AdvancePython

发布 *0.0.1*

2019 年 12 月 18 日

Contents

Contents

CHAPTER 1

第一章：导论

Introduce

1.1 1.1 课程介绍

1.1.1 你是否遇到

- 看不懂优秀库和框架的源码
- 不知道如何才能进一步优化自己的代码
- asyncio、tornado 等异步框架背后的原理
- python 代码灵活背后的设计原理
- 对生成器稀里糊涂
- 很多 python 抛的异常看不懂

1.1.2 如何进阶

- 和大神一起
- 阅读优秀源码、懂原理
- 面试、多做项目

1.1.3 课程收获

- 应对 python 高级工程师面试
- 从深度和广度都扩充自己的知识体系
- 能应对各种难度的框架和库的源码
- 深入理解 python 的设计理念和背后的原理

CHAPTER 2

第二章：Python 一切皆对象

Introduce

2.1 2.1 python 中一切皆对象

2.1.1 一切皆对象是 python 灵活性的根本

- python 是动态语言，面向对象更加彻底
- 函数和类也是对象，属于 python 的一等公民

一等公民特性

- 赋值给一个变量
- 可以添加到集合对象中
- 可以作为参数传递给函数
- 可以当做函数的返回值

```
def bar(name):
    print('Hello %s' % name)

class Person(object):
    def __init__(self, name):
```

(下页继续)

(续上页)

```
        print('Hello %s' % name)

def decorator_func():
    print('from decorator_func')
    return bar

def print_type(obj):
    print(type(obj))

# 赋值给一个变量
func = bar
func('Linda')
my_class = Person
my_class('Tom')

# 可以添加到集合对象中
obj_list = []
obj_list.append(bar)
obj_list.append(Person)
for item in obj_list:
    # 可以作为参数传递给函数
    print_type(item)

# 可以当做函数的返回值
deco_bar = decorator_func()
deco_bar('deco_bar')
```

2.1.2 小试牛刀

实际项目中定义 Model 来管理存储的数据，可以在 Model 层之上加一些元操作函数，如 get，update，delete 等等

定义 handle 接收函数预处理数据

```
def get_record_by_id(_id, handle=None):
    record = Model.get(id=_id)
    if not record:
        return False, '无相关记录'
    if handle:
        record = handle(record)
```

(下页继续)

(续上页)

```
    return True, record.to_dict
```

2.2 2.2 type、object 和 class 之间的关系

2.2.1 type 实例化常见类型

```
>>> name = 'linda'
>>> type(num)
<class 'int'>
>>> type(int)
<class 'type'>
>>> type(object)
<class 'type'>
>>> type(type)
<class 'type'>
```

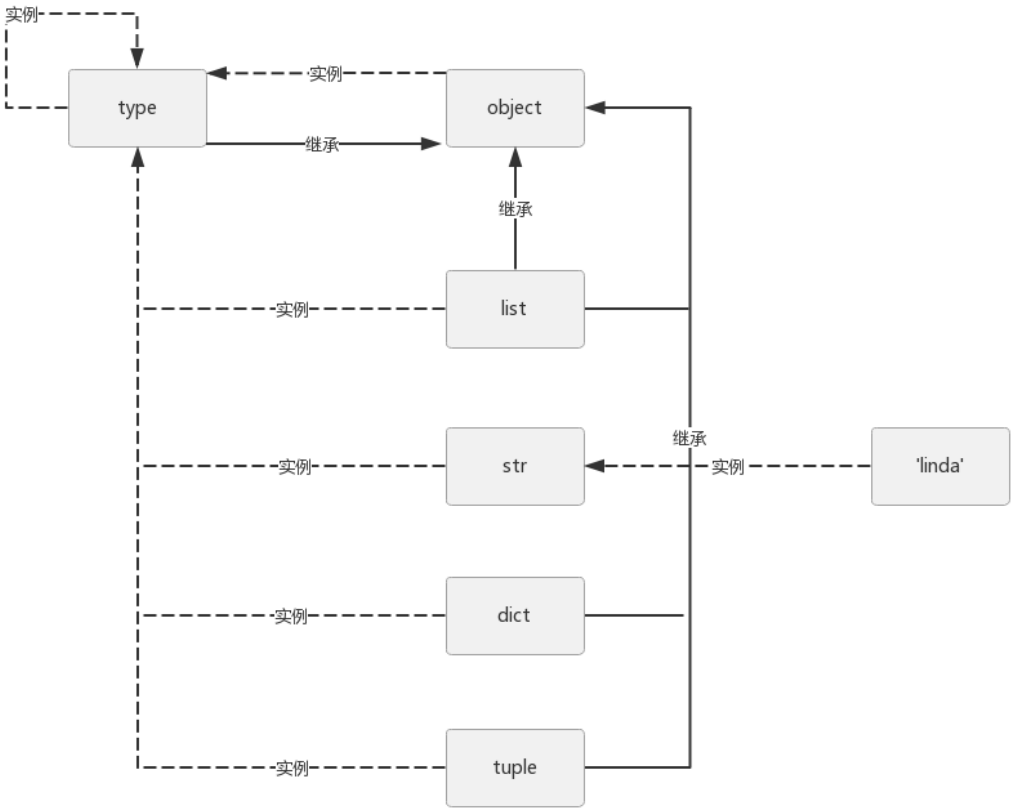
- 类 str 实例化 ‘linda’
- 类 type 实例化 str
- 类 type 实例化 object
- 类 type 实例化 type (具体实现类似 C 语言指针)

2.2.2 object 是最顶层基类

```
>>> int.__bases__
(<class 'object'>, )
>>> type.__bases__
(<class 'object'>, )
>>> object.__bases__
()
```

- 类 str 继承 object
- 类 type 继承 object
- object.___bases___ 值为 ()

2.2.3 type、object、class 关系图



- 把 list，dict 等类做成对象，后续后修改特别方便
- object 是所有类的基类，type 也要继承它
- type 也是自身的实例，一切皆对象，一切继承 object

2.3 2.3 python 中的内置类型

2.3.1 对象的三个特征

- 身份，对象在内存中的地址
- 类型，每个对象都应该有个类型，分类思想
- 值，度量大小

```
>>> name = 'linda'
>>> id(name)
2247760594928
>>> type(name)
<class 'str'>
>>> name
'linda'
```

2.3.2 python 常见数据类型

- None(全局只有一个)，即解释器启动时定义
- 数值：int、float、complex、bool
- 迭代类型
- 序列类型：list、tuple、str、bytes
- 映射类型：dict
- 集合类型：set、frozenset
- 上下文管理类型：with 语句
- 其他：模块类型、class 和实例、函数类型、方法类型、代码类型、object 对象、type 类型、notimplemented 类型

set 和 dict.keys() 实现原理相同，较快

2.3.3 小试牛刀

问：这些常见类型是划分的本质是什么呢，又如何自定义这些类型呢

答: 魔法函数

2.4 2.4 本章小结

- python 一切皆对象，以及如何实现一切皆对象
- python 一等公民的特性
- python 常见数据类型概述

CHAPTER 3

第三章：魔法函数

Introduce

3.1 3.1 什么是魔法函数

python 定义类时中，以双下划线开头，以双下划线结尾函数为魔法函数

- 魔法函数可以定义类的特性
- 魔法函数是解释器提供的功能
- 魔法函数只能使用 python 提供的魔法函数，不能自定义

```
class Company:
    def __init__(self, employee_list):
        self.employee = employee_list

    def __getitem__(self, index):
        return self.employee[index]


company = Company(['alex', 'linda', 'catherine'])
employee = company.employee

for item in employee:
```

(下页继续)

(续上页)

```
    print(item)

# for 首先去找 __iter__, 没有时优化去找__getitem__
for item in company:
    print(item)
```

3.2 3.2 python 数据模型

3.2.1 python 数据模型

数据模型，涉及到知识点其实就是魔法函数

- 魔法函数会影响 python 语法 company[:2]
- 魔法函数会影响内置函数调用 len(company)

3.3 3.3 魔法函数一览

3.3.1 非数据运算

字符串表示

- ___repr___
- ___str___

```
class Company:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return '<Company [%s]>' % self.name

    def __repr__(self):
        return '<Company [%s]>' % self.name


company = Company('Apple')
print(company)
```

(下页继续)

(续上页)

```
# Python 解释器会隐含调用
print(company.__repr__())
```

集合、序列相关

- ___len___
- ___getitem___
- ___setitem___
- ___delitem___
- ___contains___

迭代相关

- ___iter___
- ___next___

可调用

- ___call___

with 上下文管理器

- ___enter___
- ___exit___

3.3.2 数据运算

- ___abs___
- ___add___

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

    def __abs__(self):
        return abs(self.num)


n = Num(-1)
print(abs(n))
```

(下页继续)

(续上页)

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return 'Vector(%s, %s)' % (self.x, self.y)


v1 = Vector(1, 3)
v2 = Vector(2, 4)
print(v1 + v2)
```

3.4 3.4 len 函数的特殊性

CPython 时，向 list，dict 等内部做过优化，len(list) 效率高

3.5 3.5 本章小结

- 什么是魔法函数，如何增加类型的特性
- 不需要显示调用魔法函数，内置函数或相应 python 语法时会自动触发魔法函数
- 通过魔法函数去理解 python 类型
- len 内部有优化

CHAPTER 4

第四章：深入类和对象

Introduce

4.1 4.1 鸭子类型和多态

4.1.1 鸭子类型

当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来像鸭子，那么这只鸟可以被称为鸭子

```
class Cat:
    def say(self):
        print('I am a cat.')

class Dog:
    def say(self):
        print('I am a dog.')

class Duck:
    def say(self):
        print('I am a duck.')


# Python 中较灵活，只要实现 say 方法就行，实现了多态
```

(下页继续)

(续上页)

```
animal = Cat
animal().say()

# 实现多态只要定义了相同方法即可
animal_list = [Cat, Dog, Duck]
for an in animal_list:
    an().say()

"""
class Animal:
    def say(self):
        print('I am a animal.')

# 需要继承 Animal，并重写 say 方法
class Cat(Animal):
   def say(self):
       print('I am a cat.')

# Java 中定义需要指定类型
Animal an = new Cat()
an.say()
"""

li1 = ['i1', 'i2']
li2 = ['i3', 'i4']

tu = ('i5', 'i6')
s1 = set()
s1.add('i7')
s1.add('i8')

# 转变观念，传入的不单单是 list，甚至自己实现 iterable 对象
li1.extend(li2)     # iterable
li1.extend(tu)
li1.extend(s1)
print(li1)
```

- 实现多态只要定义了相同方法即可
- 魔法函数充分利用了鸭子类型的特性，只要把函数塞进类型中即可

4.2 4.2 抽象基类 (abc 模块)

4.2.1 抽象基类

- 抽象基类无法实例化
- 变量没有类型限制，可以指向任何类型
- 抽象基类和魔法函数构成了 python 的基础，即协议

在抽象基类定义了抽象方法，继承了抽象基类的类，必须实现这些方法

场景一：想判断某个对象的类型

```
# 检查某个类是否有某种方法
class Company:
    def __init__(self, name):
        self.name = name

    def __len__(self):
        return len(self.name)


company = Company('Linda Process Ltd.')
print(hasattr(company, '__len__'))

from collections.abc import Sized
print(isinstance(company, Sized))
```

场景二：强制子类必须实现某些方法

```
import abc

class CacheBase(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def get(self, key):
        pass

    @abc.abstractmethod
    def set(self, key, value):
        pass
```

(下页继续)

(续上页)

```
class MemoryCache(CacheBase):
    pass
```

注意：抽象基类容易设计过度，多继承推荐使用 Mixin

4.3 4.3 isinstance 和 type 的区别

- isinstance 会去查找继承链
- type 只判断变量的内存地址

```
class A:
    pass

class B(A):
    pass

b = B()
print(isinstance(b, B))
print(isinstance(b, A))

# is 判断 id 的意思
print(type(b) is B)
print(type(b) is A)     # False
```

4.4 4.4 类变量与实例变量

- 类变量定义与使用
- 实例变量定义与使用
- 类变量是所有实例变量共享

```
class A:
    aa = 1

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

(下页继续)

(续上页)

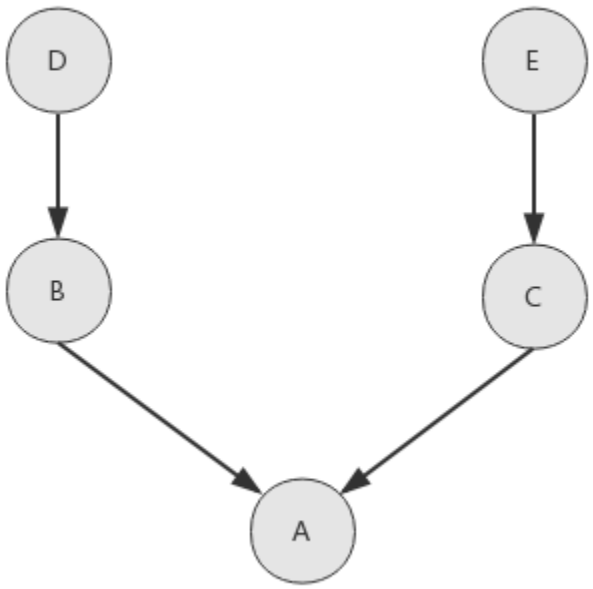
```
a = A(2, 3)
print(a.x, a.y, A.aa)

A.aa = 111
a.aa = 100  # 新建一个 a 的属性 aa, 100 赋值给该 aa
print(A.aa, a.aa)
```

4.5 4.5 类属性和实例属性以及查找顺序

4.5.1 类属性和实例属性

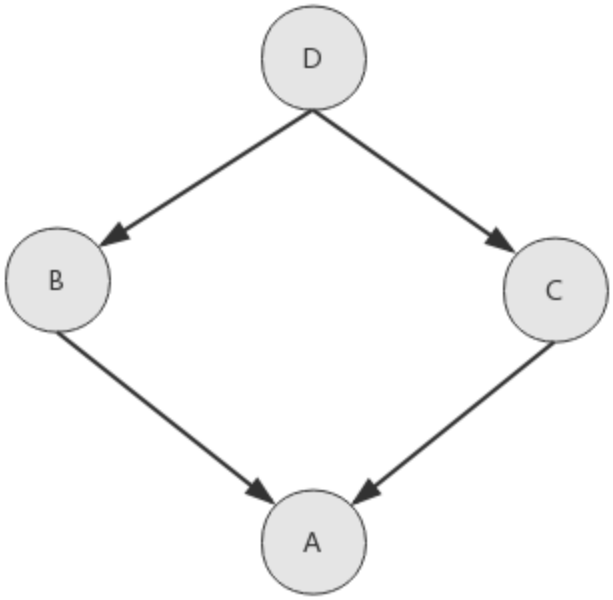
- 类属性：定义在类中的变量和方法
- 实例属性：___init___ 中定义

4.5.2 深度优先 DFS



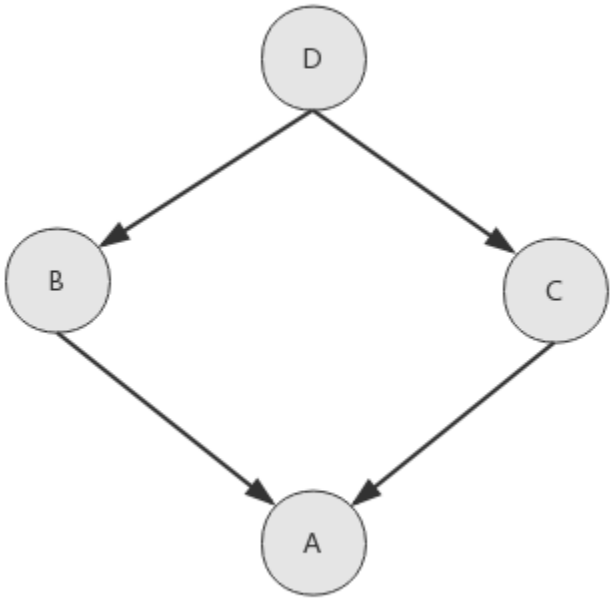
- 查找顺序为 A -> B -> D -> C -> E

- 此种场景深度优先较为合适

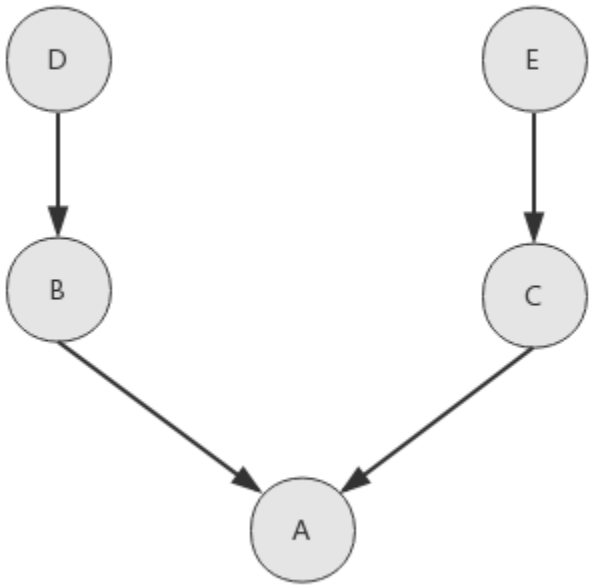


- 查找顺序为 A -> B -> D -> C
- 此种场景当 C 中重载了 D 中某个方法，该查找顺序就不合适

4.5.3 广度优先 BFS



- 查找顺序为 A -> B -> C -> D
- 此种场景深度优先较为合适



- 查找顺序为 A -> B -> C -> D -> E
- 此种场景 B 继承 D，B 和 D 是一体的，D 应该先于 C

4.5.4 MRO C3 算法

菱形功能继承 D 场景

```
class D:
pass

class C(D):
pass

class B(D):
pass

class A(B, C):
pass

print(A.__mro__)
# (<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.D'>
↪, <class 'object'>)
```

(下页继续)

(续上页)

B、C 各自继承 D、E 场景

```
class D:
    pass

class E:
    pass

class C(E):
    pass

class B(D):
    pass

class A(B, C):
    pass

print(A.__mro__)
# (<class '__main__.A'>, <class '__main__.B'>, <class '__main__.D'>, <class '__main__.C'>
↪, <class '__main__.E'>, <class 'object'>)
```

4.6 4.6 静态方法、类方法、对象方法以及参数

- 静态方法 @staticmethod
- 类方法 @classmethod
- 实例方法

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def tomorrow(self):
        self.day += 1

    @staticmethod
```

(下页继续)

(续上页)

```
    def date_from_str(date_str):
        year, month, day = tuple(date_str.split('-'))
        return Date(int(year), int(month), int(day))

    @classmethod
    def date_from_string(cls, date_str):
        year, month, day = tuple(date_str.split('-'))
        return cls(int(year), int(month), int(day))

    def __str__(self):
        return '{year}/{month}/{day}'.format(year=self.year, month=self.month, day=self.
↪day)


if __name__ == '__main__':
    new_day = Date(2020, 2, 20)
    new_day.tomorrow()
    print(new_day)

    date_str = '2020-12-12'
    print(Date.date_from_str(date_str))
    print(Date.date_from_string(date_str))
```

4.7 4.7 数据封装和私有属性

定义类时双下划线的属性，为私有属性

```
class User:
    def __init__(self):
        self.__age = 18

    def get_age(self):
        return self.__age

if __name__ == '__main__':
    user = User()
    print(user.get_age())

    # print(user.__age)
```

(下页继续)

(续上页)

```
    # _class__attr, 做了变形
    print(user._User__age)
```

- python 并不能严格限制私有属性的使用，这是一种写代码规范

4.8 4.8 python 对象的自省机制

通过一定的机制查询对象的内部结构

- ___dict___
- dir()

```
class User:
    name = 'user'

class Student(User):
    def __init__(self):
        self.school_name = 'school'

if __name__ == '__main__':
    stu = Student()

    # 通过__dict__ 查询属性，C 语言实现，经过优化，较快
    print(stu.__dict__)
    stu.__dict__['age'] = 18
    print(stu.age)

    print(User.__dict__)

    print(dir(stu))
```

4.9 4.9 super 真的是调用父类吗

CHAPTER 5

第五章：自定义序列类

Introduce

5.1 5.1 python 中的序列分类

序列是 python 中重要的协议

按照元素类型是否相同

- 容器序列：list、tuple、deque
- 扁平序列：str、bytes、bytearray、array.array

按照元素是否可变

- 可变类型：list、deque、bytearray、array.array
- 不可变：str、tuple、bytes

```
# 元素类型任意
my_list = list()
my_list.append(100)
my_list.append(True)

# 指定元素类型
import array
```

(下页继续)

(续上页)

```
my_array = array.array('i')
my_array.append(100)
# 初始化数组需要整型，附加字符串抛异常
my_array.append('abc')
```

5.2 5.2 序列的 abc 继承关系

python 中内置的 collections.abc 抽象基类，可以帮助我们理解数据类型实现细节

python 是基于协议的语言，结合鸭子类型和魔法函数，就可以达到实现某种类型

from collections.abc import *

- Iterable: ___iter___
- Reversible: ___reversed___
- Sized: ___len___
- Container: ___contains___
- Collection: Sized, Iterable, Container
- Sequence: ___getitem___, Reversible, Collection
- MutableSequence: ___setitem___, ___delitem___, Sequence

不同魔法函数的组合，构建不同的类型

5.3 5.3 序列中 +、+= 和 extend 的区别

- 加号 + 会新生成对象，并且两边类型需要一致
- 加等 += 就地加，只需要可迭代就行
- append 附加单个元素，extend 扩展多个元素

```
# + 加新增
l1 = [1, 2]
l2 = l1 + [3, 4]
print("type(l1): %d, and l1: %s" % (id(l1), l1))
print("type(l2): %d, and l2: %s" % (id(l2), l2))

# += 就地加
l1 += ['3', '4']
```

(下页继续)

(续上页)

```
l1 += ('5', '6')
l1 += range(2)
print("type(l1): %d, and l1: %s" % (id(l1), l1))


# + 两边类型需相同
# += 只需要可迭代的就行，__iadd__ 魔法函数实现
```

5.4 5.4 实现可切片的对象

5.4.1 列表切片操作

```
'''
模式 [start:end:step]

第一个数字 start 表示切片开始位置，默认 0
第二个数字 end 表示切片截止（但不包含）位置，默认列表长度
第三个数字 step 表示切片的步骤，默认为 1

当 start 为 0 时可以省略
当 end 为列表长度时可以省略
当 step 为 1 时可以省略，并且省略步长时可以同时省略最后一个冒号
当 step 为负数时，表示反向切片，这时 start 应该比 end 的值要大才行
'''

a_list = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
a_list[::]      # 返回包含原列表中所有元素的新列表
a_list[::-1]    # 返回包含原列表中所有元素的逆向新列表
a_list[::2]     # 隔一个元素取一个，获取偶数位置的元素
a_list[1::2]    # 隔一个元素取一个，获取奇数位置的元素
a_list[3:6]     # 指定切片的开始和结束位置
a_list[0:100]   # 切片结束位置大于列表长度是，从列表尾部截断
a_list[100:]    # 切片开始位置大于列表长度时，返回空列表

a_list[len(a_list):0] = [9]     # 在列表尾部增加元素
a_list[:0] = [1, 2]             # 在列表头部增加元素
a_list[3:3] = [100]             # 在列表中间位置插入元素
a_list[:2] = [100, 200]         # 替换列表元素，等号两边长度相等
```

(下页继续)

(续上页)

```
a_list[3:] = [4, 5, 6]          # 替换列表元素，等号两边长度可以不相等
a_list[:3] = []                 # 删除列表中前 3 个元素
```

5.4.2 自定义序列类

参考 collections.abc 序列类 Sequence 所需的魔法函数

```
import numbers

class Group:
    def __init__(self, group_name, company_name, staff):
        self.group_name = group_name
        self.company_name = company_name
        self.staffs = staff

    def __reversed__(self):
        self.staffs.reverse()

    def __getitem__(self, item):
        cls = type(self)
        if isinstance(item, slice):
            return cls(group_name=self.group_name, company_name=self.company_name,␣
↪staff=self.staffs[item])
        elif isinstance(item, numbers.Integral):
            return cls(group_name=self.group_name, company_name=self.company_name,␣
↪staff=[self.staffs[item]])

    def __len__(self):
        return len(self.staffs)

    def __iter__(self):
        return iter(self.staffs)

    def __contains__(self, item):
        if item in self.staffs:
            return True
        else:
            return False
```

(下页继续)

(续上页)

```
staffs = ['linda', 'alex', 'catherine', 'nash', 'curry']
group = Group(group_name='group', company_name='company', staff=staffs)
sub_group = group[0]

print('linda' in sub_group)
print(len(sub_group))
```

5.5 5.5 bisect 维护已排序序列

- bisect 维护一个升序的序列
- 内部二分查找实现，效率高

```
import bisect

# 处理已排序 序列 升序
# 内部二分查找算法实现

l1 = list()
bisect.insort(l1, 10)
bisect.insort(l1, 3)
bisect.insort(l1, 2)
bisect.insort(l1, 6)

print(l1)   # [2, 3, 6, 10]
```

5.6 5.6 什么时候我们不该用列表

- 比 list 更好的 python 内置数据结构
- array 数组连续的内存空间，性能高
- deque 双向列表

5.6.1 array 数组

array 与 list 一个重要区别，array 只能存储指定的数据类型数据

```
import array

list
my_array = array.array('i')
my_array.append(100)
my_array.append('abc')
```

- 某些应用场景，除了 list 我们还有其他更好的选择

5.7 5.7 列表推导式、生成器表达式和字典推导式

5.7.1 列表推导式

列表推导式，或列表生成式，通过一行代码生成列表

```
# 提取出 1-20 之间的奇数
odd_list = [i for i in range(21) if i % 2 == 1]
print(odd_list)

# 逻辑复杂的情况
def handle_item(item):
    return item * item

odd_list = [handle_item(i) for i in range(21) if i % 2 == 1]
print(odd_list)
```

- 列表生成式性能高于列表操作
- 逻辑过于复杂时，列表生成式可读性降低

5.7.2 生成器表达式

列表推导 [] -> ()

```
my_gen = (i for i in range(21) if i % 2 == 1)
print(type(my_gen))     # <class 'generator'>
for i in my_gen:
    print(i)
```

5.7.3 字典推导式

```
d1 = {'key1': 'value1', 'key2': 'value2'}
d2 = {v: k for (k, v) in d1.items()}
print(d2)
```

5.7.4 集合推导式

```
set1 = {v for v in d1.values()}
print(set1)
```

5.8 5.8 本章小结

- 序列类型的分类
- 序列的 abc 继承关系
- 序列的 +、+= 和 extend 的区别
- 实现可切片的对象
- bisect 管理可排序序列
- list 的其他选项
- 列表字典推导式

CHAPTER 6

第六章：深入 Python 的 set 和 dict

Introduce

6.1 6.1 dict 的 abc 继承关系

from collections.abc import *

- Sized: ___len___
- Iterable: ___iter___
- Container: ___contains___
- Collection: Sized, Iterable, Container
- Mapping: Collection
- MutableMapping: Mapping

可以看出来 dict 和 list 有一些共有的魔法函数

```
from collections.abc import Mapping

# dict 属于 Mapping 类型
d = dict()
print(isinstance(d, Mapping))
```

6.2 6.2 dict 的常用方法

借助编辑器，如 PyCharm 实时查看源码

```
d = {
    'linda': {'company': 'Yahoo'},
    'catherine': {'company': 'Google'}
}

# 清空
# d.clear()

# 浅拷贝
new_dict = d.copy()
# 深拷贝
import copy
new_dict_deep = copy.deepcopy(d)

new_dict['linda']['company'] = 'Ali'
print(d)
print(new_dict)
print(new_dict_deep)

dd = dict.fromkeys(['key1', 'key2'], 'default')
dd.get('key', None)
dd.keys(), dd.values(), dd.items()

# 获取 key 的 value, 没有时更新为默认值
print(dd.setdefault('key', 'value'))
dd.update({'name': 'linda'})
print(dd)
```

6.3 6.3 dict 的子类

- 不建议直接继承 c 语言实现的内置结构 dict，list

```
class MyDict(dict):
    def __setitem__(self, key, value):
        super().__setitem__(key, value * 2)
```

(下页继续)

(续上页)

```
my_dict = MyDict(one=1)
print(my_dict)  # 1, 某些情况下，不会调用重写的__setitem__
my_dict['one'] = 1
print(my_dict)  # 2, 触发重写的 __setitem__
```

- 可继承 UserDict 实现自定义
- python 语法模拟 c 语言实现细节
- defaultdict 实现 ___missing___

```
from collections import defaultdict

d2 = defaultdict(list)
# 实现了 __missing__ 魔法函数，可参考 UserDict __getitem__ 实现
value = d2['key']
print(value)
print(dict(d2))
```

6.4 6.4 set 和 frozenset

```
# set 不重复、无序
s1 = set('abcc')
print(s1)   # {'a', 'c', 'b'}

# frozenset 不可变可以作为 dict key
f1 = frozenset('abcc')
print(f1)
```

6.5 6.5 dict 和 set 实现原理

6.6 6.6 本章小结

CHAPTER 7

第七章：对象引用、可变性和垃圾回收

Introduce

7.1 7.1 python 的变量是什么

- java 中变量相当于申请一个盒子，盒子有类型大小之说
- python 中变量，类似一个指针，指针的值是固定的，类似便利贴，可以贴到任何对象上

```
# a 贴到 1 上面
a = 1
# a 再贴到 'abc' 上
a = 'abc'
# 注意顺序：先生成对象，然后贴便利贴

la = [1, 2, 3]
lb = la
# is，本质 id(a) 与 id(b) 比较
print(lb is la)        # True，id 相同
print(id(la), id(lb))

la.append(100)
print(lb)  # lb 和 la 指向相同对象，lb 会发生变化
```

7.2 7.2 is 和 == 区别

- is 比较 id()
- == 比较变量值

```
# is, 本质 id(a) 与 id(b) 比较
# = 右边为对象时, 表示生成新对象
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]
print(a is b)   # False, 说明 id 不同
print(id(a), id(b))
print(a == b)   # True, 值相同, 内部 __eq__ 魔法函数

# 内部优化 小整数、小字符串 全局唯一 intern 机制
a1 = 1
a2 = 1
print(a1 is a2)     # True

s1 = 'abc'
s2 = 'abc'
print(s1 is s2)     # True


class People:
    pass


# People 全局唯一
person = People()
print(type(person) is People)   # True
```

7.3 7.3 del 语句和垃圾回收

```
# python 中垃圾回收算法为 引用计数

a = 1
b = a
del a
```

- del 触发 ___del___ 逻辑

- 对象引用计数为 0 时，会被垃圾回收

7.4 7.4 一个经典的参数错误

a, b 都是整型时

```
def add(a, b):
    a += b
    return a

if __name__ == '__main__':
    a, b = 1, 2
    c = add(a, b)
    print('a: %s, b: %s, c: %s' % (a, b, c))
    # 结果为 a: 1, b: 2, c: 3
    # a 未发生变化
```

a, b 都是列表时

```
def add(a, b):
    a += b
    return a

if __name__ == '__main__':
    a, b = [1, 2], [3, 4]
    c = add(a, b)
    print('a: %s, b: %s, c: %s' % (a, b, c))
    # 结果为 a: [1, 2, 3, 4], b: [3, 4], c: [1, 2, 3, 4]
    # a 发生变化
```

a, b 都是元组时

```
def add(a, b):
    a += b
    return a


if __name__ == '__main__':
    a, b = (1, 2), (3, 4)
    c = add(a, b)
    print('a: %s, b: %s, c: %s' % (a, b, c))
```

(下页继续)

(续上页)

```
    # 结果为 a: (1, 2), b: (3, 4), c: (1, 2, 3, 4)
    # a 未发生变化
```

默认类型为可变类型时

```
class Company:
    def __init__(self, name, staff_list=[]):
        self.name = name
        self.staff_list = staff_list

    def add(self, staff):
        self.staff_list.append(staff)

    def remove(self, staff):
        self.staff_list.remove(staff)


if __name__ == '__main__':
    com1 = Company('com1', ['staff1', 'staff11'])
    com1.add('staff111')
    com1.remove('staff11')
    print(com1.staff_list)

    com2 = Company('com2')
    com2.add('staff2')

    com3 = Company('com3')
    com3.add('staff3')

    print(com2.staff_list)  # ['staff2', 'staff3']
    print(com3.staff_list)  # ['staff2', 'staff3']
```

- 默认值为可变类型 []，com2.staff_list 和 com3.staff_list 指向一块内存空间

7.5 7.5 本章小结

- python 变量的本质
- is、== 和 = 区别与联系
- del 和垃圾回收

CHAPTER 8

第八章：元类编程

Introduce

8.1 8.1 property 动态属性

```
from datetime import date

class User:
    def __init__(self, name, birthday):
        self.name = name
        self.birthday = birthday
        self._age = 0   # _ 一种编程规范

    @property
    def age(self):
        return date.today().year - self.birthday.year

    @age.setter
    def age(self, value):
        self._age = value

    def get_age(self):
```

(下页继续)

(续上页)

```
        return self._age


if __name__ == '__main__':
    user = User('linda', date(1987, 11, 14))
    print(user.age)      # @property 用变量的方式去封装逻辑
    user.age = 100       # @age.setter 接收参数
    print(user.get_age())   # self._age 实例内部有存储的变量 _age
```

- 对外展示 user.age; 内部存储 self._age
- 动态属性 property 内部有更多的逻辑操作空间
- user.age = 100 仔细体会内部处理过程

8.2 8.2 ___getattr___、___getattribute___ 魔法函数

8.2.1 ___getattr___

```
class User:
    def __init__(self, name):
        self.name = name

    def __getattr__(self, item):
        return 'Not found attribute %s' % item

if __name__ == '__main__':
    user = User('linda')
    print(user.age) # Not found attribute age
```

- ___getattr___, 在查找不到属性的时候调用
- 类似 else 机制

```
class User:
    def __init__(self, info=None):
        if not info:
            info = {}
        self.info = info

    def __getattr__(self, item):
```

(下页继续)

(续上页)

```
        return self.info[item]


if __name__ == '__main__':
    user = User({'name': 'linda', 'age': 18})
    print(user.name)
    print(user.age)
```

- 神奇的代理操作

8.2.2 ___getattribute___

```
class User:
    def __init__(self, name):
        self.name = name

    def __getattribute__(self, item):
        return 'get_attribute'


if __name__ == '__main__':
    user = User('linda')
    print(user.name)    # get_attribute
    print(user.test)    # get_attribute
    print(user.other)   # get_attribute
```

- 只要调用属性，就会触发 ___getattribute___
- 把持了整个属性调用入口，尽量不要重写这个方法
- 写框架时会涉及到

8.3 8.3 属性描述符和属性查找过程

property 实现在数据获取和设置时增加额外逻辑处理，并对外提供简单接口

在批量属性操作，如验证，则需要每个属性都要写一遍，代码重复

- 数据属性描述符：实现 ___get___ 和 ___set___ 方法
- 非数据属性描述符：实现 ___get___ 方法

```
import numbers

class IntField:
    def __init__(self):
        self._data = None

    def __get__(self, instance, owner):
        print(instance)     # <__main__.User object at 0x000002B88B270288>
        print(owner)        # <class '__main__.User'>
        print(type(instance) is owner)          # True
        print(instance.__class__ is owner)      # True
        return self._data

    def __set__(self, instance, value):
        if not isinstance(value, numbers.Integral):
            raise ValueError('Need int value')
        # 重点来了，如何保存 value 呢，instance or self
        # 如果 instance.attribute 又会触发 __set__ 描述符
        self._data = value

    def __delete__(self, instance):
        pass


class User:
    age = IntField()
    num = IntField()


if __name__ == '__main__':
    user = User()
    user.age = 18
    print(user.__dict__)    # {} "age" 并没有进入到 __dict__

    print(user.age)
```

转变原先简单的属性获取顺序

```
user 某个类实例，user.age 等价于 getattr(user, 'age')

首先调用 __getattribute__
```

(下页继续)

(续上页)

```
    如果定义了 __getattr__ 方法，调用 __getattribute__ 抛出异常 AttributeError 触发__
↪getattr__
    而对于描述符 (__get__) 的调用，则是发生在 __getattribute__ 内部

user = User(), 调用 user.age 顺序如下:
(1) 如果 'age' 是出现在 User 或基类的 __dict__ 中，且 age 是 data descriptor，那么调用其 __
↪get__(instance, owner) 方法，否则
(2) 如果 'age' 出现在 user 的 __dict__ 中，那么直接返回 user.__dict__['age']，否则
(3) 如果 'age' 出现在 User 或基类的 __dict__ 中
    (3.1) 如果 age 是 non-data descriptor，那么调用其 __get__ 方法，否则
    (3.2) 返回 User.__dict__['age']
(4) 如果 User 有 __getattr__ 方法，调用 __getattr__ 方法，否则
(5) 抛出异常 AttributeError
```

- 属性描述符优先级最高

```
class NonDataIntFiled:
    def __get__(self, instance, owner):
        print(instance)
        print(owner)
        return 100

class User:
    age = NonDataIntFiled()

if __name__ == '__main__':
    user = User()
    # user.__dict__['age'] = 18
    # user.age = 18
    # print(user.__dict__)
    print(user.age)
```

8.4 8.4 ___new___ 和 ___init___ 的区别

- 自定义类中 ___new___：用来控制对象的生成过程，返回 self 对象，如果没有返回值，则不会调用 ___init___
- 自定义类中 ___init___：用来完善对象，如初始化
- ___new___ 在 ___init___ 之前调用

```
class User(object):

    # 新式类才有，生成对象 user 之前加逻辑
    def __new__(cls, *args, **kwargs):
        # args = ('linda', )
        # kwargs = {'age': 20}
        # 与自定义 metaclass 中的 __new__ 有区别
        print('from __new__')
        self = super().__new__(cls)
        return self

    def __init__(self, name, age=18):
        self.name = name
        self.age = age
        print('from __init__')


if __name__ == '__main__':
    user = User('linda', age=20)
```

PS: 统一描述

- 元类 -> 类对象
- 类 -> 实例

8.5 8.5 自定义元类

- class 关键字可以字面创建类

```
def create_class(name):

    if name == 'user':
        class User:
            def __str__(self):
                return 'User'
        return User

    elif name == 'company':
        class Company:
            def __str__(self):
```

(下页继续)

(续上页)

```
                return 'Company'
        return Company

MyClass = create_class('user')
obj = MyClass()
print(obj)
print(type(obj))    # <class '__main__.create_class.<locals>.User'>
```

- type 可以动态创建类，动态添加属性和方法

```
def func(self):
    return 'I am from func.'

class Base:
    def answer(self):
        return 'I am from Base.answer.'

# type 动态创建类
User = type('User', (Base, ), {'name': 'user', 'func': func})
user = User()
print(user.name)
print(user.func())
print(user.answer())
print(type(user))
```

元类创建类的类 metaclass(type) -> class -> instance

```
class MetaClass(type):
    # 用来控制 User 的创建过程 与 User 中的 __new__ 有区别
    def __new__(cls, name, bases, attrs, **kw):
        return super().__new__(cls, name, bases, attrs, **kw)


class User(object, metaclass=MetaClass):

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

    def bar(self):
        print('from bar.')
```

python 在实例化的过程 user = User()

(1) 首先寻找 metaclass，来创建 User，否则

(2) 再次寻找基类 BaseUser 的 metaclass，来创建 User，否则

(3) 接着寻找模块 metaclass，来创建 User，否则

(4) 最后默认 type 为 metaclass 来创建 User

8.6 8.6 通过元素实现 ORM

首先明确需求

```
# 简单定义
class User:
    name = CharFiled(db_column="", max_length=32)
    age = IntFiled(db_column="", min_value=0, max_value=100)
    class Meta:
        db_table = 'user'

# ORM
user = User()
user.name = 'linda'
user.age = 18
user.save()
```

迷你版 ORM

```
from collections import OrderedDict


class Field:
    pass


class IntField(Field):
    def __init__(self, db_column, min_value=0, max_value=100):
        self.db_column = db_column
        self.min_value = min_value
        self.max_value = max_value
        self._value = None
```

(下页继续)

(续上页)

```
    def __get__(self, instance, owner):
        return self._value

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('need int value')
        if value < self.min_value or value > self.max_value:
            raise ValueError('need [%s, %s] value' % (self.min_value, self.max_value))
        self._value = value


class CharField(Field):
    def __init__(self, db_column, max_length=32):
        self.db_column = db_column
        self.max_length = max_length
        self._value = None

    def __get__(self, instance, owner):
        return self._value

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('need str value')
        if len(value) > self.max_length:
            raise ValueError('len need lower than %s' % self.max_length)
        self._value = value


# 元类注入一系列属性
class MetaClass(type):
    def __new__(cls, name, bases, attrs, **kw):
        # BaseModel 也会调用 Metaclass, 但没有定义 name, age 等属性, 可特殊判断
        if name == 'BaseModel':
            return super().__new__(cls, name, bases, attrs, **kw)

        fields = {}
        for key, value in attrs.items():
            if isinstance(value, Field):
                fields[key] = value
```

(下页继续)

(续上页)

```
        attrs_meta = attrs.get('Meta', None)
        _meta = {}
        db_table = name.lower()
        if attrs_meta is not None:
            table = getattr(attrs_meta, 'db_table', None)
            if not table:
                db_table = table

        _meta['db_table'] = db_table
        attrs['_meta'] = _meta
        attrs['fields'] = fields
        if attrs.get('Meta'):
            del attrs['Meta']
        return super().__new__(cls, name, bases, attrs, **kw)


class BaseModel(metaclass=MetaClass):
    def __init__(self, **kw):
        for key, value in kw.items():
            setattr(self, key, value)
        super().__init__()

    def save(self):
        fields = OrderedDict(self.fields)
        fields_str = ", ".join([value.db_column for value in fields.values()])
        values_str = ', '.join([str(getattr(self, field)) if not isinstance(value,␣
↪CharField)
                                else "'%s'" % str(getattr(self, field))
                                for field, value in fields.items()])
        sql = "insert into %s (%s) values (%s)" % (self._meta['db_table'], fields_str,␣
↪values_str)
        print(sql)
        # insert into user (name1, age) values ('linda', 20)


# 自定义类时写少量属性，元类帮助我们注入很多通用属性
class User(BaseModel):
    name = CharField('name1', max_length=16)
    age = IntField('age', min_value=0, max_value=100)
```

(下页继续)

(续上页)

```
    class Meta:
        db_table = 'user'


if __name__ == '__main__':
    user = User(name='linda')
    user.age = 20
    user.save()
```

ORM 设计思想

- 数据属性描述符 (___set___, ___get___) 实现验证操作
- 自定义元类 (MetaClass(type)) 实现参数注入
- 自定义 ORM 类 (BaseModel) 获取元类注入的参数进行额外操作
- 自定义元类注入 objects
- 需特别注意调用层级顺序，___new___ 在 ___init___ 之前，所以 ___init___ 中可以使用元类注册测参数

8.7 8.7 本章小结

CHAPTER 9

第九章：迭代器和生成器

Introduce

9.1 9.1 python 中的迭代协议

什么是迭代协议?

- Iterable
- Iterator

迭代器是什么?

迭代器是访问集合内元素的一种方式，一般用来遍历数据。迭代器和以下标访问方式不一样，迭代器是不能返回的，迭代器提供了一种惰性访问数据的方式

```
from collections.abc import Iterable, Iterator
a = [1, 2]
print(isinstance(a, Iterable))
print(isinstance(a, Iterator))
```

9.2 9.2 什么是迭代器和可迭代对象

实现 ___iter___ 时，必须返回 Iterator 对象

```
from collections.abc import Iterator


class MyIterator(Iterator):
    def __init__(self, employee):
        self.employee = employee
        self.index = 0

    def __next__(self):
        # 真正返回迭代值的逻辑
        # 迭代器不支持切片，不会接收索引值，只能一步一步走
        # 遍历大文件
        try:
            word = self.employee[self.index]
        except IndexError:
            raise StopIteration
        self.index += 1
        return word


class Company:
    def __init__(self, employee):
        self.employee = employee

    # def __iter__(self):
    #     return 1        # TypeError: iter() returned non-iterator of type 'int'

    # def __iter__(self):
    #     return self     # TypeError: iter() returned non-iterator of type 'Company'

    # 使用内置方法 iter
    # def __iter__(self):
    #     return iter(self.employee)  # <iterator object at 0x000001F512B907C8>

    # 使用自定义 MyIterator ******
    def __iter__(self):
        return MyIterator(self.employee)    # <__main__.MyIterator object at␣
↪0x0000013462EF0848>

    def __getitem__(self, index):
```

(下页继续)

(续上页)

```
        return self.employee[index]


if __name__ == '__main__':
    company = Company(['linda', 'alex', 'catherine'])
    my_iterator = iter(company)
    print(my_iterator)
    # for 循环首先查找 __iter__; 如果没有自动生成一个__iter__, 里面遍历__getitem__
    # for item in company:
    #     print(item)

    while True:
        try:
            print(next(my_iterator))
        except StopIteration:
            break

    """
    迭代器设计模式, 不要在 Company 中实现 __next__ 方法, 而要单独实现 MyIterator 实现,
Company 中__iter__ 调用 MyIterator 就行
    """
```

- Comanpy 实例化的对象为可迭代对象，可用 for 循环遍历数据，内部实现了 ___iter___ 方法，该方法返回迭代器
- MyIterator 实现化对象为迭代器，可用 next() 获取数值，内部实现了 ___iter___ 和 ___next___ 方法

9.3 9.3 生成器函数的使用

生成器函数，函数里包含 yield 关键字

- yield
- 不再是普通的函数

```
def gen_func():
    yield 1
    yield 2
    yield 3

# 惰性求值, 延迟求值提供了可能性
```

(下页继续)

(续上页)

```
# 斐波拉契函数 0 1 1 2 3 5 8 ...
def fib(index):
    if index <= 2:
        return 1
    else:
        return fib(index-1) + fib(index-2)

def func():
    return 1

if __name__ == '__main__':
    # 返回为生成器对象，python 编译字节码的时候产生
    gen = gen_func()

    # 生成器对象也是实现了迭代协议的，可以 for 循环
    for value in gen:
        print(value)

    ret = func()
```

- 执行生成器函数得到生成器对象，可 for 循环取值
- 生成器函数可以多次返回值，流程的变化

```
# 获取对应位置的值
def fib(index):
    if index <= 2:
        return 1
    else:
        return fib(index-1) + fib(index-2)


# 获取整个过程
def fib2(index):
    ret_list = []
    n, a, b = 0, 0, 1
    while n < index:
        ret_list.append(b)
        a, b = b, a + b
        n += 1
    return ret_list
```

(下页继续)

(续上页)

```
# yield
def gen_fib(index):
    n, a, b = 0, 0, 1
    while n < index:
        yield b
        a, b = b, a + b
        n += 1


print(fib(10))
print(fib2(10))
for value in gen_fib(10):
    print(value)
```

斐波拉契 1 1 2 3 5 8 ···

- 根据位置获取对应值
- 根据位置获取所有值

9.4 9.4 python 是如何实现生成器的

- 什么场景下运用生成器
- 生成器内部实现原理
- 生成器函数与普通函数区别

9.4.1 python 中函数工作原理

python.exe 会用一个叫做 PyEval_EvalFrameEx(c 函数) 去执行 foo 函数首先会创建一个栈帧 (stack_frame)，一个上下文

```
import inspect
frame = None

def foo():
bar()
```

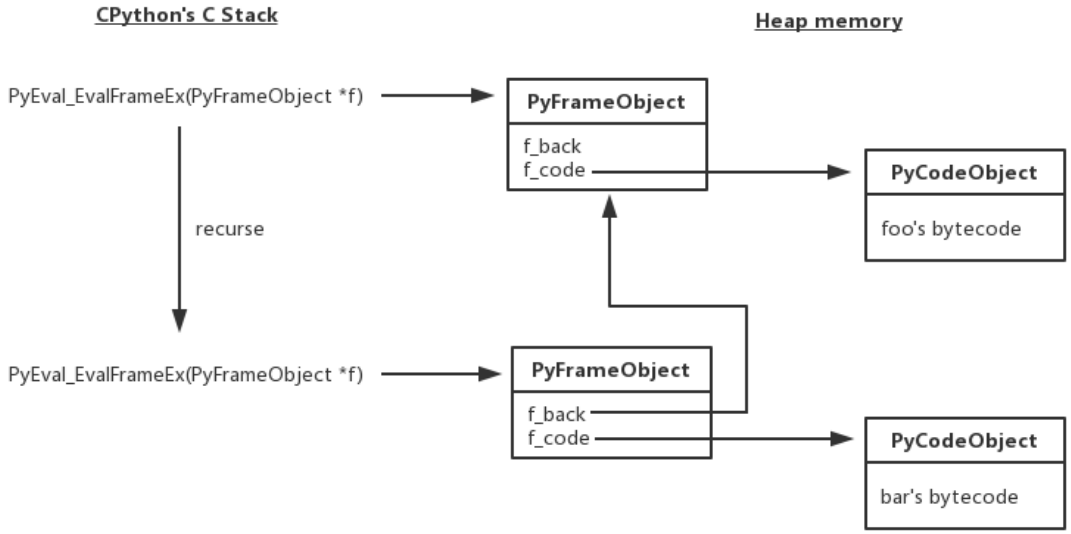
(下页继续)

(续上页)

```
def bar():
global frame
frame = inspect.currentframe()

# 查看函数实现原理
# import dis
# print(dis.dis(foo))

foo()
print(frame.f_code.co_name)          # 函数名 bar

caller_frame = frame.f_back
print(caller_frame.f_code.co_name)   # 函数名 foo
```

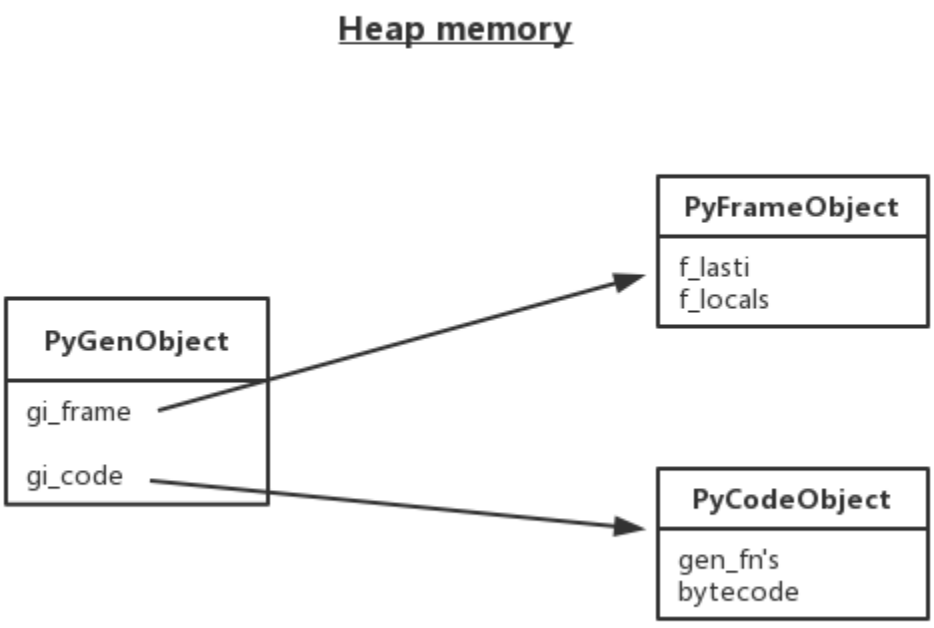
python 中一切皆对象，栈帧对象中运行 foo 函数字节码对象当 foo 调用子函数 bar，又会创建一个栈帧对象，在此栈帧对象中运行 bar 函数字节码对象

所有的栈帧都是分配再堆内存上 (不会自动释放)，这就对定了栈帧可以独立于调用者存在; 不用于静态语言的调用，静态语言是栈的形式，调用完就自动释放



9.4.2 python 中生成器函数工作原理

```
def gen_func():
    address = 'China'
    yield 1
    name = 'linda'
    yield 2
    age = 20
    return 'done'


gen = gen_func()

import dis
print(dis.dis(gen))

print(gen.gi_frame.f_lasti)
print(gen.gi_frame.f_locals)

print('\nfirst value: %s' % next(gen))
print(gen.gi_frame.f_lasti)
print(gen.gi_frame.f_locals)

print('\nsecond value: %s' % next(gen))
print(gen.gi_frame.f_lasti)
print(gen.gi_frame.f_locals)
```



- 控制整个生成器函数暂定和继续前进 gen.gi_frame.f_lasti
- 整个生成器函数作用域逐渐变化 gen.gi_frame.f_locals

9.5 9.5 生成器在 UserList 中的应用

- from collections import UserList

```
class MyList:
    def __init__(self):
        self.data = []

    def __getitem__(self, index):
        return self.data[index]

    def __setitem__(self, index, value):
        self.data[index] = value

    def insert(self, index, item):
        self.data.insert(index, item)
```

(下页继续)

(续上页)

```
ll = MyList()
ll.insert(0, 1)
ll.insert(0, 2)
ll.insert(0, 3)
print(ll.data)
```

- from collections import UserDict

```
class MyDict:
    def __init__(self):
        self.data = {}

    def __getitem__(self, item):
        return self.data[item]

    def __setitem__(self, key, value):
        self.data[key] = value

    def update(self, **kw):
        for key, value in kw.items():
            self[key] = value

dd = MyDict()
print(dd.data)

dd.update(key1='value1', key2='value2')
print(dd['key1'])
print(dd.data)
```

具体源码分析参考模块 collections

9.6 9.6 生成器如何读取大文件

场景：500G 文件特殊只有一行，特殊分割符号 {|}

```
def my_readline(f, newline):
    buf = ''
    while True:
```

(下页继续)

(续上页)

```
        while newline in buf:
            pos = buf.index(newline)
            yield buf[:pos]
            buf = buf[pos + len(newline):]
        chunk = f.read(4096 * 10)
        if not chunk:
            yield buf
            break
        buf += chunk

with open('input') as f:
    for line in my_readline(f, '{|}'):
        print(line)
```

9.7 9.7 本章小结

CHAPTER 10

关于作者

关于作者

- 姓名：ni-ning
- Email：nining1314@gmail.com
- 博客：https://ni-ning.cn/
- GitHub：https://github.com/ni-ning

CHAPTER 11

Roadmap

2019/12/02 - 2019/12/31:

| `github` 项目搭建, `readthedocs` 文档生成。
| 整个项目的框架完成