# 一 工厂模式

初步的工厂：

*# 橘子味甜蛋糕***class** OrangeCake(object):  
 **def** taste(self):  
 print(**'橘子味甜蛋糕'**)

*# 苹果味甜蛋糕***class** AppleCake(object):  
 **def** taste(self):  
 print(**'苹果味甜蛋糕'**)  
  
*# 生产蛋糕的工厂***def** cakefac(type):  
 **if**(type == **'Apple'**):  
 **return** AppleCake()  
 **if**(type == **'Orange'**):  
 **return** OrangeCake

cake = cakefac(**'Apple'**)  
cake.taste()

最终的工厂模式：

*# 橘子味甜蛋糕***class** OrangeCake(object):  
 **def** taste(self):  
 print(**'橘子味甜蛋糕'**)  
  
*# 苹果味甜蛋糕***class** AppleCake(object):  
 **def** taste(self):  
 print(**'苹果味甜蛋糕'**)  
  
*# 基本的蛋糕店***class** CakeStore(object):  
 **def** createCake(self,type):  
 **pass  
 def** order(self,type):  
 self.cake = self.createCake(type)  
 **return** self.cake  
  
*# 子类：甜味蛋糕店***class** SweetCakeStore(CakeStore):  
 **def** createCake(self,type):  
 self.cakeFactory = CakeFactory();  
 **return** self.cakeFactory.createCake(type)  
  
*# 生产蛋糕的工厂***class** CakeFactory(object):  
 **def** createCake(self,type):  
 self.type = type  
 **if**(self.type == **'Apple'**):  
 self.cake = AppleCake()  
 **elif** self.type == **'Orange'**:  
 self.cake = OrangeCake();  
 **return** self.cake  
  
cakestore = SweetCakeStore()  
cake = cakestore.order(**'Apple'**)  
cake.taste()

定义了一个创建对象的接口(可以理解为函数)，但由子类决定要实例化的类是哪一个，工厂方法让类的实例化推迟到子类，抽象的CakeStore提供了一个创建对象的方法createCar，也叫作工厂方法。

子类真正实现这个createCake方法创建出具体产品。 创建者类不需要直到实际创建的产品是哪一个，选择了使用了哪个子类，自然也就决定了实际创建的产品是什么。

# 二 单例模式

## 1 \_\_new\_\_方法

\_\_new\_\_至少要有一个参数cls，代表要实例化的类，此参数在实例化时由Python解释器自动提供；

\_\_new\_\_必须要有返回值，返回实例化出来的实例，这点在自己实现\_\_new\_\_时要特别注意，可以return父类\_\_new\_\_出来的实例，或者直接是object的\_\_new\_\_出来的实例

\_\_init\_\_有一个参数self，就是这个\_\_new\_\_返回的实例，\_\_init\_\_在\_\_new\_\_的基础上可以完成一些其它初始化的动作，\_\_init\_\_不需要返回值；

## 2 单例模式

# 实例化一个单例class Singleton(object):

\_\_instance = None

\_\_first\_init = False

def \_\_new\_\_(cls, age, name):

if not cls.\_\_instance:

cls.\_\_instance = object.\_\_new\_\_(cls)

return cls.\_\_instance

def \_\_init\_\_(self, age, name):

if not self.\_\_first\_init:

self.age = age

self.name = name

Singleton.\_\_first\_init = True

a = Singleton(18, "dongGe")

b = Singleton(8, "dongGe")

print(id(a))

print(id(b))

print(a.age)

print(b.age)

a.age = 19

print(b.age)

# 三 异常

## 1 异常捕获

try:

print('-----test--1---')

open('123.txt','r')

print('-----test--2---')

except IOError:

pass

## 2 捕获多个异常

#coding=utf-8try:

print('-----test--1---')

open('123.txt','r') # 如果123.txt文件不存在，那么会产生 IOError 异常

print('-----test--2---')

print(num)# 如果num变量没有定义，那么会产生 NameError 异常

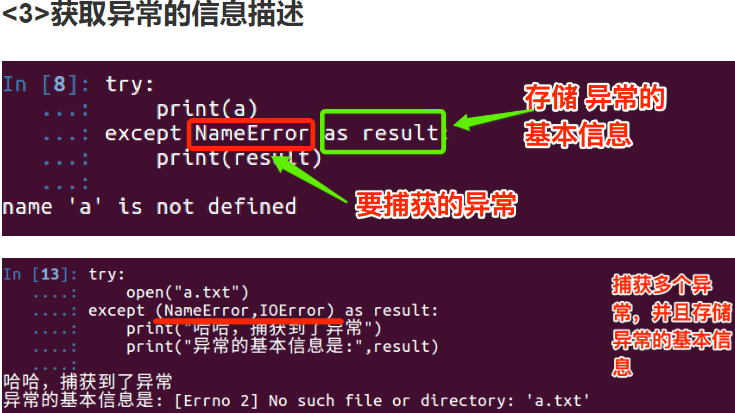
except (IOError,NameError):

#如果想通过一次except捕获到多个异常可以用一个元组的方式

# errorMsg里会保存捕获到的错误信息

print(errorMsg)

当捕获多个异常时，可以把要捕获的异常的名字，放到except 后，并使用元组的方式仅进行存储。



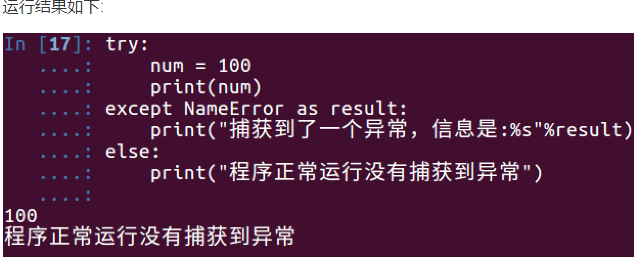
try:

num = 100

print numexcept NameError as errorMsg:

print('产生错误了:%s'%errorMsg)else:

print('没有捕获到异常，真高兴')



try...finally...语句用来表达这样的情况：

在程序中，如果一个段代码必须要执行，即无论异常是否产生都要执行，那么此时就需要使用finally。 比如文件关闭，释放锁，把数据库连接返还给连接池等

demo:

import time

try:

f = file('test.txt')

while True:

content = f.readline()

if len(content) == 0:

break

time.sleep(2)

print(content)

finally:

f.close()

print('关闭文件')

说明:

test.txt文件中每一行数据打印，但是我有意在每打印一行之前用time.sleep方法暂停2秒钟。这样做的原因是让程序运行得慢一些。在程序运行的时候，按Ctrl-c中断/取消程序。

我们可以观察到KeyboardInterrupt异常被触发，程序退出。但是在程序退出之前，finally从句仍然被执行，把文件关闭。

异常的传递：

def test1():

print("----test1-1----")

print(num)

print("----test1-2----")

def test2():

print("----test2-1----")

test1()

print("----test2-2----")

def test3():

try:

print("----test3-1----")

test1()

print("----test3-2----")

except Exception as result:

print("捕获到了异常，信息是:%s"%result)

print("----test3-2----")

test3()

print("------华丽的分割线-----")

test2()

如果一个异常是在一个函数中产生的，例如函数A---->函数B---->函数C,而异常是在函数C中产生的，那么如果函数C中没有对这个异常进行处理，那么这个异常会传递到函数B中，如果函数B有异常处理那么就会按照函数B的处理方式进行执行；如果函数B也没有异常处理，那么这个异常会继续传递，以此类推。。。如果所有的函数都没有处理，那么此时就会进行异常的默认处理，即通常见到的那样

注意观察上图中，当调用test3函数时，在test1函数内部产生了异常，此异常被传递到test3函数中完成了异常处理，而当异常处理完后，并没有返回到函数test1中进行执行，而是在函数test3中继续执行

# **抛出自定义的异常**

你可以用raise语句来引发一个异常。异常/错误对象必须有一个名字，且它们应是Error或Exception类的子类

下面是一个引发异常的例子:

class ShortInputException(Exception):

'''自定义的异常类'''

def \_\_init\_\_(self, length, atleast):

#super().\_\_init\_\_()

self.length = length

self.atleast = atleasttry:

s = input('请输入 --> ')

if len(s) < 3:

# raise引发一个你定义的异常

raise ShortInputException(len(s), 3)

except EOFError:

print("你输入了一个结束标记EOF")except ShortInputException as result:#x这个变量被绑定到了错误的实例

print('ShortInputException: 输入的长度是 %d,长度至少应是 %d'% (result.length, result.atleast))else:

print('没有异常发生.')

以上程序中，关于代码#super().\_\_init\_\_()的说明

这一行代码，可以调用也可以不调用，建议调用，因为\_\_init\_\_方法往往是用来对创建完的对象进行初始化工作，如果在子类中重写了父类的\_\_init\_\_方法，即意味着父类中的很多初始化工作没有做，这样就不保证程序的稳定了，所以在以后的开发中，如果重写了父类的\_\_init\_\_方法，最好是先调用父类的这个方法，然后再添加自己的功能

异常处理中抛出异常：

class Test(object):

def \_\_init\_\_(self, switch):

self.switch = switch #开关

def calc(self, a, b):

try:

return a/b

except Exception as result:

if self.switch:

print("捕获开启，已经捕获到了异常，信息如下:")

print(result)

else:

#重新抛出这个异常，此时就不会被这个异常处理给捕获到，从而触发默认的异常处理

raise

a = Test(True)

a.calc(11,0)

print("----------------------华丽的分割线----------------")

a.switch = False

a.calc(11,0)