匿名函数：lambda

1. 语法：
    1. 函数名 = lambda 函数：返回值（必须写）
    2. 匿名函数的名字就叫lambda
    3. lambda是定义函数的关键字，相当于函数的def
    4. 只可以返回一个数据类型（：x 必须写）

```
面试题：
print ([lambda : i for i in range(5)])  #返回的是五个内存地址
print(lst[0]()) -- #输出结果为4，因为循环最后一次输出是4，调用最后一次看到的就是4，不加后面的小括号就是调用的最后一个的内存地址，加了括号就是调用函数
```

```
lst = [lambda x :x+1 for i in range(5)]
print(lst[0](5))
输出的结果为6，返回的是x+1
```

```
tu = (lambda :i for i in range(3))
print(tu[0])          #输出错误，不能使用索引
print(tu)             #输出内存地址
print(next(tu))       #一个函数地址
print(next(tu)())     #输出0
```

```
lst = [lambda : i for i in range(3)]
print(lst[0]())      #lst后面加[0]就是为了调用那一个参数的地址，后面再加括号就是为了调用函数，输出是3
tu = (lambda : i for i in range(3))
print(next(tu)())
```

2. 函数体中存放的是代码，生成器中存放的也是代码，

   原因：yield导致函数和生成器的执行结果不一致

```
lst = [lambd x:x+5  for i in range(2)]
print([i(2) for i in lst])

解开顺序：
lst = []
for i in range (2):
    lst.append(lambda x :x+5)
new_list=[]
for i in lst:
    print(i)          #两个函数的内存地址
    new_list.append(i(2))
print(new_list)       #输出列表【7，7】

输出结果：【7，7】
```

3. lambda == def == 关键字
    1. 第一个x是普通函数的形参（放的是位置参数和默认参数），可以写任意多个，也可以不写
    2. ： 后面的是普通函数的返回值，必须写，没有默认的值，必须要有返回值，只能写一个数据类型，（：冒号一定不能丢）
4. 三种调用方式：
    1. （lambda x:x）(5)
    2. [lambda x:x] [0] (5)
    3. f= lambda x : x

       f (5)

内置函数

普通函数

1. 两个字典合并成一个：
    1. dict.updata({新字典}) -- 用法示例：dic2.updata(dic1),在字典2中加入字典1
    2. dict(** dic1,** dic2)：打散--把两个字典打散成一个字典
    3. dic([(1,2)],[(3,3)]) -- 把列表进行迭代，元祖也可以，括号里面的数字少一个都不行，输出结果为：{1: 2, 3: 3}
    4. dict(k=1,k1=2)--- 输出结果为：{'k': 1, 'k1': 2}
2. print()内函数：
    1. sep()：每个元素之间的分隔符，默认是空格

```
print(1,2,3,sep = "|")
输出结果：1|2|3
```

    2. end()：print执行之后的结束语句，默认是换行
    3. file：是文件句柄

```
print (1,2,3,4,5,file = open ("test","w",encoding="utf-8"))
```

    4. flush：刷新
3. abs()：取绝对值,把所有的值全部转换成正数，不管你是负数还是正数
4. dir()：查看当前对象具有什么方法
5. sum()：求和，开始值

```
print (sum ([1,2,3,1]))
print (sum ([1,2,3,1],100))      #100是起始值，就是从100开始进行取和，指定开始位置的值,不写也可以
```

6. zip()：拉链，当长度不一致时，选择长度最短的进行合并，可以是多个

```
面试题：
list1 =[1,2,3,4]
lst2 = ["alex","wusir","meet"]
print(list(zip(lst1,lst2)))

输出结果：[(1,"alex"),(2,"wusir"),(3,"meet")]

print(dict(zip(lst1,lst2))) #返回一个字典
输出结果：{1:"alex",2:"wusir",3:"meet"}
```

7. format():格式转换

    1. 对齐方式

```
print(format("alex","> 20"))    #    右对齐
print(format("alex","< 20"))    #    左对齐
print(format("alex","^ 20"))    #    居中
```

    2. 进制转换

```
print(format(10,"b"))       #十进制转二进制
print(format(10,"08b"))     #不够就补0
```

```
print(format(10,"08o"))     # 最后一个是小写的o，全拼是oct，十进制转换成八进制
print(format(10,"08x"))     #最后一个是小写的x，全拼是hex，十进制转换成十六进制
```

```
print(format(0b1010,"d"))   #二进制转换成十进制
```

    3. reversed()：将一个序列进行翻转，返回翻转序列的迭代器

```
l = reversed('你好')  # l 获取到的是一个生成器
print(list(l))
ret = reversed([1, 4, 3, 7, 9])
print(list(ret))  # [9, 7, 3, 4, 1]
```

8. 禁止使用：eval 和 exec -- 都是转换成字符串类型的函数，但是禁止使用
9. hash()：区分可变数据类型和不可变数据类型

```
print(hash("123"))
print(hash(12))
print(hash(-10))
输出结果：
1701559176827422869，这个数据一直是变动的，不是固定的
12
-10
```

10. help()：查看帮助信息
11. callable()：查看对象是否可以调用

```
# lst = [1,23,4,]
# print(callable(lst))

返回结果是：False，证明列表lst不可以被调用
```

12. int()：函数用于将一个字符串或数字转换成整型

```
print(int())          # 0
print(int('12'))      # 12
print(int(3.6))       # 3
print(int('0100',base=2))  # 将2进制的 0100 转化成十进制。结果为 4
```

13. float：函数用于将整数和字符串转换成浮点数。
14. complex：函数用于创建一个值为 real + imag * j 的复数或者转化一个字符串或数为复数。如果第一个参数为字符串，则不需要指定第二个参数。

```
print(float(3))  # 3.0
print(complex(1,2))  # (1+2j)
```

15. divmod：计算除数与被除数的结果，返回一个包含商和余数的元组(a // b, a % b)。
16. round：保留浮点数的小数位数，默认保留整数。
17. pow：求x**y次幂。（三个参数为x**y的结果对z取余

```
print(divmod(7,2))  # (3, 1)
print(round(7/3,2))  # 2.33
print(round(7/3))  # 2
print(round(3.32567,3))  # 3.326
print(pow(2,3))  # 两个参数为2**3次幂
print(pow(2,3,3))  # 三个参数为2**3次幂，对3取余，结果为2
```

18. bytes：用于不同编码之间的转化。建议使用decode

```
# s = '你好'
# bs = s.encode('utf-8')
# print(bs)
# s1 = bs.decode('utf-8')
# print(s1)
# bs = bytes(s,encoding='utf-8')
# print(bs)
# b = '你好'.encode('gbk')
# b1 = b.decode('gbk')
# print(b1.encode('utf-8'))
```

19. ord:输入字符找当前字符编码的位置，在unicode表位中
20. chr():通过表位序号查找元素，ascii码中
21. repr()：返回一个对象的string形式，俗称原形毕露

```
# %r  原封不动的写出来
# name = 'taibai'
# print('我叫%r'%name)

# repr 原形毕露
print(repr('{"name":"alex"}'))
print('{"name":"alex"}')
```

22. all：可迭代对象中，全都是True才是True
23. any：可迭代对象中，有一个True 就是True

高阶函数：内部帮忙做了一个for循环的函数

1. filter：筛选过滤

```
语法：filter（function，iterable）
function ：1：指定过滤规则（指的是函数的内存地址）
           2：用来筛选的函数，在filter中会自动的把iterable中的元素传递给function ，然后根据function返回的True 或者False来判断是否保留此项数据

iterable：指的是一个可迭代对象

### 写函数名切记不加括号

面试题：

lst = [{'id':1,'name':'alex','age':18},
       {'id':1,'name':'wusir','age':17},
       {'id':1,'name':'taibai','age':16},]

ls = filter(lambda e:e['age'] > 16,lst)

print(list(ls))

结果：
[{'id': 1, 'name': 'alex', 'age': 18},
 {'id': 1, 'name': 'wusir', 'age': 17}]
```

2. map()：映射函数，将每个元素都执行了指定的方法（面试必问）

```
print(list(map()))      --- 语法格式

map(function,iterable)--可以对可迭代对象的每一个元素进行映射，分别取知行function，计算列表中的每个元素，返回新列表

例：计算列表里的每个数平方
lst = [1,2,3,4,5]
print(list(map(lambda s : s*s ,lst)))
输出结果：[1, 4, 9, 16, 25]

例：计算两个列表的和
lst1 = [1,2,3,4,5]
lst2 = [3,4,5,23,54]
print(list(map(lambda x,y : x + y ,lst1,lst2)))

输出结果：[4,6,8,27,59]
```

3. sorted()：排序函数，默认返回的是列表

```
语法：sorted(iterable,key =None,reversed = False)

print(sorted([1,-22,3,5,78,65],key = abs))  # key的意思是知行排序规则，abs 后面不加括号

在sorted 内部会将可迭代对象中的每一个元素传递给这个函数的参数，根据函数运算的结果进行排序


lst = [{'id':1,'name':'alex','age':18},
    {'id':2,'name':'wusir','age':17},
    {'id':3,'name':'taibai','age':16},]

# 按照年龄对学生信息进行排序
print(sorted(lst,key=lambda x : list(x.values())))  #  值排序
print(sorted(lst,key=lambda e:e['age']))

结果：
[{'id': 3, 'name': 'taibai', 'age': 16}, {'id': 2, 'name': 'wusir', 'age': 17}, {'id': 1, 'name': 'alex', 'age': 18}]
```

4. max()：最大值

```
print(max(10,3,56,4,34,2,23))   #输出  56
print(max(10,3,4,5,2,-6,76), key= abs)   # 输出 76
```

5. min()：最小值 ，用法与max用法基本相同，
6. reduce()：计算，一层一层垒起来计算，指定的函数方法必须接收两个形参，使用之前必须先进性导入第三方reduce库

```
from functools import reduce

reduce:指定的函数方法必须接受两个形参，累计算
```

```
print(reduce(lambda x,y:x+y,[1,2,3,4,5]))       #冒号等同于return，结果15
```

闭包

1. 定义：在嵌套函数内，使用非全局变量（非本层变量）就是闭包
    1. 闭包必须是内层函数对外层函数的变量（非全局变量）的引用
    2. 函数执行完毕后，函数体内的空间自行销毁

```
def func():
    a = 1
    def foo():
        print (a)
    print(foo.__closure__)       #判断是不是闭包，返回None 就不是闭包
func()
```

2. 作用：
    1. 保护数据的安全性
    2. 装饰器
3. 查看是不是闭包的方法：
    - 函数名.__ closure __
4. 例题：

```
# 例一：
# def wrapper():
#     a = 1
#     def inner():
#         print(a)
#     return inner
# ret = wrapper()

# a = 2
# def wrapper():
#     def inner():
#         print(a)
#     return inner
# ret = wrapper()

# def wrapper(a,b):
#     def inner():
#         print(a)
#         print(b)
#     inner()
#     print(inner.__closure__)
# a = 1
# b = 2
# wrapper(11,22)
```

装饰器：

1. 原则：开放封闭原则：在不修改源代码及调用方式，对功能进行额外的添加就是开放封闭原则
    1. 开放：对代码扩展进行开放，允许对代码进行添加新功能
    2. 封闭：修改源代码是封闭的
2. 装饰：额外功能， 器： 工具（函数）
3. 简单版函数进行传参：不符合开放封闭原则，每次都需要重新修改inner(index)

```
import time
def index():
    time.sleep(2)  # 模拟一下网络延迟以及代码的效率
    print('欢迎访问博客园主页')

def home(name):
    time.sleep(3)  # 模拟一下网络延迟以及代码的效率
    print(f'欢迎访问{name}主页')

def timmer(func):  # func == index 函数
    start_time = time.time()
    func()  # index()
```

```
    end_time = time.time()
    print(f'此函数的执行效率为{end_time-start_time}')

timmer(index)
```

4. 真正的装饰器：闭包的执行过程必须清楚

```
import time
def index():
    print("这是一个函数")
def timer(func):    #func = inner
    def inner():    #
        start_time=time.time()
        func()
        end_time = time.time()
        print(f"此函数的执行效率为{end_time-start_time}")
    return inner
f = timer(index)        #f == inner
f()

输出结果：
这是一个函数
此函数的执行效率为0.0
```

5. 带返回值的装饰器：

```
import time
def index():
    print("这是一个函数")
    return "还给你"
def func(foo):
    def inner():
        start = time.time()
        ret = foo()         #ret == 还给你
        end = time.time()
        print(f"此函数执行了{end - start}")
        return ret
    return inner
index = func(index)       #index == inner == ret == "还给你"
print(index())
```

6. 被装饰函数带参数的装饰器：

```
import time
# def index():
#     time.sleep(2)
#     print("这是函数")
#     return "返回值"

# def home(name,age):
#     time.sleep(2)
#     print(name,age)
#     print(f"欢迎{name}")
#
# def func(foo):
#     def inner(*args,**kwargs):
```

```
#         start = time.time()
#         foo(*args,**kwargs)
#         end = time.time()
#         print(f"此函数执行效率{end-start}")
#     return inner
#
# home = func(home)
# home("太白","13")
```

7. 标准版装饰器：切记，在装饰器中return后面的参数不加括号，语法糖必须加在装饰函数的正上方

```
def timer(func):  # func = home
    def inner(*args,**kwargs):
        start_time = time.time()
        func(*args,**kwargs)
        end_time = time.time()
        print(f'此函数的执行效率为{end_time-start_time}')
    return inner

@timer  # home = timer(home)
def home(name,age):
    time.sleep(3)  # 模拟一下网络延迟以及代码的效率
    print(name,age)
    print(f'欢迎访问{name}主页')

home('太白',18)
```

原理：home函数如果想要加上装饰器那么你就在home函数上面加上@home，就等同于那句话 home = timer(home)。

```
def wrapper(func):
    def inner(*args,**kwargs):
        '''执行被装饰函数之前的操作'''
        ret = func
        '''执行被装饰函数之后的操作'''
        return ret
    return inner
```

8. 例题：用装饰器实现博客园登陆

```
login_status = {
    'username': None,
    'status': False,
}

def auth(func):
    def inner(*args,**kwargs):
        if login_status['status']:
            ret = func()
            return ret
        username = input('请输入用户名：').strip()
        password = input('请输入密码：').strip()
        if username == '太白' and password == '123':
            login_status['status'] = True
            ret = func()
            return ret
```

```
    return inner

@auth
def diary():
    print('欢迎访问日记页面')

@auth
def comment():
    print('欢迎访问评论页面')

@auth
def home():
    print('欢迎访问博客园主页')

diary()
comment()
home()
```

```
def warpper(f):
    def inner(*args,**kwargs):
         print("被装饰函数执行前")
         ret = f(*args,**kwargs)
         print("被装饰函数执行后")
         return ret
     return inner

 @warpper
 def func(*args,**kwargs):
     print(f"被装饰的{args,kwargs}")
     return "我是func函数"

 @warpper
 def index(*args,**kwargs):
     print(11111)

 print(func(1,2,3,4,5,6,7,8,a=1))
```

装饰器的进阶

1. 有参装饰器:

```
# def warpper(func):
#     def inner(*args,**kwargs):
#         user = input("user:")
#         pwd = input("pwd:")
#         if user == 'alex' and pwd == "dsb":
#             func(*args,**kwargs)
#     return inner
#
# @warpper
# def foo():
#     print("被装饰的函数")
#
# foo()
```

```
# def auth(argv):
#     def warpper(func):
#         def inner(*args,**kwargs):
#             if argv == "博客园":
#                 print("欢迎登录博客园")
#                 user = input("user:")
#                 pwd = input("pwd:")
#                 if user == 'alex' and pwd == "dsb":
#                     func(*args,**kwargs)
#             elif argv == "码云":
#                 print("欢迎登录码云")
#                 user = input("user:")
#                 pwd = input("pwd:")
#                 if user == 'alex' and pwd == "jsdsb":
#                     func(*args, **kwargs)
#
#         return inner
#     return warpper
#
# def foo():
#     print("被装饰的函数")
#
# msg = input("请输入您要登录的名字:")
# a = auth(msg)
# foo = a(foo)
# foo()
```

```
def auth(x):
    def auth2(func):
        def inner(*args, **kwargs):
            if login_status['status']:
                ret = func()
                return ret

            if x == 'wechat':
                username = input('请输入用户名：').strip()
                password = input('请输入密码：').strip()
                if username == '太白' and password == '123':
                    login_status['status'] = True
                    ret = func()
                    return ret
            elif x == 'qq':
                username = input('请输入用户名：').strip()
                password = input('请输入密码：').strip()
                if username == '太白' and password == '123':
                    login_status['status'] = True
                    ret = func()
                    return ret
        return inner
    return auth2

@auth('wechat')
def jitter():
    print('记录美好生活')

@auth('qq')
```

```
def pipefish():
    print('期待你的内涵神评论')
```

解题思路：
@auth('wechat') ：分两步：

- 第一步先执行auth('wechat')函数，得到返回值auth2
- 第二步@与auth2结合，形成装饰器@auth2 然后在依次执行。

2. 多个装饰器装饰一个函数规则：先执行离被装饰函数最近的装饰器

```
def wrapper1(func):
    def inner1(*args,**kwargs):
        print("这是装饰器一开始")
        func(*args,**kwargs)
        print("这是装饰器一结束")
    return inner1

def wrapper2(func):
    def inner2(*args,**kwargs):
        print("这是装饰器二开始")
        func(*args,**kwargs)
        print("这是装饰器二结束")
    return inner2


@wrapper1
@wrapper2
def func():
    print("这是被装饰的函数")

func()
```

打印结果：
这是装饰器一开始
这是装饰器二开始
这是被装饰的函数
这是装饰器二结束
这是装饰器一结束

```
 def auth(func): # wrapper1装饰器里的 inner
     def inner(*args,**kwargs):
         print("额外增加了一道 锅包肉")
         func(*args,**kwargs)
         print("锅包肉 38元")
     return inner

 def wrapper1(func): # warpper2装饰器里的 inner
     def inner(*args,**kwargs):
         print("额外增加了一道 日魔刺生")
         func(*args,**kwargs)
         print("日魔刺生 白吃")
     return inner
```

```
def wrapper2(func):  # 被装饰的函数foo
    def inner(*args,**kwargs):
         print("额外增加了一道 麻辣三哥")
         func(*args,**kwargs)
         print("难以下嘴")
     return inner

 @auth         # 1            7
 @wrapper1     #   2        6
 @wrapper2     #     3    5
 def foo():    #       4
     print("这是一个元宝虾饭店")
```

小技巧：按v字执行

语法糖拆解：

```
foo = wrapper2(foo) # inner = wrapper2(foo)
foo = wrapper1(foo) # inner = wrapper1(inner)
foo = auth(foo)     # inner = auth(inner)
foo()               # auth里边的inner()
```

顺序流程图：

```
1  def auth(func): # wrapper1装饰器里的 inner
14     def inner(*args,**kwargs):
18         print("额外增加了一道 锅包肉")
19         func(*args,**kwargs)
27         print("锅包肉 38元")
15     return inner

2  def wrapper1(func): # warpper2装饰器里的 inner
10     def inner(*args,**kwargs):
20         print("额外增加了一道 日魔刺生")
21         func(*args,**kwargs)
26         print("日魔刺生 白吃")
11     return inner

3  def wrapper2(func):  # 被装饰的函数foo
6      def inner(*args,**kwargs):
22         print("额外增加了一道 麻辣三哥")
23         func(*args,**kwargs)
25         print("难以下嘴")
7      return inner

4  def foo():
24     print("这是一个元宝虾饭店")

8  foo = wrapper2(foo) # inner = wrapper2(foo)      此时的foo就是wrapper2中的inner
12 foo = wrapper1(foo) # inner = wrapper1(inner)    此时的foo就是wrapper1中的inner
16 foo = auth(foo)     # inner = auth(inner)       此时的foo就是auth中inner
17 foo()               # auth里边的inner()         auth中的inner()
```

递归：不断地调用自身，用函数实现

1. 死递归（死循环）：

```
def func():
    print(1)
    func()
func()
```

知识点：官方说明最大深度1000，但实际执行998或997以下，看电脑性能

2. 满足两个条件才是递归
    1. 不断调用自身

2. 有明确的终止条件

3. 举例：以计算年龄为例

```
def age(n):
    if n == 1:
        return 18
    else:
        return age(n-1)+2
print(age(3))        #3代表执行三次
```

拆解：

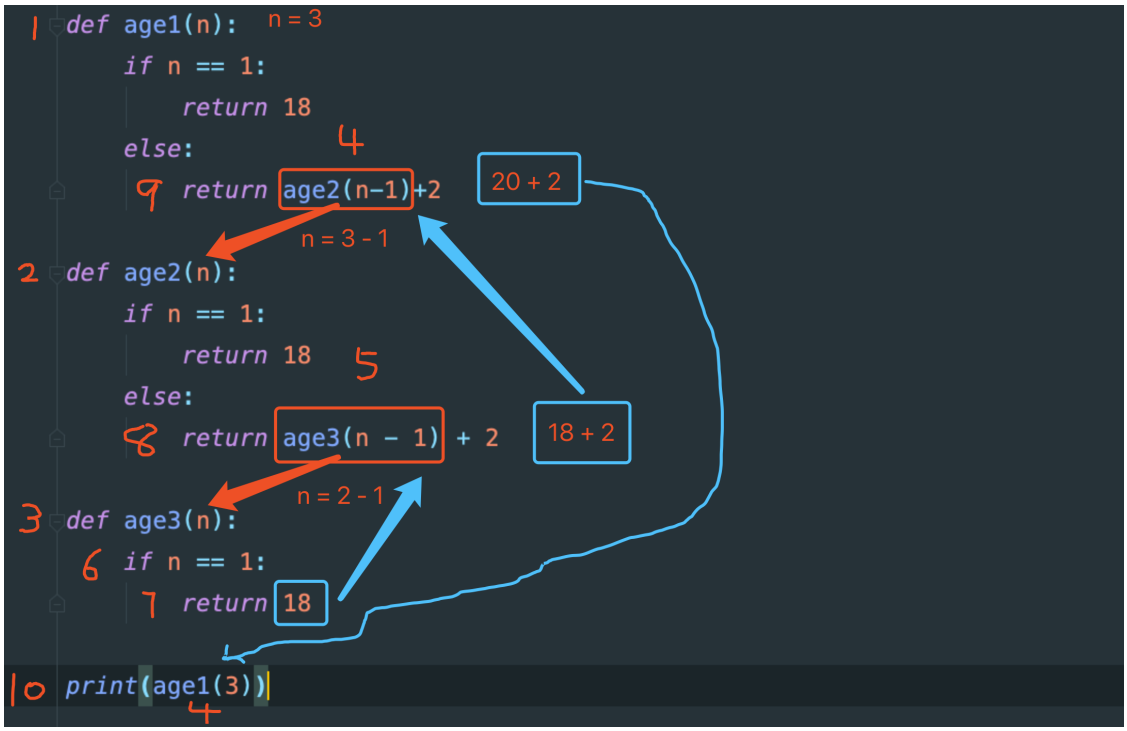
```
![image-20190627140416004](C:\Users\heyul\Desktop\image-
20190627140416004.pngdef age1(n):
    if n == 1:
        return 18
    else:
        return age2(n-1)+2

def age2(n):
    if n == 1:
        return 18
    else:
        return age3(n - 1) + 2

def age3(n):
    if n == 1:
        return 18

print(age1(3))
```

流程图：
图中红色箭头是递的过程，蓝色箭头是归的过程



模块：

自定义模块

1. 定义：一个文件就是一个模块（能被调用的文件，模块就是一个工具箱，工具就是函数）
2. 作用：
    1. 将代码文家化管理，提高可读性，避免重复代码
    2. 拿来就用（避免重复造轮子），python中类库比较多，提升开发效率
3. import导入时会做三件事：

```
# import test  # 只能将整个工具箱拿来
# a = test.t1
# b = test.t2
#
# a()
# b()
```

    1. 将.py文件中的所有代码读取到当前文件
    2. 在当前文件开辟空间
    3. 等待被调用
4. import导入同一个模块名时，只执行一次
5. import 和from 文件名较长时，都可以用as起别名，目的是为了防止内置模块名与自定义模块名重复，自定义模块会覆盖内置模块
6. 导入模块名时后面不能加后缀
7. 每个模块都有一个独立的内存空间，理论上是全局空间
8. from ： 推荐使用from

```
# from test import t1 as t # 从test工具箱中将t1这个工具拿过来
#
# def t1():
#     print("高级工程师")
# t1()
```

9. import 和 from的区别：
    1. from只能执行导入的工具
    2. import能后执行整个模块中所有的功能
    3. from容易将当前文件中定义的功能覆盖
    4. from 比import灵活
10. import只能导入当前文件夹下的模块
11. import 后边不能加点操作 ***
12. import 和 from 使用的都是相对路径
13. 飘红不代表报错
14. sys：和python解释器交互的接口
    1. sys.path.append(r"被导入的模块路径")
        - 内存 > 内置 > 第三方> 自定义
    2. sys.path.insert(0.r"被导入的模块路径")
        - *内存> 自定义 > 内置 > 第三方*
15. 模块的两种用法
    1. 当做模块导入：使用import 和 from， __ name __ 返回的是当前模块名
    2. 当作脚本执行：__ name __ 返回 '__ main __'

16. 只有py文件当做模块被导入时，字节码（pyc）才会进行保留
17. 导入模块时遇到的坑：
    1. 注意自己的定义的模块名字与系统名字冲突
    2. 注意自己的思路---循环导入时建议导入模式放后边一点，需要的地方，不要互相查找内容
18. 不建议一行导入多个
19. from test import *　意思是：拿整个工具箱过来
    1. 通过 __ all __ 控制要导入的内容
    2. __ all __ = ["a","func"] 控制 import *

time模块：import time

1. time.time()：时间戳，是一个浮点数，按秒来计算
2. time.sleep（）：睡眠，程序暂停多少秒执行
3. python中时间日期格式化符号：

```
必背

%y 两位数的年份表示（00-99）
%Y 四位数的年份表示（000-9999）
%m 月份（01-12）
%d 月内中的一天（0-31）
%H 24小时制小时数（0-23）
%I 12小时制小时数（01-12）
%M 分钟数（00=59）
%S 秒（00-59）
```

```
简单记忆，了解就好

%a 本地简化星期名称
%A 本地完整星期名称
%b 本地简化的月份名称
%B 本地完整的月份名称
%c 本地相应的日期表示和时间表示
%j 年内的一天（001-366）
%p 本地A.M.或P.M.的等价符
%U 一年中的星期数（00-53）星期天为星期的开始
%w 星期（0-6），星期天为星期的开始
%W 一年中的星期数（00-53）星期一为星期的开始
%x 本地相应的日期表示
%X 本地相应的时间表示
%Z 当前时区的名称
%% %号本身
```

4. 时间格式转换：
    1. 时间戳（以秒计算）---> 结构化时间
        - time.localtime(time.time)---是一个命名元祖，可以使用索引和名字查
    2. 结构化时间 （2019-09-13 09：30：00）---> 字符串时间，有8个小时的时差，使用时减去八小时
        - time.strftime("%Y-%m-%d %H:%M:%S","结构化时间")
    3. 字符串时间 ---> 结构化时间

- time.strptime("时间字符串"，“%Y-%m-%d %H:%M:%S”)

4. 结构化时间 ----> 时间戳
   - time.mktime(结构化时间)
5. 总结：

```
# time.time() 时间戳
# time.sleep() 睡眠
# time.localtime() 时间戳转结构化
# time.strftime() 结构化转字符串
# time.strptime() 字符串转结构化
# time.mktime() 结构化转时间戳
```

datetime模块

1. 格式：from datetime import datetime
2. datetime.now()：获取当前时间 2019-08-24 14:47:46.428588
3. 获取指定时间：datetime(2019,08,22,12,56,00)
4. datetime（对象）----> 时间戳

```
d = datetime.now()
print(d.timestamp())

输出结果：1566629391.388825
```

5. 时间戳（秒） ---> 对象，使用fromtimestamp

```
from datetime import datetime
import time
f_t = time.time()
print(datetime.fromtimestamp(f_t))

结果：2019-08-24 14:45:05.664260
```

6. 字符串---> 对象(格式：2019-09-13 14：52：44)

```
d = "2018-12-31 10:11:12"
datetime.strptime(d,"%Y-%m-%d %H:%M:%S")
输出结果：2018-12-31 10:11:12
```

7. 对象 ----> 字符串（）：如果已经有了datetime对象，要把它格式化为字符串显示给用户，就需要转换为str，转换方法是通过 `strftime()` 实现的，同样# 需要一个日期和时间的格式化字符串

```
d = datetime.now()
d.strftime("%Y-%m-%d %H:%M:%S")

输出结果：2019-08-24 15-21-23
```

8. datetime 的加减，先导入模块 from datetime import datetime,timedelta

```
对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用+和-运算符，不过需要导入timedelta这个类：
```

1. 减：timedelta ，最大只能按周减
2. 加也是一样

```
from datetime import datetime, timedelta
now = datetime.now()
now
datetime.datetime(2015, 5, 18, 16, 57, 3, 540997)
now + timedelta(hours=10)
datetime.datetime(2015, 5, 19, 2, 57, 3, 540997)
now - timedelta(days=1)
datetime.datetime(2015, 5, 17, 16, 57, 3, 540997)
now + timedelta(days=2, hours=12)
datetime.datetime(2015, 5, 21, 4, 57, 3, 540997)
```

9. 指定datetime时间

```
current_time = datetime.datetime.now()
print(current_time.replace(year=1977))  # 直接调整到1977年
print(current_time.replace(month=1))  # 直接调整到1月份
print(current_time.replace(year=1989,month=4,day=25))  # 1989-04-25
18:49:05.898601
```

random模块

1. 导入：import random
2. 随机小数：random.random()：大于0小于1之间的小数
3. 指定数字之间的小数，不包含指定的最大值：random.uniform()
4. 随机整数：random.randint(1，5)：大于等于1且小于等于5之间的整数
5. 指定奇数或偶数，使用步长：random.randrange(1,19,步长)
6. 随机选择一个并返回，会出现重复元素，以列表形式返回：random.choice((1,2,3,),k=3),出现3个元素
7. 任意选择三个元素出现，不会重复：random.sample([1"23"3432]k=3)
8. random.shuffle():打乱输入的顺序，在原地（原内存）打乱

```
练习：随机生成验证码
import random

def v_code():

    code = ''
    for i in range(5):

        num=random.randint(0,9)
        alf=chr(random.randint(65,90))
        add=random.choice([num,alf])
        code="".join([code,str(add)])

    return code

print(v_code())
```

软件命名规范：分文件存储

1. 当代码存放在一个py文件中时会存在一下缺点：
    1. 不便于管理
    2. 可读性差
    3. 加载速度慢
2. 是Django的雏形
3. 程序员预定俗称的一些东西
    1. 启动文件：也叫启动接口，通常文件夹名字使用bin，存放启动程序，通常使用starts命名py文件
    2. 公共文件：是大家都可以使用的文件，功能，通常使用lib命名文件夹，py文件使用common命名py文件
    3. 配置文件：也叫静态文件，存储的都是变量，数据库的一些连接方式，获取到的是都是redis，文件夹命名为conf，py文件命名为settings
    4. 主逻辑：程序主要是干什么的，是程序的核心文件，通常使用core命名文件夹，py文件使用src命名
    5. 用户相关数据：存储用户的账户，密码等文件，文件夹命名为db，py文件一般命名为register
    6. 日志：记录重要信息，记录开发人员的行为，文件夹命名为log，py文件为logg

序列化（背）

1. json ： 将数据类型转换成自负循环（序列化），将字符串装换成原数据类型（反序列），支持dict，list，tuple等，序列后都变成了列表
    1. dumps，loads ------ 用于网络传输
        1. json.dumps：将数据类型转换成字符串
        2. json.loads：将字符串转换成原数据类型
    2. dump，load ------ 用于文件传输
        1. json.dump：一个load对应一个dump
        2. 中文转换时，必须加ensure_ascii = False
    3. 转换后的数据类型排序：sort_keys =True
2. pickle:只有python有，几乎可以序列Python中所有数据类型（匿名函数不行）
    1. 用于网络传输--dumps，loads
        1. dumps：将原数据类型转换成类似字节的东西
        2. loads：将类似于字节的东西转换成源数据类型
    2. 用于文件写读--dump，load
        1. dump：写入文件的时候用的是wb模式，没有解码encoding
        2. load：反序列化

os 文件夹 文件 路径

工作路径：和操作系统做交互（全背）

1. os.getcwd()：获取当前文件的路径
2. os.chdir(绝对路径)：改变当前工作目录
3. os.curdir（）：返回当前目录："."
4. os.pardir（）：返回父级目录：".."

文件夹

1. os.mkdir()：创建文件夹
2. os.rmdir()：删除空的文件夹，不为空的不删除
3. os.makedirs()：创建多层文件夹，以递归的方式创建
4. os.removedirs（）：若目录为空则删除，并递归到上一层继续删除空文件夹
5. os.listdir():列表显示指定文件夹下的所有内容，并以列表的形式打印

文件

1. os.remove()：删除文件，彻底删除，不能撤回 ***
2. os.rename()：重命名文件夹 ***
3. os.stat()：获取文件/目录信息

路径

1. os.path.abspath()：返回的是绝对路径 ***
2. os.path.split()：返回的是将路径分割成目录和文件名的元祖
3. os.path.dirname()：返回到上级目录
4. os.path.basename()：获取到当前文件名
5. os.path.join("")：路径拼接，多个路径拼合后返回 ***
6. os.path.exists(路径)：判断路径是否存在
7. os.path.isabs()：判断是不是绝对路径
8. os.path.isfile()：判断文件存不存在
9. os.path.isdir()：判断是不是文件夹
10. os.path.getatime()：返回文件所指向的文件或者目录的最后访问时间
11. os.path.getmtime()：返回文件所指向的文件或者目录的最后修改时间
12. os.path.getsize()：返回文件的大小 *** 获取文件较准确

sys：与python解释器做交互的一个接口

1. sys.path : 返回模块的搜索路径，模块查找的顺序 ***
2. sys.argv : 只能在终端执行
3. sys.modules：查看所有已加载到内存的模块
4. sys.platform：查看当前操作系统平台
5. sys.version：查看当前Python解释器版本