7      def score(self, value):
8          if not isinstance(value, int):
9              raise ValueError('score必须是整数!')
10         if value < 0 or value > 100:
11             raise ValueException('score必须在0到100之间')
12         self._score = value
```

raise 示例 II

```
1 s = Student()
2 s.score = 60
3 print(s.score)
4 s.score = 999 # Error!
```

异常可以自定义 I

- 例如：把以上的 score 判断条件不满足时，抛出的异常更改为自定义异常

```
1  class ScoreException(Exception):
2      def ___init___(self, msg):
3          self.msg = msg
4
5      def ___str___(self):
6          return 'ScoreException: ' + repr(self.msg)
7
8  class Student(object):
9      @property
10     def score(self):
11         return self._score
12
```

异常可以自定义 II

```
13      @score.setter
14      def score(self, value):
15          if not isinstance(value, int):
16              raise ScoreException('score必须是整数!')
17          if value < 0 or value > 100:
18              raise ScoreException('score必须在0到100之间')
19          self._score = value
20
21  try:
22      s = Student()
23      s.score = 60
24      print(s.score)
25      s.score = 999
26  except ScoreException as e:
```

异常可以自定义 III

```
27      print(e)
28
```

异常总结

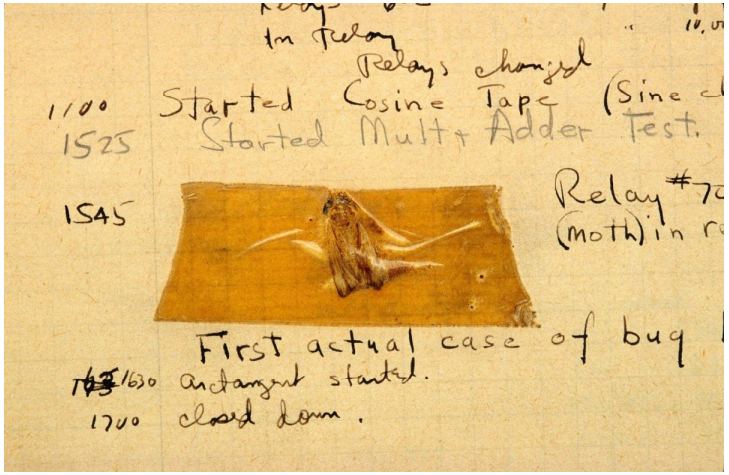
- Python 内置的 try...except...finally 用来处理错误十分方便
  - 出错时，会分析错误信息并定位错误发生的代码位置更为关键
- 程序也可以主动抛出错误，让调用者来处理相应的错误
  - 应在文档中写清楚可能会抛出哪些错误，以及错误产生的原因

调试

Bug and Debug

格蕾丝 · 赫柏 (Grace Murray Hopper)

赫柏是一位为美国海军工作的电脑专家。1945 年的一天，赫柏对 Harvard Mark II 设置好 17000 个继电器进行编程后，技术人员在进行整机运行时，它突然停止了工作。于是他们爬上去找原因，发现这台巨大的计算机内部一组继电器的触点之间有一只飞蛾，这显然是由于飞蛾受光和热的吸引，飞到了触点上，然后被高电压击死。所以在报告中，赫柏用胶条贴上飞蛾，并把“bug”来表示”一个在电脑程序里的错误”。



- 程序一次编写就能成功运行的概率很小，通常会有各种各样的错误需要调试，因此，掌握错误的调试方法非常重要。

常用的调试方法

- 观察出错的提示信息
- 利用 print 函数输出信息，观察输出结果和预期结果是否一致
- 通过日志 logging 替代 print
- 利用 assert 断言
- pdb

利用 print 进行调试

```
1 books = ['Python', 'XML', 'Information Retrieval']
2 for book in books:
3     print(book)
4
5 print('Run here!')
6 if book.price > 50:
7     print('High price.')
8
```

以上代码会抛出异常信息：

```
Traceback (most recent call last):
  File "bug.py", line 5, in <module>
    if book.price > 50:
AttributeError: 'str' object has no attribute 'price'
```

如果我们觉得前三行代码不太可能出问题，而问题很可能在后面，那

利用 logging 进行调试

print() 的结果默认输出到控制台上，不够灵活和方便，可以使用 logging 进行控制

```
1  import logging
2  logging.basicConfig(level=logging.INFO)
3
4  books = ['Python', 'XML', 'Information Retrieval']
5  for book in books:
6      logging.info(book)
7
8  logging.debug('Run here!')
9  if book.price > 50:
10     logging.warn('High price.')
11
```

assert 调试

assert 断言用于判断给定的逻辑表达式是否成立，如果不成立，就会抛出 AssertionError 异常

```
1 books = ['Python', 'XML', 'Information Retrieval']
2 assert len(books) == 3
```

启动 Python 解释器时可以用-O 参数来关闭 assert，此时的 assert 语句可看作是 pass

pdb

Python 的调试器，可以单步运行 Python 脚本，查看当前运行的代码，查看变量值

bug.py:

```
1 books = ['Python', 'XML', 'Information Retrieval']
2 for book in books:
3     print(book)
4
5 if book.price > 50:
6     print('High price.')
7
```

运行：pdb bug.py
或者：python -m pdb bug.py

pdb

- n: 执行下一条语句
- p xxx: 查看变量 xxx 的当前值
- l: 列出

pdb.set_trace()

在可能出错的地方放置 pdb.set_trace()，程序运行到该条语句时，会暂停并进入 pdb 调试环境，此时，可以用 p 指令查看变量，或者用 c 继续运行

bug2.py:

```
1 import pdb
2 books = ['Python', 'XML', 'Information Retrieval']
3 for book in books:
4     print(book)
5
6 pdb.set_trace()
7 if book.price > 50:
8     print('High price.')
9
```

单元测试

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

—END—

Web 服务器

- 最简单的 Python Web 服务器
- Flask

Simple Http Server

- 最轻便的 Web 服务器[2]：
- 启动方式：
  - python -m http.server

[2]参数 m 的作用参考：`http://www.tuicool.com/articles/jMzqYzF`

Flask

- Flask 是一种简便的基于 Python 语言的 Web 应用程序开发框架
  - Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions. And before you ask: It's BSD licensed!
- 相关中文文档可在线参考：
  - `http://dormousehole.readthedocs.io/en/latest/quickstart.html`
- 安装：
  - pip install Flask
- 文档
  - http://flask.pocoo.org/docs/0.10/.latex/Flask.pdf

Flask 简单例子

```
1  from flask import Flask
2
3  app = Flask(__name__, static_folder='.', static_url_path='')
4
5
6  @app.route('/')
7  def home():
8      return app.send_static_file('index.html')
9
10
11 @app.route('/fac/<n>')
12 def factorial(n):
13     total = 1
14     print(type(n))
15     m = int(n)
16     for i in range(m):
17         total = total*(i+1)
18     return total
19
```

Flask 简单例子 I

```
1  from flask import Flask
2  from flask import request
3
4  app = Flask(__name__, static_folder='.', static_url_path='')
5
6
7  @app.route('/')
8  def home():
9      return app.send_static_file('index.html')
10
11
12 @app.route('/echo/<thing>')
13 def echo(thing):
14     return "Say hello to my little friend: %s" % thing
15
16
17 @app.route('/fetch', methods=['GET', 'POST'])
```

Flask 简单例子 II

```
18 def fetch():
19     if request.method == 'POST':
20         url = request.form['url']
21         return url
22     return "Sorry"
23
24 app.run(port=9999, debug=True)
```

练习

实现一个 Web 程序，在网页上输入一个 url 地址，提交后，通过浏览器显示该地址所包含的所有图片。