# Python-理论

2022年5月13日 9:05

- zip(a, b)：将a, b中的元素一一对应，并组成一个元组，返回由这些元素组成的对象

eg: a = [1, 2, 3]  b = [4, 5, 6, ]

list(zip(a, b)) = [(1, 4), (2, 5), (3, 6)]

zip(*zip(a, b))：旨在将zip压缩得的对象，解压返回 a, b.

- 字符串：

re.sub(old, new, str, counts)：将字符串str中的字符 old 用 new 替换，最多替换 counts 次

str.replace(old, new, counts)：同上

list.pop(index)：将 str 转换成 list 后，删除指定位置的元素.

str.isalnum()：由数字 or 字母组成，则返回 True.

str.isdigit()：为由数字组成，则返回 True.

str.isalpha()：为由字母组成，则返回 True.

chr(s)：返回s对应 ASCII 码号　　ord(m)：返回ASCII码m对应的字母or数字.

- 类中的 self 表示的是类所对应的实例，self 不是关键字号，它也可以改成其他字母.

- 单下划线，双下划线，头尾双下划线：

__init()__：表示类的内部定义的特殊方法

__foo：表示类的私有变量，为的不类本身进行访问

_foo：表示受保护 (protected) 变量，类及其子类可以进行访问.

- 身份运算符：

is：用于判断两个变量的引用是否相同，也就是比较两个对象的存储单元 (id() 号)

==：用于判断两个引用变量的值是否相同.

• 位运算符：

>>：右移，将变量当作二进制来看

<<：左移，将变量当作二进制来看

eg：a = 60　　bin(a) = 00111100

c = a << 2　　=> 11110000

d = a >> 2　　=> 00001111

• 替换字符串中指定位置index处的元素

① 先将字符串 -> list，再替换，替换后将 list 转回字符串

l = list(str) -> l[index] = m -> str = ''.join(l)

② 切片替换

str = str[:index] + 'm' + str[index+1:]

## Python 垃圾回收机制

Python 的垃圾回收采用 引用计数 的方法

定义一个内部跟踪变量，称为引用计数器，用于记录对象被引用的次数

当引用计数器的值为 0 时，由解释器在适当的时机将垃圾对象占用的空间收回。

垃圾回收机制不仅针对引用计数器为 0 的对象，还可以处理循环引用的情况。

循环引用：两个对象互相引用，但没有其他变量引用它们

在这种情况下，解释器也会将它们回收，处理没有被引用的循环

析构函数：__del__ __del__ 在对象销毁的时候被调用，当对象不再被使用时，__del__ 方法运行。

## Python 闭包

### 什么是闭包

在函数内部再定义了一个函数，外面的函数称为外函数，内部的函数称为内函数。

闭包函数：内函数引用了外函数的变量 or 参数，称该内函数为闭包函数。

闭包：闭包是一种状态，闭包中既包含内函数又包括闭包外的参数。

- 闭包的作用
  - 保存外部函数的变量，该变量不会随着外部函数调用结束而销毁。
  - 形成闭包之后，闭包函数会获得一个 `__closure__` 属性用于存储闭包变量。
  - 一般情况下，如果一个函数结束，那么函数内部所有东西都会被释放掉，还给内存。

    但形成闭包后，当外部函数结束时发现有当前的临时变量会被内部函数引用，就会把该变量与内部函数先绑定，然后再结束。
- 闭包形成的条件：
  1. 函数的嵌套定义
  2. 内函数使用外函数的变量或参数
  3. 外部函数的返回值是内函数的引用
- 闭包中内函数修改外函数变量的方法：
  1. `nonlocal` 关键字声明：表示该变量不是当前局部变量空间的变量，需要向上一层变量空间寻找
  2. 将闭包变量修改成可变数据类型，比如列表

注：闭包变量为同一份，每次开启内函数都是在使用同一份闭包变量。

```
1  #修改闭包变量的实例
2  # outer是外部函数 a和b都是外函数的临时变量
3  def outer( a ):
4      b = 10  # a和b都是闭包变量
5      c = [a] #这里对应修改闭包变量的方法2
6      # inner是内函数
7      def inner():
8          #内函数中想修改闭包变量
9          # 方法1 nonlocal关键字声明
10         nonlocal  b
11         b+=1
12         # 方法二、把闭包变量修改成可变数据类型 比如列表
13         c[0] += 1
14         print(c[0])
15         print(b)
16     # 外函数的返回值是内函数的引用
17     return inner
18
19 if __name__ == '__main__':
20
21     demo = outer(5)
22     demo() # 6  11
```

- 闭包的用途：装饰器，面向对象，实现单例模式.

- 闭包举例：

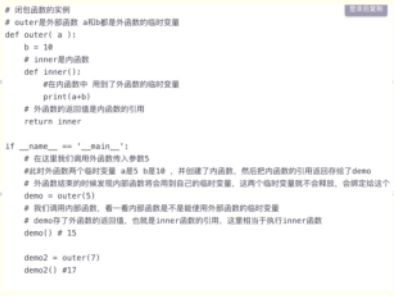
```
# 函数嵌套
def func_out():
    # 定义外部函数
    num1 = 10

    def func_inner(num2):
        # 定义内部函数
        # 内部函数必须使用了外部函数的变量
        # 这里使用了外部的变量num1
        result = num1 + num2
        print("结果:", result)

    # 外部函数返回内部函数 这个使用了外部函数变量的内部函数称为闭包
    return func_inner


# 获取闭包对象
# 这个new_func 就是闭包
# 这里的new_func = func_inner
new_func = func_out()
# 执行闭包
new_func(1)
```

CSDN @墨鱼A



```
# 闭包函数的实例
# outer是外部函数 a和b都是外函数的临时变量
def outer( a ):
    b = 10
    # inner是内函数
    def inner():
        #在内函数中 用到了外函数的临时变量
        print(a+b)
    # 外函数的返回值是内函数的引用
    return inner

if __name__ == '__main__':
    # 在这里我们调用外函数传入参数5
    #此时外函数两个临时变量 a是5 b是10 ， 并创建了内函数，然后把内函数的引用返回存给了demo
    # 外函数结束的时候发现内部函数将会用到自己的临时变量，这两个临时变量就不会释放，会绑定给这个
    demo = outer(5)
    # 我们调用内部函数，看一看内部函数是不是能使用外部函数的临时变量
    # demo存了外函数的返回值，也就是inner函数的引用，这里相当于执行inner函数
    demo() # 15

    demo2 = outer(7)
    demo2() #17
```

## Python 装饰器.

- 什么是装饰器.

装饰器本质上也是一个闭包，装饰器在不改变原函数的情况下，对原有函数进行额外的功能扩展

- 装饰器的形成条件

① 不修改已有函数的源代码

② 不修改已有函数的调用方式

③ 给已有函数增加额外的功能

- 装饰器与闭包的区别

装饰器属于闭包函数，但装饰器的参数有且仅有一个是函数类型

- 装饰器的用途

① 添加计时逻辑，检测函数性能

② 给函数加事物能力

③ 权限控制

④ 缓存网页

```python
import time

# 定义装饰器
def decorator(func):
    def inner():
        # 内部函数对已有函数进行装饰
        # 获取时间 距离1970-1-1:0:0:1的时间差
        begin = time.time()
        func()  # 这里的func() 相当于最早下面的函数work
        end = time.time()
        result = end - begin
        print("函数执行完成耗时", result)

    return inner


@decorator  # work = decorator(work) work = inner
def work():
    for i in range(10000):
        print(i)

# 调用函数
work()
```



- 装饰器实现的语法：

① 不带参数的装饰器

2 不带参数的装饰器(装饰器,被装饰函数都不带参数)

```python
import time
def showtime(func):
    def wrapper():
        start_time = time.time()
        func()
        end_time = time.time()
        print('spend is {}'.format(end_time - start_time))

    return wrapper

@showtime  #foo = showtime(foo)
def foo():
    print('foo..')
    time.sleep(3)

@showtime #doo = showtime(doo)
def doo():
    print('doo..')
    time.sleep(2)

foo()
doo()
```

② 被装饰函数带参数的装饰器

3 带参数的被装饰的函数

```python
import time
def showtime(func):
    def wrapper(a, b):
        start_time = time.time()
        func(a,b)
        end_time = time.time()
        print('spend is {}'.format(end_time - start_time))

    return wrapper

@showtime #add = showtime(add)
def add(a, b):
    print(a+b)
    time.sleep(1)

@showtime #sub = showtime(sub)
def sub(a,b):
    print(a-b)
    time.sleep(1)

add(5,4)
sub(3,2)
```

③ 有参数的装饰器

4 带参数的装饰器(装饰函数)

```python
import time
def time_logger(flag = 0):
    def showtime(func):
        def wrapper(a, b):
            start_time = time.time()
            func(a,b)
            end_time = time.time()
            print('spend is {}'.format(end_time - start_time))

            if flag:
                print('调用操作保留信息')

        return wrapper

    return showtime

@time_logger(1)  #等同时间装饰器showtime,add = showtime(add)
def add(a, b):
    print(a+b)
    time.sleep(1)

add(3,4)
```

④ 类装饰器

5 类装饰器:一般依靠类内部的__call__方法

```python
import time
class Foo(object):
    def __init__(self, func):
        self._func = func

    def __call__(self):
        start_time = time.time()
        self._func()
        end_time = time.time()
        print('spend is {}'.format(end_time - start_time))

@Foo  #bar = Foo(bar)
def bar():
    print('bar..')
    time.sleep(2)

bar()
```

- 常见的内置装饰器：

  ① @staticmethod：把类中定义的实例方法变为静态方法

  实例方法第1个参数self是self，只能被实例对象调用。静态方法和类方法可以把类or类的实例对象调用

  如果用staticmethod，当函数参数有self时，直接报错，当做一个普通函数使用。

  ② @classmethod：把类中定义的实例方法变为类方法

  类方法第一个参数都是表示当前类的cls参数

  ③ @property：把类中定义的实例方法变为属性

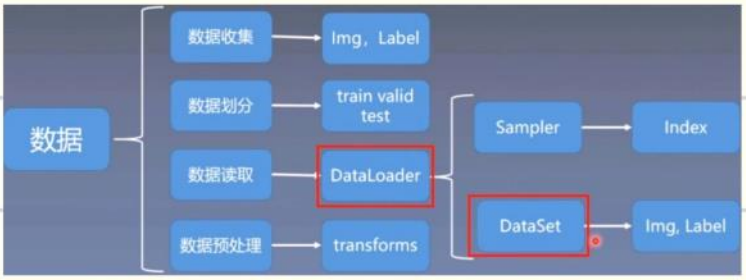
- 偏函数：

  - 函数：表示函数的数据，就是描述函数属性的信息
  - 使用装饰器位函数传参：因为return的是经过闭包封装后的新函数
  - 偏函数：利用 @functools.wraps(func)

- 高阶装饰器：

  内函数调用外函数的参数，外函数返回内函数的引用，在不改变原函数源代码的基础上扩展新功能。

- PyTorch 数据加载与处理：




- Dataset类：

torch.utils.data.Dataset 是一个抽象类，用户想要加载自定义的数据只需要继承这个类，并且覆写其中的两个方法即可：

1. __len__： 覆写这个方法使得 len(dataset) 可以返回整个数据集的大小
2. __getitem__： 覆写这个方法使得 dataset[i] 可以返回数据集中第 i 个样本
3. 不覆写这两个方法会直接返回错误，其源码如下：

```
def __getitem__(self, index):
    raise NotImplementedError

def __len__(self):
    raise NotImplementedError
```

登录后复制

```
from torch.utils.data import DataLoader, Dataset
from skimage import io, transform
import matplotlib.pyplot as plt
import os
import torch
from torchvision import transforms
import numpy as np

class AnimalData(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return 20

    def __getitem__(self, idx):
        filenames = os.listdir(self.root_dir)
        filename = filenames[idx]
        img = io.imread(os.path.join(self.root_dir, filename))
        # print filename[:-5]
        if (int(filename[:-5]) > 10):
            lable = np.array([0])
        else:
            lable = np.array([1])
        sample = {'image': img, 'lable':lable}

        if self.transform:
            sample = self.transform(sample)
        return sample
```

- Dataloader

Dataloader也是对Dataset的读取工作

torch.utils.data.DataLoader()：构建可迭代的数据加载器

- 收获：原始数据 → Dataset → 通过 DataLoader 批量数据的读取操作

## 二、Dataloader

torch.utils.data.DataLoader(): 构建可迭代的数据装载器, 我们在训练的时候，每一个for循环，每一次iteration，就是从DataLoader中获取一个batch_size大小的数据的。

```
DataLoader( dataset,
            batch_size=1,
            shuffle=False,
            sampler=None,
            batch_sampler=None,
            num_workers=0,
            collate_fn=None,
            pin_memory=False,
            drop_last=False,
            timeout=0,
            worker_init_fn=None,
            multiprocessing_context=None)
```

DataLoader的参数很多，但我们常用的主要有5个：

dataset: Dataset类， 决定数据从哪读取以及如何读取
bathsize: 批大小
num_works: 是否多进程读取机制
shuffle: 每个epoch是否乱序
drop_last: 当样本数不能被batchsize整除时， 是否舍弃最后一批数据

- torchvision:

  torchvision是pytorch官方用来处理图像的库

  - torchvision.datasets: 包含了常用的图像数据集 (Cifar10, ImageNet, MNIST等)
  - torchvision.models: 包含了常用的网络模型 (ResNet, AlexNet, VGG, DenseNet等)
  - torchvision.transforms: 包含了常用的图像转换操作, 用作数据处理和图像增强

```
print('########## 利用torchvision创建数据加载器 ##########')
data_transform = transforms.Compose([
    transforms.RandomSizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0
# ants_bees_dataset = datasets.ImageFolder(root='ants_bees_data/train', t
# dataset_loader = DataLoader(ants_bees_dataset, batch_size=4, shuffle=Tr
```