Python2和Python3中新式类、经典类(旧式类)的区别

他们最明显的区别在于继承搜索的顺序发生了改变，即经典类多继承搜索顺序(深度优先)：先深入继承树左侧查找，然后再返回，开始查找右侧，新式类多继承搜索顺序(广度优先)：先在水平方向查找，然后再向上查找

deepcopy和copy

deepcopy:

1. 深拷贝做递归拷贝，可以递归拷贝所有的内部嵌套数据
2. 深拷贝递归拷贝遇到可变类型则创建新的内存空间（不会互相干扰）
3. 深拷贝递归拷贝遇到不可变数据类型则拷贝引用；

递归拷贝遇到可变对象时，就整体分配新id（创建新的内存空间）；

创建完全独立复制对象，原对象为不可变对象时，id相同（拷贝引用）；

copy:

1.浅拷贝只做最顶层的数据类型判断；

2.如果顶层是可变类型则创建新的内存空间，

3.如果顶层是不可变数据类型就拷贝引用，

4.不管顶层为可变元素还是不可变元素，内层全部做引用拷贝，若内层有可变元素，则修改会互相干扰；

浅复制对象为不可变对象时，id相同，原对象改变不会影响新对象；

浅复制对象为可变对象，改变原对象复杂子对象时，新对象会受影响，其他情况（原对象无复杂子对象或改变原对象非复杂子对象）新对象不会受影响;id不同

=赋值：无论可变还是不可变对象，均拷贝引用，不会分配新的对象；

不会产生独立的对象存在，只是在原有数据块打上一个新标签，当一个标签被改变时，数据块发生变化，另一个标签所指内容会随之改变；

pyinstaller 包:

将py文件变为可执行文件.exe

```
pyinstall -F document.py #将py文件转化为可执行文件（不加 -F 默认-D ,生成可执行文件夹）
```

实例方法、静态方法、类方法

实例方法：最为常见，第一个

参数为self,将实例传入方法

类方法：

定义：用@classmethod装饰器定义，第一个参数不再是传入的实例，而是传入调用者的最底层类，即可通过类调用也可通过实例调用；（cls：子类若继承基类，实例化时基类中cls表示子类，）

使用场景：方法不需访问实例成员，需访问基类或派生类成员，比如统计继承树上每个类实例化次数；

静态方法：用@staticmethod定义，该方法不需传入额外的参数，可通过实例化过非实例化调用;

主要区别：是否与类或者实例进行绑定

抽象基类声明

```
class Tombola(abc.ABC): # python3.4后
```

```
class Tombola(metaclass=abc.ABCMeta): # python3.4以前
```

```
class Tombola(object): #这是Python 2! ! !
__metaclass__ = abc.ABCMeta
```

抽象基类无法被实例化

抽象方法 @abstractmethod

使用此装饰器要求类的元类是 ABCMeta 或是从该类派生，对于抽象基类里的抽象方法，子类实现了该抽象方法后才能被实例化；

@abstractclassmethod、@abstractstaticmethod 和 @abstractproperty 器三个装饰器从 Python 3.3 起废弃了，因为装饰器可以在 @abstractmethod上堆叠，这三个就显得多余了。

声明抽象类方法的推荐方式：

```
class MyABC(abc.ABC):
@classmethod
@abc.abstractmethod
def an_abstract_classmethod(cls, ...):
pass
```

接口：若干抽象方法的集合

多进程和多线程 协程

多线程：由于GIL锁，同一时刻只能有一个线程在运行，遇到阻塞时切换下一个线程，适用于IO密集型（文件处理，网络爬虫），如果是纯计算的程序，没有 I/O 操作，解释器会每隔 100 次操作就释放这把锁，让别的线程有机会执行（这个次数可以通过sys.setcheckinterval 来调整）；

Python多线程相当于单核多线程，多线程有两个好处：CPU并行，IO并行，单核多线程相当于自断一臂。所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那可通过C扩展来实现，不过这样就失去了Python简单易用的特点。

如何实现让多线程使用多核资源：

通过C扩展：可以把一些计算密集型任务用C语言编写，然后把.so链接库内容加载到Python中，因为执行C代码，GIL锁会释放，这样一来，就可以做到每个核都跑一个线程的目的；Python 还有另一种与 C 模块进行互通的机制 : ctypes,ctypes 会在调用 C 函数前释放 GIL。所以，我们可以通过 ctypes 和 C 动态库来让 python 充分利用物理内核的计算能力。

多进程：可充分利用多核cpu资源，适用于CPU密集型（循环处理，计数等），CPU密集型对多线程不友好，因为频繁切换消耗资源；

python-parallelize包实现多进程方式处理for循环；

协程：协程（coroutine），又称微线程。协程不是线程也不是进程，它的上下文关系切换不是由CPU控制，一个协程由当前任务切换到其他任务由当前任务来控制。一个线程可以包含多个协程，对于CPU而言，不存在协程这个概念，它是一种轻量级用户态线程（即只针对用户而言）。协程拥有自己的寄存器上下文和栈，协程调度切换到其他协程时，将寄存器上下文和栈保存，在切回到当前协程的时候，恢复先前保存的寄存器上下文和栈。

第三方库**gevent；asyncio**提供的框架以事件循环(event loop)为中心；

进程和线程区别：

联系：一个进程包括多个线程；

1.根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位

2.资源分配：系统在运行的时候会为每个进程分配不同的内存空间；而对线程而言，除了CPU外，系统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源；

3.效率方面：由于每个进程都有独立的代码和数据空间（程序上下文），进程之间切换开销大，效率低一些；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小，效率相对高一些；

4.在执行过程。每个独立的进程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

4.应用场景：多进程适合计算密集型，多线程适合IO密集型，

协程相对于线程优势：

1.数据安全上：协作式多任务，自主将控制权让步给中央调度程序，多线程为抢占式多任务，必须保留锁，防止数据处于无效保护状态；任意时刻只有一个协程在运行，无需保留锁，自身便会同步；

2.资源占用上：线程需要占用大量内存，每个正在执行的线程大约占用8MB内存，如果成千上万千个，计算机无法接受，启动生成器协程的开销和调用函数的开销相仿，处于活跃状态的协程在其耗尽之前，只会占用大约1KB的内存；

3.效率上：协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快，而线程的切换涉及到锁的操作，较慢。

.应用场景上：协程只能使用单核，放弃了多核使用的优势，需权衡使用；

线程同步与互斥：

多线程对同一资源访问修改时，若不进行控制，会出现不可预期的结果，这种线性叫做"线程不安全",因此需要对线程进行同步控制，需引入互斥机制（互斥锁、信号量、线程池）；

线程同步互斥的4种方式（临界区 ）

1．临界区（仅允许一个进程进入）：每个进程中访问临界资源的那段程序称为临界区（临界资源是一次仅允许一个进程使用的共享资源）。每次只准许一个进程进入临界区，进入后不允许其他进程进入。 适合一个进程内的多线程访问公共区域或代码段时使用；

2．互斥量 (LOCK临界资源锁定)：以牺牲性能换取代码的安全性，适合不同进程内多线程访问公共区域或代码段时使用，与临界区相似。（加锁）

条件变量：条件变量是线程可用的另一种同步机制，即等待某一条件的发生，和信号相似。它是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号），唤起挂起的线程。为了防止竞争（保护条件变量），条件变量的使用总是和一个互斥锁结合在一起。条件变量就是一种通知模型的同步方式，大大的节省了CPU的计算资源，减少了线程之间的竞争，而且提高了线程之间的系统工作的效

率。简单的来说条件变量就是某一线程必须得在某一条件发生后其锁住互斥量才可以令事件发生，否则条件没发生锁住互斥量还是无法完成自己的操作

总的来说互斥锁是线程间互斥的机制，条件变量则是同步机制。

3. 信号量（线程同步，告知可用）：与临界区和互斥量不同，可以实现多个进程同时访问公共区域数据，原理与操作系统中PV操作类似，先设置一个访问公共区域的线程最大连接数，每有一个线程访问共享区资源数就减一，直到资源数小于等于零。互斥量用于线程的互斥，信号量用于线程的同步

3. 事件（Event）：通过线程间触发事件移交控制权实现同步互斥(协程)

互斥量与临界区的主要区别：

1、 临界区只能用于对象在同一进程里线程间的互斥访问；互斥体可以用于对象进程间或线程间的互斥访问；

2、 临界区是非内核对象，只在用户态进行锁操作，速度快；

3、 互斥体是内核对象，在核心态进行锁操作，速度慢。

4、 临界区和互斥体在Windows平台都下可用；Linux下只有互斥体可用。

“鸭子类型”：

对象的类型无关紧要，只要实现了特定的协议即可。

传参可迭代参数

```
a=[1,2,3]
class A():
 def __init__(self,a):
 self.a=a #不推荐直接引用参数，会修改原有对象
 self.a=a[:] #创建副本 推荐
 self.a=list(a)# 创建副本 推荐
 self.ap(self.a)
 def ap(self,m):
 m.append('a')
```

虚拟子类

即便不继承，也有办法把一个类注册为抽象基类的虚拟子类。这样做时，我们保证注册的类忠实地实现了抽象基类；定义的接口注册虚拟子类的方式是在抽象基类上调用 register 方法；虚拟子类的好处是你实现的第三方子类不需要直接继承自基类但是仍然可以声称自己子类中的方法实现了基类规定的接口（issubclass(), issubinstance()）,不需要实现抽象方法，

```
from abc import ABC, abstractmethod


class Shape(ABC):
 @abstractmethod
 def getArea(self):
 pass
```

```
8
9  @Shape.register # 注册虚拟子类
10  class House():
11   def __init__(self, area):
12   self.area = area
13
14   def showArea(self):
15   return self.area
16
17  # Shape.register(House) # 注册虚拟子类
18
19  house = House(100) # 重新定义变量
20  print(house.showArea()) # 100
21  print(issubclass(House, Shape)) # True
22  print(isinstance(house, Shape)) # True
```

子类化内置类型

直接子类化内置类型（如 dict、list 或 str）容易出错，因为内置类型的
方法通常会忽略用户覆盖的方法。不要子类化内置类型，用户自己定义的类应该继承
collections 模块中的类，例如UserDict、UserList 和 UserString

enumerate将迭代器包装成生成器

```
1  a=[1,2,3,4,5]
2  for i ,num in enumerate(a):
3   # enumerate将迭代器包装成生成器，尽量不要用range（len(a)）写法
4   print(i,num)
```

判断对象类型

```
1  from collections import abc
2  isinstance(obj, abc.MutableSequence) # 必定是列表
3  isinstance(obj, abc.Mapping): #字典
```

作用域

引用变量时，python解释器按照如下顺序遍历各作用域
1.当前函数作用域；
2.任何外围函数作用域（例如包含当前函数的其他函数）
2.包含当前代码模块作用域（全局作用域）
4.内置作用域（包含len、str等函数的作用域）

异常处理不要返回None

```
1  def divide(a,b):
2   try:
3   return True,a/b
4   except ZeroDivisionError:
```

```
 return False,None
success,result = divide(2,5)
```

bytes str unicode

python3 有两种表示字符序列的类型：bytes和str，前者实例包含原始的8位值（二进制），后者的实例包含Unicode字符，encode方法把Unicode字符转换为二进制数据，decode方法把二进制数据转换为Unicode字符；

以@classmethod多态的构建对象

第一个参数为cls代表类本身

使用@property 装饰器进行属性重构

将一个直接访问的属性转变为函数触发式属性,可实现数值验证或额外的实时运算

缺点：无法复用移植到其他类使用，可通过描述符改写(推荐),或者mix-in类（混合类）

```
class Person:
 def __init__(self, name):
 print('init')
 self.name = name # 调用setter
 @property #getter方法
 def name(self):
 print('getter')
 return self._name
 @name.setter
  def name(self,name):
  print('setter')
  if name<10:
  self._name=name*10
  else:
  self._name=name/10
P=Person(1000) # 调用setter
print(P.name) #调用getter
>>init >>setter >>getter >>100.0
```

with语句的工作原理：

目的在于从流程图中把 try,except 和finally 关键字和

资源分配释放相关代码统统去掉，简化try....except....finlally的处理流程。

with通过__enter__方法初始化，然后在__exit__中做善后以及处理异常。

紧跟with后面的语句会被求值，返回对象的__enter__()方法被调用，这个方法的返回值将被赋值给as关键字后面的变量，当with后面的代码块全部被执行完之后，将调用前面返回对象的__exit__()方法。

with语句最关键的地方在于被求值对象必须有__enter__()和__exit__()这两个方法，那我们就可以通过自己实现这两方法来自定义with语句处理异常。

结合正则的split方法自定义分隔符

```
a='1as2da234sds4'
```

```
print([i for i in re.compile(r'[^0-9]+').split(a)])
>> ['1', '2', '234', '4']
```

python的动态性

1. 动态语言是在运行时确定数据类型的语言。 变量使用之前不需要类型声明;

2.可动态添加方法或属性;

python的解释性

计算机是不能够识别高级语言的，所以当我们运行一个高级语言程序的时候，就需要一个“翻译机”来从事把高级语言转变成计算机能读懂的机器语言的过程。这个过程分成两类，第一种是编译，第二种是解释。解释型语言就没有这个编译过程，而是在程序运行的时候，通过解释器对程序逐行做出解释，然后直接运行;

Python是一门先编译后解释的语言；PyCodeObject则是Python编译器真正编译成的结果;

看下Import模块的源码其实不难发现，它在写入pyc文件的时候，写了一个Long型变量，变量的内容则是文件的最近修改日期，同理，在pyc文件中，每次在载入之前都会检查一下py文件和pyc文件保存的最后修改日期，如果不一致则重新生成新的pyc文件;

import机制

使用:

1. 导入模块:

模块除了内建模块（可以用过dir(__builtins__)查看有哪些内建函数），就是非内建模块，这一部分模块就需要用import导入；该过程把module_name.py文件的全部内容加载到内存并赋值给与模块同名的变量，这个变量的类型是'module';

对于python来说，所有被加载到内存的模块都是放在sys.modules里面，所以执行import时会首先去该列表中查询是否已添加，如果未添加则进行查找，默认先在当前目录下查找，然后再在系统中查找。系统查找的范围是：sys.path下的所有路径，按顺序查找。

2.导入包:

"import package_name"导入包的本质就是执行该包下的__init__.py文件，在执行文件后，会在"package_name"目录下生成一个"__pycache__ / __init__.cpython-35.pyc" 文件。

闭包：内部函数引用外部非全局变量，这个函数叫做闭包;

(1)要有函数的嵌套（要有外部函数，内部函数）
(2)内部函数要使用到外部函数的变量;
(3)外部函数必须有返回值，返回内部函数名

有效59个方法:

40条，41条，58条

https://blog.csdn.net/zzyzgg/article/details/91957055