Python代码结构

讲解如何组织代码和数据。

语法和句法

注释

在Python中使用`#`字符标记注释，从`#`开始到当前行结束的部分都是注释。可以把注释作为单独的一行或者把注释和代码放在同一行。

```
print("Hello, World!")  # 这是一个单行注释
```

把语句分成多行写

有时候一行代码太长，我们想分开，或者是出于美观、清晰的需要，可以用`\`来连接多行。

```
if (1 == 1) and \
   (2 == 2):
    print('Frivolous!')
```

使用`:`将代码块的头和身体分开

```
if (user == 'green'):       # 头
    print('hello, green')   # 身体
```

缩进

前面讲过，Python使用缩进来表示代码的逻辑结构，建议使用4个空格进行缩进（可参考PEP8 Python编码规范）。

同一行写多个语句

同一行写多个语句，使用`;`分隔。

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
# 这样可行但并不好，降低了代码的可读性
```

`if`-`elif`-`else`语句

```
today = input("Input: ")

if today == 'Mon':
    print('Today is Monday')
elif today == 'Tue':
    print('Today is Tuesday')
elif today == 'Wed':
    print('Today is Wednesday')
else:
    print('Today is a boring day')
```

`elif`和C语言中的`else if`是等效的，C语言中的`switch case`结构在Python中没有等效的实现。

C语言中使用`if`和`else`最常见的一个问题就是**else悬挂**，很多时候初学者弄不清除哪个`else`和哪个`if`是一对（C语言中，任何一个`else`与其上方最近的一个`if`是一对）。这个问题在Python中不存在，之前我们说过，Python中的缩进所起到的作用不仅仅上是排版层面上的，更是逻辑层面上的。没有了大括号，使用缩进来控制流程的层次深度，便使得程序员无需考虑悬挂问题，只要**保证同样逻辑层次的语句有相同的缩进量**即可：

```
if username_is_right:
    if passwd_is_right:
        if cash > 0:
            print("you have extra money")
        else:
            print("you are in debt")
    else:
        print("wrong password")
else:
    print("username is not existed!")
```

条件表达式（三元操作符）

语法：`X if C else Y`

```
x, y = 3, 4
small = x if x < y else y
small
```

多重比较

如果想同时进行多重比较判断，可使用布尔操作符`and`、`or`或者`not`连接来决定最终表达式的布尔取值。布尔操作符的优先级没有比较表达式的代码段高，即，表达式要先计算然后再比较。

```
x = 7
x > 5 and x < 10   # True
```

避免混淆的办法是加圆括号。

```
(x > 5) or not (x < 10)  # True
```

什么是真值

下面的情况会被认为是**False**：

布尔	False
null类型	`None`
整型	`0`
浮点型	`0.0`
空字符串	`''`
空列表	`[]`
空元组	`()`
空字典	`{}`
空集合	`set()`

剩下的都会被认为是`True`。

如果是在判断一个表达式是否，Python会先计算表达式的值，然后返回布尔型结果。

`while`循环

语法：

```
while expression:
    loop code
```

无限循环：

```
white True:
    ...
```

`for`循环

`for`循环是Python中最强大也最使用广泛的循环类型。它可以遍历序列成员，由此可用于列表解析和生成器表达式中。

语法：

```
for item_var in iterable:    # iterable可以是序列、迭代器或者可迭代的对象
    code to process item_var
```

一些例子：

```
nameList = ['Walter', 'Nicole', 'Steven']
# 使用序列项迭代
for eachName in nameList:
    print(eachName, 'Lim')

# 使用序列索引迭代
for nameIndex in range(len(nameList)):
    print("Liu,", nameList[nameIndex])

# 使用项和索引迭代
for i, eachLee in enumerate(nameList):
    print("%d %s Lee" % (i+1, eachLee))
```

`break`, `continue`, `pass`

和C语言中一样，`break`用于终止循环，可以使得流程跳出所在的`while`或者`for`循环。

和C语言中一样，`continue`用于终止当前循环，忽略剩下的语句，回到循环开始（即终止当前循环，开始下一次循环）。

顾名思义，pass就是什么都不做，通常用作占位符，在函数原型设计中很有用：

```
# main function
def main():
    pass  # 以后有功夫再来完善吧
```

迭代器和`iter()`函数

什么是迭代器

迭代器是一种特殊的数据结构，在Python中，它也是以对象的形式存在的。迭代器提供了一种遍历类序列对象的方法。对于一般的序列类型，可以利用索引从0一直迭代到序列的最后一个元素；对于字典、文件、自定义对象类型等，你可以自定义迭代方式，从而实现对这些对象类型的遍历。

可以这样理解：对于一个集体中的每一个元素，我们要遍历，那么针对这个集体的迭代器定义了遍历每一个元素的顺序或者方法。例如：0, 1, 2, 3, 4...，或者1, 3, 5, 7, 9...

如何迭代

迭代器有一个`__next__()`方法，每次调用这个方法而实现计数（计数不是通过索引实现的），在循环中，如果要获得下一个对象，迭代器自己调用`__next__()`方法，这个过程是透明的。当遍历结束后（集合中再无未访问的项）会遇到`StopIteration`异常，从而结束循环。

迭代器有一些限制，你只能向前迭代，不能后退，即**迭代是单向的**，当然，你可以独立创建一个反向的迭代器。所以迭代器不能复制，一旦你需要重新迭代某个对象，你必须重新创建一个该对象的迭代器。

`reversed()`返回一个序列对象的逆序迭代器。 `enumerate()`也可以返回迭代器。

使用迭代器

```
mylist = ['green', 'red', 'blue', 'white']
i = iter(mylist)
i.__next__()
i.__next__()
type(i.__next__())

ri = reversed(mylist)  # reversed返回一个逆序迭代器
ri.__next__()
```

```
i = iter(mylist)
while True:
    try:  # try-except用来捕获异常
        print(i.__next__())
    except StopIteration:
        break
```

注意： 在迭代一个对象的过程中最好不要去修改对象本身，否则会发生难以预料的迭代异常，使得代码具有潜在的缺陷。

`iter()`函数

1. `iter(obj)` 如果`obj`是一个序列类型，那么可以根据其索引从0开始迭代。
2. `iter(callable, sentinel)` 每次迭代调用`callable`，直至迭代的下一个值返回`sentinel`。

推导式（Comprehensions）

推导式是从一个或者多个迭代器快速简洁地创建数据结构的一种方法，最早来源于Haskell编程语言，它可以将循环和条件判断结合，从而避免语法冗长的代码。

列表推导（解析）

语法：`[expr for item_var in iterable]`

该语句的核心是for循环：

```
print([x**2 for x in range(9)])
print([ss[::-1] for ss in ('green', 'red', 'blue')])
```

扩展语法可以实现条件控制：`[expr for item_var in iterable if cond_expr]`

```
print([x for x in range(100) if x % 17 == 0])  # 100以内17的倍数

# 在对应的推导中有多个for语句
print([(x, y) for x in range(9) for y in range(9)])  # 生成笛卡尔积
```

字典推导

语法：`{key_expr: value_expr for expr in iterable}`

类似于列表推导，字典推导也有`if`条件判断以及多个`for`循环迭代语句。

```
word = 'letters'
letter_counts = {letter: word.count(letter) for letter in set(word)}
letter_counts
```

集合推导

与上面列表推导类似。

生成器表达式

生成器表达式是对列表解析的扩展，列表解析有一个不足，它一次性生成所有数据，用以创建列表，这样在数据量很大，内存很有限的时候便会造成问题。生成器表达式解决了这个问题，它一次生成一个数据，然后暂停，当下一次使用时，生产出下一个数据，工作的过程就像迭代器一样。

语法很简单：`(expr for item_var in iterable if cond_expr)`

也就是将列表的`[]`换成`()`即可：

```
type((x for x in range(10000) if x * x == (x - 3) * (x + 4)))  # generator
num = (x for x in range(10000) if x * x == (x - 3) * (x + 4))
num.__next__()
```

注意：一个生成器只能运行一次，其只在运行中产生值，不会被存下来，所以不能重新使用或者备份一个生成器。