python杂七杂八

- __all__
  - __all__是一个字符串list，用来定义模块中对于from XXX import *时要对外导出的符号，即要暴露的借口.
  - 但它只对import *和 from XXX import *起作用，对from XXX import XXX不起作用。
- __init__.py
  - __init__.py 在包被导入时会首先被执行。
  - __init__.py 的工作路径与导入该包的程序的工作路径一致。
- python反射机制（自省）
  - 反射机制
    - 通过字符串的形式，导入模块。
    - 通过字符串的形式，去模块寻找指定函数，并执行。
    - 利用字符串的形式去对象（模块）中操作（查找/获取/删除/添加）成员，一种基于字符串的事件驱动!
  - 反射机制核心函数
    - hasattr(obj,'name')
      - 判断对象中是否含有字符串形式的方法名或属性名，返回True、False。
    - getattr(obj,'name',None)
      - 返回对象中的方法或属性： obj.name，如果没有此方法或属性，返回None。
    - setattr(obj,'name',value)
      - 设置对象中方法或属性的值： obj.name = value。
    - delattr(obj,'name')
      - 删除对象中的方法或属性。
- python构建单例模式
  - 使用模块
    - Python 的模块就是天然的单例模式：因为模块在第一次导入时，会生成 .pyc 文件，当第二次导入时，就会直接加载 .pyc 文件，而不会再次执行模块代码。

```
# mysingleton.py
class Singleton(object):
    def foo(self):
        pass
singleton = Singleton()

# 外面文件导入文件中对象
from a import singleton
```

  - 使用装饰器

- 使用类
- 基于__new__方法实现

```
import threading

class Singleton(object):
    _instance_lock = threading.Lock()

    def __init__(self):
        pass

    def __new__(cls, *args, **kwargs):
        if not hasattr(Singleton, "_instance"):
            with Singleton._instance_lock:
                if not hasattr(Singleton, "_instance"):
                    Singleton._instance = object.__new__(cls)
        return Singleton._instance

obj1 = Singleton()
obj2 = Singleton()
print(obj1,obj2)
```

- 基于metaclass方式实现

1. 类由type创建，创建类时，type的__init__方法自动执行，类() 执行type的 __call__方法(类的__new__方法,类的__init__方法)
2. 对象由类创建，创建对象时，类的__init__方法自动执行，对象()执行类的 __call__ 方法

```
import threading

class SingletonType(type):
    _instance_lock = threading.Lock()
    def __call__(cls, *args, **kwargs):
        if not hasattr(cls, "_instance"):
            with SingletonType._instance_lock:
                if not hasattr(cls, "_instance"):
                    cls._instance =
super(SingletonType,cls).__call__(*args, **kwargs)
        return cls._instance

class Foo(metaclass=SingletonType):
    def __init__(self,name):
        self.name = name


obj1 = Foo('name')
```

```
obj2 = Foo('name')
print(obj1,obj2)
```

- 钩子函数
  - 钩子函数，顾名思义，就是把我们自己实现的hook函数在某一时刻挂接到目标挂载点上。
    - hook函数，就是我们自己实现的函数，函数类型与挂载点匹配（返回值，参数列表）。
    - 挂接，也就是hook或者叫注册（register）,使得hook函数对目标可用。
    - 目标挂载点，也就是挂我们hook函数的地方。
  - 钩子方法就是通过子类的行为去反向控制父类的行为的一种方法
    - 在模板方法模式中，由于面向对象的多态性，子类对象在运行时将覆盖父类对象，子类中定义的方法也将覆盖父类中定义的方法，因此程序在运行时，具体子类的基本方法将覆盖父类中定义的基本方法，子类的钩子方法也将覆盖父类的钩子方法，从而可以通过在子类中实现的钩子方法对父类方法的执行进行约束，实现子类对父类行为的反向控制。

- @property
  - 用@property装饰器来创建只读属性，@property装饰器会将方法（函数）转换为相同名称的只读属性,可以与所定义的属性配合使用，这样可以防止属性被修改。
    - 修饰方法，是方法可以像属性一样访问。
      - 加了@property后，可以用调用属性的形式来调用方法（函数）,后面不需要加括号（）。
    - 与所定义的属性配合使用，这样可以防止属性被修改。

- popen和system
  - popen本身是不阻塞的，要通过标准io的读取（fread等）使它阻塞。
    - popen相当于是先创建一个管道，然后fork，关闭管道的一端，执行exec，返回一个标准的io文件指针。
  - system本身就是阻塞的
    - system相当于是先后调用了fork， exec，waitpid来执行外部命令。

- python进程池

```
from multiprocess import Pool

process_pool=Pool(10)

process_pool.apply(func,args=(...))

process_pool.apply_async(func,args=(...))

process_pool.map(func,iterable)
```

  - apply的结果就是func的返回值,**同步**提交，需等待上一个进程运行完，才能执行下一个进程。
  - apply_async的结果也是func的返回值，但是**异步**提交，无需等待上一个进程运行完，直接执行下一个进程。

        - 多进程执行完，需要在代码中加入`process_pool.close()`和`process_pool.join()`，否则不会执行被调用函数。
    - map
        - 此方法将iterable切换为多个块，并将其作为单独的任务提交给进程池。因此，可以利用池中的所有进程。
        - 结果以与参数顺序相对应的顺序返回。
    - map_async
        - 其子参数任务并不是独立的，如果其中的某个子参数任务抛出异常，同时也会导致其他的子参数任务停止。
        - 结果以与参数顺序相对应的顺序返回。

	Multi-args	Concurrence	Blocking	Ordered-results
map	no	yes	yes	yes
apply	yes	no	yes	no
map_async	no	yes	no	yes
apply_async	yes	yes	no	no

- struct模块
    - struct 模块用于 Python 值和用 Python 字节对象表示的 C 结构体之间的转换。
        - 给 C 结构打包时一般包含了填充字节（pad bytes），在打/拆包时需考虑对齐的问题。
        - struct 模块使用 Format Strings 作为 C 结构布局的简洁描述以及与 Python 值的预期转换。
    - pack(fmt, v1, v2, ...)
        - 按照给定的格式(fmt)，把数据v1, v2, ...封装成字符串(实际上是类似于c结构体的字节流)。
    - unpack(fmt, string)
        - 按照给定的格式(fmt)解析字节流string，返回解析出来的tuple。
    - calcsize(fmt)
        - 计算给定的格式(fmt)需要占用多少字节的内存。

Character	Byte order	Size and alignment
@	native	native 凑够4个字节
=	native	standard 按原字节数
<	little-endian	standard 按原字节数
>	big-endian	standard 按原字节数
!	network (= big-endian)	standard 按原字节数

Format	C Type	Python	字节数
x	pad byte	no value	1
c	charf	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1

Format	C Type	Python	字节数
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer or long	4
l	long	integer	4
L	unsigned long	long	4
q	long long	long	8
Q	unsigned long long	long	8
f	float	float	4
d	double	float	8
s	char[]	string	1
p	char[]	string	1
P	void *	long	

- partial
    - partial是偏函数
        - 和装饰器一样，它可以扩展函数的功能。
            - 接收一个函数，返回一个固定该函数某些参数值的新函数。
- concurrent
    - 并发模块
    - concurrent.futures: 启动并行任务
        - 可由 ThreadPoolExecutor 使用线程实现
        - 由 ProcessPoolExecutor 使用单独的进程来实现
        - submit
            - 调度可调用对象 fn，以 fn(*args **kwargs) 方式执行并返回 Future 对象代表可调用对象的执行。
        - result(timeout=None)
            - 返回被调用函数返回的值。
        - concurrent.futures.as_completed(fs, timeout=None)
            - 返回一个包含 fs 所指定的 Future 实例的迭代器，这些实例会在完成时生成 future 对象（包括正常结束或被取消的 future 对象）。
            - 任何由 fs 所指定的重复 future 对象将只被返回一次。
            - 任何在 as_completed() 被调用之前完成的 future 对象将优先被生成。
- namedtuple(具名元组)

    - 具名元组的实例和普通元组消耗的内存一样多，因为字段名都被存在对应的类里面。
        - 这个类跟普通的对象实例比起来也要小一些，因为 Python 不会用 __dict__ 来存放这些实例的属性。

```
  collections.namedtuple(typename, field_names, verbose=False,
rename=False)
  # typename：元组名称
  # field_names：元组中元素的名称
  # rename：如果元素名称中含有 python 的关键字，则必须设置为
rename=True
  # verbose
```

- json.dumps(), json.loads()和json.dump(), json.load()的区别
    - dumps是将dict转化成str格式，loads是将str转化成dict格式。
    - dump和load也是类似的功能，只是与文件操作结合起来了。
        - dump需要一个类似于文件指针的参数，以将dict转成str然后存入文件中。而dumps只是将dict转成str。

- 多进程和多线程
    - IO密集型任务 VS 计算密集型任务
        - 计算密集型任务，是指CPU计算占主要的任务，CPU一直处于满负荷状态。
        - IO密集型任务，是指磁盘IO、网络IO占主要的任务。
    - 由于GIL的存在，Python中的多线程适合IO密集型任务，而不适合计算密集型任务。
        - 在多线程的环境中，python虚拟机按一下 方式执行：
            1. 设置GIL(global interpreter lock)
            2. 切换到一个线程执行
            3. 运行：指定数量的字节码指令、线程主动让出控制（可以调用time.sleep(0)）（例如，等待IO）
            4. 把线程设置为睡眠状态
            5. 解锁GIL
            6. 再次重复以上步骤。
    - Python中的多进程适合计算密集型任务，可以非常有效的使用CPU资源。

- Python利用CPU多核
    - 对所有面向I/O的（会调用内建的操作系统C代码的）程序来说，GIL会在这个I/O调用之前被释放，以允许其他线程在这个线程等待I/O的时候获得GIL锁而运行。
        - 我们可以把一些 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，充分地利用CPU每个核。
    - 使用多进程。

- python multiprocessing启动进程的方法
    - spawn

        - 可在Unix和Windows上使用。 Windows上的默认设置。
        - 父进程会启动一个全新的 python 解释器进程。 子进程将只继承那些运行进程对象的 run() 方法所必需的资源。特别地，来自父进程的非必需文件描述符和句柄将不会被继承。
        - 使用此方法启动进程相比使用 fork 或 forkserver 要慢上许多。
    - fork
        - 只存在于Unix。Unix中的默认值。
        - 父进程使用 os.fork() 来产生 Python 解释器分叉。子进程在开始时实际上与父进程相同。父进程的所有资源都由子进程继承。
    - forkserver
        - 可在Unix平台上使用，支持通过Unix管道传递文件描述符。
        - 程序启动并选择* forkserver * 启动方法时，将启动服务器进程。从那时起，每当需要一个新进程时，父进程就会连接到服务器并请求它分叉一个新进程。分叉服务器进程是单线程的，因此使用 os.fork() 是安全的。没有不必要的资源被继承。

- pypy比Cpython性能好
    - 使用JIT(Just In Time，即时编译)技术。
        - 首先让代码解释执行，同时收集信息，在收集到足够信息的时候，将代码动态编译成CPU指令，然后用CPU指令替代解释执行的过程。
        - 优点
            - 提升效率。
            - 编译器可以获得静态编译期所没有的信息。例如知道哪些函数是被大量使用的，可以让编译器针对性优化这部分代码。

- +,+=,extend的区别
    - 调用 += 运算的时候就是调用__iadd__函数，这个函数内部调用extend()方法。
    - extend()方法内部使用for循环来append()元素，接收一个可迭代序列。

- 动态加载模块
    - 使用importlib.import_module()在代码中动态加噪模块。

- bisect(数组二分查找算法)
    - 模块对有序列表提供了支持，使得他们可以在插入新数据仍然保持有序。
    - bisect_left(a, x, lo=0, hi=len(a))
        - 在有序列表 a 中找到 x 合适的插入点以维持有序。
        - 参数 lo 和 hi 可以被用于确定需要考虑的子集；默认情况下整个列表都会被使用。
        - 如果 x 已经在 a 里存在，那么插入点会在已存在元素之前（也就是左边）。
    - bisect_right(a, x, lo=0, hi=len(a))
        - 类似于bisect_left，但是返回的插入点是 a 中已存在元素 x 的右侧。
    - bisect(a, x, lo=0, hi=len(a))
        - 与bisect_right一致。

- shutil
    - shutil.copy2()与shutil.copy()的区别
        - shutil.copy2()与shutil.copy()都将文件src拷贝到文件或者目录dst，但是 shutil.copy2() 还会尝试保留文件的元数据。

        - coshutil.copy2() 会使用 copystat() 来拷贝文件元数据。
    - shutil.copytree
        - 将以 src 为根起点的整个目录树拷贝到名为 dst 的目录并返回目标目录。
            - 目录的权限和时间会通过 copystat() 来拷贝，单个文件则会使用 copy2() 来拷贝。
    - shutil.copystat
        - 从 src 拷贝权限位、最近访问时间、最近修改时间以及旗标到 dst。
        - 在 Linux上，copystat() 还会在可能的情况下拷贝“扩展属性”。

- python re.sub只替换一部分内容
    - (re)
        - 对正则表达式分组并记住匹配的文本
    - \1\2...
        - 匹配第一个第二个分组...的内容。
    - 将everything is alright.修改为everybody is alright.

```
import re

src = 'everything is alright.'

dst=re.sub('(every).*?(\\s.*\\.)',r'\1body\2',src)  # dst='everybody is
alright.'

# \1 对应着 (every) 的匹配内容，即 every
# \2 对应着 (\\s.*\\.) 的匹配内容，即  is alright.
# 将 .*? 替换为 body ， .*?的匹配内容是thing
```

- numpy中flat/flatten用法区别
    - ndarray.flat
        - 将数组转换为1-D的迭代器,flat返回的是一个迭代器，可以用for访问数组每一个元素。
    - ndarray.flatten(order='C')
        - 将数组的副本转换为一个维度，并返回
        - 可选参数，order：{'C','F','A','K'}
            - 'C'：C-style，行序优先
            - 'F'：Fortran-style，列序优先
            - 'A'：if a is Fortran contiguous in memory ,flatten in column_major order
            - 'K'：按照元素在内存出现的顺序进行排序

- pybind11