Абстрактные базовые классы и интерфейсы — близкие по назначению и смыслу сущности. Как первые, так и вторые представляют собой своеобразный способ документирования кода и помогают ограничить (decouple) взаимодействие отдельных абстракций в программе (классов).  
  
Питон — очень гибкий язык. Одна из граней этой гибкости — возможности, предоставляемые метапрограммированием. И хотя в ядре языка абстрактные классы и интерфейсы не представлены, первые были реализованы в стандартном модуле abc, вторые — в проекте Zope (модуль zope.interfaces).  
  
Нет смысла одновременно использовать и то и другое, и поэтому каждый программист должен определить для себя, какой инструмент использовать при проектировании приложений. 

## Абстрактные базовые классы (abс)

Начиная с версии языка 2.6 в стандартную библиотеку включается модуль abc, добавляющий в язык абстрактные базовые классы (далее АБК).  
  
АБК позволяют определить класс, указав при этом, какие методы или свойства обязательно переопределить в классах-наследниках:

from abc import ABCMeta, abstractmethod, abstractproperty

class Movable():

\_\_metaclass\_\_=ABCMeta

@abstractmethod

def move():

"""Переместить объект"""

@abstractproperty

def speed():

"""Скорость объекта"""

Таким образом, если мы хотим использовать в коде объект, обладающий возможностью перемещения и определенной скоростью, то следует использовать класс Movable в качестве одного из базовых классов.  
  
Наличие необходимых методов и атрибутов объекта теперь гарантируется наличием АБК среди предков класса:

class Car(Movable):

def \_\_init\_\_:

self.speed = 10

self.x = 0

def move(self):

self.c += self.speed

def speed(self):

return self.speed

assert issubclass(Car, Movable)

assert ininstance(Car(), Movable)

Видно, что понятие АБК хорошо вписывается в иерархию наследования классов, использовать их легко, а реализация, если заглянуть в исходный код модуля abc, очень проста. Абстрактные классы используются в стандартных модулях collections и number, задавая необходимые для определения методы пользовательских  
классов-наследников.  
  
Подробности и соображения по поводу использования АБК можно найти в PEP 3119  
(<http://www.python.org/dev/peps/pep-3119/>).

## Интерфейсы (zope.interfaces)

Реализация проекта Zope в работе над Zope3 решила сделать акцент на компонентной архитектуре; фреймворк превратился в набор практически независимых компонент. Клей, соединяющий компоненты — интерфейсы и основывающиеся на них адаптеры.   
  
Модуль zope.interfaces — результат этой работы.   
  
В простейшем случае использвание интерфейсов напоминает примерение АБК:

import zope.interface

class IVehicle(zope.interface.Interface):

"""Any moving thing"""

speed = zope.interface.Attribute("""Movement speed""")

def move():

"""Make a single step"""

class Car(object):

zope.interface.implements(IVehicle)

def \_\_init\_\_:

self.speed = 1

self.location = 1

def move(self):

self.location = self.speed\*1

print "moved!"

assert IVehicle.implementedBy(Car)

assert IVehicle.providedBy(Car())

В интерфейсе декларативно показывается, какие атрибуты и методы должны быть у объекта. Причем класс реализует (implements) интерфейс, а объект класса — предоставляет (provides). Следует обратить внимание на разницу между этими понятиями!   
  
«Реализация» чем-либо интерфейса означает, что только «производимая» сущность будет обладать необходимыми свойствами; а «предоставление» интерфейса говорит о конкретных возможностях оцениваемой сущности. Соответственно, в Питоне классы, кстати, могут как реализовывать, так и предоставлять интерфейс.  
  
На самом деле декларация implement(IVehicle) — условность; просто обещание, что данный класс и его объекты ведут себя именно таким образом. Никаких реальных проверок проводиться **не будет**

class IVehicle(zope.interface.Interface):

"""Any moving thing"""

speed = zope.interface.Attribute("""Movement speed""")

def move():

"""Make a single step"""

class Car(object):

zope.interface.implements(IVehicle)

assert IVehicle.implementedBy(Car)

assert IVehicle.providedBy(Car())

Видно, что в простейших случаях интерфейсы только усложняют код, как, впрочем, и АБК  
  
Компонентная архитектура Zope включает еще одно важное понятие — адаптеры. Вообще говоря, это простой шаблон проектирования, корректирующий один класс для использования где-то, где требуется иной комплект методов и атрибутов. Итак,

## Адаптеры

Рассмотрим, сильно упростив, пример из [Comprehensive Guide to Zope Component Architecture.](http://www.muthukadan.net/docs/zca.html)  
  
Предположим, что имеется пара классов, Guest и Desk. Определим интерфейсы к ним, плюс класс, реализующий интерфейс Guest:

import zope.interface

from zope.interface import implements

from zope.component import adapts, getGlobalSiteManager

class IDesk(zope.interface.Interface):

def register():

"Register a person"

class IGuest(zope.interface.Interface):

name = zope.interface.Attribute("""Person`s name""")

class Guest(object):

implements(IGuest)

def \_\_init\_\_(self, name):

self.name=name

Адаптер должен учесть анонимного гостя, зарегистрировав в списке имен:

class GuestToDeskAdapter(object):

adapts(IGuest)

implements(IDesk)

def \_\_init\_\_(self, guest):

self.guest=guest

def register(self):

guest\_name\_db.append(self.guest.name)

Существует реестр, который ведет учет адаптеров по интерфейсам. Благодаря ему можно получить адаптер, передав в вызов класса-интерфейса адаптируемый объект. Если адаптер не зарегистрирован, то вернется второй аргумент интерфейса:

guest = Guest("Ivan")

adapter = IDesk(guest, alternate=None)

print adapter

>>>>None found

gsm = getGlobalSiteManager()

gsm.registerAdapter(GuestToDeskAdapter)

adapter = IDesk(guest, alternate="None found")

print adapter

>>>>\_\_main\_\_.GuestToDeskAdapter object at 0xb7beb64c>

Такую инфраструктуру удобно использовать для разделения кода на компоненты и их связывания.   
  
Один из ярчайших примеров использования такого подхода помимо самого Zope — сетевой фреймворк Twisted, где изрядная часть архитектуры опирается на интерфейсы из zope.interfaces. 

## Вывод

При ближайшем рассмотрении оказывается, что интерфейсы и абстрактные базовые классы — разные вещи.  
  
Абстрактные классы в основном жестко задают обязательную интерфейсную часть. Проверка объекта на соответствие интерфейсу абстрактного класса проверяется при помощи встроенной функции isinstance; класса — issubclass. Абстрактный базовый класс должен включаться в иерархию в виде базового класса либо mixin`а.  
  
Минусом можно считать семантику проверок issubclass, isinstance, которые пересекаются с обычными классами (их иерархией наследования). На АБК не выстраивается никаких допонительных абстракций.  
  
Интерфейсы — сущность декларативная, они не ставят никаких рамок; просто утверждается, что класс реализует, а его объект предоставляет интерфейс. Семантически утверждения implementedBy, providedBy являются более корректными. На такой простой базе удобно выстраивать компонентную архитектуру при помощи адапетров и других производных сущностей, что и делают крупные фреймворки Zope и Twisted.  
  
Надо понимать, что использование обоих инструментов имеет смысл только при построении и использовании сравнительно крупных ООП-систем — фреймворков и библиотек, в малых программах они могут только запутать и усложнить код код лишними абстракциями.

Источник: https://habr.com/post/72757/