《Django 教程》

- 讲师: 魏明择
- 时间: 2019

目录

day16

文件 File

什么是文件

- 文件是用于数据存储的单位。
- 文件通常用来长期存储数据。
- 文件中的数据是以字节为单位进行顺序存储的

文件的操作流程

1. 打开文件
2. 读/写文件

3. 关闭文件

> 任何的操作系统，一个应用程序同时打开文件的数量有最大数限制

文件的打开和关闭(提示)

- 文件需要在使用时先打开文件才能读写
- 在不需要读写文件时，应及关闭文件以释放系统资源

文件的打开函数

```
open(file, mode='rt') # 用于打开一个文件，返回此文件流对象.
                      # 如果打开文件失败，则会触发OSError 错误!
                      # 如果要打开的文件不存在，则会触发FileNotFoundError 错误!
```

文件的打开函数

字符	含义
open(file, mode='rt')	用于打开一个文件，返回此文件流对象. 如果打开文件失败，则会触发OSError 错误! 如果要打开的文件不存在，则会触发FileNotFoundError 错误!

文件的关闭方法

```
F.close()             # 关闭文件,释放系统资源
```

文件的打开和关闭示例

```
# file_open.py
try:
    file = open("while.py") # 打开文件
    file.close()
except OSError:
    print("打开文件失败")
```

mode 模式字符的含义

字符	含义

字符	含义
'r'	以只读方式打开(默认)
'w'	以只写方式打开，删除原有文件内容(如果文件不存在，则创建该文件并以只写方式打开)
'x'	创建一个新文件, 并以写模式打开这个文件,如果文件存在则会产生"FileExistsError"错误
'a'	以只写文件打开一个文件，如果有原文件则追加到文件末尾
'b'	用二进制模式打开
't'	文本文件模式打开 (默认)
'+'	为更新内容打开一个磁盘文件 (可读可写)

- 缺省模式是 'rt'
- 'w+b' 可以实现二进制随机读写，当打开文件时，文件内容将被清零
- 'r+b' 以二进制读和更新模式打开文件,打开文件时不会清空文件内容
- 'r+' 以文本模式读和更新模式打开文件,打开文件时不会清空文件内容

python 文件常用方法:

方法	说明
F.close()	关闭文件(关闭后文件不能再读写会发生ValueError错误)
读取数据方法	
F.read(size=-1)	从一个文件流中最多读取size个字符(文本文件)或字节(二进制文件),如果不给出参数，则默认读取文件中全部的内容并返回
F.readline()	读取一行数据, 如果到达文件尾则返回空行
F.readlines(max_chars=-1)	返回每行字符串的列表,max_chars为最大字符(或字节)数
写入数据方法	
F.write(text)	写一个字符串到文件流中，返回写入的字符数(文本文件)或字节数(二进制文件)
F.writelines(lines)	将字符串的列表或字符串的列表中的内容写入文件
F.tell()	返回当前文件流读写指针的绝对位置(字节为单位)
F.seek(offset, whence=0)	改变数据流读写指针的位置，返回新的绝对位置
F.flush()	把写入文件对象的缓存内容写入到磁盘

文本文件操作：

Python文本文件模式:

- 操作模式字符: 't'

文本文件操作说明:

- 对文本文件的读写需要用字符串(str)进行读取和写入数据
- 默认文件中存储的都为字符数据，在读定中会自动进行编解码操作
- 以行为单位分隔,在python内部统一用'\n'作为换行符进行分隔

各操作系统的换行符:

- Linux换行符: '\n'
- Windows换行符: '\r\n'
- 旧的Macintosh换行符:'\r'
- 新的Mac OS换行符: '\n'

文本文件操作说明

- 文本文件模式下,各操作系统的换行符在读入python内部时转换为字符串'\n'
- 字符串中的'\n'在写入字符串时会自动转为各操作系统的换行符

文件迭代读取

- 文件流对象是可迭代对象,迭代过程将以换行符'\n'作为分隔符依次获取

```
f = open("hello.py")
for line in f:
print(line)
```

day16 pm

二进制文件操作：

- 二进制文件模式字符: 'b'
- 默认文件中存储的都以字节(byte)为单位的数据，通常有人为规定的格式
- 对二进制文件的读写需要用字节串(bytes)进行操作

什么是二进制文件：

文件中以字节(byte)为单位存储进行读写操作,不以字符串为单位进行读写的的文件操作方式

区分内容的文件读写方法

F.read() 返回类型:

对于文本文件，F.read()返回类型 为 字符串(str) 对于二进制文件，F.read()返回类型为 字节串(bytes)

F.write(x)

以十六进制方式查看文件内容的命令： $ xxd 文件名 例如： $ xxd bbb.txt

F.seek方法()

作用: 设置文件的读写位置 F.seek(偏移量, whence=相对位置) 偏移量： 大于0的数代表向文件末尾方向移动 小于0代表向文件头方向移动 相对位置: 0代表从文件头开始偏移 1代表从当前读写位置开始偏移 2代表从文件尾开始偏移 作用: 改变当前文件的读写位置

F.tell方法:

> 作用：
> 返回当前文件的读写位置(从文件头以字节为单位)

见tell.py

汉字编码

汉字编码的种类

- 国标系列:
  - GB18030 (二字节或四字节编码)
    - GBK (二字节编码)
      - GB2312 (二字节编码)
  - (windows常用)
- 国际标准: UNICODE <-> UTF-8
  - (Linux / Mac OS X / IOS / Android等常用)

汉字编码问题:

- 十字汉字是多少个字节?
  - 答: GBK: 20个字节, UTF-8:30个字节

GB2312-80编码:

- 1980年发布
- 用两个字节进行编码,编码范围(A1A1~FEFE)
- 包含:汉字6763个，和拉丁字母、希腊字母、日文平假名及片假名字母、俄语西里尔字母在内的682个全角字符

GBK编码:

- 1995年12月1日制订
- 用两个字节进行编码,编码范围(8140~FEFE)（剔除xx7F）共收录了21003个汉字
  - 第一个字节81-FE
  - 第二个字节40-7E, 80-FE
- 完全兼容GB2312-80标准

GB18030-2005编码:

- 2005年制订,收录了27533个汉字:
- 用两个字节或四字节进行编码
- 两个字节编码范围(8140~FEFE)（剔除xx7F）共收录了21003个汉字
  - 第一个字节81-FE
  - 第二个字节40-7E, 80-FE
- 四个字节编码:
  - 第一字节0x81-0x82
  - 第二字节0x30-0x39
  - 第三字节0x81-0xFE
  - 第四字节0x30-0x39
  - 四字节编码部分 共6530
- 27533 = 两字节部分:21003 + 四字节部分:6530汉字
- 双字节部分:6763+6080+8160=21003个汉字(GBK的21003个汉字)

Unicode编码

- Unicode16
  - 16位统一编码(两字节0~65535)
- Unicode32
  - 32位统一编码(四字节0~4294967295)

Unicode16编码(两字节0 ~ 2 ** 32-1)

```
所有字符用2字节编码(包含ASCII)
'A'  ASCII    0x41(1字节)
'A'  UNICODE  0x0041(2字节)
'ABC' -->bytes b'\x00\x41\x00\x42\x00\x43'
```

UTF-8（8-bit Unicode Transformation Format）

```
Unicode  <<--->> UTF8 互转
0000 ~ 007F 一字节(ASCII)
0080 ~ 07FF 二字节
0800 ~ FFFF 三字节(汉字落在此区)
...
```

详见： https://baike.baidu.com/item/UTF-8

Python 编码(encode)字符串:

```
'gb2312'
'gbk'
'gb18030'
'utf-8'
'ascii'
...
```

编码注释:

```
在源文件的第一行或第二行写入如下内容:
  # -*- coding:gbk -*-
  # 设置源文件编码格式为:gbk
或
  # -*- coding:utf-8 -*-
  # 设置源文件编码格式为:UTF-8
```

day17

面向对象编程 Object-Oriented Programming

什么是对象

- 对象是指现实中的物体或实体

什么是面向对象

- 把一切看成对象（实例），用各种对象之间的关系来描述事务。

对象都有什么特征

- 对象有很多属性(名词) 姓名, 年龄, 性别,
- 对象有很多行为(动作,动词) 学习,吃饭,睡觉,踢球, 工作

什么是类:

- 拥有相同属性和行为的对象分为一组,即为一个类
- 类是用来描述对象的工具,用类可以创建此类的对象(实例)

面向对象 示意

```
车(类)  ------>> BYD  E6(京A.88888) 实例, 对象
       \
        \.--->> BMW  X5(京B.00000) 实例(对象)
```

```
狗(类)  ------>> 小京巴(户籍号:000001)
       \
        \.--->> 导盲犬（户籍号:000002）


int(类)  ---->> 100 （对象）
        \
         \.-->> 200 （对象）
```

类的创建class语句

类的创建语句语法:

```
class 类名 (继承列表):
    "类文档字符串"
    实例方法(类内的函数method) 定义
    类变量(class variable) 定义
    类方法(@classmethod) 定义
    静态方法(@staticmethod) 定义
```

类的作用:

- 创建一个类
- 类用于描述对象的行为和属性
- 类用于创建此类的一个或多个对象（实例）

- 继承列表可以省略，省略继承列表表示类继承自object.

类的创建示例:

```
# file : class.py
class Dog:  # 定义一个Dog类
    pass
```

类的创建的说明:

- 类名必须为标识符(与变量的命名相同,建议首字母大写)
- 类名实质上就是变量，它绑定一个类

构造函数

构造函数调用表达式

```
类名([创建传参列表])
```

构造函数作用

- 创建这个类的实例对象，并返回此实例对象的引用关系

用类创建对象示例:

```
# file : constructor.py
class Dog:
    pass

# 创建第一个实例:
dog1 = Dog()
print(id(dog1))  # 打印这个对象的ID
# 创建第二个实例对象
dog2 = Dog()  # dog2 绑定一个Dog类型的对象
print(id(dog2))

lst1 = list()
print(id(lst1))
lst2 = list()
print(id(lst2))
```

实例说明

- 实例有自己的作用域和名字空间,可以为该实例添加实例变量（也叫属性)
- 实例可以调用类方法和实例方法
- 实例可以访问类变量和实例变量

实例方法(instance method)

```
class 类名(继承列表):
    def 实例方法名(self, 参数1, 参数2, ...):
        "文档字符串"
        语句块
```

实例方法的作用

- 用于描述一个对象的行为,让此类型的全部对象都拥有相同的行为

实例方法说明

- 实例方法的实质是函数，是定义在类内的函数
- 实例方法至少有一个形参，第一个形参绑定调用这个方法的实例,一般命名为"self"

实例方法的调用语法

```
实例.实例方法名(调用传参)
# 或
类名.实例方法名(实例, 调用传参)
```

带有实例方法的简单的Dog类

```
# file: instance_method.py
class Dog:
    """这是一个种小动物的定义
    这种动物是狗(犬)类，用于创建各种各样的小狗
    """
    def eat(self, food):
        '''此方法用来描述小狗吃东西的行为'''
        print("小狗正在吃", food)
    def sleep(self, hour):
        print("小狗睡了", hour, "小时!")
    def play(self, obj):
        print("小狗正在玩", obj)


dog1 = Dog()
dog1.eat("骨头")
dog1.sleep(1)
dog1.play('球')

dog2 = Dog()
dog2.eat("窝头")
dog2.sleep(2)
dog2.play('飞盘')

>>> help(Dog) # 可以看到Dog类的文档信息
```

实例属性 attribute（也叫实例变量)

- 每个实例可以有自己的变量，称为实例变量(也叫属性)

属性的使用语法

```
实例.属性名
```

属性的赋值规则

- 首次为属性赋值则创建此属性.
- 再次为属性赋值则改变属性的绑定关系.

作用

- 记录每个对象自身的数据

属性使用示例:

```
# file : attribute.py
class Dog:
    def eat(self, food):
         print(self.color, '的', self.kinds, '正在吃', food)
    pass
# 创建一个实例:
dog1 = Dog()
dog1.kinds = "京巴"  # 添加属性
dog1.color = "白色"
dog1.color = "黄色"  # 改变属性的绑定关系
print(dog1.color, '的', dog1.kinds)

dog2 = Dog()
dog2.kinds = "藏獒"
dog2.color = "棕色"
print(dog2.color, '的', dog2.kinds)
```

实例方法和实例属性(实例变量)结合在一起用:

```
class Dog:
    def eat(self, food):
        print(self.color, '的',
              self.kinds, '正在吃', food)


# 创建第一个对象
dog1 = Dog()
dog1.kinds = '京巴'  # 添加属性kinds
dog1.color = '白色'  # 添加属性color
# print(dog1.color, '的', dog1.kinds)  # 访问属性
dog1.eat("骨头")

dog2 = Dog()
```

```
dog2.kinds = '牧羊犬'
dog2.color = '灰色'
# print(dog2.color, '的', dog2.kinds)  # 访问属性
dog2.eat('包子')
```

day17 pm

del 语句删除实例属性

```
del 对象.实例属性名
```

del 语句 最终总结:

```
1) 删除变量
2) 删除列表中的元素
3) 删除字典中的键值对
4) 删除对象的属性(也叫实例变量)
```

点(.) 在模块中的用法(选讲)

```
模块.属性名
模块.函数名
模块.类名
```

初始化方法

初始化方法的作用:

- 对新创建的对象添加属性

初始化方法的语法格式:

```
class 类名(继承列表):
    def __init__(self[, 形参列表]):
        语句块
# [] 代表其中的内容可省略
```

初始化方法的说明:

- 初始化方法名必须为__init__ 不可改变
- 初始化方法会在构造函数创建实例后自动调用,且将实例自身通过第一个参数self传入__init__方法
- 构造函数的实参将通过__init__方法的 参数列表 传入到 __init__方法中
- 初始化方法内如果需要return语句返回，则必须返回None

初始化方法示例:

```
file : init_method.py
class Car:
    def __init__(self, c, b, m):
        self.color = c  # 颜色
        self.brand = b  # 品牌
        self.model = m  # 型号
    def run(self, speed):
        print(self.color, "的", self.brand, self.model, "正在以", speed, "公
里/小时的速度行驶")
    def change_color(self, c):
        self.color = c

a4 = Car("红色", "奥迪", "A4")
a4.run(199)
a4.change_color("黑色")
a4.run(230)
```

析构方法

```
def __del__(self)
```

- 析构函数在对象被销毁时被自动调用
- python语言建议不要在对象销毁时做任何事情,因为销毁的时间难以确定

析构方法示例

```
# file : del_method.py
class Car:
    def __init__(self, name):
        self.name = name
        print("汽车", name, "对象已创建!")
    def __del__(self):
        print("汽车", self.name, "对象已销毁")

c1 = Car("BYD E6")
del c1
```

预置实例属性

`__dict__` 属性

- `__dict__` 属性绑定一个存储此实例自身属性的字典

`__dict__` 属性示例：

```
class Dog:
    pass

dog1 = Dog()
print(dog1.__dict__)
dog1.kinds = "京巴"
print(dog1.__dict__)
dog1.color = "白色"
print(dog1.__dict__)
```

`__class__` 属性

- `__class__` 属性绑定创建此实例的类

`__class__` 属性作用

- 可以借助于此属性来访问创建此实例的类

示例：

```
class Dog:
    pass

dog1 = Dog()
print(dog1.__class__)
dog2 = dog1.__class__()
print(dog2.__class__)
```

面向对象综合示例：

```
有两个人：
  1.
    姓名：张三
    年龄：35
  2.
```

```
    姓名: 李四
    年龄: 8
  行为:
    1. 教别人学东西 teach
    2. 赚钱 work
    3. 借钱 borrow
    4. 显示自己的信息 show_info
事情:
   张三 教 李四 学 python
   李四 教 张三 学 王者荣耀
   张三 上班赚了 1000元钱
   李四向张三借了 200元
   35 岁的 张三 有钱 800 元，它学会的技能: 王者荣耀
   8 岁的 李四 有钱 200 元，它学会的技能: python
# file : object_relative.py
class Human:
    '''人类,用于描述人的行为和属性'''
    def __init__(self, n, a):
        self.name = n  # 姓名
        self.age = a  # 年龄
        self.money = 0  # 钱数为0

    def teach(self, other, subject):
        print(self.name, "教", other.name,
              '学', subject)

    def works(self, money):
        self.money += money
        print(self.name, '工作赚了', money, '元钱')

    def borrow(self, other, money):
        '描述一个人向其它人借钱的行为,它人有钱必借,不够不借'
        if other.money > money:
            print(other.name, "借给", self.name,
                money, '元钱')
            other.money -= money
            self.money += money
        else:
            print(other.name, '不借给', self.name)

    def show_info(self):
        print(self.age, '岁的', self.name,
            '存有', self.money, '元线')

# 以下的类的使用
zhang3 = Human('张三', 35)
li4 = Human('李四', 8)
# 张三 教 李四 学 python
zhang3.teach(li4, 'Python')
# 李四 教 张三 学 王者荣耀
li4.teach(zhang3, '跳皮筋')
```

```
# 张三 上班赚了 1000元钱
zhang3.works(1000)
# 李四向张三借了 200元
li4.borrow(zhang3, 200)
# 35 岁的 张三 有钱 800 元，它学会的技能: 王者荣耀
zhang3.show_info()
# 8 岁的 李四 有钱 200 元，它学会的技能: Python
li4.show_info()
```

用于类的函数

函数	说明
isinstance(obj, class_or_tuple)	返回这个对象obj 是否是 某个类的对象,或者某些类中的一个类的对象,如果是返回True,否则返回False
type(obj)	返回对象的类型