# python特点

**开源** 面向对象 简单易学等等.

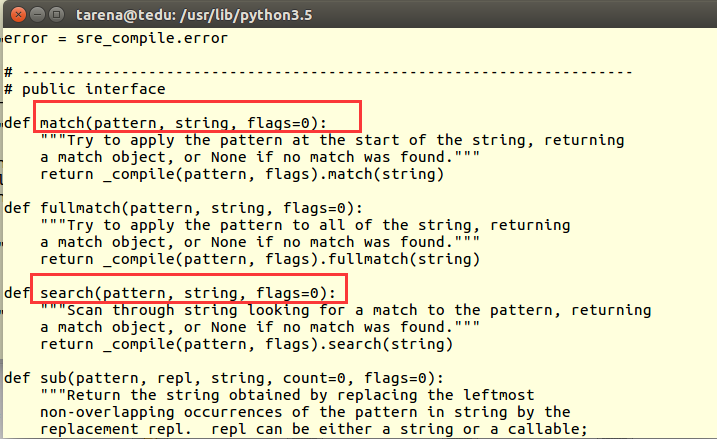
Ubuntu：

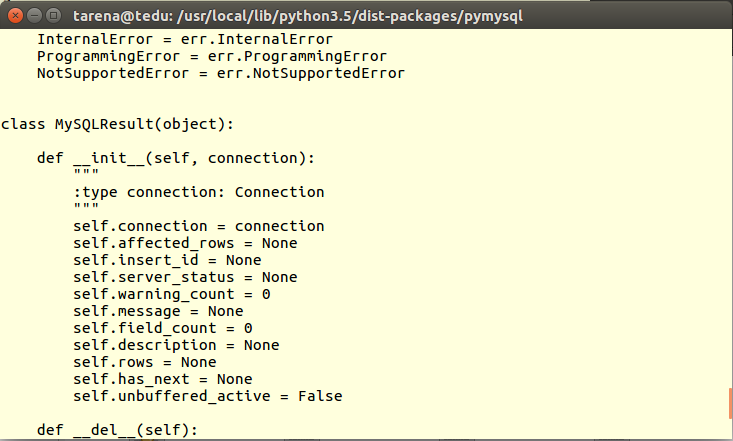
进入交互模式：python3

导入模块：import re [/os/string]

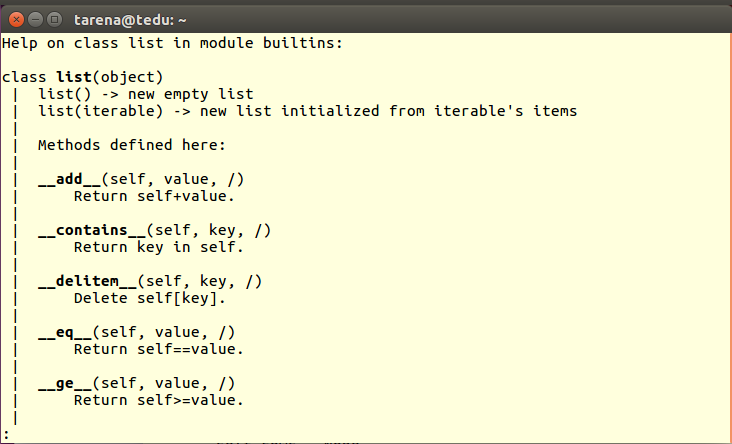
查看模块路径os.\_\_file\_\_：/usr/lib/python3.5/re.py

查看模块源代码：cat /usr/lib/python3.5/re.py





help(list)



pycharm：

ctrl键

**python2与python3:**

1.运算符 ：python2.x中不等于有两种写法，!=和<>

Python3.x中，去掉了<>，只有 != 一种写法。

2. Print语句 ：Python2中print是一个语法结构 没有参数 也可以没有括号

python3中print是一个内置函数，有多个参数。

3. 编码格式： python2 编码是ascii　　如果有中文需要指定编码

# -\*- coding:UTF-8　-\*-

python3 系统默认编码是utf-8

4.迭代器： python2使用xrange和range都可以 Python3取消了xrange

5.input函数：在Python2里面可以用raw\_input和input

而且使用input的时候会自动判断他的数据类型 并不都是str

python3取消了raw\_input

6.基本类型： Python2.x中有两种整数类型，一般的32位整数和长整数，

长整数都是以L或者l，超过32位长度之后会自动转换为长整型。

Python3.x中，只有一种类型，没有长度限制。

**#!/usr/bin/env python3 用法**

#!/usr/bin/python3 和 #!/usr/bin/env python3 的区别

#!/usr/bin/python3 表示 python3 解释器所处的绝对路径就是 /usr/bin/python3, 路径被写死了.

如果python 解释器不在该路径下的话, 脚本就无法执行了!

#!/usr/bin/env/ python3 表示从 "PATH 环境变量"中查找 python3 解释器的位置, 路径没有被写死, 而是在"环境变量"中寻找 python3 解释器的安装路径, 再调用该路径下的解释器来执行脚本.

#!/usr/bin/env python3 的写法更灵活更具有通用性

看下文件是否有可执行权限： ls –l 文件名

如果没有： chmod +x temp.py #增加执行权限

./temp.py #运行脚本

# 进制

# 变量

变量是指向各种类型值的名字，以后再用到某个值时，直接引用这个名字即可，不用再写具体的值，在python中，变量的使用环境非常宽松，没有明显的变量声明，而且类型不是固定的。如果你不能确定变量或数据的类型，就用解释器内置的函数type确认

eg：type（“hello，world”）

注意：

在使用变量之前，需要对其进行赋值 =是赋值的意思，左边是名字，右边是值

变量名的命名规则:数字字母和下划线组成 不能是纯数字,不能以数字开头

不能使用python的关键字

# 运算符

Python语言支持以下类型的运算符:

## [算术运算符](http://www.runoob.com/python/python-operators.html#ysf1)

|  |  |
| --- | --- |
| + | 加 - 两个对象相加 |
| - | 减 - 得到负数或是一个数减去另一个数 |
| \* | 乘 - 两个数相乘或是返回一个被重复若干次的字符串 |
| / | 除 - x除以y |
| % | 取模 - 返回除法的余数 |
| \*\* | 幂 - 返回x的y次幂 |
| // | 取整除 - 返回商的整数部分（**向下取整**） |

## [比较（关系）运算符](http://www.runoob.com/python/python-operators.html#ysf2)

|  |  |
| --- | --- |
| == | 等于 - 比较对象是否相等 |
| != | 不等于 - 比较两个对象是否不相等 |
| <> | 不等于 - 比较两个对象是否不相等 |
| > | 大于 - 返回x是否大于y |
| < | 小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。 |
| >= | 大于等于 - 返回x是否大于等于y。 |
| <= | 小于等于 - 返回x是否小于等于y。 |

## [赋值运算符](http://www.runoob.com/python/python-operators.html#ysf3)

|  |  |
| --- | --- |
| = | 简单的赋值运算符 |
| += | 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](http://www.runoob.com/python/python-operators.html#ysf4)

|  |  |  |
| --- | --- | --- |
| and | x and y | 布尔"与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。 |
| or | x or y | 布尔"或" - 如果 x 是非 0，它返回 x 的值，否则它返回 y 的计算值。 |
| not | not x | 布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。 |

## [位运算符](http://www.runoob.com/python/python-operators.html#ysf5)

|  |  |
| --- | --- |
| & | 按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0 |
| | | 按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。 |
| ^ | 按位异或运算符：当两对应的二进位相异时，结果为1 |
| ~ | 按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1 。~x 类似于 -x-1 |
| << | 左移动运算符：运算数的各二进位全部左移若干位，由 << 右边的数字指定了移动的位数，高位丢弃，低位补0。 |
| >> | 右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，>> 右边的数字指定了移动的位数 |

## [成员运算符](http://www.runoob.com/python/python-operators.html#ysf6)

|  |  |
| --- | --- |
| in | 如果在指定的序列中找到值返回 True，否则返回 False。 |
| not in | 如果在指定的序列中没有找到值返回 True，否则返回 False。 |

## [身份运算符](http://www.runoob.com/python/python-operators.html#ysf7)

|  |  |
| --- | --- |
| is | is 是判断两个标识符是不是引用自一个对象 |
| is not | is not 是判断两个标识符是不是引用自不同对象 |

==

## [运算符优先级](http://www.runoob.com/python/python-operators.html#ysf8)

|  |  |
| --- | --- |
| \*\* | 指数 (最高优先级) |
| ~ + - | 按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@) |
| \* / % // | 乘，除，取模和取整除 |
| + - | 加法减法 |
| >> << | 右移，左移运算符 |
| & | 位 'AND' |
| ^ | | 位运算符 |
| <= < > >= | 比较运算符 |
| <> == != | 等于运算符 |
| = %= /= //= -= += \*= \*\*= | 赋值运算符 |
| is is not | 身份运算符 |
| in not in | 成员运算符 |
| not or and | 逻辑运算符 |

# 基本输入输出函数

a=**input(‘请输入：’)**

如果输入123.4

需要先使用float() 将字符串123.4转换为浮点数123.4 再使用int()函数 将浮点数123.4转化为整数123

如果输入是1234 可以直接使用int()将字符串1234转换成整数1234

**print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)**

**sep:**数据和数据之间使用sep隔开，默认是用空格隔开

**end：**数据输出完毕之后，最后追加的内容，默认情况“\n”,换行。

**file=sys.stdout**：设置输出设备，及把print中的值打印到什么地方，默认输出到终端，

可以设置file= 文件储存对象，把内容存到该文件中，如下：

f = open(r'a.txt', 'w')

print('python is good', file=f)

f.close()

则把python is good保存到 a.txt 文件中

**flush=False：** 该参数主要是刷新， 默认False，不刷新

（先将内容存在内存中，只有当文件对象关闭时才将缓冲区的内容存在文件中），

Ture时刷新（及时性），如下:

import time

f = open(r'a.txt', 'w')

print('python is good ', file=f, flush=False)

time.sleep(10)

f.close()

# 语句和表达式

比如赋值语句 a=1+2 输入输出语句print(1+2)

表达式：由值、变量和运算符组成。 1+2

语句由表达式组成，通常情况下一条语句可以独立执行的，有一个确定的结果。一条语句最好放在一行。多条语句可以放在一行，语句和语句之间使用分号“；”隔开。

# if 语句

if 表达式1：

语句块1

elif 表达式2：

语句块2

elif 表达式3：

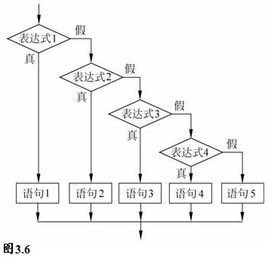
语句块3

elif 表达式4：

语句块4

else:

语句块5



注意：

按照顺序执行判断

当表达式1成立 不再判断其他表达式 ，否则判断表达式2是否成立，成立的话不再判断其他表达式 。。

如果所有表达式都不成立，执行else

# 字符串的格式化

字符串的格式化表达式

语法格式:

格式化字符串 % 参数值

或

格式化字符串 % (参数值1, 参数值2, ....)

说明:

% 右侧为参数值,当有多个参数值是用括号()括起来,各参数间用

逗号(,) 分隔,

% 左侧为格式化字符串,中间的%开头的字符为占位符和类型码

**print( '%s is a %s' %('jerry','mouse') )**

**print( '{ } is a { }'.format('tom','cat') )**

**print('{} is a {}'.format('tom','cat'))**

# 循环:

## for循环

s = input()

count\_a=0

for c in s:

if c=="a":

count\_a += 1

print(count\_a)

range函数：

range(stop):生成0--stop之间的数据（整数），不含是stop

range(10)--0 ,1,2,3,4,5,6,7,8,9

range(start,stop[,step])

生成start---stop 之间的数据，默认步长是1

包含start不包含stop

作用：用来创建一个生成一系列整数的可迭代对象。

注意：可以使用for语句把可迭代对象中的数据取出来。

## while循环

x = 100

while(x<=1000):

print('我爱python')

x+=2

目的:

for循环的目的是为了限制循环体的执行次数，使结果更精确。

while循环的目的是为了反复执行语句或代码块。

continue语句：（跳过）

作用：

结束本次循环，进入到下一次循环。重新开始循环。

continue语句后面的内容不被执行。

break语句：（停止）

用来结束当前循环

#现在有１张试卷　　复印10份

for x in range(1,11):

if x==6:

continue #表示跳过　　本次循环内容不再执行　执行下一次

# break#表示结束本层循环　　不再执行

print('我正在做第',x,'份试卷')

#现在有１张试卷　　复印10份　　放在１个箱子里面　　　10个同样的箱子

for x in range(1,11):

for y in range(1,11):

if y==6:

#continue #每箱的第６份试卷都没做

break#每箱的试卷只做前５份

print('我正在做第',x,'箱的第',y,'份试卷')

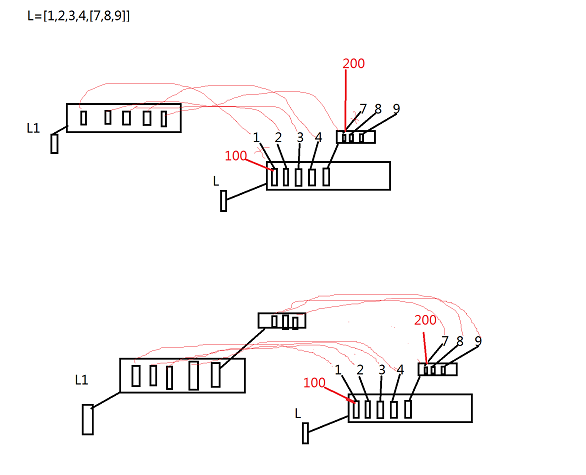
# python的数据类型

## 字符串

## 列表

**(深浅拷贝)**

|  |
| --- |
| 深拷贝（你和你的克隆）和 浅拷贝（你和你的影子）：  >>> L = [1,2,3,4,[7,8,9]]  >>> L1 = L.copy()  >>> L[0] = 100  >>> L[4][0]=200  L.copy():浅拷贝只复制一层变量，不会复制深层变量对应的数据  >>> L  [100, 2, 3, 4, [200, 8, 9]]  >>> L1  [1, 2, 3, 4, [200, 8, 9]]  深拷贝：deepcopy  复制变量以及变量关联的对象一起复制的过程。  import copy#导入模块；  L=[1, 2, 3, 4, [7,8, 9]]  L1=copy.deepcopy(L)  L[0]=100  L[4][0]=200  >>> L  [100, 2, 3, 4, [200, 8, 9]]  >>> L1  [1, 2, 3, 4, [7, 8, 9]] |



## 元组

## 字典

## 集合

# 函数

函数：

函数是可以重复执行的语句块，可以重复调用，

函数的作用：

用于封装语句，提高代码的重用性

语法规则：

def 函数名(形式参数)：

语句块（代码块）

例如：

def sum(a,b,c):

d=a+b+c

print(d)

return d

print(sum(34,1,2))

函数名和这段代码进行绑定

return 语句：

return 表达式

注意：表达式也可以没有

作用：

1、结束函数的执行。

2、返回调用函数的地方

3、返回一个对象的引用关系到函数调用的地方。

注意：

1、没有return语句或者return语句后没有内容，返回的是None

注意区分一下return 和print 没有任何关系

## 传参

位置传参：

def myfun(a,b,c):

pass

myfun(10, 20, 30)

序列传参：

def myfun(a,b,c):

pass

myfun(\*(10, 20, 30))

关键字传参：

def myfun(a,b,c):

pass

myfun(a = 10,b = 20,c =30)

字典传参：

def myfun(a,b,c):

pass

T={"a":10,"b":20,"c":30}

myfun(\*\*T)

函数传参综合使用：

1.函数的传参方式，在能够确认形参唯一匹配的情况下，可以任意组合

2.要求位置传参先与关键字传参（位置传参在关键字传参的左边）

myfun(100,300,c = 200)

3.关键字传参和字典关键字传参可混合使用，顺序可以颠倒。

myfun(a = 100,\*\*{"b":200,"c":300})

myfun(\*\*{"b":200,"c":300},a = 100)

## 缺省参数

函数的缺省参数：

def 函数名（形参名1 = 默认数据1，形参名2 =默认数据2，.......）：

pass

默认传参必须是从右到左，即只要是有默认传参，右边一点是默认传参。

默认传参可以是0个或者是多个，甚至全部都是默认传参。

def myfun(a,b,c)

……..

def myfun(a = 100,b=100,c=100):

....

def myfun(a,b= 100,c =200):

....

def myfun(a =100,b,c=200):#error

def myfun(a =100,b=200,c):#error

def myfun(a ,b=200,c): #error

def myfun(a=100,b=200,c): #error

print(a+b+c)

myfun(300)

## 函数嵌套

1. 函数的嵌套

def语句可以位于任何位置

2. 函数可以赋值给变量,例如：

def my\_add(a,b,c)

调用改函数时，不通过函数名称调用，将改函数赋值给变量，通过变量名称进行调用：

f = my\_add

print(f(a,b,c)) 等同于 print(my\_add(a,b,c))

3. 函数可以作为参数传递，例如：

def test(a,b,c,func)

return (func(a,b,c)) # 调用func函数，将a,b,c作为参数

print(test(3,4,5,my\_add)) # 在test函数中调用my\_add函数

4. 函数可以作为返回值，例如：

def get\_func():

return my\_add; # 返回函数，而不是变量

使用：

f2 = get\_func(); # f2是指向函数的变量

print(f2(3,4,5)); # 等价于调用 my\_add(3,4,5)

## 变量作用域LEGB

1).本地作用域（L）

局部变量仅仅是暂时的存在，依赖创建该局部作用域的函数是否处于活动的状态。所以，一般建议尽量少定义全局变量，因为全局变量在模块文件运行的过程中会一直存在，占用内存空间。

2).上一层本地作用域（E）

E也包含在def关键字中，E和L是相对的，E相对于更上层的函数而言也是L。与L的区别在于，对一个函数而言，L是定义在此函数内部的局部作用域，而E是定义在此函数的上一层父级函数的局部作用域。主要是为了实现Python的闭包，而增加的实现。

def outer():

a=10

def inner():

b=10

3).全局作用域（G）

在模块文件顶层声明的变量具有全局作用域

全局作用域的作用范围仅限于单个模块文件内

4)内置作用域（B）

max(val1, val2) # max is a built-in fun,

函数也可以赋值给变量，所以所有的内置函数的作用域都可视作B

## lambda表达式

1）用来创建匿名函数，不用return语句

表达式的值就是匿名函数的返回值

2）对匿名函数支持有限，在简单情况下使用

foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]

print map(lambda x: x \* 2 + 10, foo)

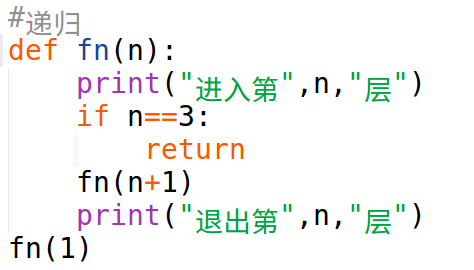
[14, 46, 28, 54, 44, 58, 26, 34, 64]

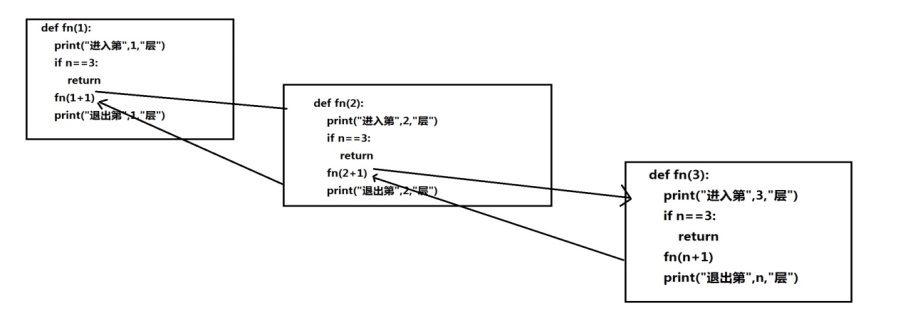
g = lambda x:x+1

g(1)

g(2)

## 递归函数





|  |
| --- |
| 递归函数有层级限制，超过最大层级报错  可以通过sys.getrecursionlimit()查看  可以通过sys.setrecursionlimit()修改  import sys  def fn(n):  if n==1:  return 1  else:  return n\*fn(n-1)  print(fn(998))  sys.setrecursionlimit(1500)  print(sys.getrecursionlimit()) |

## 高阶函数

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

def add(x, y, f):

return f(x) + f(y)

当我们调用add(-5, 6, abs)时，参数x，y和f分别接收-5，6和abs，根据函数定义，我们可以推导计算过程为：

x ==> -5

y ==> 6

f ==> abs

f(x) + f(y) ==> abs(-5) + abs(6) ==> 11

用代码验证一下：

>>> add(-5, 6, abs)

11

编写高阶函数，就是让函数的参数能够接收别的函数。

小结

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

# 闭包

3个条件：

必须有内嵌函数

外部函数返回内嵌函数

内嵌函数内部调用外部嵌套函数的变量

# 装饰器



|  |
| --- |
| import time  def check\_time(fn):  def myplay(t,f):  n1=time.time()  fn(t,f)  n2=time.time()  print("本次播放消耗时间",n2-n1,"秒")  return myplay  @check\_time #play=check\_time(play)  def play(title,frame):  print("正在播放",title,"的","第",frame,"帧")  play("猫和老鼠",3) |

闭包是为了做装饰器吗 装饰不用闭包可以实现吗？ 闭包的好处是什么？

# 系统模块

问题：当软件规模增大时，编码、维护难度、错误率增加

解决方式：对问题规模进行分解，把大的问题分解成若干小的问题，从而降低复杂度

1）模块：

-把函数（代码）按照功能分类，一类功能放到一个文件

一个文件就是一个模块

-Python模块中包含函数、变量

2）模块化编程的优点

- 能跟好地组织代码，使代码更清晰、更具逻辑性、更易维护

- 能提高代码重用率

- 有利于解决函数、变量名字冲突

3）Python中模块分类

- 内置模块：解释器自带的，可以直接使用的

- 标准库：安装Python已经安装好，使用需先导入

- 第三方库：使用之前需安装

例如：pip3 install pymongo

4）模块使用

import sys # 导入整个模块

from math import pi # 导入math下的pi属性

from math import sin # 导入math的sin函数

from random import \* # 导入random所有内容

import time # 导入多个模块

time.sleep()

from time import \*

sleep()

# 异常

try:

#执行某个操作(可能引发异常)

#如果这里出现了异常

#则跳转到except分支执行

except 错误类型1 [as 变量1]:

#异常处理语句1

except 错误类型2 [as 变量2]:

#异常处理语句2

except （错误类型3，错误类型4）:

#异常处理语句3

else:

#try语句块没有异常执行的代码

finally:

#不管是否发生异常

#都会执行本代码块

其中：

as 给异常对象命名，可以省略

except 可以是一个或多个

else 最多一个，可以省略

finally 最多一个，可以省略

通常是用来执行一些清理操作

Python的标准异常

BaseException 所有异常的基类

Exception 常规异常基类

数值计算异常

ZeroDivisionError 0除数异常

OverflowError 数值计算超过最大限制

ArithmeticError 所有数值计算异常的基类

I/O类异常

IOError 输入/输出异常

EOFError 文件结束标记

属性，变量异常

NameError 未声明/初始化的对象

AttributeError 对象没有该属性

内存，缓冲区异常

IndexError 索引异常，入列表越界

MemoryError 内存溢出

数据类型

TypeError 对类型的无效操作

ValueError 无效的参数

系统类

SyntaxError 语法错误

SystemError 一般的解释器错误

主动抛出异常：raise语句

语法： raise [异常类型, 异常对象]

目的：告诉调用者此处有异常

# 主动触发异常示例

try:

a = input("请输入一个数字:")

if a == "": #如果读入的是空串

raise ValueError #触发参数异常

if not a.isdigit():#读入不是数字

raise ValueError

if a == "0":

raise ZeroDivisionError

except ValueError:

print("输入数据不合法")

except ZeroDivisionError:

print("除数不能为0")

else:

print("1/a=",1/int(a))

断言：assert

1）作用：用来对某个表达式进行判断

如果该表达式的结果为False

则强行抛出一个AssertionError异常

通常用于判断参数或数据的合法性

2）语法：

assert 表达式, 错误数据

\* 当表达式不满足时，用后面的错误数据

构建一个AssertionError异常并抛出

# 演示assert断言的用法

def get\_age():

age = int(input("请输入年龄:"))

#age的正常范围0～130

assert age < 130, "年龄超出范围"

assert age > 0, "年龄必须大于0"

return age

try:

age = get\_age()

except AssertionError as e:

print(e)

else:

print("age:", age)

# 可迭代对象

**迭代器**

1）可迭代对象：包含多个对象或元素 可以逐一取出

什么是迭代器：访问可迭代对象的工具

2）作用：遍历可迭代对象

3）在Python中，使用iter函数

取得可迭代对象的迭代器

it = iter(s) #s是可迭代对象

4）迭代器可以使用next()函数

取得可迭代对象的下一个元素

当没有数据（或遍历完成）

会收到StopIteration异常

5）常见的**可迭代对象**

列表，元组，字符串，集合

6）主要函数

- iter(iterable)

功能：从可迭代对象iterable中

取得一个迭代器

- next(iterator)

功能：从迭代器iterator中

获取下一个元素

如果无法获取下一个元素时

抛出StopIteration异常

7)迭代器的优点

- 不管可迭代对象是什么,迭代器都使用同一种方式访问遍历

- 不知道可迭代对象元素个数的情况下,可以通过next()函数取出下一个元素

直到无法取出为止

**生成器**：生成器函数，生成器表达式

1）生成器函数

含有yield语句的函数就是生成器函数

此函数被调用时，动态创建一个对象返回

yield语法：

yield 表达式

用于def函数中，目的是将该函数作为

一个生成器函数，每执行一条yield语句

就动态创建一个对象，供迭代器的,next()函数取值

生成器函数说明：

- 生成器函数将返回一个生成器对象，该对象是一个可迭代对象

- 迭代器从这个可迭代对象中取值时，执行生成器函数中的yield产生一个对象

yield语句后面的部分不再执行

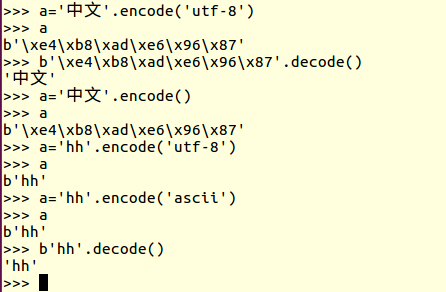
- 下一次迭代器从该可迭代对象中， 取值的时候，从上一次yield语句后面开始执行，直到下一个yield，语句产生新的数据对象

- 如果该生成器函数内执行return

语句，则会触发StopInteration

异常，该异常表明生成数据结束

# 字节 字节串



# 文件操作

- 文本文件：存储的内容是文字内容

使用文本编辑器打开，可以看到完整的内容

- 二进制文件：存储的内容是一个一个字节

使用文本编辑器打开，人无法识别

4)文件操作 \*\*\*\*\*

- open: 打开文件

格式： open(file, mode='rt')

file 文件名/文件路径

mode 打开模式

返回值：返回文件流对象

文件打开模式：决定了文件的对象的类型

决定了可执行的操作

'r' 只读方式打开

'w' 只写方式打开，覆盖文件原内容若文件不存在，则创建一个文件，

并且以只写方式打开

'a' 以只写方式打开文件，写入的内容不会覆盖原内容，追加模式

'b' 以二进制模式打开文件

't' 以文本模式打开

'+' 为更新内容打开文件（可读可写）

'x' 创建新文件

- readline:

格式： data = f.readline(size = -1)

功能： 读取一行数据，如果到达文件尾返回空

参数： size -1表示读取数两不受限制

文本文件：表示最多读取size个字符

二进制文件：表示最多读取size个字节

返回值： 返回读取数据

对于文本文件，返回字符串

对于二进制文件，返回字节串

以换行符作为一行的结束

换行符：

- linux: '\n'

- windwos: '\r\n'

- 旧Macintosh: '\r'

- 新的MacOS: '\n'

- readlines:返回以行为单位的字符串/字节串列表

语法：data = f.readlines(hint = -1)

参数： hint -1 读取全部文件

文本文件 表示读取的最少字符个数

二进制文件 表示读取的最少的字节个数

- read: 从文件流对象中读取制定个数字符/字节数据

语法：f.read(size = -1)

参数：size 制定读取个数

-1表示不受限制

文本文件：读取最大字符数

二进制文件：读取最大字节数

- write: 向文件对象中写入制定个数的

字节数或字符数

语法：n = f.write(data)

参数: data 要写入的数据

文本文件：字符串

二进制：字节串

# **面向对象**

**面向对象和面向过程**

**类和对象**

**类变量**

**实例变量**

**类方法**

**实例方法**

**私有属性：**

**私有方法：**

**构造方法**

**析构方法**

**三个特征：封装继承多态**

**继承和派生**

**函数的重写**

**运算符重载**

## 面向对象和面向过程

举例：厨师

## 2. 类和对象

**“万物皆对象” “类也是对象” “类是对象的抽象化 对象是类的具体化”**

**举例：人类 张三 = People（）**

## 3.类里面的概念

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

有的属性和方法。对象是类的实例。**实例化：**创建一个类的实例，类的具体对象。

**对象(实例)：**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

**方法(代表行为)：**类中定义的函数。类中的方法与普通的函数只有一个特别的区别——它们必须有一个额外的**第一个参数名称**, 按照惯例它的名称是 self。

**变量(代表属性)：**在类的声明中，属性是用变量来表示的。分类类属性和实例属性.

类变量在整个实例化的对象中是公用的. 通常不作为实例变量使用。

**方法重写：**如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方

法的覆盖（override），也称为方法的重写。

## 4.封装继承多态

## 5.函数重写

函数重写是指在自定义的类内添加相应的方法,让自定义的类创建的实例可以像内建对象一样进行函数操作

内建函数重写

方法名 函数名

\_\_abs\_\_(self) abs(obj) 函数

\_\_len\_\_(self) len(obj) 函数

\_\_reversed\_\_(self) reversed(obj)

\_\_round\_\_(self) round(obj)

## 6.运算符重载

让自定义的类生成的对象(实例)能够使用运算符进行操作

算术运算符重载

方法名 运算符和表达式 说明

\_\_add\_\_(self, rhs) self + rhs 加法

\_\_sub\_\_(self, rhs) self - rhs 减法

\_\_mul\_\_(self, rhs) self \* rhs 乘法

\_\_truediv\_\_(self, rhs) self / rhs 除法

\_\_floordiv\_\_(self, rhs) self // rhs 地板除

\_\_mod\_\_(self, rhs) self % rhs 求余

\_\_pow\_\_(self, rhs) self \*\* rhs 幂运算