1、python里面的内置函数

callable(object)

Return True if the object argument appears callable, False if not. If this returns true, it is still possible that a call fails, but if it is false, calling object will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a \_\_call\_\_() method.

如果object是可调用的，那么返回true，如果不是，则返回false.如果返回是true,那么调用有可能失败。如果返回false，那么调用object一定不会成功。注意，类class是调用的（调用一个类，返回一个实例），如果类里面实现了\_\_call\_\_()方法，那么实例是可调用的

**说明：**

1. 该方法用来检测对象是否可被调用，可被调用指的是对象能否使用()括号的方法调用。使用函数式调用法

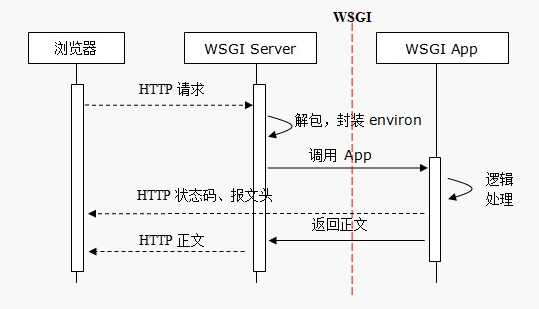
2. 可调用对象，在实际调用也可能调用失败；但是不可调用对象，调用肯定不成功。

3. 类对象都是可被调用对象，类的实例对象是否可调用对象，取决于类是否定义了\_\_call\_\_方法。

**2、Django与WSGI**

根据Python的惯例，Django不是一个完整的Web后端框架，它只负责开发WSGI应用程序 ，在生产环境中Django应用应当与一个WSGI服务器配套，由WSGI服务器负责网络通讯部分。

WSGI，全称 *Web Server Gateway Interface*，或者 Python Web Server Gateway Interface ，是为 Python 语言定义的Web*服务器*和Web*应用程序*之间的一种简单而通用的接口



WSGI将Web服务分成两个部分：服务器和应用程序。WGSI服务器只负责与网络相关的两件事：接收浏览器的 HTTP请求、向浏览器发送HTTP应答；而对HTTP请求的具体处理逻辑，则通过调用WSGI应用程序进行。

实现一个WSGI应用，只需要满足3个要求：

1. 是可调用的，比如是一个函数，或者是一个可调用类（具有\_\_call\_\_方法）的实例
2. WSGI应用应当返回一个可迭代（iterable）的值，比如字符串列表
3. WSGI应用在返回之前，应当调用WSGI服务器传入的start\_response函数发送状态码和HTTP报文头（start\_response函数，这个函数是wsgi服务给定的，也是wsgi服务器端实现的，类似与信号里面的回调函数，接受到这个信息，就调用这个函数的功能）

**最小WSGI应用**

一种最简单的满足WSGI规约的应用程序需要实现一个指定形式的函数：

from wsgiref.simple\_server import make\_server

def wsgi\_app(environ,start\_response):

start\_response('200 OK',[('Context-Type','text/plain')])

return 'such a tiny wsgi app!'

httpd = make\_server('0.0.0.0',80,wsgi\_app) ----把这个wsgi\_app函数注册给wsgi服务器，从而使wsgi服务器知道调用哪个应用处理程序

httpd.serve\_forever()

environ是一个包含全部HTTP请求信息的字典/Dict，由WSGI服务器解包HTTP请求生成

==================

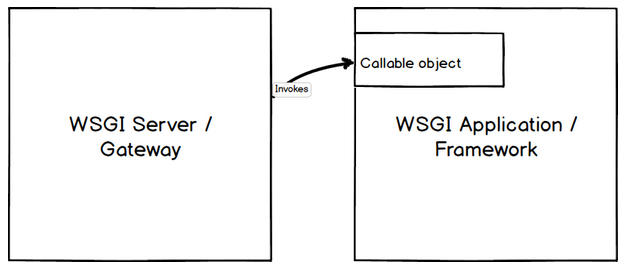
实际生产中，python 程序是放在服务器的 http server（比如 apache， nginx 等）上的。现在的问题是 **服务器程序怎么把接受到的请求传递给 python 呢，怎么在网络的数据流和 python 的结构体之间转换呢？**这就是 wsgi 做的事情：一套关于程序端和服务器端的规范，或者说统一的接口。

面向 http 的 python 程序需要关心哪些内容：

* 请求
  + **请求的方法 method**
  + **请求的地址 url**
  + **请求的内容**
  + 请求的头部 header
  + 请求的环境信息
* 响应
  + **状态码 status\_code**
  + **响应的数据**
  + 响应的头部

WSGI（Web Server Gateway Interface） 的任务就是把上面的数据在 http server 和 python 程序之间简单友好地传递。它是一个标准，被定义在[PEP 333](http://legacy.python.org/dev/peps/pep-0333/#rationale-and-goals)。需要 http server 和 python 程序都要遵守一定的规范，实现这个标准的约定内容，才能正常工作

invoke 乞灵，祈求; 提出或授引…以支持或证明; 召鬼; 借助



**应用程序端**

WSGI 规定每个 python 程序（Application）必须是一个可调用的对象（实现了\_\_call\_\_ 函数的方法或者类），接受两个参数 environ（WSGI 的环境信息） 和 start\_response（开始响应请求的函数），并且返回 iterable（可迭代值）。几点说明：

1. environ 和 start\_response 由 http server 提供并实现
2. environ 变量是包含了环境信息的字典
3. Application 内部在返回前调用 start\_response
4. start\_response也是一个 callable，接受两个必须的参数，status（HTTP状态）和 response\_headers（响应消息的头）
5. 可调用对象要返回一个值，这个值是可迭代的。

代码示例:

**服务器程序端**

标准要能够确切地实行，必须要求程序端和服务器端共同遵守。上面提到， envrion 和 start\_response 都是服务器端提供的。下面就看看，服务器端要履行的义务

* 准备 environ 参数
* 定义 start\_response 函数
* 调用程序端的可调用对象

PEP 333 (WSGI协议标准)里给出了一个 wsgi server 的简单实现

import os, sys

def run\_with\_cgi(application):    # application 是程序端的可调用对象

# 准备 environ 参数，这是一个字典，里面的内容是一次 HTTP 请求的环境变量

    environ = dict(os.environ.items())

    environ['wsgi.input']        = sys.stdin

    environ['wsgi.errors']       = sys.stderr

    environ['wsgi.version']      = (1, 0)

    environ['wsgi.multithread']  = False

    environ['wsgi.multiprocess'] = True

    environ['wsgi.run\_once']     = True

    environ['wsgi.url\_scheme'] = 'http'

    headers\_set = []

    headers\_sent = []

# 把应答的结果输出到终端

    def write(data):

        sys.stdout.write(data)

        sys.stdout.flush()

# 实现 start\_response 函数，根据程序端传过来的 status 和 response\_headers 参数，

# 设置状态和头部

    def start\_response(status, response\_headers, exc\_info=None):

        headers\_set[:] = [status, response\_headers]

       return write

# 调用客户端的可调用对象，把准备好的参数传递过去

    result = application(environ, start\_response)

    # 处理得到的结果，这里简单地把结果输出到标准输出。

    try:

        for data in result:

            if data:    # don't send headers until body appears

                write(data)

    finally:

        if hasattr(result, 'close'):

            result.close()

**中间层 middleware（在openstack里是路由的概念）**

有些程序可能处于服务器端和程序端两者之间：对于服务器程序，它就是应用程序；而对于应用程序，它就是服务器程序。这就是中间层 middleware。middleware 对服务器程序和应用是透明的，它像一个代理/管道一样，把接收到的请求进行一些处理，然后往后传递，一直传递到客户端程序，最后把程序的客户端处理的结果再返回

middleware 做了两件事情：

1. 被服务器程序（有可能是其他 middleware）调用，返回结果回去
2. 调用应用程序（有可能是其他 middleware），把参数传递过去

PEP 333 上面给出了 middleware 的可能使用场景：

* 根据 url 把请求给到不同的客户端程序（url routing）---这个在openstack里面经常用
* 允许多个客户端程序/web 框架同时运行，就是把接到的同一个请求传递给多个程序。
* 负载均衡和远程处理：把请求在网络上传输
* 应答的过滤处理

一个简单地 url routing 的 middleware示例：

class Router(object):

    def \_\_init\_\_(self):

        self.path\_info = {}

    def route(self, environ, start\_response):

        application = self.path\_info[environ['PATH\_INFO']]

        return application(environ, start\_response)

    def \_\_call\_\_(self, path):

        def wrapper(application):

            self.path\_info[path] = application

        return wrapper

router = Router()

在程序中的应用

#here is the application

@router('/hello') #调用 route 实例，把函数注册到 paht\_info 字典

def hello(environ, start\_response):

    status = '200 OK'

    output = 'Hello'

    response\_headers = [('Content-type', 'text/plain'),

                        ('Content-Length', str(len(output)))]

    write = start\_response(status, response\_headers)

    return [output]

@router('/world')

def world(environ, start\_response):

    status = '200 OK'

    output = 'World!'

    response\_headers = [('Content-type', 'text/plain'),

                        ('Content-Length', str(len(output)))]

    write = start\_response(status, response\_headers)

    return [output]

#here run the application

result = router.route(environ, start\_response)

for value in result:

    write(value)

=============================

所有的函数都是可调用对象

一个类实例也可以变成一个可调用对象，只需要类里面实现一个特殊方法\_\_call\_\_()。

Python中有一个有趣的语法，只要定义类型的时候，实现\_\_call\_\_函数，这个类型就成为可调用的。

换句话说，我们可以把这个类型的对象当作函数来使用，相当于 重载了括号运算符。

\_\_call()\_\_

Python中的函数是一级对象。这意味着Python中函数的引用可以作为输入传递到其他的函数/方法中，并在其中被执行。   
而Python中类的实例（对象）可以被当做函数对待。也就是说，我们可以将它们作为输入传递到其他的函数/方法中并调用他们，正如我们调用一个正常的函数那样。而类中\_\_call()\_\_函数的意义正在于此。为了将一个类实例当做函数调用，我们需要在类中实现\_\_call()\_\_方法。也就是我们要在类中实现如下方法：def \_\_call\_\_(self, [args...])。这个方法接受一定数量的变量作为输入。   
假设x是X类的一个实例。那么调用x.\_\_call(1,2)\_\_等同于调用x(1,2)。这个实例本身在这里相当于一个函数。

\_\_call()\_\_的作用是使实例能够像函数一样被调用，同时不影响实例本身的生命周期（\_\_call()\_\_不影响一个实例的构造和析构）。但是\_\_call()\_\_可以用来改变实例的内部成员的值。

Python中，如果在创建class的时候写了\_\_**call**（）\_\_方法， 那么该class实例化出实例后， 实例名()就是调用\_\_**call**（）\_\_方法。

===================

openstack里面的wsgi的知识点

OpenStack 中的每一个提供 REST API Service 的组件，比如 cinder-api，nova-api 等，其实是一个 WSGI App，其主要功能是接受客户端发来的 HTTP Requst，然后进行用户身份校验和消息分发。代码实现上，主要使用了几种技术：WSGI Server、WSGI Application、Paste deploy 和 Router

简单来说，python中的 WSGI 是 Python 应用程序或框架与 Web 服务器之间的一种接口，它定义了一套接口来实现服务器与应用端的通信规范，它将 web 组件分为三类:

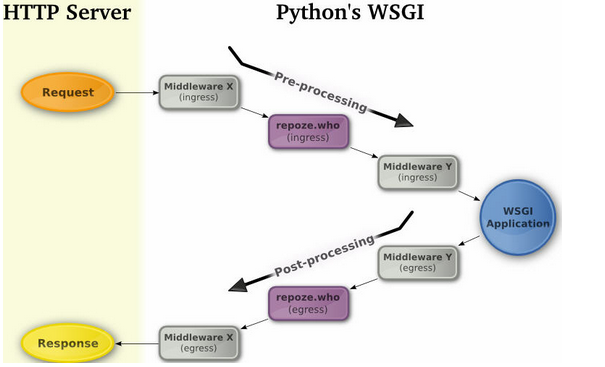
* web 服务器（Service)：接受客户端发来的 request，并返回 app 产生的 response 发回给客户端
* web 应用程序（App）: 每个 app 是一个 callable 对象。一个函数, 方法, 类, 或者实现了 \_\_call\_\_ 方法的实例对象都可以用来作为应用程序对象。服务器每次收到 http 客户端发来的请求都会调用相应的应用程序的 \_\_call\_\_ 方法去去处理。
* web 中间件（Middleware）：某些对象（app）可以在一些应用程序面前是服务器, 而从另一些服务器看来却是应用程序

（1）WSGI Server唯一的任务就是接收来自 client 的请求，然后将请求传给 application，最后将 application 的response 传递给 client。

（2）在使用 Middleware 的情况下，WSGI的处理模式为 WSGI Server -> (WSGI Middleware)\* -> WSGI Application。其实 middleware 本身已是一个 app，只是你需要有一个 App 做为实现主要功能的 Application。Middleware可以：

* 可以根据目的 URL 将一个请求分发 (routing) 给不同的应用程序对象, 并对 environ 做相应修改。这种 middleware 称为 router。
* 允许多个应用程序或框架在同一个进程中一起运行。可以依次调用多个 middleware。
* 通过分析 request，在网络上转发请求和响应, 进行负载均衡和远程处理.
* 对 response 内容进行后加工, 比如应用 XSL 样式.

（3）在 Server 和 Application 之间，可以使用若干个 Middleware 来处理 request 和 response：



具体过程描述如下：

1. 服务器为 Applicaiton 实例化一个 WSGIService 实例，实例化过程中调用 Paste Deployment 来加载/注册各 middleware 和app。
2. 服务器启动 WSGIService ， 包括创建 socket，监听端口，然后等待客户端连接。
3. 当有请求来时，服务器解析客户端信息放到环境变量 environ 中，并调用绑定的 handler 来处理请求。handler 解析这个 http 请求，将请求信息例如 method，path 等放到 environ 中。wsgi handler还会将一些服务器端信息也放到 environ 中，最后服务器信息，客户端信息，本次请求信息全部都保存到了环境变量environ中
4. wsgi handler 调用注册的各 app （包括 middleware） 来处理 request，比如 middleware 来进行数据检查、整理、用户验证等， wsgi app 生成 reponse header/status/body 回传给wsgi handler。
5. response 生成以后和回传之前，一些等待 response 的 被阻塞的 middleware 的方法可以对response 进行进一步的处理，比如添加新的数据等
6. 最终 handler 通过socket将response信息塞回给客户端

以 cinder-api 为例，其 WSGI 实例初始化和启动 service 的代码如下：

1. launcher = service.process\_launcher()
2. server = service.WSGIService('osapi\_volume') # 使用 Paste delopment 方式 加载 osapi\_volume 这个 application，这过程中包括加载各 middleware app
3. launcher.launch\_service(server, workers=server.workers) launcher.wait() #真正启动service

（launcher 发射器; 发射者; 运载火箭）

（关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict）