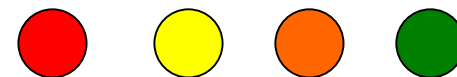


# Python基础 C03 — 类

信息科学技术学院

胡俊峰



# 本次课主要内容

- 文件操作、异常处理
- 类定义与对象声明（python类的基本用法）
- 可调用对象与类装饰器
- 类型定义应用实例 —— 树结构
- 类的继承
- Ipython常用内置函数（魔法函数）



# 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。



## • 文本文件读取

```
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, 我们会把file
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) ← 退出自动关闭文件  
4 myfile.closed == 1
```

sds0123456789abcdef





# 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。该子句无论如何都会执行，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
```



# Python的类

- 基于词典的内容管理
- 类定义与继承
- 内置函数的应用



# 词典数据:

```
course1 = {  
    'name' : 'Data Science with Python',  
    'instructor' : 'Hu',  
    'capacity' : 200,  
    'Num_students' : 0  
}  
course2 = {  
    'name' : 'Text Mining',  
    'instructor' : 'Junfeng',  
    'capacity' : 20,  
    'Num_students' : 0  
}
```

```
course1['Num_students'] = 120
```

```
course1
```

```
{'name': 'Data Science with Python',  
'instructor': 'Hu',  
'capacity': 200,  
'Num_students': 120}
```

```
studentA = {  
    'name' : 'Wang',  
    'selected course' : []  
}
```

```
studentB = {  
    'name' : 'Li',  
    'selected course' : []  
}
```

```
studentA['selected course']
```

```
[]
```



## 函数与数据分离：

```
def add_course(self, course):
    course['Num_students'] += 1
    self['selected course'].append(course['name'])

add_course(studentA, course1)

add_course(studentB, course2)

print(studentA['selected course'], course1['Num_students'])
print(studentB['selected course'], course1['Num_students'])

['Data Science with Python'] 121
['Text Mining'] 121
```

```
course2['enrolled students'] = []

def enroll_course(self, course):
    course['Num_students'] += 1
    course['enrolled students'].append(self['name'])

    self['selected course'].append(course['name'])

enroll_course(studentB, course2)

print(studentB)

{'name': 'Li', 'selected course': ['Text Mining']}
```



# 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 Course():
    def __init__(self, name, instructor, capacity, Num_students = 0):
        self.name = name
        self.instructor = instructor
        self.capacity = capacity
        self.Num_students = Num_students
```

```
class Student:
    def __init__(self, name, selected_courses = []):
        self.name = name
        self.selected_courses = selected_courses

    def add_course(self, course):
        self.selected_courses.append(course)
        course.Num_students += 1
```

```
c1 = Course('Python & DataScience', 'Hu', 200)
```

```
c1
```

```
: <__main__.Course at 0x117ba8b31d0>
```



```
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'}

对象的内省 ( introspection )



# 类实例、类属性、类方法

- 类也是对象实例，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__开始的为私有属性
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()
```



# @property装饰器定义只读属性

- 由于python进行属性的定义时，没办法设置私有属性，因此要通过来进行设置。这样可以隐藏属性名，让用户进行使用的时候无法随意修改。

```
class DataSet(object):  
    def __init__(self):  
        self._images = 10  
        self._labels = 1 #定义属性的名称  
  
    @property  
    def images(self): #方法加入@property后，这个方法相当于一个属性，这个属性可以让用户进行使用，  
        return self._images  
  
    @property  
    def labels(self):  
        return self._labels  
  
A = DataSet() #直接调用images即可，而不用知道属性名_images，因此用户无法更改属性，从而保护了类的  
print(A.images) #加了@property后，可以用调用属性的形式来调用
```





使用`@<attribute_name>.setter`装饰器可以将一个方法转换为可写属性的setter方法。例如，如果我们要为Person类的name属性定义一个setter方法，我们可以这样做：

```
1 class Person:
2     def __init__(self, name):
3         self._name = name
4
5     @property
6     def name(self):
7         return self._name
8
9     @name.setter
10    def name(self, value):
11        self._name = value
12
13 p = Person("Alice")
14 print(p.name) # 输出 "Alice"
15
16 p.name = "Bob"
17 print(p.name) # 输出 "Bob"
```

[登录复](#)

# 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> ← 可迭代对象
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>

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):
        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)
```

← 返回同一个迭代器实例

```
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



属性的设置：

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>
	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hash对象



# 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： callabel("hello") 返回 False
- 函数及类定义， callabel返回True
- 普通对象实例： callabel返回False
- 实现 \_\_call\_\_()方法的对象实例callabel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 ()**：保证是可调用对象，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



# 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
```

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

```
    @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
```



# 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.

# Ipthon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

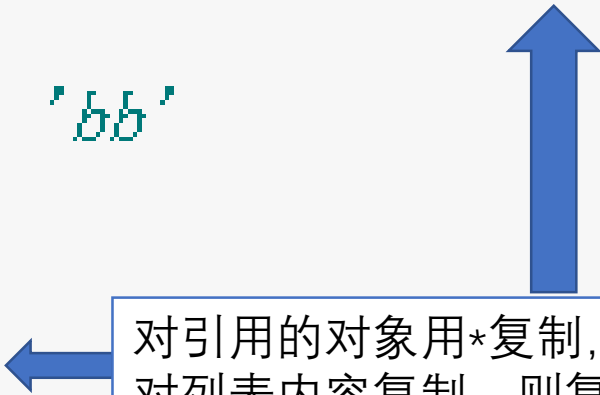


# 使用 `writetfile` 将cell中的内容写入文件:

```
%%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 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.

# ipython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}]
```

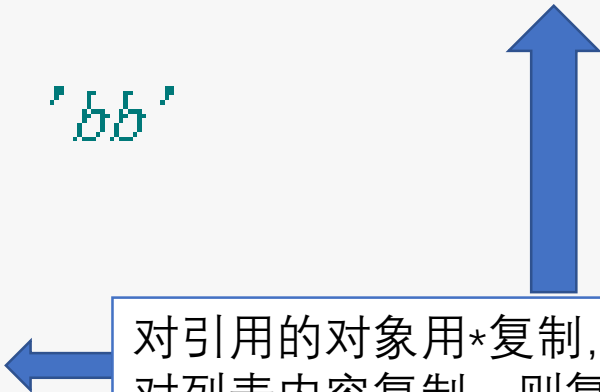


# 使用 `writetfile` 将cell中的内容写入文件:

```
%%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))
```

# 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 *
```

