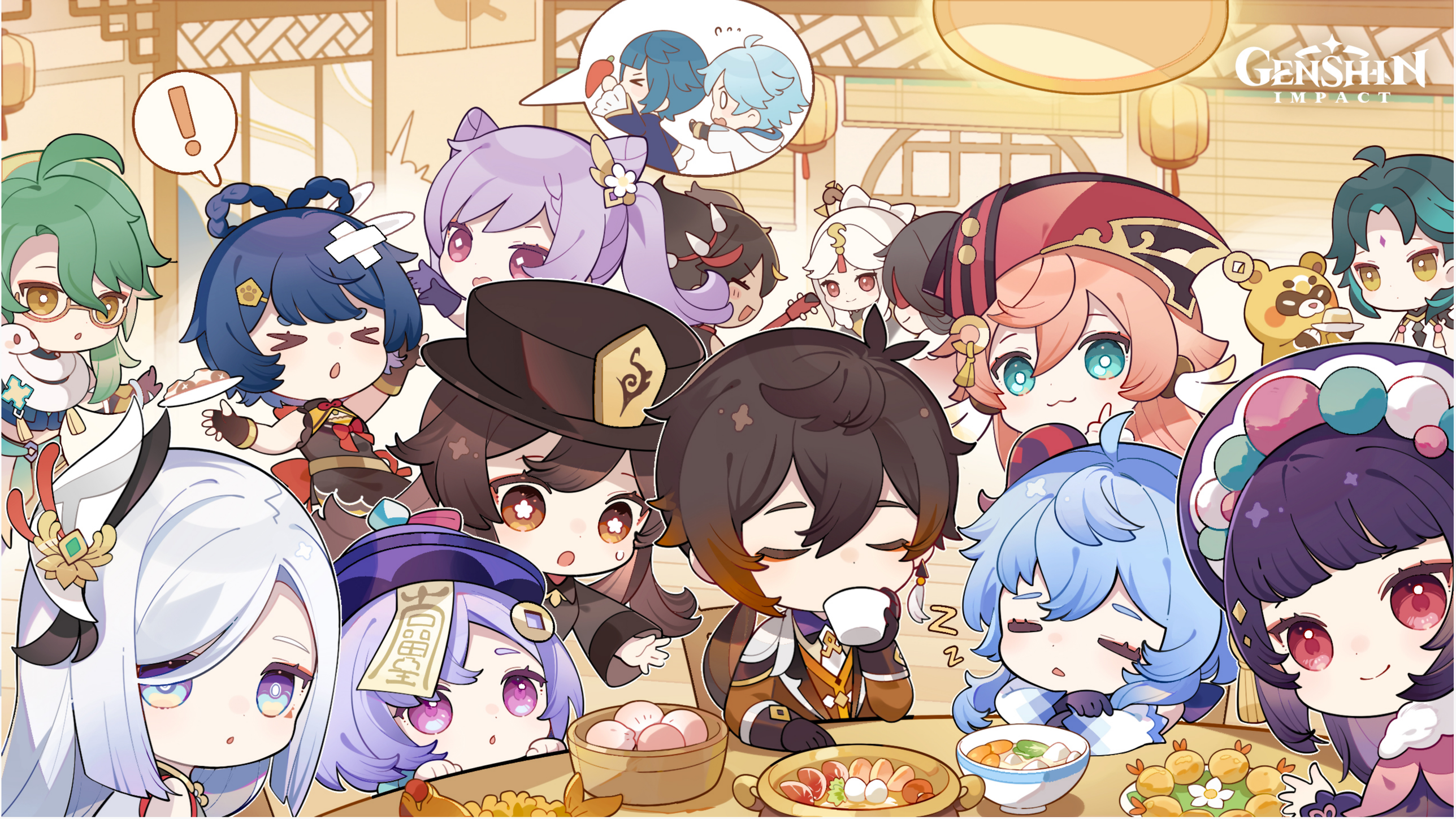

GENSHIN
IMPACT

Python 快速入门

- Python类（面向对象）
- Python异常处理
- Python自定义模块及导入方法
- Python常用标准库
- Python正则表达式
- Python数据库编程

Python 类（面向对象）

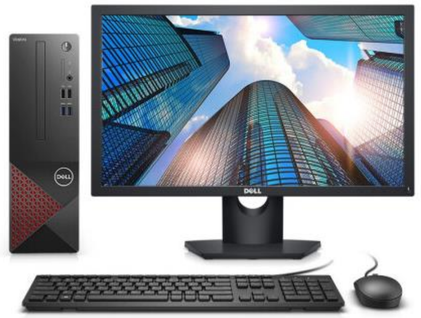
- 什么是面向对象编程?
- 类的定义
- 类的书写规范
- 类实例化
- 初始化函数

什么是面向对象编程?

面向过程编程：是一种**以过程为中心**的编程思想。这些都是以什么正在发生为主要目标进行编程。

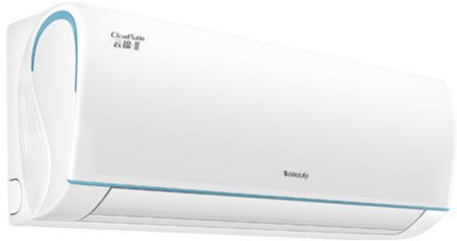
面向对象编程：是一种计算机编程架构，以**对象为中心**的编程思想，对现实世界理解和抽象的方法。

什么是面向对象编程?



电脑（类）：

- 每台电脑功能都一样，可用于办公、上网、看片等
- 每台电脑都包含相同的硬件，例如主机、显示器、鼠标、键盘等



空调（类）：

- 每台空调功能一样，用于制冷、制热、除湿等
- 每台空调也都包含相同的属性，例如品牌、冷暖类型、变频等

人（类）：

- 每个人都有相同的生活方式，例如工作、学习、睡觉等
- 每个人有着不同的特征，例如姓名、性别、年龄、身高等

什么是面向对象编程?

类、对象它们之间的关系：

- 类：类是对现实生活中一类具有共同特征的事物的抽象描述。例如电脑类、空调类、人类
- 对象：类的实体，实际存在的事物，例如电脑类的“主机”、“显示器”
- 类与对象的关系：类是由对象来定，这个过程叫做抽象化。用类创建对象，这个过程称为实例化

类的其他特点：

- 封装：把相同对象的功能（函数）、属性（变量）组合在一起
- 方法：对象的功能（例如电脑能上网、看片），实际在类里面就是函数，称为成员函数，也可以称为方法
- 属性：对象的特征（例如电脑都有主机、显示器）
- 实例化：用类创建对象，这个对象具体是什么东西，例如你用的电脑、我这个人

类的定义

使用class关键字定义类：

```
class ClassName():
    def funcName(self):
        pass
```

```
电脑（类）：
  特征（属性）：主机，显示器，键盘，鼠标...
    host = "4C8G"
    displayer = "27寸"
    keyboard = "机械键盘"
    mouse = "无线鼠标"
  功能（方法）：办公，上网，看片...
    def office():
      办公
    def internet():
      上网
    def movies():
      玩游戏
    ...
```

类的书写规范

示例：定义一个电脑类

```
class Computer():
    '''
    电脑类
    '''
    # 属性
    def __init__(self):
        self.host = "4C8G"
        self.displayer = "27寸"
        self.keyboard = "机械键盘"
        self.mouse = "无线鼠标"
    # 方法
    def office(self):
        return "办公"
    def internet(self):
        return "上网"
    def movies(self):
        return "看片"
```

类的书写规范：

- 类一般采用大驼峰命名，例如MyClass
- 类注释，用于说明该类的用途，提高可阅读性
- 类中只存在两种数据：属性和方法
- 声明属性必须赋值
- 声明方法的第一个参数必须是self，其他与普通函数一样
- 一般会使用__init__方法给类指定的初始状态属性

类的实例化

通常称为：

- 实例化：用类创建对象的过程
- 类实例：用类创建对象
- 实例属性：对象的属性
- 实例方法：对象调用的方法

示例：

```
pc = Computer()  # 类实例化，pc是类实例
print(pc.host)       # 访问类属性，查看电脑配置
print(pc.movies()) # 访问类方法，让电脑做事
```

初始化函数

示例：让某个人使用电脑做事

```
class Computer():
    '''
    电脑类
    '''
    # 属性
    def __init__(self, name):
        self.host = "4C8G"
        self.displayer = "27寸"
        self.keyboard = "机械键盘"
        self.mouse = "无线鼠标"
        self.name = name
    # 方法
    def office(self):
        return "%s在办公" %self.name
    def internet(self):
        return "%s在上网" %self.name
    def movies(self):
        return "%s在看片" %self.name

zhangsan = Computer("张三")
print(zhangsan.office())

lisi = Computer("李四")
print(lisi.movies())
```

初始化函数

示例：

```
class Calc():
    '''计算器类'''
    def __init__(self, num1, num2):
        self.num1 = num1
        self.num2 =num2
    def jia(self):
        return self.num1 + self.num2
    def jian(self):
        return self.num1 - self.num2
    def cheng(self):
        return self.num1 * self.num2
    def chu(self):
        return self.num1 / self.num2

calc = Calc(6, 6)
print(calc.jia())
```

Python 异常处理

- 什么是异常
- 捕获异常语法
- 异常类型
- 异常处理

什么是异常

什么是异常?

顾名思义，异常就是程序因为某种原因无法正常工作了，比如缩进错误、缺少软件包、环境错误、连接超时等都会引发异常。

一个健壮的程序应该把所能预知的异常都应做相应的处理，保障程序长期运行。

捕获异常语法

语法：

```
try:
    <代码块>
except [异常类型]:
    <发生异常时执行的代码块>
```

如果在执行 try 块里的业务逻辑代码时出现异常，系统会自动生成一个异常对象，该异常对象被提交给 Python 解释器，这个过程被称为引发异常。

当 Python 解释器收到异常对象时，会寻找能处理该异常对象的 except 块，如果找到合适的 except 块，则把该异常对象交给该 except 块处理，这个过程被称为捕获异常。如果 Python 解释器找不到捕获异常的 except 块，则运行时环境终止，Python 解释器也将退出。

常见异常类型

异常类型	用途
SyntaxError	语法错误
IndentationError	缩进错误
TypeError	对象类型与要求不符合
ImportError	模块或包导入错误；一般路径或名称错误
KeyError	字典里面不存在的键
NameError	变量不存在
IndexError	下标超出序列范围
IOError	输入/输出异常；一般是无法打开文件
AttributeError	对象里没有属性
KeyboardInterrupt	键盘接受到Ctrl+C
Exception	通用的异常类型；一般会捕捉所有异常
UnicodeEncodeError	编码错误

异常处理

示例：打印一个没有定义的变量

```
try:
    print(name)
except NameError:
    print("发生名称错误时，执行的代码")
```

示例：当不确定异常类型时，可以使用通用异常类型

```
try:
    print(name)
except Exception:
    print("发生名称错误时，执行的代码")
```

示例：保存异常信息

```
try:
    print(name)
except Exception as e:
    print("错误：%s" %e)
    print("发生名称错误时，执行的代码")
```

Python 自定义模块及导入方法

- 自定义模块
- __name__==“__main__” 作用
- 模块帮助文档

自定义模块

一个较大的程序一般应分为若干个程序块，若个程序块称为模块，每个模块用来实现一部分特定的功能。这样做的目的是为了将代码有组织的存放在一起，方便管理和重复使用。

定义一个模块：

```
# vi mymodule.py
name =  “aliang”
def count(a, b):
   result = a * b
   return f"{a}与{b}的乘积是: {result}  “
```

使用模块的方法：

```
import <模块名称>
from <模块名称> import <方法名>
```

注：模块名称即py文件名称

__name__==“__main__” 作用

mymodule.py作为一个模块，我们希望保留末尾测试代码，即上面调用函数和类，但也不希望再导入模块的时候执行。该怎么办呢？可以利用Python文件的一个内置属性__name__实现，如果直接运行Python文件，__name__的值是"__mian__"，如果import一个模块，那么模块的__name__的值是"文件名"。

```
# vi mymodule.py
name = "aliang“
def count(a, b):
    result = a * b
    return f"{a}与{b}的乘积是: {result}“

if __name__ == "__main__":
    print("我在手动执行这个程序")
    print(count(6,6))
    print(name)
```

模块帮助文档

我们知道，在定义函数或者类时，可以为其添加说明文档，以方便用户清楚的知道该函数或者类的功能。自定义模块也不例外，也可以添加说明文档，与函数或类的添加方法相同，即只需在模块开头的位置定义一个字符串即可。

```
# vi mymodule.py
"""
count()函数用于计算两个数值乘积
"""
def count(a, b):
    result = a * b
    return f"{a}与{b}的乘积是: {result}"
```

查看模块帮助文档：

方式1：

```
print(mymodule.__doc__)
```

方式2：

```
help(mymodule)
```

Python 常用标准库

模块	描述
os	操作系统管理
sys	解释器交互
platform	操作系统信息
glob	查找文件
shutil	文件管理
random	随机数
subprocess	执行Shell命令
pickle	对象数据持久化
json	JSON编码和解码
time	时间访问和转换
datetime	日期和时间
urllib	HTTP访问

官方文档标准库列表：https://docs.python.org/zh-cn/3.8/library/index.html

标准库：os

os库主要对目标和文件操作。

方法	描述
os.name	返回操作系统类型
os.environ	以字典形式返回系统变量
os.putenv(key, value)	改变或添加环境变量
os.listdir(path=’.’)	列表形式列出目录下所有目录和文件名
os.getcwd()	获取当前路径
os.chdir(path)	改变当前工作目录到指定目录
os.mkdir(path [, mode=0777])	创建目录
os.makedirs(path [, mode=0777])	递归创建目录
os.rmdir(path)	移除空目录，不能删除有文件的目录
os.remove(path)	移除文件
os.rename(old, new)	重命名文件或目录
os.stat(path)	获取文件或目录属性
os.chown(path, uid, gid)	改变文件或目录所有者
os.chmod(path, mode)	改变文件访问权限
os.symlink(src, dst)	创建软链接
os.unlink(path)	移除软链接
os.getuid()	返回当前进程UID
os.getlogin()	返回登录用户名
os.getpid()	返回当前进程ID
os.kill(pid, sig)	发送一个信号给进程
os.walk(path)	目录树生成器，生成一个三组 (dirpath, dirnames, filenames)
os.system(command)	执行shell命令，不能存储结果
os.popen(command [, mode='r' [, bufsize]])	打开管道来自shell命令，并返回一个文件对象

标准库：os

os.path类用于获取文件属性。

方法	描述
os.path.basename(path)	返回最后一个文件或目录名
os.path.dirname(path)	返回最后一个文件所属目录
os.path.abspath(path)	返回一个绝对路径
os.path.exists(path)	判断路径是否存在，返回布尔值
os.path.isdir(path)	判断是否是目录
os.path.isfile(path)	判断是否是文件
os.path.islink(path)	判断是否是链接
os.path.ismount(path)	判断是否挂载
os.path.getatime(filename)	返回文件访问时间戳
os.path.getctime(filename)	返回文件变化时间戳
os.path.getmtime(filename)	返回文件修改时间戳
os.path.getsize(filename)	返回文件大小，单位字节
os.path.join(a, *p)	加入两个或两个以上路径，以正斜杠"/"分隔。常用于拼接路径
os.path.split(	分隔路径名
os.path.splitext(	分隔扩展名

标准库：sys

sys库用于与Python解释器交互。

方法	描述
sys.argv	从程序外部传递参数 argv[0] #代表本身名字 argv[1] #第一个参数 argv[2] #第二个参数 argv[3] #第三个参数 argv[N] #第N个参数 argv #参数以空格分隔存储到列表
sys.exit([status])	退出Python解释器
sys.path	当前Python解释器查找模块搜索的路径，列表返回。
sys.getdefaultencoding()	获取系统当前编码
sys.platform	返回操作系统类型
sys.version	获取Python版本

标准库：platform

platform库用于获取操作系统详细信息。

方法	描述
platform.platform()	返回操作系统平台
platform.uname()	返回操作系统信息
platform.system()	返回操作系统平台
platform.version()	返回操作系统版本
platform.machine()	返回计算机类型
platform.processor()	返回计算机处理器类型
platform.node()	返回计算机网络名
platform.python_version()	返回Python版本号

标准库：glob

glob库用于文件查找，支持通配符（*、？ 、[]）

示例1：查找目录中所有以.sh为后缀的文件:

```
>>> glob.glob('/home/user/*.sh')
['/home/user/b.sh', '/home/user/a.sh', '/home/user/sum.sh']
```

示例2：查找目录中出现单个字符并以.sh为后缀的文件:

```
>>> glob.glob('/home/user/?.sh')
['/home/user/b.sh', '/home/user/a.sh']
```

示例3：查找目录中出现a.sh或b.sh的文件:

```
>>> glob.glob('/home/user/[a|b].sh')
['/home/user/b.sh', '/home/user/a.sh']
```

标准库：random

random库用于生成随机数。

方法	描述
random.randint(a,b)	随机返回整数a和b范围内数字
random.random()	生成随机数，它在0和1范围内
random.randrange(start, stop[, step])	返回整数范围的随机数，并可以设置只返回跳数
random.sample(array, x)	从数组中返回随机x个元素
choice(seq)	从序列中返回一个元素

标准库：subprocess

subprocess库用于执行Shell命令，工作时会fork一个子进程去执行任务，连接到子进程的标准输入、输出、错误，并获得它们的返回代码。

这个模块将取代os.system、os.spawn*、os.popen*、popen2.*和commands.*。

subprocess的主要方法：

subprocess.run()，subprocess.Popen(),subprocess.call

语法：subprocess.run(args, *, stdin=None, stdout=None, stderr=None, shell=False, cwd=None, timeout=None, check=False, encoding=None)

参数	说明
args	要执行的shell命令，默认是一个字符串序列，如['ls', '-al']或('ls', '-al')；也可是一个字符串，如'ls -al'，同时需要设置shell=True。
stdin stdout stderr	run()函数默认不会捕获命令运行结果的正常输出和错误输出，可以设置stdout=PIPE, stderr=PIPE来从子进程中捕获相应的内容；也可以设置stderr=STDOUT，使标准错误通过标准输出流输出。
shell	如果shell为True，那么指定的命令将通过shell执行。
cwd	改变当前工作目录
timeout	设置命令超时时间。如果命令执行时间超时，子进程将被杀死，并弹出TimeoutExpired 异常。
check	如果check参数的值是True，且执行命令的进程以非0状态码退出，则会抛出一个CalledProcessError的异常，且该异常对象会包含参数、退出状态码、以及stdout和stderr(如果它们有被捕获的话)。
encoding	如果指定了该参数，则 stdin、stdout 和 stderr 可以接收字符串数据，并以该编码方式编码。否则只接收 bytes 类型的数据。

标准库：subprocess

示例：

```
import subprocess
cmd = "pwd"
result = subprocess.run(cmd, shell=True, timeout=3, stderr=subprocess.PIPE,
stdout=subprocess.PIPE)
print(result)
```

run方法返回CompletedProcess实例，可以直接从这个实例中获取命令运行结果：

```
print(result.returncode) # 获取命令执行返回状态码
print(result.stdout) # 命令执行标准输出
print(result.stderr) # 命令执行错误输出
```

标准库：pickle

pickle模块实现了对一个Python对象结构的二进制序列化和反序列化。

主要用于将对象持久化到文件存储。

pickle模块主要有两个函数：

- dump() 把对象保存到文件中（序列化），使用load()函数从文件中读取（反序列化）
- dumps() 把对象保存到内存中，使用loads()函数读取

```
# 序列化
import pickle
computer = {"主机":5000,"显示器":1000,"鼠标":60,"键盘":150}
with open("data.pkl", "wb") as f:
    pickle.dump(computer, f)

# 反序列化
import pickle
with open("data.pkl", "rb") as f:
    print(pickle.load(f))
```

标准库：json

JSON是一种轻量级数据交换格式，一般API返回的数据大多是JSON、XML，如果返回JSON的话，需将获取的数据转换成字典，方面在程序中处理。

json与pickle有相似的接口，主要提供两种方法：

- dumps() 对数据进行编码
- loads() 对书籍进行解码

示例：将字典类型转换为JSON对象

```
import json
computer = {"主机":5000,"显示器":1000,"鼠标":60,"键盘":150}
json_obj = json.dumps(computer)
print(type(json_obj))
print(json_obj)
```

将JSON对象转换为字典：

```
import json
data = json.loads(json_obj)
print(type(data))
```

标准库：time

time库用于满足简单的时间处理，例如获取当前时间戳、日期、时间、休眠。

方法	描述
time.ctime(seconds)	返回当前时间时间戳
time.localtime([seconds])	当前时间，以stuct_time时间类型返回
time.mktime(tuple)	将一个stuct_time时间类型转换成时间戳
time.strftime(format[, tuple])	将元组时间转换成指定格式。[tuple]不指定默认是当前时间
time.time()	返回当前时间时间戳
time.sleep(seconds)	延迟执行给定的秒数

示例1：将当前时间转换为指定格式

```
import time
time.strftime("%Y-%m-%d %H:%M:%S")
```

示例2：将时间戳转换指定格式

```
now = time.time()
struct_time = time.localtime(now)
time.strftime('%Y-%m-%d %H:%M:%S',struct_time)
```

标准库：datetime

datetime库用于处理更复杂的日期和时间。

提供以下几个类：

类	描述
datetime.date	日期，年月日组成
datetime.datetime	包括日期和时间
datetime.time	时间，时分秒及微秒组成
datetime.timedelta	时间间隔
datetime.tzinfo	时区信息对象

```
from datetime import date, datetime
# 将当前系统时间转换指定格式
date.strftime(datetime.now(), '%Y-%m-%d %H:%M:%S')
# 获取当前系统时间
date.isoformat(date.today())
# 将时间戳转换指定格式
date_array = datetime.fromtimestamp(time.time())
date_array.strftime("%Y-%m-%d %H:%M:%S")

# 获取昨天日期
from datetime import date, timedelta
yesterday = date.today() - timedelta(days=1)
yesterday = date.isoformat(yesterday)
print(yesterday)
# 获取明天日期
tomorrow = date.today() + timedelta(days=1)
# 将时间对象格式化
tomorrow = date.isoformat(tomorrow)
print(tomorrow)
```

标准库：urllib

urllib库用于访问URL。

urllib包含以下类：

- urllib.request 打开和读取 URL
- urllib.error 包含 urllib.request 抛出的异常
- urllib.parse 用于解析 URL
- urllib.robotparser 用于解析 robots.txt 文件

用的最多是urllib.request 类，它定义了适用于在各种复杂情况下打开 URL，例如基本认证、重定向、Cookie、代理等。

标准库：urllib

```
from urllib import request
res = request.urlopen("http://www.ctnrs.com")
```

res是一个HTTPResponse类型的对象，包含以下方法和属性：

方法	描述
getcode()	获取响应的HTTP状态码
geturl()	返回真实URL。有可能URL3xx跳转，那么这个将获得跳转后的URL
headers	返回服务器header信息
read(size=-1)	返回网页所有内容。size正整数指定读取多少字节
readline(limit=-1)	读取下一行。size正整数指定读取多少字节
readlines(hint=0, /)	列表形式返回网页所有内容，以列表形式返回。sizehint正整数指定读取多少字节

标准库：urllib

示例1：自定义用户代理

```
from urllib import request
url = "http://www.ctnrs.com"
user_agent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108
Safari/537.36"
header = {"User-Agent": user_agent}
req = request.Request(url, headers=header)
res = request.urlopen(req)
print(res.getcode())
```

示例2：向接口提交用户数据

```
from urllib import request, parse
url = "http://www.ctnrs.com/login"
post_data = {"username":"user1","password":"123456"}
post_data = parse.urlencode(post_data).encode("utf8") #将字典
转为URL查询字符串格式，并转为bytes类型
req = request.Request(url, data=post_data, headers=header)
res = request.urlopen(req)
print(res.read())
```

Python 正则表达式

- 什么是正则表达式
- re 标准库
- 代表字符
- 代表数量
- 代表边界
- 代表分组
- 贪婪模式与非贪婪模式
- re 标准库其他方法
- 标志位

正则表达式

正则表达式是对字符串操作的一种逻辑方式，就是用实现定义好的一些特定字符及这些特定字符的组合，组成一个规则字符串，这个规则字符串就是表达对字符串的逻辑，给定一个正则表达式和另一个字符串，通过正则表达式从字符串我们想要的部分。

re 标准库

Python正则表达式主要由re标准库提供，拥有了基本所有的表达式。

方法	描述
re.compile(pattern, flags=0)	把正则表达式编译成一个对象
re.match(pattern, string, flags=0)	匹配字符串开始，如果不匹配返回None
re.search(pattern, string, flags=0)	扫描字符串寻找匹配，如果符合返回一个匹配对象并终止匹配，否则返回None
re.split(pattern, string, maxsplit=0, flags=0)	以匹配模式作为分隔符，切分字符串为列表
re.findall(pattern, string, flags=0)	以列表形式返回所有匹配的字符串
re.finditer(pattern, string, flags=0)	以迭代器形式返回所有匹配的字符串
re.sub(pattern, repl, string, count=0, flags=0)	字符串替换，repl替换匹配的字符串，repl可以是一个函数

re.compile方法

语法：re.compile(pattern, flags=0)

pattern 指的是正则表达式。flags是标志位的修饰符，用于控制表达式匹配模式

示例1：

```
import re
s = "this is test string"
pattern = re.compile('this')
result = pattern.match(s)
print(result.group())  #匹配成功后，result对象会增加一个group()方法，可以用它来获取匹配结果
```

re.match()方法

语法：re.match(pattern, string, flags=0)

示例：

```
result = re.match('this', s)
print(result.group())
```

代表字符

字符	描述
.	任意单个字符（除了\n）
[]	匹配中括号中的任意1个字符。并且特殊字符写在[]会被当成普通字符来匹配
[.-.]	匹配中括号中范围内的任意1个字符，例如[a-z],[0-9]
[^]	匹配[^字符]之外的任意一个字符
\d	匹配数字，等效[0-9]
\D	匹配非数字字符，等效[^0-9]
\s	匹配单个空白字符（空格、Tab键），等效[\t\n\r\f\v]
\S	匹配空白字符之外的所有字符，等效[^\t\n\r\f\v]
\w	匹配字母、数字、下划线，等效[a-zA-Z0-9_]
\W	与\w相反，等效[^a-zA-Z0-9_]

原始字符串符号“r”

“r”表示原始字符串，有了它，字符串里的特殊意义符号就会自动加转义符。

示例：

```
s = "123\\abc"
result = re.match(r"123\\abc", s)
print(result)
```

代表数量

字符	描述
*	匹配前面的子表达式0次或多次（无限次）
+	匹配前面的子表达式1次或多次
?	匹配前面的子表达式0次或1次
{n}	匹配花括号前面字符n个字符
{n,}	匹配花括号前面字符至少n个字符
{n,m}	匹配花括号前面字符至少n个字符，最多m个字符

代表数量

字符	描述
^	匹配以什么开头
$	匹配以什么结尾
\b	匹配单词边界
\B	匹配非单词边界

代表分组

字符	描述
|	匹配竖杠两边的任意一个正则表达式
(re)	匹配小括号中正则表达式。 使用\n反向引用，n是数字，从1开始编号，表示引用第n个分组匹配的内容。
(?P<name>re)	分组别名，name是表示分组名称
(?P=name)	引用分组别名

贪婪和非贪婪匹配

贪婪模式：尽可能最多匹配

非贪婪模式：尽可能最少匹配，一般在量词（*、+）后面加个？问号就是非贪婪模式。

```
s = "hello 666666"
result = re.match("hello 6+", s) # 贪婪匹配
print(result)
result = re.match("hello 6+?", s) # 非贪婪匹配
print(result)
```

其他方法

re.search(pattern, string, flags=0)	扫描字符串寻找匹配，如果符合返回一个匹配对象并终止匹配，否则返回None
re.split(pattern, string, maxsplit=0, flags=0)	以匹配模式作为分隔符，切分字符串为列表
re.findall(pattern, string, flags=0)	以列表形式返回所有匹配的字符串
re.finditer(pattern, string, flags=0)	以迭代器形式返回所有匹配的字符串
re.sub(pattern, repl, string, count=0, flags=0)	字符串替换，repl替换匹配的字符串，repl可以是一个函数

标志位

字符	描述
re.I/re.IGNORECASE	忽略大小写
re.S/re.DOTAIL	匹配所有字符，包括换行符\n，如果没这个标志将匹配除了换行符

Python 数据库编程

- pymysql模块
- 基本使用
- 增删改查

pymysql模块

pymysql是Python中操作MySQL的模块，其使用方法和MySQLdb几乎相同。但目前pymysql支持python3.x而后者不支持3.x版本。

pymysql是第三方模块，需要单独安装，首选通过pip安装PyMySQL:

```
# pip3 install pymysql
```

基本使用

先创建一个test库，再创建一张users表：

```
create database test;
use test;
create table user(
    id int primary key not null auto_increment,
    username varchar(50) not null,
    password varchar(50) not null
);
```

```
import pymysql
conn = pymysql.connect(host='192.168.31.61',
                       port=3308,
                       user='root',
                       password='123456',
                       db='test',
                       charset='utf8',
                       cursorclass=pymysql.cursors.DictCursor)
try:
    with conn.cursor() as cursor:
        # 创建一条记录
        sql = "insert into user(username, password) values('aliang', '123456')"
        cursor.execute(sql)

    # 写入到数据库
    conn.commit()

    with conn.cursor() as cursor:
        # 读取一条记录
        sql = "select id,username,password from user"
        cursor.execute(sql)
        result = cursor.fetchone()
        print(result)
finally:
    conn.close()
```

基本使用

connect()函数常用参数：

方法	描述
host	数据库主机地址
user	数据库账户
passwd	账户密码
db	使用的数据库
port	数据库主机端口，默认3306
connect_timeout	连接超时时间，默认10，单位秒
charset	使用的字符集
cursorclass	自定义游标使用的类。上面示例用的是字典类，以字典形式返回结果，默认是元组形式。

连接对象常用方法：

方法	描述
commit()	提交事务。对支持事务的数据库和表，如果提交修改操作，不适用这个方法，则不会写到数据库中
rollback()	事务回滚。对支持事务的数据库和表，如果执行此方法，则回滚当前事务。在没有commit()前提下。
cursor([cursorclass])	创建一个游标对象。所有的sql语句的执行都要在游标对象下进行。MySQL本身不支持游标，MySQLdb模块对其游标进行了仿真。

游标对象常用方法：

方法	描述
close()	关闭游标
execute(sql)	执行sql语句
excutemany(sql)	执行多条sql语句
fetchone()	从运行结果中取第一条记录，返回字典
fetchmany(n)	从运行结果中取n条记录，返回列表
fetchall()	从运行结果中取所有记录，返回列表

增删改查

示例：遍历查询结果

```
import pymysql
conn = pymysql.connect(host='192.168.31.61',
                       port=3308,
                       user='root',
                       password='123456',
                       db='test',
                       charset='utf8',
                       cursorclass=pymysql.cursors.DictCursor)
cursor = conn.cursor()

try:
    with conn.cursor() as cursor:
        sql = "select id,username,password from user"
        cursor.execute(sql)
        result = cursor.fetchall()
    for dict in result:
        print(f"ID: {dict['id']}, 用户名: {dict['username']}, 密码: {dict['password']}")
finally:
    conn.close()
```

谢谢