Python面向对象

> 面向对象非常的重要。这个文档必不可能涵盖所有的内容。希望大家可以通过csdn等方式在课后继续学习面向对象。

> 面向对象的三大特性：封装、继承、多态

1 面向过程与面向对象

- 面向过程

我们之前编写程序就是面向过程编程。是为了实现一个目的而编写的代码。所有的代码仅仅为了完成当前这个目标。这是种自上而下的编程方式。首先通过目标来确定代码核心框架，然后再向框架中增添详细功能实现代码。这种编程方式对于较大体量的程序来说，既不方便编写，也不方便维护。为了定位实现某一步功能的代码，可能需要将代码完整的浏览一遍。

- 面向对象

我们将有着相似特性和行为的一类代码进行抽象封装，让它形成一个具有一定属性和功能的类来方便我们调用。通过构建一个个类并调用类来构成我们的程序。这是种自下而上的编程方式。首先要构建相关类，通过一个个组件逐步构建程序。

2 类与对象

2.1 什么是类

在现实生活中，我们会给东西分类。比如鼠标，尽管有不同的品牌，不同的形状等等，它们都因为有左右按键以及一个滚轮并可以操作系统中的光标而被称为鼠标。我们通过总结事物所具有的属性和功能，将相似的归为一类。

对于代码，也是这样。我们可以将具有相似属性和功能的代码块进行抽象总结，构建一个类。

比如登陆这个非常常见的操作。它出现在非常多的网站上，有着不同的登陆界面。我们可以对其进行总结抽象。

首先，有三个最基础的功能：输入账号、输入密码以及提交。有两个属性：账号与密码。

通过上面的抽象总结，我们可以把登陆这个模块设计成一个类。

像这样把相似属性和功能代码放置在一个类中的操作称为封装。

```python
class Login():
    # 两个属性
    username = None
    password = None

    # 三个方法
    def getUsername(self):
        pass

    def getPassword(self):
        pass

    def submit(self):
        pass
```

2.2 什么是对象

> 学好了这里七夕的时候大家就可以有好多对象陪着了ヾ(≧▽≦)ゝ

当将一个类被实例化的时候，就可以获得一个对象。

好比我们讲鼠标，这只是一个抽象的概念，它指那一类物品，并不具体到哪一个。

当你买的罗技G502被生产出来的时候，就是鼠标这个类被实例化了。你的G502就是一个对象，它是鼠标类生成的一个对象。

我们使用前面的Login类来进行讲解

```python
# 实例化Login类
a = Login() # 这时候a就是一个对象，是一个Login对象
```

类是无法被直接调用的，我们可以通过实例化类获得这个类的对象来调用其中的方法。

2.3 类的基本结构

2.3.1 属性与方法

类内部由属性和方法两部分组成。

- 属性：对象的数据描述，为类的成员变量。
- 方法：对象的方法

使用class来声明一个类。

```
```

```python
class Login():

    # 可以直接在这里声明（创建）属性，在没有初始值的情况下推荐优先初始化为None。
    # 在后面学了构造方法之后更推荐在构造方法中初始化相关属性。
    username = None
    password = None

    # 在类中声明的函数即为类的方法
    def  getUsername(self):
        pass

    def getPassword(self):
        pass

    def submit(self):
        pass
```

> self的事情等等再讲。

2.3.2 self变量

self变量在类中处处存在，也是类内数据交换的桥梁。

self指向这个方法的调用对象。谁调用了这个方法，在这个方法内self就指向谁。 self由调用者自动传入，不需要我们传入。

```python
class Login():

    def getUsername(self):
        print(self)

a = Login()
b = Login()
a.getUsername() # 不需要传入self，self由调用者a传入
print(a)
b.getUsername()
print(b)
```

运行结果

```
<__main__.Login object at 0x000001FD1F059FD0>
<__main__.Login object at 0x000001FD1F059FD0>
<__main__.Login object at 0x000001FD1F059FA0>
<__main__.Login object at 0x000001FD1F059FA0>
```

可以看到，a指向的对象和a调用getUsername方法后getUsername中self指向的对象是一致的。

有了self我们就可以在类内部调用对象的属性以及方法了。

```
class Login():
    username = None
    password = None

    def getUsername(self):
        print(self.username)
        print("get username ...")
        self.getUsername()

a = Login()
print(a.username)
a.getUsername()
```

2.3.3 构造方法

通过实例化类可以获得类的对象。而这个实例化过程由类的构造方法提供。

```
class Login():

    # 名为__init__的函数即为Python中类构造方法。注意：这里的__是两个下划线_
    def __init__(self):
        pass
```

但是在之前的例子中我们并没有编写过任何构造方法，类是如何实例化的?

如果一个类没有提供任何构造方法，那么在这个类被加载的时候，Python解释器会为这个类自动添加一个空参构造器（即没有任何参数的构造器）（就是上面示例中写的那个样子）。

有了构造方法就可以在实例化类的时候为对象传入一定参数了。

比如我们想让前面那个Login类在实例化的时候就获得一个默认用户名，就可以使用带参构造器来实现。

```
class Login():

    def __init__(self, username):
        self.username = username
        self.password = None


a = Login('Python')
print(a.username)
```

2.3.4 访问权限

> Python极大弱化了访问权限这个属性，推荐大家在课后去了解Java中的访问权限修饰符。

你手里有两本书，如果别人想看，你不是很介意。你手里有2000块钱，别人想用，你恨不得直接把他按在地上。

你是这么想的，你的对象也是这么想的。那该如何实现访问控制呢。

Python并没有像Java或C++那样提供访问控制符。我们只需要在在变量名/方法名前加上__就无法从外部访问了，这个变量/方法将会转为私有变量/私有方法。

```
class Login() :

    def __init__(self):
        self.__username = "Python"
        self.__password = None

    def getUsername(self) :
        print(self.__username)

a = Login()
print(a.__username)
```

这时候解释器会直接抛出AttributeError，告诉你对象中这个变量不存在（实际上是存在的，只是不允许你在这个对象内部以外的地方调用）

```
b = Login()
b.getUsername()
print(b.__username)
```

通过运行结果我们可以看到，__username这个私有变量只能在对象内部调用，无法从外部访问。

通过创建私有变量/私有方法的方式可以有效防止数据泄露。

虽然我们无法直接访问私有变量，但是我们可以提供公共方法来访问私有变量。比如经常使用的一种逻辑：

```
class User():

    def __init__(self, username, password):
        self.__username = username
        self.__password = password

    def getUsername(self):
        return self.__username

    def setUsername(self, username):
        self.__username = username

```

通过get...和set...来完成私有变量的访问。

> 但是！但是！！但是！！！
>
> 实质上Python是没有私有变量的，Python中所谓的私有变量，只是将其重命名为："_类名__变量名"
>
> 比如前文中的__username，无法直接通过实例化对象访问__username，但是可以访问_User__username，这二者在内存数据指向上是等价的。

3 继承与多态

> 继承与多态大大扩展了类的能力，方便了代码复用以及扩展。可以说是类的精髓所在。

> 继承与多态比较难，希望大家能多尝试，多问，一定要学会。

3.1 继承

顾名思义，从父辈那里获得他们已有的东西。

在类这个层面上，我们可以为类指定父亲，让他去继承指定类所拥有的属性和方法。

比如这里有个类

```
class Person():

    def __init__(self, name, gender, age):
        self.__name = name
        self.__gender = gender
        self.__age = age

    def getName(self):
        return self.__name

    def getGender(self):
        return self.__gender

    def getAge(self):
        return self.__age
```

众所周知，用户一般是人，所以我们可以通过扩展Person这个类来简化User类的实现，即使用User类继承Person`类

```
class User(Person):

    def __init__(self, id, name, gender, age):
        super(User, self).__init__(name, gender, age)
        self.__id = id

    def getId(self):
        return self.__id

# 上面的代码等价于下面这种写法，哪种更好一目了然
class User(Person):

    def __init__(self, id, name, gender, age):
        self.__name = name
        self.__gender = gender
        self.__age = age
        self.__id = id
```

```
18
19      def getName(self):
20          return self.__name
21
22      def getGender(self):
23          return self.__gender
24
25      def getAge(self):
26          return self.__age
27
28      def getId(self):
29          return self.__id
```

> super()类，用于调用父类内部的方法。
>
> 比如super(User, self)，可以指向User类的父类。在上面继承过程中，使用了super(User, self).__init__(name, gender, age)，即调用父类的构造方法。
>
> 之所以这么做，是因为在继承过程中，子类被实例化的时候不会主动调用父类中的有参构造方法。
>
> 当然，直接使用Person.__init__(name, gender, age)来调用父类构造器也是可以的。

Python支持多继承，即同时继承多个类。

```
1  class sample(clazz1, clazz2):
2      pass
```

3.2 多态

多态有两种出现形式：重载与重写

3.2.1 重载

有的时候，同一个功能方法我们可能根据需求不同传入不同的参数，其内部处理逻辑有一定的不同。如果为每一个都创建一个独立的方法名，可能调用会非常的不方便。为了方便调用，出现了重载这个概念。

重载，指在同一个类中，创建数个相同的方法名，但是参数列表不完全相同方法。

解释器会根据你传入的参数列表长度来自动确定你所调用的方法。

```
1  class UserFactory() :
2
```

```
    def __init__(self) :
        pass

    def getUser(self):
        id = None
        name = None
        gender = None
        age = None
        return User(id, name, gender, age)

    # 重载
    @overload
    def getUser(self, id):
        id = id
        name = None
        gender = None
        age = None
        return User(id, name, gender, age)
```

> Python作为弱类型语言简化了重载这个概念，不然重载能多写一页纸。

3.2.2 重写

有的时候我们并不满意父类所提供的某个方法，可以通过重写（overwrite）来重新编写这个方法。

```
class User(Pserson):

    def __init__(self, id, name, gender, age):
        super(User, self).__init__(name, gender, age)
        self.__id = id

    @overwrite
    def getName(self):
        print("try to get name ....")
        print("success")
        return self.__name

    def getId(self):
        return self.__id
```

这时候我们再使用User对象调用getName方法，就不止会返回用户姓名了。

> 虽然重写了但是仍然可以使用super()来调用父类中的方法。

4 特殊方法

Python的基类提供数个需要重写才可调用的特殊方法。

- 实例创建方法__new__

当创建一个对象时，先调用__new__方法，由__new__方法调用__init__方法。

__new__默认返回类对象，当返回值为空或返回值不为当前类的对象时，均不会调用__init__进行初始化

```
class test():
    def __new__(cls): # __new__带有一个默认参数cls，一般情况下不使用
        print("进入__new__")
        return super().__new__(cls)

    def __init(self):
        print("进入__init__")
```

- 构造方法__init__

在创建对象时自动调用，来初始化对象。

- 直接调用方法__call__

可以将对象当作函数直接调用

```
class test():
    def __call__(self):
        print("直接调用")

a = test()
a()
```

- 析构方法__del__

当对象生命周期结束时，自动调用析构方法。

- 对象字符串方法__str__和__repr__

当直接输出对象时，会直接输出__str__方法或__repr__方法返回的字符串。

> 如果不重写这两个方法，在直接输出对象时会返回对象的类别和内存地址

一般认为__str__返回内容面向用户。而__repr__方法返回的内容面向开发者，主要为调试信息。

一般要求至少重写__repr__方法

当两个方法在一个类中均被重写时，直接输出对象时会调用__str__。

```
class NoOverWrite():
    def __init__(self):
        pass

class OverWrite():
    def __init__(self):
        pass

    def __repr__(self):
        return "__repr__"

    def __str__(self):
        return "__str__"

a = NoOverWrite()
b = OverWrite()
print(a)
print(b)
```

- __len__

当对该对象使用len()函数时的操作

```
class test():
    def __len__(self):
        return 4

a = test()
len(a)
```

- 除上述特殊方法之外，还有：

魔术方法	含义
基本魔术方法	
__new__(cls[, ...])	1. __new__ 是在一个对象实例化的时候所调用的第一个方法 2. 它的第一个参数是这个类，其他的参数是用来直接传递给 __init__ 方法 3. __new__ 决定是否要使用该 __init__ 方法，因为 __new__ 可以调用其他类的构造方法或者直接返回别的实例对象来作为本类的实例，如果 __new__ 没有返回实例对象，则 __init__ 不会被调用 4. __new__ 主要是用于继承一个不可变的类型比如一个 tuple 或者 string
__init__(self[, ...])	构造器，当一个实例被创建的时候调用的初始化方法
__del__(self)	析构器，当一个实例被销毁的时候调用的方法
__call__(self[, args...])	允许一个类的实例像函数一样被调用：x(a, b) 调用 x.__call__(a, b)
__len__(self)	定义当被 len() 调用时的行为
__repr__(self)	定义当被 repr() 调用时的行为
__str__(self)	定义当被 str() 调用时的行为
__bytes__(self)	定义当被 bytes() 调用时的行为
__hash__(self)	定义当被 hash() 调用时的行为
__bool__(self)	定义当被 bool() 调用时的行为，应该返回 True 或 False
__format__(self, format_spec)	定义当被 format() 调用时的行为
有关属性	
__getattr__(self, name)	定义当用户试图获取一个不存在的属性时的行为
__getattribute__(self, name)	定义当该类的属性被访问时的行为
__setattr__(self, name, value)	定义当一个属性被设置时的行为
__delattr__(self, name)	定义当一个属性被删除时的行为
__dir__(self)	定义当 dir() 被调用时的行为
__get__(self, instance, owner)	定义当描述符的值被取得时的行为
__set__(self, instance, value)	定义当描述符的值被改变时的行为
__delete__(self, instance)	定义当描述符的值被删除时的行为
比较操作符	
__lt__(self, other)	定义小于号的行为：x < y 调用 x.__lt__(y)
__le__(self, other)	定义小于等于号的行为：x <= y 调用 x.__le__(y)
__eq__(self, other)	定义等于号的行为：x == y 调用 x.__eq__(y)
__ne__(self, other)	定义不等号的行为：x != y 调用 x.__ne__(y)
__gt__(self, other)	定义大于号的行为：x > y 调用 x.__gt__(y)
__ge__(self, other)	定义大于等于号的行为：x >= y 调用 x.__ge__(y)
算数运算符	
__add__(self, other)	定义加法的行为：+
__sub__(self, other)	定义减法的行为：-
__mul__(self, other)	定义乘法的行为：*
__truediv__(self, other)	定义真除法的行为：/
__floordiv__(self, other)	定义整数除法的行为：//
__mod__(self, other)	定义取模算法的行为：%
__divmod__(self, other)	定义当被 divmod() 调用时的行为
__pow__(self, other[, modulo])	定义当被 power() 调用或 ** 运算时的行为
__lshift__(self, other)	定义按位左移位的行为：<<
__rshift__(self, other)	定义按位右移位的行为：>>
__and__(self, other)	定义按位与操作的行为：&
__xor__(self, other)	定义按位异或操作的行为：^
__or__(self, other)	定义按位或操作的行为：|
反运算	
__radd__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rsub__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rmul__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rtruediv__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rfloordiv__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rmod__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rdivmod__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rpow__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rlshift__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rrshift__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__rxor__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
__ror__(self, other)	（与上方相同，当左操作数不支持相应的操作时被调用）
增量赋值运算	
__iadd__(self, other)	定义赋值加法的行为：+=
__isub__(self, other)	定义赋值减法的行为：-=
__imul__(self, other)	定义赋值乘法的行为：*=
__itruediv__(self, other)	定义赋值真除法的行为：/=
__ifloordiv__(self, other)	定义赋值整数除法的行为：//=
__imod__(self, other)	定义赋值取模算法的行为：%=

__ipow__(self, other[, modulo])	定义赋值幂运算的行为：**=
__ilshift__(self, other)	定义赋值按位左移位的行为：<<=
__irshift__(self, other)	定义赋值按位右移位的行为：>>=
__iand__(self, other)	定义赋值按位与操作的行为：&=
__ixor__(self, other)	定义赋值按位异或操作的行为：^=
__ior__(self, other)	定义赋值按位或操作的行为：|=
一元操作符	
__neg__(self)	定义正号的行为：+x
__pos__(self)	定义负号的行为：-x
__abs__(self)	定义当被 abs() 调用时的行为
__invert__(self)	定义按位求反的行为：~x
类型转换	
__complex__(self)	定义当被 complex() 调用时的行为（需要返回恰当的值）
__int__(self)	定义当被 int() 调用时的行为（需要返回恰当的值）
__float__(self)	定义当被 float() 调用时的行为（需要返回恰当的值）
__round__(self[, n])	定义当被 round() 调用时的行为（需要返回恰当的值）
__index__(self)	1. 当对象是被应用在切片表达式中时，实现整形强制转换 2. 如果你定义了一个可能在切片时用到的定制的数值型,你应该定义 __index__ 3. 如果 __index__ 被定义，则 __int__ 也需要被定义，且返回相同的值

5 异常与异常捕获

5.1 异常类

之前我们讲到如果程序执行出现问题就可能会抛出相应的错误，比如用只读模式打开一个不存在的文件就会抛出 FileNotFoundError。 FileNotFoundError 就是一个异常，它是一个类。除此之外还有诸如 ImportError 等等数十种异常类。它们的基类是 BaseException。类间关系如下。

```
BaseException           所有异常的基类
 +-- SystemExit           解释器请求退出
 +-- KeyboardInterrupt         用户中断执行(通常是输入^C)
 +-- GeneratorExit          生成器(generator)发生异常来通知退出
 +-- Exception            常规错误的基类
      +-- StopIteration            迭代器没有更多值
      +-- StopAsyncIteration            必须通过异步迭代器对象的__anext__()方法引发以停止迭代
      +-- ArithmeticError             所有数值计算错误的基类
           +-- FloatingPointError           浮点计算错误
           +-- OverflowError              数值运算超出最大限制
           +-- ZeroDivisionError           除(或取模)零 (所有数据类型
      +-- AssertionError              断言语句失败
      +-- AttributeError              对象没有这个属性
      +-- BufferError               与缓冲区相关的操作时引发
      +-- EOFError                没有内建输入,到达EOF 标记
      +-- ImportError               导入失败
           +-- ModuleNotFoundError       找不到模块
```

```
    +-- LookupError           无效数据查询的基类
         +-- IndexError           序列中没有此索引(index)
         +-- KeyError             映射中没有这个键
    +-- MemoryError           内存溢出错误
    +-- NameError             未声明、初始化对象
         +-- UnboundLocalError       访问未初始化的本地变量
    +-- OSError               操作系统错误，
         +-- BlockingIOError         操作将阻塞对象设置为非阻塞操作
         +-- ChildProcessError       子进程上的操作失败
         +-- ConnectionError         与连接相关的异常的基类
              +-- BrokenPipeError         在已关闭写入的套接字上写入
              +-- ConnectionAbortedError  连接尝试被对等方中止
              +-- ConnectionRefusedError  连接尝试被对等方拒绝
              +-- ConnectionResetError    连接由对等方重置
         +-- FileExistsError         创建已存在的文件或目录
         +-- FileNotFoundError       请求不存在的文件或目录
         +-- InterruptedError        系统调用被输入信号中断
         +-- IsADirectoryError       在目录上请求文件操作
         +-- NotADirectoryError      在不是目录的事物上请求目录操作
         +-- PermissionError         在没有访问权限的情况下运行操作
         +-- ProcessLookupError      进程不存在
         +-- TimeoutError            系统函数在系统级别超时
    +-- ReferenceError        弱引用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
         +-- RecursionError       解释器检测到超出最大递归深度
    +-- SyntaxError           Python 语法错误
         +-- IndentationError     缩进错误
              +-- TabError        Tab 和空格混用
    +-- SystemError           一般的解释器系统错误
    +-- TypeError             对类型无效的操作
    +-- ValueError            传入无效的参数
         +-- 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
    +-- Warning               警告的基类
         +-- DeprecationWarning      关于被弃用的特征的警告
         +-- PendingDeprecationWarning  关于构造将来语义会有改变的警告
         +-- RuntimeWarning          可疑的运行行为的警告
         +-- SyntaxWarning           可疑的语法的警告
         +-- UserWarning             用户代码生成的警告
         +-- FutureWarning           有关已弃用功能的警告的基类
```

```
61          +-- ImportWarning          模块导入时可能出错的警告的基类
62          +-- UnicodeWarning         与Unicode相关的警告的基类
63          +-- BytesWarning           bytes和bytearray相关的警告的基类
64          +-- ResourceWarning  与资源使用相关的警告的基类。。
```

可以看到绝大部分异常都是继承自 Eexcption 类。在 Exception 下衍生出的子类的子类也有着十分明显的逻辑关系。通过在执行程序过程中抛出对应的异常类对象来快速了解程序运行状况与问题所在。异常的抛出可以帮助我们迅速定位程序的问题。

5.2 异常的捕获

5.2.1 try .. except ..

很多时候我们并不希望异常的抛出终止我们程序的执行，毕竟无法100%避免运行中异常的出现。Python提供了异常捕获的办法。

```
try:
    pass
except:
    pass
```

通过使用 try .. except .. 可以实现异常的捕获与处理。

解释器会自动捕获 try 语句块中的抛出的异常，并执行 except 语句块中的内容。

不进行异常捕获：

```
print("程序开始")
print(5 / 0)
print("程序结束")
```

进行异常捕获：

```
try:
    print("进入try")
    print(5 / 0)
    print("退出try")
except:
    print("进入except")
print("程序结束")
```

通过运行上面的示例，可以看到在不捕获异常时执行到第2行 print(5/0) 的时候解释器会抛出 ZeroDivisionError 异常，程序异常退出，第3行并没有被执行。而捕获异常时，第2、6、7行的内容被成功输出，而第4行的 print() 并没有执行。

try语句块中如果没有出现异常，则跳过except语句块。如果出现异常，则异常语句之后的代码不再执行，直接跳转至except代码块。但这时候try代码块中抛出的异常被except捕获，不会引起解释器异常退出，程序其余部分正常执行。

可以在except后添加finally代码块，来完成部分必要操作，比如资源的关闭，连接的关闭。

> 不论是否存在异常， finally的代码都会在try .. except ..后执行。

也可以在except后添加else代码块，来完成当异常不存在时可以进行的操作。

```
try:
    # 检测是否存在异常
except:
    # 捕获异常时执行
else:
    # 未捕获到异常时执行
finally:
    # 不论是否捕获到异常均执行
```

5.2.2 捕获特定异常

有的时候针对不同的异常我们需要进行不同的处理，比如打开一个文件，如果文件不存在，需要创建文件；如果文件被占用，需要进入等待并且轮询。它们抛出的异常是不同的，需要分别进行捕获。

只需要在except后添加需要捕获的异常类类名即可

```
try:
    5 / 0
except ZeroDivisionError:
    print("除数为0")
print("程序结束")
```

如果抛出的异常和要捕获的异常不同，会继续向外层抛出（如果有多层捕获）直至被捕获处理，如果被解释器捕获则会输出异常并停止程序执行。比如下面这个。

```
try:
    try:
        5 / 0
    except FileNotFoundError:
        print("文件未找到")
except ZeroDivisionError:
    print("除数为0")
print("程序结束")
```

针对可能存在的不同异常，可以使用多个except语句。

```
try:
    open("1.txt", "r")
    5 / 0
except FileNotFoundError:
    print("文件未找到")
except ZeroDivisionError:
    print("除数为0")
```

这时候被抛出的异常会按照except的顺序依次查看是否会被捕获，它会被第一个可以捕获自己的except捕获。

通过捕获父类异常可以捕获子类异常，但是子类异常无法捕获父类异常。

```
# 父类在前
try:
    5 / 0
except Exception:
    print("出现异常")
except ZeroDivisionError:
    print("除数为0")

# 子类在前
try:
    5 / 0
except ZeroDivisionError:
    print("除数为0")
except Exception:
    print("出现异常")
```

因此通常在进行异常捕获的时候，优先捕获子类，最后使用父类进行保底。

5.3 异常的抛出

可以使用raise来手动抛出指定异常。

```
a = 4
if a > 3:
    raise Exception
else:
    pass
```

5.4 自定义异常

既然大部分异常类都是通过继承Exception类来实现的，通过相同的方法我们可以创建自己的异常类。

```
class MyError(Exception):
    def __init__(self):
        super().__init__()

    def __str__(self):
        return "自定义异常"
```

> 当然也可以继承任何Exception的子类来创建自定义异常类。

6 填坑

在前面个基础知识的时候我挖了不少坑。我来兑现承诺填坑了。