

Python基础 C03 — 类



胡俊峰 北京大学

2023/03/02

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 `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]
```

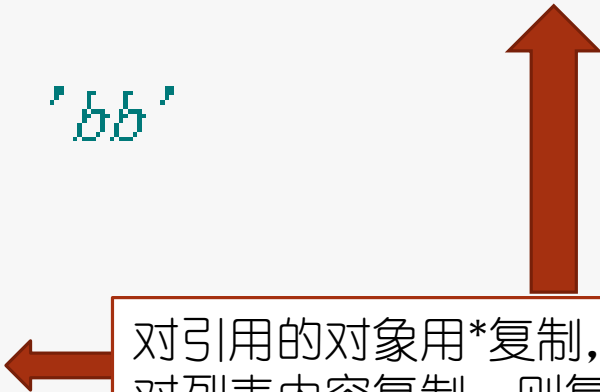
```
[{'k1': 10}, {'k1': 10}, {'k1': 10}, {'k1': 10}]
```


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

Python的模块 (Modules)

- 是以.py文件组织的实现特定功能的预定义的函数或环境变量代码
- 可以用import (路径+文件名) 的形式加载到当前代码环境中

```
[1]: 1 %%writefile calc.py
      2
      3 def mod10sum(li):
      4     return int(sum(li)) % 10
      5
      6 def modXsum(li, x = 10):
      7     int(x)
      8     return int(sum(li)) % x
```

Overwriting calc.py

```
[3]: 1 import calc
      2
      3 w = [1, 2, 3, 4, 5, 6, 7]
      4
      5 print(calc.mod10sum(w)) # 要加上模块名前缀
      6 print(calc.modXsum(w, 8))
```

8

4

直接加载模块中的对象：

```
1 w =[1, 2, 3, 4, 5, 6, 7]
2
3 # from calc import *
4 from calc import mod10sum, modXsum
5
6 print(mod10sum(w))    # 直接用函数或命令名引用
7 print(modXsum(w, 3))
```

8

1

__name__属性

- 模块（.py文件）在创建之初会自动加载一些内建变量，__name__就是其中之一
- **if __name__ == '__main__':**
保护模块私有的执行（调试）代码不被包含到其他模块中

```
1  #只有当文件被当作脚本执行的时候, __name__的值才会是 '__main__',  
2  #if __name__ == '__main__':  
3  #    localtest()  
4  
5  print(calc.__name__)
```

calc

包 (package) : 按层级目录组织的模組集合

```
sound/                                Top-level package
__init__.py                          Initialize the sound package
formats/                             Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
effects/                             Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
filters/                             Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...
```

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

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

对象复制 VS 对象引用的复制

```
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}]

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!")
```

定义类，创建独立的数据对象

```
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、实例属性 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__()方法赋值。

```
1  this_cat.sit()           # 调用实例方法：加入了 print(self) 语句
2
3  that_cat.roll_over()    # 此时self.name.title() 分别指向不同对象的name字段
4
5  print(this_cat.__dict__)
6  print(that_cat.__dict__)
7
8  class C:pass            # 定义一个空类
9  print(set(dir(this_cat)) - set(dir(C))) # 列出个性化属性和方法名
```

胖橘 is now sitting.

<__main__.Cat object at 0x00000231121187F0>

Ketty rolled over!


<__main__.Cat object at 0x0000023112118A30>

{'name': '胖橘', 'age': 6}

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

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

对象的自省 (introspection)



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
```

```

1 class Cat():
2
3     catfood = 20          # 类属性
4
5     @classmethod
6     def eat(cls):          # 这里改成CLS指针: 类方法
7         if cls.catfood > 0:
8             cls.catfood -= 3    # 访问自身对象属性
9             print("cls.catfood =", Cat.catfood)
10
11     def __init__(self, name, age):    # 采用__开始的为内部函数
12         self.name = name
13         self.age = age
14
15     def sit(self):                # 外部可见的实例方法
16         print(self.name.title() + " is now sitting.")
17
18     def roll_over(self):
19         print(self.name.title() + " rolled over!")
20         Cat.catfood -= 1
21         print("Cat.catfood =", Cat.catfood)
22
23     卷尾 = Cat('卷尾', 1)    # 创建实例
24     胖橘 = Cat('胖橘', 6)    # 创建实例

```

为实例对象添加新属性 (一般不建议)

```

1 胖橘.eat()
2 卷尾.eat()
3 print(卷尾.catfood, 胖橘.catfood)
4 胖橘.catfood -= 5
5 print(卷尾.catfood, 胖橘.catfood)
6 胖橘.eat()
7 卷尾.eat()
8 胖橘.roll_over()
9 print(卷尾.catfood, 胖橘.catfood)

```

```

cls.catfood = 17
cls.catfood = 14
14 14
14 9
cls.catfood = 11
cls.catfood = 8
胖橘 rolled over!
Cat.catfood = 7
7 9

```

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

为实例添加新的方法函数

```
1 def eatm(self):
2     self.catfood-=2
3     print('my cat food =', self.catfood)
4
5 import types
6 胖橘.eatmy = types.MethodType(eatm, 胖橘)
7
8 胖橘.eatmy()
9 print(胖橘.catfood)
```

```
my cat food = 7
7
```

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

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

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。

```
1 胖橘.eat()  
2 卷尾.eat()  
3 print(卷尾._Cat__catfood, 胖橘._Cat__catfood)  # AttributeError: 'Cat' object has no attribute '__catfood'  
4 胖橘.roll_over()
```

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 < y</code>	<code>x.__lt__(y)</code>
	小于或等于	<code>x <= y</code>	<code>x.__le__(y)</code>
	大于	<code>x > y</code>	<code>x.__gt__(y)</code>
	大于或等于	<code>x >= 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f，
- **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

```
1 class Memoize:
2     def __init__(self, f):
3         self.f = f    # 被装饰函数
4         self.memo = {}
5     def __call__(self, *args):
6         if not args in self.memo:
7             self.memo[args] = self.f(*args)    # 执行被装饰函数
8             print(args, 'not in; ', end = '')
9         return self.memo[args]                # 直接返回结果
```



```
In [52]: 1 class Memoize:
2         def __init__(self, f):
3             self.f = f    # 被装饰函数
4             self.memo = {}
5         def __call__(self, *args):
6             if not args[0] in self.memo:
7                 self.memo[args[0]] = self.f(*args)    # 执行被装饰函数
8                 print(args[0], 'not in; ', end = '')
9             return self.memo[args[0]]                # 直接返回结果
```

```
In [53]: 1 def factorial(k):    # 定义一个需要被装饰的函数
2
3         if k < 2:
4             return 1
5
6         return k * factorial(k - 1)
7
8 factorial = Memoize(factorial)    # 实例化一个可执行对象
```

```
In [54]: 1 print('\n', factorial(4))
2         factorial(5)
```

```
1 not in; 2 not in; 3 not in; 4 not in;
24
5 not in;
```

Out[54]: 120

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

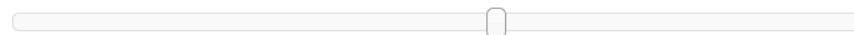
Python 3.6
[known limitations](#)

```
1 class Memoize:
2     def __init__(self, f):
3         self.f = f
4         self.memo = {}
5     def __call__(self, *args):
6         if not args in self.memo:
7             self.memo[args] = self.f(*args)
8         return self.memo[args]
9
10 def factorial(k):
11
12     if k < 2:
13         return 1
14
15     return k * factorial(k - 1)
16
17 factorial = Memoize(factorial)
18 factorial(4)
19 factorial(5)
```

[Edit this code](#)

→ line that just executed

→ next line to execute



<< First < Prev Next > Last >>

Step 34 of 58

Visualized with pythontutor.com

NEW: [subscribe](#) to our YouTube

[Move and hide objects](#)

Frames

Objects

Global frame

Memoize

factorial

__call__

self

args

factorial

k

__call__

self

args

factorial

k

__call__

self

args

factorial

k

__call__

self

args

Memoize class

__call__

function

__call__(self, *args)

__init__

function

__init__(self, f)

Memoize instance

f

function

factorial(k)

memo

dict

0

4

0

3

0

2

0

1

tuple


0

1

tuple

0

1



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()
```

静态方法：不会被重新创建，直接按名引用

```
# 实现多个初始化函数
class Book(object):

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

    # @classmethod
    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)
```

```
    @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)
```

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

```
    X = 3
```

```
    Y = 5
```

静态方法设定为恒定的当前运行环境，
类方法的运算环境可以随着继承关系而进化

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

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

Python设计模式

- 设计模式是面型对象方法里的一种解决方案的抽象
- 目的是把一些常见的应用抽象为一种类设计模式，在具体实现中只要套用或稍作修改，就能完成逻辑清晰的类实现方案
- 通过对设计模式的学习也可以达到对类体系的深入理解

单例模式 — 所有生成实例都指向同一个对象

```
class Singleton(object):  
  
    attr = None                # 类属性  
  
    def __init__(self):  
        print("Do something.") ← 调用 __new__()方法创建实例对象  
  
    def __new__(cls, *args, **kwargs):                # 重载__new__()方法  
        if not cls.attr:  
            cls.attr = super(Singleton, cls).__new__(cls) ←  
  
        return cls.attr  
  
obj1 = Singleton()  
obj2 = Singleton()  
print(obj1, obj2)
```

Do something.

Do something.

<__main__.Singleton object at 0x00000215E55BC308> <__main__.Singleton object at 0x00000215E55BC308>

单例模式 — 所有生成实例都指向同一个对象

```
class Singleton(object):
    def __init__(self):
        print("Do something new.")

    def __new__(cls, *args, **kwargs):
        if not hasattr(Singleton, "_instance"):
            Singleton._instance = object.__new__(cls)

        return Singleton._instance
```

在cls空间动态生成一个内部属性
(另一种实现单例模式的方案)

```
obj1 = Singleton()
obj2 = Singleton()
print(obj1, obj2)
```

Do something new.

Do something new.

<__main__.Singleton object at 0x00000215E55B2A88> <__main__.Singleton object at 0x00000215E55B2A88>

instances = {} # 一个全局的 类-实例 对的记忆buffer

```
def singleton(cls): # 接受一个类
    def get_instance(*args, **kw):
        cls_name = cls.__name__ # 获得当前类名称
        print('已经创建过了')
        if not cls_name in instances: # 过去没有创建实例
            print('第一次创建')
            instance = cls(*args, **kw) # 创建实例
            instances[cls_name] = instance # 加入记忆buffer
        return instances[cls_name] # 返回实例引用
    return get_instance # 返回装饰器函数
```

@singleton

class User:

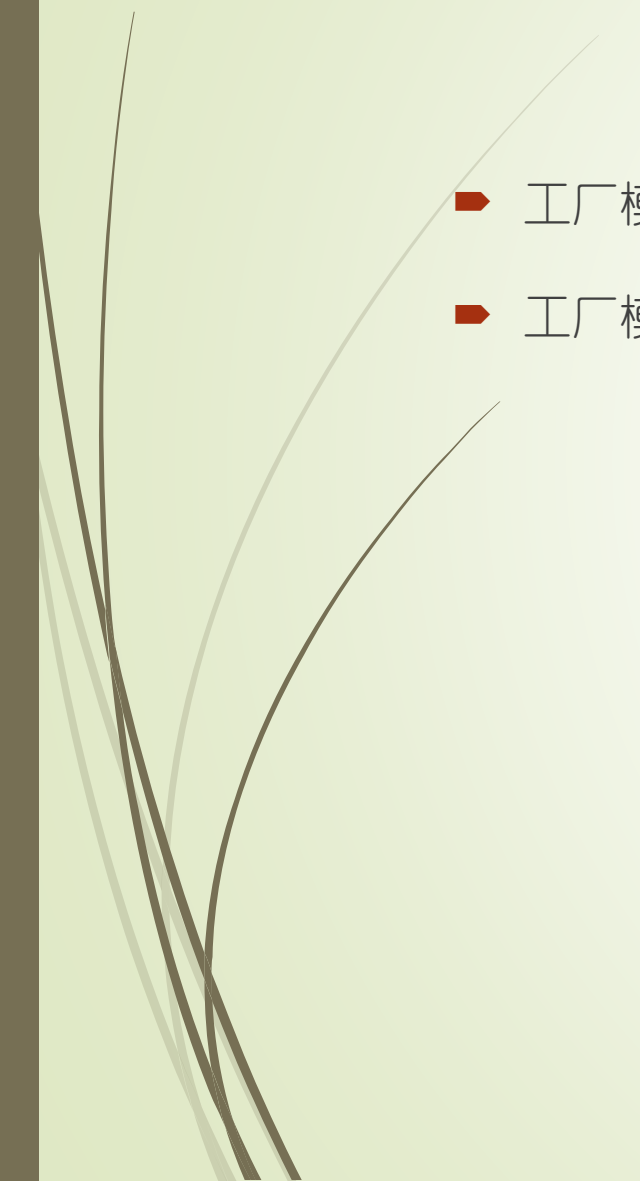
 _instance = None //?

```
    def __init__(self, name):
        print('==== 3 ====')
        self.name = name
```

User("xiao wang")



工厂模式简介：用来实现参数化定制类

- 工厂模式的本质上是用类定制类，然后到具体实例
 - 工厂模式的基础是共性抽象，是把相关类的共性和个性化定制相融合的解决方案
- 

工厂类的示例：

```
class StandardFactory(object):
```

```
    @staticmethod                # 静态方法
```

```
    def get_factory(factory):    # 实际可以传入更多的参数
```

```
        ''' 根据参数找到对实际操作的工厂'''
```

```
        if factory == 'cat':
```

```
            return CatFactory() # 这里如果要带参数，就会用到类属性，类方法
```

```
        elif factory == 'dog':
```

```
            return DogFactory()
```

```
        raise TypeError('Unknown Factory.')
```

```
class DogFactory(object):
```

```
    def get_pet(self): # 这里还可以带参数，甚至组合其他类，来定义不同类的dog
```

```
        return Dog(); # 返回一个dog类的实例
```

```
class CatFactory(object):
```

```
    def get_pet(self):
```

```
        return Cat();
```

工厂类的示例（抽象类）：

```
class Pet(abc.ABC):          # 抽象类可以通过MyIterable方法来查询所有的派生子类
    @abc.abstractmethod      # 强制子类必须实现此方法
    def eat(self):
        pass

    def jump(self):           # 不能创建实例，但可以被继承
        print("jump...")

# Dog类的具体实现
class Dog(Pet):
    def eat(self):           # 必须实现抽象类Pet中规定的方法
        return 'Dog eat...'

class Cat(Pet):
    def eat(self):
        return 'Cat eat...'
```

工厂类的示例（实际使用）：

```
if __name__ == "__main__":    # 如果被包含则 __name__ 会等于模块名，下面代码不会执行
    factory = StandardFactory.get_factory('cat')    # 配置抽象工厂参数，生成一个猫工厂
    cat = factory.get_pet()    # 生成一个猫实例
    print (cat.eat())    # cat eat

    factory = StandardFactory.get_factory('dog')
    dog = factory.get_pet()    ##这里工厂的操作与上面的生成cat是完全一样的，但结果不同
    print (dog.eat())    # dog eat
    dog.jump()    # 继承自抽象类的jump
    cat.jump()
    #Pet().jump() TypeError: Can't instantiate abstract class Pet with abstract
```

Cat eat...

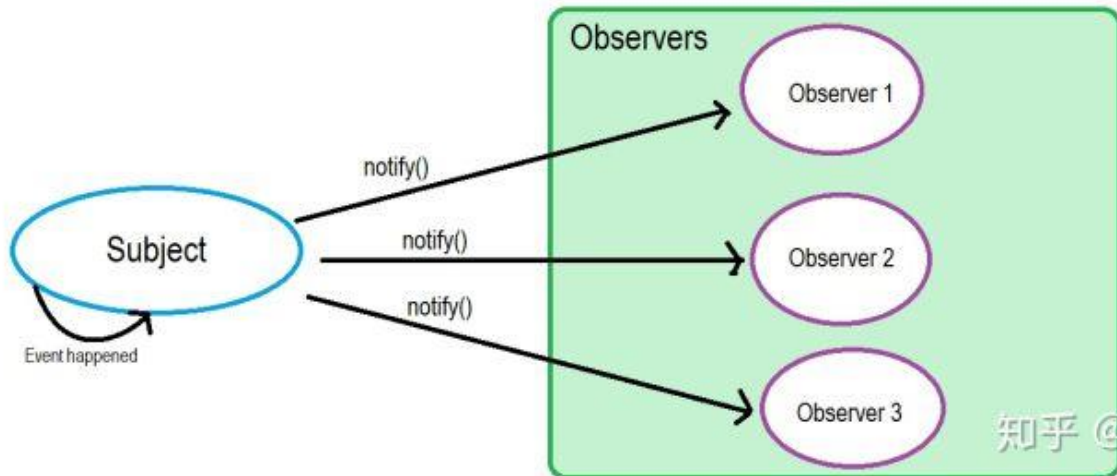
Dog eat...

jump...

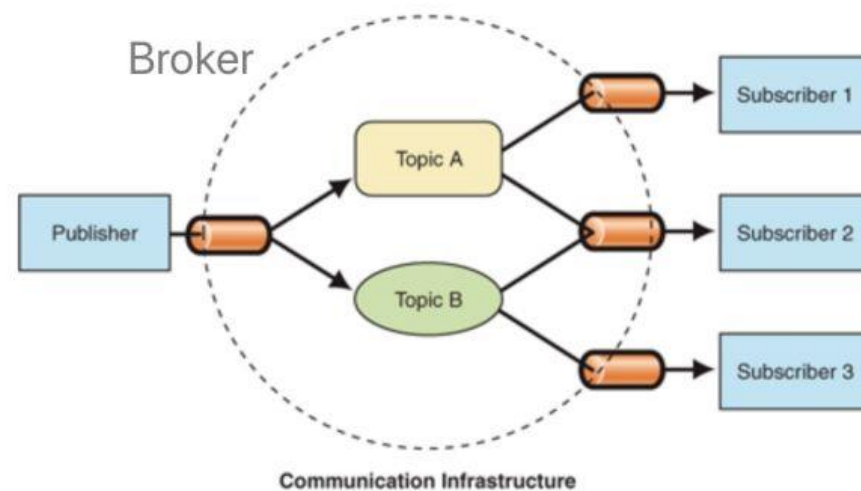
jump...

观察者模式（发布-订阅模式？）

- 多使用一种“注册—通知—撤销注册”的形式
- Observer) 将自己注册到被观察对象 (Subject) 中，被观察对象将观察者存放在一个容器 (Container) 里
- 被观察对象发生了某种变化，从容器中得到所有注册过的观察者，将变化通知观察者



知乎 @柳树



Pub-Sub Pattern (image credit: MSDN blog)

知乎 @柳树

```
class Subject:
    """ 某主题 """
    def __init__(self):
        self.observers = []

    def add_observers(self, observer):
        self.observers.append(observer) # 这里利用了list的append方法
        return self

    def remove_observer(self, observer):
        self.observers.remove(observer)
        return self

    def notify(self, msg):
        for observer in self.observers:
            observer.update(msg)

xiaoming = Observer("xiaoming")
lihua = Observer("lihua")

rain = Subject() # 生成主题。可以有主题词?

# 添加订阅
rain.add_observers(xiaoming)
rain.add_observers(lihua)

rain.notify("下雨了!")

# 取消订阅
rain.remove_observer(lihua) # 可以主动订阅? 条件约束订阅?

xiaoming收到信息: 下雨了!
lihua收到信息: 下雨了!
```


命令模式：

- 命令模式是一种行为设计模式，他用于封装触发事件（完成任何一个操作）所包含的所有信息。初衷是用于适配复合交互指令的需要。
 - 优点
 - 把调用操作的类与执行该操作分离（解耦合，多了一个任务管理前台）
 - 结合队列可以更加灵活的构造新命令
 - 添加新命令不用改现有代码框架
 - 可以实现用命令模式定义层级回滚系统
 - 缺点
 - 体系结构复杂度高
 - 每个单独的命令都是一个类，增加了实现和维护的类的数量

Python的类与对象:

