2018 年秋-第 11 周

本周做的主要工作：

1、从头到尾再学习了一遍 python

2、学习莫烦 python 中的 pytorch 教程

3、在 pytorch 官网学习教程

现阶段：

能用 pytorch 实现简单的分类或者回归网络，比如 pytorch 官网上的手写字体识别，但对于 SSN 的复现，这些例子的支持还不足。以下记录一下值得日后复习的要点。

Python：

python 学习记录

记录一些值得复习的细节：

http://www.runoob.com/python3/python3-tutorial.html

Python3 基础语法

多行语句

Python 通常是一行写完一条语句，但如果语句很长，我们可以使用反斜杠(\)来实现多行语句，例如：

```
total = item_one + \
        item_two + \
        item_three
```

在 [], {}, 或 () 中的多行语句，不需要使用反斜杠(\)，例如：

```
total = ['item_one', 'item_two', 'item_three',
        'item_four', 'item_five']
```

- 转义符 '\'
- 反斜杠可以用来转义，使用r可以让反斜杠不发生转义。。 如 r"this is a line with \n" 则\n会显示，并不是换行。
- 按字面意义级联字符串，如"this " "is " "string"会被自动转换为this is string。
- 字符串可以用 + 运算符连接在一起，用 * 运算符重复。
- 字符串的截取的语法格式如下：`变量[头下标:尾下标:步长]`

同一行显示多条语句

Python可以在同一行中使用多条语句，语句之间使用分号(;)分割，以下是一个简单的实例：

实例(Python 3.0+)

```
#!/usr/bin/python3

import sys; x = 'runoob'; sys.stdout.write(x + '\n')
```

Print 输出

print 默认输出是换行的，如果要实现不换行需要在变量末尾加上 **end=""**：

```
# 不换行输出
print( x, end=" " )
print( y, end=" " )
print()
```

Python3 基本数据类型

多个变量赋值

Python允许你同时为多个变量赋值。例如：

```
a = b = c = 1
```

以上实例，创建一个整型对象，值为 1，从后向前赋值，三个变量被赋予相同的数值。

您也可以为多个对象指定多个变量。例如：

```
a, b, c = 1, 2, "runoob"
```

Python3 的六个标准数据类型中：

- **不可变数据（3 个）**：Number（数字）、String（字符串）、Tuple（元组）；
- **可变数据（3 个）**：List（列表）、Dictionary（字典）、Set（集合）。

内置的 type() 函数可以用来查询变量所指的对象类型。

isinstance 和 type 的区别在于：

- type()不会认为子类是一种父类类型。
- isinstance()会认为子类是一种父类类型。

您也可以使用del语句删除一些对象引用。

del语句的语法是：

```
del var1[,var2[,var3[....,varN]]]
```

- 3、数值的除法包含两个运算符：`/` 返回一个浮点数，`//` 返回一个整数。

List（列表）

List（列表） 是 Python 中使用最频繁的数据类型。

列表可以完成大多数集合类的数据结构实现。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含列表（所谓嵌套）。

列表是写在方括号 `[]` 之间、用逗号分隔开的元素列表。

Tuple（元组）

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号 `()` 里，元素之间用逗号隔开。

构造包含 0 个或 1 个元素的元组比较特殊，所以有一些额外的语法规则：

```
tup1 = ()    # 空元组
tup2 = (20,) # 一个元素，需要在元素后添加逗号
```

Set（集合）

集合（set）是由一个或数个形态各异的大小整体组成的，构成集合的事物或对象称作元素或是成员。

基本功能是进行成员关系测试和删除重复元素。

可以使用大括号 `{ }` 或者 `set()` 函数创建集合，注意：创建一个空集合必须用 `set()` 而不是 `{ }`，因为 `{ }` 是用来创建一个空字典。

创建格式：

```
parame = {value01,value02,...}
或者
set(value)
```

实例

```
#!/usr/bin/python3

student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'}

print(student)   # 输出集合，重复的元素被自动去掉

# 成员测试
if 'Rose' in student :
    print('Rose 在集合中')
else :
    print('Rose 不在集合中')


# set可以进行集合运算
a = set('abracadabra')
b = set('alacazam')
```

Dictionary（字典）

字典是一种映射类型，字典用"{ }"标识，它是一个无序的**键(key)**：**值(value)**对集合。

实例

```
#!/usr/bin/python3

dict = {}
dict['one'] = "1 - 菜鸟教程"
dict[2]     = "2 - 菜鸟工具"

tinydict = {'name': 'runoob','code':1, 'site': 'www.runoob.com'}
```

构造函数 dict() 可以直接从键值对序列中构建字典如下：

实例

```
>>>dict([('Runoob', 1), ('Google', 2), ('Taobao', 3)])
{'Taobao': 3, 'Runoob': 1, 'Google': 2}

>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}

>>> dict(Runoob=1, Google=2, Taobao=3)
{'Runoob': 1, 'Google': 2, 'Taobao': 3}
```

当键入一个多行结构时，续行是必须的。我们可以看下如下 if 语句：

```
>>> flag = True
>>> if flag :
...     print("flag 条件为 True!")
...
flag 条件为 True!
```

Python3 运算符

**	幂 - 返回x的y次幂	a**b 为10的21次方

//	取整除 - 向下取接近除数的整数	`>>> 9//2`
		`4`
		`>>> -9//2`
		`-5`

^	按位异或运算符：当两对应的二进位相异时，结果为1	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1。`~x` 类似于 `-x-1`	(~a) 输出结果 -61 ，二进制解释： 1100 0011，在一个有符号二进制数的补码形式。
<<	左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数	a >> 2 输出结果 15 ，二进制解释： 0000 1111

Python逻辑运算符

Python语言支持逻辑运算符，以下假设变量 a 为 10, b为 20:

运算符	逻辑表达式	描述	实例
and	x and y	布尔"与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。	(a and b) 返回 20。
or	x or y	布尔"或" - 如果 x 是 True，它返回 x 的值，否则它返回 y 的计算值。	(a or b) 返回 10。
not	not x	布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。	not(a and b) 返回 False

Python身份运算符

身份运算符用于比较两个对象的存储单元

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	**x is y**, 类似 **id(x) == id(y)** , 如果引用的是同一个对象则返回 True，否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	**x is not y** ， 类似 **id(a) != id(b)**。如果引用的不是同一个对象则返回结果 True，否则返回 False。

注： id() 函数用于获取对象内存地址。

实例(Python 3.0+)

```
#!/usr/bin/python3

a = 20
b = 20

if ( a is b ):
   print ("1 - a 和 b 有相同的标识")
else:
   print ("1 - a 和 b 没有相同的标识")

if ( id(a) == id(b) ):
   print ("2 - a 和 b 有相同的标识")
else:
   print ("2 - a 和 b 没有相同的标识")

# 修改变量 b 的值
b = 30
if ( a is b ):
   print ("3 - a 和 b 有相同的标识")
else:
   print ("3 - a 和 b 没有相同的标识")

if ( a is not b ):
   print ("4 - a 和 b 没有相同的标识")
else:
   print ("4 - a 和 b 有相同的标识")
```

以上实例输出结果：

```
1 - a 和 b 有相同的标识
2 - a 和 b 有相同的标识
3 - a 和 b 没有相同的标识
4 - a 和 b 没有相同的标识
```

is 与 == 区别:

is 用于判断两个变量引用对象是否为同一个， == 用于判断引用变量的值是否相等。

```
>>>a = [1, 2, 3]
>>> b = a
>>> b is a
True
>>> b == a
True
>>> b = a[:]
>>> b is a
False
>>> b == a
True
```

is 和 ==

`is` 判断两个变量是否是引用同一个内存地址。

`==` 判断两个变量是否相等。

如果不用 a = b 赋值，int 型时，在数值为 **-5~256（64位系统）**时，两个变量引用的是同一个内存地址，其他的数值就不是同一个内存地址了。

也就是，a b 在 -5~256（64位系统）时：

```
a = 100
b = 100
a is b # 返回 True
```

其他类型如列表、元祖、字典让 a、b 分别赋值一样的时：

```
a is b  # 返回False
```

在交互模式中，最后被输出的表达式结果被赋值给变量 _ 。例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

此处， _ 变量应被用户视为只读变量。

Python3 字符串

[:]	截取字符串中的一部分，遵循**左闭右开**原则，str[0,2] 是不包含第 3 个字符的。	a[1:4] 输出结果 **ell**

Python字符串格式化

Python 支持格式化字符串的输出 。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。

在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法。

实例(Python 3.0+)

```
#!/usr/bin/python3

print ("我叫 %s 今年 %d 岁!" % ('小明', 10))
```

Python3 列表

删除列表元素

可以使用 del 语句来删除列表的的元素，如下实例：

```
del list[2]
```

Python3 元组

元组中只包含一个元素时，需要在元素后面添加逗号，否则括号会被当作运算符使用：

实例(Python 3.0+)

```
>>>tup1 = (50)
>>> type(tup1)     # 不加逗号，类型为整型
<class 'int'>

>>> tup1 = (50,)
>>> type(tup1)     # 加上逗号，类型为元组
<class 'tuple'>
```

修改元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，

删除元组

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组，

Python3 字典

删除字典元素

能删单一的元素也能清空字典，清空只需一项操作。

显示删除一个字典用del命令，如下实例：

实例

```
#!/usr/bin/python3

dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}

del dict['Name'] # 删除键 'Name'
dict.clear()     # 清空字典
del dict         # 删除字典
```

字典键的特性

字典值可以是任何的 python 对象，既可以是标准的对象，也可以是用户定义的，但键不行。

两个重要的点需要记住：

1）不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住，如下实例：

2）键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行，如下实例：

end 关键字

关键字end可以用于将结果输出到同一行，或者在输出的末尾添加不同的字符，实例如下：

实例(Python 3.0+)

```
#!/usr/bin/python3

# Fibonacci series: 斐波纳契数列
# 两个元素的总和确定了下一个数
a, b = 0, 1
while b < 1000:
    print(b, end=',')
    a, b = b, a+b
```

执行以上程序，输出结果为：

```
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

range()函数

如果你需要遍历数字序列，可以使用内置range()函数。它会生成数列，例如:

实例

```
>>>for i in range(5):
...     print(i)
...
0
1
2
3
4
```

也可以使range以指定数字开始并指定不同的增量(甚至可以是负数，有时这也叫做'步长'):

负数：

实例

```
>>>for i in range(-10, -100, -30) :
    print(i)


-10
-40
-70
>>>
```

您可以结合range()和len()函数以遍历一个序列的索引,如下所示:

实例

```
>>>a = ['Google', 'Baidu', 'Runoob', 'Taobao', 'QQ']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Google
1 Baidu
2 Runoob
3 Taobao
4 QQ
>>>
```

pass 语句

Python pass是空语句，是为了保持程序结构的完整性。

pass 不做任何事情，一般用做占位语句，如下实例

实例

```
>>>while True:
...     pass  # 等待键盘中断 (Ctrl+C)
```

Python3 函数

语法

Python 定义函数使用 def 关键字，一般格式如下：

```
def 函数名（参数列表）：
    函数体
```

参数传递

在 python 中，类型属于对象，变量是没有类型的：

```
a=[1,2,3]

a="Runoob"
```

以上代码中，**[1,2,3]** 是 List 类型，**"Runoob"** 是 String 类型，而变量 a 是没有类型，她仅仅是一个对象的引用（一个指针），可以是指向 List 类型对象，也可以是指向 String 类型对象。

可更改(mutable)与不可更改(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

- **不可变类型：**变量赋值 **a=5** 后再赋值 **a=10**，这里实际是新生成一个 int 值对象 10，再让 a 指向它，而 5 被丢弃，不是改变a的值，相当于新生成了a。
- **可变类型：**变量赋值 **la=[1,2,3,4]** 后再赋值 **la[2]=5** 则是将 list la 的第三个元素值更改，本身la没有动，只是其内部的一部分值被修改了。

python 函数的参数传递：

- **不可变类型：**类似 c++ 的值传递，如 整数、字符串、元组。如fun（a），传递的只是a的值，没有影响a对象本身。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。
- **可变类型：**类似 c++ 的引用传递，如 列表，字典。如 fun（la），则是将 la 真正的传过去，修改后fun外部的la也会受影响

不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述 2 种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号 `*` 的参数会以元组(tuple)的形式导入，存放所有未命名的变量参数。

实例(Python 3.0+)

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, *vartuple ):
   "打印任何传入的参数"
   print ("输出: ")
   print (arg1)
   print (vartuple)

# 调用printinfo 函数
printinfo( 70, 60, 50 )
```

加了两个星号 `**` 的参数会以字典的形式导入。

实例(Python 3.0+)

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, **vardict ):
   "打印任何传入的参数"
   print ("输出: ")
   print (arg1)
   print (vardict)

# 调用printinfo 函数
printinfo(1, a=2,b=3)
```

声明函数时，参数中星号 `*` 可以单独出现，例如:

```
def f(a,b,*,c):
    return a+b+c
```

如果单独出现星号 `*` 后的参数必须用关键字传入。

```
>>> def f(a,b,*,c):
...     return a+b+c
...
>>> f(1,2,3)   # 报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes 2 positional arguments but 3 were given
>>> f(1,2,c=3) # 正常
6
>>>
```

匿名函数

python 使用 lambda 来创建匿名函数。

所谓匿名，意即不再使用 def 语句这样标准的形式定义一个函数。

- lambda 只是一个表达式，函数体比 def 简单很多。
- lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda表达式中封装有限的逻辑进去。
- lambda 函数拥有自己的命名空间，且不能访问自己参数列表之外或全局命名空间里的参数。
- 虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

语法

lambda 函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

如下实例：

实例(Python 3.0+)

```
#!/usr/bin/python3

# 可写函数说明
sum = lambda arg1, arg2: arg1 + arg2

# 调用sum函数
print ("相加后的值为 : ", sum( 10, 20 ))
print ("相加后的值为 : ", sum( 20, 20 ))
```

变量作用域

Python 中，程序的变量并不是在哪个位置都可以访问的，访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。Python的作用域一共有4种，分别是：

- L （Local） 局部作用域
- E （Enclosing） 闭包函数外的函数中
- G （Global） 全局作用域
- B （Built-in） 内建作用域

以 L –> E –> G –>B 的规则查找，即：在局部找不到，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再者去内建中找。

```
x = int(2.9)  # 内建作用域

g_count = 0  # 全局作用域
def outer():
    o_count = 1  # 闭包函数外的函数中
    def inner():
        i_count = 2  # 局部作用域
```

Python 中只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域，其它的代码块（如 if/elif/else/、try/except、for/while等）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问，如下代码：

global 和 nonlocal关键字

当内部作用域想修改外部作用域的变量时，就要用到global和nonlocal关键字了。

以下实例修改全局变量 num：

实例(Python 3.0+)

```
#!/usr/bin/python3

num = 1
def fun1():
    global num  # 需要使用 global 关键字声明
    print(num)
    num = 123
    print(num)
fun1()
print(num)
```

如果要修改嵌套作用域（enclosing 作用域，外层非全局作用域）中的变量则需要 nonlocal 关键字了，如下实例：

实例(Python 3.0+)

```
#!/usr/bin/python3

def outer():
    num = 10
    def inner():
        nonlocal num   # nonlocal关键字声明
        num = 100
        print(num)
    inner()
    print(num)
outer()
```

Python3 数据结构

列表

Python中列表是可变的，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

以下是 Python 中列表的方法：

方法	描述
list.append(x)	把一个元素添加到列表的结尾，相当于 a[len(a):] = [x]。
list.extend(L)	通过添加指定列表的所有元素来扩充列表，相当于 a[len(a):] = L。
list.insert(i, x)	在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 a.insert(0, x) 会插入到整个列表之前，而 a.insert(len(a), x) 相当于 a.append(x) 。
list.remove(x)	删除列表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。
list.pop([i])	从列表的指定位置移除元素，并将其返回。如果没有指定索引，a.pop()返回最后一个元素。元素随即从列表中被移除。（方法中 i 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记。）
list.clear()	移除列表中的所有项，等于del a[:]。
list.index(x)	返回列表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。
list.count(x)	返回 x 在列表中出现的次数。
list.sort()	对列表中的元素进行排序。
list.reverse()	倒排列表中的元素。
list.copy()	返回列表的浅复制，等于a[:]。

列表推导式

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。

每个列表推导式都在 for 之后跟一个表达式，然后有零到多个 for 或 if 子句。返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表。如果希望表达式推导出一个元组，就必须使用括号。

这里我们将列表中每个数值乘三，获得一个新的列表：

这里我们对序列里每一个元素逐个调用某方法：

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

我们可以用 if 子句作为过滤器：

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

我们可以用 if 子句作为过滤器：

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

在序列中遍历时，索引位置和对应值可以使用 enumerate() 函数同时得到：

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

描述

enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中。

语法

以下是 enumerate() 方法的语法:

```
enumerate(sequence, [start=0])
```

参数

- sequence -- 一个序列、迭代器或其他支持迭代对象。
- start -- 下标起始位置。

返回值

返回 enumerate(枚举) 对象。

实例

以下展示了使用 enumerate() 方法的实例：

```
>>>seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>>list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>>list(enumerate(seasons, start=1))       # 小标从 1 开始
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

同时遍历两个或更多的序列，可以使用 zip() 组合：

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}?  It is {1}.'.format(q, a))
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Python3 模块

模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用 python 标准库的方法。

__name__属性

一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用__name__属性来使该程序块仅在该模块自身运行时执行。

```
#!/usr/bin/python3
# Filename: using_name.py

if __name__ == '__main__':
   print('程序自身在运行')
else:
   print('我来自另一模块')
```

说明： 每个模块都有一个__name__属性，当其值是'__main__'时，表明该模块自身在运行，否则是被引入。

说明：**__name__** 与 **__main__** 底下是双下划线， `_ _` 是这样去掉中间的那个空格。

包

包是一种管理 Python 模块命名空间的形式，采用"点模块名称"。

比如一个模块的名称是 A.B， 那么他表示一个包 A中的子模块 B 。

就好像使用模块的时候，你不用担心不同模块之间的全局变量相互影响一样，采用点模块名称这种形式也不用担心不同库之间的模块重名的情况。

这样不同的作者都可以提供 NumPy 模块，或者是 Python 图形库。

不妨假设你想设计一套统一处理声音文件和数据的模块（或者称之为一个"包"）。

现存很多种不同的音频文件格式（基本上都是通过后缀名区分的，例如： .wav，:file:.aiff，:file:.au，），所以你需要有一组不断增加的模块，用来在不同的格式之间转换。

并且针对这些音频数据，还有很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所以你还需要一组怎么也写不完的模块来处理这些操作。

这里给出了一种可能的包结构（在分层的文件系统中）：

```
sound/                          顶层包
      __init__.py               初始化 sound 包
      formats/                  文件格式转换子包
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  声音效果子包
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  filters 子包
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

在导入一个包的时候，Python 会根据 sys.path 中的目录来寻找这个包中包含的子目录。

目录只有包含一个叫做 __init__.py 的文件才会被认作是一个包，主要是为了避免一些滥俗的名字（比如叫做 string）不小心的影响搜索路径中的有效模块。

最简单的情况，放一个空的 :file:__init__.py就可以了。当然这个文件中也可以包含一些初始化代码或者为（将在后面介绍的）__all__变量赋值。

从一个包中导入*

设想一下，如果我们使用 from sound.effects import *会发生什么？

Python 会进入文件系统，找到这个包里面所有的子模块，一个一个的把它们都导入进来。

但是很不幸，这个方法在 Windows平台上工作的就不是非常好，因为Windows是一个大小写不区分的系统。

在这类平台上，没有人敢担保一个叫做 ECHO.py 的文件导入为模块 echo 还是 Echo 甚至 ECHO。

（例如，Windows 95就很讨厌的把每一个文件的首字母大写显示）而且 DOS 的 8+3 命名规则对长模块名称的处理会把问题搞得更纠结。

为了解决这个问题，只能烦劳包作者提供一个精确的包的索引了。

导入语句遵循如下规则：如果包定义文件 __init__.py 存在一个叫做 __all__ 的列表变量，那么在使用 from package import * 的时候就把这个列表中的所有名字作为包内容导入。

作为包的作者，可别忘了在更新包之后保证 __all__ 也更新了啊。你说我就不这么做，我就不使用导入*这种用法，好吧，没问题，谁让你是老板呢。这里有一个例子，在:file:sounds/effects/__init__.py中包含如下代码:

```
__all__ = ["echo", "surround", "reverse"]
```

这表示当你使用from sound.effects import *这种用法时，你只会导入包里面这三个子模块。

如果 **__all__** 真的没有定义，那么使用**from sound.effects import** *这种语法的时候，就不会导入包 sound.effects 里的任何子模块。他只是把包sound.effects和它里面定义的所有内容导入进来（可能运行__init__.py里定义的初始化代码）。

Python3 输入和输出

str.format() 的基本使用如下:

```
>>> print('{}网址： "{}!"'.format('菜鸟教程', 'www.runoob.com'))
菜鸟教程网址： "www.runoob.com!"
```

括号及其里面的字符 (称作格式化字段) 将会被 format() 中的参数替换。

在括号中的数字用于指向传入对象在 format() 中的位置，如下所示:

```
>>> print('{0} 和 {1}'.format('Google', 'Runoob'))
Google 和 Runoob
>>> print('{1} 和 {0}'.format('Google', 'Runoob'))
Runoob 和 Google
```

如果在 format() 中使用了关键字参数, 那么它们的值会指向使用该名字的参数。

```
>>> print('{name}网址： {site}'.format(name='菜鸟教程', site='www.runoob.com'))
菜鸟教程网址： www.runoob.com
```

位置及关键字参数可以任意的结合:

```
>>> print('站点列表 {0}, {1}, 和 {other}。'.format('Google', 'Runoob',
                                                  other='Taobao'))
站点列表 Google, Runoob, 和 Taobao。
```

在 ':' 后传入一个整数, 可以保证该域至少有这么多的宽度。 用于美化表格时很有用。

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> for name, number in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, number))
...
Runoob     ==>          2
Taobao     ==>          3
Google     ==>          1
```

下面是 python 部分还剩下的需要继续复习的内容：

Python3 错误和异常
Python3 面向对象
Python3 标准库概览
Python3 实例

Pytorch

1 - Torch vs Numpy

自由地转换 numpy array 和 torch tensor ：

```
import torch
import numpy as np

np_data = np.arange(6).reshape((2, 3))

torch_data = torch.from_numpy(np_data)      # numpy array 到 torch tensor
tensor2array = torch_data.numpy()   # torch tensor 到 numpy array
```

Torch 中的数学运算：

```
torch.FloatTensor(data)    # 转换成 32 位浮点 tensor
torch.abs(tensor)
torch.sin(tensor)
torch.mean(tensor)

# matrix multiplication 矩阵点乘
data = [[1,2], [3,4]]
tensor = torch.FloatTensor(data)    # 转换成 32 位浮点 tensor
np.matmul(data, data)
torch.mm(tensor, tensor)
```

2 - 变量 (Variable)

什么是 Variable

```
import torch
from torch.autograd import Variable # torch 中 Variable 模块
tensor = torch.FloatTensor([[1,2],[3,4]])
variable = Variable(tensor, requires_grad=True)
```

Variable 计算, 梯度

v_out = torch.mean(variable*variable) 就是在计算图中添加的一个计算步骤, 计算误差反向传递的时候有他一份功劳,

```
v_out.backward()        # 模拟 v_out 的误差反向传递
print(variable.grad)        # 初始 Variable 的梯度
\'\'\'
 0.5000    1.0000
 1.5000    2.0000
\'\'\'
```

获取 Variable 里面的数据：

```
print(variable)         #    Variable 形式
print(variable.data)        # tensor 形式
```

```
print(variable.data.numpy())        # numpy 形式
```

3 - 激励函数 (Activation)

Torch 中的激励函数

平时要用到的就这几个. relu, sigmoid, tanh, softplus

```
import torch
import torch.nn.functional as F     # 激励函数都在这
from torch.autograd import Variable

# 做一些假数据来观看图像
x = torch.linspace(-5, 5, 200)  # x data (tensor), shape=(100, 1)
x = Variable(x)
```

生成不同的激励函数数据:

```
x_np = x.data.numpy()   # 换成 numpy array, 出图时用

# 几种常用的 激励函数
y_relu = F.relu(x).data.numpy()
y_sigmoid = F.sigmoid(x).data.numpy()
y_tanh = F.tanh(x).data.numpy()
y_softplus = F.softplus(x).data.numpy()
# y_softmax = F.softmax(x)  softmax 比较特殊, 不能直接显示, 不过他是关于概率的, 用于分类
```

画图的代码: (部分)

```
import matplotlib.pyplot as plt # python 的可视化模块
plt.figure(1, figsize=(8, 6))
plt.subplot(221)
plt.plot(x_np, y_relu, c=\'red\', label=\'relu\')
plt.ylim((-1, 5))
plt.legend(loc=\'best\')
```

4. Python 2.7 可以通过 import __future__ 来将 2.7 版本的 print 语句移除，让你可以 Python3.x 的 print()功能函数的形式。例如：

```
from __future__ import print_function
print('hello', end='\t')
```

The output of torchvision datasets are PILImage images of range [0, 1]. We transform them to Tensors of normalized range [-1, 1].

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

5、定义网络

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
```

```
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

6、损失函数

```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

7、更新权重

```
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```