模块和包

什么是模块?

一个函数用来封装一个功能，但是，一个软件的设计不可能都只写在一个文件里面，将一些常用功能放到一个文件中，就是模块。模块的本质就是一个`.py`文件。

为什么要使用模块?

- 将程序分成一个个的功能文件，更加方便管理以及重复使用
- 提升开发效率，避免重复造轮子

模块的分类

- 内置模块：比如之前学习过的time、random模块
- 第三方模块：比如pygame模块，需要自己使用pip安装`pip install pygame`
- 自定义模块，比如起个名字叫做`ayst.py`

导入模块(import 模块名)

> 每个模块都是一个独立的命名空间，定义在这个模块中的函数，把这个模块的命名空间当做全局命名空间，两个不同模块中的同名变量不会发生冲突
>
> 模块第一次被导入后就将模块加载到内存中，重复导入模块会直接引用内存中已经加载好的结果

模块导入方式

- import 模块名
- import 模块名 as 新名称
- from ..... import .....

模块文件中的测试代码

> 一个模块中的功能也是需要进行测试的，但是，模块被导入时，就会自动执行模块中的print语句

- 脚本：直接作为文件执行`__name__ 等于 __main__`
- 模块：被导入执行`__name__ 等于 模块名`

```
class People:

    def __init__(self, name, gender, age):

        self.name = name
        self.gender = gender
        self.age = age

```

```python
a = 1
b = "hello"


print("哈哈哈，我导入就被执行了")

# 内置全局变量 __name__
print(__name__)

if __name__ == "__main__":

    print("只有以脚本形式运行我才会执行")
```

内置模块

sys模块

> 与解释器交互的接口

- sys.argv 传递命令行参数
- sys.version 获取python解释器版本信息
- sys.platform 返回操作系统平台名称
- sys.modules Python解释器启动时加载到内存中的模块
- sys.path 返回模块的搜索路径

```python
import sys

print(sys.version)
print(sys.platform)
print(sys.path)
print(sys.modules)

print(sys.argv)

print(sum([int(i) for i in sys.argv[1:]]))
```

> 执行代码：python3 demo.py 1 2

模块的搜索路径

- 内存中已经加载的模块 `sys.modules`
- 内置模块
- 依次查找sys.path列表中的模块

> 注：模块名不可以与系统内置模块重名

我们自定义的模块如果全部放在site-packages文件夹下面，时间久了会非常混乱，因此可以创建一个文件夹专门存储模块

```python
import sys

sys.path.append('D:\cat')

import hello

print(hello.info)
```

包

什么是包?

- 包就是一个包含`__init__.py`的文件夹，创建包的目的就是为了将文件组织起来
- Python3中，包下面没有`__init__.py`，导入包不会报错，而Python2中会报错
- 创建包就是为了导入，而不是直接运行

导入规则

- 凡是导入时带点的，点的左边一定是包的名字
- import cat
- import cat.examples.test
- import cat.examples.test as t
- from cat.db import register
- from cat.api import *

注：`__all__ = ["manage", "versions"]`可以限定导入的模块

文件中代码：

```
cat/__init__.py:

from . import api
print("我是包下面的init文件")
```

api:

```
api/__init__.py:

from . import run

print("我是api下面的init文件")

__all__ = ["manage", "versions"]
```

```
api/manage.py:

def f1():

    print("I am api.manage")
```

```
api/run.py:

def f1():

    print("I am api.run")
```

```
api/versions.py:

def f1():

    print("I am api.versions")
```

db:

```
db/__init__.py:

print("我是db下面的init文件")
```

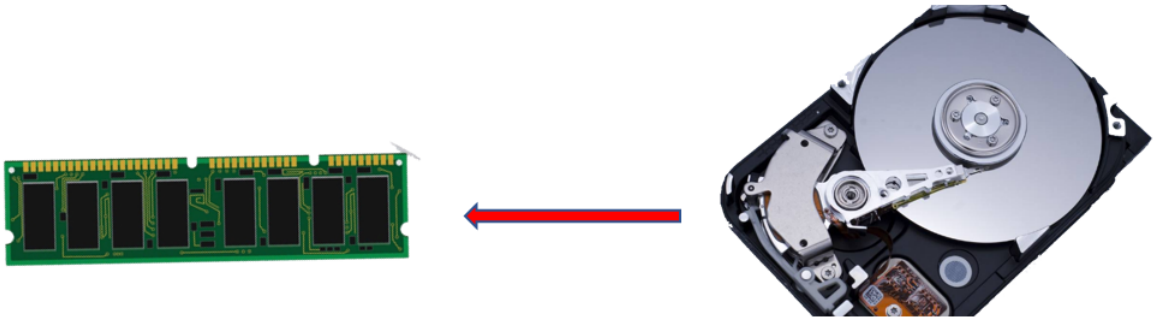
```
db/register.py:

def f1():
    print("I am db")
```

文件操作

文件的概念和作用

- 计算机中的文件，就是存储在某种长期储存设备上的一段数据
- 长期存储设备包括：硬盘、U盘、移动硬盘、光盘.....
- 使用文件时，CPU将保存在硬盘中的文件加载到内存当中



文件的存储方式

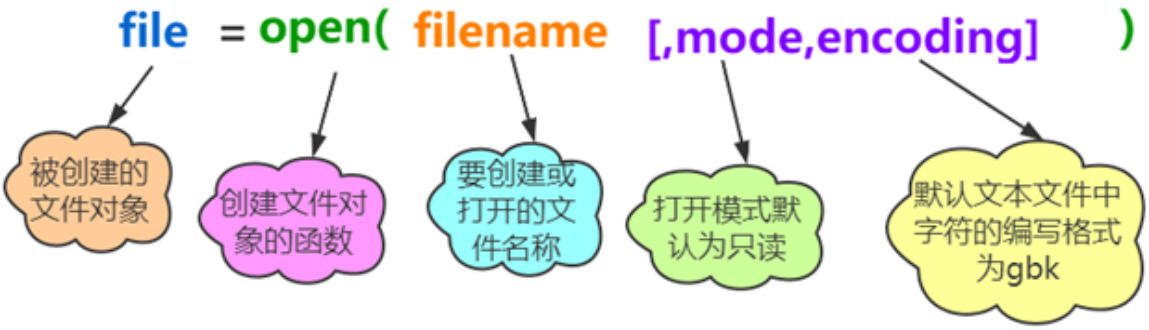
- 计算机中，文件以二进制的方式保存在磁盘上
- 文本文件和二进制文件
    - 文本文件，可以直接使用文本编辑软件打开
    - 文本文件，本质也是二进制文件，比如txt、py、cpp
    - 二进制文件无法用文本编辑软件打开，比如图片、音频、视频

文件读写

- open函数负责打开文件，并返回文件对象
- read方法可以一次性读入并返回文件的所有内容，并将文件指针移动到文件的末尾
- close方法负责关闭文件，如果忘记关闭文件，会造成系统资源消耗，影响后续对文件的访问

文件指针：

- 文件指针标记从哪个位置开始读取数据
- 第一次打开文件，文件指针指向文件的开始位置
- 当执行read方法后，文件指针移动到读取内容的末尾
- 频繁的移动文件指针，会影响文件的读写效率，开发中更多的时候会以只读、只写的方式来操作文件



按行读取文件

readline

- readline方法可以一次读取一行内容
- readline方法可以一次读取一行内容减少内存占用
- 方法执行后，会把 文件指针 移动到下一行，准备再次读取

```
f = open("hello.txt", mode="r", encoding="utf-8")
eof = False
while not eof:
    line = f.readline()
    if line:
        if line != "\n":
            print(line.strip())
    else:
        print("\n文件结束")
        eof = True
f.close()
```

文件复制

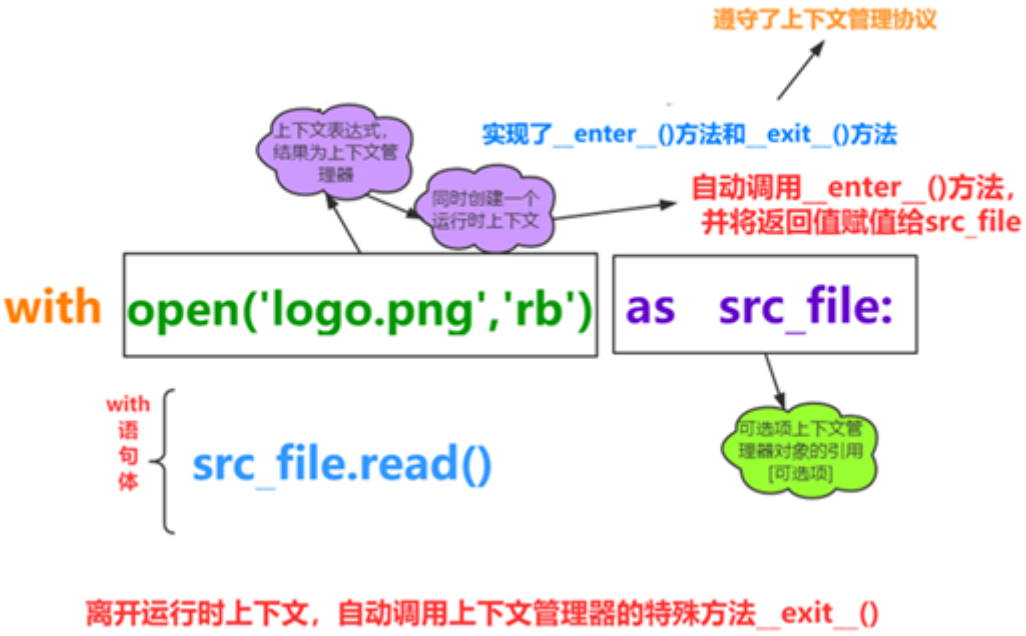
```
f_read = open("hello.txt", mode='r', encoding='utf-8')
f_write = open("hello[副本].txt", mode='w', encoding='utf-8')
text = f_read.read()
f_write.write(text)
f_read.close()
f_write.close()
```

大文件复制

```
f_read = open("hello.txt", mode='r', encoding='utf-8')
f_write = open("hello[副本].txt", mode='w', encoding='utf-8')
while True:
    text = f_read.readline()
    if not text:
        break
    f_write.write(text)
f_read.close()
f_write.close()
```

with语句(上下文管理器)

> with语句可以自动管理上下文资源，不论什么原因跳出with块，都能确保文件正确的关闭，以此来达到释放资源的目的



异常处理

Bug的由来

- 世界上第一部万用计算机的进化版-马克2号(Mark II)

- Debug

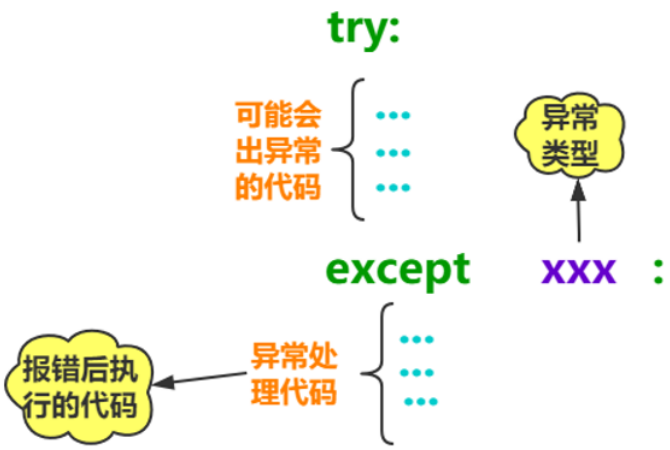
什么是异常?

- 程序运行时，如果Python解释器遇到错误，就会停止程序，触发异常
- 程序员编写特定代码，专门捕获这个异常，进入另一个处理分支，让程序不会崩溃，这就是异常处理
- 保证程序的稳定新和健壮性

常见异常类型

序号	异常类型	描述
1	ZeroDivisionError	除(或取模)零 (所有数据类型)
2	IndexError	序列中没有此索引(index)
3	KeyError	映射中没有这个键
4	NameError	未声明/初始化对象 (没有属性)
5	SyntaxError	Python 语法错误
6	ValueError	传入无效的参数

基本语法




```
try:
    尝试执行的代码
except:
    出现错误的处理
```

```
try:
    num = int(input("请输入数字："))
    print(num)
except:
    print("请输入正确的数值类型")
```

错误类型捕获

程序运行过程中，遇到的异常类型很可能是不同的，需要针对不同类型的异常，做不同的响应

```
try:
    pass

except 错误类型1:
    pass

except 错误类型2:
    pass

except Exception as e:
    print("未知错误 %s" % e)

else:
    print("没有异常才会执行的代码")

finally:
    print("不论是否异常都会执行的代码")
```

```
fruits = ["apple", "banana", "pear", "orange"]

```

```
try:
    print(hi)

except TypeError:
    print("类型错误")

except IndexError:
    print("下标索引错误")

except Exception as e:
    print("未知错误 %s" % e)

else:
    print("没有异常才会执行的代码")

finally:
    print("不论是否异常都会执行的代码")
```

抛出raise异常

在开发过程中，除了代码执行错误Python解释器会抛出异常之外，还可以根据业务需求主动抛出异常。

```
def check_passwd():

    passwd = input("请输入你的密码: ")

    if len(passwd) >= 8:

        return passwd

    raise Exception("密码长度至少8位")

try:
    passwd = check_passwd()
    print(passwd)

except Exception as e:
    print("错误类型为: ", e)
```

使用traceback模块打印异常信息

```
import traceback

try:
    print(10 / 0)
except:
    traceback.print_exc()
```

总结