

# Python基础 C03 - Class

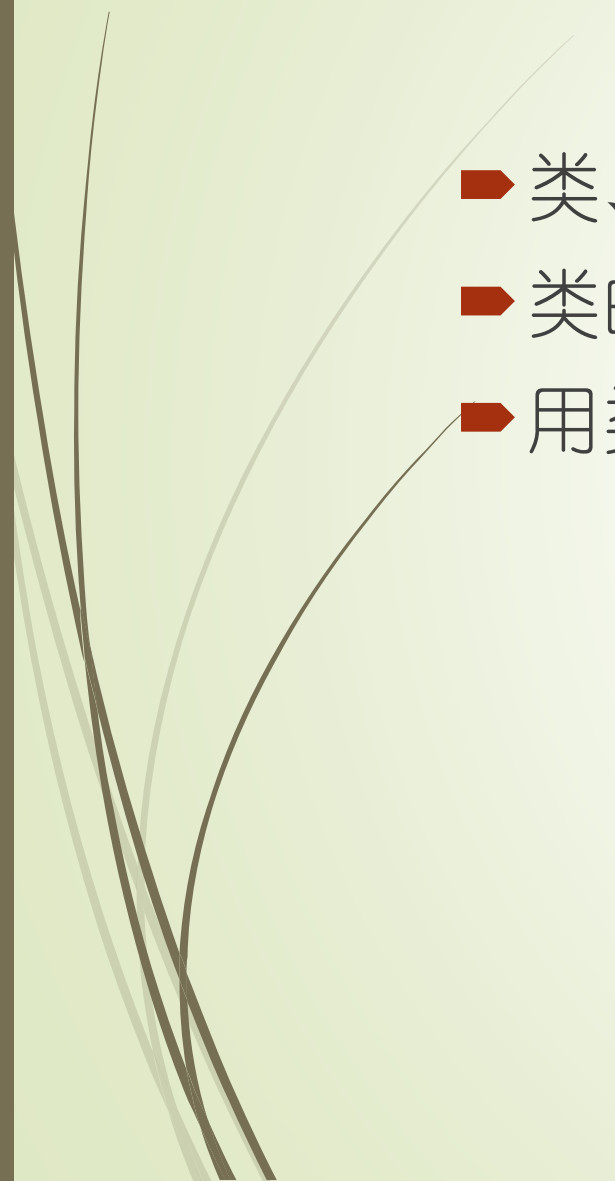


胡俊峰 2020/03/02

北京大学信息科学技术学院



# 主要内容

- 类、对象基础
  - 类的继承与组合
  - 用类实现基本数据结构定义
- 

# 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。但是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。

# Python的类对象

```
class MyClass:
```

```
    # 定义 MyClass 类的属性
```

```
    # 等价于定义 MyClass名字空间下的局部变量，可类比struct实例
```

```
    a = 1
```

```
    b = a + 1
```

```
    # MyClass 下的语句都会被执行
```

```
    for i in range(10):
```

```
        print(i, end = ' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

```
cls1 = MyClass()
```

```
cls2 = MyClass()
```

```
cls2.a = 30      # 生成对象实例属性复本
```

```
print(cls1.a, cls2.a)
```

```
print(MyClass.a) # 类属性不变
```

```
1 30
```

```
1
```

```
class MyClass:
```

```
    a = 1
```

```
    b = a + 1
```

```
    # 定义 MyClass类的实例属性, 实例方法
```

```
    def __init__(self, name= 'John', age= 18):
```

```
        self.name = name
```

```
        self.age = age
```

```
    # 通过第一个参数 self 将实例对象与类对象联系起来
```

```
    def f( self):
```

```
        print('name =', self.name)
```

```
myCls1 = MyClass('Tom')
```

```
myCls1.f()
```

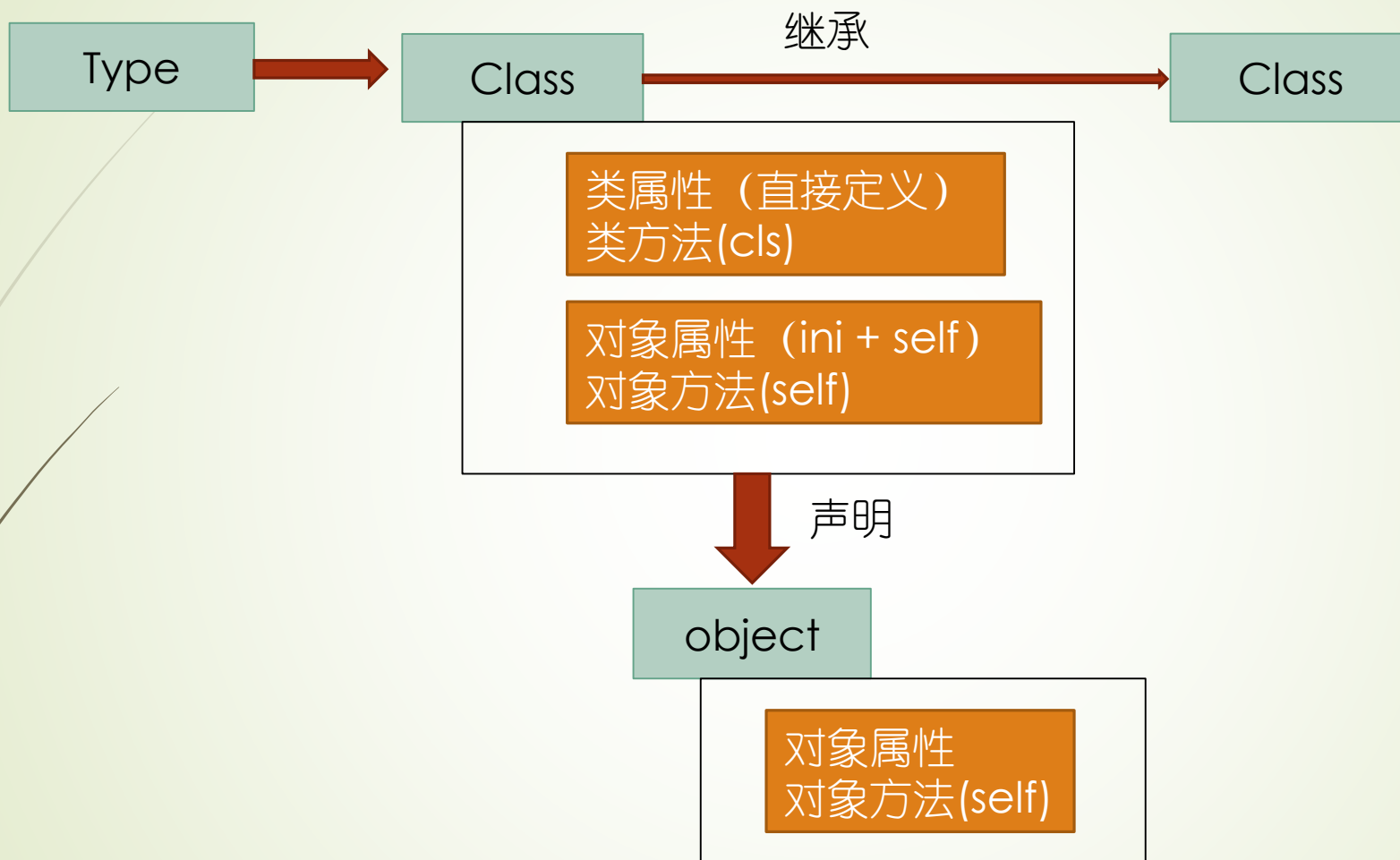
```
myCls2 = MyClass()
```

```
myCls2.f()
```

```
name = Tom
```

```
name = John
```

# Python的类与对象



正确的做法是通过 实例变量 为每个实例对象维护独有的数据

为此, 一般做法是定义一个初始化 `__init__` 方法, 其会在实例化时被自动调用

在 `__init__` 方法中通过 `self` 参数为其实例对象定义变量

```
▶ 1 class MyClassWithInitMethod:
   2     def __init__(self):
   3         self.ls = []
```

设置对象实例的属性

```
▶ 1 my_instance1, my_instance2 = MyClassWithInitMethod(), MyClassWithInitMethod()
   2 print(f'my_instance1.ls = {my_instance1.ls}, my_instance2.ls = {my_instance2.ls}')
   3 my_instance1.ls.append(1)
   4 print(f'my_instance1.ls = {my_instance1.ls}, my_instance2.ls = {my_instance2.ls}')
```

```
my_instance1.ls = [], my_instance2.ls = []
```

```
my_instance1.ls = [1], my_instance2.ls = []
```

也可像命名空间一样为类对象设置新的属性与方法

```
MyClass.d = 4  #添加了一个新的类属性
```

```
def g(self, e):  
    return e + 1
```

← 类似设置一个新的属性（添加一个词典项）

```
MyClass.g = g  #添加了一个新的（实例）方法
```

```
cls3 = MyClass()
```

```
print(f'MyClass.d = {MyClass.d}')  
print(f'MyClass.g(None, 5) = {MyClass.g(None, 5)}')  
print(f'cls3.g(6) = {cls3.g(6)}')
```

```
MyClass.d = 4
```

```
MyClass.g(None, 5) = 6
```

```
cls3.g(6) = 7
```



至此, 我们给出一个一般化的类定义格式

```
1  class MyClass:
2      # 写一点文档
3      """这是一个类 MyClass"""
4
5      # 定义类变量
6      xxx = 1
7
8      # 定义初始化方法: self 即对应的实例对象
9      def __init__( self,
10                     a, b  # 其他参数
11                 ):
12         # 定义实例变量
13         self.a = a
14         self.b = b
15         self.c = a * b
16
17         # 可在方法内通过实例对象 self 调用该类其他方法 和访问类属性
18         self.f(yyy=1, zzz=2)
19
20     # 定义其他方法
21     def f(self, yyy, zzz):
22         pass
```

# 类的 继承

命名 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

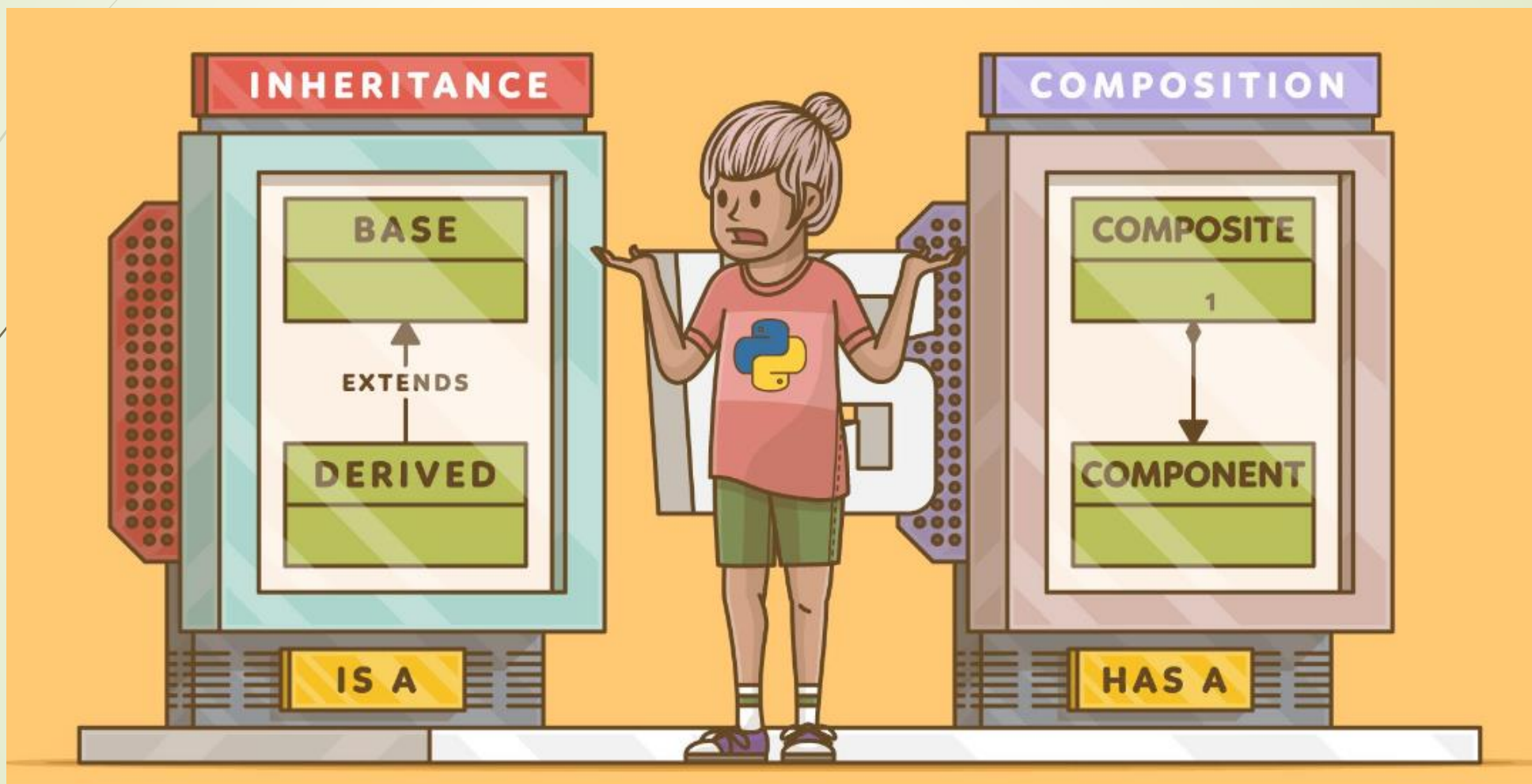
Xiao Wang is speaking chinese

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()
```

# 继承与组合 (Inheritance and Composition)



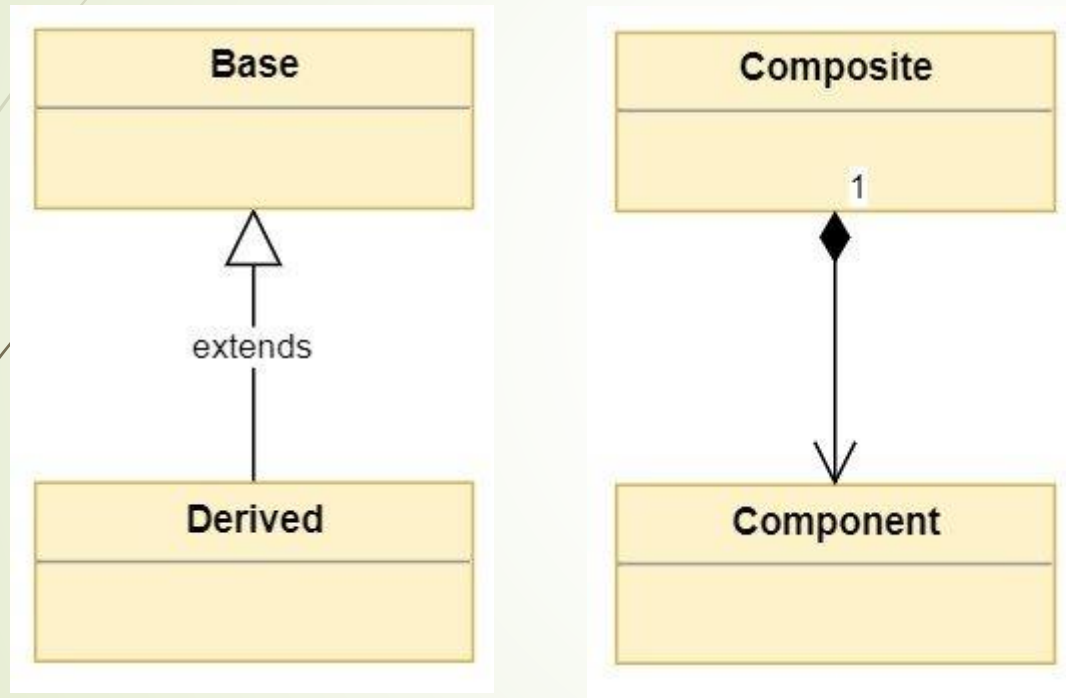


# Inheritance

- Classes that inherit from another are called derived classes, subclasses, or subtypes.
- Classes from which other classes are derived are called base classes or super classes.
- A derived class is said to derive, inherit, or extend a base class.



# 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))
```

# 类的基础方法

序号	目的	所编写代码	Python 实际调用
①	初始化一个实例	<code>x = MyClass()</code>	<code>x.<u>__init__</u>()</code>
②	字符串的“官方”表现形式	<code>repr(x)</code>	<code>x.<u>__repr__</u>()</code>
③	字符串的“非正式”值	<code><u>str</u>(x)</code>	<code>x.<u>__str__</u>()</code>
④	字节数组的“非正式”值	<code>bytes(x)</code>	<code>x.__bytes__()</code>
⑤	格式化字符串的值	<code>format(x, format_spec)</code>	<code>x.__format__(format_spec)</code>

1. 对 `__init__()` 方法的调用发生在实例被创建 之后。如果要控制实际创建进程, 请使用 `__new__()` 方法。
2. 按照约定, `__repr__()` 方法所返回的字符串为合法的 Python 表达式。
3. 在调用 `print(x)` 的同时也调用了 `__str__()` 方法。
4. 由于 `bytes` 类型的引入而从 *Python 3* 开始出现。

## 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>

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迭代器。

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>

age (8)

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>

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>

序号	目的	所编写代码	Python 实际调用
	自定义对象的复制	<code>copy.copy(x)</code>	<code>x.__copy__()</code>
	自定义对象的深度复制	<code>copy.deepcopy(x)</code>	<code>x.__deepcopy__()</code>
	在 pickling 之前获取对象的状态	<code>pickle.dump(x, file)</code>	<code>x.__getstate__()</code>
	序列化某对象	<code>pickle.dump(x, file)</code>	<code>x.__reduce__()</code>
	序列化某对象 (新 pickling 协议)	<code>pickle.dump(x, file, protocol_version)</code>	<code>x.__reduce_ex__(protocol_version)</code>
*	控制 unpickling 过程中对象的创建方式	<code>x = pickle.load(file)</code>	<code>x.__getnewargs__()</code>
*	在 unpickling 之后还原对象的状态	<code>x = pickle.load(file)</code>	<code>x.__setstate__()</code>

\* 要重建序列化对象，Python 需要创建一个和被序列化的对象看起来一样的新对象，然后设置新对象的所有属性。`__getnewargs__()` 方法控制新对象的创建过程，而 `__setstate__()` 方法控制属性值的还原方式。

# 一些常用数据结构实现

## Singly Linked List Implementation

In this lecture we will implement a basic Singly Linked List.

Remember, in a singly linked list, we have an ordered list of items as individual Nodes that have pointers to other Nodes.

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

Now we can build out Linked List with the collection of nodes:

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

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

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

In a Linked List the first node is called the **head** and the last node is called the **tail**. Let's discuss the pros and cons of Linked Lists:



# Doubly Linked List Implementation

In this lecture we will implement a Doubly Linked List

```
1 class DoublyLinkedListNode(object):
2
3     def __init__(self, value):
4
5         self.value = value
6         self.next_node = None
7         self.prev_node = None
```

Now that we have our node that can reference next *and* previous values, let's begin to build our linked list!

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

```
1 # Setting b after a
2 b.prev_node = a
3 a.next_node = b
```

```
1 # Setting c after a
2 b.next_node = c
3 c.prev_node = b
```

Having a Doubly Linked list allows us to go through our Linked List forwards **and** backwards.

# Nodes and References Implementation of a Tree

In this notebook is the code corresponding to the lecture for implementing the representation of a Tree as a class with nodes and references!

```
1 class BinaryTree(object):
2     def __init__(self, rootObj):
3         self.key = rootObj
4         self.leftChild = None
5         self.rightChild = None
6
7     def insertLeft(self, newNode):
8         if self.leftChild == None:
9             self.leftChild = BinaryTree(newNode)
10        else:
11            t = BinaryTree(newNode)
12            t.leftChild = self.leftChild
13            self.leftChild = t
14
15    def insertRight(self, newNode):
16        if self.rightChild == None:
17            self.rightChild = BinaryTree(newNode)
18        else:
19            t = BinaryTree(newNode)
20            t.rightChild = self.rightChild
21            self.rightChild = t
22
23
24    def getRightChild(self):
25        return self.rightChild
26
27    def getLeftChild(self):
28        return self.leftChild
29
30    def setRootVal(self, obj):
31        self.key = obj
32
33    def getRootVal(self):
34        return self.key
```

We can see some examples of creating a tree and assigning children. Note that some outputs are Trees themselves!

```
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

We can also encapsulate the memoization process into a class:

```
: ▶ 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]
```

Then all we would have to do is:

```
: ▶ 1 def factorial(k):
    2
    3     if k < 2:
    4         return 1
    5
    6     return k * factorial(k - 1)
    7
    8 factorial = Memoize(factorial)
```

Try comparing the run times of the memoization versions of functions versus the normal recursive solutions!

如果想要调用基类的同名方法, 可以使用如下语句 `BaseClassName.methodname(self, arguments)`

```
1 Person.talk(c)
```

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

python中有两个检验类的built-in函数 `isinstance(obj, classinfo)` 检验是否是某个类的实例  
`issubclass(obj, classinfo)` check 继承 \

```
1 isinstance(c, Chinese)
```

```
2]: True
```

```
1 isinstance(c, Person)
```

```
3]: True
```

```
1 issubclass(Chinese, Person)
```

```
4]: True
```

还有一些内建的方法

```
type()
```

```
dir()
```

# python pickle模块

- 1 持久性就是指保持对象，甚至在多次执行同一程序之间也保持对象。通过本文，您会对 Python对象的各种持久性机制（从关系数据库到 Python 的 pickle以及其它机制）有一个总体认识。另外，还会让您更深一步地了解Python 的对象序列化能力。

```
1 import pickle
2 t1 = ('this is a string', 42, [1, 2, 3], None)
3 t2 = pickle.dumps(t1, 0)
4 t2
```

```
b'(\x00this is a string\n\x0042\n(\x001\n\x002\n\x003\n\x00N\x00t\x00p\x002\n.'
```

```
1 t3 = pickle.loads(t2)
2 t3
```

('this is a string', 42, [1, 2, 3], None)

```
1 a = [1, 2, 3]
2 b = a
3 a.append(4)
4 c = pickle.dumps((a, b))
5 d, e = pickle.loads(c)
6 d
```

[1, 2, 3, 4]

```
1 d.append(5)
2 e
```

[1, 2, 3, 4, 5]

```
>>> a1 = 'apple'
>>> b1 = {1: 'One', 2: 'Two', 3: 'Three'}
>>> c1 = ['fee', 'fie', 'foe', 'fum']
>>> f1 = file('temp.pkl', 'wb')
>>> pickle.dump(a1, f1, True)
>>> pickle.dump(b1, f1, True)
>>> pickle.dump(c1, f1, True)
>>> f1.close()
>>> f2 = file('temp.pkl', 'rb')
>>> a2 = pickle.load(f2)
>>> a2
'apple'
>>> b2 = pickle.load(f2)
>>> b2
{1: 'One', 2: 'Two', 3: 'Three'}
>>> c2 = pickle.load(f2)
>>> c2
['fee', 'fie', 'foe', 'fum']
```



# 格式化输出

之前我们简单说过在python3.6之后的几种格式化字符串的方式，这里统一做一下总结。

## %格式化输出方法

总的来说，python的格式化字符串有两大类方式，一类是像C一样，用 `%参数` 作为格式化参数，在字符串变量的后面通过 `%` 连接一个tuple，用来替换字符串中的参数，例如

```
1 buff = 'the length of (%s) is %d' % (s, len(s))
2 print(buff)
```

the length of (string) is 6

其他的格式化符号如下：

- `%c` 格式化字符及其ASCII码
- `%s` 格式化字符串
- `%d` 格式化整数
- `%u` 格式化无符号整型
- `%o` 格式化无符号八进制数
- `%x` 格式化无符号十六进制数
- `%X` 格式化无符号十六进制数（大写）

## format格式化方法


另一类是使用 `str.format()` 函数，通过大括号占位，使用 `format()` 中的参数替换相应的位置。

下面前三个例子展示了可以直接使用默认顺序，也可以手动指定变量的位置，甚至通过命名参数通过参数名指定替换的位置。

后两个例子则是 `python3.6` 后支持的f字符串，通过在字符串前面加f来指明这是一个格式化字符串，然后在大括号内直接使用变量名甚至语句来替换内容。

```
1 print('the length of {} is {}'.format(s, len(s)))
2 print('the length of {1} is {0}'.format(len(s), s))
3 print('the length of {var1} is {var2}'.format(var1=s, var2=len(s)))
4 print(f'the length of {s} is {len(s)}')
5 print(f'the length of {s} is {s.__len__()}')
```

```
the length of string is 6
the length of string is 6
the length of string is 6
the length of string is 6
the length of string is 6
```



```
1 for x in range(1, 11):  
2     print(' {0:2d}  {1:4d}  {2:4d}'.format(x, x*x, x*x*x))  
3
```

1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000



# 文件读写

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()
```



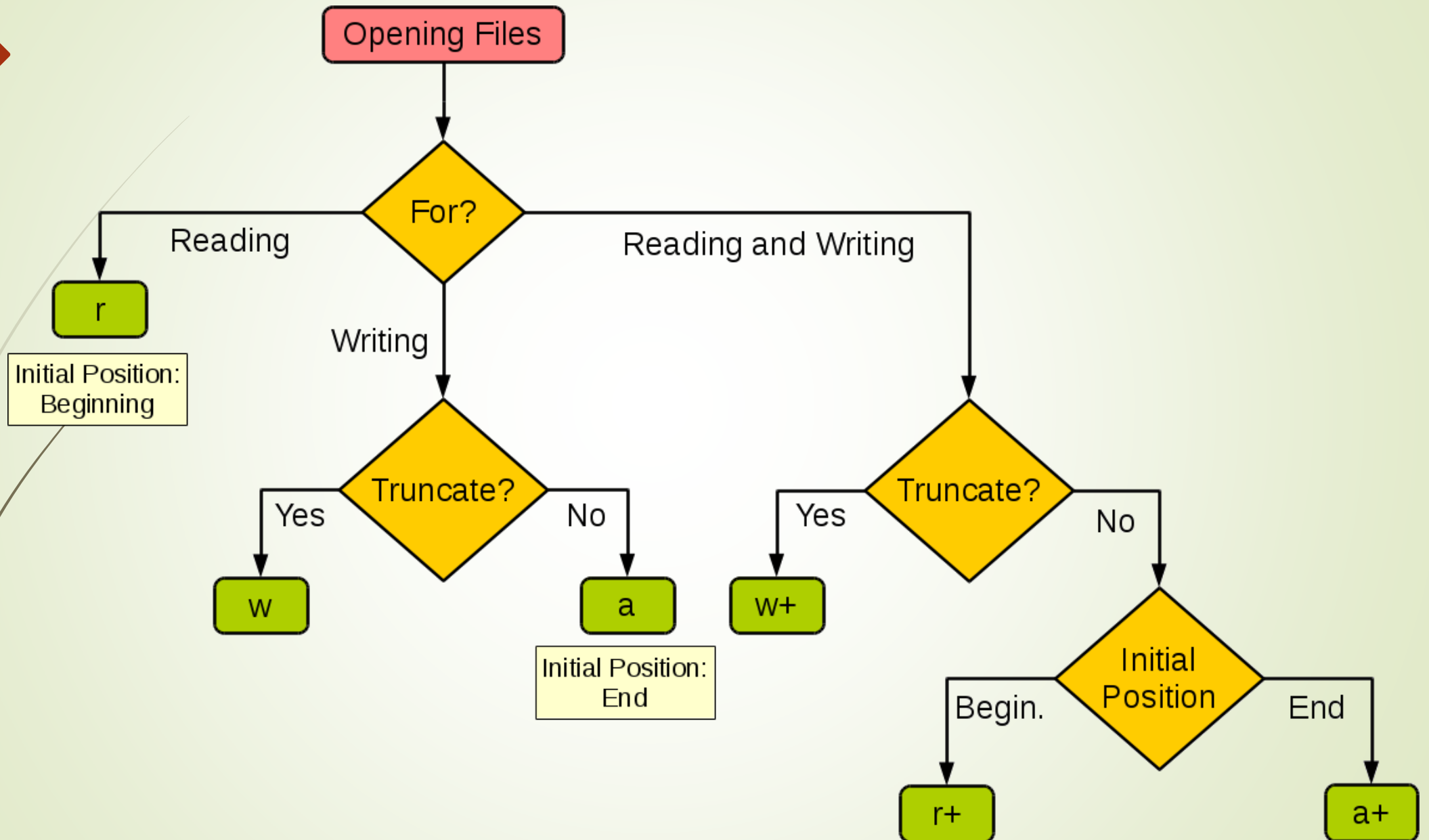
## 字节文件的直接存取

```
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' 2'

6

b' d'








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

```
sds0123456789abcdef
```

```
hello world!
```

```
True
```



作业下午在群里布置

