Python进阶

1. PEP8 编码规范, 及开发中的一些惯例和建议

- 为什么要有编码规范
  - 编码是给人看的还是给机器看的?
  - 美观是重点吗?
    1. 美观
    2. 可读性
    3. 可维护性
    4. 健壮性
  - 团队内最好的代码状态
- 练习: 规范化这段代码

```
from django.conf import settings
import sys, os
mod=0xffffffff
def foo  (a , b =123 ):
    c= { 'x' : 111 , 'y' : 222}
    d =[1, 3 ,5 ]
    return a ,b, c
def bar(x):
    if x%2== 0: return True
```

- 代码编排:
  - 缩进 4 个空格, 禁止空格与 Tab 混用
  - 行长 80 字符: 防止单行逻辑过于复杂
- import
  - 不要使用 `from xxx import *`
  - 顺序
    1. 标准库
    2. 第三方库
    3. 自定义库
  - 单行不要 import 多个库
  - 模块内用不到的不要去 import
- 空格
  - `: , ;` 后面跟一个空格, 前面无空格 (行尾分号后无空格)
  - 二元操作符前后各一个空格, 包括以下几类:
    1. 数学运算符: `+ - * / // = & |`

    2. 比较运算符: `== != > < >= <= is not in`
    3. 逻辑运算符: `and or not`
    4. 位运算符: `& | ^ << >>`
  - 当 = 用于指示关键字参数或默认参数值时, 不要在其两侧使用空格
- 适当添加空行
  - 函数间: 顶级函数间空 2 行, 类的方法之间空 1 行
  - 函数内: 同一函数内的逻辑块之间, 空 1 行
  - 文件结尾: 留一个空行 (Unix 中 \n 是文件的结束符)
- 注释
  - **忌**: 逐行添加注释, 没有一个注释
  - 行内注释: 单行逻辑过于复杂时添加
  - 块注释: 一段逻辑开始时添加
  - 引入外来算法或者配置时须在注释中添加源连接, 标明出处
  - 函数和类尽可能添加 `docstring`
- 命名
  - 除非在 lambda 函数中, 否则不要用 **单字母** 的变量名 (即使是 lambda 函数中的变量名也应该尽可能的有意义)
  - 包名、模块名、函数名、方法名全部使用小写, 单词间用下划线连接
  - 类名、异常名使用 CapWords (首字母大写) 的方式, 异常名结尾加 `Error` 或 `Wraning` 后缀
  - 全局变量尽量使用大写, 一组同类型的全局变量要加上统一前缀, 单词用下划线连接
- 语意明确、直白
  - `not xx in yy` *VS* `xx not in yy`
  - `not a is b` *VS* `a is not b`
- 程序的构建
  - 函数是模块化思想的体现
  - 独立的逻辑应该抽离成独立函数，让代码结构更清晰，可复用度更高
  - **一个函数只做一件事情, 并把这件事做好**
  - **大的功能用小函数之间灵活组合来完成**
  - 避免编写庞大的程序, **“大” 意味着体积庞大, 逻辑复杂甚至混乱**
- 函数名必须有动词, 最好是 do_something 的句式, 或者 somebody_do_something 句式
- 自定义的变量名、函数名不要与标准库中的名字冲突
- pip install pycodestyle pylint flake8

2. * 和 ** 的用法

- 函数定义

```
def foo(*args, **kwargs):
    pass
```

- 参数传递

```
def foo(x, y, z, a, b):
    print(x)
    print(y)
    print(z)
    print(a)
    print(b)
lst = [1, 2, 3]
dic = {'a': 22, 'b': 77}
foo(*lst, **dic)
```

- import * 语法
  - 文件 xyz.py

```
__all__ = ('a', 'e', '_d')

a = 123
_b = 456
c = 'asdfghjkl'
_d = [1,2,3,4,5,6]
e = (9,8,7,6,5,4)
```

  - 文件 abc.py

```
from xyz import *
print(a)
print(_b)
print(c)
print(_d)
print(e)
```

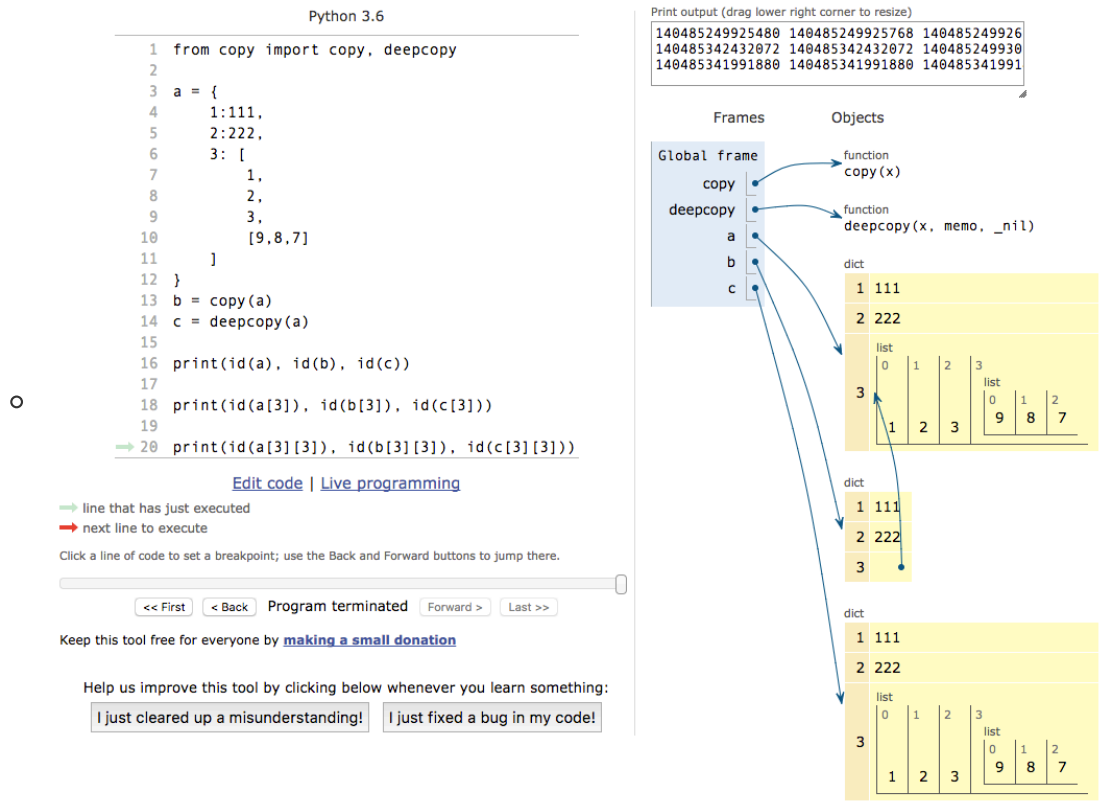
- 强制命名参数

```
def foo(a, *, b, c=123):
    pass
```

- 解包语法: `a, b, *ignored, c = [1, 2, 3, 4, 5, 6, 7]`

3. Python 的赋值和引用

- `==, is`: == 判断的是值, is 判断的是内存地址 (即对象的id)
- 小整数对象: [-5, 256]
- `copy, deepcopy` 的区别
  - `copy`: 只拷贝表层元素
  - `deepcopy`: 在内存中重新创建所有子元素



- 练习1: 说出执行结果

```
def extendList(val, lst=[]):
    lst.append(val)
    return lst

list1 = extendList(10)
list2 = extendList(123, [])
list3 = extendList('a')
```

- 练习2: 说出下面执行结果

```
from copy import copy, deepcopy
from pickle import dumps, loads

a = ['x', 'y', 'z']
b = [a] * 3
c = copy(b)
d = deepcopy(b)
e = loads(dumps(b, 4))

b[1].append(999)
b.append(777)
c[1].append(999)
c.append(555)
d[1].append(999)
d.append(333)
e[1].append(999)
e.append(111)
```

- 自定义 deepcopy: `my_deepcopy = lambda item: loads(dumps(item, 4))`

4. 迭代器, 生成器

```
class Range:
    def __init__(self, start, end=None):
        if end is None:
            self.start, self.end = 0, start
        else:
            self.start = start
            self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        current = self.start
        self.start += 1
        if current < self.end:
            return current
        else:
            raise StopIteration()
```

- iterator: 任何实现了 `__iter__` 和 `__next__` (python2中是 `next()`) 方法的对象都是迭代器.
    - `__iter__`返回迭代器自身
    - `__next__` 返回容器中的下一个值
    - 如果容器中没有更多元素, 则抛出 StopIteration 异常
- generator: 生成器是一种特殊的迭代器, 不需要自定义 `__iter__` 和 `__next__`
    - 生成器函数 (yield)
    - 生成器表达式
- 练习1: 定义一个随机数迭代器, 随机范围为 [1, 50], 最大迭代次数 30

```
import random

class RandomIter:
    def __init__(self, start, end, times):
        self.start = start
        self.end = end
        self.max_times = times
        self.count = 0

    def __iter__(self):
        return self
```

```
    def __next__(self):
        self.count += 1
        if self.count <= self.max_times:
            return random.randint(self.start, self.end)
        else:
            raise StopIteration()
```

- 练习2: 自定义一个迭代器, 实现斐波那契数列

```
class Fib:
    def __init__(self, max_value):
        self.prev = 0
        self.curr = 1
        self.max_value = max_value

    def __iter__(self):
        return self

    def __next__(self):
        if self.curr < self.max_value:
            res = self.curr
            self.prev, self.curr = self.curr, self.prev +
self.curr
            return res
        else:
            raise StopIteration()
```

- 练习3: 自定义一个生成器函数, 实现斐波那契数列

```
def fib(max_value):
    prev = 0
    curr = 1
    while curr < max_value:
        yield curr
        prev, curr = curr, curr + prev
```

- 迭代器、生成器有什么好处?
  - 节省内存
  - 惰性求值
- itertools
  - 无限迭代
    - count(start=0, step=1)
    - cycle(iterable)
    - repeat(object [,times])
  - 有限迭代
    - chain(*iterables)

    - 排列组合
        - `product(*iterables, repeat=1)` 笛卡尔积
        - `permutations(iterable[, r-length])` 全排列
        - `combinations(iterable, r-length)` 组合
- 各种推导式
    - 分三部分：生成值的表达式，循环体，过滤条件表达式
    - 列表：`[i * 3 for i in range(5) if i % 2 == 0]`
    - 字典：`{i: i + 3 for i in range(5)}`
    - 集合：`{i for i in range(5)}`

5. 装饰器

- 最简装饰器

```
def deco(func):
    def wrap(*args, **kwargs):
        return func(*args, **kwargs)
    return wrap

@deco
def foo(a, b):
    return a ** b
```

- 原理
    - 对比被装饰前后的 `foo.__name__` 和 `foo.__doc__`

```
from functools import wraps

def deco(func):
    '''i am deco'''
    @wraps(func)  # 还原被装饰器修改的原函数属性
    def wrap(*args, **kwargs):
        '''i am wrap'''
        return func(*args, **kwargs)
    return wrap
```

    - 简单过程

```
fn = deco(func)
foo = fn
foo(*args, **kwargs)
```

    - 多个装饰器调用过程

```
@deco1
@deco2
@deco3
```

```
def foo(x, y):
    return x ** y

# 过程拆解 1
fn3 = deco3(foo)
fn2 = deco2(fn3)
fn1 = deco1(fn2)
foo = fn1
foo(3, 4)

# 过程拆解 2
# 单行: deco1( deco2( deco3(foo) ) )(3, 2)
deco1(
    deco2(
        deco3(foo)
    )
)(3, 4)
```

- 带参数的装饰器

```
def deco(n):
    def wrap1(func):
        def wrap2(*args, **kwargs):
            return func(*args, **kwargs)
        return wrap2
    return wrap1

# 调用过程
wrap1 = deco(n)
wrap2 = wrap1(foo)
foo = wrap2
foo()

# 单行形式
check_result(30)(foo)(4, 8)
```

- 装饰器类和 `__call__`

```
class Deco:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        return self.func(*args, **kwargs)

@Deco
def foo(x, y):
    return x ** y
```

```
# 过程拆解
fn = Deco(foo)
foo = fn
foo(12, 34)
```

- 使用场景
  - 参数、结果检查
  - 缓存、计数
  - 日志、统计
  - 权限管理
  - 重试
  - 其他
- 练习1: 写一个 timer 装饰器, 计算出被装饰函数调用一次花多长时间, 并把时间打印出来

```
import time
from functools import wraps

def timer(func):
    @wraps(func)  # 修正 docstring
    def wrap(*args, **kwargs):
        time0 = time.time()
        result = func(*args, **kwargs)
        time1 = time.time()
        print(time1 - time0)
        return result
    return wrap
```

- 练习2: 写一个 Retry 装饰器

```
import time

class retry(object):
    def __init__(self, max_retries=3, wait=0, exceptions=
(Exception,)):
        self.max_retries = max_retries
        self.exceptions = exceptions
        self.wait = wait

    def __call__(self, func):
        def wrapper(*args, **kwargs):
            for i in range(self.max_retries + 1):
                try:
                    result = func(*args, **kwargs)
                except self.exceptions:
                    time.sleep(self.wait)
                    continue
                else:
```

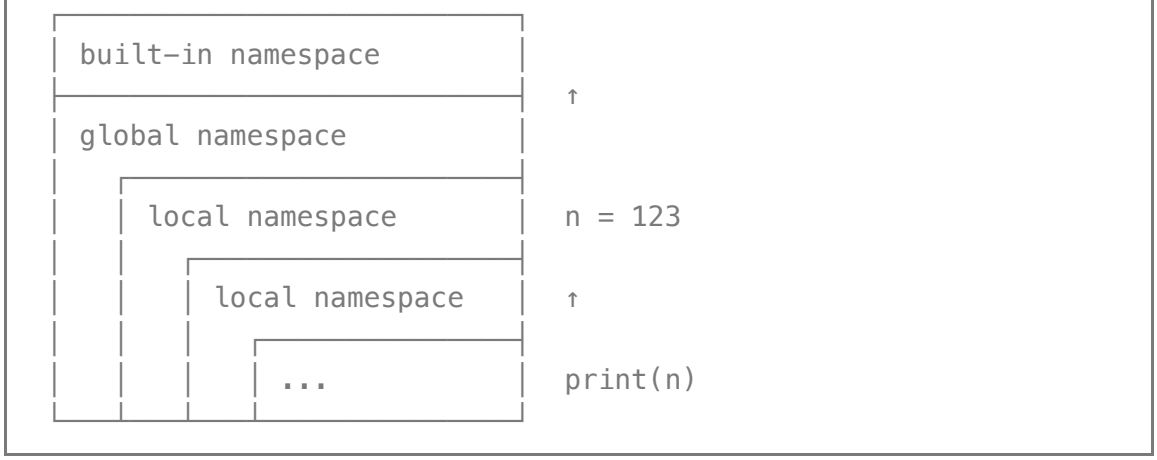
```
                    return result
        return wrapper
```

6. 函数闭包

- Function Closure: 引用了自由变量的函数即是一个闭包. 这个被引用的自由变量和这个函数一同存在, 即使已经离开了创造它的环境也不例外.
- 说出下面函数返回值

```python
def foo():
    l = []
    def bar(i):
        l.append(i)
        return l
    return bar

f1 = foo()
f2 = foo()

# 说出下列语句执行结果
f1(1)
f1(2)
f2(3)
```

- 深入一点: `object.__closure__`
- 作用域



  - 声明全局变量: `global`
  - 声明非本层的 **局部变量** : `nonlocal`
  - 查看全局变量: `globals()`
  - 查看局部变量: `locals()`
  - 查看变量: `vars([object]) # 不传参数相当于 locals(), 传入对象后, 会得到 object.__dict__`

7. 类方法和静态方法

- method: 通过实例调用时, 可以引用类内部的任何属性和方法
- classmethod: 无需实例化, 可以调用类属性和类方法, 无法取到普通的成员属性和方法
- staticmethod: 无论用类调用还是用实例调用, 都无法取到类内部的属性和方法, 完全独立的一个方法
- 练习: 说出下面代码的运行结果

```
class Test(object):
    x = 123

    def __init__(self):
        self.y = 456

    def bar1(self):
        print('i am a method')

    @classmethod
    def bar2(cls):
        print('i am a classmethod')

    @staticmethod
    def bar3():
        print('i am a staticmethod')

    def foo1(self):
        print(self.x)
        print(self.y)
        self.bar1()
        self.bar2()
        self.bar3()

    @classmethod
    def foo2(cls):
        print(cls.x)
        print(cls.y)
        cls.bar1()
        cls.bar2()
        cls.bar3()

    @staticmethod
    def foo3(obj):
        print(obj.x)
        print(obj.y)
        obj.bar1()
        obj.bar2()
        obj.bar3()
```

```
t = Test()
t.foo1()
t.foo2()
t.foo3()
```

8. 继承相关问题

- 继承
- 多态

```
class Animal:
    pass

class Cat(Animal):
    pass

class HelloKitty(Cat):
    pass

kitty = HelloKitty()
isinstance(kitty, Cat)
isinstance(kitty, Animal)
```

- 多继承
    - 方法和属性的继承顺序: `Cls.mro()`
    - 菱形继承问题

```
# 继承关系示意
#
#       A.foo()
#     /   \
#   B      C.foo()
#     \   /
#       D

class A:
    def foo(self):
        print('I am A')

class B(A):
    pass

class C(A):
    def foo(self):
        print('I am C')

class D(B, C):
```

```
    pass

d = D()
d.foo()
print(D.mro())
```

- Mixin: 通过单纯的 mixin 类完成功能组合
- super

```
class A:
    def __init__(self):
        print('enter A')
        self.x = 111
        print('exit A')

class B(A):
    def __init__(self):
        print('enter B')
        self.y = 222
        A.__init__(self)
        # super().__init__()
        print('exit B')

class C(A):
    def __init__(self):
        print('enter C')
        self.z = 333
        A.__init__(self)
        # super().__init__()
        print('exit C')

class D(B, C):
    def __init__(self):
        print('enter D')
        B.__init__(self)
        C.__init__(self)
        # super().__init__()
        print('exit D')

d = D()
```

9. Python 魔术方法

1. `__str__`, `__repr__`
2. `__init__` 和 `__new__`
    - `__new__` 返回一个对象的实例, `__init__` 无返回值
    - `__new__` 是一个类方法

- 单例模式

```
class A(object):
    '''单例模式'''
    obj = None
    def __new__(cls, *args, **kwargs):
        if cls.obj is None:
            cls.obj = object.__new__(cls)
        return cls.obj
```

3. 数学运算、比较运算
   - 运算符重载
     - +: `__add__(value)`
     - -: `__sub__(value)`
     - *: `__mul__(value)`
     - /: `__truediv__(value)` (Python 3.x), `__div__(value)` (Python 2.x)
     - //: `__floordiv__(value)`
     - %: `__mod__(value)`
     - &: `__and__(value)`
     - |: `__or__(value)`
   - 练习: 实现字典的 `__add__` 方法, 作用相当于 d.update(other)

```
class Dict(dict):
    def __add__(self, other):
        if isinstance(other, dict):
            new_dict = {}
            new_dict.update(self)
            new_dict.update(other)
            return new_dict
        else:
            raise TypeError('not a dict')
```

   - 比较运算符的重载
     - ==: `__eq__(value)`
     - !=: `__ne__(value)`
     - >: `__gt__(value)`
     - >=: `__ge__(value)`
     - <: `__lt__(value)`
     - <=: `__le__(value)`
   - 练习: 完成一个类, 实现数学上无穷大的概念

```
class Inf:
    def __lt__(self, other):
        return False
    def __le__(self, other):
        return False
    def __ge__(self, other):
        return True
    def __gt__(self, other):
        return True
    def __eq__(self, other):
        return False
    def __ne__(self, other):
        return True
```

4. 容器方法
    - `__len__` -> len
    - `__iter__` -> for
    - `__contains__` -> in
    - `__getitem__` 对 `string, bytes, list, tuple, dict` 有效
    - `__setitem__` 对 `list, dict` 有效
    - `__missing__` 对 `dict` 有效, 字典的预留接口, dict 自身并没有实现

```
class Dict(dict):
    def __missing__(self, key):
        self[key] = None  # 当检查到 Key 缺失时, 可以做任何默认
行为
```

5. 可执行对象: `__call__`
6. 上下文管理 with:
    - `__enter__` 进入 `with` 代码块前的准备操作
    - `__exit__` 退出时的善后操作
7. `__setattr__, __getattribute__, __getattr__, __dict__`
    - 常用来做属性监听

```
class A:
    '''TestClass'''
    z = [7,8,9]
    def __init__(self):
        self.x = 123
        self.y = 'abc'

    def __setattr__(self, name, value):
        print('set %s to %s' % (name, value))
        object.__setattr__(self, name, value)
```

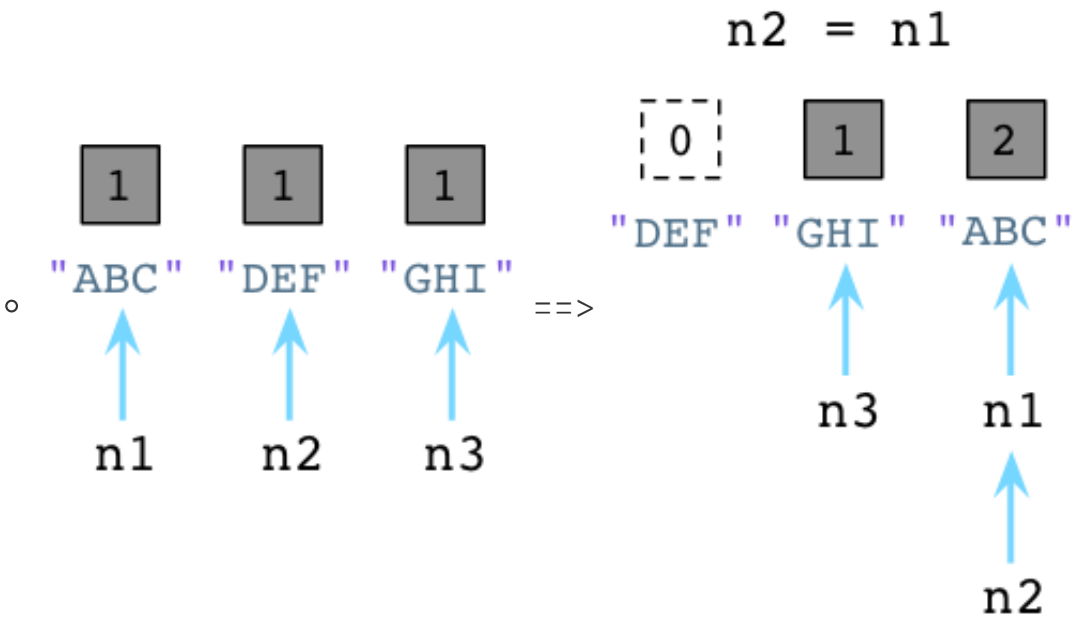
```
    def __getattribute__(self, name):
        print('get %s' % name)
        return object.__getattribute__(self, name)

    def __getattr__(self, name):
        print('not has %s' % name)
        return -1

    def foo(self, x, y):
        return x ** y

# 对比
a = A()
print(A.__dict__)
print(a.__dict__)
```

8. 槽: `__slots__`
    - 固定类所具有的属性
    - 实例不会分配 `__dict__`
    - 实例无法动态添加属性
    - 优化内存分配

```
class A:
    __slots__ = ('x', 'y')
```

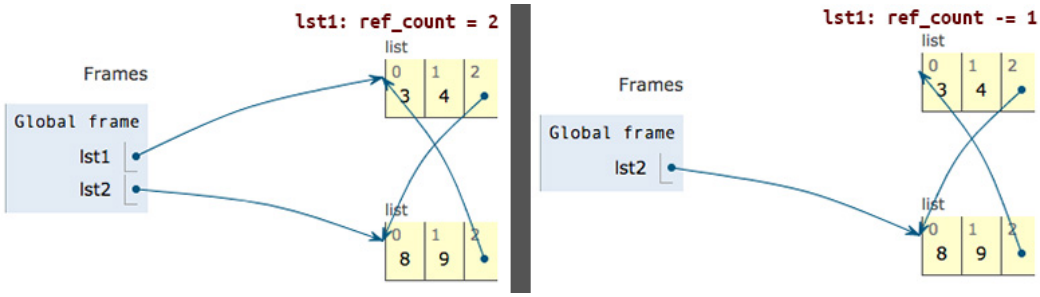
10. 垃圾收集 (GC)

- Garbage Collection (GC)
- 引用计数
    - 优点: 简单、实时性高



- 缺点: 消耗资源、循环引用

```
lst1 = [3, 4]           # lst1->ref_count 1
lst2 = [8, 9]           # lst2->ref_count 1

# lst1 -> [3, 4, lst2]
lst1.append(lst2)       # lst2->ref_count 2

# lst2 -> [8, 9, lst1]
lst2.append(lst1)       # lst1->ref_count 2

del lst1                # lst1->ref_count 1
del lst2                # lst2->ref_count 1
```



- 标记-清除, 分代收集

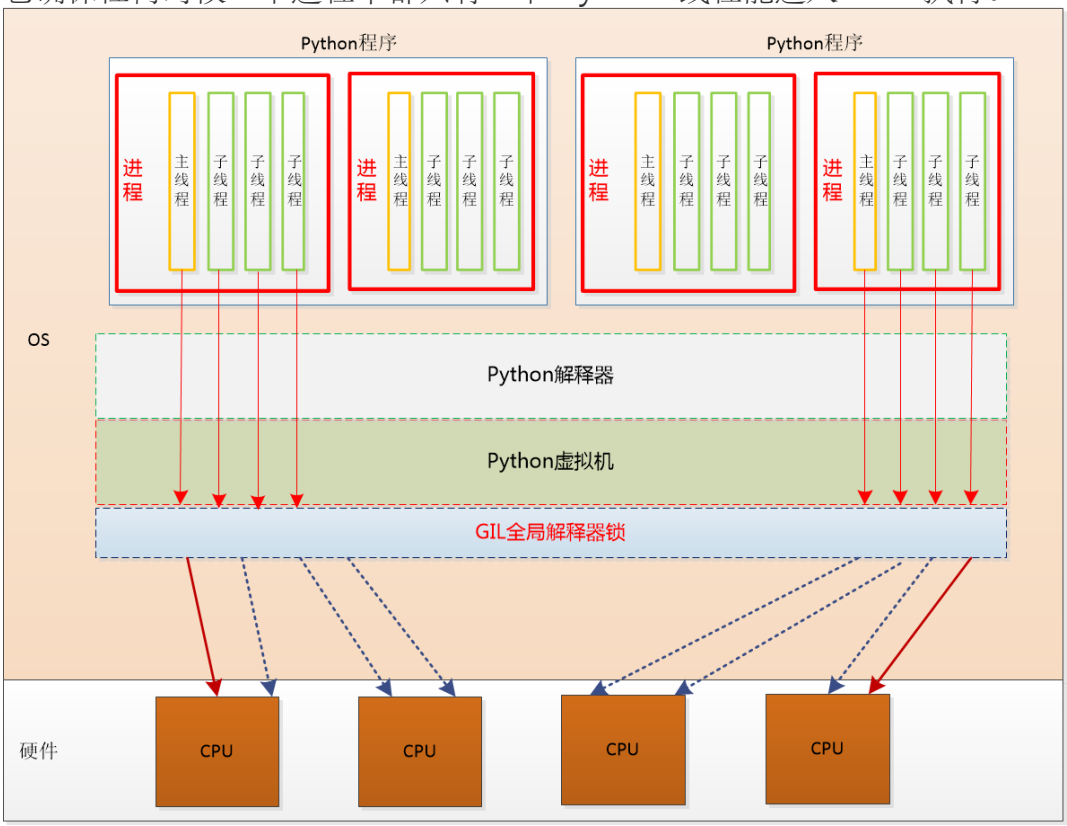
11. Python 性能之困

1. 计算密集型
   - CPU 长时间满负荷运行, 如图像处理、大数据运算、圆周率计算等
   - 计算密集型: 用 C 语言补充
   - Profile, timeit
2. I/O 密集型
   - 网络 IO, 文件 IO, 设备 IO 等
   - 一切皆文件

3. 多任务处理
    - 进程、线程、协程调度的过程叫做上下文切换
    - 进程、线程、协程对比

名称	资源占用	数据通信	上下文切换 (Context)
进程	大	不方便 (网络、共享内存、管道等)	操作系统按时间片切换, 不够灵活, 慢
线程	小	非常方便	按时间片切换, 不够灵活, 快
协程	非常小	非常方便	根据I/O事件切换, 更加有效的利用 CPU

4. 全局解释器锁 (GIL)
    - 它确保任何时候一个进程中都只有一个 Python 线程能进入 CPU 执行。



    - 通过多进程来利用多个 CPU 核心
5. 什么是同步、异步、阻塞、非阻塞?
    - 同步, 异步: 客户端调用服务器接口时
    - 阻塞, 非阻塞: 服务端发生等待
    - 阻塞 -> 非阻塞
    - 同步 -> 异步
6. 事件驱动 + 多路复用

    - 轮询: select, poll
    - 事件驱动: epoll 有效轮询
7. Stackless / greenlets / gevent | tornado / asyncio

```
import asyncio

async def foo(n):
    for i in range(10):
        print('wait %s s' % n)
        await asyncio.sleep(n)
    return i

task1 = foo(1)
task2 = foo(1.5)
tasks = [asyncio.ensure_future(task1),
         asyncio.ensure_future(task2)]

loop = asyncio.get_event_loop()
loop.run_until_complete( asyncio.wait(tasks) )
```

8. 线程安全, 锁
    - 获得锁之后, 一定要释放, 避免死锁
    - 获得锁之后, 执行的语句, 只跟被锁资源有关
    - 区分普通锁 Lock, 可重入锁 RLock
    - 线程之间的数据交互尽量使用 Queue
9. gevent
    - monkey.patch
    - gevent.sleep 非阻塞式等待
    - Queue 协程间数据交互, 避免竞争

12. 一些技巧和误区

1. 格式化打印
    - `json.dumps(data, indent=4, sort_keys=True, ensure_ascii=False)`
    - json 压缩: `json.dumps(obj, separators=[',',':'])`
    - `from pprint import pprint`
2. 确保能取到有效值
    - `d.get(k, default)`
    - `d.setdefault`
    - `defaultdict`
    - `a or b`
    - `x = a if foo() else b`
3. try...except... 的滥用
    - 不要把所有东西全都包住, 程序错误需要报出来
    - 使用 `try...except` 要指明具体错误, `try` 结构不是用来隐藏错误的, 而是

用来有方向的处理错误的

4. 利用 dict 做模式匹配

```
def do1():
    print('i am do1')

def do2():
    print('i am do2')

def do3():
    print('i am do3')

def do4():
    print('i am do4')

mapping = {1: do1, 2: do2, 3: do3, 4: do4}
mod = random.randint(1, 10)
func = mapping.get(mod, do4)
func()
```

5. `inf, -inf, nan`
6. 字符串拼接尽量使用 `join` 方式: 速度快, 内存消耗小
7. property: 把一个方法属性化

```
class C(object):
    @property
    def x(self):
        "I am the 'x' property."
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

8. else 子句: `if, for, while, try`
9. collections 模块
    - defaultdict
    - OrderedDict
    - Counter
    - namedtuple
10. venv, pyenv, 命名空间
    - venv: 创建虚拟环境, 做环境隔离, venv 目录直接放到项目的目录里

- pyenv: 管理 Python 版本