

Python:

起源: 荷兰人 Guido van Rossum 于 1989 年发明, 第一个公开发行人版发行于 1991 年

特点:

高级的数据结构, 缩短开发时间和代码量

面向对象

实现了代码的重用

扩展性强

可移植性

关键字少, 结构简单, 语法清晰

易读

内存管理由 python 解释器负责

官方网站: <https://www.python.org/>

官方提供源码压缩包, 编译安装需要依赖包:

gcc,gcc-c++,zlib-devel,openssl-devel,readline-devel,libffi-devel,sqlite-devel,tk-devel,tcl-devel

运行方式:

交互式: [root@fzr ~]# python3

使用解释器运行: [root@fzr untitled2]# python3 hello.py

赋予可执行权限后执行: [root@fzr untitled2]# ./hello.py

Python IDE: pycharm 官网: <http://www.jetbrains.com/pycharm/>

对象关系映射(Object Relational Mapping,ORM)

面向对象编程(Object Oriented Programming,OOP)

Python 语法结构:

python 代码块通过缩进对齐表达代码逻辑

缩进相同的一组语句构成一个代码块,称之为代码组

代码组的首行以关键字开始,以冒号结束,该行之后的一行或多行代码构成代码组

python 注释语句从#字符开始, 解释器会忽略掉该行#之后的所有内容

过长的语句可以使用\分解成几行

分号允许将多个语句写在同一行上, 但不推荐使用

结构:

起始行: 申明环境变量或解释器

文档字符串: 对模块说明, 使用单双三引号皆可

导入模块

全局变量声明

定义类

定义函数

程序主体

输出语句: print()

格式: print(value,...,sep=' ',end='\n')

不同的 value 之间用逗号隔开, 输出时自动添加空格

不同的 value 之间用加号表示字符拼接

使用*号可以将 value 重复 n 次

sep 参数定义字符之间的间隔符

end 表示输出完成后执行的字符串, 默认是回车

示例:

```
#!/usr/local/bin/python3
```

```
print('hello world')
```

```
print('hello','world')
print('hello'+'world')
print('hello','world',sep='+++')
print('hello world'*5)
print('hello world',end="")
```

输入语句: `input()`

注意: 读入的语句默认是字符串

示例:

```
user=input("请输入用户名")
print("hello",user)
```

python 标识符字符串规则:

第一个字符只能是大小写字母或下划线

其他字符只能是大小写字母或数字或下划线

变量名区分大小写

不需要预先声明变量的类型

变量名建议全部采用小写字母

变量名要简短、有意义

多个单词组成的变量名之间建议用下划线分隔

变量名建议用名词, 函数名建议用动词+名词, 类名建议采用驼峰形式

尽量不能与关键字重新

关键字: 通过 `keyword` 模块查看

示例:

```
import keyword
keyword.kwlist
keyword.iskeyword('pass')
```

变量赋值:

变量的类型和值在赋值的时候被初始化

变量赋值通过等号来执行, 支持增量赋值

示例:

```
a=5+6
print(a)
a=a+1
print(a)
```

变量作用域:

全局变量: 除非被删除掉, 否则到脚本运行结束前一致存在, 且对于所有的函数, 他们的值都是可以访问的

局部变量: 暂时存在, 仅仅只在定义它们的函数被调用时进入作用域生效。当函数被调用时, 局部变量被声明, 临时替换全局变量, 一旦函数结束, 局部变量离开作用域

global: 申明函数内的变量为全局变量

名称空间: 存放名字与变量值绑定关系的空间, 标识符依次搜索局部名称空间、全局名称空间和内建名称空间

运算符:

标准算术运算符:

+	加
-	减
*	乘

/ 除
// 取商的整数部分
% 取模，余数
** 幂

比较运算符：返回 1 表示真，返回 0 表示假。与 True 和 False 等价

== 等于
!= 不等于
> 大于
< 小于
>= 大于等于
<= 小于等于

赋值运算符：

= 简单赋值运算符
+= 加法赋值运算符
-= 减法赋值运算符
*= 乘法赋值运算符
/= 除法赋值运算符
%= 取模赋值运算符
**= 幂赋值运算符
//= 取整除赋值运算符

逻辑运算符：

and 布尔"与"：x and y，如果 x 为 False，返回 False，否则返回 y 的值。
not 布尔"非"：not x，如果 x 为 True，返回 False。如果 x 为 False，返回 True。
or 布尔"或"：x or y，如果 x 是 True，它返回 x 的值，否则返回 y 的值。

示例：

```
print(False and '5') #False
print('1' and '5')   #5
print(False or '5')  #5
print('1' or '5')    #1
print(not False)     #True
print(not 5)          #False
```

数字类型：

int: 有符号整数
bool: 布尔值：True:1； False:0
float: 浮点数
complex: 复数
数字表示方式：默认十进制
0o+8 进制数
0x+16 进制数
0b+2 进制数

字符串：被引号包含的字符的集合
支持使用成对的单引号或双引号
三引号可以用来包含特殊字符
只能重新定义，不可修改

切片：

使用索引运算符[]和切片运算符[:]可得到子字符串
格式：[起始下标:结束下标:步长]

从左到右第一个字符的索引是 0

从右往左第一个字符是-1

子字符串包含切片中的起始下标,但不包含结束下标

起始下标不能超过字符串长度，默认起始下标为 0

结束下标超过字符串长度的部分不显示，默认结束下标为-1

示例：

```
py_str='python'
len(py_str)      #取长度
py_str[0]        #第 1 个字符
py_str[2:5]      #第 3 个字符到第 5 个字符
py_str[1:]       #从第 2 个字符取到最后一个
py_str[:6]       #从第 1 个字符取到第 6 个
py_str[:]        #从头取到位
py_str[:2]       #从头取到位，步长为 2
py_str[1::-1]    #从第 2 个开始倒着取到开头
```

列表：

格式： [...]

可以保存任意数量任意类型的 python 对象

列表支持下标和切片操作

使用 in 或 not in 判断成员关系

使用 append 方法向列表中追加元素

示例：

```
alist=[10,'a','Bob',[1,2,3]]
len(alist)
alist[-1]        #取出列表倒数第一个对象
alist[2][1]      #取出第 3 个对象的第 2 个字符
alist[-1][1:]    #取出最后 1 个对象的第 2 个字符及后面的
'b' in alist     #判断 b 是否在列表中，结果为 False
'b' in alist[2]  #判断 b 是否在列表的第 3 个对象中，结果为 True
alist.append(10) #在列表末尾追加 10
alist.remove(10) #删除列表中第一个 10
alist[-1]=20     #最后一个值修改为 20
a=alist          #将 alist 的指针赋予 a，类似快捷方式
b=alist[:]       #将 alist 的值取出来赋予 b
```

元组：

格式： (...)

定义后不能修改，其他与列表类似

示例：

```
atuple=(1,2,"tom","alice")
'tom' in atuple  #结果为 True
atuple[0]        #取出第 1 个对象
```

字典：

格式： {...}

由键-值对(key-value)构成的映射数据类型

字典是无序的，只能通过键取值，不支持切片下标操作

示例：

```
user_dict={'name':'bob', 'age':23}
```

```
'bob' in user_dict      #结果为 False，无法取到值
'name' in user_dict     #结果为 True，可以取到键
user_dict['name']       #查看键为 name 的值
user_dict['sex']='male' #新增键值
```

数据类型：

按存储模型分类：

标量类型:数值、字符串

容器类型:列表、元组、字典

按更新模型分类：通过 hash()来判断

可变类型:列表、字典

不可变类型:数值、字符串、元组

按访问模型分类：

直接访问:数值

顺序访问:字符串、列表、元组

映射访问:字典

条件语句：

语法结构：

if 判断 1:

 语句 1

elif 判断 2:

 语句 2

else:

 其他情况执行的语句

表达式：

True，非零数值，非空的数据类型，空格等表示 True

False，0，空字符串，空列表，空元组，空字典表示 False

示例 1：

```
if ' ':
    print('xixi')
if "":
    print('lala')
```

示例 2：

```
import getpass
username=input('输入用户名')
password=getpass.getpass('密码')
if username=='tom' and password=='123456':
    print('success')
else:
    print('wrong')
```

示例 3：

```
import random
choice=['石头','剪刀','布']
prompt = ""0.石头
1.剪刀
2.布
请选择(0/1/2):"
```

```

computer=random.randint(0,2)
player=int(input(prompt))
if player not in [0,1,2]:
    print('瞎选，默认是布')
    player=2
print('你选择的是 %s, 计算机选择的是 %s'%(choice[player],choice[computer]))
c=player-computer
if c==1 or c==2:
    print('\033[31;1m 你赢了\033[0m')
elif c==0:
    print('\033[32;1m 平局\033[0m')
else:
    print('\033[31;1m 你输了\033[0m')

```

循环语句：

循环次数未知的情况下，建议采用 **while** 循环
 循环次数可以预知的情况下，建议采用 **for** 循环
break 语句可以结束当前循环体，跳转到下条语句
continue 语句终止本次循环，回到循环的顶端

while 语法结构：

```

while 条件:
    执行的语句

```

while-else 语法结构： **else** 子句只在循环完成后执行， **break** 语句也会跳过 **else**

```

while 条件:
    语句 1
else:
    语句 2

```

for 语法结构：

```

for 参数 in 可迭代对象:
    语句

```

range 函数：提供循环条件，起始默认为 0，结尾取不到，步长默认为 1

```

range(start, end, step)

```

示例 1：100 以内的正偶数相加

```

sum100 = 0
counter = 0
while counter < 100:
    counter += 1
    if counter % 2 :
        continue
    sum100 += counter
print ("result is %d" % sum100)
或
sum2=0
for i in range(2,101,2):
    sum2+=i
print(sum2)

```

示例 2：猜拳 5 次，如果没猜对则显示正确数字

```

import random
computer=random.randint(1,10)
counte=0
while counte<5:
    cai=int(input('请输入你猜的数字'))
    if computer > cai:
        print('猜小了')
    elif computer < cai:
        print('猜大了')
    else:
        print('猜对了')
        break
    counte+=1
else:
    print('正确的是',computer)

```

示例 3：测试所有类型的数据的循环

```

astr='hello'
alist=[10,20,30]
atuple=('bob','tom','jack')
adict={'name':'lisi','age':15}
for st in astr:
    print(st)
for i in alist:
    print(i)
for name in atuple:
    print(name)
for key in adict:
    print('%s:%s' %(key,adict[key]))

```

示例 4：兔子数列

```

num=int(input('数列的长度'))
fib=[0,1]
for i in range(num-len(fib)):
    fib.append(fib[-1]+fib[-2])
print(fib)

```

示例 5：九九乘法表

```

while True:
    num=int(input('请输入乘法表阶数，取值范围 1-9: '))
    if num in range(1,10):
        for i in range(1,num+1):
            for j in range(1,i+1):
                print('%s*%s=%s' %(j,i*i*j),end='\t')
            print()
        break
    else:
        print('输入错误，请重新输入')

```

列表解析：

用来动态地创建列表

语法: [表达式 for 参数 in 可迭代对象]

表达式应用于列表的每个成员, 最后所有成员组合成列表

示例:

```
[10+i for i in range(1,10,2)]  
[10+i for i in range(1,10) if i % 2 ]  
['192.168.1.%s' %i for i in range(1,255)]
```

文件打开:

`open()`以及 `file()`提供了初始化输入/输出(I/O)操作的通用接口

成功打开文件后会返回一个文件对象, 失败则报错

`open()`和 `file()`可以完全相互替换

语法:

```
变量=open(file,mode,buffering...)
```

访问模式 `mode`:

- `r` 以读方式打开(文件存在则打开, 不存在则报错)
- `w` 以写方式打开(文件存在则清空, 不存在则创建)
- `a` 以追加模式打开(必要时创建新文件)
- `r+` 以读写模式打开(类似 `r`)
- `w+` 以读写模式打开(类似 `w`)
- `a+` 以读写模式打开(类似 `a`)
- `b` 以二进制模式打开

示例: 读入非文件

```
f=open('/root/1.jpg','rb')  
print(f.read(4096))  
f.close()
```

文件迭代: 结合 `for` 循环逐行处理文件

示例: 读入文件

```
f=open('/tmp/passwd')  
for line in f:  
    print(line,end="")  
f.close()
```

文件读取:

`read(n)`: 将文件内容读取放入内存, `n` 可以指定一次读取的字节数, 默认值为-1, 表示读取直至末尾

注意: 随着 `read` 读写的进行, 文件指针向后移动, 直到结尾. 所以第二次执行 `read` 会得到空字符串

`readline(n)`: 读取文件到换行符\n结束, 作为字符串返回, `n` 可以指定字节, 但是会返回不完整的行

`readlines()`: 把每一行的数据读取存入放入列表中返回

示例:

```
f=open('/tmp/passwd')  
data=f.read()  
print(data)  
data=f.read()  
print(data)  
f.close()  
f=open('/tmp/passwd')  
data=f.read(4)  
print(data)
```



```
data=f.readline()
print(data)
data=f.readlines()
print(data)
f.close()
```

文件写入：

write(): 把含有文本数据或二进制数据块的字符串写入文件

writelines(): 将列表作为参数写入文件

注意：**write** 和 **writelines** 写入文件时，都不会自动添加行结束标志，需要手工输入

flush()会立刻保存数据到磁盘

close()时，会自动保存

示例：

```
f=open('/tmp/test','w')
f.write('hello')
f.flush()
f.writelines(['world\n','new'])
f.close()
```

with: 在将打开文件的操作放在 **with** 语句中，代码块结束后，文件将自动关闭

格式：

```
with open... as 参数:
    语句
```

示例：

```
with open('/tmp/passwd') as f:
    print(f.readline(),end='')
    #print(f.readlines())      #文件已关机，无法读取，会报错
```

seek(offset,whence): 移动文件指针到不同位置

offset: 相对 **whence** 的偏移量

whence: 0 表示文件开头,1 表示当前位置,2 表示文件的结尾

注意：如果文本文件没有使用二进制模式打开，只允许从文件头开始计算相对位置

tell():查看当前文件指针的位置

示例：

```
f=open('/tmp/passwd','rb')
print(f.tell())
print(f.read(4))
print(f.tell())
f.seek(2,1)
print(f.tell())
f.seek(-5,2)
print(f.tell())
f.seek(0,0)
print(f.tell())
```

标准文件：需要载入 **sys** 模块

sys.stdin: 标准输入，一般是键盘

sys.stdout: 标准输出，一般是显示器缓冲输出

sys.stderr: 标准错误，一般是显示器非缓冲输出

函数：

对程序逻辑进行结构化或过程化的一种方法

把重复代码放到函数中，既能节省空间，也有助于保持一致性

将整块代码巧妙地隔离成易于管理的小块

创建函数：

函数定义后，存储在内存中

函数内推荐写一个说明文档

内部函数：在函数体内创建另外一个函数

格式：

```
def 函数名(参数的集合):  
    函数体
```

调用函数：

前向引用：不允许在函数未声明之前被引用或者调用

函数名+圆括号调用函数

如果不加括号，只是对函数的引用，引用能查看函数的信息，例如函数名、在内存中的位置等

函数返回值：

使用 **return** 返回值，没有 **return** 时，默认返回 **None**

函数参数：

形式参数：函数定义时的参数是形式参数，由于不是实际存在的变量，又称虚拟变量

实际参数：调用函数时的参数是实际参数

传递参数：调用函数时，实参的个数需要与形参个数一致，实参依次将代表的值传递给形参

关键字参数：调用时通过参数名字来区分参数，允许参数缺失或者不按顺序

位置参数：使用 **sys** 模块的 **argv** 以列表方式接收，**sys.argv[0]** 为程序本身，**sys.argv[n]** 为第 **n** 个位置参数

默认参数：声明了参数的默认值，在函数调用时，不给该参数传值则调用默认值

参数组：把元组(非关键字参数)或字典(关键字参数)作为参数组传递给函数

定义时：

*表示元组

**表示字典

调用时：

*表示拆开后面的数据类型

**表示调用字典

示例：

```
def mk_fib(length=8):  
    "说明文件：这是一个兔子数列"  
    fib=[0,1]  
    for i in range(length-len(fib)):  
        fib.append(fib[-1]+fib[-2])  
    return fib  
print('兔子数列示例：')  
exam=mk_fib()  
print(exam)  
print('-'*50)  
n=int(input('想生成的数列长度：'))  
print(mk_fib(n))
```

示例：拷贝文件

```
import sys
```

```
def copy(src_name,dst_name):
    src_f=open(src_name,'rb')
    dst_f=open(dst_name,'wb')
    while True:
        data=src_f.read(4096)
        if not data:
            break
        dst_f.write(data)
    src_f.close()
    dst_f.close()
copy(sys.argv[1],sys.argv[2])
[root@fzr test]#python3 day2.py /etc/passwd /tmp/passwd
```

示例：参数组

```
def use_node(name, age):
    print('%s is %s' % (name, age))
```

```
use_node('bob', 25)
use_node(name='bob', age=25)
use_node('bob', age=25)
```

```
def fun1(*args):
    print(args)
```

```
def fun2(**kwargs):
    print(kwargs)
```

```
def fun3(x, y):
    print(x * y)
```

```
fun1()
fun1(10)
fun1('bob', 20)
fun2()
fun2(name='bob', age=20)
fun3(*[10, 5])
use_node(**{'name': 'bob', 'age': 25})
```

匿名函数：

lambda:

lambda 可以创造匿名函数，不需要以标准的 def 方式来声明

格式：lambda [arg1,arg2,...argN]: expression

一个完整的 lambda 语句就是一个表达式，定义体必须和声明放在同一行

filter:

调用一个布尔函数来迭代遍历每个序列中的元素;返回一个使函数返回值为 **true** 的元素的序列

格式: `filter(function, iterable)`

如果布尔函数比较简单, 可以直接使用 **lambda** 匿名函数代替

map:

接收一个函数和一个列表, 使用函数依次加工列表中的每个元素, 得到一个新的列表并返回

格式: `map(func, *iterables)`

示例:

```
import random

a = lambda x, y: x + y
print(a(3, 4))
alist = [random.randint(1, 100) for i in range(10)]
print(alist)
result = filter(lambda x: x % 2, alist)
print(list(result))
result2 = map(lambda x: x * 2 + 1, alist)
print(list(result2))
```

偏函数:

带有多个参数的函数, 如果其中某些参数基本上固定的, 那么就可以通过偏函数为这些参数赋默认值

格式: `functools.partial(func, *args, **keywords)`

示例:

```
import functools

def foo(a, b, c, d, e):
    return a + b + c + d + e

add = functools.partial(foo, a=1, b=2, c=3, d=4)
add1 = functools.partial(foo, *[1, 2, 3, 4])
add2 = functools.partial(foo, **{'a': 1, 'b': 2, 'c': 3, 'd': 4})
print(add(e=5))
print(add1(5))
print(add2(e=5))
```

递归函数: 函数包含了对自身的调用

示例: 阶乘

```
def factorial(n):
    if n == 1:
        return n
    return n * factorial(n - 1)

print(factorial(6))
```

生成器 (带 `yield` 语句的函数):

`yield` 语句返回一个值给调用者并暂停执行, `next()` 能从暂停的地方继续执行

当所有的 `yield` 语句都被执行后, 将不会再有返回

示例:

```
def mygen():
```

```
yield 'hello'
a = 10 + 20
yield a
yield [1, 2, 3]
```

```
for i in mygen():
    print(i)
```

闭包：函数中嵌套定义了另一个函数，内嵌函数引用了外部函数的变量，外部函数返回内嵌函数

用途：保护函数内的变量安全、闭包内的变量和内嵌函数会一直维持在内存中

装饰器：函数调用时用闭包进行装饰

应用情况：

引入日志

增加计时逻辑来检测性能

给函数加入事务的能力

示例：输出红色字体

```
def colour(func):
    def red(*args):
        return '\033[031;1m%s\033[0m' % func(*args)
    return red
```

```
@colour
def hello(word):
    return 'hello %s' % word
```

```
def welcome():
    return 'welcome'
```

```
print(hello('world'))
print(colour(welcome()))
```

示例：函数计时器

<https://github.com/a1441668968/test/blob/master/day5.py>

模块：

每一个以.py 为结尾的 python 文件都是一个模块

模块文件名去掉扩展名(.py)即为模块名

模块名不能与系统中已存在的模块重名，且要遵循命名规则

代码量过大了，需要将代码拆分为一些有组织的代码段，即模块

导入模块：

使用 import 导入整个模块

模块默认导入路径可通过 sys.path 查看列表，当前路径优先级最高

支持从 ZIP 归档文件导入模块，导入时会把 ZIP 文件当作目录处理

from 模块名 import 属性名：导入模块中的部分属性

模块被导入时会被加载，一个模块无论被导入多少次，只被加载一次，防止多重导入时无限相互加载

导入模块时，可以为模块取别名

导入模块时,模块的顶层代码会被执行

模块被导入后,程序会自动生成 pyc 的字节码文件以提升性能, python3 存放在__pycache__目录, python2 存放在当前目录

通过“模块名.属性”的方法调用

模块导入的特性:

模块具有一个__name__特殊属性

当模块文件直接执行时, __name__的值为__main__

当模块被另一个文件导入时, __name__的值就是该模块的名字

示例: 生成随机密码模块

```
import random
import string

key_poll=string.ascii_letters+string.digits

def gen_pass(n=8):
    "说明: 生成密码"
    result=""
    for i in range(n):
        result+=random.choice(key_poll)
    return result

if __name__ == '__main__':
    print(gen_pass())
    print(gen_pass(4))
    print(gen_pass(12))
```

包: 有层次的文件目录结构, 为平坦的名称空间加入有层次的组织结构, 把有联系的模块组合到一起
绝对导入:

相对导入: 应用于 from import 语句

shutil 模块: 兼容 shell

shutil.copyfileobj:

格式: copyfileobj(fsrc, fdst, length=16384)

将类似文件的对象 fsrc 的内容复制到类似文件的对象 fdst

shutil.copyfile:

格式: copyfile(src, dst, *)

将 src 的文件的内容(无元数据)复制到名为 dst 的文件, 然后返回目标文件路径

shutil.copy:

格式: copy(src, dst, *)

将文件 src 复制到 dst 文件或目录下, 返回目标文件路径。等同于: cp src dst

shutil.copy2:

格式: copy2(src, dst, *)

与 copy()类似, 会保留所有文件元数据。等同于: cp -p src dst

shutil.move:

格式: move(src, dst, copy_function=copy2)

递归地将文件或目录移动到目标位置, 保留元数据, 并返回目标路径。等同于: mv -r src dst

shutil.copytree:

格式: `copytree(src, dst, symlinks=False, ignore=None, copy_function=copy2)`

递归地复制目录, 返回目标目录, 目标目录不能已经存在。等同于: `cp -r src dst`

`shutil.rmtree:`

格式: `rmtree(path, ignore_errors=False, onerror=None)`

递归删除目录, 路径必须指向目录。等同于: `mv -r path`

`shutil.copymode:`

格式: `copymode(src, dst, *)`

将 `src` 的权限位复制到 `dst`。文件内容, 属主和属组不受影响

`shutil.copystat:`

格式: `copystat(src, dst, *)`

将 `src` 的权限位, 最后访问时间, 上次修改时间和标志复制到 `dst`

`shutil.chown:`

格式: `chown(path, user=None, group=None)`

更改给定路径的属主和属组, 默认参数 `None` 表示不修改, `user` 和 `group` 都不指定则报错

示例:

```
import shutil

with open('/etc/passwd', 'rb') as fsrc:
    with open('/tmp/user.txt', 'wb') as fdst:
        shutil.copyfileobj(fsrc, fdst)

shutil.copyfile('/etc/passwd', '/tmp/user2.txt')
shutil.copy('/etc/passwd', '/tmp')
shutil.copy2('/etc/passwd', '/tmp')
shutil.move('/tmp/user.txt', '/var/tmp')
shutil.copytree('/var/log', '/tmp/log')
shutil.rmtree('/tmp/log')
shutil.copymode('/etc/shadow', '/tmp/user2.txt')
shutil.copystat('/etc/shadow', '/tmp/user2.txt')
shutil.chown('/tmp/user2.txt', user='mysql', group='mysql')
shutil.chown('/tmp/user2.txt', group='ceph')
shutil.chown('/tmp/user2.txt', user='root')
```

`hashlib` 模块: 提供了 `md5`、`sha1`、`sha224`、`sha256`、`sha384`、`sha512` 等算法

`hashlib.md5()`

`hashlib.sha1()`

`hashlib.sha224()`

`hashlib.sha256()`

`hashlib.sha384()`

`hashlib.sha512()`

示例:

```
import hashlib
import sys

def check_md5(fname):
    with open(fname, 'rb') as f:
```

```

m = hashlib.md5()
while True:
    data = f.read(4096)
    if not data:
        break
    # for data in f.readlines():
    m.update(data)

return m.hexdigest()

print(check_md5(sys.argv[1])

```

tarfile 模块：允许创建、访问 tar 文件，向 tar 包追加文件，支持 gzip、bzip2 格式

tarfile.open(name, mode)

mode:

- 'r' 或 'r: *' 通过透明压缩打开阅读（推荐）。
- 'r:' 无压缩打开专用读取。
- 'r:gz' 使用 gzip 压缩打开阅读。
- 'r:bz2' 使用 bzip2 压缩打开阅读。
- 'r:xz' 使用 lzma 压缩打开阅读。
- 'x'或'x:' 完全无压缩地创建 tarfile。如果已存在,引发 FileExistsError 异常。
- 'x:gz' 使用 gzip 压缩创建 tarfile。如果已存在,引发 FileExistsError 异常。
- 'x:bz2' 使用 bzip2 压缩创建 tarfile。如果已存在,引发 FileExistsError 异常。
- 'x:xz'使用 lzma 压缩创建 tarfile。如果已存在,引发 FileExistsError 异常。
- 'a' 或 'a:' 打开，无需压缩。如果文件不存在，则创建该文件。
- 'w' 或 'w:' 打开未压缩的写入。
- 'w:gz' 打开 gzip 压缩写入。
- 'w:bz2' 打开 bzip2 压缩写入。
- 'w:xz' 打开 lzma 压缩写入。
- 'r|*' 打开用于透明压缩读取的 tar 块的流。
- 'r|' 打开未压缩的 tar 块的流以进行读取。
- 'r|gz' 打开 gzip 压缩的流进行阅读。
- 'r|bz2' 打开 bzip2 压缩的流进行阅读。
- 'r|xz' 打开 lzma 压缩的流进行阅读。
- 'w|' 打开未压缩的流进行写入。
- 'w|gz' 打开 gzip 压缩的流进行写入。
- 'w|bz2' 打开 bzip2 压缩的流进行写入。
- 'w|xz' 打开 lzma 压缩的流进行写入。

TarFile.add(name,recursive=True): 将文件名称添加到归档中。

name: 任何类型的文件

recursive: 递归设置

TarFile.extractall(path=".", numeric_owner=False): 解压归档中的所有成员

path 指定解压路径

numeric_owner=True 时，tarfile 中的 uid 和 gid 数字用于设置提取的文件的所有者/组

TarFile.extract(member, path="", set_attrs=True, *, numeric_owner=False): 解压归档中的指定成员

member 可以是文件名或 TarInfo 对象(TarInfo 对象：TarFile 中的一个成员)

path 指定解压路径

set_attrs=True 设置文件属性: owner, mtime, mode。False 表示不设置

numeric_owner=True, tarfile 中的 uid 和 gid 数字用于设置提取的文件的所有者/组。

TarFile.close(): 关闭 TarFile

示例:

```
import tarfile

tar = tarfile.open('/tmp/test.tar.gz', 'w:gz')
tar.add('/etc/hosts')
tar.add('/etc/passwd')
tar.close()
tar = tarfile.open('/tmp/test.tar.gz', 'r:gz')
tar.extractall()
tar.close()
```

内建函数（内置函数）:

例如: len(), str(), input(), help()等

查看: <https://www.runoob.com/python3/python3-built-in-functions.html>

序列:

序列包括列表、元组、字符串

序列类型操作符:

sequence[index]	获得该下标的元素
sequence[index1:index2]	获得下标 1 到下标 2 之间的元素集合
sequence * n	序列重复 n 次
sequence1 + sequence2	连接序列 1 和序列 2
object in sequence	判断对象是否包含在序列中
object not in sequence	判断对象是否不包含在序列中

序列相关的内建函数:

list(iterable)	把可迭代对象转换为列表, list()创建一个空列表
str(object)	把对象转换成字符串
tuple(iterable)	把可迭代对象转换成元组, tuple()创建空元组
len(object)	返回一个容器中对象的数量, 大部分情况是长度
max(iterable)	返回可迭代对象中的最大值
enumerate(iterable)	接受一个可迭代对象作为参数, 返回索引及迭代值
reversed(sequence)	接受一个序列作为参数, 转化为逆序返回
sorted(iterable)	接受一个可迭代对象作为参数, 返回一个按升序排列的列表

示例:

```
import random

alist = [10, 'jack']
for ind in range(len(alist)):
    print('%s %s' % (ind, alist[ind]))
for ind, val in enumerate(alist):
    print('%s %s' % (ind, val))
atuple = [random.randint(1, 100) for i in range(10)]
sorted(atuple)
```

```
sorted('hello')
list(reversed(atuple))
```

字符串：

比较操作符：字符串大小按 ASCII 码值大小进行比较

切片操作符：[], [:], [::]

成员关系操作符：in, not in

格式化操作符：

- %c 转换成 ascii 字符
- %s 优先用 str()函数进行字符串转换
- %d 转成有符号十进制整数
- %o 转成无符号八进制数
- %x 转成无符号十六进制数
- %e 转成科学计数法
- %f 转成浮点数

辅助指令：

- * 定义宽度或者小数点精度
- 填充字符放在后面
- + 在正数前面显示加号，只对%d有效
- % d(%和d之间有一个空格) 在正数前面显示空格
- # 在八进制数前面显示0,在十六进制前面显示'0x'，只对%o和%x有效
- 0 显示的位数不够时，在数字前面填充0，默认是空格，只对%d有效

原始字符串操作符：字符将不再具有特殊含义，直接按字面意思使用

格式：r'字符串'

format 函数：

使用位置参数：

```
'{} is {}'.format('bob', 15)
'{1} is {0}'.format(15, 'bob')
```

使用关键字参数：

```
'name is {name},age is {age}'.format(name='bob',age=23)
'姓名：{name},年龄：{age}'.format(**{'name': 'bob', 'age': 23})
'姓名：{0[name]},年龄：{1[age]}'.format({'name': 'bob', 'age': 23},{'age':24})
```

填充与格式化：

```
{[填充字符][对齐方式：<表示左对齐，>表示右对齐][宽度]}
'{:<10} is {:<8}'.format('bob', 15)
'{:<10} is {:>8}'.format('bob', 15)
```

使用索引：

```
'姓名：{0[0]},年龄：{0[1]}'.format(['bob',23])
```

subprocess.call：在 python 中使用 shell 命令

需要载入 subprocess 模块

格式：subprocess.call('命令',shell=False)

使用时，需要把 shell 改为 True

字符串相关的内建函数：

字符串.capitalize()：把字符串的第一个字符改为大写

字符串.center(width, fillchar)：将字符串居中，其余部分用空格填充，fillchar 指定填充字符

字符串.count(sub, start, end)：统计 sub 在字符串中出现的次数，start 和 end 可以指定范围

字符串.endswith(suffix, start, end)：判断字符串是否以 suffix 结束，start 和 end 可以指定范围

字符串.startswith(prefix, start, end): 判断字符串是否以 prefix 开头, start 和 end 可以指定范围

字符串.islower(): 判断字符串中是否有字母且都是小写

字符串.isupper(): 判断字符串中是否有字母且都是大写

字符串.strip('chars'): 删除字符串两端的字符, 默认是空白字符

字符串.lstrip(): 只删除左边

字符串.rstrip(): 只删除右边

字符串.upper(): 将字符串中的小写字母改为大写

字符串.lower(): 将字符串中的大写字母改为小写

字符串.split(sep=",maxsplit=-1): 以 sep 为分隔符对字符串切片, maxsplit 指定最大切片数量, 默认不限制

示例:

```
'%s %d %s' % ('bob', -25, 60)
'%s %d %f' % ('bob', -25, -60.2)
'%s %d %5.1f' % ('bob', -25, -60.22)
'%s %d %5.4f' % ('bob', -25, -60.22)
'%c' % 97
'%#o' % 11
'%#x' % 11
'%10s%5s' % ('name', 20)
'%-10s%5s' % ('name', 20)
'%-10s%05d' % ('name', 20)
'{} is {}'.format('bob', 15)
'{1} is {0}'.format('bob', 15)
'{:<10} is {:<8}'.format('bob', 15)
'姓名: {0[0]},年龄: {0[1]}'.format(['bob',23])
```

示例:

```
import subprocess
import sys
import randpass
```

```
def add_user(username, password, fname):
    data = "用户信息:%s,%s"
    subprocess.call('useradd %s' % username, shell=True)
    subprocess.call('echo %s | passwd --stdin %s' % (password, username), shell=True)
    with open(fname, 'a') as f:
        f.write(data % (username, password))

if __name__ == '__main__':
    username = sys.argv[1]
    password = randpass.gen_pass()
    add_user(username, password, '/tmp/adduser.txt')
```

示例:

```
astr = 'hello world 2018!'
astr.capitalize()
astr.title()
```

```
astr.center(50)
astr.center(50, '-')
astr.count('w')
astr.count('l', 3, 12)
astr.endswith('l')
astr.endswith('o', 3, 12)
astr.startswith('e', 1, 10)
astr.islower()
astr.isdigit()
astr.isalnum()
astr.upper()
astr.strip()
astr.lstrip()
astr.rstrip()
astr.upper()
astr.lower()
"192.168.1.1".split('.')
'-'.join(['hello', 'world', '2018'])
```

列表：

更新列表：列表[下标]=新值

列表内建函数：

列表.append(object)：在列表最后追加对象

列表.count(value)：统计对象在列表中出现的次数

列表.insert(index, objec)：在指定位置之前插入对象

列表.reverse()：将列表倒转

列表.clear()：清空列表

列表.extend(iterable)：将对象拆分后追加入列表

列表.pop(index)：删除列表中指定的一个对象，默认是最后一个

列表.sort(reverse=False)：对列表进行升序排序，reverse=True 时降序排列。排序时，列表中数字和字符串不能同时存在

列表.copy()：复制列表

列表.index(value, start, stop)：返回索引，可以通过 start 和 stop 指定范围

列表.remove(value)：删除第一个匹配项

示例：

```
alist=[10,2,3,'bob','tom']
alist[0]=10
alist[1:3]=[20,30]
alist[2:2]=[22,24,26,28]
alist.pop()
alist.pop(3)
alist.pop(alist.index('bob'))
alist.sort()
alist.append(40)
alist.extend('new')
alist.extend(['hello','world','2018'])
alist.remove(20)
```

```
alist.index('bob')
blist=alist.copy()
alist.insert(1,'alice')
alist.reverse()
alist.count(30)
alist.clear()
```

示例：用列表模拟栈的功能

<https://github.com/a1441668968/test/blob/master/day3.py>

元组：

注意：

创建单元素元组的时候,需要在元素后加上一个逗号，否则会变成字符串

元组本身不可变，但元组内的元素存在列表和字典，则该元素中的内容可变

元组内建函数：

元组.count(value)：查询值在元组中出现的次数

元组.index(value)：查询值在元组中的索引

字典：

创建字典：

通过 {} 操作符创建字典

通过 dict() 创建字典

通过 fromkeys() 创建具有相同值的默认字典

访问字典：使用相应的键

更新字典：通过键更新字典

使用 update() 更新或者直接引用键更新

字典中有该键时：更新相关值，键不允许重复

字典中没有该键时：向字典中添加新值

删除字典：

通过 del 可以删除字典中的元素或整个字典

使用 clear() 可以清空字典

使用 pop() 可以删除字典中的元素

字典操作符：

查找操作符：[键]，查找键所对应的值

in 和 not in 判断键是否存在于字典中

相关函数：

字典.keys()：查看字典中键的列表

字典.values()：查看字典中所有值的列表

字典.items()：查看字典中键值对的列表，以元组的方式显示

字典.get(key, default)：查看 key 对应的值，如果不存在则返回 default 的值，default 默认为 None

字典.copy()：复制字典

字典.setdefault(key, default)：如果字典中不存在 key，则赋值为 default，default 默认为 None

len(字典)：返回字典中键的数目

示例：

```
adict=dict()
print(dict(['ab','cd']))
bdict=dict([('name','bob'),('age',25)])
cdict = {}.fromkeys(['zhang3', 'li4', 'wang5'], 20)
```

```

for key in cdict:
    print('%s:%s' % (key, cdict[key]))
print('%(name)s:%(age)s'%bdict)
bdict['name']='jack'
bdict['email']='abc@123.com'
len(bdict)
bdict.keys()
bdict.values()
bdict.items()
bdict.get('name')
bdict.get('qq','not found')
bdict.get('age','not found')
bdict.update({'tel':'123456789'})
adict = bdict.copy()
bdict.setdefault('add', 'china')

```

集合：

由不同的元素组成，集合的成员通常被称做集合元素
 集合对象是一组无序排列的值
 集合相当于无值的字典，所有的成员都相当于字典的键
 set 定义可变集合

frozenset 定义不可变集合

集合类型操作符：

- in 和 not in 检查成员
- len()检查集合长度
- 集合成员可用 for 迭代
- | 并集
- & 交集
- 差补，前者有，后者没有的成员

集合内建函数：

集合 1.intersection(集合 2)：交集，等同于：集合 1&集合 2

集合 1.union(集合 2)：并集，等同于：集合 1|集合 2

集合 1.difference(集合 2)：差补，等同于：集合 1-集合 2

集合.add()：添加单个集合成员

集合.update()：以列表形式批量添加集合成员

集合.remove()：移除成员

集合 1.issubset(集合 2)：判断集合 1 是否是集合 2 的子集，返回 True 和 False

集合 1.issuperset(集合 2)：判断集合 1 是否是集合 2 的超集，返回 True 和 False

示例：

```

aset=set('abcd')
bset=set('defg')
cset=aset|bset
aset.union(bset)
aset & bset
aset.intersection(bset)
aset-bset
aset.difference(bset)

```

```

aset.issubset(cset)
cset.issuperset(aset)
aset.add('new')
aset.update(['aaa','bbb'])
aset.remove('bbb')

```

示例：比对两个文件的差异

```

with open('passwd') as f:
    aset = set(f)
with open('mima') as f:
    bset = set(f)
with open('diff', 'w') as f:
    f.writelines(aset - bset)

```

time 模块：

表示方式：

timestamp 时间戳

UTC(Coordinated Universal Time,世界协调时)

元组：struct_time

struct_time 模块：

含义	属性	取值范围
年	tm_year	2000
月	tm_mon	1-12
日	tm_mday	1-31
小时	tm_hour	0-23
分	tm_min	0-59
秒	tm_sec	0-61
周	tm_wday	0-6(0 表示周一，以此类推)
第几天	tm_yday	1-366
夏令时	tm_isdst	默认为-1

模块功能：

time.localtime(seconds)：返回当前时区指定秒数的 struct_time，seconds 不指定时表示当前时间

time.gmtime(seconds)：返回 UTC 时区指定秒数的 struct_time，seconds 不指定时表示当前时间

time.time()：查看当前时间的秒数，相对于 1970.1.1

time.mktime()：将 struct_time 转化为秒数

time.sleep()：线程推迟指定的时间运行。单位秒

time.asctime(tuple)：把一个表示时间的元组或 struct_time 转化为字符串，tuple 不指定时，默认

time.localtime()

time.ctime(seconds)：把指定秒数以 time.asctime()的格式展示，seconds 不指定时表示当前时间

time.strftime(format, tuple)：把一个表示时间的元组或 struct_time 转化为 format 参数指定格式的字符串，tuple 不指定时，默认 time.localtime()

time.strptime(string, format)：将符合 format 参数指定格式的时间转化为 struct_time

时间格式：

格式	含义	格式	含义
%a	本地简化星期名称	%m	月份（01 - 12）
%A	本地完整星期名称	%M	分钟数（00 - 59）
%b	本地简化月份名称	%p	本地am或者pm的相应符
%B	本地完整月份名称	%S	秒（01 - 61）
%c	本地相应的日期和时间	%U	一年中的星期数（00 - 53，星期日是一个星期的开始）
%d	一个月中的第几天（01 - 31）	%w	一个星期中的第几天（0 - 6，0是星期天）
%H	一天中的第几个小时（24小时制，00 - 23）	%x	本地相应日期
%I	第几个小时（12小时制，01 - 12）	%X	本地相应时间
%j	一年中的第几天（001 - 366）	%y	去掉世纪的年份（00 - 99）
%Z	时区的名字	%Y	完整的年份

示例：

```
import time
time.localtime()
time.gmtime()
time.time()
time.mktime(time.localtime())
time.sleep(1)
time.asctime()
time.ctime()
time.strftime('%Y-%m-%d')
time.strftime('%H-%M-%S')
time.strptime('2018-07-20','%Y-%m-%d')
```

datetime 模块：

datetime.datetime.today(): 返回当前时区当前时间的 datetime 对象

datetime.datetime.now(tz): 返回指定时区当前时间的 datetime 对象，tz 默认 None，表示当前时区

datetime.datetime.strptime(string, format): 将有格式的字符串转化为 datetime 对象

datetime.datetime.ctime(datetime): 将 datetime 对象转化为字符串

datetime.datetime.strftime(datetime,format): 将 datetime 对象转化为指定格式的字符串

datetime.timedelta(时间参数): 时间计算

时间参数：days=天,hours=时,minutes=分,seconds=秒,microseconds=毫秒,weeks=周

示例：

```
import datetime
dt=datetime.datetime.today()
datetime.datetime.now()
datetime.datetime.strptime('2018/7/20','%Y/%m/%d')
datetime.datetime.strptime('2018~7~20','%Y~%m~%d')
datetime.datetime.ctime(dt)
datetime.datetime.strftime(dt,'%Y*%m*%d')
dt+datetime.timedelta(days=10,hours=3)
```

异常处理：

异常信息：<http://www.runoob.com/python/python-exceptions.html>

NameError: 未声明或初始化对象

IndexError: 序列中没有此索引

SyntaxError: 语法错误

KeyboardInterrupt: 用户中断执行

EOFError: 没有内建输入，到达 EOF 标记

IOError: 输入输出操作失败

try-except 语句:

try:

被监控的语句

except 异常原因:

异常处理代码

else:

不发生异常执行的语句

finally:

无论是否发生异常都执行的语句

异常参数: 将异常原因传递给参数

except 异常原因 as e:

示例:

try:

n = int(input('输入一个数: '))

result=100/n

except (ValueError,ZeroDivisionError):

print('无效的数字')

except (KeyboardInterrupt, EOFError):

print('byebye')

else:

print(result)

finally:

print('over')

触发异常:

raise: 可以指定一个异常类，执行时，会自动创建指定异常类的一个对象

assert: 断言，等价于布尔值为真的判定

示例:

def set_age(name, age):

if not 0 < age < 150:

raise ValueError('超过范围')

print('%s is %s' % (name, age))

def set_age2(name, age):

assert 0 < age < 150, '超过范围'

print('%s is %s' % (name, age))

if __name__ == '__main__':

set_age('zhang3', 30)

set_age2('lisi', 20)

os 模块: python 访问操作系统功能的主要接口

`os.getcwd()`: 返回当前工作目录, 相当于 `pwd`

`os.listdir(path)`: 列出指定目录下文件, `path` 不指定时表示当前目录, 相当于 `ls`

`os.mkdir(path)`: 创建目录, 相当于 `mkdir`

`os.chdir(path)`: 改变工作目录, 相当于 `cd`

`os.mknod(path)`: 创建文件, 相当于 `touch`

`os.symlink(src, dst)`: 创建 `src` 的软链接 `dst`

`os.popen(command[, mode[, bufsize]])`: 从一个命令打开一个管道

`command` 指定命令

`mode` 指定模式权限, 默认为 `r`, 可选 `w`

`bufsize` 指明了文件需要的缓冲大小, 默认为使用系统的默认值。0 表示无缓冲; 1 表示行缓冲; 其它正值表示使用参数大小的缓冲(单位: 字节)。负值表示默认

`os.path.isfile(path)`: 判断 `path` 是否是文件

`os.path.islink(path)`: 判断 `path` 是否是软链接

`os.path.isdir(path)`: 判断 `path` 是否是目录

`os.path.exists(path)`: 判断 `path` 是否存在

`os.path.basename(path)`: 取出 `path` 的文件名, `path` 可以不存在

`os.path.dirname(path)`: 取出 `path` 的目录名, `path` 可以不存在

`os.path.split(path)`: 将 `path` 拆分为目录名和文件名, 以元组形式显示, `path` 可以不存在

`os.path.splitext(path)`: 将目标文件的文件名和扩展名拆分

`os.path.join(pathname)`: 将至少 2 个目录或文件名拼接

`os.path.abspath(path)`: 显示 `path` 的绝对路径

示例:

```
import os
os.getcwd()
os.listdir()
os.listdir('/tmp')
os.mkdir('/tmp/a')
os.chdir('/tmp/a')
os.mknod('test')
os.symlink('/etc/passwd', 'link')
os.path.isfile('test')
os.path.islink('link')
os.path.isdir('/tmp')
os.path.exists('/tmp')
os.path.basename('/ttd/aaa/bbb')
os.path.dirname('/ttd/aaa/bbb')
os.path.split('/ttd/aaa/bbb')
os.path.join('/ttd/aaa', 'bbb')
os.path.abspath('test')
```

`pickle` 模块:

可以在一个文件中储存任何 `python` 对象, 取出来的时候数据类型不变, 以二进制方式存储

`pickle.dump(obj, file)`: 将对象存储入文件

`pickle.load(file)`: 从文件中读取

示例:

```
import pickle
```

```

first_list = ['a', 'b', 2]
with open('/tmp/listdata', 'wb') as f:
    pickle.dump(first_list, f)
with open('/tmp/listdata', 'rb') as f:
    new_list = pickle.load(f)
print(new_list)

```

示例：钱包案例

<https://github.com/a1441668968/test/blob/master/day4.py>

OOP(Object Oriented Programming)面向对象编程：

类(Class)：用来描述具有相同的属性和方法的对象的集合。定义了该集合中每个对象所共有的属性和方法。

对象：通过类定义的数据结构实例。对象包括两个数据成员(类变量和实例变量)和方法。

创建类：类名建议使用驼峰形式

```
class 类名:
```

...

创建实例：实例是根据类创建出来的具体对象。创建时，自动将实例本身作为第一个参数传递

构造器：__init__，默认自动调用，实例本身作为第一个参数传递给 self

除构造器外的方法绑定：实例方法需要明确指定名称调用

示例：

```

class BearToy:
    def __init__(self, bear_name, colour, size):
        self.name = bear_name
        self.colour = colour
        self.size = size

    def sing(self):
        print('lala')
        print('my name is %s' % self.name)

```

```
tidy = BearToy('Tidy', 'write', 'large')
```

```
tidy.sing()
```

组合：在一个类中创建其它类的实例作为组件，既增加功能和代码重用性，又增强类对象功能

继承：基类(父类)的属性继承给派生类(子类)

子类可以继承基类的任何属性，包括数据属性和方法

子类中有和父类同名的方法，父类方法将被覆盖

super(type, type2)：子类调用父类同名方法时使用，父类方法将被覆盖，用于子类拥有父类没有的类时使用

多重继承：一个类可以是多个父类的子类，子类拥有所有父类的属性。如果有相同的属性，调用顺序：先下后上，先左后右

示例：组合和继承

```
import time
```

```
class Contact:
```

```
    def __init__(self, phone, email):
```

```

        self.phone = phone
        self.email = email

    def call(self):
        print(self.phone)

class BearToy:
    def __init__(self, colour, size, phone, email):
        self.colour = colour
        self.size = size
        self.vendor = Contact(phone, email)

class NewBear(BearToy):
    def run(self):
        print('running...')

class UpdateBear(BearToy):
    def __init__(self, colour, size, phone, email, date):
        super(UpdateBear, self).__init__(colour, size, phone, email)
        self.date = date

    def test(self):
        print('running')

bigbear = BearToy('write', 'big', 12345, 'bear@qq.com')
bigbear.vendor.call()
newbear = NewBear('write', 'big', 12345, 'bear@qq.com')
newbear.run()
big_new_bear = UpdateBear('black', 'large', 56789, 'hello@qq.com', time.localtime())
big_new_bear.vendor.call()
big_new_bear.test()

```

示例：多重继承

```

class A:
    def foo(self):
        print('A')

class B:
    def foo(self):
        print('B+')
    def bar(self):
        print('B')

```

```
class C(A, B):
    def foo(self):
        print('C')
```

```
C().foo()
```

```
C().bar()
```

类方法：使用 `classmethod` 装饰器定义，在不创建实例时即可调用类

参数 `cls` 表示类本身

格式：

```
@classmethod
```

```
def ...(cls,...):
```

```
...
```

静态方法：使用 `staticmethod` 装饰器定义的一个函数

注意：

由于静态方法没有创建实例，也就没有对象，所以也就没有字段，没有方法

由于静态方法不会访问到 `class` 本身，所以可以放在 `class` 的内部，也可以放在 `class` 外部

格式：

```
@staticmethod
```

```
def ...(...):
```

```
...
```

示例：

```
class Date:
```

```
    def __init__(self, year, month, day):
```

```
        self.year = year
```

```
        self.month = month
```

```
        self.day = day
```

```
    @classmethod
```

```
    def create(cls, dstr):
```

```
        y, m, d = map(int, dstr.split('-'))
```

```
        dt = cls(y, m, d)
```

```
        return dt
```

```
    @staticmethod
```

```
    def is_date_valid(dstr):
```

```
        y, m, d = map(int, dstr.split('-'))
```

```
        return 1 <= d <= 31 and 1 <= m <= 12 and y < 4000
```

```
day = Date.create('2018-7-5')
```

```
birth_day = Date(2000, 5, 6)
```

```
print(day.year)
```

```
print(birth_day.year)
```

```
print(Date.is_date_valid('2018-7-23'))
```

`__init__`方法：实例化类时默认会调用的方法

`__str__`方法: 打印、显示实例时调用方法, 返回字符串

格式:

```
def __str__(self):  
    ...  
    return ...
```

`__call__`方法: 用于创建可调用的实例

格式:

```
def __call__(self):  
    print(...)
```

示例:

```
class Book:  
    def __init__(self, title, author):  
        self.title = title  
        self.author = author  
  
    def __str__(self):  
        return 'book name is %s' % self.title  
  
    def __call__(self, *args, **kwargs):  
        print('%s is written by %s' % (self.title, self.author))  
  
my_book = Book('hello world', 'zhang3')  
print(my_book)  
my_book()
```

`re` 模块: 支持扩展正则表达式

补充:

- `\d` 匹配任意数字,与[0-9]同义
- `\D` `\d` 取反
- `\w` 匹配任意数字字母字符,与[0-9a-zA-Z_]同义
- `\W` `\w` 取反
- `\s` 匹配空白字符,与[`\r\n\t\f`]同义
- `\S` `\s` 取反

`*`、`+`和`?`都是贪婪匹配操作符, 即最大长度匹配, 在其后加上`?`可以取消其贪婪匹配行为

正则表达式匹配对象通过 `groups` 函数获取子组

`re.match(pattern, string)`: 在字符串(string)的开头搜索正则表达式(pattern), 匹配则返回字符, 未能匹配则返回 `None`

`re.search(pattern, string)`: 在 `string` 中查找 `pattern` 的第一次出现, 匹配成功,则返回匹配对象, 否则返回 `None`

`group()`: `match` 或 `search` 匹配成功后,返回的匹配内容

`re.findall(pattern, string)`: 在 `string` 中查找 `pattern` 的所有(非重复)出现, 返回一个匹配对象的列表

`re.finditer(pattern, string)`: 等同于 `re.findall()`, 返回一个迭代器, 对于每个匹配,该迭代器返回一个匹配对象

`re.sub(pattern, repl, string)`: `string` 中所有匹配 `pattern` 的地方替换成新字符串(`repl`)

`re.split(pattern, string)`: 用 `pattern` 指定的分隔符把 `string` 分割为一个列表, 返回成功匹配的列表

`re.compile(pattern)`: 对正则表达式模式进行编译, 返回一个正则表达式对象, 可以提升效率

示例：

```
import re
m=re.match('f..','food')
print(m.group())
m=re.search('foo','seafood')
m=re.findall('foo','seafood is food')
for m in re.finditer('f..','seafood is food'):
    print(m.group())
m=re.sub('f..','123','fish is food')
m=re.split('\.','192.168.1.1')
patt=re.compile('f..')
m=patt.search('seafood')
```

socket 模块：

创建 TCP 服务器：

创建服务器套接字：s=socket.socket(socket_family, socket_type, protocol)

socket_family：套接字家族，默认为 AF_INET(基于 IPv4)。AF_UNIX(基于 Unix)或 AF_INET6(IPv6)

socket_type：套接字类型，默认为 SOCK_STREAM(面向连接)。SOCK_DGRAM(非连接)

protocol：与特定的地址家族相关的协议，默认为 0，表示自动选择一个合适的协议

设置套接字参数：s.setsockopt(level, option, value)

SOL_SOCKET：使用 socket 选项

SO_REUSEADDR：socket 关闭后，本地端用于该 socket 的端口号可以立刻被重用。1 表示 True

SO_KEEPALIVE：可以使 TCP 通信的信息包保持连续性。

绑定地址到套接字：s.bind(address)，如果不是指定本机地址，需要(host,port)格式指定

启动监听进程：s.listen(backlog)，backlog 可以不指定，如果指定进程数，最少是 0

等待客户连接：s.accept()，等待连接，返回 socket object, address info

与客户端通信

 读取 TCP 数据：recv()

 发送 TCP 数据：send()

关闭套接字:s.close()

示例：

```
#!/usr/local/bin/python3
import socket

host = ""
port = 12345
address = (host, port)
s = socket.socket()
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(address)
s.listen(1)
while True:
    cli_sock, cli_address = s.accept()
    print(cli_address)
    while True:
        data = cli_sock.recv(1024)
        if data.strip() == b'end':
```

```

        break
    print(data.decode('utf8'))
    data = input() + '\n'
    cli_sock.send(data.encode('utf8'))
    cli_sock.close()
s.close()

```

创建 TCP 客户端：

创建客户端套接字： `c=socket.socket()`
 初始化 TCP 连接服务器： `connect((hostname, port))`
 与服务器通：
 发送 TCP 数据： `c.send()`
 接收 TCP 的数据： `c.recv()`
 关闭客户端套接字： `c.close()`
 示例：

```

import socket

host = ''
port = 12345
addr = (host, port)
client = socket.socket()
client.connect(addr)
while True:
    data = input('> ') + '\n'
    client.send(data.encode('utf8'))
    if data.strip() == 'end':
        break
    data = client.recv(1024)
    print(data.decode('utf8'))
client.close()

```

创建 UDP 服务器：

创建服务器套接字： `s=socket.socket()`
 绑定服务器套接字： `s.bind()`
 接收 UDP 数据： `s.recvfrom()`
 发送 UDP 数据： `s.sendto()`
 关闭套接字： `s.close()`
 示例：

```

import socket
from time import strftime

host = ''
port = 12345
addr = (host, port)
server = socket.socket(type=socket.SOCK_DGRAM)
server.bind(addr)
while True:
    data, client_addr = server.recvfrom(1024)

```



```

clock = strftime('%H:%M:%S')
data = data.decode('utf8')
data = '[%s] %s' % (clock, data)
server.sendto(data.encode('utf8'))
server.close()

```

创建 UDP 客户端:

```

创建客户端套接字:c=socket.socket()
接收 UDP 数据: s.recvfrom()
发送 UDP 数据: s.sendto()
关闭客户端套接字:c.close()

```

示例:

```

import socket

host = ''
port = 12345
addr = (host, port)
client = socket.socket(type=socket.SOCK_DGRAM)
while True:
    data = input('> ')
    if data.strip() == 'end':
        break
    client.sendto(data.encode('utf8'))
    print(client.recvfrom(1024)[0].decode('utf8'))
client.close()

```

forking 多进程:

适用于计算密集型程序, 主要考虑 CPU 的效率

os.fork()函数实现 forking 功能, 创建子进程, 接下来的所有函数都会在父子进程中同时运行

forking 编程基本思路:由于父子进程资源相同, 注意避免资源冲突

```

pid = os.fork()
if pid:
    ...
else:
    ...

```

使用轮询解决子进程僵尸问题:

父进程通过 os.wait()来得到子进程是否终止的信息

在子进程终止后到父进程调用 wait()之间的这段时间, 子进程被称为 zombie(僵尸)进程

如果子进程还没有终止, 父进程先退出了, 那么子进程会持续工作。系统自动将子进程的父进程设置为 system 进程, system 将来负责清理僵尸进程

可以使用 waitpid(pid, options)来处理子进程:

pid 参数设置为-1, 表示与 wait()函数相同;options 参数设置为 0 表示挂起父进程, 直到子程序退出, 设置为 1 表示不挂起父进程

返回值为(pid, status), 如果子进程尚未结束则返回 0, 否则返回子进程的 PID; 返回的 status 无用

示例:多进程扫描存活主机

```

import subprocess
import os

```

```
def ping(host):
    result = subprocess.call(
        'ping -c 2 %s &> /dev/null' % host, shell=True
    )
    if result == 0:
        print('%s is up' % host)
    else:
        print('%s is down' % host)

if __name__ == '__main__':
    ip_pool = ['192.168.4.%s' % i for i in range(1, 255)]
    for ip in ip_pool:
        pid = os.fork()
        if not pid:
            ping(ip)
            exit() #执行完毕后一定要退出子进程,不然子进程也会进入 for 循环
```

示例:僵尸进程处理

```
import os
import time

def fork(pt, ct):
    pid = os.fork()
    if pid:
        time.sleep(pt)
        print(os.waitpid(-1, 1))
    else:
        time.sleep(ct)

fork(3, 5)
fork(10, 5)
```

thread 和 threading 多线程:

适用于 IO 密集型程序, 需要等待数据的提供或返回

thread 模块提供了基本的线程和锁的支持, 更加贴近线程的底层

threading 提供了更高级别、功能更强的线程管理功能

注意:由于线程执行完就停止了, 不会产生僵尸进程

1.传递函数给 Thread 类

传递函数给 threading 模块的 Thread 类

Thread 对象使用 start()方法开始线程的执行, 使用 join()方法挂起程序, 直到线程结束

示例:多线程扫描存活主机

```
import subprocess
import threading
```

```
def ping(host):
    result = subprocess.call(
        'ping -c2 %s &> /dev/null' % host, shell=True
    )
    if result == 0:
        print('%s is up' % host)
    else:
        print('%s is down' % host)
```

```
if __name__ == '__main__':
    ip_pool = ['192.168.4.%s' % i for i in range(1, 255)]
    for ip in ip_pool:
        t = threading.Thread(target=ping, args=(ip,))
        t.start()
```

2. 传递可调用类给 Thread 类

类对象里可以使用类的强大的功能，可以保存更多的信息，更加灵活
示例：

```
import threading
import subprocess
```

```
class Ping:
    def __init__(self, host):
        self.host = host

    def __call__(self):
        result = subprocess.call(
            'ping -c2 %s &> /dev/null' % self.host, shell=True
        )
        if result == 0:
            print('%s is up' % self.host)
        else:
            print('%s is down' % self.host)
```

```
if __name__ == '__main__':
    ip_pool = ['192.168.4.%s' % i for i in range(1, 255)]
    for ip in ip_pool:
        t = threading.Thread(target=Ping(ip))
        t.start()
```

pypi(python package index):python 软件仓库

官方网站:<https://pypi.org/>

该网站下的包可以使用 pip 安装：依赖包为 gcc

下载本地安装:[root@room9pc01 ~]# pip3 install PyMySQL-0.9.2.tar.gz

在线安装:[root@room9pc01 ~]# pip3 install pymysql

配置国内镜像站点:

```
[root@room9pc01 ~]# mkdir .pip
[root@room9pc01 ~]# vim .pip/pip.conf

[global]
index-url=https://mirrors.aliyun.com/pypi/simple/
[install]
trusted-host=mirrors.aliyun.com
```

关系型数据库六种范式: 后一个范式必须满足前一个范式的所有条件

第一范式(1NF): 原子性, 数据库表的每一列都是不可分割的原子数据项, 而不能是集合, 数组, 记录等非原子数据项

第二范式(2NF): 要求数据库表中的每个实例或记录必须可以被唯一地区分。选取一个能区分每个实体的属性或属性组, 作为实体的唯一标识。

第三范式(3NF): 任何非主属性不依赖于其它非主属性, 消除传递依赖

巴斯-科德范式(BCNF): 任何非主属性不能对主键子集依赖, 消除对主码子集的依赖

第四范式(4NF)

第五范式(5NF, 又称完美范式)

PyMySQL:

安装: pip3 install pymysql

连接数据库: pymysql.connect(host='127.0.0.1', port=3306, user='root', passwd='', db='tedu', charset='utf8')

游标: 一条 sql 取出对应 n 条结果资源的接口、句柄就是游标, 沿着游标可以一次取出一行, cursor()

执行 mysql 命令: 需要 commit()执行

单条: execute()

多条: executemany()

查询数据:

取出一条记录: fetchone()

取出多条记录: fetchmany()

取出全部记录: fetchall()

移动游标:

从开头为起点移动 value: scroll(value, mode='absolute')

以当前位置为起点移动 value: scroll(value)

示例:

```
import pymysql

conn = pymysql.connect(host='127.0.0.1', port=3306, user='root', passwd='', db='tedu', charset='utf8')
cursor = conn.cursor()
insert_dep1 = 'insert into departments VALUES (%s,%s)'
delete1 = 'delete from departments WHERE dep_name=%s'
update_dep1 = 'update departments set dep_name=%s WHERE dep_name=%s'
cursor.execute(insert_dep1, ('1', '人事部'))
cursor.executemany(insert_dep1, [('2', '财务部'), ('3', '业务部'), ('4', '采购部'), ('5', '生产部')])
cursor.execute(update_dep1, ('人力资源部', '人事部'))
cursor.execute(delete1, ('人力资源部',))
conn.commit()

find='select * from departments'
```

```

cursor.execute(find)
result = cursor.fetchone()
cursor.scroll(1)
result2 = cursor.fetchmany(2)
cursor.scroll(0, mode='absolute')
result3 = cursor.fetchall()
print(result, result2, result3)
cursor.close()
conn.close()

```

SQLAlchemy:

安装: `pip3 install sqlalchemy`

简介:

提供 SQL 工具包及对象关系映射(ORM)工具, 使用 MIT 许可证发行
提供能兼容众多数据库的企业级持久性模型

ORM 模型: 对象关系映射模型

数据库表是一个二维表, 包含多行多列。Python 可以用一个 list 表示多行, list 的每一个元素是 tuple, 表示一行记录

用 tuple 表示一行很难看出表的结构。如果把一个 tuple 用 class 实例来表示, 就可以更容易地看出表的结构

连接 mysql: `create_engine('mysql+pymysql://用户名:密码@主机/库名', encoding=编码, echo=False)`

`echo=True` 表示将日志输出到终端屏幕

声明映射: `declarative_base()`

当使用 ORM 的时候, 配置过程从描述数据库表开始

通过自定义类映射相应的表

通过声明系统定义基类

通过声明系统实现类映射

创建映射类:

`class 类名(声明):`

`__tablename__ = 表名`

`字段名 1 = Column(Integer, primary_key=True)`

`字段名 2 = Column(String(20), unique=True)`

Integer 表示 int 类型

String 表示字符串类型

创建架构: 表的信息将被写入到表的元数据(metadata)

`metadata.create_all()`

创建实例: 将要添加的记录存入缓存, 不会真正在表中添加记录

创建会话类: ORM 访问数据库的句柄被称作 Session

添加新对象: 实例初次使用, 它将从 Engine 维护的连接池中获得一个连接; 可以通过列表批量添加; 需

要 commit 执行

外键约束: `ForeignKey()`, 使用时, 确认调用的外键已经事先创建

示例: `sqlalchemy_test`

```

from sqlalchemy import create_engine, Column, Integer, String, Date, ForeignKey

```

```

from sqlalchemy.ext.declarative import declarative_base

```

```

from sqlalchemy.orm import sessionmaker

```

```

engine = create_engine('mysql+pymysql://root:@localhost/tarena', encoding='utf8', echo=True)

```

```
Base = declarative_base()
Session = sessionmaker(bind=engine)
```

```
class Departments(Base):
    __tablename__ = 'departments'
    dep_id = Column(Integer, primary_key=True)
    dep_name = Column(String(20), unique=True)

    def __str__(self):
        return '%s:%s' % (self.dep_id, self.dep_name)
```

```
class Employees(Base):
    __tablename__ = 'employees'
    id = Column(Integer, primary_key=True)
    name = Column(String(20), nullable=False)
    gender = Column(String(6))
    birthday = Column(Date)
    email = Column(String(50))
    dep_id = Column(Integer, ForeignKey('departments.dep_id'))

    def __str__(self):
        return '%s:%s' % (self.id, self.name)
```

```
class Salary(Base):
    __tablename__ = 'salary'
    auto_id = Column(Integer, primary_key=True)
    date = Column(Date)
    id = Column(Integer, ForeignKey('employees.id'))
    basic = Column(Integer)
    awards = Column(Integer)
```

```
if __name__ == '__main__':
    Base.metadata.create_all(engine)
    hr = Departments(dep_name='人事部')
    op = Departments(dep_id=2, dep_name='运维部')
    dev = Departments(dep_id=3, dep_name='开发部')
    qa = Departments(dep_id=4, dep_name='测试部')
    bob = Employees(id=1, name='bob', gender='male', birthday='1992-05-15', email='bob@tedu.cn',
dep_id=2)
    alice = Employees(id=2, name='alice', gender='female', birthday='1999-12-28', email='alice@tedu.cn',
dep_id=1)
    tom = Employees(id=3, name='tom', gender='male', birthday='1989-01-01', email='tom@tedu.cn',
dep_id=3)
```

```

session = Session()
session.add(hr)
session.add_all([op, dev, qa])
session.commit()
session.add_all([bob, alice, tom])
session.commit()
session.close()

```

SQLAlchemy 的查询：

基本查询：通过 `query()` 函数创建查询语句

修改显示字段名： `label()`

排序： `order_by()`，按指定字段排序

提取部分数据： `query()[:]`，通过切片提取部分数据\

结果过滤： `filter()`，可以叠加使用

过滤操作符：

相等： `==`

不等： `!=`

模糊查询： `like()`

包含： `in_()`

不包含： `~...in_()`

空字段： `is_(None)`

非空字段： `isnot(None)`

多重条件：

同时满足： `and_`

满足任一： `or_`

查询对象返回值：

返回列表： `all()`

返回结果中的第一条记录： `first()`

取出只有一个结果的记录，如果不是一个结果则报错： `one()`

返回第一列的值： `scalar()`

统计： `count()`

多表连接查询： `join()`

`join()` 中要写后加入的表名

示例：

```

from sqlalchemy_test import Departments, Session, Salary, Employees
from sqlalchemy import and_, or_

```

```

session = Session()

```

```

qselect1 = session.query(Departments).order_by(Departments.dep_id)
print(qselect1)
for dep in qselect1:
    print(dep)

```

```

qselect2 = session.query(Departments.dep_id, Departments.dep_name).order_by(Departments.dep_id)
print(qselect2)
for id, name in qselect2:
    print(id, name, sep=':')

```

```
qcut = session.query(Departments)[1:3]
print(qcut)
for dep in qcut:
    print(dep.dep_name)

qfilter = session.query(Departments.dep_name).filter(Departments.dep_id == 2)
print(qfilter)
for dep in qfilter:
    print(dep.dep_name)

qselect3 = session.query(Salary.date, Salary.id, Salary.basic + Salary.awards)
print(qselect3)
for date, id, money in qselect3:
    print(date, id, money, sep=':')

qfind = session.query(Departments.dep_id).filter(Departments.dep_name.in_(['运维部', '开发部']))
print(qfind)
for id in qfind:
    print(id)

qnotfind = session.query(Departments.dep_id).filter(~Departments.dep_name.in_(['运维部', '开发部']))
print(qnotfind)
for id in qnotfind:
    print(id)

qselect4 = session.query(Employees).filter(and_(Employees.gender == 'male', Employees.dep_id == 2))
print(qselect4)
for employ in qselect4:
    print(employ.name)

qselect5 = session.query(Employees).filter(or_(Employees.gender == 'female', Employees.dep_id == 1))
print(qselect5)
for employ in qselect5:
    print(employ.name)

print(qselect2.all())
print(qselect2.first())
print(qselect5.one())
print(qselect5.scalar())

qcount = session.query(Departments).count()
print(qcount)

qjoin = session.query(Employees.name, Departments.dep_name).join(Departments, Employees.dep_id
== Departments.dep_id)
print(qjoin.all())
```


SQLAlchemy 的更新:

通过会话的 `update()` 方法更新

通过会话的字段赋值更新

SQLAlchemy 的删除: `delete()`

示例:

```
session = Session()
up2 = session.query(Departments).get(1)
up2.dep_name = '人事部'
session.commit()
qdel = session.query(Employees).get(3)
session.delete(qdel)
session.commit()
session.close()
```

urllib:

Python2 版本中, 有 `urllib` 和 `urllib2` 两个库可以用来实现 `request` 的发送; 在 Python3 中, 统一为 `urllib` 模块:

`urllib.request` 可以用来发送 `request` 和获取 `request` 的结果

`urllib.error` 包含了 `urllib.request` 产生的异常

`urllib.parse` 用来解析和处理 URL

`urllib.robotparse` 用来解析页面的 `robots.txt` 文件

爬取网页:

导入 `urllib.request` 模块

使用 `urllib.request.urlopen` 打开并爬取一个网页

读取内容:

`read()`: 读取文件的全部内容, 赋给一个字符串变量

`readlines()`: 读取文件的全部内容, 赋值给一个列表变量

`readline()` 读取文件的一行内容

打开网络资源并下载: 结合 `urlopen` 和 `read()`

伪装浏览器:

`header={'User-Agent': '...'}`

`request.Request(url, headers=header)`

数据编码:

URL 标准中只会允许数字、字母和部分符号, 其他字符需要编码

异常处理模块:

`urllib.error` 模块, 当访问的页面不存在或拒绝访问时, 程序提示异常

示例:

```
from urllib import request, error
import os
import re

html = request.urlopen('http://www.baidu.com/')
print(html.readline())
print(html.read(4096))
print(html.readlines())
```

```

def download(url, fname):
    header = {'User-Agent': '"Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0"'}
    r = request.Request(url, headers=header)
    try:
        html = request.urlopen(r)
    except error.HTTPError as e:
        print(e)
    else:
        with open(fname, 'wb') as f:
            while True:
                data = html.read(4096)
                if not data:
                    break
                f.write(data)

```

```

def search_url(fname, patt):
    patt_list = []
    cpatt = re.compile(patt)
    with open(fname) as f:
        for line in f:
            m = cpatt.search(line)
            if m:
                item = m.group()
                patt_list.append(item)
    return patt_list

```

```

if __name__ == '__main__':
    download('https://www.baidu.com/', '/tmp/baidu.html')
    download(

```

```

'https://timgsa.baidu.com/timg?image&quality=80&size=b9999_10000&sec=1535435104365&di=479f195cd2
1bfc3ef8becca251976d43&imgtype=jpg&src=http%3A%2F%2Fimg4.imgtn.bdimg.com%2Fit%2Fu%3D2326766
919%2C2993649423%26fm%3D214%26gp%3D0.jpg',
    '/tmp/girl.jpg')
    r = request.quote('你好')
    print(r)
    ur = request.unquote(r)
    print(ur)
    img_dirs = '/tmp/imgs'
    if not os.path.exists(img_dirs):
        os.mkdir(img_dirs)
    photo = '/tmp/photo'
    download('http://www.tmooc.cn/', photo)
    img_patt = 'http://[\\w./]+\\.\\.(jpg|jpeg|gif|png)'
    img_list = search_url(photo, img_patt)

```

```

for url in img_list:
    fname = url.split('/')[-1]
    fname = os.path.join(img_dirs, fname)
    download(url, fname)

```

paramiko: 远程连接模块

创建 ssh 的实例: paramiko.SSHClient()

设置自动添加主机密钥: set_missing_host_key_policy(paramiko.AutoAddPolicy())

连接 ssh 服务器: connect(host, username=, password=, port=)

执行指定命令: exec_command()

示例:

```

import paramiko

ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect('176.19.5.195', username='root', password='123456', port=2222)
ssh.exec_command('mkdir /tmp/demo')
result = ssh.exec_command('id root')
print(result[1].read().decode('utf8'))
print(result[2].read())
ssh.close()

```

smtpplib: 发送邮件模块

创建 SMTP 对象: SMTP(host="", port=0), port 不指定即为默认 25 端口

设置邮件:

邮件内容: MIMEText(_text=, _subtype='plain', _charset=None)

邮件头部信息: Header(s=None, charset=None)

From:发件人

To:收件人

Subject:主题

发送邮件: sendmail(from_addr, to_addrs, msg)

使用第三方的邮件服务器:

连接第三方 SMTP 服务器: connect(host='localhost', port=0)

输入第三方 SMTP 服务器账号密码: login(user, password)

示例: 使用网易邮件代理发送邮件

```

from email.header import Header
from email.mime.text import MIMEText
from smtpplib import SMTP

sender = 'd1452823030@163.com'
receiver = 'd1452823030@163.com'
message = MIMEText(_text='这是内容', _charset='utf8')
message['From'] = Header(sender, 'utf8')
message['To'] = Header(receiver, 'utf8')
message['Subject'] = Header('测试邮件', 'utf8')
smtp = SMTP()
smtp.connect('smtp.163.com')

```

```
smtp.login('d1452823030', 'zxcv123456')
smtp.sendmail(sender, receiver, message.as_string())
```

JSON 模块:

编码: `json.dumps()`

解码: `json.loads()`

示例: 查询天气

```
import json
from urllib import request

adict = {'name': '张三', 'age': 25}
print(json.dumps(adict))
print(json.loads(json.dumps(adict)))
weather = request.urlopen('http://www.weather.com.cn/data/sk/101210101.html')
info = request.urlopen('http://www.weather.com.cn/data/cityinfo/101210101.html')
zs = request.urlopen('http://www.weather.com.cn/data/zs/101210101.html')
weather_data = weather.read()
print(json.loads(weather_data))
print('#' * 100)
info_data = info.read()
print(json.loads(info_data))
print('#' * 100)
zs_data = zs.read()
print(json.loads(zs_data))
```

requests 模块: 网页连接

发送一个 GET 请求: `requests.get(url, params=None, headers, cookies)`

`params` 可以处理参数键值对

`headers` 设定请求头

POST 请求传递数据: `requests.post(url, data=None, json=None, params, file, headers, cookies)`

`data` 可以发送请求数据

`file` 可以传文件

响应格式:

`text`: 按文本读取服务器响应

`encoding`: 输出编码, 可以指定

`content`: 以字节方式读取服务器响应, 如图片, 声音

`json`: 以 json 方式读取服务器响应

响应状态码:

`status_code`: 查看响应状态码

`requests.codes.状态`: 状态码查询对象

`ok`

`not_acceptable`

`not_allowed`

`not_extended`

`not_found`

`not_implemented`

`not_modified`

headers: 查看响应头

cookies 模块: 查看 cookies

Set-Cookie:key=value;expires=date;domain=domain;path=path;secure

key/value: 在服务端可跟踪、可识别的用户信息

expires: Cookie 结束日期, 如果没指定会在用户退出浏览器时过期

domain: 告诉浏览器这个 Cookie 可以被发送到哪个域名, 如果没指定, 默认为产生 Cookie 的服务
器主机名, 浏览器会存储很多不同网站的 Cookie, 浏览器会根据 domain 的值将 Cookie 发送到对应的域
名下

path: 指定 Cookie 对哪些请求路径生效, 如果没指定, 默认为产生 Cookie 的 URL 路径

secure: 在使用 SSL 安全连接时才发送 Cookie, 若没设置 secure, 则没限制

示例:

import requests

```
payload = {'wd': 'centos7'}
r1 = requests.get('https://www.baidu.com/s', params=payload)
header = {'User-Agent': '"Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0"'}
r2 = requests.get(url='https://www.baidu.com/', headers=header)
# data = {'username': 'lili', 'password': '123456'}
# r3 = requests.post(url='127.0.0.1', data=data)
r4 = requests.get('https://www.baidu.com')
print(r4.text)
print(r4.encoding)
r4.encoding = 'utf8'
print(r4.text)
print(r4.cookies)
r5 = requests.get('https://pic.feizl.com/upload/allimg/170808/813cooff2o0b24.jpg')
with open('/tmp/a.jpg', 'wb') as f:
    f.write(r5.content)
r6 = requests.get('http://www.weather.com.cn/data/sk/101210101.html')
r6.encoding = 'utf8'
print(r6.json())
print(r6.status_code)
if r6.status_code == requests.codes.ok:
    print('ok')
print(r6.headers)
```

Zabbix API:

允许以编程方式检索和修改 Zabbix 的配置, 并提供对历史数据的访问

基于 Web 的 API, 作为 Web 前端的一部分提供。使用 JSON-RPC2.0 协议:

API 包含一组独立的方法:

客户端和 API 之间的请求和响应使用 JSON 格式进行编码。

需要向 api_jsonrpc.php 位于前端目录中的文件发送 HTTP POST 请求

请求的 Content-Type 头部必须设置为以下值之一:

application/json-rpc

application/json

application/jsonrequest

JSON 请求对象的属性:

jsonrpc: API 使用的 JSON-RPC 协议的版本, 版本 2.0

method: 调用的 API 方法

params: 传递给 API 方法的参数

id: 请求的任意标识符, 随便填一个数字

auth: 用户认证令牌; 如果没有, 可以设置为 None

user.login 获取令牌: 登录并获取身份验证令牌

API 返回的响应包含用户身份验证令牌:

jsonrpc: JSON-RPC 协议的版本

result: 返回的数据

id: 相应请求的标识符。

示例:

```
import requests
import json

url = 'http://192.168.1.53/api_jsonrpc.php'
headers = {'Content-Type': 'application/json-rpc'}
data = {
    "jsonrpc": "2.0",
    "method": "apiinfo.version",
    "params": {},
    "id": 1,
    "auth": None
}

r = requests.post(url=url, headers=headers, data=json.dumps(data))
print(r.json())
data1={
    "jsonrpc": "2.0",
    "method": "user.login",
    "params": {
        "user": "Admin",
        "password": "zabbix"
    },
    "id": 1
}
r1=requests.post(url=url,headers=headers,data=json.dumps(data1))
print(r1.json())
```

输出:

```
{'jsonrpc': '2.0', 'result': '3.4.4', 'id': 1}
{'jsonrpc': '2.0', 'result': '9ba4de5941d8dfa149e3ec896e7a41be', 'id': 1}
```

操作 Zabbix 主机:

更多参考: <https://www.zabbix.com/documentation/3.4/zh/manual/api/reference>

host.get: 检索所有已配置主机的 ID、主机名和接口。auth 属性设置为获得的身份验证令牌

hostgroup.get: 获取主机组

host.create: 创建主机

host: 主机名称

interfaces: 接口

type: 类型。1: agent; 2: SNMP; 3: IPMI; 4: JMX

main: 该接口是否是主机上的默认接口。1 表示 True

useip: 是否通过 IP 进行连接。1 表示 True

示例: 续上

```
data2 = {
    "jsonrpc": "2.0",
    "method": "host.get",
    "params": {
        "output": ["hostid"],
        "selectGroups": "extend",
        "filter": {
            "host": [
                "Zabbix server"
            ]
        }
    },
    "auth": "9ba4de5941d8dfa149e3ec896e7a41be",
    "id": 2
}
r2 = requests.post(url=url, headers=headers, data=json.dumps(data2))
print(r2.json())

data3 = {
    "jsonrpc": "2.0",
    "method": "hostgroup.get",
    "params": {
        "output": "extend",
        "filter": {
            "name": [
                "Zabbix servers",
                "Linux servers",
                "Templates"
            ]
        }
    },
    "auth": "9ba4de5941d8dfa149e3ec896e7a41be",
    "id": 3
}
r3 = requests.post(url=url, headers=headers, data=json.dumps(data3))
for item in r3.json()['result']:
    print(item['groupid'], item['name'], sep=':')

data4 = {
    "jsonrpc": "2.0",
    "method": "template.get",
    "params": {
        "output": "extend",
    },
}
```

```

        "auth": "9ba4de5941d8dfa149e3ec896e7a41be",
        "id": 4
    }
    r4 = requests.post(url=url, headers=headers, data=json.dumps(data4))
    for item in r4.json()['result']:
        print(item['templateid'], item['host'], sep=':')

data5 = {
    "jsonrpc": "2.0",
    "method": "host.create",
    "params": {
        "host": "My server",
        "interfaces": [
            {
                "type": 1,
                "main": 1,
                "useip": 1,
                "ip": "192.168.1.53",
                "dns": "",
                "port": "10050"
            }
        ],
        "groups": [
            {
                "groupid": "1"
            }
        ],
        "templates": [
            {
                "templateid": "10001"
            }
        ],
        "inventory_mode": 0,
    },
    "auth": "9ba4de5941d8dfa149e3ec896e7a41be",
    "id": 5
}
r5 = requests.post(url=url, headers=headers, data=json.dumps(data5))

```

命名元组：在普通元组的基础上，增加了可以根据名称引用元组中的项
collections 模块的 namedtuple 函数提供了创建自定义的元组数据类型
示例：

```

from collections import namedtuple

p1 = (10, 20)
print(p1[0])
test = namedtuple('test', ['x', 'y', 'z'])

```



```

p2 = test(1, 2, 3)
print(p2[0])
print(p2.x)

```

ansible 模块:

具体可以查看: https://docs.ansible.com/ansible/latest/dev_guide/developing_api.html

模块函数必须按此加载:

from **ansible.parsing.dataloader** import **DataLoader**: 加载解析 yaml 文件或 JSON 内容, 并且支持 vault 的解密

from **ansible.vars.manager** import **VariableManager**: 管理变量的类, 包括主机、组、扩展等变量

from **ansible.inventory.manager** import **InventoryManager**: 创建主机清单, 主机清单的源采用配置文件或是逗号分开主机名字符串

from **ansible.playbook.play** import **Play**: 创建 play 对象, 能够通过 play_source 提供的信息自动创建任务对象

from **ansible.executor.task_queue_manager** import **TaskQueueManager**: 处理进程池中的多进程。队列管理器负责加载 play 策略插件, 以便在选定的主机上执行任务

from **ansible.plugins.callback** import **CallbackBase**: 回调函数类, 可以继承

import **ansible.constants**: ansible 的预定义变量

示例:

```

import shutil
from collections import namedtuple
from ansible.parsing.dataloader import DataLoader
from ansible.vars.manager import VariableManager
from ansible.inventory.manager import InventoryManager
from ansible.playbook.play import Play
from ansible.executor.task_queue_manager import TaskQueueManager
import ansible.constants as C

Options = namedtuple('Options',
                    ['connection', 'module_path', 'forks', 'become', 'become_method',
                    'become_user', 'check', 'diff'])
options = Options(connection='smart', module_path=['/to/mymodules'], forks=10, become=None,
become_method=None,
                    become_user=None, check=False, diff=False)

loader = DataLoader()
passwords = dict(vault_pass='secret')# 用于存储加密密码、远程连接密码等

inventory = InventoryManager(loader=loader, sources='localhost,')

variable_manager = VariableManager(loader=loader, inventory=inventory)

play_source = dict(
    name="Ansible Play",# Play 名称
    hosts='all',# 在哪些主机上执行命令
    gather_facts='no',# 是否收集主机信息, no 表示否
    # 执行的命令

```

```

tasks=[
    dict(action=dict(module='shell', args='ls'), register='shell_out'),
    dict(action=dict(module='debug', args=dict(msg='{{shell_out.stdout}}'))))
]
)

play = Play().load(play_source, variable_manager=variable_manager, loader=loader)

tqm = None
try:
    tqm = TaskQueueManager(
        inventory=inventory,
        variable_manager=variable_manager,
        loader=loader,
        options=options,
        passwords=passwords,
    )
    result = tqm.run(play)
finally:
    if tqm is not None:
        tqm.cleanup()

shutil.rmtree(C.DEFAULT_LOCAL_TMP, True)

```

Django: 开放源代码的 Web 应用框架

核心组件:

- 用于创建模型的对象关系映射
- 为最终用户设计的完美管理界面
- 一流的 URL 设计
- 设计者友好的模板语言
- 缓存系统

Model-View-Controller(MVC):

把数据存取逻辑、业务逻辑和表现逻辑组合在一起

Model 代表数据存取层

View 代表的系统中选择显示什么和怎么显示的部分

Controller 指系统中根据用户输入并视需要访问模型，以决定使用哪个视图的那部分

MTV:

模型(**Model**): 负责业务对象和数据库的关系映射(**ORM**)

模板(**Template**): 负责如何把页面展示给用户(**html**)

视图(**View**): 负责业务逻辑，并在适当时候调用 **Model** 和 **Template**

URL 分发器: 将 **URL** 的页面请求分发给不同的 **View** 处理，**View** 再调用相应的 **Model** 和 **Template**

python 虚拟环境:

- 1.虚拟环境所在的目录: `[root@room9pc01 test]# mkdir -v py3env`
- 2.创建虚拟环境: `[root@room9pc01 test]# python3 -m venv py3env/dj_env`
- 3.激活环境: `[root@room9pc01 test]# source py3env/dj_env/bin/activate`
- 4.创建项目: `(dj_env) [root@room9pc01 test]# django-admin startproject mysite_test`

目录子文件说明:

mysite_test: 项目名称

manage.py: 命令行工具

mysite: 项目的真正的 Python 包, 导入任何东西时需要使用的 Python 包的名字

__init__.py: 空文件, 告诉 Python 这个目录应该被看做一个 Python 包

settings.py: 项目的设置/配置

urls.py: 项目的 URL 声明

wsgi.py: 与 WSGI 兼容的 Web 服务器入口

5.在 pycharm 中设置虚拟环境: 在 setting 中添加/root/test/py3env/dj_env/bin/python3

6.创建数据库: MariaDB [(none)]> create database dj_test default charset utf8;

7.修改 settings.py 中的配置, 使其支持中文:

```
ALLOWED_HOSTS = '*'
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'dj_test',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': '127.0.0.1',
        'PORT': '3306',
    }
}
```

```
LANGUAGE_CODE = 'zh-Hans'
```

```
TIME_ZONE = 'Asia/Shanghai'
```

8.在__init__.py 中加载 mysql 模块:

```
import pymysql
```

```
pymysql.install_as_MySQLdb()
```

9.生成数据库: (dj_env) [root@room9pc01 test]# python mysite_test/manage.py migrate

10.创建后台管理员: (dj_env) [root@room9pc01 test]# python mysite_test/manage.py createsuperuser

11.启动服务: (dj_env) [root@room9pc01 test]# python mysite_test/manage.py runserver 0:80

12.后台管理入口: <http://127.0.0.1/admin/>

Django 页面制作:

1.首页测试: urls.py 中 url() 的 name 字段指定首页

```
view_test.py:
```

```
from django.shortcuts import HttpResponseRedirect
```

```
def index(request):
```

```
    return HttpResponseRedirect('<h1>hello world</h1>')
```

```
urls.py:
```

```
from . import view_test
```

```
urlpatterns = [
    url(r'^admin/', admin.site.urls),
    url(r'^$', view=view_test.index, name='index')
]
```

2.创建应用: (dj_env) [root@room9pc01 mysite_test]# python manage.py startapp poll_test

3.修改 settings.py, 在项目中注册应用:

```
INSTALLED_APPS = [
    ...
    'poll_test',
]
```

4.修改 mysite_test/urls.py 设置路由: 访问 127.0.0.1/poll_test/时, 转交给 poll_test 下的 urls.py:

```
from . import view_test

urlpatterns = [
    ...
    url(r'^poll_test/',include('poll_test.urls'))
]
```

5.创建应用映射文件 poll_test/urls.py:

```
from django.conf.urls import url
from . import views

urlpatterns = [
    url(r'^$', views.polls_index, name='polls_index'),
]
```

6.创建模板目录和测试网页

```
(dj_env) [root@room9pc01 test]# mkdir -p mysite_test/poll_test/templates/polls
创建一个名为 poll_test_index.html 的网页, 内容随意
```

7.修改 poll_test/views.py, 指向测试网页

```
from django.shortcuts import render

def polls_index(request):
    return render(request, 'polls/poll_test_index.html')
```

Django 模型建立:

1.修改 poll_test/models.py, 创建 ORM 映射类:

```
from django.db import models

class Question(models.Model):
    question_text = models.CharField(max_length=200)
    publish_date = models.DateTimeField()
    def __str__(self):
        return self.question_text

class Choice(models.Model):
    choice_text = models.CharField(max_length=200)
```

```
votes = models.IntegerField(default=0)
question = models.ForeignKey(Question, models.CASCADE)
```

2.运行，在数据库中生成表：

```
(dj_env) [root@room9pc01 test]# python mysite_test/manage.py makemigrations
(dj_env) [root@room9pc01 test]# python mysite_test/manage.py migrate
```

3.修改 poll_test/admin.py，把模型注册到后台管理界面：

```
from django.contrib import admin
from .models import Question, Choice

admin.site.register(Question)
admin.site.register(Choice)
```

python shell:

1.加载：(dj_env) [root@room9pc01 test]# python mysite_test/manage.py shell

2.导入模型：>>> from poll_test.models import Question,Choice

3.查看所有的 Question：>>> Question.objects.all()

无内容：<QuerySet []>

有内容：<QuerySet [<Question: t1>, <Question: t2>]>

4.取出第一个 Question 并访问：

```
>>> q=Question.objects.all()[0]
>>> q.id
>>> q.question_text
>>> q.publish_date
```

5.创建 Question：

```
>>> from django.utils import timezone
>>> q1=Question(question_text="第一个问题",publish_date=timezone.now())
>>> q1.save()
```

6.修改 Question：

```
>>> q=Question.objects.all()[0]
>>> q.question_text="q1"
>>> q.save()
```

7.models.py 下添加可用于查询的自定义方法后，需要重新载入 shell：

```
class Question(models.Model):
    ...
    def recently(self):
        return self.publish_date >= timezone.now() - timedelta(days=1)
```

8.通过实例查询：由于该方法是一个判断，所以返回 True 或 False

```
>>> q=Question.objects.all()[0]
>>> q.recently()
```

9.通过主键查询：返回实例，如果不存在则报错

```
>>> Question.objects.get(id=1)
>>> Question.objects.get(question_text="q1")
```

10.通过灵活的双下划线实现属性查找：

```
>>> Question.objects.filter(question_text__startswith="q")
>>> Question.objects.filter(publish_date__year=2018)
```

11.创建 choice 条目：不需要 save 可以直接写入数据库

由于 Question 是 Choice 的外键，一个 question 有多个 choice，django 可以通过 Question 对象可以

反向得到 Choice 对象集

```
>>> q=Question.objects.get(id=2)
>>> q.choice_set.create(choice_text="问题 1 ",votes=0)
>>> c=Choice.objects.create(choice_text="问题 2 ",votes=0,question_id=2)
>>> q.choice_set.all()
```

12.创建详细页面：通过正则表达式匹配数字，将这些数字作为参数传递给视图函数

修改 urls.py

```
urlpatterns = [
    ...
    url(r'^(?P<q_id>\d+)/$', views.detail, name='detail'),
]
```

13.创建 detail 模板：

在 views.py 中增加 detail 标签：

```
def detail(request, q_id):
    return render(request=request, template_name='polls/detail.html', context={'question_id': q_id})
```

创建/poll_test/templates/pollsdetail.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>问题详情</title>
    <link rel="stylesheet" href="https://cdn.bootcss.com/bootstrap/3.3.7/css/bootstrap.min.css"
```

```
integrity="sha384-BVYiSiFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u">
```

```
</head>
<body>
<div class="container">
    <p>正在查看第{{ question_id }}个问题</p>
</div>
</body>
</html>
```

14.创建 result 模板：同 detail

urls.py:

```
urlpatterns = [
    ...
    url(r'^(?P<q_id>\d+)/result/$', views.result, name='result'),
]
```

views.py:

```
def result(request, q_id):
    return render(request=request, template_name='polls/result.html', context={'question_id': q_id})
```

result.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>结果</title>
    <link rel="stylesheet" href="https://cdn.bootcss.com/bootstrap/3.3.7/css/bootstrap.min.css"
```

```

integrity="sha384-BVYiSiFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u">
</head>
<body>
<div class="container">
    这是第{{ question_id }}个问题的结果
</div>
</body>
</html>

```

15.修改 detail.html，加入表单，选择某一项之后，将数据使用 post 发送给 poll_test/1/vote/处理

16.修改 views.py 中的 vote 函数，接收表单传过来的 choice_id，再找到数据库中 choice_id 对应的 choice

17.修改 views.py 中的 result 函数，增加 get_object_or_404 功能，在无法找到页面时不报错。修改 result.html，将结果以列表方式展示

注：最终程序参考 https://github.com/a1441668968/test/tree/master/mysite_test

入口映射：

基本形式：

```

urls.py:
urlpatterns = [
    url(r'正则表达式', view=views.py 中的函数, kwargs=附加参数, name=别名, ),
    ...
]

```

正则表达式：采用 match 操作

r'^\$', 匹配空 URL，通常用来设定应用的根，即默认入口

r'^abc': 匹配以 abc 开头的任意 URL

r'^abc/\$': 匹配 abc 且后面无信息的 URL，如/abc, /abc/

参数：URL 本身可以用于传递信息，该信息不符合 HTTP 协议，是非标准化信息，需要依赖框架来解释。

?P<变量名>：获取变量，传递给 views.py 中对应的函数的形参

多级规划：通过建立应用级的 urls 映射，构建层级访问控制

1.添加 app：在项目的 setting 中，在 INSTALLED_APPS 中添加一个应用名称

2.配置 URLCONF：url(r'^student/',include('student.urls'))。注意：这里不能加\$，因为这只是个一级目录，是完全目录的一部分

3.在应用目录下的 urls 中添加：url(r'^add/(?P<name>.+)/(?P<age>.+)/\$',add), ...

信息提取：

GET: request.GET.get[]

默认方法

POST: request.POST.get[]

Django 对 form 表单的 POST 请求做了提交限制

Cookie：保存在浏览器端的键值对，与域名绑定，有存活时间，用户可以通过浏览器设置里的功能手动清除，或者禁止启用。

存在于 Request 的请求 head 中，供读取。也存在于 response 的返回 head 中，供写入

Session：服务器端的存储对象，可以存储任意类型的数据，但其索引保存在浏览器端的 cookie 中，但不一定非要依赖 cookie，其主体信息保存在服务器上。

基于 session 的认证机制：通过一个 session 对象，受保护页面检测是否登陆。如果没有，则跳转登陆，登陆成功后设置该对象

Django 模板：

模板引擎通过 `settings.py` 的 `TEMPLATES` 列表来配置。

是一个设置选项列表，与引擎一一对应。默认的空列表。

由 `startproject` 命令生成的 `settings.py` 定义了一些值

Django 项目可以配置一个或多个模板引擎。

Django 的模板系统自带内建的后台，称为 Django 模板语言(DTL)：

`django.template.backends.django.DjangoTemplates`。以及另外一种流行的 Jinja2：

`django.template.backends.jinja2.Jinja2`

绝大多数引擎都是从文件加载模板，所以每种模板引擎都包含两项通用设置：

DIRS：定义了一个目录列表，默认为空列表，模板引擎按列表顺序搜索这些目录以查找模板源文件

APP_DIRS：True/False，是否从每个已安装的应用中查找模板

使用模板：

视图函数把 `context` 字典传递给模板，字典中的 `key` 相当于是传递给模板的变量名，模板只要使用这个变量名，就可以得到相应的值

注意：不是把 `context` 这个名字传过去，传过去的是字典

模板元素：

变量：使用 `{{ ... }}` 的格式

简单变量：`{{变量名}}`

对象变量：`{{ 对象名.变量名 }}`

列表对象：`{{ 列表名.列表下标 }}`

字典对象：`{{ 字典名.键 }}`

显示所有键：`{{ 字典名.keys }}`

显示所有值：`{{ 字典名.values }}`

显示所有键值对：`{{ 字典名.items }}`

标签/指令：使用 `{% ... %}` 的格式

字符串：普通字符

循环结构：

`{% for ... %}`

...

`{% endfor %}`

条件分支：

`{% if ... %}`

...

`{% elif ... %}`

...

`{% else %}`

...

`{% endif %}`

变量过滤：

首字母大写：`{{ 变量 | capfirst }}`

转义：`{{ 变量 | addslashes }}`

显示行号：`{{ 变量 | linenumbers }}`

大写+行号：`{{ 变量 | upper | linenumbers }}`

指定日期格式：`{{ 日期 | date:"F j, Y" }}`

url 编码：`{{ url | urlencode }}`

列表连接：`{{ 列表变量 | join:", " }}`

模板继承：

很多网页的页面整体结构完全一样，只是其中的内容不一样，可以先建立一个基础模板页面，其他

页面继承这个模板，就可以拥有一致的风格

Django 模型：

模型是数据库表的 Python 对象表达

模型提供了数据库操作的基本功能

模型提供了数据库表数据的存储功能

模型对单表数据库操作简单易用，但在多表联查、复杂查询时，表达并不简洁，对数据库方面的约束会很多

在 `models.py` 中定义数据库格式后，通过 `makemigrations` 和 `makemigrate` 会生成文件型数据库文件 `db.sqlite3`，可通过 `sqlite3` 本地命令查看该数据

`.tables`：查看所有表

`.schema`：查看表结构

数据检索：

获取表的所有记录：表名.objects.all()

获取特定条件的记录：表名.objects.get(字段="值")

获取前 10 条记录：表名.objects.all()[:10]

排序：

表名.objects.order_by('字段')

过滤：

严格匹配：

表名.objects.filter(字段="值")

表名.objects.filter(字段__exact="值")

不区别大小写：

表名.objects.filter(字段__iexact="值")

包含：

表名.objects.filter(字段__contains="值")

包含且不区别大小写：

表名.objects.filter(字段__icontains="值")

正则过滤：

采用正则进行：

表名.objects.filter(字段__regex="^值")

采用正则表达式且不区分大小写：

表名.objects.filter(字段__iregex="^值")

双下划线：

`__exact`：精确等于

`__iexact`：精确等于且忽略大小写

`__contains`：包含

`__icontains`：包含且忽略大小写

`__gt`：大于

`__gte`：大于等于

`__lt`：小于

`__lte`：小于等于灵活的双下划线

`__in`：存在于一个 list 范围内

`__startswith`：以...开头

`__istartswith`：以...开头且忽略大小写

`__endswith`：以...结尾

`__iendswith`：以...结尾且忽略大小写

`__range`: 在...范围内
`__year`: 日期字段的年份
`__month`: 日期字段的月份
`__day`: 日期字段的日
`__isnull=True/False`: 是否为空值

排除过滤:

排除:

表名.objects.exclude(字段__contains="值")

过滤+排除的连续操作

表名.objects.filter(字段 1__contains="值 1").exclude(字段 2='值 2')

创建记录:

表名.objects.create(字段 1='值 1',字段 2=值 2,...)

表名.objects.get_or_create(字段 1='值 1',字段 2=值 2,...)

创建实例:

p = 表名(字段 1='值 1',字段 2=值 2,...)

p.save()

修改属性:

p = 表名(字段 1='值 1')

p.字段 1 = '值 1'

p.save()

通过 save 方法修改记录:

s = 表名(字段 1='值 1',字段 2=值 2,...)

s.save()

通过 update 方法修改记录: 表名.objects.filter(字段 1='值 1').update(字段 2='值 2',字段 3=值 3,...)

直接删除记录:

s = 表名()

s.id = 13

s.delete()

查找对象后删除记录:

s=表名.objects.get(id=13)

s.delete()

