Python常考面试题汇总（附答案）（持续更新）

写在前面

- 本文面向中高级Python开发，太基本的题目不收录。
- 本文只涉及Python相关的面试题，关于网络、MySQL、算法等其他面试必考题会另外开专题整理。
- 不是单纯的提供答案，**抵制八股文！！** 更希望通过代码演示，原理探究等来深入讲解某一知识点，做到融会贯通。
- 部分演示代码也放在了我的github的该目录下。

语言基础篇

Python的基本数据类型

Python3 中有六个标准的数据类型：

- Number（数字）(包括整型、浮点型、复数、布尔型等)
- String（字符串）
- List（列表）
- Tuple（元组）
- Set（集合）
- Dictionary（字典）

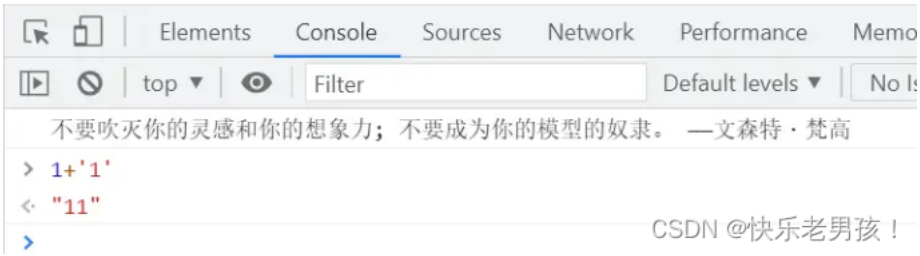
Python3 的六个标准数据类型中：

- 不可变数据（3 个）：Number（数字）、String（字符串）、Tuple（元组）；
- 可变数据（3 个）：List（列表）、Dictionary（字典）、Set（集合）。

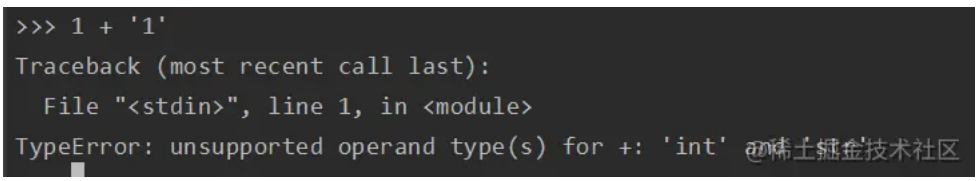
Python是静态还是动态类型？是强类型还是弱类型?

- 动态强类型语言（不少人误以为是弱类型）
- 动态还是静态指的是编译器还是运行期确定类型
- 强类型指的是不会发生隐式类型转换

js就是典型的弱类型语言，例如在console下面模拟一下数字和字符串相加 ，会发现发生了类型转换。



而Python会报TypeError



什么是鸭子类型

“当一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

鸭子类型关注的是对象的行为，而不是类型。比如file,StringIO,socket对象都支持read/write方法，再比如定义了__iter__魔术方法的对象可以用for迭代。

下面用一个例子来模拟鸭子类型：

```
class Duck:
    def say(self):
        print("嘎嘎")


class Dog:
    def say(self):
        print("汪汪")


def speak(duck):
    duck.say()


duck = Duck()
dog = Dog()
speak(duck) # 嘎嘎
speak(dog) # 汪汪
```

什么是自省

自省是运行时判断一个对象类型的能力。

python一切皆对象，用type, id, isinstance获取对象类型信息。

自省，也可以说是反射，自省在计算机编程中通常指这种能力：检查某些事物以确定它是什么、它知道什么以及它能做什么。

与其相关的主要方法:

- hasattr(object, name)检查对象是否具体 name 属性。返回 bool.
- getattr(object, name, default)获取对象的name属性。
- setattr(object, name, default)给对象设置name属性
- delattr(object, name)给对象删除name属性
- dir([object])获取对象大部分的属性
- isinstance(name, object)检查name是不是object对象
- type(object)查看对象的类型
- callable(object)判断对象是否是可调用对象

python3和python2的对比

- print成为函数
- 编码问题。python3不再有unicode对象，默认str就是unicode
- 除法变化。python3除号返回浮点数，如果要返回整数，应使用//
- 类型注解。帮助IDE实现类型检查
- 优化的super()方便直接调用父类函数。Python3.x 和 Python2.x 的一个区别是: Python 3 可以使用直接使用 super().xxx 代替 super(Class, self).xxx :
- 高级解包操作。a, b, *rest = range(10)
- keyword only arguments。限定关键字参数
- chained exceptions。python3重新抛出异常不会丢失栈信息
- 一切返回迭代器。range, zip, map, dict.values, etc. are all iterators
- 性能优化等。。。

python如何传递参数

python官方文档上的话：

>

> “Remember that arguments are passed by assignment in Python. Since assignment just creates references to objects, there’s no alias between an argument name in the caller and callee, and so no call-by-reference per Se.”

准确地说，Python 的参数传递是赋值传递 （pass by assignment），或者叫作对象的 引用传递 （pass by object reference）。Python 里所有的数据类型都是对象，所以参数传递时，只是让新变量与原变量指向相同的对象而已，并不存在值传递或是引用传递一说。

根据对象的引用来传递，根据对象是可变对象还是不可变对象，得到两种不同的结果。如果是可变对象，则直接修改。如果是不可变对象，则生产新对象，让形参指向新对象

可以具体结合下面的代码实例来模拟：

```
def flist(l):
    l.append(0)
    print(id(l))    # 每次打印的id相同
    print(l)


ll = []
print(id(ll))
flist(ll)   # [0]
flist(ll)   # [0,0]

print("=" * 10)


def fstr(s):
    print(id(s)) # 和入参ss的id相同
    s += "a"
    print(id(s))  # 和入参ss的id不同，每次打印结果不相同
    print(s)


ss = "sun"
print(id(ss))
fstr(ss)    # a
fstr(ss)    # a

```

python的可变/不可变对象

不可变对象： bool/int/float/tuple/str/ frozenset 可变对象：list/set/dict

这里继续看两个代码例子，看下输出是什么

```
def clear_list(l):
    l = []

ll = [1,2,3]
clear_list(ll)
print(ll)

def fl(l=[1]):
    l.append(1)
    print(l)
fl()
fl()
```

答案是

```
[1,2,3]
[1]
[1,1]
```

对于第一题，`l = []`这一步，创建了一个新的对象，并将l贴上去（注意函数里面的l和外面的l是形参和实参的区别，不要以为是同一个），所以原来的 l 并没有改变

对于第二题，默认参数只计算一次。

有兴趣的小伙伴可以再试一下这个例子：

```
a = 1
def fun(a):
    print("func_in",id(a))
    a = 2
    print("re-point",id(a), id(2))
print("func_out",id(a), id(1))

fun(a)
```

答案是：

```
func_out 2602672810288 2602672810288
func_in 2602672810288
re-point 2602672810320 2602672810320
```

关于Python的参数传递,可变/不可变对象，再推荐一个stackoverflow上面的回答。

> Arguments are passed by assignment. The rationale behind this is twofold:
>
> the parameter passed in is actually a reference to an object (but the reference is passed by value)some data types are mutable, but others aren't
>
> So: If you pass a mutable object into a method, the method gets a reference to that same object and you can mutate it to your heart's delight, but if you rebind the reference in the method, the outer scope will know nothing about it, and after you're done, the outer reference will still point at the original object.
>
> If you pass an immutable object to a method, you still can't rebind the outer reference, and you can't even mutate the object.

Python中的 *args 和 **kwargs

用来处理可变参数，*args被打包成tuple，**kwargs被打包成dict

我们看一些代码例子：

```
def print_multiple_args(*args):
    print(type(args), args)
    for idx, val in enumerate(args):  # enumerate()枚举函数
        print(idx, val)

print_multiple_args('a', 'b', 'c')
# 通过将列表前加*打包成关键字参数，指明了接收值参数必须是*args
print_multiple_args(*['a', 'b', 'c'])

def print_kwargs(**kwargs):
    print(type(kwargs), kwargs)
    for k, v in kwargs.items():
        print('{}: {}'.format(k, v))


print_kwargs(a=1, b=2)
# 给字典前加**打包成关键字参数,指明接收值的参数必须是**kwargs
print_kwargs(**dict(a=1, b=2))

def print_all(a, *args, **kwargs):
    print(a)
    if args:
        print(args)
    if kwargs:
```

```
        print(kwargs)

print_all('hello', 'world', name='monki')
```

输出为：

```
<class 'tuple'> ('a', 'b', 'c')
0 a
1 b
2 c
<class 'tuple'> ('a', 'b', 'c')
0 a
1 b
2 c
<class 'dict'> {'a': 1, 'b': 2}
a: 1
b: 2
<class 'dict'> {'a': 1, 'b': 2}
a: 1
b: 2
hello
('world',)
{'name': 'monki'}
```

python异常机制

可参考Python官方文档上的异常层级分类

docs.python.org/zh-cn/3/lib…

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
```


python异常代码块示例：

```python
try:
    # func   # 可能会抛出异常的代码
except (Exception1, Exception2) as e:  # 可以捕获多个异常并处理
    # 异常处理的代码
else:
    # pass  # 异常没有发生的时候代码逻辑
finally:
    pass     # 无论异常有没有发生都会执行的代码，一般处理资源的关闭和释放
```

什么是Python中的GIL?

全局解释器锁 GIL，英文名称为 Global Interpreter Lock，它是解释器中一种线程同步的方式。

对于每一个解释器进程都具有一个 GIL ，它的直接作用是限制单个解释器进程中多线程的并行执行，使得即使在多核处理器上对于单个解释器进程来说，在同一时刻运行的线程仅限一个。 对于 Python 来讲，GIL 并不是它语言本身的特性，而是 CPython 解释器的实现特性。

Python 代码被编译后的字节码会在解释器中执行，在执行过程中，存在于 CPython 解释器中的 GIL 会致使在同一时刻只有一个线程可以执行字节码。 GIL 的存在引起的最直接的问题便是：在一个解释器进程中通过多线程的方式无法利用多核处理器来实现真正的并行。

因此，Python的多线程是伪多线程，无法利用多核资源，同一个时刻只有一个线程在真正的运行。

GIL的限制了程序的多核执行

- 同一个时间只能有一个线程执行字节码
- CPU密集程序难以利用多核优势
- IO期间会释放GIL，对IO密集程序影响不大

面对GIL的存在，我们有可以有多个方法帮助我们提升性能

- 在 IO 密集型任务下，我们可以使用多线程或者协程来完成。
- 可以选择更换 Jython 等没有 GIL 的解释器，但并不推荐更换解释器，因为会错过众多 C 语言模块中的有用特性。
- CPU密集可以使用多进程+进程池。
- 将计算密集型任务转移到 Python 的 C / C++ 扩展模块中完成。

为什么有了GIL还要关注线程安全?

GIL 保证的是每一条字节码在执行过程中的独占性，即每一条字节码的执行都是原子性的。GIL 具有释放机制，所以 GIL 并不会保证字节码在执行过程中线程不会进行切换，即在多个字节码之间，线程具有切换的可能性。

我们可以用python的dis模块去查看a += 1执行的字节码，发现需要有多个字节码去完成，线程具有切换的可能性，所以它是非线程安全的。

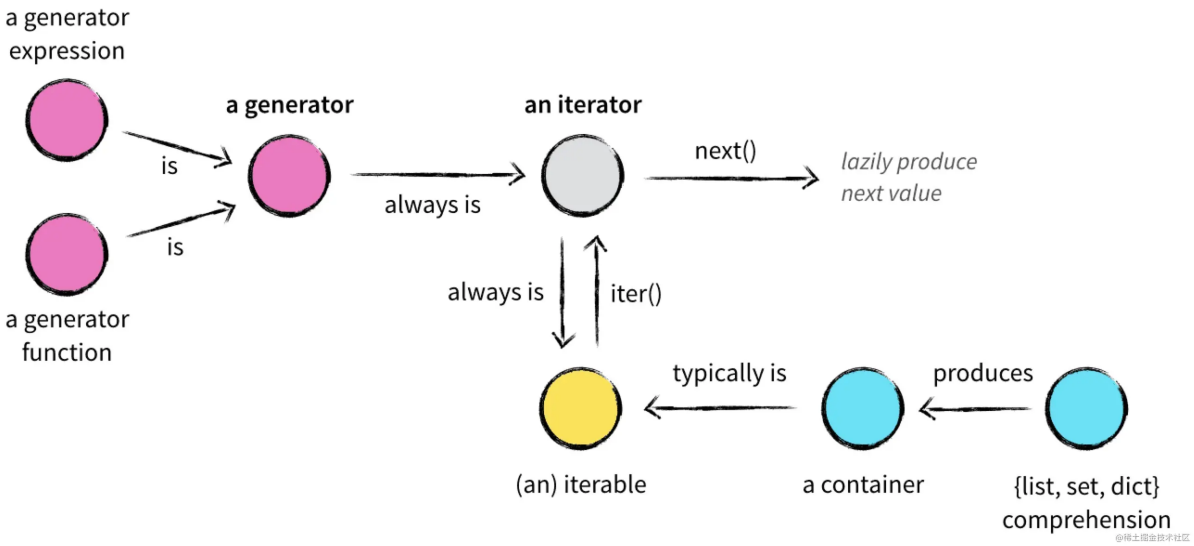
一个操作如果是一个字节码指令可以完成就是原子的，非原子操作不是线程安全的，原子的是可以保证线程安全的。

GIL 和线程互斥锁的粒度是不同的，GIL 是 Python 解释器级别的互斥，保证的是解释器级别共享资源的一致性，而线程互斥锁则是代码级（或用户级）的互斥，保证的是 Python 程序级别共享数据的一致性，所以我们仍需要线程互斥锁及其他线程同步方式来保证数据一致。

具体关于Python的GIL的介绍，可参考我的另一篇文章《详解Python中的GIL》

什么是迭代器和生成器?

这张图比较精彩，把各种概念都总结了。



容器（container）

container 可以理解为把多个元素组织在一起的数据结构，container 中的元素可以逐个地迭代获取，可以用 in, not in 关键字判断元素是否包含在容器中。比如Python中常见的container对象有list，deque，set

可迭代对象(iterables)

大部分的 container 都是可迭代对象，比如 list or set 都是可迭代对象，可以说只要是可以返回一个迭代器的都可以称作可迭代对象。

迭代器（iterator）

python中的容器有许多，比如列表、元组、字典、集合等，对于容器，可以很直观地想象成多个元素在一起的单元，所有的容器都是可迭代的（iterable）。

我们通常使用for in 语句对可迭代的对象进行枚举，其底层机制在于：

而可迭代对象，通过 iter() 函数返回一个迭代器（iterator），迭代器提供了一个 next 的方法。调用用这个方法后，你要么得到这个容器的下一个对象，要么得到一个StopIteration 的错误。

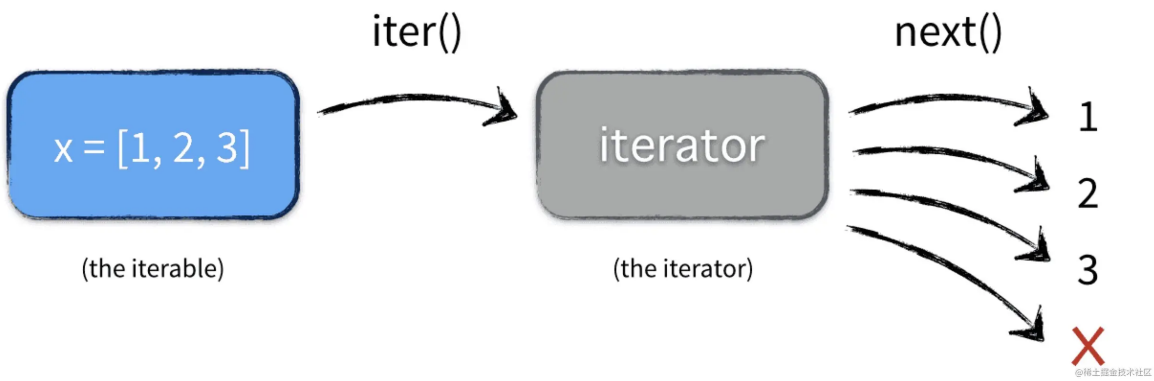
举个例子：

```
>>> x = [1, 2, 3]
>>> # Get the iterator
>>> y = iter(items) # Invokes items.__iter__()
>>> # Run the iterator
>>> next(y) # Invokes it.__next__()
1
>>> next(y)
2
>>> next(y)
3
>>> type(x)
<class 'list'>
>>> type(y)
<class 'list_iterator'>
>>> next(y)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
StopIteration
>>>

```

上面的例子中，x=[1,2,3]是可迭代对象，这里也叫容器。y=iter(x)则是迭代器，且实现了__iter__和__next__方法。

它们之间的关系如下图所示：



可见通过 iter 方法后就是迭代器。它是一个带状态的对象，调用 next 方法的时候返回容器中的下一个值，可以说任何实现了iter和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了，则抛异常。

迭代器就像一个懒加载的工厂，等到有人需要的时候才给它生成值返回，没调用的时候就处于休眠状态等待下一次调用。

生成器（generator）

生成器(generator)可以简单理解为懒人版本的迭代器。

它相比于迭代器的优势是，生成器并不会像迭代器一样占用大量内存。比如声明一个迭代器：[i for i in range(100000000)]就可以声明一个包含一亿个元素的列表，每个元素在生成后都会保存到内存中。但实际上我们也许并不需要保存那么多东西，只希望在你用 next() 函数的时候，才会生成下一个变量，因此生成器应运而生，在python中的写法为(i for i in range(100000000))

此外，生成器还可以有别的形式，比如生成器函数，通过yield关键字，把结果返回到next()方法中，举个例子：

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment
```

```
for n in frange(0, 2, 0.5):
    print(n)

0
0.5
1.0
1.5
```

相比于迭代器，生成器具有以下优点：

1. 减少内存
2. 延迟计算
3. 有效提高代码可读性

什么是闭包?

在函数内部再定义一个函数，并且这个函数用到了外边函数的变量，那么将这个函数以及用到的一些变量称之为闭包。

简单的说，如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包(closure)。来看几个简单的例子:

最简单的例子，实现加法

```
def addx(x):
    def adder(y):
        return x + y
    return adder

c = addx(8)
print(type(c))
print(c.__name__)
print(c(10))
```

```
<class 'function'>
adder
18
```

利用闭包实现斐波那契数列

```
from functools import wraps

def cache(func):
    store = {}
    @wraps(func)
    def _(n):
        if n in store:
            return store[n]
        else:
            res = func(n)
            store[n] = res
            return res
    return _

@cache
def f(n):
    if n <= 1:
        return 1
    return f(n-1) + f(n-2)

print(f(10))
```

什么是python深拷贝和浅拷贝?

注意引用和copy(),deepcopy()的区别

可以具体看下面这个例子：

```
import copy

a = [1, 2, 3, 4, ['a', 'b']]  # 原始对象

b = a  # 赋值，传对象的引用
c = copy.copy(a)  # 对象拷贝，浅拷贝
d = copy.deepcopy(a)  # 对象拷贝，深拷贝

a.append(5)  # 修改对象a
a[4].append('c')  # 修改对象a中的['a', 'b']数组对象

print('a = ', a)
print('b = ', b)
print('c = ', c)
print('d = ', d)
```

输出结果为：

```
a =  [1, 2, 3, 4, ['a', 'b', 'c'], 5]
b =  [1, 2, 3, 4, ['a', 'b', 'c'], 5]
c =  [1, 2, 3, 4, ['a', 'b', 'c']]
d =  [1, 2, 3, 4, ['a', 'b']]
```

Python的内存管理

Python 有内存池机制，Pymalloc 机制，用于对内存的申请和释放管理。先来看一下为什么有内存池：

当创建大量消耗小内存的对象时，c 中频繁调用 new/malloc 会导致大量的内存碎片，致使效率降低。

内存池的概念就是预先在内存中申请一定数量的，大小相等的内存块留作备用，当有新的内存需求时，就先从内存池中分配内存给这个需求，不够了之后再申请新的内存。这样做最显著的优势就是能够减少内存碎片，提升效率。

查看源码，可以看到 Pymalloc 对于小的对象，Pymalloc 会在内存池中申请空间，一般是少于236kb，如果是大的对象，则直接调用 new/malloc 来申请新的内存空间。

有了内存的创建，那就需要回收，垃圾回收机制，也是 Python 面试当中必问的一个知识点，接下来看看垃圾回收机制是什么。

Python的垃圾回收机制

GC要做的有 2 件事，一是找到内存中无用的垃圾对象资源，二是清除找到的这些垃圾对象，释放内存给其他对象使用。

Python GC主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记-清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率。

引用计数

每一个对象在源码中的结构体表示如下：

```
typedef struct_object {
 int ob_refcnt;
 struct_typeobject *ob_type;
} PyObject;
```

PyObject是每个对象必有的内容，其中ob_refcnt就是做为引用计数。当一个对象有新的引用时，它的ob_refcnt就会增加，当引用它的对象被删除，它的ob_refcnt就会减少。引用计数为0时，该对象立即被回收，对象占用的内存空间将被释放。

优点:

- 简单
- 实时性，一旦没有引用，内存就直接释放了。不用像其他机制等到特定时机。

缺点:

- 需要额外的空间维护引用计数。
- 不能解决对象的循环引用。(主要缺点)

接下来说一下什么是循环引用:

A 和 B 相互引用而且没有外部引用 A 与 B 中的任何一个。也就是对象之间互相应用，导致引用链形成一个环。

```
>>>>>>a = { } #对象A的引用计数为 1
>>>b = { } #对象B的引用计数为 1
>>>a['b'] = b  #B的引用计数增1
>>>b['a'] = a  #A的引用计数增1
>>>del a #A的引用减 1，最后A对象的引用为 1
>>>del b #B的引用减 1，最后B对象的引用为 1
```

执行 del 后，A、B 对象已经没有任何引用指向这两个对象，但是这两个对象各包含一个对方对象的引用，虽然最后两个对象都无法通过其它变量来引用这两个对象了，这对 GC 来说就是两个非活动对象或者说是垃圾对象。理论上是需要被回收的。

按上面的引用计数原理，要计数为 0 才会回收，但是他们的引用计数并没有减少到零。因此如果是使用引用计数法来管理这两对象的话，他们并不会被回收，它会一直驻留在内存中，就会造成了内存泄漏（内存空间在使用完毕后未释放）。

为了解决对象的循环引用问题，Python 引入了标记清除和分代回收两种 GC 机制。

标记-清除机制

标记清除主要是解决循环引用问题。

标记清除算法是一种基于追踪回收（tracing GC）技术实现的垃圾回收算法。

它分为两个阶段：第一阶段是标记阶段，GC 会把所有的 活动对象 打上标记，第二阶段是把那些没有标记的对象 非活动对象 进行回收。那么 GC 又是如何判断哪些是活动对象哪些是非活动对象的呢?

对象之间通过引用（指针）连在一起，构成一个有向图，对象构成这个有向图的节点，而引用关系构成这个有向图的边。从根对象（root object）出发，沿着有向边遍历对象，可达的（reachable）对象标记为活动对象，不可达的对象就是要被清除的非活动对象。根对象就是全局变量、调用栈、寄存器。

分代技术

分代回收是一种以空间换时间的操作方式。

Python 将内存根据对象的存活时间划分为不同的集合，每个集合称为一个代，Python 将内存分为了 3“代”，分别为年轻代（第 0 代）、中年代（第 1 代）、老年代（第 2 代），他们对应的是 3 个链表，它们的垃圾收集频率与对象的存活时间的增大而减小。新创建的对象都会分配在年轻代，年轻代链表的总数达到上限时，Python 垃圾收集机制就会被触发，把那些可以被回收的对象回收掉，而那些不会回收的对象就会被移到中年代去，依此类推，老年代中的对象是存活时间最久的对象，甚至是存活于整个系统的生命周期内。同时，分代回收是建立在标记清除技术基础之上。

面向对象篇

什么是组合和继承?

- 组合是使用其他的类实例作为自己的一个属性（Has-a关系）
- 继承是子类继承父类的属性和方法（Is a关系）
- 优先使用组合保持代码简单

类变量和实例变量的区别?

- 类变量由所有实例共享
- 实例变量由实例单独享有，不同实例之间不影响
- 当我们需要在一个类的不同实例之间共享变量的时候使用类变量

classmethod和staticmethod区别?

- 都可以通过Class.method()的方式使用
- classmethod第一个参数是cls，可以引用类变量
- staticmethod使用起来和普通函数一样，只不过放在类里去组织
- classmethod是为了使用类变量，staticmethod是代码组织的需要，完全可以放到类之外

通过下面这个例子，看到类变量，实例变量，类方法，普通方法，静态方法的使用

```
class Person:
    Country = 'china'

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

    def print_name(self):
        print(self.name)

    @classmethod
    def print_country(cls):
        print(cls.Country)

    @staticmethod
    def join_name(first_name, last_name):
        return print(last_name + first_name)

a = Person("Bruce", "Lee")
a.print_country()
a.print_name()
a.join_name("Bruce", "Lee")
Person.print_country()
Person.print_name(a)
Person.join_name("Bruce", "Lee")
```

__new__和__init__区别?

- __new__是一个静态方法,而__init__是一个实例方法.
- __new__方法会返回一个创建的实例,而__init__什么都不返回.
- 只有在__new__返回一个cls的实例时后面的__init__才能被调用.
- 当创建一个新实例时调用__new__,初始化一个实例时用__init__.

我们可以做几个有趣的实验。

```
class Person:
    def __new__(cls, *args, **kwargs):
        print("in __new__")
        instance = super().__new__(cls)
        return instance

    def __init__(self, name, age):
        print("in __init__")
        self._name = name
        self._age = age

p = Person("zhiyu", 26)
print("p:", p)
```

这段程序输出为：

```
in __new__
in __init__
p: <__main__.Person object at 0x00000261FE562E50>
```

可以看到先执行 new 方法创建对象，然后 init 进行初始化。假设将new方法中不返还该对象，会有什么结果了？

```
class Person:
    def __new__(cls, *args, **kwargs):
        print("in __new__")
        instance = super().__new__(cls)
        #return instance
```

```
    def __init__(self, name, age):
        print("in __init__")
        self._name = name
        self._age = age

p = Person("zhiyu", 26)
print("p:", p)
```

发现如果new没有返回实例化对象，init就没法初始化了。

输出结果为：

```
in __new__
p: None
```

什么是元类?

元类(meta class)是创建类的类

- 元类允许我们控制类的生成，比如修改类的属性等
- 使用type来定义元类
- 元类最常见的一个使用场景就是ORM框架

什么是Python中的装饰器?

- python中一切皆对象，函数也可以当做参数传递
- 装饰器是接受函数作为参数，添加功能后返回一个新函数的函数（类）
- python中通过@使用装饰器，语法糖

例子：编写一个记录函数耗时的装饰器：

```
import time

def log_time(func):  # 接受一个函数作为参数
    def _log(*args, **kwargs):
        beg = time.time()
        res = func(*args, **kwargs)
        print('use time: {}'.format(time.time() - beg))
        return res

    return _log

@log_time  # 装饰器语法糖
def mysleep():
    time.sleep(1)

mysleep()

# 另一种写法，和上面的调用方式等价
def mysleep2():
    time.sleep(1)

newsleep = log_time(mysleep2)
newsleep()
```

当然，装饰器有可以带参数

```
def log_time_with_param(use_int):
    def decorator(func): # 接受一个函数作为参数
        def _log(*args, **kwargs):
            beg = time.time()
            res = func(*args, **kwargs)
            if use_int:
                print('use time: {}'.format(int(time.time()-beg)))
            else:
```

```
                print('use time: {}'.format(time.time()-beg))
            return res
        return _log
    return decorator

@log_time_with_param(True)
def my_sleep6():
    time.sleep(1)
```

也可以用类做装饰器

```
class LogTime:
    def __call__(self, func): # 接受一个函数作为参数
        def _log(*args, **kwargs):
            beg = time.time()
            res = func(*args, **kwargs)
            print('use time: {}'.format(time.time()-beg))
            return res
        return _log

@LogTime()
def mysleep3():
    time.sleep(1)

mysleep3()
```

还可以给类装饰器加上参数

```
class LogTime2:
    def __init__(self, use_int=False):
        self.use_int = use_int

    def __call__(self, func): # 接受一个函数作为参数
        def _log(*args, **kwargs):
            beg = time.time()
            res = func(*args, **kwargs)
            if self.use_int:
                print('use time: {}'.format(int(time.time()-beg)))
            else:
                print('use time: {}'.format(time.time()-beg))
            return res
        return _log

@LogTime2(True)
def mysleep4():
    time.sleep(1)

mysleep4()

@LogTime2(False)
def mysleep5():
    time.sleep(1)

mysleep5()
```

另外讲一下装饰器的输出顺序

```
@a
@b
@c
def f ():
    pass
```

上面一段程序的执行顺序为`f = a(b(c(f)))`

python里的魔术方法

- __new__用来生成实例
- __init__用来初始化实例

这两个上面有提到过，此外魔术方法还有：

- __call__

先需要明白什么是可调用对象，平时自定义的函数、内置函数和类都属于可调用对象，但凡是可以把一对括号()应用到某个对象身上都可称之为可调用对象，判断对象是否为可调用对象可以用函数 callable。

可参照下面的代码示例理解：

```
class A:
    def __init__(self):
        print("__init__ ")
        super(A, self).__init__()

    def __new__(cls):
        print("__new__ ")
        return super(A, cls).__new__(cls)

    def __call__(self):  # 可以定义任意参数
        print('__call__ ')

a = A()
a()
print(callable(a))  # True
```

输出结果为：

```
__init__
__call__
True
```

执行a()时会打印出__call__。a 是一个实例化对象，也是一个可调用对象。

- __del__, 析构函数，当删除一个对象时，则会执行此方法，对象在内存中销毁时，自动会调用此方法。

```
import time
class People:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __del__(self): # 在对象被删除的条件下，自动执行
        print('__del__')

obj = People("zhiyu", 26)
# del obj
time.sleep(5)
```

等到程序执行完成后，可以发现5s后，控制台输出了

```
__del__
```


点点关注喵~