Python面试题记录总结

一、is 相关 以及部分常量内容

1、is和==的关系

==是python标准操作符中的比较操作符，用来比较判断两个对象的value(值)是否相等；

is也被叫做同一性运算符，这个运算符比较判断的是对象间的唯一身份标识，也就是id是否相同

1.1、小整数使用对象池存储

Python 为了优化速度，使用了小整数对象池，避免为整数频繁申请和销毁内存空间。而Python 对小整数的定义是 [-5, 256]，只有数字在-5到256之间它们的id才会相等，超过了这个范围就不行了，同样的道理，字符串对象也有一个类似的缓冲池，超过区间范围内自然不会相等了。

总的来说，**只有数值型和字符串型，并且在通用对象池中的情况下，a is b才为True**，否则当a和b是int，str，tuple，list，dict或set型时，a is b均为False。**(在命令行中才对整数范围有限制，在IDE中做了优化，数字多大都是同一个)**

1.2、字符串的intern机制存储

python中虽然字符串对象也是不可变对象,但python有个**intern机制**，简单说就是维护一个字典，这个字典维护已经创建字符串(key)和它的字符串对象的地址(value),每次创建字符串对象都会和这个字典比较,没有就创建，重复了就用指针进行引用就可以了。**相当于python对于字符串也是采用了对象池原理。（对于有特殊字符的字符串，就不是同一个对象）**

1.3、由于在Python中，部分对象为内置常量，所以两个变量绑定时，这两个变量为同一对象

如：True, False, None（NoneType 类型的唯一值）, NotImplemented, Ellipsis，__debug__
注解 变量名 None，False，True 和 __ debug__ 无法重新赋值（赋值给它们，即使是属性名，将引发 SyntaxError ），所以它们可以被认为是“真正的”常数。

使用模块和类实现常量

const.py

```
class _const:
    class ConstError(TypeError) : pass

    def __setattr__(self, key, value):
       # self.__dict__
        if key in self.__dict__:
            data="Can't rebind const (%s)" % key
            raise self.ConstError(data)
        self.__dict__[key] = value

import sys
sys.modules[__name__] = _const() # 把const类注册到sys.modules这个全局字典中
```

test.py

```python
import const
const.c=1
print(const.c)  # 1
const.c=2   #const.ConstError: Can't rebind const (c)
```

2、总结

所以在python中如果创建了多个变量(不同变量名，此外不是通过变量引用方式创建的变量)，那么这些变量的引用地址都是不一样的。那么这些变量之间使用is 去比较的话，就是False的结果。但是除了小整数和字符串除外。

二、数据类型相关

1、列表

1.1、列表的+=操作右边只要是可迭代对象就行，不用非得是列表

```
>>> l = [1,2]
>>> l += '34'
>>> l
[1, 2, '3', '4']
```

1.2、列表的切片赋值

列表的切片赋值的运算符右边也必须是可迭代对象

注意：对于**步长不等于1**的切片赋值，赋值运算符的右侧可迭代对象提供的元素个数一定要等于切片切出的段数

```
>>> l = [1,2,3]
>>> l[0:2] = 'ab'
>>> l
['a', 'b', 3]

>>> l[::2] = 'abc'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 3 to extended slice of size 2
```

1.3 、深拷贝与浅拷贝

浅拷贝：是指在复制过程中只复制一层变量，创建新对象，其内容是原对象的引用

```
>>> a = [1,2,[1]]
>>> b = a.copy()
>>> a is b
False
>>> a[-1] is b[-1]
True
```

深拷贝：和浅拷贝对应，深拷贝拷贝了对象的所有元素，包括多层嵌套的元素。深拷贝出来的对象是一个全新的对象，不再与原来的对象有任何关联

注：深拷贝通常只对可变对象进行复制，不可变对象通常不变

```
>>> a = [1,2,[1]]
>>> import copy
>>> b = copy.deepcopy(a)
>>> a is b
False
>>> a[-1] is b[-1]
False
```

1.4、列表推导式

列表推导式的嵌套：
语法：
[表达式1
for 变量1 in 可迭代对象1 if 真值表达式1
for 变量2 in 可迭代对象2 if 真值表达式2
...]

```
L1 = [2,3,5]
        L2 = [7,11,13]
        #将L1中的全部元素与L2中的元素依次相乘后放到L3中
        L3 = [x * y for x in L1 for y in L2]
        print(L3) #[14, 22, 26, 21, 33, 39, 35, 55, 65]
```

2、字典

2.1、字典的构造

字典的构造函数dict
dict() 创建一个空字典，等用于{}
dict(iterable) 用可迭代对象初始化一个字典
dict(**kwargs) 关键字传参形式生成一个字典
示例：
d = dict()
d = dict([('name','tarena'),('age',15)]) # {'name': 'tarena', 'age': 15}
d = dict(name='tarena',age=15) # {'name': 'tarena', 'age': 15}

3、集合

3.1、集合的创建

集合内的元素必须都为不可变对象（可hash）

3.2、集合的运算

交集 并集 补集 子集 超集

```
& 生成两个集合的交集
s1 = {1,2,3}
s2 = {4,2,3}
s = s1 & s2  # s = {2,3}
| 生成两个集合的并集
s1 = {1,2,3}
```

```
s2 = {4,5,6}
s = s1 | s2  # s = {1,2,3,4,5,6}
- 生成两个集合的补集
s1 = {1,2,3}
s2 = {4,5,6}
s = s1 - s2  # s = {1,2,3} s中的元素属于s1但不属于s2
^ 生成两个集合的对称补集
s1 = {1,2,3}
s2 = {2,3,6}
s = s1 ^ s2  # s = {1,6} s中的元素只属于s1或只属于s2
< 判断一个集合是另一个集合的子集
> 判断一个集合是另一个集合的超集
s1 = {1,2,3}
s2 = {2,3}
s1 > s2      # True 判断超集
s2 < s1      # True 判断子集
== != 集合相同/不同
s1 = {1,2,3}
s1 = {2,3,1}
s1 == s2  # True
s1 != s2  # False
```

三、函数相关

1、函数的参数说明

缺省参数，位置形参，*元组形参，命名关键字形参和**字典形参可以混合使用

函数参数自左至右的顺序为：位置形参、、元组形参、命名关键字形参、字典形参
综合示例：

```
def f1(a,b,*args,c,**kwargs): 这里*args收集了除了a、b外的所有按位置传参的参数
        pass
    f1(1, 2, 3, 4, d=6, c=5, e=7) # a=1, b=2, c=5
    f1(*'hello', d=6, **{'c':5, 'e':7})
```

2、eval、exec、complie函数

2.1、eval

eval（source, globals=None, locals=None）

用于执行单个的python表达式，globals和locals中以字典的形式分别为表达式传入全局变量和局部变量的命名空间

2.2、exec

exec(source, globals=None, locals=None,)

动态执行python代码。也就是说exec可以执行复杂的python代码，而不像eval函数那样只能计算一个表达式的值

2.3、eval()函数和exec()函数的区别

eval()函数只能计算单个表达式的值，而exec()函数可以动态运行代码段。

eval()函数可以有返回值，而exec()函数返回值永远为None。

2.4、complie

complie对象可以作为eval和exec函数的参数

compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)

参数说明:

source：字符串或AST对象，表示需要进行编译的python代码

filename：指定需要编译的代码文件，如果不是文件读取代码则传递一些可辨认的值。

mode：用于标识必须当做那类代表来编译；如果source是由一个代码语句序列组成，则指定mode='exec'，如果source由单个表达式组成，则指定mode='eval'；如果source是由一个单独的交互式语句组成，则指定modo='single'。**必须要指定，不然肯定会报错。**

```
s = """              #一大段代码
for x in range(10):
    print(x, end='')
print()
"""
code_exec = compile(s, '<string>', 'exec')   #必须要指定mode，指定错了和不指定就会报错。
code_eval = compile('10 + 20', '<string>', 'eval')   #单个表达式
code_single = compile('name = input("Input Your Name: ")', '<string>', 'single')   #交互式

a = exec(code_exec)   使用的exec，因此没有返回值
b = eval(code_eval)

c = exec(code_single)  交互
d = eval(code_single)

print('a: ', a)
print('b: ', b)
print('c: ', c)
print('name: ', name)
print('d: ', d)
print('name; ', name)
```

结果

```
0123456789  #有print就会打印
Input Your Name: kebi
Input Your Name: kebi
a:  None
b:  30
c:  None
name:  kebi
d:  None
name;  kebi
```

3、高阶函数

满足下列任意一个条件的函数即为高阶函数：
1.函数接收一个或多个函数作为参数传入
2.函数返回一个函数

python中内建（Bulitins）的高阶函数：map filter sorted

3.1、map

map(fun, *iterables)

用函数和可迭代对象中的每一个元素作为参数计算出新的**可迭代对象**。当最短的一个可迭代对象不再提供数据时，此可迭代对象生成结束

3.2、filter

filter(func,iterable)

筛选可迭代对象iterable中的数据，返回一个**可迭代对象**，此可迭代对象将对iterable进行筛选

函数func将对每个元素进行求值，返回False则将此数据丢弃，返回True则保留此数据

3.3、sorted

sorted(iterable, key=None, reverse=False)

将原可迭代对象的数据进行排序，生成排序后的**列表**

```
L = [5, -2, -4, 0, 3, 1]
L2 = sorted(L, key=abs)  # [0, 1, -2, 3, -4, 5]
```

4、装饰器

函数装饰器是指装饰的是一个函数，传入的是一个函数，返回的也是一个函数的函数

装饰器在加载模块时立即执行

```
#定义一个装饰器函数
def mydeco(fn):
    def fx():
        print('++++++++++')
        fn()
        print('----------')
    return fx

#定义函数并装饰
@mydeco  #等同于myfunc = mydeco(myfunc)
def myfunc():
    print('myfunc被调用')
myfunc()
```

4.1被装饰的函数带有参数

```
def log_time(func):
    def make_decorater(*args,**kwargs): # 接受调用语句的实参，在下面传递给被装饰函数（原函数）
        print('现在开始装饰')
        test_func= func(*args,**kwargs) # 如果在这里return，则下面的代码无法执行，所以引用并在下面返回
        print('现在结束装饰')
        return test_func # 因为被装饰函数里有return，所以需要给调用语句（test（2））一个返回，又因为test_func = func(*args,**kwargs)已经调用了被装饰函数，这里就不用带（）调用了，区别在于运行顺序的不同。
    return make_decorater


@log_time
def test(num):
    print('我是被装饰的函数')
    return num+1

a= test(2) # test(2)=make_decorater(2)
print(a)

输出：
现在开始装饰
我是被装饰的函数
现在结束装饰
3
```

4.2、装饰器函数带有参数

在外层多加一层函数，用来接收装饰器需要的参数

```
def get_parameter(*args,**kwargs): # 工厂函数，用来接受@get_parameter('index.html/')的'index.html/'
    def log_time(func):
        def make_decorater():
            print(args,kwargs)
            print('现在开始装饰')
            func()
            print('现在结束装饰')
        return make_decorater
    return log_time

@get_parameter('index.html/')
def test():
    print('我是被装饰的函数')
    # return num+1

test() # test()=make_decorater()

输出：
现在开始装饰
我是被装饰的函数
现在结束装饰
```

4.3 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__两个内置的函数

不带参数的类装饰器，用__init__接受被装饰函数，__call__实现装饰逻辑

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

    def __call__(self, *args, **kwargs):
        print("{func}() is running...".format(func=self.func.__name__))
        return self.func(*args, **kwargs)


@logger
def say(something):
    print("say {}!".format(something))
```

带参数的类装饰器，用__init__接受传入参数，__call__接受被装饰函数，实现装饰逻辑

```
class logger(object):
    def __init__(self, level='INFO'):
        self.level = level

    def __call__(self, func):
        def wrapper(*args, **kwargs):
         print("{level}:{func}()".format(level=self.level,
self.func.__name__))
          func(*args, **kwargs)
             return wrapper


@loggerK(level='WARNING')
def say(something):
    print("say {}!".format(something))
```

5、函数的缺省参数为可变类型

函数的缺省参数为可变数据类型时，一经调用即创建，不会每次调用都创建，当实参缺省时即调用（默认参数绑定在函数对象内部，且随函数的生命一直存在）

```
L = [1, 2, 3]
def f(n, lst=[]):
    lst.append(n)
    print(lst)
f(4, L)    #[1, 2, 3, 4]
f(5, L)    #[1, 2, 3, 4, 5]
f(100)     #[100]
f(200)     #[100,200]
```

解决方法

```python
L = [1, 2, 3]
def f(n, lst=None):
    if lst is None:
        lst = []
    lst.append(n)
    print(lst)
```

6、模块和包的导入

模块中以_开头的属性为隐藏属性，在form import *语句导入时，将不被导入

模块中的__all__列表是一个用来存放可导出属性的字符串列表, 当用from import *语句导入时，只导入__all__列表内的属性

包的__init__.py内的__all__列表，用来记录此包有哪些子包或模块在用from 包 import*语句导入时是否被导入，__all__列表只对from import*语句起作用

7、迭代器和生成器

7.1、迭代器

迭代器是指用iter(可迭代对象)函数返回的对象(实例)

迭代器可以用next(it)函数获取的迭代对象的数据

对象想成为迭代器，需要实现__iter__和__next__方法，__iter__中返回对象本身（self），__next__中返回下一个值

迭代器协议：迭代器协议是指对象能够使用next函数获取下一项数据，在没有下一项数据时触发一个StopIterator来终止迭代的约定。实现了方法__iter__的对象是可迭代的，而实现了方法__next__的对象是**迭代器**

```python
class Fibs:
    def __init__(self):
        self.a = 0
        self.b = 1
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        return self.a
    def __iter__(self):
        return self
```

7.2、生成器

zip，enumerate，map都是生成器

使用了 yield 的函数被称为生成器（generator）

生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器, 每次请求值时，都将执行生成器的代码，直到遇到yield或return。yield意味着应生成一个值，而return意味着生成器应停止执行

生成器推导式：squares = (x*x for x in range(5))，类似于列表推导式，将[]变成了()

7.3、迭代器和生成器的关系

迭代器是一个可以记住遍历位置的对象，迭代器从集合的第一个元素开始访问集合，知道所有元素被访问完结束，迭代器往前不会往后退

生成器是特殊的迭代器，只能对生成器进行一次迭代，因为数据是运行时生成的，没有存储起来，只要函数定义内部有yield关键字，该函数就是生成器函数，调用生成器函数会返回一个生成器对象，即生成器函数是生成器工厂

①生成器是生成元素的，迭代器是访问集合元素的一中方式

②迭代输出生成器的内容

③迭代器是一种支持next()操作的对象

④迭代器（iterator）：其中iterator对象表示的是一个数据流，可以把它看做一个有序序列，但我们不能提前知道序列的长度，只有通过nex()函数实现需要计算的下一个数据。可以看做生成器的一个子集。

四、面向对象相关

1、调用父类方法

a、super(子类名，子类obj).方法名(参数)，如super(Child, self).__init__()

b、父类名.方法名(self, 参数)，如：Parent.__init__(self,)

c、super().父类方法名()，如：super().__init__(name) **这里不用传入实例，此方法只能在子类方法中使用**

2、魔术方法

2.1、__str__和__repr__

__str__是展示给人看的字符串

__repr__是展示给机器的表达式字符串，必须是可以代表这个对象的

1. `__repr__` 正式， `__str__` 非正式。
2. `__str__` 主要由 `str()` , `format()` 和 `print()` 三个方法调用。
3. 若定义了 `__repr__` 没有定义 `__str__` ，那么本该由 `__str__` 展示的字符串会由 `__repr__` 代替。
4. `__repr__` 主要用于调试和开发，而 `__str__` 用于为最终用户创建输出。
5. `__repr__` 看起来更像一个有效的 Python 表达式，可用于重新创建具有相同值的对象（给定适当的环境）

```
class mynumber:
    def __init__(self, v):
        self.data = v

    def __repr__(self):
        return 'mynumber(%d)' % self.data

n1 = mynumber(100)
print(eval(str(n1)).data) # 100
```

2.2、with（环境管理器）

1.类内有__enter__和__exit__实例方法的类被称为管理器

2.能够用于with语句进行管理的对象必须是环境管理器

3.__enter__方法在进入with语句时被调用，并返回由as变量管理的对象

4.__exit__将在离开with语句时被调用，且可以用参数来判断在离开with语句时是否有异常发生并作出相应的处理

```
class TmpTest:
    def __init__(self,filename):
        self.filename=filename
    def __enter__(self):
        self.f = open(self.filename, 'r')
        return self.f
    def __exit__(self, exc_type, exc_val, exc_tb):
        self.f.close()

test=TmpTest('file')

with test as t:
    print ('test result: {}'.format(t)
```

3、使用@property

使用@property装饰实例的getter方法，可将此方法转换为实例属性，并可通过setter装饰其setter方法，如果只有property没有setter，则为只读属性

```
class Student(object):

    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value

    @property
    def age(self):
        return 2015 - self._birth
```

4、元类（Meta Class）

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

常见使用场景是ORM框架

5、实现单例模式

```
方式一、
因为python的模块只有在第一次引入时会执行一次，在单例模式实现的模块中实例化一个对象，其余模块直接导入这个对象使用

方式二、
class Singleton:
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, '_instance'):
            _instance = super().__new__(cls, *args, **kwargs)
            cls._instance = _instance
        return cls._instance

class MyClass(Singleton):
    pass

c1 = MyClass()
c2 = MyClass()
print(c1 is c2) # True
```

五、线程与进程

进程：是对运行时的程序的封装，是系统资源调度和分配的基本单位

线程：是进程的子任务，cpu调度和分配的基本单位，实现进程内并发

一个进程可以包含多个线程，线程依赖进程存在，并共享进程内存

1、线程安全

1.1、一个操作可以再多线程环境内安全使用，获取正确的结果

1.2、线程安全的操作好比线程是顺序执行的而不是并发执行的

1.3、一般设计到写操作需要考虑如何让多个线程安全访问数据

2、线程同步的方式

2.1、互斥量（锁）：通过互斥机制放置多个线程同时访问公共资源

2.2、信号量：控制同一时刻多个进程访问同一个资源的线程数

2.3、事件（信号）：通过通知的方式保持多个线程同步

3、进程间通信的方式

3.1、管道/匿名管道/有名管道（pipe）

3.2、信号（Signal）：比如用户使用ctrl+c产生SIGINT程序终止信号

3.3、消息队列（message）

3.4、信号量

3.5、套接字（socket）：最常用的方式，web应用都是这种方式

4、多线程

threading模块

5、多进程

可以用多进程实现cpu密集型程序，避免GIL的影响

multiprocessing模块

Last、其他

1、单下划线和双下划线

单下划线前缀：_var: 用作变量和方法名时，表示是一个受保护的变量或方法，用来指定变量私有，原则上不允许直接访问，但外部类还是可以访问（但在import *时这种不会被导入，除非__all__列表中有此变量）

单下划线后缀：var_: 单末尾下划线也是一个约定 用来避免与python关键字产生命名冲突

双下划线前缀：__var: 以双下划线开头的类的属性和方法，为私有属性和方法，会导致python解释器重写名称，避免子类中的命名冲突：__ClassName__var，但是在该类内部还是可以直接访问，实例对象不能直接访问

双下划线前后缀：__var__：为python类的保留魔术方法

2、内存管理

python采用的是引用计数机制为主，标记-清除和分代收集两种机制为辅的策略

通过**内存池**来减少内存碎片化，提高执行效率。主要通过**引用计数**来完成**垃圾回收**，通过**标记-清除**解决容器对象循环引用造成的问题，通过**分代回收**提高垃圾回收的效率。

2.1、垃圾回收

删除计数为0的对象或者相互引用的一对对象

采用**引用计数**机制为主，**标记-清除**和**分代回收**机制为辅的策略。其中，**标记-清除**机制用来解决计数引用带来的循环引用而无法释放内存的问题，**分代回收**机制是为提升垃圾回收的效率。

手动回收：调用gc模块

```
import gc
del 变量名
gc.collect()
```

自动回收：

1、当内存中有不再使用的部分时，垃圾收集器就会把他们清理掉。它会去检查那些引用计数为0的对象，然后清除其在内存的空间。当然除了引用计数为0的会被清除，还有一种情况也会被垃圾收集器清掉：当两个对象相互引用时，他们本身其他的引用已经为0了。

2、垃圾回收机制还有一个循环垃圾回收器, 确保释放循环引用对象(a引用b, b引用a, 导致其引用计数永远不为0)

2.2、引用计数

Python中有个内部跟踪变量叫做引用计数器，每个变量有多少个引用，简称引用计数。当某个对象的引用计数为0时，就列入了垃圾回收队列

引用计数增加

1.对象被创建：x=4
2.对象被传递：y=x
3.被作为参数传递给函数：foo(x)
4.作为容器对象的一个元素：a=[1,x,'33']

引用计数减少

1.一个本地引用离开了它的作用域。比如上面的foo(x)函数结束时，x指向的对象引用减1。
2.对象的别名被显式的销毁：del x ； 或者del y
3.对象的一个别名被赋值给其他对象：x=789
4.对象从一个窗口对象中移除：myList.remove(x)
5.窗口对象本身被销毁：del myList，或者窗口对象本身离开了作用域

2.3、标记-清除

标记-清除用来解决引用计数机制产生的循环引用，进而导致内存泄漏的问题 。 循环引用只有在容器对象才会产生，比如字典，元组，列表等。

顾名思义，该机制在进行垃圾回收时分成了两步，分别是：

- 标记阶段，从**根**开始遍历所有的对象，如果是可达的（reachable），也就是还有对象引用它，那么就标记该对象为可达
- 清除阶段，再次遍历对象，如果发现某个对象没有标记为可达（即为Unreachable），则就将其回收

2.4、分代回收

简单地认为：对象存在时间越长，越可能不是垃圾，应该越少去收集。这样在执行标记-清除算法时可以有效减小遍历的对象数，从而提高垃圾回收的速度，**是一种以空间换时间的方法策略。**

Python将所有的对象分为年轻代（第0代）、中年代（第1代）、老年代（第2代）三代。所有的新建对象默认是 第0代对象。当在第0代的gc扫描中存活下来的对象将被移至第1代，在第1代的gc扫描中存活下来的对象将被移至第2代。

当某一代中被分配的对象与被释放的对象之差达到某一阈值时，就会触发当前一代的gc扫描。当某一代被扫描时，比它年轻的一代也会被扫描，因此，第2代的gc扫描发生时，第0，1代的gc扫描也会发生，即为全代扫描

2.5、内存池

内存池中登记的小对象，例如数值（[-5, 256]），字符串（不含特殊字符和空白字符），如果变量b赋值给变量a，当a的值发生变化，会重新给a分配空间，a和b的地址不同

```
这些整数对象是提前建立好的，不会被垃圾回收。在一个 Python 的程序中，所有位于这个范围内的整数使用的都是同一个对象.
小整数[-5,256]共用对象，常驻内存
单个字符共用对象，常驻内存
单个单词，不可修改，默认开启intern机制，共用对象，引用计数为0，则销毁
```

3、数据结构的增删查的时间复杂度

3.1 列表

列表采用的是线性表的顺序存储结构，其操作的复杂度如下

操作	时间复杂度
查找L[i]/append/pop last/	O(1)
insert/remove/pop 某个位置/	O(n)

4、GIL（全局解释锁）

在任意时刻只允许一个 Python 线程使用 Python 解释器，即Python 只有一个线程在运行。

对CPU密集型程序有很大影响，无法利用多核

对IO密集型程序影响不大，在线程等待IO结束时，这个线程的GIL锁会释放，其他线程可以运行

使用GIL的原因：

a、Python 的引用计数需要避免资源竞争的问题，我们需要在有两个或多个线程同时增加或减少引用计数的情况下，依然保证引用计数的结果是正确的。

b、保护多线程情况下对python对象的访问

规避GIL影响的方法：

a、CPU密集型可以使用多进程+进程池

b、IO密集型使用多线程/协程

c、cython扩展

为什么有了GIL还需要关注线程安全：

a、一个操作如果是一个字节码指令可以完成就是原子的

b、原子的是可以保证线程安全的

c、使用dis操作分析字节码