3.函数式编程

- **概念**

"函数式编程"（Functional Programming）是一种"**编程范式**"（programming paradigm），也就是**如何编写程序的方法论。**

它将电脑运算视为函数的计算。函数编程语言最重要的基础是λ演算（lambda calculus），而且λ演算的函数可以接受函数当作输入（参数）和输出（返回值）。

它**属于"结构化编程"**的一种，主要思想是把运算过程尽量写成一系列嵌套的函数调用。

例如：**print(pow(round((3+2)/2),2))**

- **主要内容包括：**
  - 匿名函数（lambda表达式）
  - 高阶函数
  - 闭包函数
  - 装饰器
  - 内置函数
  - 内置模块

3.1 匿名函数（lambda表达式）

概念：没有名字的函数，”一句话的函数“
作用：简化代码，和其他高阶函数配合使用
语法：　*lambda 参数：表达式*

> "表达式"（expression）是一个单纯的运算过程，总是有返回值；
> "语句"（statement）是执行某种操作，没有返回值。

函数式编程要求，只使用表达式，不使用语句。也就是说，每一步都是单纯的运算，而且都有返回值。

传统定义函数的写法：

```
def getInfo():
    return "这是一条消息..."

print(getInfo())
```

(1)无参的lambda表达式

```
func = lambda : "这是一条消息..."      #定义一个lambda表达式（匿名函数）
                                       # 使用func变量指向这个表达式（函数）
print(func())                          #有返回值

#函数式编程，更鼓励”计算过程的单纯性“，建议把”输入/输出“限制到最小
func = lambda : print("这是一条消息...")
func()                                 #没有返回值
```

lambda表达式的格式：lambda 形式参数：表达式

（2）有参的Lambda表达式

```
def add(x,y):
    return x+y

print(add(1,2))            #输出结果：3

#定义
func = lambda x,y : x+y
#调用
print(func(1,2))          #输出结果：3
```

- lambda表达式可以使用可变参数

```
func = lambda *args: sum(args)
print(func(1,2,3))                #输出结果：6
print(func(10,20,30,40,50))       #输出结果：150
```

- lambda表达式可以使用关键字参数

```
func = lambda **kwargs : f"{kwargs}"
print(func(id = 1,city = "北京"))

func = lambda *args,**kwargs : f"{args},{kwargs}"
print(func(1,2,3,id = 1,city = "北京"))
```

输出结果：

```
{'id': 1, 'city': '北京'}
(1, 2, 3),{'id': 1, 'city': '北京'}
```

(3) 有条件判断的lambda表达式

- 三目表达式

  相当于其他编程语言中的“三目运算符”

```
#输出较大值
a = 1
b = 5

res = a if a>b else b        # 格式：条件成立时的结果   if  条件表达式  else  条件不成立时的结果
print(res)                   #输出结果：5
```

练习：使用lambda表达式，判断给定数值是否为偶数

```
func = lambda x : f"{x}是偶数" if x%2 == 0 else f"{x}不是偶数"
print(func(21))     #输出结果：21不是偶数
```

3.2 高阶函数（扩展）

概念：把函数当成参数传递的函数就是高阶函数。

```
def printInfo():                    #定义了一个函数
    print("info.....")

printInfo()                         #输出结果：info.....

#高阶函数
def runFunc(func):                   #定义了参数
    func()                           #调用传入的函数    # 相当于调用了printInfo()

runFunc(printInfo)                  #将函数作为参数值，括号中仅仅是函数名
                                    #输出结果：info.....
```

作用：可以简化代码，结合lambda表达式，效果更佳

3.2.1 map

语法：map(function, iterable, ...)

功能：把可迭代对象中的数据逐一拿出来,经过函数处理之后，将结果放到迭代器中并返回迭代器。

参数:

- func: 自定义函数 或者 内置函数
- iterable: 可迭代对象（常用：容器类型数据，range对象，迭代器）
- 返回值：迭代器

```
# 将给定的字符串列表转化为整数列表
l1 = ["1","2","3","4"]
res = []
for i in l1:
    #print(type(int(i)))
    res.append(int(i))
print(res)                  #输出结果：[1,2,3,4]

res = map(int,l1)
print(res,type(res))

for item in res:
    print(item)

#使用list() 强制将res转化为列表对象
print(list(res))
```

输出结果：

```
<map object at 0x000001A547622EF0> <class 'map'>
1
2
3
4
[]
```

通过下面的代码可以测试上面返回的res对象是否为迭代器对象：

```
from collections.abc import Iterable,Iterator

print(isinstance(res,Iterable)) #True,是可迭代对象
print(isinstance(res,Iterator)) #True,是迭代器
```

使用自定义函数

```
# 求列表中字符串所表达数值的平方
l1 = ["1","2","3","4"]

def getSquare(x):
    return int(x)**2          #必须有返回值

res = map(getSquare,l1)
print("自定义函数：",list(res))    #输出结果：自定义函数：[1, 4, 9, 16]
```

使用lambda表达式

```
l1 = ["1","2","3","4"]

res = map(lambda x:int(x)**2,l1)     #体现出了匿名函数的特点
print("lambda表达式：",list(res))     #输出结果：lambda表达式：  [1, 4, 9, 16]
```

多个可迭代对象，作为参数

```
l1 = [1,2,3,4]
l2 = [5,6,7,8]

# 分别获取两个迭代对象的元素，作为lambda表达式的参数
res = map(lambda x,y: x+y,l1,l2 )
print(list(res))                  # 输出结果：[6, 8, 10, 12]
```

应用案例：将下面的字典进行键值对互换：
dic = {"北京":"京","上海":"沪","广东":"粤","四川":"川"}

```
#第一种实现方法：
res = {}
for a,b in dic.items():
    res[b] = a
print(res)          #输出结果：{'京': '北京', '沪': '上海', '粤': '广东', '川': '四
川'}
```

```
#第二种实现方法：使用map+自定义函数
dic = {"北京":"京","上海":"沪","广东":"粤","四川":"川"}

def change(item):            #获取到的item是元组类型
    return (item[1],item[0])    #重新构造元组的内容

res = map(change,dic.items())

res = map(lambda item:(item[1],item[0]),dic.items())
print(dict(res))                 #可以直接通过dict进行强制类型转化为字典
                            #输出结果：{'京': '北京', '沪': '上海', '粤': '广东', '川':
'四川'}
```

dict也可以应用在其他容器对象上:

```
a = [(1,2),(3,4),(5,6)]     #列表
a = ((1,2),(3,4),(5,6))     #元组
a = {(1,2),(3,4),(5,6)}     #集合
b = dict(a)
print("字典对象：",b)         #输出结果：字典对象： {1: 2, 3: 4, 5: 6}
```

3.2.2 reduce

语法: reduce(function, sequence[, initial]) -> value

功能：给定序列，依次取出1个元素作为参数，与前一次函数结果进行函数运算，最后返回运算结果

参数:

- func 自定义函数 或者 内置函数
- iterable 可迭代对象（常用：容器类型数据 range对象 迭代器）
- 返回值：最终的计算结果

```
#将列表中每个元素相乘
from functools import reduce

obj = [1,2,3,4,5]

def multi(x,y):
    return x*y

res = reduce(multi,obj)
print(res,type(res))     #输出结果：120 <class 'int'>
```

应用案例：使用lambda表达式，完成下面计算：

f(1) = 1

f(n) = f(n-1) * n + 1

```
# 步骤：分析
# f(4) = f(3) * 4 + 1   => 10 * 4 + 1 = 41
# f(3) = f(2) * 3 + 1   => 3  * 3 + 1 = 10
# f(2) = f(1) * 2 + 1   => 1  * 2 + 1 = 3

#使用递归方式实现：
def getRes(x):
    if x == 1:
        return 1
    return getRes(x-1) * x + 1
print(getRes(1))       #输出结果：1
```

使用reduce实现：

```
from functools import reduce
lst = [1,2,3,4]
res = reduce(lambda x,y:x*y+1,lst)
res = reduce(lambda x,y:x*y+1,range(1,5))
print(res)            #输出结果：41

# 如果初始值不是1，怎么传递这个数值呢？
res = reduce(lambda x,y:x+y,range(1,5),100)
print(res)            #输出结果：110
```

3.2.3 filter

语法：filter(func,iterable) -> iterator

功能：过滤数据

- 如果函数的返回值是True，代表保留当前数据
- 如果函数的返回值是False，代表舍弃当前数据

参数：

- func 自定义函数
- iterable 可迭代对象（常用：容器类型数据，range对象，迭代器）
- 返回值：迭代器

```
lst = [1,2,3,4,21,32,43,54]

def getEven(x):
    if x%2 == 0:
        return True
    else:
        return False

res = filter(getEven,lst)
print(res)
res = filter(lambda x:True if x%2 == 0 else False,lst)
print(list(res))
```

输出结果：

> <filter object at 0x000001AA4AB02EB8>
> [2, 4, 32, 54]

```
from collections.abc import Iterable,Iterator
res = filter(getEven,lst)
print(isinstance(res,Iterator))
print(isinstance(res,Iterable))
```

输出结果：

> True
>
> True

3.2.4 sorted

语法：sorted(iterable,reverse=False,key=函数) -> Iterator

功能：排序

参数：

- iterable: 可迭代对象（常用：容器类型数据，range对象，迭代器）
- reverse: 是否倒序

  默认正序reverse=False（从小到大）如果reverse=True 代表倒序（从大到小）
- key = 自定义函数 或者 内置函数
- 返回值：排序的序列

```
lst = [1,3,2,5,8,-2]
print(sorted(lst))
lst = "dfackj"
print(sorted(lst))
```

输出结果：

> [-2, 1, 2, 3, 5, 8]
> ['a', 'c', 'd', 'f', 'j', 'k']

```python
#按照绝对值进行排序

print(abs(-2))                #内置函数abs() 求指定数值的绝对值

lst = [1,3,-2,5,8,2]
print(sorted(lst,key=abs))  #逐一取出每个数值进行计算后，按照计算结果，排序输出

dic = {3:"A",2:"C",-1:"B"}
print(sorted(dic))            #字典排序，默认输出排序后的键值序列
```

输出结果：

```
2
[1, -2, 2, 3, 5, 8]
[-1, 2, 3]
```

```python
#按照余数排序
lst = [20,31,47,19,15]       #余数分别为：2，1，2，1,0

def getRemainder(x):
    return x%3

res = sorted(lst,key=getRemainder)
res = sorted(lst,key=lambda x:x%3)
print(res)          #输出结果：[15, 31, 19, 20, 47]
```

```python
#sort() 和 sorted 的区别

lst = [1,3,2,5,8,-2]
print("排序前：",lst)
lst.sort()
print("sort排序后：",lst)  #改变了原有的序列
lst2 = sorted(lst)            #产生新列表
print("新列表：",lst2)
print("sorted排序后：",lst)
```

输出结果：

```
排序前： [1, 3, 2, 5, 8, -2]
sort排序后： [-2, 1, 2, 3, 5, 8]
新列表： [-2, 1, 2, 3, 5, 8]
sorted排序后： [-2, 1, 2, 3, 5, 8]
```

3.3 返回函数

函数返回值也可以是函数

```python
def food(name):                    #外函数

    def prepare():                 #内函数
        print(f"[{name}]制作步骤：备菜...")     #内部函数可以使用外部函数的变量
```

```
    def cook():
        print(f"[{name}]制作步骤：烹饪...")

    def serve():
        prepare()
        cook()
        print(f"[{name}]制作步骤：上菜！")

    return serve
    # return (prepare,cook,serve)

m = food("番茄炒蛋")
# print(m,type(m))
# m[0]()
# m[1]()

f = food("小鸡炖蘑菇")

f()
m()
```

输出结果：

```
[小鸡炖蘑菇]制作步骤：备菜...
[小鸡炖蘑菇]制作步骤：烹饪...
[小鸡炖蘑菇]制作步骤：上菜！
[番茄炒蛋]制作步骤：备菜...
[番茄炒蛋]制作步骤：烹饪...
[番茄炒蛋]制作步骤：上菜！
```

小结：

1. 可以将内部函数的引用，返回调用的地方；外部决定执行的时机。
2. 外部不需要依次调用每个细节，实现了内部过程的“封装”

3.4 偏函数（了解）

当函数的参数个数太多，需要简化时，使用functools.partial可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

```
a = int("1100",base=10)      #默认按照十进制来转化字符串
print(a,type(a))

a = int("0b1111",base=2)
print(a,type(a))
```

输出结果：

```
1100 <class 'int'>
15 <class 'int'>
```

```
def int2(x,base=2):
    return int(x,base=base)

# x = int2("1100")
# print(x)
x = int2("1100",base = 10)
print(x)                          #输出结果：1100
```

```
import functools

int2 = functools.partial(int,base=2)     #定义新的函数，等同于：将参数base默认为2的int函数
print(type(int2))

x = int2("1100")
print(x)

#int2仅仅是设定了默认值，也可以将默认值修改为其他值
x = int2("1100",base = 10)
print(x)
```

输出结果：

```
<class 'functools.partial'>
12
1100
```

3.5 闭包函数

概念：内函数使用了外函数的局部变量，并且外函数把内函数返回出来的过程叫做闭包，这个内函数叫做闭包函数。

本质：相对于面向对象的“封装”，闭包可以理解为是函数式编程中的“封装”。

```
# 闭包函数的语法：
def outer():
    a = 5
    def inner():     #闭包函数
        print(a)     #通过使用外部函数的变量，延长了变量的使用时间
    return inner

func = outer()       #外部函数调用后，返回内部函数的地址
func()                #通过内部函数的地址，完成内部函数调用

                      #输出结果：5
```

```
# 对比正常的局部变量
# 局部变量的生命周期最短,在调用结束之后,立即释放.
def func():
     a = 5
     print(a)

f = func()       # f 函数的返回值
print(f.a)       #报错：AttributeError: 'NoneType' object has no attribute 'a'
```

```
#应用场景：
#test函数中的a，b参数不经常变化，
#内部函数test_in中的参数c每次调用都需要重新赋值

def test(a,b,c):
    print(a*b+c)

test(1,1,2)
test(1,1,3)

test(3,5,2)
test(3,5,3)
```

输出结果：

```
3

4

17

18
```

```
def test(a,b):
    def test_in(c):
        print(a*b+c)
    return test_in

num = test(1,1)
num(2)
num(3)
print("-"*30)
num_02 = test(2,2)
num_02(2)
num_02(3)

num(6)    #    这个依然会继续执行，  还是按照a=1,b=1    他们是开辟两个不同的内存空间。
```

输出结果：

```
3

4

------------------------------
```

```
6
7
7
```

小结：

闭包，构建了类似面向对象中“类”的形式，目的是为了实现：“数据的封装”

内部函数可以使用外部函数定义的属性，“多个外部函数”的内部函数不能相互共享数据

类似于：每个外部函数调用时，开辟一块新的内存空间

每个内存空间中的属性和内部函数有绑定关系

3.6 装饰器

功能：不改变现有函数的前提下，扩展函数功能

语法：使用@符号

3.6.3.1装饰器的原型

问题： 在执行printInfo函数前，进行权限校验；执行后，进行日志记录

不允许修改printInfo函数

```
def check(func):

    def checkAndLog():
        print("函数运行前：权限校验...")
        func()
        print("函数运行后：日志记录...")

    return checkAndLog

def printInfo():
    print("我是一个普通的函数")

newFunc = check(printInfo)        #返回内部函数
newFunc()                         #执行内部函数
# printInfo()
```

输出结果：

```
函数运行前：权限校验...
我是一个普通的函数
函数运行后：日志记录...
```

3.6.3.2装饰器的定义与调用

```
#定义装饰器函数

def check(func):

    def checkAndLog():
        print("函数运行前：权限校验...")
        func()
        print("函数运行后：日志记录...")

    return checkAndLog

@check                #定义装饰器修饰的位置，相当于 check(printInfo)
def printInfo():
    print("我是一个普通的函数")

@check
def test():
    print("test....")

printInfo()     #执行函数前，会先执行装饰器
test()
```

输出结果：

> 函数运行前：权限校验...
> 我是一个普通的函数
> 函数运行后：日志记录...
> 函数运行前：权限校验...
> test....
> 函数运行后：日志记录...

3.6.3.3 装饰器的嵌套

两个及多个装饰器可以装饰同一个函数。

执行的过程为：自下而上逐步修饰，完成后一次性输出。

```
def zsA(func):
    def resFunc():
        print("zsA执行前...")
        func()
        print("zsA执行后...")
    return resFunc

def zsB(func):
    def resFunc():
        print("zsB执行前...")
        func()
        print("zsB执行后...")
    return resFunc

@zsB
```

```
@zsA              #自下而上逐步修饰，完成后一次性输出
def test():
    print("这是测试函数")

test()

#相当于：
res1 = zsA(test)
res2 = zsB(res1)
res2()
```

输出结果：

```
zsA执行前...
zsB执行前...
zsA执行前...
这是测试函数
zsA执行后...
zsB执行后...
zsA执行后...
zsB执行后...
```

3.6.3.4 带有参数的装饰器

如果原函数带有参数，那么返回的新函数也要带有参数，且参数一一对应

```
def tool(func):
    def log(name,pwd):
        print(f"日志记录：【{name}:{pwd}】")
        func(name,pwd)
    return log

@tool
def login(name,pwd):
    print(f"用户名：{name},密码：{pwd}")

# res = tool()
# res("admin",123456)
login("admin",123456)
```

输出结果：

```
日志记录：【admin:123456】
用户名：admin,密码：123456
```

3.6.3.5 带返回值的装饰器

如果原函数带有返回值，装饰器也要有返回值，才能将原有函数的执行结果传递出来。

```
def tool(func):
    def log(name,pwd):
        print(f"日志记录：【{name}:{pwd}】")
        state = func(name,pwd)
```

```
        return state
    return log

@tool
def login(name,pwd):
    print(f"用户名：{name},密码：{pwd}")
    if name == "admin" and pwd == 123456:
        return "登录成功"
    return "登录失败"

# res = tool()
# res("admin",123456)
res = login("admin",12345678)
print(res)
```

输出结果：

> 日志记录：【admin:12345678】
> 用户名：admin,密码：12345678
> 登录失败

3.6.3.6 定义通用装饰器

增加装饰器的适用性，可以使用可变参数，以保证装饰器能够适用于更多的函数。

```
def tool(func):
    def log(*args,**kwargs):
        print(f"日志记录前。。。")
        state = func(*args,**kwargs)
        print(f"日志记录后。。。")
        return state
    return log

@tool
def test():
    print("test....")

@tool
def login(name,pwd):
    print(f"用户名：{name},密码：{pwd}")
    if name == "admin" and pwd == 123456:
        return "登录成功"
    return "登录失败"

test()
res = login("admin",12345678)
print(res)
```

输出结果：

> 日志记录前。。。
> test....
> 日志记录后。。。
> 日志记录前。。。

> 用户名：admin,密码：12345678
> 日志记录后。。。
> 登录失败

3.6.3.7 使用面向对象的方式定义装饰器（扩展）

调用时应用call方法调用内部函数

```
class MyTool:

    def __call__(self, func):
        return self.tool2(func)

    def tool1(func):
        def log():
            print("log....")
            func()
        return log

    def tool2(self,func):
        def check():
            print("check....")
            func()
        return check
```

```
#调用方法一
@MyTool.tool1
def test():
    print("test....")

test()
```

输出结果：

> log....
>
> test....

```
#调用方法二

@MyTool()         # MyTool通过函数的方式来调用，默认会触发__call__方法
def test():
    print("test....")

test()
```

输出结果：

> check....
>
> test....

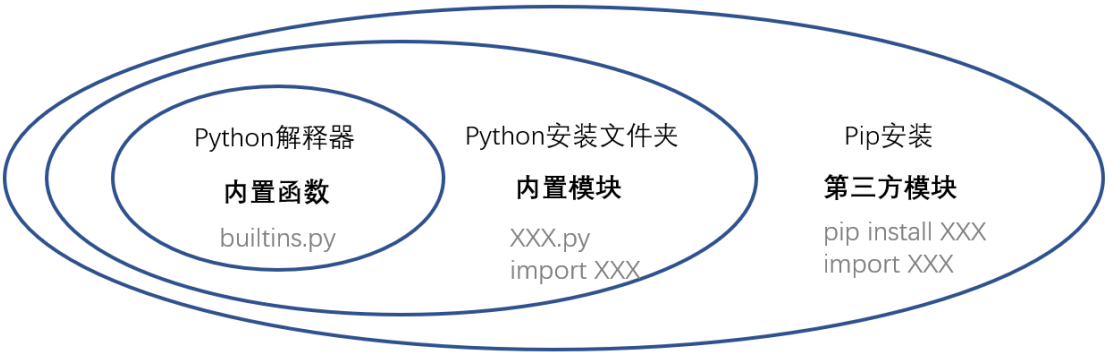
3.7 内置函数

3.7.1 Python标准库

Python 一同发行的除了语法本身，还包括标准库。它还描述了通常包含在 Python 发行版中的一些可选组件。

Python 标准库非常庞大，所提供的组件涉及范围十分广泛，正如以下内容目录所显示的。这个库包含了多个内置模块 (以 C 编写)，Python 程序员必须依靠它们来实现系统级功能，例如文件 I/O，此外还有大量以 Python 编写的模块，提供了日常编程中许多问题的标准解决方案。其中有些模块经过专门设计，通过将特定平台功能抽象化为平台中立的 API 来鼓励和加强 Python 程序的可移植性。

详细地址：https://docs.python.org/zh-cn/3/library/index.html



3.7.2 常用内置函数

Python 解释器内置了很多函数和类型：https://docs.python.org/zh-cn/3/library/functions.html

		内置函数		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

常用内置函数的，简单的功能归纳：

```
abs     绝对值函数
round   四舍五入 (n.5 n为偶数则舍去 n.5 n为奇数，则进一!)
sum     计算一个序列得和
max     获取一个序列里边的最大值
min     获取一个序列里边的最小值
pow     计算某个数值的x次方
range   产生指定范围数据的可迭代对象
bin     将10进制数据转化为二进制
oct     将10进制数据转化为八进制
hex     将10进制数据转化为16进制
chr     将ASCII编码转换为字符
ord     将字符转换为ASCII编码
eval    将字符串当作python代码执行
exec    将字符串当作python代码执行(功能更强大)
repr    不转义字符输出字符串
input   接受输入字符串
hash    生成哈希值
```

课堂练习：

结合官方文档，及下面的示例。

尝试通过学过的知识，完成内置函数已有功能。

1. abs()函数返回数字的绝对值。

```
print( abs(-45) )                   # 返回 45

print("abs(0.2):",abs(0.2))         # 返回 abs(0.2): 0.2
```

2. all() 函数用于判断给定的参数中的所有元素是否都为 TRUE，如果是返回 True，否则返回 False。元素除了是 0、空、None、False 外都算 True；空元组、空列表返回值为True。

```
print( all( [0.1,1,-1] )  )         # 返回 True
print( all( (None,1) )  )           # 返回 False（其中一个元素为None）
print( all( [0,1,-1] )  )           # 返回 False（其中一个元素为0）
print( all( [" ","a",""] )  )       # 返回 False(第三个元素为空)
```

3. any() 函数用于判断给定的参数是否全部为False，是则返回False，如果有一个为True，则返回True。 元素除了是 0、空、False外都算 TRUE。

```
# 参数全部不为 0、空、FALSE
print(any("-45"))                    #  True
print(any(["-45"]))                  #  True
print( any( ("0","ab","") ) )        #  True（注意：第一个参数0加了双引号，表示为一个字符串）

# 参数全部为 0、空、False
print( any( (0,"") ) )               #  False
print( any( (0,"",False) ) )         #  False
```

4. bin()函数返回一个整数int或者长整数long int的二进制表示。

```
print( bin(10) )         #  0b1010

print( bin(133) )        #  0b10000101
```

5. bool() 函数用于将给定参数转换为布尔类型，如果参数不为空或不为0，返回True；参数为0或没有参数，返回False。

```
print( bool(10) )                    #  True
print( bool([0]) )                   #  True
print( bool(["123","s",0]) )         #  True

print( bool(0) )                     #  False
print( bool() )                      #  False
```

6. bytearray()方法返回一个新字节数组。这个数组里的元素是可变的，并且每个元素的值范围: 0 <= x < 256（即0-255）。即bytearray()是可修改的二进制字节格式。

```
b = bytearray("abcd",encoding="utf-8")
print(b[0])              #  返回数字97，即把“abcd”的“a"对应的ascii码打印出来了
b[0] = 99                #  把字符串第一个字节修改为99（即对应字母为“c”）
print(b)                 #  返回：bytearray(b'cbcd')---第一个字节a已被修改为c
```

7. callable()函数用于检查一个对象是否可调用的。对于函数、方法、lambda函式、类以及实现了 **call** 方法的类实例, 它都返回 True。（可以加括号的都可以调用）

```
def sayhi():pass                     #  先定义一个函数sayhi()
print( callable( sayhi ) )           #  True
a = 1
print( callable( a ) )               #  False
```

8. chr()函数用一个范围在range(256)内（即0～255）的整数作参数，返回一个对应的ASCII数值。

```
# 把数字98在ascii码中对应的字符打印出来
print( chr(98) )                    #  返回：b
```

9. dict()函数用来将元组/列表转换为字典格式。

```
print(dict(a='a', b='b', t='t'))
#  返回：{'b': 'b', 'a': 'a', 't': 't'}

print(dict( [ ('one',1),('two',2),('three',3) ]  ) )    # 可迭代对象方式来构造字典
#  返回：{'two': 2, 'one': 1, 'three': 3}

print(dict(zip(["1","2","3"],["a","b","c"])))      # 映射函数方式来构造字典
#  返回：{'2': 'b', '3': 'c', '1': 'a'}
```

10. dir()函数不带参数时，返回当前范围内的变量、方法和定义的类型列表；带参数时，返回参数的属性、方法列表。

```
print( dir() )            #  获得当前模块的属性列表
#  返回：['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__']
print( dir([]) )          #  查看列表的方法
#  返回：['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

11. divmod() 函数把除数和余数运算结果结合起来，返回一个包含商和余数的元组（商x，余数y）。

```
print( divmod(5,2) )    #  返回：(2, 1)
print( divmod(5,1) )    #  返回：(5, 0)
print( divmod(5,3) )    #  返回：(1, 2)
```

12. enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中。Python 2.3. 以上版本可用，2.6 添加 start 参数。

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
print(list(enumerate(seasons)))
#  返回：[(0, 'Spring'), (1, 'Summer'),(2, 'Fall'), (3, 'Winter')]
print(list(enumerate(seasons, start=1)) )          # 下标从 1 开始
#  返回：[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

13. eval() 函数用来执行一个字符串表达式，并返回表达式的值。

```
print(eval('3 * 2'))          # 6
1 x = 7
2 print(eval('3 + x'))        # 10
```

14. exec() 执行储存在字符串或文件中的Python语句，相比于eval，exec可以执行更复杂的Python代码。

```
exec("print('Hello World')")        # 执行简单的字符串
# Hello World
```

```
exec("for i in range(5): print('iter time is %d'%i)")      # 执行复杂的for循环
# iter time is 0
# iter time is 1
# iter time is 2
# iter time is 3
# iter time is 4
```

15. filter()用于过滤序列，过滤掉不符合条件的元素，返回一个迭代器对象，可用list()来转换为列表。

注意： filter()接收两个参数，第一个为函数，第二个为序列，序列的每个元素作为参数传递给函数进行判断，然后返回True或 False，最后将返回 True 的元素放到新列表中。

```
res = filter(lambda n:n>5,range(10))   # 过滤掉0-9中不符合n>5的数据
for i in res:                          # 循环打印符合n>5的数据
    print(i)
```

输出：

```
5
6
7
8
9
```

16. format()是一种格式化字符串的函数 ， 基本语法是通过 {} 和 : 来代替以前的 % 。format 函数可以接受不限个参数，位置可以不按顺序。

```
# 位置映射

print( "{}{}".format('a','1') )
# a1

print('name:{n},url:{u}'.format(n='alex',u='www.xxxxx.com'))
# name:alex,url:www.xxxxx.com
```

```
# 元素访问

print( "{0[0]},{0[1]}".format(('baidu','com')) )                # 按顺序
# baidu,com

print( "{0[2]},{0[0]},{0[1]}".format(('baidu','com','www')) )    # 不按顺序
# www,baidu,com
```

17. float() 函数用于将整数和字符串转换成浮点数。

```
print(float(1))
# 1.0

print(float(0.1))
# 0.1

print(float('123'))
# 123.0
```

18. frozenset() 返回一个冻结的集合（一个无序的不重复元素序列），冻结后集合不能再添加或删除任何元素。

```
a = frozenset(range(10))             # 先创建一个冻结集合
print(a)
# frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

del a[0]                             # 试图删除冻结集合a中的元素，报错
# TypeError: 'frozenset' object doesn't support item deletion
```

```
b = frozenset("happy")               # 将字符串转换成一个集合
print(b)
# frozenset({'a', 'h', 'p', 'y'})    # 无序不重复
c = frozenset()                      # 创建一个空集合
print(c)
# frozenset()                        # 如果不提供任何参数，默认会生成空集合
```

19. globals() 函数会以字典格式返回当前位置的全部全局变量。

```
print(globals())     # globals 函数返回一个全局变量的字典，包括所有导入的变量。

#  {'__file__': 'C:/Users/Administrator/PycharmProjects/test/day4/内置函数-
globals().py', '__spec__': None, '__doc__': None, '__package__': None, 'a':
'append', '__cached__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x0000000000666B00>,
'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__'}
```

20. hasattr() 函数用于判断对象是否包含对应的属性。如果对象有该属性返回 True，否则返回 False。

```
class t:
    a = 1
    b = 2
    c = 3

p = t()
print(hasattr(p,'a'))    # True
print(hasattr(p,'b'))    # True
print(hasattr(p,'x'))    # False
```

21. hash() 用于获取一个对象（数字或者字符串等）的哈希值。不能直接应用于 list、set、dictionary。

注意：在 hash() 对对象使用时，所得的结果不仅和对象的内容有关，还和对象的 id()，也就是内存地址有关。

```
print(hash(1))                  # 1
print(hash(20000))              # 20000
print(hash('123'))              # -6436280630278763230
print(hash('ab12'))             # 5468785079765213470
print(hash('ab12'))             # 5468785079765213470
```

22. help() 函数用于查看函数或模块用途的详细说明。

```
help('sys')             # 查看 sys 模块的帮助
help('str')             # 查看 str 数据类型的帮助
a = [1,2,3]
help(a)                 # 查看列表 list 帮助信息
help(a.append)          # 显示list的append方法的帮助
```

23. hex() 函数用于将一个整数转换为十六进制数。返回一个字符串，以0x开头。

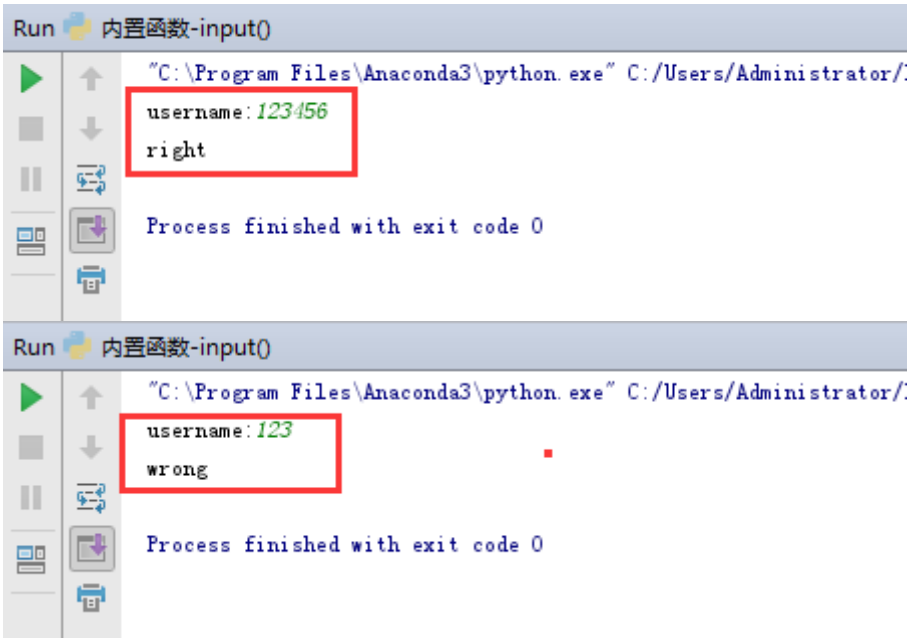
```
print(hex(1))        # 0x1

print(hex(-256))     # -0x100
print(type(hex(-256)))    #<class 'str'>
```

24. id()函数用于获取对象的内存地址。

```
a = "123"                           # 字符串
print(id(a))                        # 13870392

b = [1,2,3]                         # 列表
print(id(b))                        # 7184328

c = {'num1':1,'num2':2,'num3':3}     # 字典
print(id(c))                         # 6923656
```

25. input() 函数接受一个标准输入数据，返回为 string 类型。这个函数是最最常用的了。在Python3.x 中 raw_input() 和 input() 进行了整合，仅保留了input( )函数。

```
a = '123456'
b = input("username:")

if b == a :             # 如果b的输入数据等于a存储的数据，打印”right“
    print("right")
else:                   # 否则打印”wrong“
    print("wrong")
```



26. int() 函数用于将一个字符串或数字转换为整型。

```
print(int())                # 不传入参数时，得到结果0
print(int(0.5))             # 去掉小数部分，得到结果0
print(int(3))               # 得到结果3
print(int('0xa',16))        # 十六进制数“0xa”转换成十进制整数，得到结果10
print(int('00010',2))       # 二进制数“00010”转换成十进制整数，得到结果2
```

27. isinstance() 函数来判断一个对象是否是一个已知的类型，返回布尔值。类似 type()。

```
a = 2

print(isinstance(a,int))                    #  True
print(isinstance(a,str))                    #  False
print(isinstance(a,(str,tuple,dict)))       #  False
print(isinstance(a,(str,tuple,int)))        # 是元组其中的一个则返回True
```

- **isinstance() 与 type() 区别：**

type() 不会认为子类是一种父类类型，不考虑继承关系。
isinstance() 会认为子类是一种父类类型，考虑继承关系。
如果要判断两个类型是否相同推荐使用 isinstance()。

示例：

```
class A:
   pass

class B(A):
    pass

print(isinstance(A(),A))    # True
print( type(A()) == A )     # True

print(isinstance(B(),A))    # True
print( type(B()) == A )     # False   --type()不考虑继承关系
```

28. issubclass()用于判断参数class是否是类型参数classinfo的子类，是则返回True，否则返回False。

语法：issubclass(class,classinfo)。

```
class a:
    pass
class b(a):     # b继承了a，即b是a的子类
    pass

print(issubclass(a,b))    # 判断 a 是 b 的子类?
# False
print(issubclass(b,a))    # 判断 b 是 a 的子类?
# True
```

29. iter() 函数用来生成迭代器。list、tuple等都是可迭代对象，我们可以通过iter()函数获取这些可迭代对象的迭代器，然后可以对获取到的迭代器不断用next()函数来获取下条数据。iter()函数实际上就是调了可迭代对象的 __iter__ 方法。

注意：当已经迭代完最后一个数据之后，再次调用next()函数会抛出 StopIteration的异常，来告诉我们所有数据都已迭代完成。

```
it = [1,2,3]
it_list = iter(it)

print(next(it_list))
# 1
print(next(it_list))
# 2
print(next(it_list))
# 3
print(next(it_list))
# StopIteration
```

30. len() 方法返回对象（字符、列表、元组等）长度或元素个数。

```
# len()方法返回对象（字符、列表、元组等）长度或元素个数。
print(len('1234'))              # 字符串，返回字符长度4
print(len(['1234','asd',1]))    # 列表，返回元素个数3
print(len((1,2,3,4,50)))        # 元组，返回元素个数5

print(len(12))                  # 注意：整数类型不适用，否则报错
# TypeError: object of type 'int' has no len()
```

31. list() 方法用于将元组转换为列表。

注：元组与列表是非常类似的，区别在于元组的元素值不能修改，元组是放在括号中，列表是放于方括号中。

```
print( list((1,2,3)))           # [1, 2, 3]
```

32. map()接收函数f和list，并通过把函数f依次作用在list的每个元素上，得到一个新的list并返回。

```
res = map(lambda n: n*2,[0,1,2,3,4,5])    # 使用 lambda 匿名函数
for i in res:
    print(i)

# 返回以下数据：
# 0
# 2
# 4
# 6
# 8
# 10
```

```
# 提供了两个列表，对相同位置的列表数据进行相加

a = map(lambda x,y : x+y,[1,2,3,4,5],[2,4,6,8,10])
for i in a:
     print(i)

# 返回以下数据：
# 3
# 6
# 9
# 12
# 15
```

33. max()函数返回给定参数的最大值，参数可以为序列。

```
print("max(10,20,30):" , max(10,20,30) )
# max(10,20,30): 30

print("max(10,-2,3.4):" , max(10,-2,3.4) )
# max(10,-2,3.4): 10

print("max({'b':2,'a':1,'c':0}):" , max({'b':2,'a':1,'c':0}) )   # 字典，默认按key
排序
# max({'b':2,'a':1,'c':0}): c
```

34. min()函数返回给定参数的最小值，参数可以为序列。

```
print("min(10,20,30):" , min(10,20,30) )
# min(10,20,30): 10

print("min(10,-2,3.4):" , min(10,-2,3.4) )
# min(10,-2,3.4): -2

print("min({'b':2,'a':1,'c':0}):" , min({'b':2,'a':1,'c':0}) )   # 字典，默认按key
排序
# min({'b':2,'a':1,'c':0}): a
```

35. next() 返回迭代器的下一个项目。

```
# 首先获得Iterator对象:
it = iter([1,2,3,4,5])

# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
        print(x)
    except StopIteration:
        break
    # 遇到StopIteration就退出循环
```

36. oct() 函数将一个整数转换成八进制字符串。

```
print( oct(10) )          # 0o12
print( oct(255) )         # 0o377
print( oct(-6655) )       # -0o14777
```

37. open() 函数用于打开一个文件，创建一个 file 对象，相关的方法才可以调用它进行读写。

```
f = open("test1.txt","w",encoding="utf-8")     # 创建一个file
print(f.write("abc"))

f = open("test1.txt","r",encoding="utf-8")     # 读取文件数据
print(f.read())
```

38. ord()函数是chr()的配对函数，它以一个字符（长度为1的字符串）作为参数，返回对应的ASCII数值，或者Unicode数值，如果所给的 Unicode 字符超出了定义范围，则会引发一个 TypeError 的异常。

```
# 把字符 b（长度为1的字符串）作为参数在ascii码中对应的字符打印出来

print( ord('b') )    # 返回：98
print( ord('%') )    # 返回：37
```

39. pow()函数返回x的y次方的值。

注意：pow()通过内置的方法直接调用，内置方法会把参数作为整型，而 math 模块则会把参数转换为 float。

```
# 通过内置的方法直接调用

print( pow(2,2) )          #  2的二次方
# 4
print( pow(2,-2) )         #  2的负二次方
# 0.5
```

```
# 导入math模块

import math

print(math.pow(3,2))   # 3的负二次方
# 9.0
```

40. print()用于打印输出，最常见的一个函数。print 在Python3.x是一个函数，但在Python2.x版本只是一个关键字。

```
print( abs(-45) )           # 45
print("Hello World!")       # Hello World!
print([1,2,3])              # [1, 2, 3]
```

41. range() 函数可创建一个整数列表，一般用在 for 循环中。语法：range(start, stop[, step])

```
for i in range(10):
    print(i)            # 依次打印数字0-9

for a in range(0,10,2):     # 步长为2
    print(a)            # 打印0,2,4,6,8

for b in range(10, 0, -2):  # 步长为-2
    print(b)            # 打印10,8,6,4,2
```

42. reduce() 函数会对参数序列中元素进行累积。在Python3 ， reduce()被放置在functools模块里，如果想要使用它，需要先引入functools模块。

```
import functools

a = functools.reduce(lambda x,y:x+y,[1,2,3])
print(a)            # 6 ，  即从1加到3

b = functools.reduce(lambda x,y:x+y,range(10))
print(b)            # 45 ，  即从0加到9
```

43. repr() 函数将对象转化为供解释器读取的形式。返回一个对象的 string 格式。

```
r = repr((1,2,3))
print( r )              # (1, 2, 3)
print( type(r) )        #  <class 'str'>

dict = repr({'a':1,'b':2,'c':3})
print( dict )    # {'c': 3, 'a': 1, 'b': 2}
print( type(dict) )    # <class 'str'>
```

44. reversed() 函数返回一个反转的迭代器。 reversed(seq)要转换的序列，可以是 tuple, string, list 或 range。

```
rev = reversed( [1,2,3,4,5] )       # 列表
print(list(rev))
# [5, 4, 3, 2, 1]

rev1 = reversed( "school" )         # 元组
print(tuple(rev1))
# ('l', 'o', 'o', 'h', 'c', 's')

rev2 = reversed(range(10))          # range
print(list(rev2))
# [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

45. round() 方法返回浮点数x的四舍五入值。 （除非对精确度没什么要求，否则尽量避开用round()函数）

```
print( round(4.3))          # 只有一个参数时，默认保留到整数
# 4

print( round(2.678,2))      #  保留2位小数
#  2.68

print( round(5/3,3))      #  运算表达式并保留3位小数
#  1.667
```

46. set() 函数创建一个无序不重复元素集，可进行关系测试，删除重复数据，还可以计算交集、差集、并集等。

```
a = set('school')
print(a)             # 重复的被删除，得到结果：{'o', 'c', 's', 'l', 'h'}
```

```
b = set([1,2,3,4,5])
c = set([2,4,6,8,10])

print(b & c)      # 交集，得到结果为{2, 4}
print(b | c)      # 并集，得到结果为{1, 2, 3, 4, 5, 6, 8, 10}
print(b - c)      # 差集，得到结果为{1, 3, 5}
```

47. slice() 函数实现切片对象，主要用在切片操作函数里的参数传递。

```
a = slice("school")
print(a)             # slice(None, 'school', None)
```

48. sorted() 函数对所有可迭代的对象进行排序（默认升序）操作。

```
# 对列表进行排序
print(sorted([1,2,5,30,4,22]))    # [1, 2, 4, 5, 22, 30]

# 对字典进行排序
dict = {23:42,1:0,98:46,47:-28}
print( sorted(dict) )                      # 只对key排序
# [1, 23, 47, 98]

print( sorted(dict.items()) )              # 默认按key进行排序
# [(1, 0), (23, 42), (47, -28), (98, 46)]

print( sorted(dict.items(),key=lambda x:x[1]) )      # 用匿名函数实现按value进行排序
# [(47, -28), (1, 0), (23, 42), (98, 46)]
```

```
# 利用key进行倒序排序
test1 = [1,2,5,30,4,22]
r_list = sorted(test1,key=lambda x:x*-1)
print(r_list)                    # [30, 22, 5, 4, 2, 1]


# 要进行反向排序，也可以通过传入第三个参数 reverse=True：
test2 = [1,2,5,30,4,22]
print(sorted(test2,reverse=True))    # [30, 22, 5, 4, 2, 1]
```

49. str() 函数将对象转化为string格式。

```
a =  str((1,2,3))
print(a)            # 打印a，得到结果(1, 2, 3)
print(type(a))      # 打印a的类型，得到结果 <class 'str'>
```

50. sum()函数对参数进行求和计算。

```
print( sum([1,2,3]) )         #  6
print( sum([1,2,3],4) )       #  列表计算总和后再加4，得到结果10
print( sum( (1,2,3),4 ) )     #  元组计算总和后再加4，得到结果10
```

51. tuple()函数将列表转换为元组。

注：元组与列表是非常类似的，区别在于元组的元素值不能修改，元组是放在括号中，列表是放于方括号中。

```
print( tuple([1, 2, 3]))     # (1,2,3)
```

52. type() 函数如果你只有第一个参数则返回对象的类型，三个参数返回新的类型对象。

```
print(type(1))                  # <class 'int'>
print(type("123"))              # <class 'str'>
print(type([123,456]))          # <class 'list'>
print(type( (123,456) ) )       # <class 'tuple'>
print(type({'a':1,'b':2}) )     # <class 'dict'>
```

53. zip() 函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的对象，这样做的好处是节约了不少的内存。可以使用 list() 转换来输出列表。如果各个迭代器的元素个数不一致，则返回列表长度与最短的对象相同。利用 * 号操作符，可以将元组解压为列表。

```
a = [1,2,3]
b = [4,5,6]
c = [7,8,9,10]
```

```
for i in zip(a,b):
    print(i)

# 返回结果:
# (1, 4)
# (2, 5)
# (3, 6)

print(list(zip(a,b)))       # list() 转换为列表
# [(1, 4), (2, 5), (3, 6)]

print(list(zip(b,c)))       # 元素个数与最短的列表一致
# [(4, 7), (5, 8), (6, 9)]

a1,a2 = zip(*zip(a,b))      #  用zip(*)解压
print(list(a1))          # [1, 2, 3]
print(list(a2))          # [4, 5, 6]
```

54. __import__() 函数用于动态加载类和函数。如果一个模块经常变化就可以使用__import__() 来动态载入。

```
__import__('decorator')

# 返回结果如下:
# in the bar
# the func run time is 3.000171661376953
# 首先获得Iterator对象:
it = iter([1,2,3,4,5])
# 循环:
while True:
    try:
        # 获得下一个值:
x = next(it)
        print(x)
    except StopIteration:
        break
# 遇到StopIteration就退出循环
```

3.7.2 LEGB规则

变量作用域的规则：LEGB规则
Python 在查找"名称"时，是按照 LEGB 规则查找的：
　Local-->Enclosed-->Global-->Built in
L —— Local(function) 指的就是函数或者类的方法内部
E —— Enclosing function locals（一个函数包裹另一个函数，闭包）
G —— Global(module) 指的是模块中的全局变量
B —— Builtin(Python) 指的是 Python 为自己保留的特殊名称

```
a = 100              #全局的a，nonlocal影响不到
def outer():
    # a = 10         #如果外部函数没有找到a，报错：SyntaxError: no binding for
nonlocal 'a' found
    def inner():
        nonlocal a  #表示使用的不是重新定义的a，而是在当前函数外查找到的上一层函数的a
        a = 2        #对a进行赋值，直接影响外部函数的a
        print("inner:",a)
    inner()
    print("outer:",a)

outer()
```

如果全局变量找不到对应参数名，就会查看内置函数名称

```
# def str():                  #查找位置3
#     print("my str")

#str = "global"               #查找位置3
def outer():
    #str = "outer"            #查找位置2
    def inner():
        #str = "inner"        #查找位置1
        print("inner:",str)
    inner()
    print("outer:",str)       #查找位置4：内置函数

outer()
```

3.8 内置模块

学习目标：

对于模块及其功能有所了解，通过查询文档能够完成对应功能。

学习方法建议：

- 【有】其他编程语言学习经验的：快速浏览文档，完成对应练习；
- 【无】其他编程语言学习经验的：将文档中的每个函数抄写并运行一遍；制作思维导图(标注模块及函数功能)

常用模块：

数学模块-math	随机模块-random	时间模块-time	日历模块-calendar
操作系统模块-os	路径模块-os.path	文件操作模块-shutil	序列化模块-pickle
压缩模块-zipfile	压缩模块-tarfile	日志模块-logging	json模块

3.8.1数学模块-math

该模块提供了对C标准定义的数学函数的访问。

> **注意：**
>
> 这些函数不适用于复数；如果你需要计算复数，请使用 `cmath` 模块中的同名函数。
>
> 该模块提供了以下部分函数。除非另有明确说明，否则所有返回值均为浮点数。

- ceil() 向上取整操作 (对比内置round)
- floor() 向下取整操作 (对比内置round)
- pow() 计算一个数值的N次方(结果为浮点数) (对比内置pow)
- sqrt() 开平方运算(结果浮点数)
- fabs() 计算一个数值的绝对值 (结果浮点数) (对比内置abs)
- modf() 将一个数值拆分为整数和小数两部分组成元组
- copysign() 将参数第二个数值的正负号拷贝给第一个
- fsum() 将一个容器数据中的数据进行求和运算 (结果浮点数)(对比内置sum)
- 圆周率常数 pi

```
#ceil()  向上取整操作 (对比内置round)
import math
res = math.ceil(4.01)
print("math.ceil(4.01):",res)                #输出：math.ceil(4.01): 5

#floor() 向下取整操作 (对比内置round)
res = math.floor(3.99)
print("math.floor(3.99):",res)               #输出：math.floor(3.99): 3

#pow()  计算一个数值的N次方(结果为浮点数) (对比内置pow)
res = math.pow(2,3)
print("math.pow(2,3):",res)                  #输出：math.pow(2,3): 8.0
# math模块中的pow没有第三个参数
# res = math.pow(2,3,3) error
# print(res)

#sqrt() 开平方运算(结果浮点数)
res = math.sqrt(10)
print("math.sqrt(10):",res)                  #输出：math.sqrt(10):
3.1622776601683795

#fabs() 计算一个数值的绝对值 (结果浮点数) (对比内置abs)
res = math.fabs(-56)
print("math.fabs(-56):",res)                 #输出：math.fabs(-56): 56.0

#modf() 将一个数值拆分为整数和小数两部分组成元组
res = math.modf(14.677)
```

```
print("math.modf(14.677):",res)      #输出：math.modf(14.677): (0.6769999999999996,
14.0)

#copysign()  将参数第二个数值的正负号拷贝给第一个
res = math.copysign(-1,-5)                    # 得到浮点数结果 ， 它的正负号取决于第二个值
print("math.copysign(-1,-5):",res)            #输出：math.copysign(-1,-5): -1.0

#fsum() 将一个容器数据中的数据进行求和运算 (结果浮点数)(对比内置sum)
listvar = [1,2,3,4,5,99,6]
res = math.fsum(listvar)
print("math.fsum([1,2,3,4,5,99,6]):",res)    #输出：math.fsum([1,2,3,4,5,99,6]):
120.0

#圆周率常数 pi
print("math.pi:",math.pi)                     #输出：math.pi: 3.141592653589793
print("math.pi:","{:.2f}".format(math.pi))    #输出：math.pi: 3.14
```

3.8.2 随机模块-random

该模块实现了各种分布的伪随机数生成器。

警告: 不应将此模块的伪随机生成器用于安全目的。

- random() 获取随机0-1之间的小数(左闭右开)
- randrange() 随机获取指定范围内的整数(包含开始值,不包含结束值,间隔值)
- randint() 随机产生指定范围内的随机整数
- uniform() 获取指定范围内的随机小数(左闭右开)
- choice() 随机获取序列中的值(多选一)
- sample() 随机获取序列中的值(多选多) [返回列表]
- shuffle() 随机打乱序列中的值(直接打乱原序列)

```
import  random
#random() 获取随机0-1之间的小数(左闭右开)  0<= x < 1
res = random.random()
print("random.random():",res)            #输出：random.random():
0.48928937792346106

#randrange() 随机获取指定范围内的整数(包含开始值,不包含结束值,间隔值)
res = random.randrange(3)  # 0~2
print("random.randrange(3):",res)        #输出：random.randrange(3): 1
res = random.randrange(1,10) # 1~9
print("random.randrange(1,10):",res)     #输出：random.randrange(1,10): 4
res = random.randrange(1,10,3) # 1 4 7
print("random.randrange(1,10,3):",res)   #输出：random.randrange(1,10,3): 4

#randint()   随机产生指定范围内的随机整数 (目前唯一可以取到最大值的函数)
res = random.randint(2,5) # 2 3 4 5 最大值可以取到
print("random.randint(2,5) :",res)       #输出：random.randint(2,5) : 5

#randint 必须给2个参数 1个或者3个都不行,功能有局限性
# res = random.randint(2,5,2)        #error
# print(res)
```

```
#uniform() 获取指定范围内的随机小数(左闭右开)
res = random.uniform(1,10) # 1<= x < 10  即：[1,10）
print("random.uniform(1,10):",res)      #输出：random.uniform(1,10):
7.200697101569941
res = random.uniform(5,-3)
print("random.uniform(5,-3):",res)      #输出：random.uniform(5,-3):
-1.4349427894375966
'''
a = 5
b = -3
return a + (b-a) * self.random()
5+ (-3-5) * (0<=x<1)
5+ (-8) * (0<=x<1)
最大值 5
最小值 -3
-3 < x <=5
'''


#choice()  随机获取序列中的值(多选一)
listvar = [1,2,3,90,6,5]
res = random.choice(listvar)
print("random.choice([1,2,3,90,6,5]):",res) #输出：random.choice([1,2,3,90,6,5]):
1


#sample()  随机获取序列中的值(多选多) [返回列表]
# sample(容器类型数据,选几个)
listvar = ["金城武","彭于晏","吴彦祖","胡歌","王力宏","李易峰","丁真"]
res = random.sample(listvar,2)
print("random.sample(listvar,2):",res)      #输出：random.sample(listvar,2): ['胡
歌', '丁真']


#shuffle() 随机打乱序列中的值(直接打乱原序列)
listvar = ["金城武","彭于晏","吴彦祖","胡歌","王力宏","李易峰","丁真"]
random.shuffle(listvar)
print("random.shuffle(listvar):",listvar)
#输出：random.shuffle(listvar): ['彭于晏', '李易峰', '王力宏', '吴彦祖', '丁真', '胡
歌', '金城武']
```

练习：实现一个验证码功能 每次随机产生5个字符，包含字母范围：a-z A-Z 0-9

```
def yanzhengma():
   strvar = ''
   for i in range(5):
      # res = chr(97)
      # a-z 范围的小写字母
      a_z = chr(random.randrange(97,123))
      # A-Z 产生所有的大写字母 65 => A  90
      A_Z = chr(random.randrange(65,91))
      # 0-9 产生0-9 10个数字
      num  = str(random.randrange(0,10)) # 为了实现字符串的拼接
      # 把范围的可能出现的字符放到同一的列表中进行随机挑选
```

```
        listvar = [a_z,A_Z,num]
        # 把选好的5个随机字符 通过+来形成拼接
        strvar += random.choice(listvar)
    # 直接返回该字符串
    return strvar
res = yanzhengma()
print("验证码：",res)          #输出：验证码：  q0h27
```

3.8.3时间模块-time

该模块提供了各种时间相关的函数。相关功能还可以参阅 `datetime` 和 `calendar` 模块。

注意：

尽管此模块始终可用，但并非所有平台上都提供所有功能。 此模块中定义的大多数函数是调用了所在平台 C 语言库的同名函数。 因为这些函数的语义因平台而异,所以使用时最好查阅平台相关文档。

- time()　获取本地时间戳
- mktime()　通过[时间元组]获取[时间戳] (参数是时间元组)
- localtime()　通过[时间戳]获取[时间元组] (默认当前时间)
- ctime()　通过[时间戳]获取[时间字符串] (默认当前时间)
- asctime()　通过[时间元组]获取
- strftime()　通过[时间元组]格式化[时间字符串] (格式化字符串,[可选时间元组参数])
- strptime()　通过[时间字符串]提取出[时间元组] (时间字符串,格式化字符串)
- sleep()　程序睡眠等待
- perf_counter() 用于计算程序运行的时间

```
#time()       获取本地时间戳
import time
res = time.time()
print("time.time():",res)        #输出：time.time(): 1607001227.949147

#mktime()         通过[时间元组]获取[时间戳] (参数是时间元组)
ttl = (2020,2,20,15,45,0,0,0,0)
res = time.mktime(ttl)
print("time.mktime(ttl):",res)  #输出：time.mktime(ttl): 1582184700.0

#localtime()      通过[时间戳]获取[时间元组] (默认当前时间)
ttl = time.localtime()  # 默认使用当前时间戳
print("time.localtime() :",ttl)
#输出：time.localtime() : time.struct_time(tm_year=2020, tm_mon=12, tm_mday=3,
tm_hour=21, tm_min=15, tm_sec=16, tm_wday=3, tm_yday=338, tm_isdst=0)
ttl = time.localtime(1582184700) # 自定义时间戳,转化为时间元组
print("time.localtime(1582184700)",ttl)
#输出：time.localtime(1582184700) time.struct_time(tm_year=2020, tm_mon=2,
tm_mday=20, tm_hour=15, tm_min=45, tm_sec=0, tm_wday=3, tm_yday=51, tm_isdst=0)
'''
time.struct_time(
tm_year=2020,
tm_mon=2,
tm_mday=20,
tm_hour=15,
tm_min=45,
```

```
tm_sec=0,
tm_wday=3,    0~ 6 0代表周一 6 代表周日
tm_yday=51,
tm_isdst=0)
'''
#ctime()          通过[时间戳]获取[时间字符串] (默认当前时间)
res = time.ctime()                # 默认使用当前时间戳
print("time.ctime():",res)        #输出：time.ctime(): Thu Dec  3 21:18:16 2020
res = time.ctime(1582184700)      # 可以手动自定义时间戳
print("time.ctime(1582184700) :",res)   #输出：time.ctime(1582184700) : Thu Feb 20
15:45:00 2020

#asctime()        通过[时间元组]获取[时间字符串](参数是时间元组)
# ttl = (2020,2,20,15,45,0,1,0,0)
# res = time.asctime(ttl)
# print("time.asctime(ttl):",res) #输出:time.asctime(ttl): Tue Feb 20 15:45:00
2020

# 优化版 (推荐使用)
ttl = (2020,2,20,15,45,0,4,0,0)
res = time.mktime(ttl)
print(time.ctime(res))       #输出：Thu Feb 20 15:45:00 2020

#strftime()       通过[时间元组]格式化[时间字符串]  (格式化字符串,[可选时间元组参数])
res = time.strftime("%Y-%m-%d %H:%M:%S") # 默认以当前时间戳转化为字符串
print('time.strftime("%Y-%m-%d %H:%M:%S"):',res)
#输出：time.strftime("%Y-%m-%d %H:%M:%S"): 2020-12-03 21:22:42

# linux当中 strftime可以识别中文,windows不行
res = time.strftime("%Y-%m-%d %H:%M:%S",(2008,8,8,8,8,8,0,0,0))
print('time.strftime("%Y-%m-%d %H:%M:%S",(2008,8,8,8,8,8,0,0,0)):',res)
#输出：time.strftime("%Y-%m-%d %H:%M:%S",(2008,8,8,8,8,8,0,0,0)): 2008-08-08
08:08:08


#strptime()       通过[时间字符串]提取出[时间元组]  (时间字符串,格式化字符串)
# 注意:左右两侧的字符串要严丝合缝,有多余的空格都不行,然后按照次序,依次通过格式化占位符,提取时间
res = time.strptime("2019年3月8号15点21分30秒,发射了人造卫星嫦娥"  ,  "%Y年%m月%d号%H
点%M分%S秒,发射了人造卫星嫦娥")
print(res)
'''#输出:
time.struct_time(
tm_year=2019,
tm_mon=3,
 tm_mday=8,
 tm_hour=15,
 tm_min=21,
 tm_sec=30,
 tm_wday=4,
 tm_yday=67,
 tm_isdst=-1)

'''
#sleep()          程序睡眠等待
# time.sleep(30)
# print(11223344)

#perf_counter()  用于计算程序运行的时间
```

```
startime = time.perf_counter()
print("开始时间：",startime)
for i in range(100000000):  #输出：开始时间：  0.0696103
    pass
endtime = time.perf_counter()
# 结束时间  - 开始时间[ time.time()] 也可以实现;
res = endtime - startime
print("共计用时：",res)      #输出：共计用时：  3.8202432
```

时间模块相关知识

时间戳指从1970年1月1日0时0分0秒到指定时间之间的秒数,时间戳是秒,可以使用到2038年的某一天
UTC时间: 世界约定的时间表示方式，世界统一时间格式，世界协调时间（以前称为格林威治标准时间，或GMT）
夏令时：在夏令时时间状态下，时间会调快1个小时

时间元组是使用元祖格式表示时间的一种方式

- 格式1(自定义):
  (年，月，日，时，分,秒，周几，一年中的第几天，是否是夏令时时间)
- 格式2(系统提供):
  (tm_year = 年，tm_month = 月，tm_day = 日，tm _hour = 时， tm_min = 分, tm _sec = 秒，tm _wday = 周几， tm _yday = 一年中的第几天，tm_isdst = 是否是夏令时时间)

索引（Index）	属性（Attribute）	值（Values）
0	tm_year（年）	例如:2020
1	tm_mon（月）	范围：[1, 12]
2	tm_mday（日）	范围：[1, 31]
3	tm_hour（时）	范围：[0, 23]
4	tm_min（分）	范围：[0, 59]
5	tm_sec（秒）	范围：[0, 61]
6	tm_wday（weekday）	范围：[0, 6]，周一为 0
7	tm_yday（一年中的第几天）	范围：[1, 366]
8	tm_isdst（是否是夏令时）	0, 1 或 -1；默认为-1

格式化时间字符串:

格式	含义	备注
%a	本地（locale）简化星期名称	
%A	本地完整星期名称	
%b	本地简化月份名称	
%B	本地完整月份名称	
%c	本地相应的日期和时间表示	
%d	一个月中的第几天（01 - 31）	
%H	一天中的第几个小时（24小时制，00 - 23）	
%I	第几个小时（12小时制，01 - 12）	
%j	一年中的第几天（001 - 366）	
%m	月份（01 - 12）	
%M	分钟数（00 - 59）	
%p	本地am或者pm的相应符	1
%S	秒（01 - 61）	2
%U	一年中的星期数。（00 - 53星期天是一个星期的开始。）第一个星期天之前的所有天数都放在第0周。	3
%w	一个星期中的第几天（0 - 6，0是星期天）	3
%W	和%U基本相同，不同的是%W以星期一为一个星期的开始。	
%x	本地相应日期	
%X	本地相应时间	
%y	去掉世纪的年份（00 - 99）	
%Y	完整的年份	
%Z	时区的名字（如果不存在为空字符）	
%%	‘%’字符	

1. “%p”只有与“%I”配合使用才有效果。
2. 文档中强调确实是0 - 61，而不是59，闰年秒占两秒。
3. 当使用strptime()函数时，只有当在这年中的周数和天数被确定的时候%U和%W才会被计算。

不常用的属性函数(了解)

*gmtime()　　获取UTC时间元祖(世界标准时间)
*time.timezone　获取当前时区(时区的时间差)
*time.altzone　获取当前时区(夏令时)
*time.daylight　获取夏令时状态

3.8.4 日历模块-calendar(了解)

这个模块让你可以输出像 Unix **cal** 那样的日历，它还提供了其它与日历相关的实用函数。 默认情况下，这些日历把星期一当作一周的第一天，星期天为一周的最后一天（按照欧洲惯例）。

```
#calendar() 获取指定年份的日历字符串 (年份,w日期间的宽度,l日期间的高度,c月份间的间距,m一行显示几个月)
res = calendar.calendar(2020,w=2,l=2,c=20,m=3)
print(res)

#month() 获取指定年月的日历字符串 (年份,月份,w日期之间的宽度,l日期之间的高度)
res = calendar.month(2020,12,w = 2,l = 2)
print(res)

#monthcalendar() 获取指定年月的信息列表 (年份,月份) 0从周一开始排
res = calendar.monthcalendar(2020,12)
print(res)
#输出：[[0, 1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12, 13], [14, 15, 16, 17, 18, 19, 20], [21, 22, 23, 24, 25, 26, 27], [28, 29, 30, 31, 0, 0, 0]]

#isleap() 检测是否是闰年(能被4整除不能被100整除或能被400整除)
res = calendar.isleap(2020)
print(res)                    #输出：True

#leapdays() 指定从某年到某年范围内的闰年个数
res = calendar.leapdays(1970,2038)
print(res)                    #输出：17

#monthrange() 获取某年某月的信息 返回：（第一天是星期几，共多少天）其中周一是0
res = calendar.monthrange(2020,10)
print(res)                    #输出：(3, 31)

#weekday() 指定某年某月某日是星期几
res = calendar.weekday(2020,12,25)
print(res)                    #输出：4

#timegm() 将时间元组转化为时间戳
ttp = (2020,12,25,15,45,35,0,0,0)
res = calendar.timegm(ttp)
print(res)                    #输出：1608911135
```

3.8.5 序列化模块-pickle

模块 `pickle` 实现了对一个 Python 对象结构的二进制序列化和反序列化。

- 序列化：将对象变成二进制（字节流）形式，用于文件存储或网络传输
- 反序列化：将二进制形式的对象，变为内存中可以被程序运行的对象

> 警告
>
> `pickle` 模块**并不安全**。你只应该对你信任的数据进行unpickle操作。

- dumps 把任意对象序列化成一个bytes
- loads 把任意bytes反序列化成原来数据
- dump 把对象序列化后写入到file-like Object(即文件对象)
- load 把file-like Object(即文件对象)中的内容拿出来,反序列化成原来数据

```
import pickle
# dumps 把任意对象序列化成一个bytes
dic = {"a":1,"b":2}
res = pickle.dumps(dic)
print(res)
#输出:
b'\x80\x04\x95\x11\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x01a\x94K\x01\x8c\x01b\
x94K\x02u.'

# loads 把任意bytes反序列化成原来数据
res = pickle.loads(res)
print(res,type(res))            #输出:{'a': 1, 'b': 2} <class 'dict'>

# 函数可以序列化么?  可以的。
def func():
    print("我是一个函数")
res = pickle.dumps(func)
print("函数序列化后:",res)
res = pickle.loads(res)
print("反序列化后的函数对象调用结果:")
res()

# 迭代器可以序列化么? 可以的。
from collections.abc import Iterator
it = iter(range(10))
print(isinstance(it,Iterator))  #输出:True
res = pickle.dumps(it)
print("序列化后的迭代器对象:",res)
#输出:序列化后的迭代器对象:
b'\x80\x04\x95;\x00\x00\x00\x00\x00\x00\x00\x8c\x08builtins\x94\x8c\x04iter\x94\
x93\x94\x8c\x08builtins\x94\x8c\x05range\x94\x93\x94K\x00K\nK\x01\x87\x94R\x94\x
85\x94R\x94K\x00b.'
res = pickle.loads(res)
print(res)                      #输出:<range_iterator object at
0x000002AB3944B0D0>
for i in range(3):
    print(next(res))

# 所有的数据类型都可以通过pickle进行序列化

# dump  把对象序列化后写入到file-like Object(即文件对象)
dic = {"a":1,"b":2}
with open("content.txt",mode="wb") as fp:
    # pickle.dump(数据对象,文件对象) 先把数据变成二进制字节流 在存储在文件当中
    pickle.dump(dic,fp)

# load  把file-like Object(即文件对象)中的内容拿出来,反序列化成原来数据
with open("content.txt",mode="rb") as fp:
    res = pickle.load(fp)
print(res)                      #输出:{'a': 1, 'b': 2}
```

3.8.6 json模块

JSON (JavaScript Object Notation)，形式上是Javascript对象的表达方式。因为格式简单便于存储和传输，所以形成了目前网络传输的主要格式。

所有编程语言都能够识别的数据格式，本质上是字符串。

- dumps 把任意对象序列化成一个str
- loads 把任意str反序列化成原来数据
- dump 把对象序列化后写入到file-like Object(即文件对象)
- load 把file-like Object(即文件对象)中的内容拿出来,反序列化成原来数据

```
# 第一对 dumps 和 loads 把数据序列化或者反序列化成字符串
'''
ensure_ascii=True (默认值) 如果想要显示中文 如下:ensure_ascii = False
#输出保证将所有输入的非 ASCII 字符转义。如果 ensure_ascii 是 false，这些字符会原样输出。
sort_keys=False(默认值)  不对字典的键进行排序 (按照ascii 字符的从小到大进行排序)
'''

import json

dic = {"name": "咯咯", "age": 3, "sex": "雄性", "breed": ["ChaiQuan", "柴犬"],
"weight": 10}
res = json.dumps(dic, ensure_ascii=True, sort_keys=True)
print(res, type(res))
'''
{
"name": "\u54af\u54af",
"age": 3,
"sex": "\u96c4\u6027",
"family": ["father", "\u5988\u5988"]},
"weight": 10
<class 'str'>
'''
# 第二对 dump  和 load  应用在数据的存储的转化上
dic = {"name": "咯咯", "age": 3, "sex": "雄性", "breed": ["ChaiQuan", "柴犬"],
"weight": 10}
with open("content2.txt", mode="w", encoding="utf-8") as fp:
    json.dump(dic, fp, ensure_ascii=False)

with open("content2.txt", mode="r", encoding="utf-8") as fp:
    res = json.load(fp)
print(res, type(res))
```

json 和 pickle 两个模块的区别:

- (1)json序列化之后的数据类型是str,**所有编程语言都识别**,
  但是仅限于int float bool str list tuple dict None [不包含complex set]
  json不能连续load,只能一次性拿出所有数据

- (2)pickle序列化之后的数据类型是bytes,
  所有数据类型都可转化,但仅限于python之间的存储传输.
  pickle可以连续load,多套数据放到同一个文件中

```
# pickle  和 json 之间的用法区别
'''
json 可以连续dump , 但是不能连续load , load是一次性拿出所有数据而不能识别.
可以使用loads ,一行一行的读取,一行一行的通过loads来转化成原有数据类型
'''
# (1) json
dic = {"name": "咯咯", "age": 3, "sex": "雄性", "breed": ["ChaiQuan", "柴犬"],
"weight": 10}
with open("content3.txt", mode="w", encoding="utf-8") as fp:
    json.dump(dic, fp)
    fp.write('\n')
    json.dump(dic, fp)
    fp.write('\n')

print("【JSON从文件中反序列化】")
with open("content3.txt", mode="r", encoding="utf-8") as fp:
    # load 是一次性把所有的数据拿出来,进行识别
    # load 不能识别多个数据混在一起的情况
    # 用loads 来解决load 不能识别多个数据的情况
    # res = json.load(fp)
    # print(res)

    for i in fp:
        # print(i,type(i))
        res = json.loads(i)
        print(res, type(res))


# (2) pickle
dic = {"name": "咯咯", "age": 3, "sex": "雄性", "breed": ["ChaiQuan", "柴犬"],
"weight": 10}
import pickle

with open("content4.txt", mode="wb") as fp:
    pickle.dump(dic, fp)
    pickle.dump(dic, fp)
    pickle.dump(dic, fp)
    pickle.dump(dic, fp)
print("【Pickle从文件中反序列化】")
with open("content4.txt", mode="rb") as fp:
    # res = pickle.load(fp)
    # print(res)
    # res = pickle.load(fp)
    # print(res)
    # res = pickle.load(fp)
    # print(res)
    # res = pickle.load(fp)
    # print(res)

    try:
        while True:
            res = pickle.load(fp)
            print(res)
```

```
    except:
        pass
```

> try:
> ...
> except:
> ...
> 把有问题的代码赛到try 代码块当中
> 如果发生异常报错,直接执行except其中的代码块
> 优点:不会因为报错终止程序运行

```
try:
    listvar = [1,2]
    print(listvar[15])
except:
    pass
```

3.8.7 操作系统模块-os

os模块提供了使用操作系统相关功能的函数和属性。

- system() 在python中执行系统命令
- popen() 执行系统命令返回对象,通过read方法读出字符串
- listdir() 获取指定文件夹中所有内容的名称列表
- getcwd() 获取当前文件所在的默认路径
- chdir() 修改当前文件工作的默认路径
- environ 获取或修改环境变量
- os 模块属性
    - name 获取系统标识 linux,mac ->posix windows -> nt
    - sep 获取路径分割符号 linux,mac -> / window-> \
    - linesep 获取系统的换行符号 linux,mac -> \n window->\r\n 或 \n

功能	属性或函数(省略了形式参数)
【增】创建目录(文件夹)	os.mkdir()
【增】复制文件权限和内容	shutil.copy()
【删】删除目录(文件夹)	os.rmdir()
【改】对文件,目录重命名	os.rename()
【查】获取当前文件所在的默认路径	os.getcwd()
【查】获取指定文件夹中所有内容的名称列表	os.listdir()
【查】获取环境变量	os.environ
【查】获取系统标识	os.name
【查】获取路径分割符号	os.sep
【查】获取系统的换行符号	os.linesep
【执行】相当于在命令提示符窗口中执行指令	os.system("指令")

```
import os

#相当于在命令提示符窗口中执行指令
os.system("ipconfig")        #显示IP地址信息（在控制台显示乱码是正常的）
os.system('calc')            #打开计算器
os.system("mspaint")         #打开画图软件

# popen 可以把运行的结果,这个字符串转化成utf-8这样的编码格式在进行输出
res = os.popen("ipconfig").read()
print(res)

#listdir() 获取指定文件夹中所有内容的名称列表
lst = os.listdir(r"D:\itsishu\python_workspace\demo8")
print(lst)          #输出：['.idea', 'demo1.py', 'demo2.py', 'demo3.py',
'demo4.py', 'demo5.py', 'demo6.py', 'demo7.py']

#getcwd()  获取当前文件所在的默认路径
res = os.getcwd()
print(res)                  #输出：D:\itsishu\python_workspace\demo8

#chdir()   修改当前文件工作的默认路径
#chdir()   修改当前文件工作的默认路径
os.chdir("D:\itsishu\python_workspace\\demo8\\test")
res = os.getcwd()
print("修改后的工作路径：",res)

# os.mkdir   创建目录(文件夹)
os.mkdir("new_folder")            #当前目录下出现new_folder文件夹

#os.rmdir   删除目录(文件夹)
os.rmdir("new_folder")          #当前目录下new_folder被删除
```

```
#os.rename  对文件,目录重命名
os.rename("new_folder","rename_folder") #将new_folder文件夹更名为rename_folder
#如果没有找到对应的文件夹，会报错。FileNotFoundError: [WinError 2] 系统找不到指定的文件。: 'new_folder' -> 'rename_folder'
#如果重命名后和当前文件名冲突，会报错：FileExistsError: [WinError 183] 当文件已存在时，无法创建该文件。: 'new_folder' -> 'rename_folder'

# copy(src,dst)        #复制文件权限和内容
# . 当前路径 .. 代表上一级路径 ../../c.txt 代表上一级的上一级
#  把当前路径的a.txt 复制到 上一级的b.txt文件中
import shutil
shutil.copy("a.txt","../b.txt")      #文件所在位置为:D:\itsishu\python_workspace\b.txt

#environ   获取或修改环境变量
res = os.environ
print(res)
```

- **os 模块属性**

```
#name 获取系统标识   linux,mac ->posix      windows -> nt
print(os.name)
#sep 获取路径分割符号  linux,mac -> /       window-> \  *****
print(os.sep)  # /home/wangwen/abc  \home\wangwen #error
#linesep 获取系统的换行符号  linux,mac -> \n    window->\r\n 或 \n
print(repr(os.linesep))
```

练习：计算一个文件夹当中所有文件的大小

方法1，使用for循环：

```
import os

pathvar = r"D:\itsishu\python_workspace\demo8"
lst = os.listdir(pathvar)
print(lst)

# (1)计算文件大小
size = 0
for i in lst:
    # print(i)
    # "D:\itsishu\python_workspace\demo8"  + a.txt => 通过join 拼接新的路径
    #  "D:\itsishu\python_workspace\demo8\a.txt"
    # 拼接路径  用来判断是文件夹还是问价
    pathvar2 = os.path.join(pathvar, i)
    # 如果是文件夹 执行相应逻辑
    if os.path.isdir(pathvar2):
        print(i + "是[目录]")
    # 如果是文件，计算文件大小
    elif os.path.isfile(pathvar2):
        print(i + "是[文件]")
        # os.path.getsize 只能计算文件的大小
        # 用size 这个变量不停的累加文件大小
```

```
        size += os.path.getsize(pathvar2)

print(size)
```

方法2，使用递归：

```
pathvar = r"D:\itsishu\python_workspace\demo8"

def getallsize(pathvar):
    size = 0
    lst = os.listdir(pathvar)
    print(lst)  # ['a.txt', 'b.txt', 'test']
    '''
    #pathvar2 = os.path.join(pathvar,i)
        D:\itsishu\python_workspace\demo8\a.txt
        D:\itsishu\python_workspace\demo8\b.txt
        D:\itsishu\python_workspace\demo8\test
    '''
    for i in lst:
        pathvar2 = os.path.join(pathvar, i)
        if os.path.isdir(pathvar2):
            # 模拟鼠标点击进去的动作
            '''size += getallsize(pathvar2) 是和上一个代码唯一不一样的地方;'''
            size += getallsize(pathvar2)
            print("是文件夹")
        elif os.path.isfile(pathvar2):
            size += os.path.getsize(pathvar2)
    return size

res = getallsize(pathvar)
print(res)
```

3.8.8 路径模块 -os.path

- abspath() 将相对路径转化为绝对路径
- basename() 返回文件名部分
- dirname() 返回路径部分
- split() 将路径拆分成单独的文件部分和路径部分 组合成一个元组
- join() 将多个路径和文件组成新的路径 可以自动通过不同的系统加不同的斜杠 linux / windows\
- splitext() 将路径分割为后缀和其他部分
- getsize() 获取文件的大小
- isdir() 检测路径是否是一个文件夹
- isfile() 检测路径是否是一个文件
- islink() 检测路径是否是一个链接
- getctime() [windows]文件的创建时间,[linux]权限的改动时间(返回时间戳)
- getmtime() 获取文件最后一次修改时间(返回时间戳)
- getatime() 获取文件最后一次访问时间(返回时间戳)
- exists() 检测指定的路径是否存在
- isabs() 检测一个路径是否是绝对路径

> . 或者.. 代表的是相对路径
> d: e: f: 这个具体的路径代表绝对路径
> 在linux里面 /开头的就是绝对路径 .或者..代表相对路径

```
import os
#abspath()  将相对路径转化为绝对路径
res = os.path.abspath(".")  #输出：D:\itsishu\python_workspace\demo8
print(res)

#basename() 返回文件名部分
strvar = r"D:\itsishu\python_workspace\aa.txt"
res = os.path.basename(strvar)
print(res)                  #输出：aa.txt

#dirname()  返回路径部分
res = os.path.dirname(strvar)
print(res)                  #输出：D:\itsishu\python_workspace

#split() 将路径拆分成单独的文件部分和路径部分 组合成一个元组
res = os.path.split(strvar)
print(res)                  #输出：('D:\\itsishu\\python_workspace', 'aa.txt')

#join()  将多个路径和文件组成新的路径
path1 = "home"
path2 = "itsishu"
path3 = "myproject"
#可以自动通过不同的系统加不同的斜杠  linux / windows \
res = os.path.join(path1,path2,path3)
print(res)                  #输出：home\itsishu\myproject

#splitext() 将路径分割为后缀和其他部分
# 通过splitext 可以快速拿到一个路径中的后缀
strvar = r"D:\itsishu\python_workspace\aa.txt"
res = os.path.splitext(strvar)
print(res)                  #输出：('D:\\itsishu\\python_workspace\\aa', '.txt')

#getsize()  获取文件的大小
pathvar = r"D:\itsishu\python_workspace\demo8\demo8.py"
res = os.path.getsize(pathvar)
print(res)                  #输出：2870

#isdir()    检测路径是否是一个文件夹(路径)
res = os.path.isdir(r'D:\itsishu\python_workspace\demo8')
print(res)                  #输出：True
#isfile()   检测路径是否是一个文件
res = os.path.isfile(r'D:\itsishu\python_workspace\b.txt')
print(res)                   #True

#getctime() [windows]文件的创建时间,[linux]权限的改动时间(返回时间戳)
res = os.path.getctime(r"D:\itsishu\python_workspace\b.txt")
print(res)
import time
res = time.ctime(1607311034)
print(res)                  #输出：Mon Dec  7 11:17:14 2020
```

```
#getmtime() 获取文件最后一次修改时间(返回时间戳)
res = os.path.getmtime(r"D:\itsishu\python_workspace\b.txt")
res = time.ctime(res)
print(res)                    #输出：Mon Dec  7 11:17:14 2020

#getatime() 获取文件最后一次访问时间(返回时间戳) # 存在系统差异
res = os.path.getatime(r"D:\itsishu\python_workspace\b.txt")
res = time.ctime(res)
print(res)                    #输出：Mon Dec  7 11:17:14 2020

#exists()   检测指定的路径是否存在
res = os.path.exists(r"D:\itsishu\python_workspace\b.txt")
print(res)                    #输出：True

#isabs()    检测一个路径是否是绝对路径 (了解)
strvar  =  "."
# strvar = r"D:\itsishu\python_workspace\b.txt"
res = os.path.isabs(strvar)
print(res)                    #输出：False
```

3.8.9 文件操作模块-shutil

shutil模块提供了一系列对文件和文件集合的高阶操作。 特别是提供了一些支持文件拷贝和删除的函数。

【 shutil模块 复制/移动/】

- copyfileobj(fsrc, fdst[, length=16*1024]) 复制文件 (length的单位是字符(表达一次读多少字符))
- copyfile(src,dst) #单纯的仅复制文件内容 , 底层调用了 copyfileobj
- copymode(src,dst) #单纯的仅复制文件权限 , 不包括内容 (虚拟机共享目录都是默认777)
- copystat(src,dst) #复制所有状态信息,包括权限，组，用户，修改时间等,不包括内容
- copy(src,dst) #复制文件权限和内容
- copy2(src,dst) #复制文件权限和内容,还包括权限，组，用户，时间等
- copytree(src,dst) #拷贝文件夹里所有内容(递归拷贝)
- rmtree(path) #删除当前文件夹及其中所有内容(递归删除)
- move(path1,paht2) #移动文件或者文件夹

os 与 shutil 模块 都具备对文件的操作

【os模块具有 新建/删除/】

- os.mknod 创建文件
- os.remove 删除文件
- os.mkdir 创建目录(文件夹)
- os.rmdir 删除目录(文件夹)
- os.rename 对文件,目录重命名
- os.makedirs 递归创建文件夹
- os.removedirs 递归删除文件夹（空文件夹）

```
import shutil
```

```
#复制文件权限和内容,还包括权限，组，用户，时间等
shutil.copy2(r'D:\itsishu\python_workspace\demo8\demo6.py',
             r'D:\itsishu\python_workspace\demo8\demo6_backup.py')

# ignore表示“忽视”指定的文件，不进行复制
shutil.copytree("outer",
                "outer2",
                ignore=shutil.ignore_patterns("__init__.py","tmp.txt"))

#删除当前文件夹及其中所有内容(递归删除)
shutil.rmtree(r"D:\itsishu\python_workspace\demo8\outer2")

#移动文件或者文件夹
shutil.move(r"D:\itsishu\python_workspace\demo8",r"D:\itsishu\demo",
copy_function=shutil.copy2)

# 查看硬盘空间情况
total, used, free = shutil.disk_usage("c:\\")
print("当前磁盘共: %iGB, 已使用: %iGB, 剩余: %iGB"%(total / 1073741824, used /
1073741824, free / 1073741824))

# 压缩文件夹
shutil.make_archive('outer', 'zip',r'D:\itsishu\python_workspace\demo8\outer')

#解压缩文件夹
shutil.unpack_archive('outer.zip',r'D:\itsishu\python_workspace\demo8\unzip')
```

3.8.10 压缩模块-zipfile

(后缀为zip)

语法：zipfile.ZipFile(file[, mode[, compression[, allowZip64]]])
ZipFile(路径包名,模式,压缩or打包,可选allowZip64)
功能：创建一个ZipFile对象,表示一个zip文件.
参数：
-参数file表示文件的路径或类文件对象(file-like object)
-参数mode指示打开zip文件的模式，默认值为r
r 表示读取已经存在的zip文件
w 表示新建一个zip文档或覆盖一个已经存在的zip文档
a 表示将数据追加到一个现存的zip文档中。
-参数compression表示在写zip文档时使用的压缩方法
zipfile.ZIP_STORED 只是存储模式，不会对文件进行压缩，这个是默认值
zipfile.ZIP_DEFLATED 对文件进行压缩
-如果要操作的zip文件大小超过2G，应该将allowZip64设置为True。

- 压缩文件
  1.ZipFile() 写模式w打开或者新建压缩文件
  2.write(路径,别名) 向压缩文件中添加文件内容
  3.close() 关闭压缩文件
- 解压文件
  1.ZipFile() 读模式r打开压缩文件
  2.extractall(路径) 解压所有文件到某个路径下

extract(文件,路径)　解压指定的某个文件到某个路径下
3.close()　　关闭压缩文件

- 追加文件(支持with写法)
  ZipFile()　　追加模式a打开压缩文件
- 查看压缩包中的内容
  namelist()

```
import zipfile
# zipfile.ZIP_DEFLATED 对当前文件进行压缩
# 1.压缩文件
# (1)创建一个压缩文件
zf = zipfile.ZipFile("demo8.zip","w",zipfile.ZIP_DEFLATED)
# (2)往压缩包当中存入内容 推荐写入绝对路径
# (路径参数,别名参数)
zf.write(r"D:\itsishu\python_workspace\demo8\demo7.py","demo7.py")
zf.write(r"D:\itsishu\python_workspace\demo8\demo8.py","demo8.py")

# (3)关闭文件
zf.close()

# 2.解压文件
zf = zipfile.ZipFile("demo8.zip","r")
# (1)extractall(路径)　　解压所有文件到某个路径下
zf.extractall(r"D:\itsishu\python_workspace\demo8\demo")

# (2)extract(文件,路径)　　解压指定的某个文件到某个路径下
zf.extract("demo7.py",r"D:\itsishu\python_workspace\demo8\demo\7")
zf.close()

# # (3)追加文件(支持with写法)
with zipfile.ZipFile("demo.zip","a",zipfile.ZIP_DEFLATED) as zf:
    # 可以在write写入的时候 , 动态为该文件创建文件夹 比如 tmp/demo8.py 这个tmp就是内部文件
夹
    zf.write(r"D:\itsishu\python_workspace\demo8\demo8.py","tmp/demo8.py")

# (4)查看压缩包中的内容
with zipfile.ZipFile("demo.zip","r") as zf:
    res = zf.namelist()
print(res)              #输出：['tmp/demo8.py']
```

3.8.11 压缩模块-tarfile(了解)

(后缀为.tar | .tar.gz | .tar.bz2)

bz2模式的压缩文件较小 根据电脑的不同会产生不同的结果 (理论上:bz2压缩之后更小,按实际情况为标准)

w　单纯的套一个后缀 打包
w:bz2 采用bz2算法 压缩
w:gz 采用gz算法 压缩

- 压缩文件
  1.open('路径包名','模式','字符编码') 创建或者打开文件

2.add(路径文件,arcname="别名") 向压缩文件中添加文件
3.close() 关闭文件

- 解压文件
  1.open('路径包名','模式','字符编码') 读模式打开文件
  2.extractall(路径)　解压所有文件到某个路径下
  　extract(文件,路径)　解压指定的某个文件到某个路径下
  3.close()　　关闭压缩文件
- 追加文件
  open()　　追加模式 a: 打开压缩文件 正常添加即可
- 查看压缩包中的内容
  getnames()

3.8.12 日志模块-logging

为什么要写日志?

- 为了排错
- 用来做数据分析的

作用:

1.用来记录用户的行为 - 数据分析
2.用来记录用户的行为 - 操作审计
3.排查代码中的错误

场景举例：购物商城 - 记录到数据库里

> 什么时间购买了什么商品
> 把哪些商品加入购物车了

做数据分析的内容 - 记录到日志中
1.一个用户什么时间在什么地点 登录了购物程序
2.搜索了哪些信息,多长时间被展示出来了
3.什么时候关闭了软件
4.对哪些商品点进去看过

输出内容是有等级的 : 默认处理warning级别以上的所有信息

```
import logging

logging.debug('debug message')          # 调试
logging.info('info message')            # 信息
logging.warning('warning message')      # 警告
logging.error('error message')          # 错误
logging.critical('critical message')    # 批判性的
```

示例:

```
def cal_mul(exp):
    exp = 4*6
    logging.debug('4*6 = 24')           #输出：DEBUG:root:4*6 = 24
    return 24
def cal_div():
    pass
```

```
def cal_add():
    pass
def cal_sub(exp):
    exp = 3-24
    logging.debug('cal_sub :3-24 = 21') #输出：DEBUG:root:cal_sub :3-24 = 21
    return 21

def cal_inner_bracket(exp2):
    exp2 = 3-4*6
    ret = cal_mul(4*6)
    exp2 = 3-24
    ret = cal_sub(3-24)
    logging.debug('3-4*6 = -21')        #输出：DEBUG:root:3-4*6 = -21
    return -21

def main(exp):
    exp =(1+2*(3-4*6))/5
    ret = cal_inner_bracket(3-4*6)
    return ret

logging.basicConfig(level=logging.DEBUG)
ret = main('(1+2*(3-4))/5')
print(ret)                              #输出：-21
```

- 日志中的内容和格式都是自定义的。 通过：logging.basicconfig()

设置日志输出到屏幕的格式：

```
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s[line :%(lineno)d]-%(module)s:
%(message)s',
    datefmt='%Y-%m-%d %H:%M:%S %p',
)
logging.warning('warning message test2')
logging.error('error message test2')
logging.critical('critical message test2')
```

输出结果：

> 2020-12-10 17:22:27 PM - root - WARNING[line :81]-test11: warning message test2
> 2020-12-10 17:22:27 PM - root - ERROR[line :82]-test11: error message test2
> 2020-12-10 17:22:27 PM - root - CRITICAL[line :83]-test11: critical message test2

设置输出到文件,并且设置信息的等级

```
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s[line :%(lineno)d]-%(module)s:
%(message)s',
    datefmt='%Y-%m-%d %H:%M:%S %p',
    filename='tmp.log',
    level= logging.DEBUG

)
logging.debug('debug 信息错误 test2')
logging.info('warning 信息错误 test2')
logging.warning('warning message test2')
logging.error('error message test2')
logging.critical('critical message test2')
```

源码所在文件夹下，tmp.log文件中输出结果：

> 2020-12-10 17:23:31 PM - root - DEBUG[line :93]-test11: debug 信息错误 test2
> 2020-12-10 17:23:31 PM - root - INFO[line :94]-test11: warning 信息错误 test2
> 2020-12-10 17:23:31 PM - root - WARNING[line :95]-test11: warning message test2
> 2020-12-10 17:23:31 PM - root - ERROR[line :96]-test11: error message test2
> 2020-12-10 17:23:31 PM - root - CRITICAL[line :97]-test11: critical message test2

- 同时向多个文件和屏幕上输出，使用字符集保证乱码

```
fh = logging.FileHandler('tmp.log',encoding='utf-8')
fh2 = logging.FileHandler('tmp2.log',encoding='utf-8')
sh = logging.StreamHandler()
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s[line :%(lineno)d]-%(module)s:
%(message)s',
    datefmt='%Y-%m-%d %H:%M:%S %p',
    level= logging.DEBUG,
    handlers=[fh,sh,fh2]
    #handlers=[fh,sh]
)
logging.debug('debug 信息错误 test2')
logging.info('warning 信息错误 test2')
logging.warning('warning message test2')
logging.error('error message test2')
logging.critical('critical message test2')
```

源码所在文件夹下，tmp.log和tmp2.log文件中输出结果：

> 2020-12-10 17:25:51 PM - root - DEBUG[line :114]-test11: debug 信息错误 test2
> 2020-12-10 17:25:51 PM - root - INFO[line :115]-test11: warning 信息错误 test2
> 2020-12-10 17:25:51 PM - root - WARNING[line :116]-test11: warning message test2
> 2020-12-10 17:25:51 PM - root - ERROR[line :117]-test11: error message test2
> 2020-12-10 17:25:51 PM - root - CRITICAL[line :118]-test11: critical message test2

- 日志的切分

```
import time
from logging import handlers
```

```
sh = logging.StreamHandler()
rh = handlers.RotatingFileHandler('myapp.log', maxBytes=1024,backupCount=5)   #
按照大小做切割
fh = handlers.TimedRotatingFileHandler(filename='x2.log', when='s', interval=5,
encoding='utf-8')
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s[line :%(lineno)d]-%(module)s:
%(message)s',
    datefmt='%Y-%m-%d %H:%M:%S %p',
    level= logging.DEBUG,
    handlers=[fh,rh,sh]
)
for i in range(1,100000):
    time.sleep(1)
    logging.error('KeyboardInterrupt error %s'%str(i))
```

随着时间累积会生成多个*myapp.log.数字* 形式的文件，其中内容格式为：

> 2020-12-10 17:27:53 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 1
> 2020-12-10 17:27:54 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 2
> 2020-12-10 17:27:55 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 3
> 2020-12-10 17:27:56 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 4
> 2020-12-10 17:27:57 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 5
> 2020-12-10 17:27:58 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 6
> 2020-12-10 17:27:59 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 7
> 2020-12-10 17:28:00 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 8
> 2020-12-10 17:28:01 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 9
> 2020-12-10 17:28:02 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 10
> 2020-12-10 17:28:03 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 11
> 2020-12-10 17:28:04 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 12

x2.log数量更多，格式形式为：*x2.log.2020-12-10_17-28-32*，内容格式为：

> 2020-12-10 17:28:32 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 40
> 2020-12-10 17:28:33 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 41
> 2020-12-10 17:28:34 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 42
> 2020-12-10 17:28:35 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 43
> 2020-12-10 17:28:36 PM - root - ERROR[line :135]-test11: KeyboardInterrupt error 44

3.9 第三方包

在Python标准库以外还存在成千上万并且不断增加的其他组件 (从单独的程序、模块、软件包直到完整的应用开发框架)，访问 Python 包索引 （https://pypi.org/）即可获取这些第三方包。

3.9.1 pip的升级：

显示pip的当前信息，可以使用命令：

```
pip show pip
```

执行pip升级命令:

```
python -m pip install -U pip
```

3.9.2 pip使用国内源

- 1.**临时使用：**

格式：pip install 库的名称 -i 国内源的网站。例如：

```
pip install scrapy -i https://pypi.tuna.tsinghua.edu.cn/simple
```

国内源可选镜像：

```
    阿里云 https://mirrors.aliyun.com/pypi/simple/
    中国科技大学 https://pypi.mirrors.ustc.edu.cn/simple/
    豆瓣(douban) https://pypi.douban.com/simple/
    清华大学 https://pypi.tuna.tsinghua.edu.cn/simple/
    中国科学技术大学 https://pypi.mirrors.ustc.edu.cn/simple/
```

- 2.**永久修改**

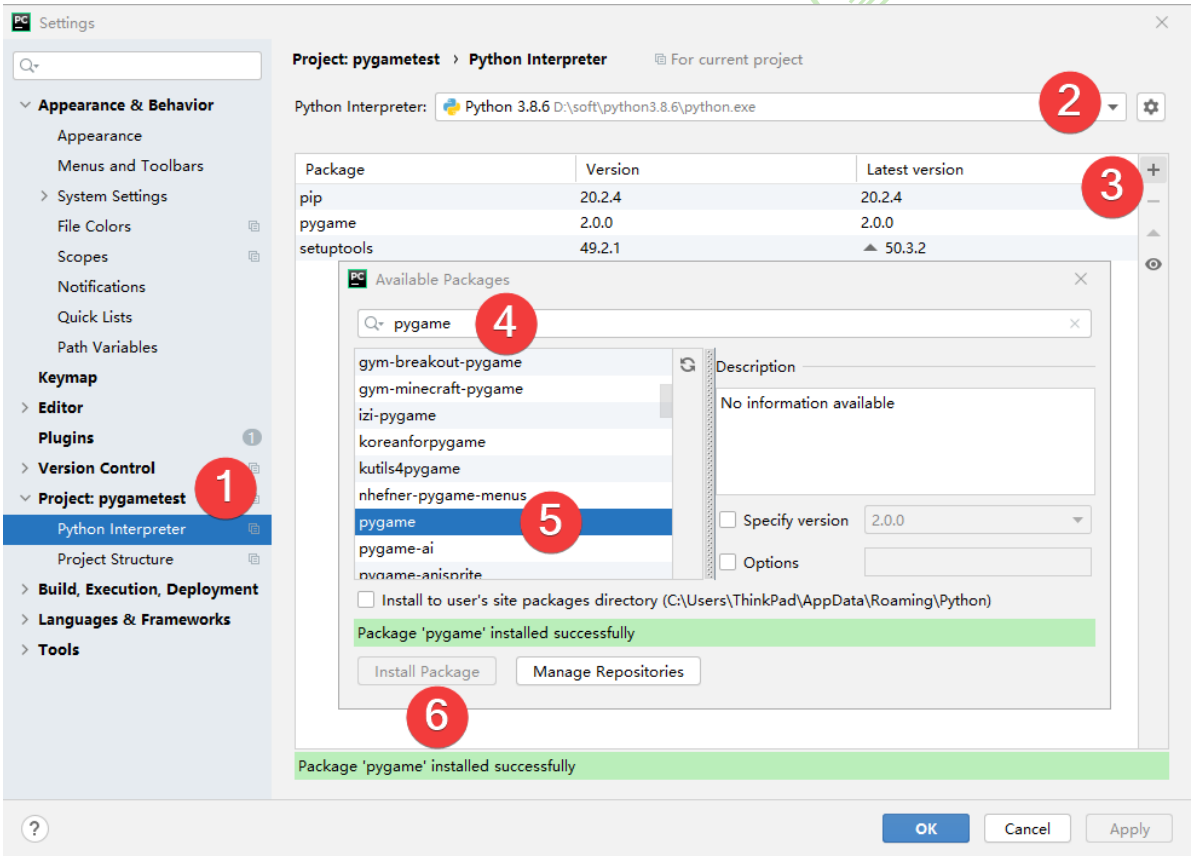
linux:
修改 ~/.pip/pip.conf (没有就创建一个)， 内容如下：

```
[global]
index-url=https://pypi.tuna.tsinghua.edu.cn/simple
```

windows:
直接在user目录中创建一个pip目录，如：C:\Users\xx\pip，新建文件pip.ini，内容如下：

```
[global]
index-url=https://pypi.tuna.tsinghua.edu.cn/simple
```

3.9.3 Pycharm中安装第三方包



1.选择当前项目的Python解释器配置项

2.选择具体想要安装包的解释器

3.点击“+”号

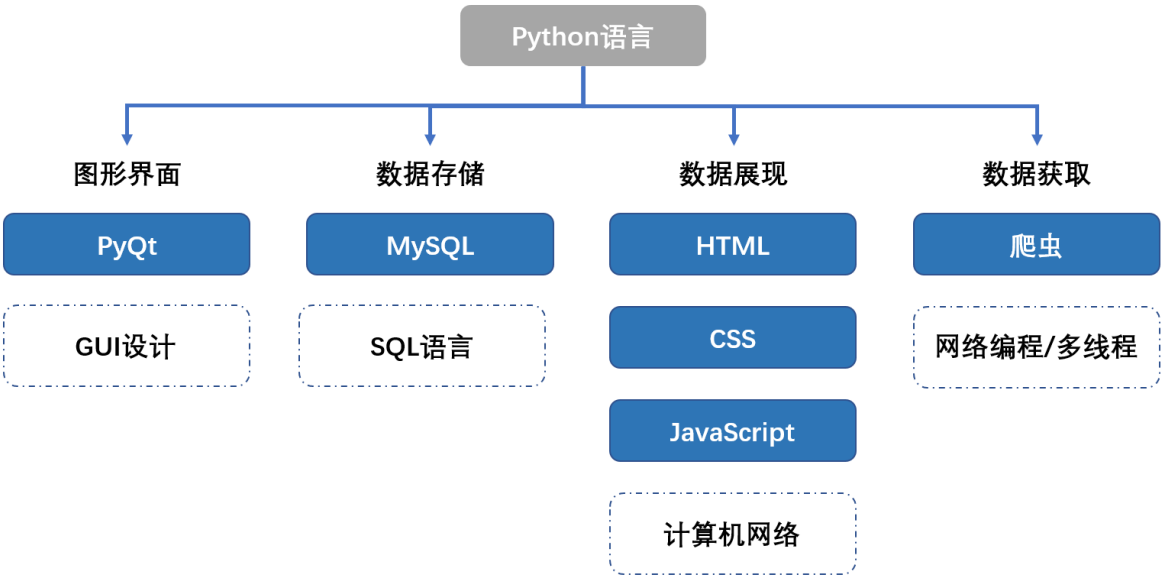
4.在弹出窗口输入想要的安装包名称

5.选中想要的安装包

6.点击“Install Package”进行安装。

安装过程可以点击“OK”关闭当前界面。进度条在Pycharm的最下方。安装成功或失败，右下方会有弹出窗口提示。

安装好的包默认在对应解释器的site-packages文件夹中，例如：D:\soft\python3.8.6\Lib\site-packages


Python语言
图形界面
数据存储
数据展现
数据获取
PyQt
MySQL
HTML
爬虫
GUI设计
SQL语言
CSS
网络编程/多线程
JavaScript
计算机网络