

# Python基础 C03



胡俊峰 2022/03/04

北京大学计算机学院

# 回顾一下上次课的重要内容

- 容器类（含生成器）对象配合高阶函数列表生成式常用于实现批量数据处理
- 函数嵌套定义为内层函数定义了私有的运行环境。
  - 嵌套函数返回内层函数实例，事实上构成了函数实例的生成。可用来实现函数的个性化定制。这种函数定义模式被称为函数闭包。所有被生成的函数实例共享母函数的私有环境。
    - 函数定制常见的一种方案是模板编程
    - 将个性化功能函数作为参数，共性功能抽象出来的闭包方案称为函数装饰器
      - 装饰器也是可以被更高层的闭包在运行中进行定制（带参数的装饰器）
  - 无论是闭包还是装饰器，都可以多层嵌套、组合构成复杂的函数定制方案

# 本次课主要内容

- ▀ ipython常用操作命令
- ▀ 对象的名字绑定、copy操作、引用计数
- ▀ 类定义与对象声明 (python类的基本用法)
- ▀ 内置函数 (魔法函数)
- ▀ 可调对象与类装饰器
- ▀ 类型定义应用实例 —— 树结构
- ▀ 类的继承
- ▀ 文件操作与异常

# ipython magic命令

python magic命令

ipython解释器提供了很多以百分号%开头的magic命令，这些命令很像linux系统下的命令行命令（事实上有些是一样的）。

查看所有的magic命令：

```
%lsmagic
```

Available line magics:

```
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cd  
%clear %cls %colors %conda %config %connect_info %copy %ddir %debug %dhist  
%dirs %doctest_mode %echo %ed %edit %env %gui %hist %history %killbgscripts  
%ldir %less %load %load_ext %loadpy %logout %logon %logstart %logstate %log  
stop %ls %lsmagic %macro %magic %matplotlib %mkdir %more %notebook %page %  
pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint %precis  
ion %prun %psearch %psource %pushd %pwd %pycat %pylab %qtconsole %quickref  
%recall %rehashx %reload_ext %ren %rep %rerun %reset %reset_selective %rmdir  
%run %save %sc %set_env %store %sx %system %tb %time %timeit %unalias %un  
load_ext %who %who_ls %whos %xdel %xmode
```

Available cell magics:

```
%%! %%HTML %%SVG %%bash %%capture %%cmd %%debug %%file %%html %%javascript  
%%js %%latex %%markdown %%perl %%prun %%pypy %%python %%python2 %%python3 %  
%ruby %%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile
```

Available line magics:

```
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cd
%clear %cls %colors %conda %config %connect_info %copy %ddir %debug %dhist
%dirs %doctest_mode %echo %ed %edit %env %gui %hist %history %killbgscripts
%ldir %less %load %load_ext %loadpy %logout %logon %logstart %logstate %log
stop %ls %lsmagic %macro %magic %matplotlib %mkdir %more %notebook %page %
pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint %precis
ion %prun %psearch %psource %pushd %pwd %pycat %pylab %qtconsole %quickref
%recall %rehashx %reload_ext %ren %rep %rerun %reset %reset_selective %rmdir
%run %save %sc %set_env %store %sx %system %tb %time %timeit %unalias %un
load_ext %who %who_ls %whos %xdel %xmode
```

Available cell magics:

```
%%! %%HTML %%SVG %%bash %%capture %%cmd %%debug %%file %%html %%javascript
%%js %%latex %%markdown %%perl %%prun %%pypy %%python %%python2 %%python3 %
%ruby %%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile
```

Automagic is ON, % prefix IS NOT needed for line magics.

line magic 以一个百分号开头, 作用与一行;

cell magic 以两个百分号开头, 作用于整个cell。

`line magic` 以一个百分号开头，作用与一行；

`cell magic` 以两个百分号开头，作用于整个cell。

使用 `whos` 查看当前的变量空间：

```
i = 5
a = 5
print(a is i)
j = 'hello world!'
a = 'hello world!'
print(a is j)
```

```
%whos
```

```
True
```

```
False
```

```
Variable      Type      Data/Info
```

```
-----
```

```
a             str      hello world!
```

```
b             list     n=4
```

```
i             int      5
```

```
j             str      hello world!
```

在Python中，整数和短小的字符，Python都会缓存这些对象，以便重复使用，不是频繁的建立和销毁。当创建多个等于整数常量的引用时，实际上是让这些引用会指向同一个对象

## 使用 `reset` 重置当前变量空间:

```
%reset -f
```

```
print (a)
```

---

```
-----  
-  
NameError                                Traceback (most recent call last)  
<ipython-input-12-a45fdcf41272> in <module>  
      1 get_ipython().run_line_magic('reset', '-f')  
      2  
----> 3 print (a)  
  
NameError: name 'a' is not defined
```

再查看当前变量空间:

```
%whos
```

```
Interactive namespace is empty.
```

## lpython下常用的一些操作：

%cd 修改目录 例： %cd c:\\data

%ls 显示目录内容

%load 加载代码

%save 保存cell

%%writefile 命令用于将单元格内容写入到指定文件中  
，文件格式可为txt、py等

%run 运行脚本

%run -d 交互式调试器

%timeit 测量代码运行时间 # %一行

%%timeit 测量代码运行时间 # %%一个代码块

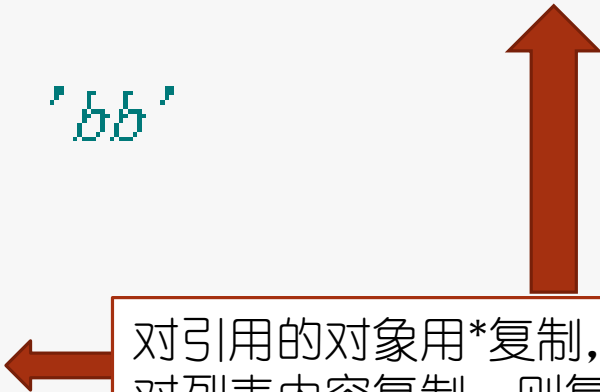


# 使用 `writetfile` 将cell中的内容写入文件:

```
%%writetfile test_magic.py
print ("%%开头的magic的作用区域延续到整个cell")

a = [3, 'aa', 34.4] * 4 # a = [[3, 'aa', 34.4]] * 4
print(a)
a[1] = 'bb'    #a[0][1] = 'bb'
print (a)

b = [{'k1': 1.5}] * 4
b[0]['k1'] = 10
print(b)
```



对引用的对象用\*复制，创建对象列表，复制引用，指向同一个对象  
对列表内容复制，则复制所有对象

Overwriting test\_magic.py

使用 `ls` 查看当前工作文件夹的文件:

使用 `run` 命令来运行这个代码:

```
: %ls
```

```
%run test_magic.py
```

驱动器 C 中的卷是 OS

卷的序列号是 8488-139B

C:\Users\hujf\2020notebooks\2020计概备课\Python\_Basics-master\python\_test 的目录

```
2020/10/21  06:47    <DIR>          .
2020/10/21  06:47    <DIR>          ..
2020/10/21  07:04                231 test_magic.py
```

1 个文件

231 字节

2 个目录 1,473,178,247,168 可用字节

%%开头的magic的作用区域延续到整个cell

```
[3, 'aa', 34.4, 3, 'aa', 34.4, 3, 'aa', 34.4, 3, 'aa', 34.4]
```

```
[3, 'bb', 34.4, 3, 'aa', 34.4, 3, 'aa', 34.4, 3, 'aa', 34.4]
```

```
[{'k1': 10}, {'k1': 10}, {'k1': 10}, {'k1': 10}]
```

## ipython 使用帮助命令

使用 `?` 查看函数的帮助, 或: 光标移动到方法上面, 按 `shift+tab`, 弹出文档, 连续按选择文档详细程度

```
In [63]: input?
```

使用 `??` 查看函数帮助和函数源代码 (如果是用python实现的) :

```
In [65]: # 查看其中sort函数的帮助
input??
```

**Signature:** `input(prompt='')`

**Source:**

```
def raw_input(self, prompt=''):
    """Forward raw_input to frontends
```

**Raises**

-----  
`StdinNotImplementedError` if active frontend doesn't support stdin.  
"""

```
if not self._allow_stdin:
    raise StdinNotImplementedError(
```

ipython 支持使用 `<tab>` 键弹出命令补全选择窗口。

使用 `_` 使用上个cell的输出结果：

```
a = 12
```

```
a
```

```
12
```

```
b = _ + 13
```

可以使用 `!` 来执行一些系统命令。

```
!ping baidu.com
```

```
正在 Ping baidu.com [220.181.38.148] 具有 32 字节的数据:
```

```
来自 220.181.38.148 的回复: 字节=32 时间=14ms TTL=46
```

```
来自 220.181.38.148 的回复: 字节=32 时间=6ms TTL=46
```

```
来自 220.181.38.148 的回复: 字节=32 时间=13ms TTL=46
```

```
来自 220.181.38.148 的回复: 字节=32 时间=16ms TTL=46
```

```
220.181.38.148 的 Ping 统计信息:
```

```
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
```

# python中对象名字绑定

- 变量、常量名本质都是引用，引用对应的是对象标识码(identity)
- 一旦对象被创建，它就有唯一的标识码。对象的标识码就不可更改。  
可以有内建函数 `id()` 获取

## python中的对象引用与对象标识码

```
1 a = dict(one=1, two=2, three=3)
2 b = {'one': 1, 'two': 2, 'three': 3}
3 d = dict([('two', 2), ('one', 1), ('three', 3)])
4 e = dict({'three': 3, 'one': 1, 'two': 2})
5 a == b == d == e
```

True

当用 == 操作符比较两个对象，是在比较他们的值是否相等

```
1 print ( a is b)
2 print ( id(a), id(b))
```

False

用is操作符比较两个对象时，就是在比较它们的ID是否相同，即是否是同一个对象

1777767271808 1777767294368

在Python中，整数和短小的字符，Python都会缓存这些对象，以便重复使用，不是频繁的建立和销毁。当创建多个等于1的引用时，实际上是让这些引用指向了同一个对象。

1	<code>a = 2.0</code>
2	<code>b = 2.0</code>
3	<code>a is b</code>

False

1	<code>a = 2</code>
2	<code>b = 2</code>
3	
4	<code>a is b</code>

True

## 对象复制 与 复制引用

```
1 li1 = [7, 8, 9, 10]
2
3 li2 = li1 # 传引用
4
5 li1[1] = 16
6
7 li2
```

li2 内容被同时改变

[7, 16, 9, 10]

```
1 b = [{'g': 1.5}] * 4 # 复制引用
2 print (b)
3
4 b[0]['g'] = '32'
5 print (b)
```


[{'g': 1.5}, {'g': 1.5}, {'g': 1.5}, {'g': 1.5}]  
[{'g': '32'}, {'g': '32'}, {'g': '32'}, {'g': '32'}]

```
1 b = [{'g': 1}] + [{'g': 1}] + [{'g': 1}] + [{'g': 1}] # 对象列表
2 b[0]['g'] = 2
3 b
```

加法 创建对象列表, 每个元素内容是独立的

[{'g': 2}, {'g': 1}, {'g': 1}, {'g': 1}]





# 拷贝机制

- 浅拷贝复制引用关系
- 深拷贝复制引用的关系及所引用的对象

## 浅拷贝 不拷贝子对象 原始数据改变 子对象会改变

```
: import copy
```

```
ls = [1, 2, 3, ['a', 'b']]
```

```
c = copy.copy(ls) ← 浅拷贝，复制容器内的引用
```

```
c
```

```
: [1, 2, 3, ['a', 'b']]
```

```
: ls[3].append('cccc')
```

```
ls.append(6)
```

```
ls
```

```
: [1, 2, 3, ['a', 'b', 'cccc'], 6]
```

```
: c
```

```
: [1, 2, 3, ['a', 'b', 'cccc']]
```

直接append列表 vs 改变列表元素的内容

深拷贝，复制容器中的对象引用，以及引用对象的内容的内容...

**深拷贝 包含对象里面的自对象的拷贝，所以原始对象的改变不会造成深拷贝里任何子元素的改变**

```
1 import copy
2
3 list = [1, 2, 3, ["a", "b"]]
4
5 d = copy.deepcopy(list)  #深拷贝，所引用的对象都重新生成
6
7 list.append(4)
8 list[3][0] = 'c'
9
10 print(d)
```

[1, 2, 3, ['a', 'b']] ← 深拷贝，所引用的容器对象内容也被生成副本

```
1 ls[3].append('cccc')
2
3 ls.append(6)
4 ls[3][0] = 'c'
5
6 print(ls)
7 print(c)
```

← 原对象内容被修改，不会被传递

← 所引用对象内容被修改

[1, 2, 3, ['c', 'b', 'cccc'], 6]  
[1, 2, 3, ['c', 'b', 'cccc']]

# 名字绑定 及 引用计数

- 名字是对一个对象的称呼，python将赋值语句认为是一个命名操作（或者称为名字绑定）。
- python中的所有对象都有引用计数
- 对象的引用计数在下列情况下会增加：
  - 赋值操作； 在一个容器（列表，序列，字典等等）中包含该对象
- 对象的引用计数在下列情况下会减少：
  - 离开了当前的名字空间（该名字空间中的本地名字都会被销毁）
  - 对象的一个名字被绑定到另外一个对象
  - 对象从包含它的容器中移除
  - 名字被显式地用del销毁（如：del i）
- 引用计数为0时会启动对象回收机制（递归引用会导致内存泄露）

```
from sys import getrefcount as grc # 引用计数
```

```
num1 = 2678
```

```
num2 = num1 + 1
```

```
print(grc(num1)) # 打印num1的引用计数
```

```
num3 = num1
```

```
print(grc(num1))
```

```
ref_dict = dict(globals()) # 获得全局引用表
```

```
print([ref for ref in ref_dict if ref_dict[ref] is num1]) # 查看全局表中引用num1的变
```

```
print(grc(num1)) # 再打印num3的引用计数
```

```
del num1
```

```
print(grc(num3))
```

通过Py\_IncRef(PyObject \*o),  
Py\_DecRef(PyObject \*o). 这对  
操作函数来动态调整每个对象实  
例的reference\_count属性值。


3

4

['num3', 'num1']

6

5



# 类的定义与对象声明

- 定义类及声明对象
- 实例属性、实例方法
- self参数与变量名作用域
- 对象的内省
- 类实例与类属性
- 类的私有属性与内置方法



# 定义一个类 class:

block # 属性、方法函数

- 类名通常首字母为大写。
- 类定义包含 属性 和 方法
- 其中对象方法（method）的形参self必不可少，而且必须位于最前面。但在实例中调用这个方法的时候不需要为这个参数赋值，Python解释器会提供**指向实例的引用**。

```
class Cat():
    def __init__(self, name, age):    # 采用__开始的为内部函数，init在创建实例的过程自动。
        self.name = name
        self.age = age
    def sit(self):                    # 外部可见的实例方法
        print(self.name.title() + " is now sitting.")
    def roll_over(self):
        print(self.name.title() + " rolled over!")

this_cat = Cat('胖橘', 6)    # 创建实例
print("这只猫的名字是： " + this_cat.name.title() + ".")
print("已经有" + str(this_cat.age) + " 岁了。")

that_cat = Cat('ketty', 3)    #
print("这只猫的名字是： " + that_cat.name.title() + ".") # title方法返回标题化的串（首字母大写）
print("有" + str(that_cat.age) + " 岁了。")
```

这只猫的名字是： 胖橘。  
已经有6 岁了。  
这只猫的名字是： Ketty。  
有3 岁了。



## self参数、实例方法 解读：

- ➡ 本质是一个占位符，用于显示的指明实例的私有名字空间。被\_\_init\_\_()方法填充
- ➡ 用来记录每个实例的引用（及派生链）
- ➡ 实例属性必须要用self.进行前缀性约束
- ➡ 实例方法的本质是实例内置函数的母函数

## 为实例对象添加新属性（一般不建议）

```
print(f'胖橘.catfood = {胖橘.catfood}') # 会优先访问实例属性
胖橘.catfood = 100 # 在对象空间创建了一个同名的实例属性
print(f'胖橘.catfood = {胖橘.catfood}')
```

```
Cat.catfood = 40
胖橘.eat()
卷尾.eat()
print(胖橘.catfood, 卷尾.catfood) # 会优先访问实例属性
```

```
胖橘.catfood = 100
胖橘.catfood = 100
catfood = 38
catfood = 36
100 36
```

# 为实例添加新的方法函数

```
: def eatm(self):  
    self.catfood -= 2  
    print(f'mycatfood = {self.catfood}')
```

```
: import types  
胖橘.eatmy = types.MethodType(eatm, 胖橘)
```

```
: 胖橘.eat()  
胖橘.eatmy()  
胖橘.catfood
```

```
catfood = 34  
mycatfood = 98
```

```
: 98
```

```
: class C(): pass
```

```
print ([x for x in dir(胖橘) if x not in dir(C)])  
#dir(胖橘)
```

```
['age', 'catfood', 'eat', 'eatmy', 'name', 'roll_over', 'sit']
```

## 对象的内省 ( introspection )

```
this_cat.sit()           # 调用实例方法

that_cat.roll_over()    # 此时self.name, title() 分别指向不同对象的name, title()

print(that_cat.__dict__)


class C:pass            # 定义一个空类
print(set(dir(this_cat)) - set(dir(C)))  # 列出个性化属性和方法名
```

胖橘 is now sitting.

Ketty rolled over!

```
{'name': 'ketty', 'age': 3}
```

```
{'name', 'roll_over', 'sit', 'age'}
```



# 类对象、类属性、类方法

- 类也是对象，因此可以有自己的属性和方法
- 类属性由该类 and 所有派生的对象实例（通过类名称访问）共享

```
class Cat():  
  
    catfood = 20           # 类属性  
    ...  
  
    def eat(self):  
        if Cat.catfood > 0:  
            Cat.catfood -= 2    # 访问所有对象共享的类属性  
            print("catfood =", Cat.catfood)  
  
    def roll_over(self):  
        print(self.name.title() + " rolled over!")  
        Cat.catfood -= 1
```

```
卷尾 = Cat('卷尾', 1)    # 创建实例  
胖橘 = Cat('胖橘', 6)    # 创建实例
```

```
胖橘.roll_over()  
胖橘.eat()  
卷尾.eat()  
卷尾.catfood
```

```
胖橘 rolled over!  
catfood = 17  
catfood = 15
```

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.__name = name
```

```
        self.age = age
```

```
    def printInfo(self):
```

```
        print(f' name: {self.__name}, age: {self.age}')
```

```
p = Person("Bob", 13)
```

```
p.printInfo()
```

```
name:Bob, age:13
```

私有变量不可以直接访问，共有变量可以直接访问

```
p.age
```

```
13
```

```
p.name
```

---

```
AttributeError
```

```
Traceback (most recent call last)
```

私有方法，在创建-初始化对象的时候被new()自动调用

# Python对象的私有变量和内置方法

- 默认情况下，Python中的成员函数和成员变量都是公开的(public)。在python中定义私有变量只需要在变量名或函数名前加上 一个（私有）或两个（伪私有）下划线，那么这个函数或变量就是(伪)私有的了
- 私有变量不可以直接访问，公有变量可以直接访问
- 伪私有变量可以通过 实例.\_\_类名\_变量名 格式来强制访问。

[\[Python\]Python中的私有变量 - 知乎 \(zhihu.com\)](#)



# 序列对象常用的一些内置方法：

## 行为方式与迭代器类似的类

序号	目的	所编写代码	Python 实际调用
①	遍历某个序列	<code>iter(seq)</code>	<code>seq.__iter__()</code>
②	从迭代器中获取下一个值	<code>next(seq)</code>	<code>seq.__next__()</code>
③	按逆序创建一个迭代器	<code>reversed(seq)</code>	<code>seq.__reversed__()</code>

← 可迭代对象

1. 无论何时创建迭代器都将调用 `__iter__()` 方法。这是用初始值对迭代器进行初始化的绝佳之处。
2. 无论何时从迭代器中获取下一个值都将调用 `__next__()` 方法。
3. `__reversed__()` 方法并不常用。它以一个现有序列为参数，并将该序列中所有元素从尾到头以逆序排列生成一个新的

## 定义一个迭代器类

```
class Foo:
    def __init__(self, n):
        self.n = n

    def __iter__(self): ← 返回一个迭代器实例
        return self

    def __next__(self):
        if self.n >= 8:
            raise StopIteration
        self.n += 1
        return self.n

f1 = Foo(5)

for i in f1:
    print(i)
```

6

7

8



```
class Infiter:
```

```
    step = 2
```

```
    def __init__(self, num):  
        self.n = num
```

```
    def __iter__(self):  
        Infiter.step = 3  
        return self
```

```
    def __next__(self):  
        self.n += Infiter.step  
        if self.n < 16:  
            return self.n  
        else:  
            raise StopIteration
```

```
f2 = Infiter(5)  
print(next(f2))  
print(next(f2))
```

```
for i in f2:  
    print(i)
```

7

9

12

15

序号	目的	所编写代码	Python 实际调用
	序列的长度	<code>len(seq)</code>	<code>seq.__len__()</code>
	了解某序列是否包含特定的值	<code>x in seq</code>	<code>seq.__contains__(x)</code>

序号	目的	所编写代码	Python 实际调用
	通过键来获取值	<code>x[key]</code>	<code>x.__getitem__(key)</code> ← 可hash对象
	通过键来设置值	<code>x[key] = value</code>	<code>x.__setitem__(key, value)</code>
	删除一个键值对	<code>del x[key]</code>	<code>x.__delitem__(key)</code>
	为缺失键提供默认值	<code>x[nonexistent_key]</code>	<code>x.__missing__(nonexistent_key)</code>

## 可重载的常见运算符函数：

序号	目的	所编写代码	Python 实际调用
	相等	<code>x == y</code>	<code>x.__eq__(y)</code>
	不相等	<code>x != y</code>	<code>x.__ne__(y)</code>
	小于	<code>x &lt; y</code>	<code>x.__lt__(y)</code>
	小于或等于	<code>x &lt;= y</code>	<code>x.__le__(y)</code>
	大于	<code>x &gt; y</code>	<code>x.__gt__(y)</code>
	大于或等于	<code>x &gt;= y</code>	<code>x.__ge__(y)</code>
	布尔上上下文环境中的真值	<code>if x:</code>	<code>x.__bool__()</code>

# 可调用对象：callable object

- `callable()`函数用来判定对象是否能被调用执行
- 普通数据：`callable("hello")` 返回 `False`
- 函数及类定义， `callable`返回`True`
- 普通对象实例：`callable`返回`False`
- 实现 `__call__()`方法的对象实例`callable`返回`True`
  - 类定义可以理解为`__call__()`方法派生出的所有可执行对象的公有运行环境

```
class LinePrint:
```

```
    def __init__(self, newline = '\n'):  
        self.line = 0  
        self.rt = newline
```

```
    def print(self, x):  
        print(self.line, x, end = self.rt)  
        self.line += 1
```

```
printf = LinePrint(" ")  
printf.print("e1")  
printf.print("e2")  
printf.print("e3")
```

```
print(callable(printf))
```

不可执行对象不能直接调用

```
printf("ss")      # TypeError: 'LinePrint' object is not callable
```

```
0 e1  1 e2  2 e3  False
```

In [27]: `class LinePrint:`

```
    def __init__(self, newline = '\n'):
```

```
        self.line = 0
```

```
        self.rt = newline
```

```
    def __call__(self, x):
```

```
        print(self.line, x, end = self.rt)
```

```
        self.line += 1
```

```
        return x
```

```
list(map(LinePrint(), [10, 20, 30])) # 派生一个可执行实例做函数参数
```

0 10

1 20

2 30

Out[27]: [10, 20, 30]



# 基于类实现的装饰器：

- 基于类装饰器的实现，必须实现 **call** 和 **init** 两个内置函数。 **init**：接收被装饰函数 func， **call ()**：保证是可调对象，同时在内部实现对输入函数的装饰逻辑

```
class logger(object):
    def __init__(self, func): # func为被装饰函数
        self.func = func

    def __call__(self, *args, **kwargs):
        print("[INFO]: the function {func}() is running..."
              .format(func=self.func.__name__)) # 装饰器逻辑
        return self.func(*args, **kwargs) # 这里返回被装饰函数

@logger
def say(something):
    print("say {}".format(something))

say("hello")
```

```
[INFO]: the function say() is running...
say hello!
```

```

class makeHtmlTagClass(object):
    def __init__(self, tag, css_class=""): # 可以接受装饰逻辑所需的参数
        self._tag = tag
        self._css_class = " class='{0}'".format(css_class) \
                               if css_class != "" else ""

    def __call__(self, fn): ← 装饰器闭包
        def wrapped(*args, **kwargs):
            return "<" + self._tag + self._css_class + ">" \
                   + fn(*args, **kwargs) + "</" + self._tag + ">"
        return wrapped

@makeHtmlTagClass(tag="b", css_class="bold_css")
@makeHtmlTagClass(tag="i", css_class="italic_css")
def say(someth):
    return "Hello, {}".format(someth)

print (say("这里是内容"))

```

<b class='bold\_css' ><i class='italic\_css' >Hello, 这里是内容</i></b>

# 例子：用类定义数据结构

- ➡ 单链表
- ➡ 树

## 例子1：单链表数据结构实现

```
1 class Node(object):  
2  
3     def __init__(self, value):  
4  
5         self.value = value  
6         self.nextnode = None
```

头指针

```
1 a = Node(1)  
2 b = Node(2)  
3 c = Node(3)
```

```
1 a.nextnode = b
```

```
1 b.nextnode = c
```

```
class BinaryTree(object):  
    def __init__(self, rootObj):  
        self.key = rootObj  
        self.leftChild = None  
        self.rightChild = None
```

二叉树结构的定义

```
def insertLeft(self, newNode):  
    if self.leftChild == None:  
        self.leftChild = BinaryTree(newNode)  
    else:  
        t = BinaryTree(newNode)  
        t.leftChild = self.leftChild  
        self.leftChild = t  
  
def insertRight(self, newNode):  
    if self.rightChild == None:  
        self.rightChild = BinaryTree(newNode)  
    else:  
        t = BinaryTree(newNode)  
        t.rightChild = self.rightChild  
        self.rightChild = t
```

## 二叉树结构的定义（续）

```
def getRightChild(self):  
    return self.rightChild
```

```
def getLeftChild(self):  
    return self.leftChild
```

```
def setRootVal(self, obj):  
    self.key = obj
```

```
def getRootVal(self):  
    return self.key
```



```
1  from __future__ import print_function
2
3  r = BinaryTree('a')
4  print(r.getRootVal())
5  print(r.getLeftChild())
6  r.insertLeft('b')
7  print(r.getLeftChild())
8  print(r.getLeftChild().getRootVal())
9  r.insertRight('c')
10 print(r.getRightChild())
11 print(r.getRightChild().getRootVal())
12 r.getRightChild().setRootVal('hello')
13 print(r.getRightChild().getRootVal())
```

a

None

<\_\_main\_\_.BinaryTree object at 0x104779c10>

b

<\_\_main\_\_.BinaryTree object at 0x103b42c50>

c

hello

# 类的继承

- BaseClassName (示例中的基类名) 必须与派生类定义在一个作用域内 (使用import即将其放入同一作用域内)
- 派生类的定义同样可以使用表达式
- 创建一个新的类实例。方法引用按如下规则解析: 搜索对应的类属性, 必要时沿基类链逐级搜索, 如果找到了函数对象这个方法引用就是合法的。

```
class DerivedClassName(BaseClassName):
```

```
    <statement-1>
```

```
    .
```

```
    .
```

```
    .
```

```
    <statement-N>
```

#同样也可以使用表达式

```
class DerivedClassName(modname.BaseClassName):
```



```
1  class Person(object):    # 定义一个父类
2
3      def talk(self):      # 父类中的方法
4          print("person is talking....")
5
6
7  class Chinese(Person):    # 定义一个子类, 继承Person类
8
9      def walk(self):       # 在子类中定义其自身的方法
10         print('is walking...')
11
12  c = Chinese()
13  c.talk()                 # 调用继承的Person类的方法
14  c.walk()                 # 调用本身的方法
```

```
person is talking....
is walking...
```

如果我们要给实例 c 传参，我们就要使用到构造函数，那么构造函数该如何继承，同时子类中又如何定义自己的属性？

\* 经典类的写法：父类名称.\_\_init\_\_(self, 参数1, 参数2, ...)

\* 新式类的写法：super(子类, self).\_\_init\_\_(参数1, 参数2, ....)

```
class Person(object):
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
        self.weight = 'weight'
```

```
    def talk(self):
```

```
        print("person is talking....")
```

```
class Chinese(Person):
```

```
    def __init__(self, name, age, language): # 先继承，再重构
```

```
        Person.__init__(self, name, age) #继承父类的构造方法，
```

```
        self.language = language # 定义类的本身属性
```

```
    def walk(self):
```

```
        print('is walking...')
```

## 子类对父类方法的重写，重写talk()方法

```
1  class Chinese(Person):
2
3      def __init__(self, name, age, language):
4          Person.__init__(self, name, age)
5          self.language = language
6          print(self.name, self.age, self.weight, self.language)
7
8      def talk(self): # 子类 重构方法
9          print('%s is speaking chinese' % self.name)
10
11     def walk(self):
12         print('is walking...')
13
14  c = Chinese('Xiao Wang', 22, 'Chinese')
15  c.talk()
```

Xiao Wang 22 weight Chinese

Xiao Wang is speaking chinese

继承关系构成了一张有向图，Python3 中，调用 `super()`，会返回广度优先搜索得到的第一个符合条件的函数。观察如下代码的输出也许方便你理解：

```
1 class A:
2     def foo(self):
3         print('called A.foo()')
4
5 class B(A):
6     pass
7
8 class C(A):
9     def foo(self):
10        print('called C.foo()')
11    def foo2(self):
12        super().foo()
13
14 class D(B, C):
15     pass
16
17 d = D()
18 d.foo()
19 d.foo2()
```

```
called C.foo()
called A.foo()
```

# 类方法、静态方法


```
# 实现多个初始化函数
class Book(object):

    def __init__(self, title):
        self.title = title

    # @classmethod                                # 注释掉以后在调用时要显示的输入类引用作为参数
    def class_method_create(cls, title):
        book = cls(title=title)
        return book

    @staticmethod
    def static_method_create(title):
        book = Book(title)
        return book

book1 = Book("use instance_method_create book instance")
book2 = Book.class_method_create(Book, "use class_method_create book instance")
book3 = Book.static_method_create("use static_method_create book instance")
print(book1.title)
print(book2.title)
print(book3.title)
```



```
class Foo(object):
```

```
    X = 1
```

```
    Y = 14
```

```
    @staticmethod
```

```
    def averag(*mixes): # "父类中的静态方法"
```

```
        return sum(mixes) / len(mixes)
```

```
    @staticmethod
```

```
    def static_method(): # "父类中的静态方法"
```

```
        print("父类中的静态方法") |
```

```
        return Foo.averag(Foo.X, Foo.Y)
```

静态方法由于不使用相对引用来标定参数，因此不会随着继承到新环境而改变运算逻辑

```
    @classmethod
```

```
    def class_method(cls): # 父类中的类方法
```

```
        print("父类中的类方法")
```

```
        return cls.averag(cls.X, cls.Y)
```

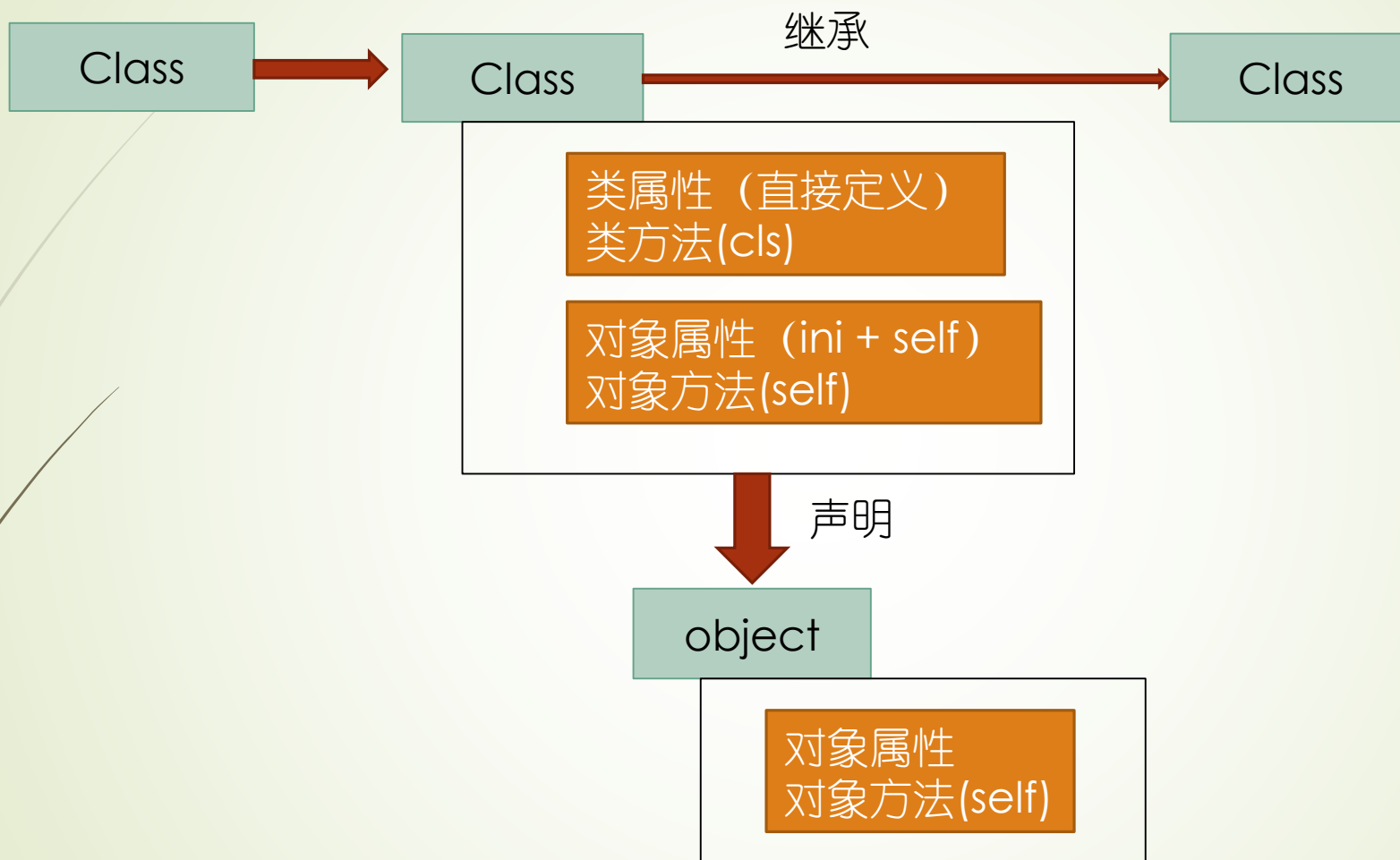
类方法由cls参数自动带入类的环境引用，因此会随着继承到新环境而改变运算逻辑

```
class Son(Foo):
```

```
    X = 3
```

```
    Y = 5
```

# Python的类与对象的概念直觉



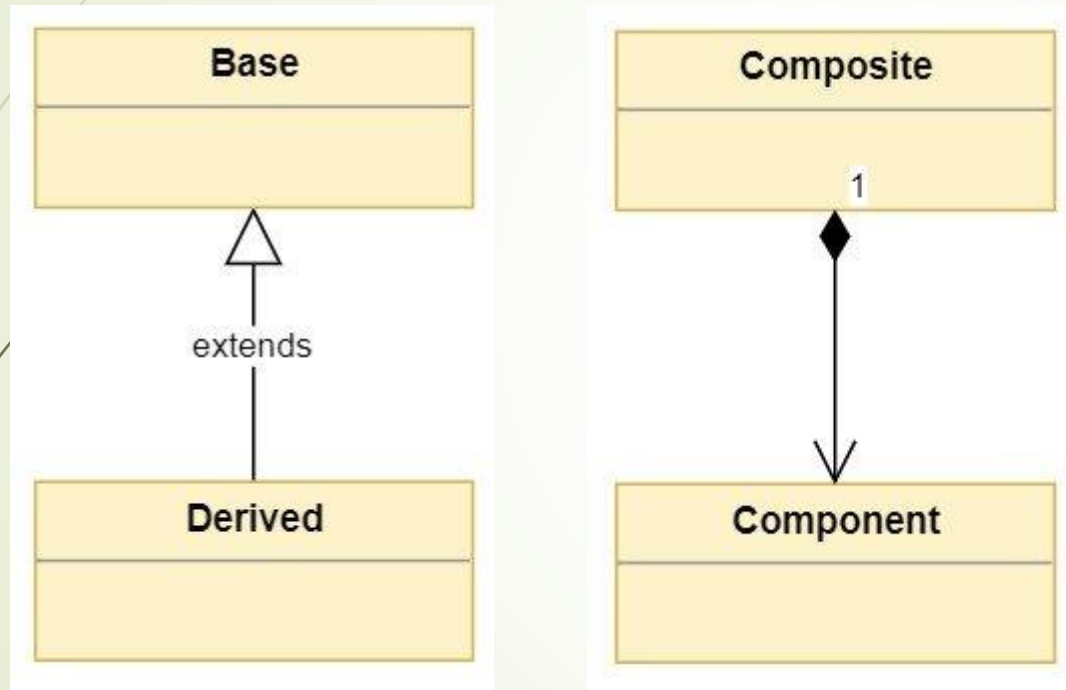


# 继承与组合 (Inheritance and Composition)

- ➡ Is a kind of: 派生
- ➡ Has a kind of: 组合



# 组合 Composition




```
class Turtle:
    def __init__(self,x):
        self.num = x

class Fish:
    def __init__(self,x):
        self.num = x

class Pool:
    def __init__(self,x,y):
        self.turtle = Turtle(x)
        self.fish = Fish(y)

    def number(self):
        print("水池里总共%s只乌龟，共%s条鱼" % (self.turtle.num,self.fish.num))
```



# Python的文件操作

- lpyhton文件操作
- 文本文件读写
- 字节文件操作

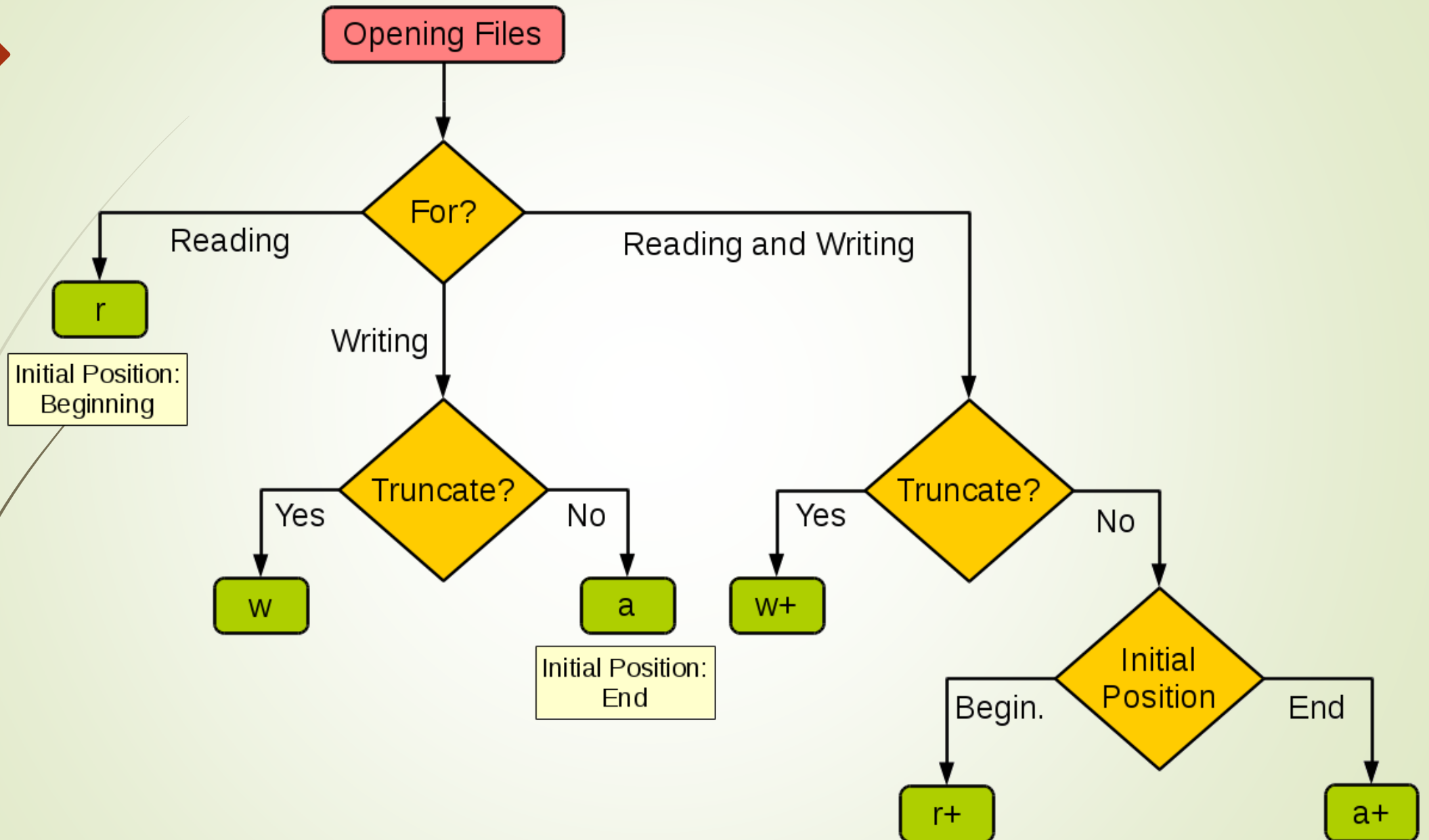
# 文件读写

可以参考对比C语言文件操作

python通过 `open()` 函数打开一个文件对象，一般的用法为 `open(filename, mode)`，其完整定义为 `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)`。

`filename` 是打开的文件名，`mode` 的可选值为：

- t 文本模式 (默认)。
- x 写模式，新建一个文件，如果该文件已存在则会报错。
- b 二进制模式。
  - 打开一个文件进行更新(可读可写)。
- r 以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
- rb 以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。一般用于非文本文件如图片等。
- r+ 打开一个文件用于读写。文件指针将会放在文件的开头。
- rb+ 以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。一般用于非文本文件如图片等。
- w 打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
- wb 以二进制格式打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。一般用于非文本文件如图片等。
- w+ 打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。



## 文件读取

```
1 # readlines() 将会把文件中的所有行读入到一个数组中
2 f = open('test_input.txt')
3 print(f.readlines())
```

```
['testline1\n', 'testline2\n', 'test line 3\n']
```

```
1 # read() 将读入指定字节数的内容
2 f = open('test_input.txt')
3 print(f.read(8))
```

```
testline
```

```
1 # 但是一般情况下, 我们
2 f = open('test_input.txt')
3 for line in f:
4     print(line)
```

```
testline1
```

```
testline2
```

```
test line 3
```

```
1 # 这种读入方法同样会保留行尾换行, 结合print() 自带的换行,
2 # 打印后会出现一个间隔的空行
3 # 所以一般我们读入后, 会对line做一下strip()
4 f = open('test_input.txt')
5 for line in f:
6     print(line.strip())
```

```
testline1
```

```
testline2
```

```
test line 3
```

## 向文件写入

python中，通过文件对象的 `write()` 方法向文件写入一个字符串。

```
1 of = open('test_output.txt', 'w')
2 of.write('output line 1')
3 of.write('output line 2\n')
4 of.write('output line 3\n')
5 of.close()
```



## 字节文件的直接存取

```
f = open('test_input.txt', 'rb+')
f.write(b'sds0123456789abcdef')
f.seek(5)          # Go to the 6th byte in the file
print(f.read(1))
print(f.tell())
f.seek(-3, 2)      # Go to the 3rd byte from the end 0-1-2
print(f.read(1))
f.close()
```

b表示字节

b' 2'  
6  
b' d'

Whence: 0代表从文件开头开始算起,  
1代表从当前位置开始算起,  
2代表从文件末尾算起



## 上下文管理器：with


```
1 with open('test_input.txt') as myfile:  
2     for line in myfile:  
3         print(line)  
4 myfile.closed == 1
```

← 退出自动关闭文件

sds0123456789abcdef

hello world!

True



# Python的异常处理

- 常规的异常处理流程
- 自定义与触发异常

# Python Errors and Built-in Exceptions

- 错误处理导致异常：软件的结构上有错误，导致不能被解释器解释或编译器无法编译。这些些错误必须在程序执行前纠正。
- 程序逻辑或不完整或不合法的输入、值域不合法导致运行流程异常；

# 语法错误、值域溢出或无法执行导致异常

```
# We can notice here that a colon is missing in the if statement.
```

```
if a < 3
```

```
File "<ipython-input-5-607a69f69f94>", line 1
```

```
    if a < 3
```

```
        ^
```

```
SyntaxError: invalid syntax
```

```
# ZeroDivisionError: division by zero
```

```
1 / 0
```

```
ZeroDivisionError
```

```
t call last)
```

```
<ipython-input-6-b710d87c980c> in <module>()
```

```
----> 1 1 / 0
```

```
ZeroDivisionError: division by zero
```

```
# FileNotFoundError
```

```
open("imaginary.txt")
```

```
FileNotFoundError
```

```
Traceback (m
```

```
t call last)
```

```
<ipython-input-7-1f07e636ec19> in <module>()
```

```
----> 1 open("imaginary.txt")
```

```
FileNotFoundError: [Errno 2] No such file or directory  
'ary.txt'
```

```
Traceback (most recen
```

# 内建异常处理流程：

- 异常：是因为程序出现了错误而在正常控制流以外采取的行为，python用异常对象(exception object)来表示异常。遇到错误后，会引发异常。
- 两个阶段：
  - 引起异常发生的错误，
  - 检测（和采取可能的措施）阶段。
- 当前流将被打断，用来处理这个错误并采取相应的操作。这就是第二阶段，异常引发后，调用很多不同的操作可以指示程序如何执行。
- 如果异常对象并未被处理或捕捉，程序就会用所谓的回溯（traceback）终止执行
- 我们可以使用local（）.\_\_builtins\_\_来查看所有内置异常，如右图所示。

```
ans = locals()['__builtins__'].__dict__
for k, v in ans.items():
    if "Error" in k:
        print(k, v)
```

```
TypeError <class 'TypeError'>
ImportError <class 'ImportError'>
ModuleNotFoundError <class 'ModuleNotFoundError'>
OSError <class 'OSError'>
EnvironmentError <class 'OSError'>
IOError <class 'OSError'>
EOFError <class 'EOFError'>
RuntimeError <class 'RuntimeError'>
RecursionError <class 'RecursionError'>
NotImplementedError <class 'NotImplementedError'>
NameError <class 'NameError'>
```

## Python Built-in Exceptions

Exception	Cause of Error
AssertionError	Raised when <code>assert</code> statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the <code>input()</code> functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's <code>close()</code> method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.

# Python 异常处理流程

- 当有异常出现时，它会使当前的进程停止，并且将异常传递给调用进程，直到异常被处理为止。
- 如：function A → function B → function C
- function C 中发生异常. 如果C没有处理，就会层层上传到B，再到A

```
def C(x):  
    x / (x-x)  
def B(x):  
    C(x)  
  
def A(x):  
    B(x)
```

```
A(2)
```

```
-----  
ZeroDivisionError                                Traceback (most recent call last)  
<ipython-input-1-cb9f0c9139a7> in <module>()  
      7     B(x)  
      8  
----> 9 A(2)  
  
<ipython-input-1-cb9f0c9139a7> in A(x)  
      5  
      6 def A(x):  
----> 7     B(x)  
      8  
      9 A(2)  
  
<ipython-input-1-cb9f0c9139a7> in B(x)  
      2     x / (x-x)  
      3 def B(x):  
----> 4     C(x)  
      5  
      6 def A(x):  
  
<ipython-input-1-cb9f0c9139a7> in C(x)  
      1 def C(x):  
----> 2     x / (x-x)  
      3 def B(x):  
      4     C(x)  
      5
```

```
ZeroDivisionError: division by zero
```



# Python中捕获与处理异常：try: ... except \* :

- 在Python中，可以使用try语句处理异常。
- 可能引发异常的关键操作放在try子句中，并且将处理异常的代码编写在except子句中。
- 如果没有异常发生，则跳过Except的内容，并继续正常流程。但是，如果发生任何异常，它将被Except捕获

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occured.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

```
The entry is a
Oops! <class 'ValueError'> occured.
Next entry.
```

```
The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.
```

```
The entry is 2
The reciprocal of 2 is 0.5
```



## Except可以指定要捕获的异常类型:

```
try:
    # do something
    pass

except ValueError:
    # handle ValueError exception
    pass

except (TypeError, ZeroDivisionError):
    # handle multiple exceptions
    # TypeError and ZeroDivisionError
    pass

except:
    # handle all other exceptions
    pass
```

```
for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
    except ValueError:
        print("Value Error")
    except (ZeroDivisionError):
        print("ZeroDivision Error")
print("The reciprocal of", entry, "is", r)
```

```
The entry is a
Value Error
The entry is 0
ZeroDivision Error
The entry is 2
The reciprocal of 2 is 0.5
```

# 主动触发异常 Raising Exceptions

- 在Python编程中，当运行时发生相应的错误时会引发异常，但是我们可以使用关键字raise强制引发它。
- 我们还可以选择将值传递给异常，以阐明引发该异常的原因。

```
try:
    a = int(input("Enter a positive integer: "))
    if a <= 0:
        raise ValueError(f"{a} is not a positive number!")
except ValueError as ve:
    print(ve)
```

```
Enter a positive integer: -3
-3 is not a positive number!
```

# Try...finally语句

- Python中的try语句可以有一个可选的finally子句。该子句无论如何执行，通常用于释放外部资源。
- 例如，我们可能通过网络或使用文件或使用图形用户界面（GUI）连接到远程数据中心。
- 在所有这些情况下，无论资源是否成功，我们都必须清除该资源。这些操作（关闭文件，GUI或与网络断开连接）在finally子句中执行，以确保执行

```
try:
    f = open("test.txt",encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

```
-----
-----
FileNotFoundError                                Traceback (most recent call last)
<ipython-input-17-5a8f24f64426> in <module>()
```

```
1 try:
----> 2     f = open("test.txt",encoding = 'utf-8')
3     # perform file operations
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'test.txt'
```

During handling of the above exception, another exception occurred:

```
NameError                                Traceback (most recent call last)
<ipython-input-17-5a8f24f64426> in <module>()
```

```
3     # perform file operations
4 finally:
----> 5     f.close()
```

```
NameError: name 'f' is not defined
```