《Django 教程》

- 讲师: 魏明择
- 时间: 2019

目录

day14

异常(基础) exception

什么是错误

- 错误是指由于逻辑或语法等导致一个程序已无法正常执行的问题

错误的特点

- 有些错误是无法预知的

什么是异常

- 异常是程序出错时标识的一种状态

- 当异常发生时，程序不会再向下执行，而转去调用此函数的地方待处理此错误并恢复为正常状态.

异常作用：

- 用作信号, 通知上层调用者有错误产生需要处理

try语句的两种语法

```
try-except语句
try-finally语句
```

try-except语句

try-except语法

```
try:
    可能触发异常的语句
except 错误类型1 [as 变量1]:
    异常处理语句1
except 错误类型2 [as 变量2]:
    异常处理语句2
except (错误类型3,错误类型4) [as 变量3]:
    异常处理语句3
...
except:
    异常处理语句other
else:
    未发生异常语句
finally:
    最终语句
```

作用：

- 偿试捕获异常，得以异常通知，将程序由异常流程转为正常流程并继续执行

try-except语法说明：

- as 子句是用于绑定错误对象的变量，可以省略
- except子句可以有一个或多个，但至少要有一个
- else子句最多只能有一个，也可以省略不写
- finally子句最多只能有一个，也可以省略不写

- try/except/else/finally中的语句要进行缩进来表明语句的所属关系

try-except 处理说明

- except 子句用来捕获和处理当某种类型的错误并处理异常，把程序由异常状态改变为正常状态
- except 子句会根据错误的类型进行匹配，如匹配成功则调用异常处理语句进行处理，然后程序转入正常状态.
- 如果没有匹配到任何错误类型，则程序的异常状态会继续下出,并向上层(调用处)传递
- 如果没有异常，则执行else子句中的语句
- 最后执行finally子句的中的语句
- 如果没有相应的错误类型与之匹配,则执行空类型的except子句(except:)异常处理语句,之后程序进入正常状态.

异常示例

```
# file : try_except.py
def div_apple(n):
    print("%d个苹果你想分给几个人?" % n)
    s = input("请输入人数: ")
    cnt = int(s)  # <<-- 可能触发ValueError错误!
    result = n/cnt  # <<-- 可能触发ZeroDivisionError 错误!
    print("每个人分了", result, "个苹果")

try:
    print("开始分苹果")
    div_apple(10)
    print("分苹果完成")
except ZeroDivisionError:
    print("发生被0除的错误!, 转为正常状态")
except ValueError:
    print("发生ValueErorr类型的错误!,转为正常状态!")
except:
    print("有其它类型的错误发生，在这里已经处理，并转为正常状态")
else:
    print("try语句内没有出现错误")
finally:
    print("try语句已经执行完毕")

print("程序正常执行并完成任务! ")
```

异常示例2

```
try:
    v = 3/0
    print("3能够被0整除")
except ZeroDivisionError:
```

```
    print("发生被0除的错误!")

print("try语句执行完毕，程序正常结束")
```

Python全部的错误类型

错误类型	说明
ZeroDivisionError	除(或取模)零 (所有数据类型)
ValueError	传入无效的参数
AssertionError	断言语句失败
StopIteration	迭代器没有更多的值
IndexError	序列中没有此索引(index)
IndentationError	缩进错误
OSError	输入/输出操作失败
ImportError	导入模块/对象失败
NameError	未声明/初始化对象 (没有属性)
AttributeError	对象没有这个属性
GeneratorExit	生成器(generator)发生异常来通知退出
TypeError	对类型无效的操作
KeyboardInterrupt	用户中断执行(通常是输入^C)
OverflowError	数值运算超出最大限制
FloatingPointError	浮点计算错误
BaseException	所有异常的基类
SystemExit	解释器请求退出
Exception	常规错误的基类
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
EOFError	没有内建输入,到达EOF 标记
EnvironmentError	操作系统错误的基类
WindowsError	系统调用失败
LookupError	无效数据查询的基类
KeyError	映射中没有这个键

错误类型	说明
MemoryError	内存溢出错误(对于Python 解释器不是致命的)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError Python	语法错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
以下为警告类型	
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

详见：help(**builtins**)

day14 pm

try-finally语句

try-finally语法

```
try:
    可能触发异常的语句
finally:
    最终语句
```

try-finally语法说明：

- finally子句不可以省略
- 一定不存在except子句

try-finally语句的作用：

- 通常用try-finally语句来做触发异常时必须要的处理的事情 无论异常是否发生，finally子句都会被执行
- try-finally语句不会改变程序的(正常/异常)状态

try-finally语法示例：

```
# file : try_finally.py
# 以煎蛋,打开燃气和关闭燃气
def fry_egg():
    print("打开天燃气点然...")
    try:
        count = int(input("请输入鸡蛋个数: "))
        print("完成煎鸡蛋,共煎了%d个鸡蛋" % count )
    finally:
        print("关闭天然气")
try:
    fry_egg()
except ValueError:
    print("煎蛋中发生错误!")
```

raise 语句

raise 语句的作用

- 触发一个错误，让程序进入异常状态
- 发送错误通知给调用者

raise 语句的语法

```
raise 异常类型
# 或
raise 异常对象
# 或
raise  # 重新触发上一次异常
```

raise 语句的示例1

```
# file:  raise.py
def make_except():
    print("begin")
    # raise ValueError
    raise ValueError("值错误!")
    print("end")

try:
    make_except()
    print("....")
except ValueError as e:
    print("发生了值错误，已处理")
    print(e)

print("程序结束")
```

示例2

```
def get_score():
    score = int(input("请输入您的成绩: "))
    if score < 0:  # 如果成绩小于零分
        raise ValueError
    if score > 150:
        raise ValueError("超出了最高成绩")
    return score

try:
    score = 0
    score = get_score()
except ValueError as err:
    print("出现值错误!, 错误的对象是:", err)
print("学生的成绩是:", score)
```

assert 语句（断言语句）

assert 语句的语法

```
assert 真值表达式, 错误数据(通常是字符串)
```

assert 语句的作用

- 当真值表达式为False时，用错误数据创建一个AssertionError类型的错误，并进入异常状态

等同于:

```
if 真值表达式 == False:
   raise AssertionError(错误数据)
```

assert 语句的示例1

```
# 获取学生成绩:
def get_score():
    s = int(input("请输入学生成绩:"))
    assert 0 <= s <= 100, "成绩超出范围!"
    return s

try:
    score = get_score()
    print("学生的成绩为:", score)
except AssertionError as err:
    print(err)
    print("检查正在进行中...")

print("程序退出")
```

assert 语句的示例2

```
def get_age():
    a = input("请输入年龄")
    a = int(a)
    assert a < 120, "年龄太大了，可能吗?"
    assert a >= 0, "还没出生?"
    return a
try:
    age = get_age()
except AssertionError as ex:
    print("发生输入年龄不合法的错误!", ex)
    age = 0
except:
    print("发生了其它的错误!,暂不处理")
    age = 0
print("您输入的年龄是:", age)
```

为什么要用异常处理机制

- 在程序调用层数较深时，向主调函数传递错误信息需要层层return返回比较麻烦,所以用异常处理机制

```
def f1():
    print("开始盖房子打地基")
    return ValueError("挖掘出文物停工!!!")

def f2()
    print("开始盖房子地面以上的部分")
    return ZeroDevisionError("规划建设高压线停工!!!")

def f3():
    "第二承包商开始找人干活"
    f1()
    f2()

def build_house():
    f3()

build_house()
```

day15

迭代器 Iterator

什么是迭代器

- 迭代器是访问可迭代对象的工具
- 迭代器是指用 iter(obj) 函数返回的对象(实例)
- 迭代器可以用next(it)函数获取可迭代对象的数据

迭代器函数iter和next

函数	说明
iter(iterable)	从可迭代对象中返回一个迭代器,iterable必须是能提供一个迭代器的对象
next(iterator)	从迭代器iterator中获取下一个记录，如果无法获取一下条记录，则触发 StopIteration 异常

迭代器说明

- 迭代器只能往前取值,不会后退
- 用iter函数可以返回一个可迭代对象的迭代器

迭代器示例:

```
# 示例 可迭代对象
L = [1, 3, 5, 7]
it = iter(L)  # 从L对象中获取迭代器
next(it)  # 1  从迭代器中提取一个数据
```

```
next(it)  # 3
next(it)  # 5
next(it)  # 7
next(it)  # StopIteration 异常
# 示例2 生成器函数
It = iter(range(1, 10, 3))
next(It)  # 1
next(It)  # 4
next(It)  # 7
next(It)  # StopIteration
```

迭代器的用途

- 迭代器对象能用next函数获取下一个元素

迭代器函数iter和next 示例：

```
L = [2, 3, 5, 7]
it = iter(L)
# 访问列表中的所有元素
while True:
    try:
        print(next(it))
    except StopIteration:
        print("迭代器访问结束")
        break

L = [2, 3, 5, 7]
for x in L:
    print(x)
else:
    print("迭代器访问结束")
```

生成器 Generator (python 2.5及之后)

什么是生成器:

- 生成器是能够动态提供数据的可迭代对象
- 生成器是在程序运行时生成数据，与容器类不同，它通常不会在内存中保存大量的数据，而是现用现生成

两种生成器:

- 生成器函数
- 生成器表达式

生成器函数

- 含有yield语句的函数是生成器函数,此函数被调用将返回一个生成器对象
- yield翻译为 (产生或生成)

yield 语句

yield语法

```
yield 表达式
```

yield语法说明

- yield 用于 def 函数中，目的是将此函数作为生成器函数使用
- yield 用来生成数据，供迭代器的next(it)函数使用.

生成器函数示例1:

```
def myyield():
    yield 2
    yield 3
    yield 5
    yield 7
    print("生成结束!")

for x in myyield():  # 调用生成器函数，返回可迭代的生成器对象
    print(x)
```

生成器函数说明:

- 生成器函数的调用将返回一个生成器对象,生成器对象是一个可迭代对象
- 在生成器函数调用return会触发一个 StopIteration 异常（即生成数据结束）

生成器函数示例2:

```
# 此生成器函数可以生成从0开始的一系列整数，到n结束(不包含n)
def myinteger(n):
    i = 0
    while i < n:
        yield i
        i += 1

for x in myinteger(3):
```

```
    print(x)  # 打印 0, 1, 2

it = iter(myinteger(2))
print(next(it))  # 0
print(next(it))  # 1
print(next(it))  # StopIteration异常
```

生成器函数示例3:

```
def even(start, end):
    s = start
    while s < end:
        if s % 2 == 0:
            yield s
        s += 1
    return "生成器函数结束"

It = even(3, 10)
next(It)
next(It)
next(It)
next(It)
next(It)
next(It)
# for in 循环
for x in even(1, 10):
    print(x)
# 列表推导式
L = [x**2 for x in even(1, 20)]
```

生成器函数的执行过程

- 生成器函数是在next(it)对生成器返回的迭代器进行操作时才开始调用，当遇到yield语句时结束调用并返回数据给next(it)函数调用
- 生成器函数每次运行到yield语句则暂停执行，保存此函数当前的执行状态，下一次调用next(it)时则从当前的执行状态开始继续执行
- 当生成器函数返回时会触发StopIteration异常来通知next(it) 函数调用结束，同时释放此生成器函数执行时占有的资源

生成器函数的执行过程示例

```
def myyield():
    print("即将生成2")
```

```
    yield 2
    print("即将生成3")
    yield 3
    print("即将生成5")
    yield 5
    print("生成结束!")

it = iter(myyield())  # it绑定迭代器
next(it)
```

day15 pm

生成器表达式

生成器表达式语法:

```
( 表达式 for 变量 in 可迭代对象 [if 真值表达式] )
```

生成器表达式语法 说明:

```
if 子句可以省略
```

生成器表达式作用:

- 用推导式形式创建一个新的生成器

生成器表达式示例

```
gen = (x**2 for x in range(1, 5))
it = iter(gen)
next(it) # 1
next(it) # 4
next(it) # 9
next(it) # 16
next(it) # StopIteration
```

迭代工具函数

- 迭代工具函数的作用是生成一个个性化可迭代对象

函数	说明

函数	说明
zip(iter1 [,iter2 [...]])	返回一个zip生成器对象, 此对象用于生成一个元组，此元组的数据分别来自于参数中每个可迭代对象,生成元组的个数由最小的可迭代对象大小决定
enumerate(iterable, start=0)	返回一个enumerate生成器对象，此对象生成类型为(索引,值对)的元组,默认索引从零开始，也可以用start指定

zip函数

zip 示例

```
numbers = [10086, 10000, 10010, 95588]
names = ["中国移动", "中国电信", "中国联通"]
# z为可迭代对象,每个元素为元组(号码,名字)
for n, a in zip(numbers, names):
    print(a, "的客服号码:", n)

R = range(100)
names = ['zhangfei', 'zhaoyun', 'guanyu']
areas = ['北京', '上海', '深圳', '天津']
for x in zip(R, names, areas):
    print(x)
# (0, 'zhangfei', '北京')
# (1, 'zhaoyun', '上海')
# (2, 'guanyu', '深圳')
```

zip函数的实现原理示例:

```
    def myzip(*args):
        ...
```

enumerate 示例

```
names = ["中国移动","中国电信", '中国联通']
for no, name in enumerate(names, 1):
    print(no, ".", name)
for x in enumerate(names, 1):
    print(x)
```

枚举函数 enumerate

enumerate 函数格式

```
enumerate(iterable[, start])
```

enumerate 函数作用

- 生成一个枚举对象，新迭代器生成的数据会将原迭代器取出的数据与索引值形成元组(index, value)形式返回值

enumerate 函数示例

```
for i,v in enumerate("ABC", 1):
    print("第", i, "个字符是",  v)
```

enumerate函数的实现原理示例:

```
    def myenumerate(iterable, start=0):
        ...
```

迭代器围棋练习:

- 用字典表示一个围棋棋盘,字典的键用字母和数字组合来表示位置:
  - 如:A1 代表左上角的位置 A19代表右上角位置,S1代表左下角位置,S19代表右下角位置
- 字典的值用来表示围棋子的分布,0表示无棋子，1表示黑子，2表示白子.
  - 如:{"B3": 1 }表示第2行，第3列摆有一个黑子.
- 用推导式生成一个字典,
- 用字典推导式生成器一个二维的棋盘,棋盘上无棋子,
- 在第9行，第11列放一个白子。
- 在第11行，第9列放一个黑子。

迭代器围棋练习答案:

```
chessboard = {}
for x in range(19):
    for y in range(19):
        k = chr(ord('A')+y) + str(x+1)
        chessboard[k] = 0
# 其它做法:
chessboard={chr(ord('A')+y) + str(x+1): 0 for x in range(19) for y in
range(19) }
```

字节串和字节数组

字节串 bytes(也叫字节序列)

字节串的作用:

- 存储以字节为单位的数据
- 字节串是不可变的字节序列

字节串说明:

- 字节是0~255之间的整数,用来表示一个字节的取值

创建空字节串字面值

```
B = b''     # B为空字节串
B = b""     # B为空字节串
B = b''''''' # B为空字节串
B = b""""""" # B为空字节串
B = bytes() # B为空字节串
```

创建非空字节串字面值

```
B = bytes([0x41, 0x42, 0x43, 0x44])
B = b'ABCD'
B = b"ABCD"  # 类型为bytes
B = b'\x41\x42'
```

字节串的构造函数bytes

函数	说明
bytes()	# 生成一个空的字节串 等同于 b''
bytes(整型可迭代对象)	# 用可迭代对象初始化一个字节串
bytes(整数n)	生成n个值为0的字节串
bytes(字符串, encoding='utf-8')	用字符串的转换编码生成一个字节串

bytes的运算

```
+ += * *=
< <= > >= == !=
```

```
in /not in
索引和切片

len()
max(x)
min(x)
sum(x)
any(x)
all(x)
```

bytes与str的区别:

- bytes存储字节
- str存储unicode字符

bytes与str转换:

```
       编码(encode)
str   ------------> bytes
    b = s.encode(encoding='utf-8')
       解码(decode)
bytes ------------> str
    s = b.decode(encoding='utf-8')
```

字节数组 bytearray

- 可变的字节序列

字节数组的构造函数 bytearray

函数	说明
bytearray()	创建空的字节数组
bytearray(整数)	用可迭代对象初始化一个字节数组
bytearray(整型可迭代对象)	生成n个值为0的字节数组
bytearray(字符串, encoding='utf-8')	用字符串的转换编码生成一个字节数组

操作:

```
+ += * *=
比较运算: < <= > >= == !=
in / not in 运算符
```

```
索引 index / 切片 slice
(字节数组的索引和切片可以赋值操作, 规则同列表的索引和切片赋值)
```

bytearray 的方法:

方法	说明
B.clear()	清空
B.append(n)	追加一个字节(n为0-255的整数)
B.remove(value)	删除第一个出现的字节,如果没有出现，则产生ValueError错误
B.reverse()	字节的顺序进行反转

B.decode(encoding='utf-8') B.find(sub[, start[, end]])