**Изучаем Python**

**Марк Лутц 2011, издание O’ Reilly**

**Том 3**

***Часть. 5. Модули*** *стр. 605*

**Глава 21. Модули: общая картина.** стр. 607

Модули - самая крупная организационная программная единица, которая вмещает код и данные, готовые для многократного использования.

Модули обычно соотв-ют файлам программ (или расширениям др. ЯП как С, Java, C#).

Каждый файл - это отдельный модуль, конечно модули могут экспортировать др. модули.

Модули обрабатываются двумя инструкциями и одной функцией:

- **import** - Позволяет клиентам (импортерам) получать модуль целиком.

- **from** - Позволяет клиентам получать определенные имена из модуля.

- **imp.reload** - Обеспечивает возможность повторной загрузки модуля без остановки интерпретатора.

**Зачем нужны модули?**

Модули обеспечивают простой способ организации компонентов в систему автономных пакетов переменных, известных как пространства имен.

Все имена, определяемые на верх. уровне модуля, становятся атрибутами объекта импортируемого модуля.

В процессе импорта глобал. область модуля образует пространство имен атрибутов объекта модуля.

Модули позволяют связывать отдельные файлы в крупные программные системы.

По теории, модули играют как минимум три роли:

- ***Повторное использование программного кода.*** Т.к. модули хранят код в виде файлов то код, т.о., хранится постоянно, т.е. его можно загружать и запускать сколько угодно раз. Так модули становятся местом, где определяются имена, известные как - атрибуты, на которые могут ссылаться внешние клиенты.

- ***Разделение системы пространства имен.*** По сути, модули - всего лишь пакеты имен. Т.о. появл-ся изоляция - мы никогда не узнаем имя в др. модуле пока не импортируем его. Выполняемый код и создаваемые объекты модуля - всегда неявно включается в модули.

- ***Реализация служб или данных для совместного использования.*** С функциональной точки зрения, модули могут использ-ся для реализации компонентов, используемых системой, вследствие чего требуется только одна копия такого компонента. Нарп. если нужен глобал. объект, использующийся более одной ф-цией или модулем, можно написать его в виде модуля, который будет импортироваться многими клиентами.

**Архитектура программы на языке Питон.**

Обычно программы состоят более чем из одного файла, даже если ваш файл всего один наверняка вы используете какие-то модули.

Изучим введение в архитектура программ - способ, которым программа делится на коллекцию файлов с исходными текстами (т.е. модулей) и увязывается в единое целое.

**Как организована программа.**

Как правило, программа состоит из множества текстовых файлов, содержащих инструкции. Программа организована как один главный файл, к которому могут подключаться доп. файлы, известные как модули.

Главный файл (или сценарий) определяет, как будет двигаться основной поток выполнения программы, - это тот файл, который нужно запустить, чтобы начать работу приложения.

Файлы модулей - библиотеки инструментов, где содержатся компоненты, используемые главным файлом (и может где-то еще). Глав. файл использ-т инструменты, определенные в файлах модулей, а модули использ. инструменты, из др. модулей.

Обычно файлы модулей ничего не делают сами по себе, - в них определяются инструментальные средства, используемые в др. файлах. Чтобы получить доступ к инструментам модуля, наз-ся атрибутами модуля, необходимо импортировать модуль.

**Импортирование и атрибуты.**

***Рисунок 21.1 Структура программ на Питон.*** стр. 610

Три файла: a.py, b.py, c.py. Файл a.py явл-ся глав. файлом программы, в нем инструкции. Файлы b.py, c.py - модули, также простые текстовые файлы с инструкциями, но как самостоятельные программы они обычно не запускаются.

Пусть b.py определяет ф-цию spam:

def spam(text):

print(text, 'spam')

Предположим, что модуль a.py использ-т spam, описание его такое:

import b

b.spam('gumby')

- получаем доступ ко всему, что определено на верхнем уровне b.py (т.е. ко всем атрибутам).

Импорт нужен, чтобы связать имена в модуле - простые переменные - с объектами загруженных модулей.

Фактически, имя модуля в инструкции import, идентифицирует внешний файл, и становится именем переменной, которая будет представлять загруженный модуль.

b.spam - вызывает ф-цию, поределенную в b.py ("извлечь значение имени psam, расположенного в b").

Мы увидим, что повсюду в Питон использ-ся нотация object.attribute (здесь точка "." - это оператор).

Любой файл может импортировать функциональные возможности из другого файла.

**Модули стандартной библиотеки.**

Интерпретатор Питон поставл-ся с обширной коллекцией доп. модулей, которая известна как стандартная библиотека. Она насчитывает около 200 модулей (это из книги цифра) и платформонезависимую поддержку разных задач: интерфейсы ОС, организацию хранилищ объектов, поиск по шаблону, сетевые взаимодействия, создания графич. интерфейса и др.

Они могут использ-ся при импортировании необходимого модуля.

**Как работает импорт.**  Стр. 612

Программисты на С любят сравнивать инструкцию import в Питон с инструкцией #include, но они в корне

неправы - импортирование в Питон - это не просто включение текста одного файла в другой.

Это самые настоящие операции времени выполнения, которые вып-т след. действия, когда программа импортирует файл:

1. *Отыскивают* файл модуля.

2. *Компилируют* в байт-код (если нужно).

3. *Запускают* программный код модуля, чтобы создать объекты, которые он определяет.

Внимание. Все три действия вып-ся только если модуль импортируется впервые во время выполнения программы - последующие операции импорта этого же модуля пропускают эти действия и просто выбирают, уже находящийся в памяти объект модуля.

Интерпретатор сохраняет информацию о загруженных модулях в словаре с именем sys.modules и проверяет его при каждой операции импортирования. Если модуль отсутствует, то выполняется процесс, описанный выше.

**1. Поиск.**

На этом этапе интерпретатор определяет местонахождение файла модуля.

Заметим, что имена модулей в инструкции import указываются без расширения .py и без пути к каталогу, эти ограничения выполняются потому что интерпретатор использует стнадартный путь поиска модулей.

**2. Компиляция (если необходимо).**

Когда файл найден интерпретатор компилирует его в байт-код, если нужно.

Интерпретатор проверяет время создания файла и пропускает компиляцию, если файл с байт-кодом .pyc не старше, чем соответствующий ему файл .py с исходным текстом.

Если Питон находит только файл с байт-кодом и не найдет файл с исходным текстом, то он просто загрузит байт-код. Это даже ускорит запуск программы.

Обратите внимание, компиляция выполняется в момент импортирования файла. По этой причине файл .pyc с байт-кодом для глав. файла программы обычно не создается, если тольк оон не был импортирован куда-то еще (файл .pyc создаются при импортировании). Байт-код файла программы создается в памяти ПК, а импортированных файлов сохр-ся в файлах для ускорения будущих импортов.

Также можно создавать глав. файлы, которые будут играть роль и главного файла и импортируемого модуля, но об этом потом.

**3. Запуск.**

На последнем шаге идет запуск байт-кода модуля. все инструкции в файле модуля выполняются по порядку, сверху вниз, и любые операции присваивания, которые встретятся на этом шаге, будут создавать атрибуты конечного объекта модуля. Т.о. этот этап создает все инструменты, которые опред-ся модулем (напр. выполняются инструкции def, создающие ф-ции и мн. др.).

Т.о. фактически запускается программный код модуля, и если код выполняет какие-то действия, то мы сможем их наблюдать во время импорта (напр. действия инструкции print).

**Путь поиска модулей.**

Нам может потребоваться сообщить интерпретатору, где искать импортируемые файлы, поэтмоу нужно знать как получить доступ к пути поиска.

Имеются следующие источники поиска:

1. Домашний каталог программы (в моей Убунте это - домашняя папка моего пользователя).

2. Содержимое переменной окружения PYTHONPATH (если таковая определена).

3. Каталоги стандартной библиотеки.

4. Содержимое любых файлов с расширением .pht (если таковые имеются).

Объединение этих компонентов составляет sys.path - список строк с именами каталогов, о нем поговорим позднее.

Первый и третий пути определяются автоматически, а второй и четвертый можно использовать для расширения пути поиска.

***Домашний каталог.***

В первую очередь интерпретатор ищем в домашнем каталоге. В зависимости от того как была запущена программа, это может быть каталог, где нах-ся глав. файл программы. У меня в Убунте модули приходится сохранять в домашней папке пользователя, а в Виндовс - в папке Питона.

***Каталоги в PYTHONPATH.***

Затем произв-ся поиск в каталогах окружения PYTHONPATH, слева направо (если переменная установлена). Переменная PYTHONPATH - это просто список имен каталогов, определяемых пользователем и системой. Мы можем добавит ьв эту переменную все каталоги где у нас будут модули. Конечно, для начала можно её не настраивать, а хранить модули в домашнем каталоге. А уж потом, как получите больше опыта, обращться к её изменению.

***Каталоги стандартной библиотеки.***

Т.к. эти каталоги автоматически участвуют в поиске, их можно не добавлять в переменную PYTHONPATH и не включат ьв файлы .pth.

***Каталоги в файле .pth.***

Эта особенность Ппитона дает возможность просто перечислисть нужные нам каталоги в текстовом файле, с расширением .pth (от слова "path" - "путь"). Это расширенная возможность, альтернатива настройки переменной PYTHONPATH.

Текстовый файл со списком каталогов помещается в соответствующий каталог и играет роль PYTHONPATH. В Виндовс в Питон 3.0, например, файл myconfig.pth можно поместить в глав. каталог (C:\Python34 - т.к. у меня установлен Питон 3.4), или в под-каталог site-packages C:\Python34\lib\site-packages стандартной библиотеки. В Unix этот файл можно поместить в каталог /usr/local/lib/python34/site-packages или /usr/local/lib/site-python.

Интерпретатор выберет все имена каталогов во всех файлах .pth, которые обнаружит.

За доп. информацией обращайтесь к документации, в частности к описанию модуля site - этот модуль позволяет создавать файлы .pth и определять местоположение библиотек, в документации модуля описывается где вообще могут располагаться файлы .pth.

Начинающим рекомендуется использовать переменную PYTHONPATH или, если очень нужно, то один файл .pth.

**Настройка пути поиска.**

Т.о. понятно, что переменная окружения PYTHONPATH и файлы .pth позволяют определять каталоги, где интерпретатор будет искать файлы при импортировании.

Способ настройки зависит от платформы, например мы можем через свойства системы в Виндовс просто записать в переменную PYTHONPATH список каталогов:

c:\pycode\utilites;d:\pycode\package1 # например таким образом (без пробелов!)

Или создать текстовый файл с именем C:\Python43\pydirs.pth с текстом примерно:

c:\pycode\utilites

d:\pycode\package1

В Приложении А мы сможем найти указания по настройке пути поиска на различных платформах.

**Автоматическое изменение пути поиска.**

На самом деле точная конфигурация поиска зависит от платформы и версии Питона.

Например, за каталогами из PYTHONPATH и перед каталогами стандарт. библиотеки интерпретатор может добавлять текущий рабочий каталог - каталог, откуда была запущена программа.

Когда прога запускается из командной строки, текущий рабочий каталог может не совпадать с домашним каталогом, где находится глав. файл проги (т.е. с каталогом где нах-ся прога). Т.к. от запуска к запуску проги текущий рабочий каталог может меняться, при обычных условиях рабочий каталог не должен иметь значения для операции импорта.

**Список sys.path**

Если хотите узнать действительный путь поиска на своем ПК, то можете посмотреть содержимое встроенного списка sys.path (т.е. содержимое атрибута path модуля sys). ***sys.path*** - это путь поиска модулей. Интерпретатор создает его во время запуска программы, автоматом объединяя в список домашний каталог (или пустую строку, что соответствует текущему рабочему каталогу), все каталоги, перечисленные PYTHONPATH и в файлах .pth, и каталоги стандарт. библиотеки.

Представление этого списка имеет два полезных результата:

- *Первое* - он обеспечивает возможность проверить настройки пути поиска, которые вы выполнили (если их нет в этом списке, то проверьте, что вы там наделали). Напр. автор добавил в PYTHONPATH каталог C:\users и в файл C:\Python34\mypath.pth каталог С:\users\mark, смотрим список (вначале пустая строка, соотв-щая текущему рабочему каталогу):

>>> import sys

>>> sys.path

['', 'C:\\users', 'C:\\Python34\\Lib\\idlelib', 'C:\\WINDOWS\\system32\\python34.zip', 'C:\\Python34\\DLLs', 'C:\\Python34\\lib', 'C:\\Python34', 'С:\\users\\mark', 'C:\\Python34\\lib\\site-packages']

- здесь мы видим где располагаются в списке оба добавленных каталога.

- *Второе* - если мы понимаем как формируется список, можем дать сценариям возможность самостоятельно задавать свои пути поиска. Изменяя sys.path мы можем изменить путь поиска для всех последующих операций импорта. Не забывайте - эти изменения продолжают действовать пока выполняется сценарий (иногда это даже полезно, напр. на веб-серверах, ведь там миллионы пользователей в день бывает). PYTHONPATH и файлы .pth обеспечивают более долговременное хранение пути.

**Выбор файла модуля.**

Интерпретатор выбирает первый, найденный в пути поиска, файл, который соотв-ет указанному имени (имена в import пишем без расширений, помним). Наприме import b мог бы загрузить:

- Файл с кодом, с именем b.py;

- Файл с байт-кодом, с именем b.pyc;

- Содержимое каталога b при импорте пакета (см. гл. 23);

- Скомпилированный модуль расширения (С или С++) в виде динамической библиотеки (b.so в Linux и b.dll или b.pyd в Cygwin и Windows);

- Скомпилированный встроенный модуль, на С и статически скомпонованный с интерпретатором;

- Файл zip-архива с компонентом, который автоматич. извлек-ся при импорте;

- Образ памяти для фиксированных двоичных исполняемых файлов;

- Класс Java в Jython;

- Компонент .NET в IronPython.

Для импортера не важны типы загружаемых файлов как при импорте так и при обращении к атрибутам модуля. Напри, некоторые модули, которыми будем польз-ся, на самом деле написаны на С, - но для клиентов это не имеет значения.

Если у нас в разных каталогах есть файлы b.py и b.so, интерпретатор будет загружать тот, что будет найден в каталоге, который располагается раньше (левее) в пути поиска модулей (поиск в списке sys.path идет слева направо).

Если же оба файла b.py и b.so находятся в одном каталоге, интерпретатор будет следовать стандарт. порядку выбора файлов (который может измениться в др. версиях языка, например, т.е. гарантий нет).

Т.о. видим, что мы должны избегать зависимости от порядка выбора файлов интерпретатором в одном и том же каталоге - давайте модулям различные имена и настраивайте путь поиска так ,чтобы порядок выбора был очевидным.

**Дополнительные возможности выбора модуля.**

Мы можем изменять обычную операцию импорта, и переопределять её большую часть, используя то, что называется - программными ловушками импорта. Эти ловушки придают операции импорта доп. возможности, напр. загрузка файлов из архивов, расшифровывание и т.д.

Сам интерпретатор использ. эти ловушки, чтобы извлекать компоненты из zip-архивов (см. выше). Нарп. один из каталогов стандарт. библиотеки в sys.path, на сегодняшний день (от атора книги) явл-ся файлом .zip.

За доп. информацией обращайтесь с описанию ф-ции \_\_import\_\_, которая использует инструкцию import.

Питон поддерживает также понятие файлов с оптимизированным байт-кодом (.pyo), которые созд-ся и запуск-ся интерпретатором из командной строки с флагом -0, - однако выполняются немногим быстрее чем обычные файлы .pyc, поэтому использ-ся редко.

**Стороннее программное обеспечение: distutils.**

Настройка пути поиска в первую очередь относится к коду, который мы пишем самостоятельно.

Сторонние расширения Питон обычно использ-т для автоматической установки самих себя такой инструмент, как disrutils, входящий в состав стандарт. библиотеки.

Системы, использующие distutils, обычно поставляются со сценарием setup.py, который запускается для установки таких систем - этот сценарий импортирует и использ. модуль distutils, чтобы поместить систему в каталог, котоырй уже явл-ся частью пути поиска модулей (обычно подкаталог Lib\sitepackages в каталоге, куда был установлен Питон).

Этот инстурмент также обеспечивает возможность компиляции расширений на С на машине, где происходит установка. Обратите также внимание на систему "eggs" (новая система на момент напсиания книги).

**Глава 22. Основы программирования модулей.** Стр. 623

**Создание модуля.**

В текстовом редакторе вводим программный код в текстовый файл и сохраняем его с расширением .py - любой такой файл автоматически будет считаться модулем Питона (практически любой текстовый файл может играть роль модуля).

Все имена, которым будет выполнено присваивание на верхнем уровне модуля, станут его атрибутами (именами ассоциированными с объектами модуля) и будут доступны для использования клиентами.

Напр. инструкция def в файле module1.py, импортируя его мы получаем объект модуля с атрибутом - printer, которое ссылается на объхект ф-ции:

def printer(x): # Атрибут модуля

print(x)

Мы можем называть файлы как угодно, главное расширение .py, если собираемся импортировать модуль.

Глав. файлы, которые будут запускаться, но не будут импортироваться, расширение не обязательно должно быть .py, но желательно использовать его (вдруг захотим импортировать глав. файл в будущем).

Т.к. имена файлов становятся именами переменных, в инструкции import, они должны следовать правилам именования обычных переменных (см. гл. 11).

Напр. можно создать файл модуля с именем if.py, но его невозможно импортировать, т.к. if - это зарезервированное слово, мы получим ошибку, если попытаемся импортировать.

И имена модулей и имена каталогов (использ-ч при импортировании пакетов) должны соотв-ть правилам именования переменных из гл. 11, т.е. содержать только алфавитные символы, цифры и символы подчеркивания.

При импортировании преобразуется имя модуля в имя внеш. файла, добавляя в начало путь из каталога путей поиска и расширение .py, или др. расширение.

Также ест ьвозможность создать модуль на др. ЯП, такие модули наз-ся модулями расширений. Когда они импортируются, то ведут себя как обычные модули, написанные на Питон.

**Использование модулей.**

Клиенты могут использовать модуль выполнив import или from. Обе инструкции отыскивают, компилируют и запускают программный код модуля, если он еще не был загружен. import загружает модуль целиком, поэтому имена нужно дополнять именем модуля (мы с этим уже столкнулись), а инструкция from загружает (или копирует) из модуля отдельные имена (и имя модуля уже не нужно).

Создаем в папке C:\Python34 файл module1.py с ф-цией printer, и далее в примерах используем этот модуль различными способами.

**Инструкция import.**

В этом примере module1 идентифицирует внеш. файл и вревращается в имя переменной, которая ссылается на объект модуля после загрузки файла:

>>> import module1 # Загрузить модуль целиком

>>> module1.printer('Hello World!') # имя дополняем именем модуля

Hello World!

**Инструкция from.**

Инструкция from копирует имена из области видимости одного файла в область видимости другого:

>>> from module1 import printer # Копировать одну переменную

>>> printer('HelloWorld!') # Имя не требует дополнения

HelloWorld!

Инструкция from явл-ся всего лишь расширением инструкции import - она импортирует модуль как обычно, но выполняет доп. шаг, на котором копирует одно или более имен.

**Инструкция from \***

Когда используется символ \* в from, копируются все имена. В этом случае точно так же можно использовать скопированное имя printer, не предваря его именем модуля:

>>> from module1 import \*

>>> printer('HelloWorld!')

HelloWorld!

В Питон 3.0 инструкция from ... \*, может использ-ся только на верхнем уровне модуля - она не может вызываться внутри функций. В 2.6 можно вызывать внутри ф-ций, но показывает предупреждение.

**Импорт выполняется только один раз.**

Почему же модуль выполняет свои операции только при первом импортировании (напр. print()), а при последующем ничего не происходит? На самом деле этот эффект и должен быть.

Модули загружаются и запускаются первой и только первой инструкцией import или from. Это сделано потому, что импортирование - это дорогостоящая операция и поэтому выполняется один раз. А при последющем импорте просто получают объект загруженного модуля.

Т.к. код на верх. уровне модуля выполняется всего один раз, это можно использ. для инициализации переменных. Рассмотрим пример модуля simple.py:

print('Hello')

spam = 1

В этом примере инструкции print и = выполняются во время первого импортирования модуля и в это же время инициализируется переменная spam:

>>> import simple # Первая инструкция import: загружает и запускает код модуля

Hello!

>>> simple.spam # Операция присваивания создает атрибут

1

Все последующие операция импорта не приводят к перезапуску кода модуля - они просто получают объект модуля из внутренней таблицы модулей интерпретатора:

>>> simple.spam = 2 # Изменяем атрибут модуля

>>> import simple # Эта инструкция получает уже загруженный ранее модуль

>>> simple.spam # Код не перезапускается: атрибут не изменился

2

**Инструкции import и from - операции присваивания.**

Как и def, инструкции import и from явл-ся исполняемыми инструкциями, а не объявлениями времени компиляции. Они могут вкладываться в инструкции if, присутствовать в объявлениях def и т.д., и они не имеют эффекта пока интерпретатор не достигнет их в ходе выполнения проги. Т.е. импортируемые модули и имена не будут доступны пока не будут выполнены import или from.

Также import и from - это явные операции присваивания:

- *import* присваивает объект модуля единственному имени;

- *from* присваивает одно или более имен объектам с теми же именами в др. модуле.

Всё, что обсуждалось ранее применимо и к модулям. Например, имена, копируемые from, становятся ссылками на разделямые объекты - как и аргументы ф-ций, повторное присваивание полученному имени не оказывает воздействия на модуль, откуда это имя было скопировано, но модификация изменяемого объекта может оказывать влияние на объект в модуле, откуда он был импортирован. Пример с файлом small.py:

x = 1

y = [1, 2]

>>> from small import x, y # Скопировать два имени

>>>

>>> x = 42 # Изменяется только локальная переменная х

>>> y[0] = 42 # Изменяется непосредственно изменяемый объект

Имена y в импортирующем и импортируемом модулях ссылаются на один и тот же объект списка:

>>> import small # Получить имя модуля (инструкция from его не дает)

>>> small.x # x в модуле small - это не наша локал. переменная х

1

>>> small.y # а изменяемый объект исполз-ся повсеместно

[42, 2]

Кстати, вернитесь к рис. 18.1 и замените слова "вызывающая программа" и "функция" на "импортируемый модуль" и "импортирующий модуль" и получите графическое изображение наших примеров.

**Изменение значений имен в других файлах.**

В примере выше переменная х менялась только в IDLE, но не менялась в файле, т.к. между именем, скопированным from и именем в файле нет никакой связи.

Чтобы изменить глобал. имя в др. файле, нужно использовать import:

>>> from small import x, y # Копируем два имени

>>> x

1

>>> x = 42 # Изменяем локал. имя х

>>> x

42

>>> import small # Получаем имя модуля

>>> x # Локальное х по-прежнему 42

42

>>> small.x # А х в модуле по-прежнему равно 1

1

>>> small.x = 42 # Изменяем х в модуле (но не в файле на диске, конечно)

>>> x

42

>>> small.x # Теперь оба х равны 42

42

- По этому явлению см. главу 17. Такое изменение переменных может быть источником проблем (связи все такое), нужно быть очень внимательными.

**Эквивалентность инструкций import и from.**

Инструкция from, приведенная ниже:

from module import name1, name2 # Копировать только эти два имени

- эквивалентна следующей последовательности:

import module # Получаем объект модуля

name1 = module.name1 # Копирванием имена с помощью присваивания

name2 = module.name2

del module # Удалить имя модуля (from не копирует сам модуль)

Т.о. from создает новые переменные в импортирующем модуле, которые ссылаются на объект с теми же имена в импортируемом модуле.

**Потенциальные проблемы инструкции from.**

from делает местоположение переменных менее явным (напр. name и module.name), поэтому некоторые рекомендуют использ. import вместо from. Но ве-таки from тоже полезна, ведь удобнее писат ьимена без приставки имени модуля (особенно если модуль имеет много атрибутов).

Суть проблемы в том, что from способна повреждать пространства имен, - если при импорте сущ-ют одноименные переменные в имеющейся области видимости, то они будут перезаписаны.

import не дает такой проблемы, т.к. имя модуля указывается вместе с атрибутами. Но если from использовать внимательно и осторожно, то можно избежать таких проблем.

С др. стороны, имеются более серьезные проблемы при использ. from вместе с reload, т.к. импортированные имена могут ссылаться на предыдущие версии объектов.

Также инструкция в форме from module import \* может повреждать пространства имен и затрудняет понимание имен.

Отдавайте предпочтение import для простых модулей, явно перечисляйте имена в from (без \*, если можно) и не используйте форму from \* для импорта более чем одного файла в модуле. И конечно осторожность и внимательность.

**Когда необходимо использовать инструкцию import.**

Главное - это когда нужно использовать одно и тоже имя, присутствующее в двух разных модулях. Напр. если файлы по-разному определяют одно и то же имя:

# M.py

def func():

...выполнить что-то одно...

# N.py

def func():

...выполнить что-то другое...

В этом случае использ. from нельзя, т.к. получим одно, а не два имени:

# O.py

from M import func

from N import func # перезапишет имя, импортированное из модуля M

func() # Будет вызвана N.func

В этом случае нужен import:

# O.py

import M, N # Получаем модуль целиком

M.func() # Теперь можем вызывать обе ф-ции

N.finc() # Наличие имени модуля делает их уникальными

**Пространства имен модулей.** Стр. 630

Модули будут понятны, если представлять их как пакеты имен - т.е. место, где определяются переменные, которые должны быть доступны остальной системе.

Технически каждому модулю соотв-т отдельный файл, и интерпретатор создает объект модуля, содержащий все имена, которым присвоены значения в файле модуля.

**Файлы создают пространства имен.**

Каждое имя, которому присв-ся некое значение на верх. уровне модуля (т.е. не вложенное в ф-ции или классы), превращается в атрибут этого модуля.

Напр. Х = 1 на верх. уровне модуля M.py превращает имя Х в атрибут модуля М (доступно как М.Х). Также имя Х становится глобал. переменной для программного кода внутри модуля M.py.

Уточним некоторые моменты:

- **Инструкции модуля выполняются во время первой попытки импорта.** Первый импорт - созда-ся пустой объхект модуля и выполняются инструкции одна за другой.

- **Операции присваивания, выполняемые на верхнем уровне, создают атрибуты модуля.**

- **Доступ к пространствам имен модулей можно получить через атрибут \_\_dic\_\_ или dir(M).** Пространства имен модулей - это словари - доступ к ним ест ьчерез атрибут \_\_dict\_\_, и с помощью ф-ции dir.

- **Модуль - это единая область видимости (локальная является глобальной).** Имена на верх. уровне модуля выполняют те же правила обращения/присваивания, что и имена в ф-циях (т.е. соотв-ют правилу LEGB только без уровней L и E). В модулях область видимости модуля, после загрузки модуля, превращается в атрибут-словарь объекта модуля.

Примеры. Создаем файл module2.py:

print('starting to load...')

import sys

name = 42

def func(): pass

class klass: pass

print('done loading.')

А теперь импортируем его (в это время вып-ся все инструкции):

>>> import module2

starting to load...

done loading.

Модуль загружен и его область видимостипревращается в пространство имен атрибутов объекта модуля. Теперь можно обращаться к атрибутам в этом пространсте:

>>> module2.sys

<module 'sys' (built-in)>

>>> module2.name

42

>>> module2.func

<function func at 0x0227B588>

>>> module2.klass

<class 'module2.klass'>

Обратите внимание на атрибут sys - инструкции import действительно присваивают объекты модулей именам, а любая операция присваивания на верх. уровне создает атрибут модуля.

Пространство имен внутри интерпретатора - это обычный словарь и можно обратиться к нему через атрибут \_\_dict\_\_ модуля (и list не забудьте):

>>> list(module2.\_\_dict\_\_.keys())

['sys', 'klass', '\_\_cached\_\_', '\_\_spec\_\_', '\_\_package\_\_', 'func', 'name', '\_\_file\_\_', '\_\_name\_\_', '\_\_builtins\_\_', '\_\_loader\_\_', '\_\_doc\_\_']

Имена в файле модуля становятся ключами внутреннего словаря (мы видим список т.к. выборку ключей обернули в list). Например \_\_file\_\_ содержит имя файла, из которого загружен модуль, а \_\_name\_\_ - это имя модуля, которое пишется в инструкции import.

**Квалификация имен атрибутов.**

В Питон, для доступа к атрибутам любого объекта использ. синтаксис квалификации имени onject.attribute.

Квалификация имени явл-ся выражением, возвращающим значение, присвоенное имени атрибута, ассоциированного с объектом.

module2.sys возвращает значение атрибута sys в объекте module2. Также, если есть встроенный объект списка L, L.append вернет метод append, ассоциированный с этим списком.

Принятые правила квалификации:

*- Простые переменные.* Если используем краткую форму имени, напр. Х, то будет идти поиск в текущих областях видимости (согласно LEGB).

- *Квалифицированные имена.* Имя X.Y означает, что будет поиск имени Х в текущих областях видимости, а затем будет поиск атрибута Y в объекте X (не в областях видимости).

- *Квалифицированные пути.* Имя X.Y.Z означает, что будет поиск имени Y в объекте X, а затем имени Z в объекте X.Y

- *Общий случай.* Квалификация имени применима ко всем объектам, имеющим атрибуты: модулям, классам, расшиерниям типов на С и т.д.

**Импортирование и область видимости.**

Примеры. Модуль moda.py опредлеляет переменную Х, и ф-цию, изменяющую Х:

X = 88 # X - глобальная только для этого файла

def f():

global X # Изменяет Х в этом файле

X = 99 # Имена в др. модулях недоступны (они внутри ф-ции)

Второй модуль modb.py имеет свою глобал. переменную Х, а также импортирует и вызывает ф-цию из первого модуля:

X = 11 # X - глобальная только для этого файла

import moda # получаем доступ к именам в модуле moda

moda.f() # Изменяет переменную moda.X, но не Х в modb

print(X, moda.X)

Глобальной областью ф-ции moda.f всегда явл-ся файл, вмещающий её, независимо от того, из какого модуля она была вызвана:

>>> import modb

11 99

Т.е. операция импортирования никогда не изменяет область видимости для программ. кода в импортируемом файле - из импортируемого файла нельзя получить доступ к именам в импортирующем файле.

- Функциям никогда не будут доступны имена, определнные в др. ф-циях, если только они не вложены друг в друга.

- Программ. коду модуля никогда не будут доступны имена, определенные в др. модулях, если только они явно не были импортированы.

Области видимости не подвержены влияниям вызовов ф-ций или операции импортирования, они полностью определяются физическим расположением этого программного кода в файле.

**Вложенные пространства имен.**

Импорт не дает доступ к внеш. областям видимости, но дает доступ к вложенным областям. Используя квалифицированные пути к именам атрибутов, можно погрузиться в сколь угодно глубоко вложенные модули и получить доступ к их атрибутам.

Файл mod3.py определяет единственное глобальное имя и атрибут операцией присваивания:

Х = 3

Файл mod2.py определяет свою переменную Х, затем импортирует модуль mod3 и использует квалификацию имени, чтобы получить доступ к атрибуту импортированного модуля:

Х = 2

import mod3

print(X, end=' ') # Моя глобальная переменная Х

print(mod3.X) # Глобал. переменная Х из модуля mod3

У файла mod1.py своя переменная Х, импорт mod2 и получает значения атрибутов обоих модулей:

X = 11

import mod2

print(X, end=' ') # Моя глобалю переменная Х

print(mod2.X, end=' ') # Переменная Х из модуля mod2

print(mod2.mod3.X) # Переменная Х из модуля mod3

Когда mod1 импортирует mod2, он создает двухуровневое вложение пространств имен. Используя путь mod2.mod3.X, он может погрузиться в модуль mod3, котоырй вложен в импортированный модуль mod2. Суть в том, что модуль mod1 может обращаться к переменным Х вовсех трех файлах, и поэтмоу имеет доступ ко всем трем глоабл. областям видимости:

>>> import mod1

2 3

1 2 3

А модуль mod3 не имеет доступ к именам в mod2, а модуль mod2 не имеет доступ к именам в mod1. Если переведем взгляд на объекты, задействованные в примене, то увидим, что mod2 внутри mod1 - это всего лишь имя, которое ссылается на объект с атрибутами, некоторые из них могут ссылаться на др. объекты с атрибутами. Пути вида mod2.mod3.X Питон вычисляет слева направо, извлекая атрибуты из объектов.

В mod1 можно вставить инструкцию import mod2 и использовать mod2.mod3.X, но нельзя записать import mod2.mod3 - такой синтаксис использ. для операции импортирования пакетов (каталогов).

**Повторная загрузка модулей.** Стр. 635

Повторно загрузить и зщапустить модуль можно при помощи ф-ции reload. Исследуем некоторые сообенности повторной загрузки:

- При вызове операции импортирования (import и from) программный код модуля загружается и выполняется, тольк окогда модуль импортируется в первый раз за время работы проги.

- При последующих попытках импорта модуль будет использ-ть объект уже загруженного модуля. Повторная загрузка и запуск кода при этом не происходит.

- Ф-ция reload принудительно выполняет повторную загрузку уже загруженного модуля и запускает его код. Инструкции присваивания при этом будут изменять существующий объект модуля.

Т.о. reload позволяет изменять части проги, не останавливая всю прогу. Напр. мы работаем с БД, которая при запуске соединяется с сервером - благодаря повторной загрузке изменения и настройки можно применять в ходе сеанса работы с БД, не перезапускаясь.

Модули загружаются динамически уже запущенной программой (нет компиляции/компоновки как на С), повторная загрузка повышает **производительность** таких систем. Сейчас (в момент написания книги) ф-ция reload может работать только с модулями на Питоне, модули расширений не поддерживают такую операцию.

*Примечание.*

В 2.6 reload - встроенная ф-ция, а в 3.0 она перемещена в модуль imp, и доступна как imp.reload. Т.е. сначала нужно загрузит ьеё, выполнив import или from.

**Основы использования функции reload.**

В отличие от import, from:

- reload - это не инструкция, а функция.

- Ф-ции reload передается существующий объект модуля, а не имя.

- В 3.0 ф-ция reload находится в модуле imp, который нужно импортировать, чтобы получить доступ к ф-ции.

Ф-ция reload ожидает получить объект, поэтмоу к моменту её вызова модуль должен быть импортирован. К тому же синтаксис отличается от import, пример:

import modul # Первоначальное импортирование

...используются атрибуты модуля...

... # Теперь выполняются изменения в файле модуля

...

from imp import reload # Импортируем ф-цию reload (в 3.0)

reload(module) # Загрузить обновленный модуль

...используются атрибуты модуля...

Это типичная ситуация, напр. когда мы изменяем код загруженного модуля и хотим его перезагрузить.

Ф-ция reload - изменяет непосредственно сам объект модуля, она не удаляет и не создает его повторно. Т.е. все ссылки, имеющиеся в программе, автоматически будут учитывать изменения, произошедшие после в ходе повторной загрузки.

Подробнее о ходе повторной загрузки:

- **Ф-ция reload запускает новый программный код в файле модуля в текущем пространстве имен модуля.** Текущее пространство имен перезаписывается.

- **Инструкции присваивания на верхнем уровне файла замещают имена новыми значениями.** Повторный запуск, например def, замещает предыдущую версию ф-ции в пространстве имен модуля, т.к. выполняет повторную операцию присваивания имени ф-ции.

- **Повторная загрузка оказывает воздействие на всех клиентов, использовавших инструкцию import для получения доступа к модулю.** После повторной загрузки клиенты будут получать новые значения атрибутов.

- **Повторная загрузка будет воздействовать лишь на тех клиентов, которые еще только будут использовать инструкцию from в будущем.** Клиенты, которые использовали from для получения доступа к атрибутам в прошлом, не заметят изменений, после повторной загрузки - они будут ссылаться на старые объекты, полученные до выполнения перезагрузки.

**Пример использования reload.**

Создадим модуль changer.py:

message = "First version" # Имя связано со строкой

def printer(): # А это имя связано с ф-цией

print(message)

Импортируем модуль и вызовем ф-цию:

>>> import changer

>>> changer.printer()

First version

А теперь немного изменим файл модуля:

message = "After editing"

def printer():

print('reloaded: ', message)

Попробуем снова импортировать модуль и затем перезагрузить его:

>>> import changer

>>> changer.printer() # Используется прежняя версия модуля!

First version

>>> from imp import reload

>>> reload(changer) # Принудительная загрузка/выполнение нового кода

<module 'changer' from 'C:\\Python34\\changer.py'>

>>> changer.printer() # Теперь заработала новая версия кода

reloaded: After editing

Как видим, ф-ция reload, на самом деле, возвращает объект, поэтому интерпретатор и вывел строку <module 'changer' from 'C:\\Python34\\changer.py'>.

**Повторная загрузка модуля.**

Повторная загрузка может использ-ся и в крупных системах, где стоимость перезапуска всего приложения слишком высока, например системы соединенные с серверами сети.

Также эта возможность использ-ся в приложениях с графич. интерфейсом (напр. обработчики событий). И при использ-ии Питон в качестве встроенного языка в прогах на С или С++.

Обычно повторная загрузка позволяет реализовывать высокодинамичные интерфейсы. Напр. Питон часто использ-ся для настройки больших систем (можно изменять код без перекомпиляции всей системы).

Также использ-ся для создания сценариев настройки в крупных системах.

**Глава 23. Пакеты модулей.** Стр. 641

Помимо импортирования имени модуля есть возможность импортировать имена каталогов. Каталог, в Питон, называется - пакетом, поэтому такой импорт наз-ся импортированием пакетов.

Импортирование пакета превращает имя каталога в разновидность пространства имен, в котором атрибуты это подкаталоги и файлы модулей. Эта иерархическая структура оказывается удобной для организации файлов в крупных системах.

**Основы операции импортирования пакетов.**

Вместо имени файла, указываем список имен в пути к каталогу, разделяя их точкой:

import dir1.dir2.mod

То же самое и from:

from dir1.dir2.mod import x

***Внимание!***  Как видим мы импортируем все-таки файл (mod.py или mod.so иди mod.pyc, не важно), а не каталог, в котором куча модулей, как я подумал вначале.

Такой вот путь (dir1.dir2) должен вести к файлу mod.py (или любое др. соответствующее расширение). Т.е. у нас есть каталог dir1, в котором каталог dir2, в которм файл mod.py.

Кроме того предполагается, что dir1 находится внутри некоего контейнерного каталога dir0, который нах-ся в пути поиска модулей. Т.е. структура каталогов выглядит примерно так:

dir0\dir1\dir2\mod.py

dir0 нужно будет добавить в путь поиска модулей (конечно если это не домашний каталог глав. файла программы), как если бы dir1 было именем модуля.

**Пакеты и настройка пути поиска.**

Итак, вместо того, чтобы использовать ошибочный синтаксис:

import C:\mycode\dir1\dir2\mod # Ошибка: недопустимый синтаксис

Мы добавим путь C:\mycode в переменную окружения PYTHONPATH или в файл .pth, и используем такую инструкцию:

import dir1.dir2.mod

Для этого я создал переменную PYTHONPATH в свойствах системы (в Виндовс) и вписал в неё путь, правда пришлось еще перезагрузиться.

**Файлы \_\_init\_\_.py пакетов.**

Условие импорта пакетов: каждый каталог в пути, указанном в инструкции импортирования пакета, должен содержать файл с именем \_\_init\_\_.py, иначе операция импорта будет неудачна. Они могут не содержать никакой код, это некий защитный механизм Питон.

Т.е. dir1 и dir2 должны содержать файл \_\_init\_\_.py, а dir0 может не содержать и второе правило - dir0, но не dir0\dir1 должен быть прописан в каком либо из путей поиска модулей.

***Внимание!*** Когда я настроил переменную PYTHONPATH у меня заработало импортирование каталогов и без этих файлов, модуль mod.py импортировался и выполнился:

X = 88

def f():

global X

X = 99

print("X равно: ", X)

>>> import dir1.dir2.mod

X равно: 88

- но наверное все-равно создадим файлы \_\_init\_\_.py где нужно.

В общем случае файл \_\_init\_\_.py предназначен для выполнения действий по инициализации пакета, создания пространства имен для каталога и реализации поведения инструкций from \* (т.е. для from... import \*), когда они использ-ся для импортирования каталогов:

***Инициализация пакета.*** Когда Питон импортирует каталог в первый раз, он автоматически запускает программный код файла \_\_init\_\_.py этого каталога. Поэтому, обычно, в эти файлы помещается код, выполняющий действия по инициализации, необходимые для файлов в пакете. Обычно файлы \_\_init\_\_.py не предназначены для непосредственного выполнения - они запускаются автоматически, когда выполняется первое обращение к пакету.

***Инициализация пространства имен модуля.*** При импортировании пакетов, пути к каталогам превр-ся в настоящие иерархии вложенных объектов. Напр. наш пример модуля путь dir1.dir2, импорт возвратит объект модуля, чье пространство имен будет содержать все имена, определяемые файлом \_\_init\_\_.py из каталога dir2.

***Поведение инструкции from \*.*** В файлах \_\_init\_\_.py можно использовать списки \_\_all\_\_, чтобы определить, что будет импортироваться из каталога инструкцией from \*. Список \_\_all\_\_ в файлах \_\_init\_\_.py представляет собой список имен субмодулей, которые должны импортироваться, когда в инструкции from \* указывается имя пакета (каталога). Если \_\_all\_\_ отсутствует, from \* не будет автоматически загружать субмодули, вложенные в каталог, - она загрузит только имена, определяемые инструкциями присваивания в файле \_\_init\_\_.py, включая любые субмодули, явно импортируемые программным кодом в этом файле.

И не путайте файлы \_\_init\_\_.py пакетов с методами-конструкторами \_\_init\_\_ классов.

**Пример импортирования пакета.** Стр. 645

Создадим файлы в каталогах:

# Файл: dir1\\_\_init\_\_.py

print('dir1 init')

x = 1

# Файл: dir1\dir2\\_\_init\_\_.py

print('dir2 init')

y = 2

# Файл: dir1\dir2\mod.py

print('in mod.py')

z = 3

Напоминаем - в родительской папке C:\mycode файл \_\_init\_\_.py не нужен.

Итак, import выполняет файлы инициализации в каждом каталоге, а print поможет их отследить:

>>> import dir1.dir2.mod

dir1 init

dir2 init

in mod.py

>>>

>>> import dir1.dir2.mod # Повторное импортирование не выполняется

>>>

>>> from imp import reload

>>> reload(dir1)

dir1 init

<module 'dir1' from 'C:\\mycode\\dir1\\\_\_init\_\_.py'>

>>>

>>> reload(dir1.dir2) # reload также принимает цепочку имен через точку

dir2 init

<module 'dir1.dir2' from 'C:\\mycode\\dir1\\dir2\\\_\_init\_\_.py'>

После операции импорта, путь, указанный в import, становится цепочкой вложенных объектов. Здесь mod - это объект, вложенный в объект dir2, который вложен в dir1:

>>> dir1

<module 'dir1' from 'C:\\mycode\\dir1\\\_\_init\_\_.py'>

>>> dir1.dir2

<module 'dir1.dir2' from 'C:\\mycode\\dir1\\dir2\\\_\_init\_\_.py'>

>>> dir1.dir2.mod

<module 'dir1.dir2.mod' from 'C:\\mycode\\dir1\\dir2\\mod.py'>

Каждый каталог в пути, фактически становится переменной, которой присваивается объект модуля, пространство имен которого инициализируется всеми инструкциями присваивания в файле \_\_init\_\_.py, находящемся в этом каталоге.

Имя dir1.x ссылается на переменную х в файле dir1\\_\_init\_\_.py, и остальные точно так же:

>>> dir1.x

1

>>> dir1.dir2.y

2

>>> dir1.dir2.mod.z

3

**Инструкции from и import для пакетов.**

import может быть несколько неудобной для импортирвоания пакетов, т.к. в таком случае нам придется вводить полные пути к именам в программе. Напр. мы не можем написать коротко из пред. примера, получим ошибку т.к. нужно писать полный путь:

>>> dir2.mod

NameError: name 'dir2' is not defined

>>> mod.z

NameError: name 'mod' is not defined

Поэтому при импорте пакетов удобно использовать from. А если, например, произойдет реструктуризации дерева каталогов, то from достаточно изменить нужную инструкцию, а не полностью перезапускать моудль.

Кстати есть также расширение import as, определяющие синонимы для длинных путей.

>>> from dir1.dir2 import mod

dir1 init

dir2 init

in mod.py

>>>

>>> mod.z # Благодаря from указывать полный путь не требуется

3

>>> z

NameError: name 'z' is not defined

>>> from dir1.dir2.mod import z # Теперь тоже самое для z

>>> z

3

>>> import dir1.dir2.mod as mod # А вот и использование коротких синонимов

>>> mod.z

3

**Когда используется операция импортирования пакетов?**

Конечно перед использованием пакетов нужно научиться работать с простыми модулями. Пакеты тоже полезны, особенно в крупных прогах, с нмии более информативна операция импорта, это и организационный инструмент, упрощение поиска файлов модулей и т.д.

Если мы организовали дерево каталогов по функциональным признакам, то импорт пакетов выделяет роль пакета и повышает удобочитаемость кода. Например сравним:

import utilites # Обычная операция импорта

import database.client.utilites

- эта операция импорта несет больше информации чем обычная.

Также плюсом явл-ся запись единой корневой папки dir0 - мы знаем точно где файл и куда поместить новый модуль.

**История о трех программах.**

Одна из основных ролей импорта пакетов - когда на ПК установлено много программ имеющими файлы с одинаковыми именами. Рассмотрим теоретический пример.

Пусть есть программа, которая содержит файл utilites.py, с доп. кодом и файл main.py, для запуска проги. Все файлы проги вызывают инструкцию import utilites для загрузки и использования общего кода.

Программа рспространяется в фархиве .tar или .zip, и при установке все файлы распаковываются в каталог system1:

system1\

utilites.py # Общие ф-ции, классы и т.п.

mail.py # Этот файл запускает прогу

other.py # Импортирует и использует модуль utilites

Теперь другая прога, в которой также есть utilites.py и main.py и инструкция import utilites во всех файлах. Во время её установки создается каталог system2:

system2\

utilites.py # Общие ф-ции, классы и т.п.

mail.py # Этот файл запускает прогу

other.py # Импортирует и использует модуль utilites

Пока никаких проблем нет, всё будет работать, т.к. поиск файлов модулей будет начинаться с домашней папки каждого исполняемого файла main.py и каждая прога будет использовать свои файлы utilites.py.

Но предположим, что мы решили использ-ть ф-ции из обоих файлов utilites.py в своей проге (третья программа).

В такой ситуации нам необходима возможность из своего кода, в третьем каталоге, загрузить один из двух файлов:

import utilites

utilites.func('spam')

А вот, чтобы это работало нужно будет включить в путь поиска модулей, оба каталога с utilites. Но какой поместить первым?

Проблема в линейной природе пути поиска - он всегда идет слева направо. Т.е. мы всегда будем получать utilites.py из каталога который левее в пути поиска.

Эта проблема как раз и решается с помощью пакетов. Ведь мы можем установить файлы utilites.py в подкаталоги с общим корнем. Определим новую иерархию программного кода:

root\

system1\

\_\_init\_\_.py

utilites.py

main.py

other.py

system2\

\_\_init\_\_.py

utilites.py

main.py

other.py

system3\ # Вот и наша (третья) программа

\_\_init\_\_.py

myfile.py

А теперь мы можем использовать даже оба файла сразу! С учетом полного пути конечно:

import system1.utilites

import system2.utilites

system1.utilites.function('spam')

system2.utilites.function('eggs')

Если имена файлов различаются, то лучше, конечно, использовать from, дабы не писать все пути полностью в коде.

Кстати, каталог system3 не обязательно должен быть в каталоге root, здесь должны быть только импортируемые каталоги. Но с прицелом на будущее так все-таки надежнее.

**Импортирование относительно пакета.**

До этого времени мы использовали ипорт пакетов из-за пределов этих пакетов. Но можно использовать спец. правила поиска модулей внутри пакета, упрощающие инструкцию import. Т.е. вместо полного пути указывать относительный.

Правила такого импортирования разнятся в разных версиях Питона.

**Изменения в Питон 3.0**

Изменения в операции импортирования внутри пакетов:

- Изменилась семантика пути поиска модулей, теперь импорт модуля по умолч. пропускает собственный каталог пакета. Она проверяет только компоненты пути поиска. Эта операция на-ся импортированием по "абсолютному" пути.

- Расширен синтаксис инструкции from, теперь можно явно указывать что поиск импортиуремых модулей должен идти только в каталоге пакета. Эта операция наз-ся импортированием по "относительному" пути.

Суть этих изменений в том, что мы должны использовать спец. синтаксис from для импорта модулей, находящихся в том же пакете, что и импортирующий модуль, если не указываете полный путь к модулю. Если этот синтаксис не использ-ть, интерпретатор может не найти модуль в пакете.

**Основы импортирования по относительному пути.**

В from можно использ-ть точки, в пути поиска модулей, чтобы указать поиск в том же самом пакете (импортирвоание относительно пакета), а не в др. месте, в пути поиска (импортирование по абсолютному пути):

- Точки в инструкции from могут указывать проводить импорт относительно вмещающего пакета - в таком случае поиск будет идти только внутри пакета, а модули находящиеся в пути поиска (sys.path) буду тнедоступны.

- В 2.6 обычный импорт (без точек) по умолч. выполняется так - сначала поиск относительно пакета, потом аюсолютный поиск. Но в 3.0 по умолч. поиск в импорте иде тсначала по абсолютному пути, т.е.е когда точек нет идет поиск в sys.path.

Напр. в обоих версиях инструкция вида:

from . import spam # Импорт относительно текущего пакета

- дает задание интерпретатору импортировать модуль spam, находящийся в том же пакете, что и файл, где нах-ся эта инструкция.

След. инструкция:

from .spam import name

- означает "из модуля с именем spam, расположенного в том же пакете, что и файл с инструкуцией, импортировать переменную name".

В 2.6 импорт без точки идет по пути "сначала относительно пакета, потом - по абсолютному пути", если только в импортирующем файле не будет инструкции:

from \_\_future\_\_ import absolute\_import # Включает поиск по абсолютному пути

В 3.0 импорт без точки идет по абсолютному пути (sys.path):

import string # Пропустит поиск модуля в пакете

from . import string # Поиск пойдет в пределах пакета

Ведущий символ точки может использ-ся только в инструкции from, но не в import.

В 3.0 инструкция import modname всегда выполняет импортирование по абсолютному пути, пропуская поиск в каталоге.

Приведем еще неск-ко примеров с каталогом mypkg:

from .string import name1, name2 # импорт имен из mypkg.string

from . import string # импорт mypkg.string

from .. import string # импорт string из родительского каталога

**Зачем необходим импорт относительно пакета?**

Эта возможность дана, чтобы дать сценариям возможность ликвидировать неоднозначности, которые возникают когда в разных местах в пути поиска присутствует неск-ко одноименных модулей. Пример:

mypkg\

\_\_init\_\_.py

main.py

string.py

Это каталог пакета с именем mypkg, содержащий модули mypkg.main и mypkg.string. Если модуль main попытается импортировать модуль с именме string, в 2.6, например, то поиск будет идти относительно пакета и импортируется файл string.py и присвоится имя string из пространства имен модуля mypkg.main.

А если нам нужен стандартный модуль string? В 2.6 это будет трудной задачей..

Единственно, это постараться не давать своим файлам имена стандартных модулей.

**Решение проблемы с импортированием относительно пакета в 3.0**

Итак, произошли изменения и теперь, след. инструкция import, нашего файла mypkg\main.py, всегда будет находить модуль string за пределами пакета, за счет абсолютного пути sys.path:

import string # Импортирует модуль string за пределами пакета

Инструкция from, в которой нет точки, также выполняет поиск по абсолютному пути:

from string import name # импортирует имя name из модуля string за пределами пакета

А если же мы хотим импортировать модуль из своего пакета (не указывая полный путь), можно использ. from:

from . import string # Импортирует mypkg.string (относительно пакета)

Кроме того, можно копировать имена из модуля, синтаксисом импортирования относительно пакета:

from .string import name1, name2 # Импортирует имена из mypkg.string

- эта инструкция также ссылается на модуль string в текущем пакете. Если её поместить в модуль mypkg.main, напр., она будет импортировать имена name1 и name2 из модуля mypkg.string.

По сути символ точки "." в импорте - это каталог в пакете, содержащий файл, где выполняется операция импортирования. Доп. точка означает выполнить относительный импорт, начиная с родительского каталога текущего пакета:

from .. import spam # Импортирует модуль одного уровня с пакетом mypkg

- эта инструкция загрузит модуль, находящийся на том же уровне иерархии, что и пакет mypkg, - т.е. модуль spam, находящийся в каталоге, родительском по отношению к пакету mypkg.

from . import D # Импортирует A.B.D (. означает A.B)

from .. import E # Импортирует A.E (.. означает A)

from .D import X # Импортирует A.B.D.X (. означает A.B)

from ..E import X # Импортирует A.E.X (.. означает A)

**Импорт в пакетах по относительному и абсолютному пути.**

Как вариант, в файле можно явно указать имя его пакета в импорте по абсолютному пути. Пример, пакет mypkg найдется по абсолютному пути sys.path:

from mypkg import string

Но это зависит от настроек и порядка каталогов в пути поиска. Нужно включить путь к каталогу пакета mypkg в путь поиска модулей.

from system.section.mypkg import string

- а эта инструкция просматривает все каталоги левее каталога пакета в sys.path, если имя пакета указывается явно (как в примере).

Если пакет большой или глубокая вложенность, то такой импорт буде твыполняться дольше чем с точкой:

from . import string # Ведет поиск только в объемлющем пакете

**Правила импортирования по относительному пути.**

Опишем ключевые моменты:

- ***Операция импортирования по относительному пути применяется исключительно для импортирования внутри пакета.*** Но это только при импорте внутри модуля, помещенного в пакет. Обычный импорт в файлах за пределами пакетов сначала выполняет поиск в каталоге, содержащем главный файл сценария.

- ***Импортирование по относительному пути возможно только с помощью инструкции from.*** И считается недопустимым в инструкции import (см. примеры с точками).

***- Неоднозначность терминологии.*** На самом деле все операции импорта выполняются относительно чего-нибудь. За пределами пакетов импорт идет относительно каталогов в пути sys.path (т.е. домашний каталог, значение PYTHONPATH, каталоги файлов .pth и стандарт. библиотеки).

Импорт внутри пакета в 2.6 выполняет поиск модуля сначала внутри самого пакета. В 3.0 - обычный импорт по "абсолютному" пути пропускает каталог пакета, импорт со "специальным" синтаксисом относительн отекущего пакета выполняют поиск только в пределах пакета.

**Правила поиска модулей.**

Правила поиска в 3.0:

- Для простых имен пакетов (напр. А) поиск выполняется во всех каталогах, перечисленных в списке sys.path, слева направо.

- Пакеты - это обычные каталоги с модулями на языке Питон, содержащие спец. файл \_\_init\_\_.py, который дает указывать цепочки каталогов вида A.B.C. Чтобы импортировать A.B.C, каталог А должен нах-ся в одном из каталогов sys.path, B должен быть подкаталогом пакета в каталоге А, а С должен быть модулем или др. компонентом в каталоге B.

- Внутри файлов пакета обычные инструкции import выполняют поиск модулей в sys.path по правилам импорта в любых др. модулях. Но при импорте с from и начальных точек в именах поиск идет относительно текущего пакета - т.е. только в каталоге текущего пакета, а поиск в sys.path не выполняется.

**Относительный импорт в примерах.** стр. 657

**Импортирование за пределами пакетов.**

>>> import string # Загружает модуль string из стандарт. библиотеки

>>> string

<module 'string' from 'C:\\Python34\\lib\\string.py'>

Но если в текущий рабочий каталог добавить одноименный модуль, будет загружен именно он, т.к. рабочий каталог стоит на первом месте в поиске:

#D:\test\string.py

print('string' \* 8)

***- Внимание!*** Пришлось создать доп. каталог D:\test и уже в нем модуль string.py, т.к. при попытке создать модуль string.py в папке C:\Python34 (как мы обычно делали) я получал ошибку от IDLE. Видимо в корневой папке нельзя создавать.

А теперь запускаем командную строку:

>>> import string

stringstringstringstringstringstringstringstring

>>> string

<module 'string' from 'D:\\test\\string.py'>

Другими словами, обычный импорт выполняется относительно "домашнего" каталога (где нах-ся глав. сценарий или относительно текущего рабочего каталога). Синтаксис импорта относительно текущего пакета считается недопустимым для использ. в файлах, не входящих в состав пакета:

>>> from . import string

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

SystemError: Parent module '' not loaded, cannot perform relative import

- Замечание! В книге ошибка другого вида. Я пытался из разных папок её вызывать и в IDLE, текст везде один и тот же, в книге текст ошибки совсем другой да и тип тоже - ValueError: Attempted relative import in non-package.

И еще один небольшой пример, как выполняется импортирование когда на первом месте sys.path не пустая строка, а абсолютный путь к каталогу:

#test\main.py

import string

print(string)

Запускаем этот сценарий именно так как написано:

D:\test>python main.py

stringstringstringstringstringstringstringstring

<module 'string' from 'D:\\test\\string.py'>

**Импортирование внутри пакетов.**

Теперь удалим локальный модуль string и создадим каталог пакета (pkg) с двумя модулями, включая пустой \_init\_\_.py:

# test\pkg\spam.py

import eggs

print(eggs.X)

# test\pkg\eggs.py

X = 99999

import string

print(string)

Чтобы операция импорта в spam.py сработала в 2.6 и 3.0 (сейчас она не будет работать в 3.0) нужно в первом файле использовать спец. синтаксис операции импорта относительно текущего пакета (а eggs.py останется тем же):

# test\pkg\spam.py

from . import eggs

print(eggs.X)

D:\test>python

>>> import pkg.spam

<module 'string' from 'C:\\Python34\\lib\\string.py'>

99999

- чтобы эта команда работала мне пришлось в переменную окружения PYHONPATH добавить путь к каталогу D:\test и, после перезагрузки ПК, всё заработало.

**Импорт по-прежнему выполняется относительно текущего рабочего каталога.**

Видим, что модули пакета по-прежнему могут импортировать модули из стандарт. библиотеки, такие как string.

Если мы снова добавим в каталог D:\test свой модуль string, то импорт будет обнарживать его, а не модуль в стандарт. библиотеке. В 3.0 можно пропустить каталог пакета при импорте, но невозможно пропустить домашний каталог программы, которая импортирует пакет (внимательно смотрите откуда мы запускаем интерактивный сеанс):

# test\string.py

print('string'\*8)

# test\pkg\spam.py

from . import eggs

print(eggs.X)

# test\pkg\eggs.py

X = 99999

import string

print(string)

D:\test>python

>>> import pkg.spam

stringstringstringstringstringstringstringstring

<module 'string' from 'D:\\test\\string.py'>

99999

**Выбор модулей операциями импортирования по относительному и абсолютному пути.**

Если теперь мы удалим наш string из каталога D:\test и поместим его в D:\test\pkg:

# test\pkg\spam.py

import string

print(string)

# test\pkg\string.py

print('Ni'\*8)

В Питон 3.0 импортирование в первом файле (import pkg.spam) будет идти по абсолютному пути и пропустит каталог пакета (т.е. импортирует стандарт. модуль string), а в 2.6 нет - там импортируется string из test\pgk.

Использование импортирования относительно текущего пакета вынудит 3.0 выполнить поиск в пакете, как это делается в 2.6:

# test\pkg\spam.py

from . import string

print(string)

# test\pkg\string.py

print('Ni'\*8)

D:\test>python

>>> import pkg.spam

NiNiNiNiNiNiNiNi

<module 'string' from 'D:\\test\\pgk\\string.py'>

Если же мы сейчас удалим файл string.py, то операция импорта будет вызывать ошибку, т.к. одноименный модуль в стандарт. библиотеке не будет находиться (да и в любом др. месте тоже).

Модули, которые упоминаются в инструкциях импорта относительн отекущего пакета, должны существовать в каталоге пакета.

**Импорт по-прежнему выполняется относительно текущего рабочего каталога (еще раз).**

Импорт по абсолютному пути позволяет пропускать модули пакета, но они по-прежнему зависят от др. элементов списка sys.path.

В этом примере мы создадим два модуля string:

# test\string.py

print('string'\*8)

# test\pkg\spam.py

from . import string

print(string)

# test\pkg\string.py

print('Ni'\*8)

Итак, когда мы импортируем относительно пакета, получаем версию модуля из пакета (в 2.6 и в 3.0 одно и то же):

>>> import pkg.spam

NiNiNiNiNiNiNiNi

<module 'string' from 'D:\\test\\pgk\\string.py'>

Но, когда импорт идет по абсолютному пути, версия модуля зависит от версии интерпретатора. Например в 3.0 интерпретатор пропускае каталог пакета и импортирует модуль из текущего рабочего каталога, а 2.6 интерпретирует эту инструкцию как импорт относительно пакета:

# test\pkg\string.py

print('string'\*8)

# test\pkg\spam.py

import string

print(string)

# test\pkg\string.py

print('Ni'\*8)

Python 2.6

>>> import pkg.spam

NiNiNiNiNiNiNiNi

<module 'string' from 'D:\\test\\pgk\\string.py'>

Python 3.0

>>> import pkg.spam

stringstringstringstringstringstringstringstring

<module 'string' from 'D:\\test\\string.py'>

Помните, что импорт относительно пакетов не всегда лучший выбор. Иногда путь sys.path предпочтительнее.

**Пакеты модулей.**

Сейчас можно встретить крупные расширения сторонних разработчиков, распространяемые не как плоский список модулей, а как набор каталогов с пакетами. Напр. пакет win32all для Питон от Виндовс.

Многие модули этого пакета импортируются посредством указания пути. Например загрузка средств работы с технологией COM:

from win32com.client import constants, Dispatch

Импортирование пакетов использ-ся и в Jython (Питон на Java).

**Глава 24. Дополнительные возможности модулей.** стр. 665

**Сокрытие данных в модулях.**

Как мы знаем модули экспортируют все имена, которым присвоены значения на верхнем уровне. И фактически нет способов предотвратить возможность изменения имен в модуле извне.

Сокрытие данных, в Питоне, регулируется соглашениями, а не синтаксическими инструкциями. Конечно нам ничто не мешает повредить модуль, изменяя имена в нем, но это не приветствуется.

Кто-то может сказать, что нужно использ. инкапсуляцию, но в Питон инкапсуляция скорее относится к организации пакетов, чем к накладыванию ограничений.

**Минимизация повреждений, причиняемых инструкцией from \*: \_X и \_\_all\_\_**

Можно начинать имена с одного символа подчеркивания (\_Х), чтобы защитить их от переписывания, когда выполняется импорт инструкцией from \*. На самом деле это только для минимизации загрязнения пространства имен. Подчеркивание это не объявление "частных данных", мы по-прежнему можем видеть эти имена и изменять их с помощью др. форм импортирования, как import напр.

Альтернатива сокрытия данных - присвоение, на верх. уровне модуля, переменной \_\_all\_\_ списка строк с иенами переменных:

\_\_all\_\_ = ["Error", "encode", "decode"]

При этом from \* будет копировать только имена, перечисленные в списке \_\_all\_\_. Это соглашение обратное \_Х: переменная \_\_all\_\_ идентифицирует имена, доступные для копирования, тогда как соглашение \_Х идентифицирует имена, недоступные для копирования. Питон сначала отыскивает список \_\_all\_\_ в модуле, и если он отсутствует, from \* копирует все имена, которые не начинаются с единственного символа подчеркивания.

И \_Х и \_\_all\_\_ имеют смысл только для инструкции from \* и не явл-ся объявлением частных данных. Можно использ-ть любой приём.

**Включение будущих возможностей языка.**

В языке периодически появл-ся изменения, которые могут повлиять на существующий код. Обычно они появл-ся в виде расширений, которые отключены по умолч. Чтобы такие расширения включить использ-ся спец. импорт:

from \_\_future\_\_ import имя\_функциональной\_особенности

Эта строка должна появл-ся в самом начале модуля, т.к. она включает спец. режим компиляции кода для каждого отдельно взятого модуля. Можно выполнить её и в интерактив. сеансе.

Мы уже включали её когда изучали генераторы, истинное деление, ф-цию print и импорт пакетов по абсолютному пути.

Эти изменения вводятся так осторожно потому что могут сказаться на работоспособности кода для Питно 2.6.

**Смешанные режимы использования: \_\_name\_\_ и \_\_main\_\_**

Рассмотрим прием, который импортирует файл как модули и запускает их как самостоятельные программы. Каждый модуль обладает атрибутом \_\_name\_\_, который устанавливается след. образом:

- Если файл запускается как глав. файл проги, атрибуту \_\_name\_\_ на запуске присва-ся значение "\_\_main\_\_".

- Если файл импортируется, атрибуту \_\_name\_\_ присв-ся имя модуля, под которым он будет известен клиенту.

Поэтому модуль может проверить свой атрибут \_\_name\_\_ и узнать, запущен ли он как программа или импортирован др. модулем.

Пример с модулем runme.py, который экспортирует ф-цию tester:

def tester(): # Этот модуль определяет ф-цию для клиентов

print("It's Christmas in Heaven...")

if \_\_name\_\_ == '\_\_main\_\_': # Только когда модуль запускается,

tester() # а не импортируется - условие вызова tester()

Импортируем модуль в IDLE:

>>> import runme

>>> runme.tester() # Ф-ция запустилась только при её вызове

It's Christmas in Heaven...

А теперь запустим модуль:

C:\Python34>python runme.py # Запустили модуль как самостоятельную программу

It's Christmas in Heaven... #Ф-ция запустилась по условию if

:cal SetSyn("python")

Т.о. переменная \_\_name\_\_ может играть роль флага. Мы будем часто встречать этот прием.

Чаще всего проверка \_\_name\_\_ выполняется в коде для самопроверки модуля. Т.е. в конец модуля код, проверяющий экспортируемые элементы (по условию if напр.) используя \_\_main\_\_. Вообще это одна из основ модульного тестирования Питон.

Прием на основе проверки \_\_name\_\_ част оиспольз-ся при создании файлов, которые могут использ-ся и как утилиты командной строки, и как библиотеки инструментов. Напр. мы пишем сценарий поиска файлов. Будет больше пользы, если реализовать его в виде ф-ций и добавить проверку \_\_name\_\_ для организации вызова этих ф-ций, когда файл запускается как самостоятельная прога.

**Тестирование модулей с помощью \_\_name\_\_**

Вспомним сценарий, который находил мин. значение из множества аргументов, файл min.py:

def minmax(test, \*args):

res = args[0]

for arg in args[1:]:

if test(arg, res):

res = arg

return res

def lessthan(x, y): return x < y

def grtthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3)) # Код самопроверки

print(minmax(grtthan, 4, 2, 1, 5, 6, 3))

>>> import min

1

6

Чтобы результаты самопроверки не выводились каждый раз на экран, при импорте модуля, нужно обернуть проверочные вызовы ф-ций в условную конструкцию, проверяющую атрибут \_\_name\_\_, так чтобы они выполнялись, только когда файл запускается как самостоятельная программа, а не при импорте:

print('I am: ', \_\_name\_\_)

def minmax(test, \*args):

res = args[0]

for arg in args[1:]:

if test(arg, res):

res = arg

return res

def lessthan(x, y): return x < y

def grtthan(x, y): return x > y

if \_\_name\_\_ == '\_\_main\_\_':

print(minmax(lessthan, 4, 2, 1, 5, 6, 3)) # Код самопроверки

print(minmax(grtthan, 4, 2, 1, 5, 6, 3))

Здесь, вначале, мы выводим атрибут \_\_name\_\_, чтобы проверить его визуально. Интерпретатор создает эту переменную и присваивает ей значение во время загрузки файла. Затем, когда файл запускается как программа, этому имени присв-ся значение '\_\_main\_\_', и смотрим выполнение:

user@home:~$ python3.4 min.py # Запустил из терминала

I am: \_\_main\_\_

1

6

- как видим код проверки выполняется, т.к. модуль запущен как прога.

>>> import min # А это запустил из IDLE, для удобства

I am: min

>>> min.minmax(min.lessthan, 12, 13, 15, 16)

12

- а вот результат вывода при импорте модуля.

**Обработка аргументов командной строки с помощью \_\_name\_\_**

Рассмотрим модуль formats.py, который определяет вспомогательные ф-ции форматирования строк. Кроме этого проверяет имя модуля, чтобы узнать запущен ли он как прога, и если это так, он проверяет и использует аргументы командной строки для запуска встроенного или конкретного текста.

Список sys.argv в Питон содержит аргументы командной строки - список строк со словами, введенными в командной строке, где первый элемент всегда содержит имя файла сценария:

"""

Различные спец. ф-ции форматирования строк.

Модуль можно протестировать с пом. тестов или посредством передачи аргументов командной строки.

"""

def commas(N):

"""

Форматирует целое положительное число, добавляя запятые,

разделяющие группы разрядов: xxx, yyy, zzz

"""

digits = str(N)

assert(digits.isdigit())

result = ''

while digits:

digits, last3 = digits[:-3], digits[-3:]

result = (last3 + ',' + result) if result else last3

return result

def money(N, width=0):

"""

Форматирует число N, добавляя запятые, оставляя 2 десятичных знака

в дробной части, добавляя в начало символ $ и знак числа, и, при необходимости,

- отступ: $ -xxx, yyy.zz

"""

sign = '-' if N < 0 else ''

N = abs(N)

whole = commas(int(N))

fract = ('%.2f' % N)[-2:]

format = '%s%s.%s' % (sign, whole, fract)

return '$%\*s' % (width, format)

if \_\_name\_\_ == '\_\_main\_\_':

def selftest():

tests = 0, 1

tests += 12, 123, 1234, 12345, 123456, 1234567

tests += 2\*\*32, 2\*\*100

for test in tests:

print(commas(test))

print('')

tests = 0, 1, -1, 1.23, 1., 1.2, 3.14159

tests += 12.34, 12.344, 12.345, 12.346

tests += 2\*\*32, (2\*\*32 + .2345)

tests += 1.2345, 1.2, 0.2345

tests += -1.2345, -1.2, -0.2345

tests += -(2\*\*32), -(2\*\*32 + .2345)

tests += (2\*\*100), -(2\*\*100)

for test in tests:

print('%s [%s]' % (money(test, 17), test))

import sys

if len(sys.argv) == 1:

selftest()

else:

print(money(float(sys.argv[1]), int(sys.argv[2])))

Этот модуль одинаково работает и в 2.6 и в 3.0. Для управл-ия тестированием он использует аргументы командной строки. Чтобы протестить определенные строки, передайте их в командной строке вместе со значением минимальной ширины поля вывода:

user@home:~$ python3.4 formats.py 999999999 0

$999,999,999.00

user@home:~$ python3.4 formats.py -999999999 0

$-999,999,999.00

user@home:~$ python3.4 formats.py 123456789012345 0

$123,456,789,012,345.00

user@home:~$ python3.4 formats.py -123456789012345 0

$-123,456,789,012,345.00

user@home:~$ python3.4 formats.py 123.456 0

$123.46

user@home:~$ python3.4 formats.py -123.456 0

$-123.46

А сейчас посмотрим работу встроенных тестов, просто запустив модуль:

user@home:~$ python3.4 formats.py

0

1

12

123

1,234

12,345

123,456

1,234,567

4,294,967,296

1,267,650,600,228,229,401,496,703,205,376

$ 0.00 [0]

$ 1.00 [1]

$ -1.00 [-1]

$ 1.23 [1.23]

$ 1.00 [1.0]

$ 1.20 [1.2]

$ 3.14 [3.14159]

$ 12.34 [12.34]

$ 12.34 [12.344]

$ 12.35 [12.345]

$ 12.35 [12.346]

$ 4,294,967,296.00 [4294967296]

$ 4,294,967,296.23 [4294967296.2345]

$ 1.23 [1.2345]

$ 1.20 [1.2]

$ 0.23 [0.2345]

$ -1.23 [-1.2345]

$ -1.20 [-1.2]

$ -0.23 [-0.2345]

$-4,294,967,296.00 [-4294967296]

$-4,294,967,296.23 [-4294967296.2345]

$1,267,650,600,228,229,401,496,703,205,376.00 [1267650600228229401496703205376]

$-1,267,650,600,228,229,401,496,703,205,376.00 [-1267650600228229401496703205376]

Также этот модуль может импортироваться и можно импортировать его ф-ции:

>>> from formats import money, commas

>>> money(123.456)

'$123.46'

>>> money(-9999999.99, 15)

'$ -9,999,999.99'

>>> X = 9999999999999999999

>>> '%s (%s)' % (commas(X), X)

'9,999,999,999,999,999,999 (9999999999999999999)'

Также мы можем исследовать инструменты модуля, с помощью ф-ции help и использования строк документирования, которые есть в файле:

>>> import formats

>>> help(formats)

Help on module formats:

NAME

formats

DESCRIPTION

Различные спец. ф-ции форматирования строк.

Модуль можно протестировать с пом. тестов или посредством передачи аргументов командной строки.

FUNCTIONS

commas(N)

Форматирует целое положительное число, добавляя запятые,

разделяющие группы разрядов: xxx, yyy, zzz

money(N, width=0)

Форматирует число N, добавляя запятые, оставляя 2 десятичных знака

в дробной части, добавляя в начало символ $ и знак числа, и, при необходимости,

- отступ: $ -xxx, yyy.zz

FILE

/home/user/formats.py

Если нам потребуются более широкие возможности обработки аргументов командной строки, обратите внимание на модули getopt и optparse - они входят в стандарт библиотеку.

**Изменение пути поиска модулей**. стр. 672

Мы пока еще не знаем как сами программы могут изменять путь поиска, изменяя встроенный список с именем sys.path.

sys.path инициализируется во время запуска программы, однако и после этого допускается удалять, добавлять и изменять компоненты списка как нам нужно:

>>> import sys

>>> sys.path

['', '/home/user', '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-i386-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']

>>> sys.path.append('/home/user/mydocs') # Дополнение пути поиска модулей

>>> import string # новый каталог будет участвовать

# в поиске

>>> sys.path

['', '/home/user', '/usr/bin', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-i386-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages', '/home/user/mydocs']

Как только будут внесены изменения, они будут действовать на все операции импорта далее. Список поиска можно изменять произвольным образом:

# Перезагружем интерактивную оболочку и модифицируем sys.path

>>> import sys

>>> sys.path = [r'/home/user/temp']

>>> sys.path.append('/home/user/my\_notes')

>>> sys.path

['/home/user/temp', '/home/user/my\_notes']

>>> import string

...

>>> ImportError: No module named string

- ***Внимание!*** В своей IDLE я провел все эти команды и НЕ ПОЛУЧИЛ ошибку!!! Не знаю почему, но у меня в оболочке не вышла ошибка импортирования, хотя sys.path я изменил и в моих каталогах нет никаких модулей string. То же самое и на работе, на Виндовс 7.

- В общем очередная загадка или Питон 3.4 или моих незнаний. Возможно импортировать нужно какой-нибудь другой модуль, может string не дает ошибки в таких случаях.

Не забывайте, что такие изменения списка sys.path действуют только в рамках интерактивного сеанса или внутри программы (технически - в рамках процесса), где были выполнены эти изменения - они не сохраняются после завершения работы интерпретатора.

**Расширение as для инструкций import и from.**

Обе инструкции были расширены так, чтобы можно было давать модулям, в сценарии, другие имена. Напр. инструкция import:

import longmodulename as name

- эквивалентна инструкциям:

impport longmodulename

name = longmodulename

del longmodulename # Не сохранять оригинальное имя

- теперь ссылки на модуль longmodulename будут идти по имени name.

Точно также и для инструкции from:

from module import longname as name # Изменяем имена, импортируемые из файла

Это расширение инструкций используется для создания коротких синонимов для длинных имен и для избежания конфликтов имен (во избежание перезаписи):

import reallylongmodulename as name # Использовать короткий псевдоним

name.func()

from module1 import utility as util1 # Избегаем повторения имен в сценарии

from module1 import utility as util2

util1(); util2()

Кроме того, это расширение может пригодиться для замены длинных путей из цепочек каталогов, при импорте пакетов:

import dir1.dir2.mod as mod # С такой заменой гораздо удобнее

mod.func()

**Модули - это объекты: метапрограммы.** стр. 674

Т.к. модули экспортируют большую часть своих свойств в виде атрибутов, то можно легко создавать программы, управляющие др. программами. Такие менеджеры программ наз-ся - метапрограммы, потому что работают поверх др. программ. Этот прием также наз-ют - интроспекцией, потому что программы могут видеть внутр. устройство объектов и на основе этого действовать.

Напр., чтобы получить значение атрибута name в модуле М, можно использ. полное имя атрибута или обратиться к нему с пом. словаря атрибутов модуля (встр. атрибут \_\_dict\_\_, см. гл. 22). Также интерпретатор экспортирует список всех загруженных модулей в виде словаря sys.modules (т.е. атрибут modules модуля sys) и предоставляет ф-цию getattr, которая дает доступ ко всем атрибутам по строкам с их именами (типо выражения object.attr, только attr - это строка времени выполнения). Благодаря этому все следующие выражения дают один и тот же атрибут:

M.name # Полное имя объекта

M.\_\_dict\_\_['name'] # Доступ через словарь пространства имен

sys.modules['M'].name # Доступ через таблицу загруженных модулей

getattr(M, 'name') # Доступ с пом. встроенной ф-ции

Ниже пример модуля mydir.py, в которм реализуется измененная версия встроенной ф-ции dir. Этот модуль определяет и экспортирует ф-цию с именем listing, которая принимает объект модуля (аргумент) и выводит отформатированный листинг пространства имен модуля:

"""

Модуль выводит содержимое пространства имен др. модулей

"""

seplen = 60

sepchr = '-'

def listing(module, verbose=True):

sepline = sepchr \* seplen

if verbose:

print(sepline)

print('name: ', module.\_\_name\_\_, 'file: ', module.\_\_file\_\_)

print(sepline)

count = 0

for attr in module.\_\_dict\_\_: # Сканировать пространство имен

print("%02d %s" % (count, attr), end=' ')

if attr.startswith('\_\_'):

print("<built-in name>") # Пропустить \_\_file\_\_ и др.

else:

print(getattr(module, attr)) # То же, что и .\_\_dict\_\_[attr]

count = count+1

if verbose:

print(sepline)

print(module.\_\_name\_\_, 'has %d names' % count)

print(sepline)

if \_\_name\_\_ == "\_\_main\_\_": # А этот код был в учебнике на одном уровне с телом ф-ции,

import mydir # и НЕ работал, я поставил его на уровне def и он ЗАРАБОТАЛ

listing(mydir) # Код самопроверки: вывести своё пространство имен

Мы можем просмотреть документацию с пом. \_\_doc\_\_ или help:

>>> import mydir

>>> help(mydir)

Help on module mydir:

NAME

mydir - Модуль выводит содержимое пространства имен др. модулей

FUNCTIONS

listing(module, verbose=True)

DATA

sepchr = '-'

seplen = 60

FILE

c:\python34\mydir.py

А теперь посмотрим как выполняется этот файл:

C:\Python34>python mydir.py

------------------------------------------------------------

name: mydir file: C:\Python34\mydir.py

------------------------------------------------------------

00 \_\_builtins\_\_ <built-in name>

01 listing <function listing at 0x01302618>

02 sepchr -

03 \_\_name\_\_ <built-in name>

04 \_\_file\_\_ <built-in name>

05 \_\_package\_\_ <built-in name>

06 seplen 60

07 \_\_cached\_\_ <built-in name>

08 \_\_doc\_\_ <built-in name>

09 \_\_spec\_\_ <built-in name>

10 \_\_loader\_\_ <built-in name>

------------------------------------------------------------

mydir has 11 names

------------------------------------------------------------

- в книге всего 8 позиций (07 включительно), но думаю это не важно, главное у меня всё заработало, а то из-за ошибок в файле не запускался код самопроверки (см. комменатрии в модуле выше).

Чтобы использ. этот модуль как инструмент инспекции др. модулей просто передайте ф-ции listing объект требуемого модуля. Пример с модулем tkinter из стандарт. библиотеки:

>>> import mydir

>>> import tkinter

>>> mydir.listing(tkinter)

------------------------------------------------------------

name: tkinter file: C:\Python34\lib\tkinter\\_\_init\_\_.py

------------------------------------------------------------

00 Grid <class 'tkinter.Grid'>

01 CASCADE cascade

02 simpledialog <module 'tkinter.simpledialog' from 'C:\\Python34\\lib\\tkinter\\simpledialog.py'>

03 SINGLE single

04 Tributton <class 'tkinter.Tributton'>

05 getboolean <function getboolean at 0x02045BB8>

...часть позиций мы пропустим...

162 \_stringify <function \_stringify at 0x020454B0>

163 ACTIVE active

164 Canvas <class 'tkinter.Canvas'>

165 NUMERIC numeric

166 END end

167 SCROLL scroll

------------------------------------------------------------

tkinter has 168 names

**Импортирование модулей по имени в виде строки.**

Имя модуля в import или from является именем переменной. Иногда наша программа будет получать имя модуля в виде строки (напр. внутри графич. интерфейса). Напрямую его нельзя так передавать:

>>> import "string"

SyntaxError: invalid syntax

Также нельзя импортировать модуль просто присвоив строку переменной:

x = "string"

import x

- в этом случае интерпретатор будет импортировать файл x.py, а не модуль string - имя в инструкции import превращается в имя переменной, которой присв-ся объект модуля и идентифицирует внеш. файл буквально.

Т.е. нам нужен спец. инструмент для решения этой проблемы. Обычно для этого конструируется строка программ. кода, содержащая import, которая затем передается встроенной ф-ции exec для исполнения:

>>> modname = "string"

>>> string

NameError: name 'string' is not defined

>>> exec("import " + modname) # Выполняется как строка программного кода

>>> string

<module 'string' from 'C:\\Python34\\lib\\string.py'>

- видим, что модуль был импортирован.

Ф-ция exec (и родственная ей ф-ция eval) скомпилирует строку в код и передаст интерпретатору для исполнения.

Единственный недостаток в том, что ф-ция exec должна скомпилировать инструкцию import всякий раз когда она запускается - т.е. если импортировать придется часто, то прога может работать немного быстрее при сипольз-нии встр. ф-ции \_\_import\_\_, которая выполняет загрузку модуля, получая его имя в виде строки.

Результат тот же самый, но \_\_import\_\_ возвращает объект модуля, поэтому его нужно присвоить переменной, чтобы сохранить:

>>> modname = "string"

>>> string

NameError: name 'string' is not defined

>>> string = \_\_import\_\_(modname)

>>> string

<module 'string' from 'C:\\Python34\\lib\\string.py'>

**Транзитивная перегрузка модулей.**

Ранее мы изучали повторную загрузку модулей как способ ввести изменения без перезапуска программы. Когда выполняется повторная загрузка модуля, перезагружается только конкретный файл модуля - при этом НЕ выполняется повторная загрузка модулей, которые были импортированы перезагружаемым модулем.

Напр. если перезагружем модуль А, который импортирует модули В и С, то будет перезагружен только модуль А, но не В и С. Инструкции импорта внутри модуля А просто вернут нам объекты, уже загруженных модулей В и С.

Т.о. не следует полагаться на транзитивную перезагрузку модулей - лучше неск-ко раз вызывать ф-цию reload для обновления субкомпонентов. Конечно это может может привести к знач. объему работы. Конечно можно предусмотреть автоматическую перезагрузку компонентов, добавив вызов reload в родительские модули (напр. А), но это усложнит реализацию модуля.

Еще лучше было бы написать универсальный инструмент для транзитивной перезагрузки модулей, сканируя содержимое атрибутов \_\_dict\_\_ модулей и проверяя атрибут type в каждом элементе, чтобы отыскать вложенные модули.

Такие ф-ции могут рекурсивно вызывать себя и обходить любые цепочки импортирования.

Пример. Модуль reloadall.py содержит ф-цию reload\_all, которая автоматом перезагружает модуль каждого импортируемого им модуля и т.д., до самого конца цепочки импортирования. Она использ. словарь, с пом. которого отыскивает уже загруженные модули, рекурсию - для обхода цепочки импорта и модуль types из стандарт. библиотеки, в котором просто предопределены значения атрибута type для всех встр. типов.

Словарь visited применяется для избежания зацикливания в случае появления рекурсивных или избыточных инструкций импорта. Этот прием основан на том факте, что объекты модулей могут играть роль ключей словаря:

"""

Транзитивная перегрузка вложенных модулей

"""

import types

from imp import reload

def status(module):

print('reloading ' + module.\_\_name\_\_)

def transitive\_reload(module, visited):

if not module in visited: # Пропустить повторные посещения

status(module) # Перезагрузить модуль

reload(module) # И посетить дочерние модули

visited[module] = None

for attrobj in module.\_\_dict\_\_.values(): # Для всех атрибутов

if type(attrobj) == types.ModuleType: # Рекурсия, если модуль

transitive\_reload(attrobj, visited)

def reload\_all(\*args):

visited = {}

for arg in args:

if type(arg) == types.ModuleType:

transitive\_reload(arg, visited)

if \_\_name\_\_ == '\_\_main\_\_':

import reloadall # Тест: перезагрузить самого себя

reload\_all(reloadall) # Должна перезагрузить этот модуль и модуль types

Когда файл запускается как самостоятельная программа, его код самопроверки выполнит проверку самого модуля - он должен импортировать самого себя, потому что его собственное имя не будет поределено в файле без инструкции импорта:

C:\Users\afisenko>reloadall.py

reloading reloadall

reloading types

А чтобы воспользоваться утилитой нужно импортировать ф-цию reload\_all и передать ей имя уже загруженного модуля.

Например обратим внимание, что модуль os импортируется модулем tkinter, при этом tkinter импортирует модуль sys раньше чем модуль os, смотрим пример:

>>> from reloadall import reload\_all

>>> import os, tkinter

>>> reload\_all(os)

reloading os

reloading ntpath

reloading stat

reloading genericpath

reloading sys

reloading errno

>>> reload\_all(tkinter)

reloading tkinter

reloading tkinter.dialog

reloading tkinter.simpledialog

reloading tkinter.colorchooser

reloading tkinter.constants

reloading tkinter.commondialog

reloading tkinter.messagebox

reloading tkinter.filedialog

reloading tkinter.font

reloading itertools

reloading re

reloading \_locale

reloading copyreg

reloading sre\_parse

reloading sre\_compile

reloading \_sre

reloading sys

reloading fnmatch

reloading os

reloading ntpath

reloading stat

reloading genericpath

reloading errno

reloading posixpath

reloading functools

reloading tkinter.\_fix

reloading \_tkinter

reloading ctypes

reloading ctypes.\_endian

А ниже приведем пример в котором показаны различия между обычной и транзитивной перезагрузкой - при обычной изменения во вложенных файлах не вступят в силу после перезагрузки, только после транзитивной:

import b # a.py

X = 1

import c # b.py

Y = 2

Z = 3 # c.py

>>> import a

>>> a.X, a.b.Y, a.b.c.Z

(1, 2, 3)

- Теперь изменим значения переменных во всех трех файлах (11, 22 и 33):

>>> from imp import reload

>>> reload(a)

<module 'a' from 'C:\\Python34\\a.py'>

>>> a.X, a.b.Y, a.b.c.Z

(11, 2, 3)

- как видим перезагрузился только модуль а.py, остальные данные остались прежними. Придется прибегнуть к транзитивной перезагрузке с помощью ф-ции reload\_all:

>>> from reloadall import reload\_all

>>> reload\_all(a)

reloading a

reloading b

reloading c

>>> a.X, a.b.Y, a.b.c.Z

(11, 22, 33)

**Концепции проектирования модулей.**

Ознакомимся с некоторыми основными положениями проектирования модулей:

- **В языке Питон вы всегда находитесь в модуле.** Нельзя написать код, который не находился бы ни в каком модуле. Даже код вводимый в интерактив. оболочке, на самом деле, относится к встр. модулю с именем \_\_main\_\_ - а особенность оболочки лиш ьв том, что код, после выполнения, удаляется, а результаты выводлятся автоматически.

- **Минимизируйте взаимозависимость модулей: глобальные переменные.** Модули должны быть максимально независимы от глобал. имен в др. модулях; модули работают лучше когда они написаны как самостоятельные программы.

- **Максимизируйте согласованность внутри модуля: общая цель.** Увеличение согласованности модуля, когда все компоненты модуля использ-ся для достижения общей цели, - этой прямой путь к уменьшению взаимозависимиости (см. пред. пункт), что есть хорошо.

- **Модули должны редко изменять переменные в др. модулях.** Мы узнали об этом еще в гл. 17, но повторим. Использ-ие глобал. переменных из др. модулей (именно так клиенты импортируют службы) - совершенное норм. явление, тогда как изменение глоабл. переменных в др. модулях - это уже признак проблем проектирования. Бывают исключения конечно, но все же мы должны стараться обмениваться данными через аргументы и возвращаемые значения ф-ций, не прибегая к прямому изменению переменых в др. модулях.

**Рисунок. 24.1 Среда выполнения модуля.** стр. 682

**Типичные проблемы при работе с модулями**. стр. 682

**Порядок следования инструкций на верхнем уровне имеет значение.**

Когда модуль впервые импортируется (или перезагружается), интерпретатор выполняет инструкции в нем одну за другой, сверху вниз. Отсюда несколько замечаний, касающихся опережающих ссылок на переменные:

- Инструкции программ. кода на верх. уровне в файле модуля (не вложенные в ф-цию) выполняются, как только интерпретатор достигает их в процессе импорта. Поэтому он не может ссылаться на имена, присваивание которым производится ниже.

- Программ. код внутри ф-ций не выполняется, пока ф-ция не будет вызвана - разрешение имен внутри ф-ций не производится до момента их вызова, поэтому они обычно могут ссылаться на имена, расположенные в любой части файла.

Опережающие ссылки беспокоят только в коде верхнего уровня, который вып-ся немедленно; ф-ции могут ссылаться на любые имена. Пример:

func1() # Ошибка: имя "func1" еще не существует

def func1():

print(func2()) # ОК: поиск имени "func2" будет выполнен позднее

func1() # Ошибка: имя "func2" еще не существует

def func2():

return "Hello"

func1() # ОК: "func1" и "func2" определены

- Здесь у нас инструкции вып-ся сверху вниз (напр. импортируем и запускаем как программу). Первый вызов func1 неудачен, т.к. инструкция def для func1 еще не выполнена. Вызоы func2 внутри func1 будет работать без ошибок при условии, что к моменту вызова func1 инструкция def func2 уже будет выполнена (этого как раз не произошло к моменту второго вызова func1 на верх. уровне). Ну и понятно, что последний вызов func1 будет успешным, т.к. оба имени уже определены.

Итак видим, что смешивание инструкций def с программ. кодом верх. уровя не только осложняет его чтение, но и ставит его работоспособность в зависимость от порядка следования инструкций.

Возьмите за правило помешать код с инструкциями def в начало файла, а код верхнего уровня - в конец файла, если уж вам позарез нужно их объединить.

**Инструкция from создает копии, а не ссылки.**

Итак, from при выполнении присваивания именам в области видимости импортирующего модуля не создает синонимы, а копирует имена. Рез-т такой же как и у др. инструкций присваивания, но есть одно отличие, особенно когда код, использующий объекты совместно, находится в разных файлах.

Пусть у нас есть модуль nested1.py:

# nested1.py

X = 99

def printer(): print(X)# nested1.py

Если импортировать эти два имени с пом. from в др. модуле (nested2.py), будут получены копии этих имен, а не ссылки на них. Т.е. изменение имени в импортирующем модуле приведет к изменениям только локал. версии этого имени, а имя в nested1.py будет иметь прежнее значение:

# nested2.py

from nested1 import X, printer # Копировать имена

X = 88 # Изменит только локал. версию "X"!

printer() # Х в nested1 по-прежнему будет равно 99

Проверяем:

>>> import nested2

99

Однако, если импортировать весь модуль с пом. import и затем изменить значение с использованием полного имени, то это приведет к изменению имени в файле nested1.py:

# nested3.py

import nested1

nested1.X = 88

nested1.printer()

И проверяем:

>>> import nested3

88

**Инструкция from\* может затушевать смысл переменных.**

Об этом мы знаем уже, - т.к. в инструкции from module import \* не указываются конкретные имена, она может перезаписать имена, уже используемые в области видимости импортирующего модуля. И поиск проблемы в этом случае очень тяжелый.

Например если инструкция from \* применяется к трем модулям, а затем вызывается какая-либо ф-ция, то у нас не будет иного способа узнать какая же на самом деле вызывается ф-ция, кроме как выполнить поиск во всех трех файлах модулей, что очень затратно и сложно.

Решение в том, чтобы так не делать. Старайтесь явно перечислять атрибуты в from и ограничивайте применение from \* одним модулем на файл (с тщательным вниманием). Отказаться совсем от from было бы неудобно, т.к. это удобный инструмент, даже пример с тремя импортами может быть полезен, главное всё делат ьвнимательно и тщательно проверять.

**Функция reload может не оказывать влияния, если импорт осуществлялся инструкцией from.**

Еще одна ловушка from: как мы знаем from копирует (присваивает) имена при выполнении, поэтому нет обратной связи с модулем, откуда были скопированы имена.

Вследствии этого повторная загрузка модуля может не оказывать воздействия на клиентов, которые импортирвоали его имена с пом. from. Т.е. имена в модулях-клиентах будут по-прежнему ссылаться на оригинальные объекты, поленные from, даже если имена в оригинальном модуле будут переопределены:

from module import X # X может не измениться после перезагрузки!

...

from imp import reload

reload(module) # Изменится модуль, но не мои имена

Х # По-прежнему ссылается на старый объект

Чтобы этого избежать исполз-те import и полные имена переменных вместо from:

import module # Получаем объект модуля, а не имена

...

from imp import reload

reload(module) # Изменит сам объект модуля

module.Х # Текущее значение Х: отражает рез-т перезагрузки

**reload, from и тестирование в интерактивной оболочке.**

Как говорилось в гл. 3, из-за сложностей лучше не использовать импорт и повторную загрузку для запуска программ. Это еще более осложняется когда появл-ся from.

Опишем частую проблему. Представим, что после открытия модуля в окне редактирования, мы запускаем интерактивный сеанс, чтобы загрузить и протестировать модуль с пом. from:

from module import function

function(1, 2, 3)

Отыскав ошибку, мы переходим в редактор и исправляем её, затем пытаемся перезагрузить модуль:

from imp import reload

reload(module)

Но мы не получим нужного эффекта - инструкция from создала имя function, но не создала module. Чтобы сослаться на модуль в ф-ции reload, его сначала нужно импортировать инструкцией import:

from imp import reload

import module

reload(module)

function(1, 2, 3)

Но и теперь мы не получим нужный эффект - reload обновит объект модуля, но имена, такие как function, скопированные из модуля ранее, по-прежнему продолжают ссылаться на старые объекты. Чтобы получить доступ к новой ф-ции после перезагрузки модуля, нужно вызывать её как module.function или повторно запустить from:

from imp import reload

import module

reload(module)

from module import function

function(1, 2, 3) # Или использовать module.function

Мы видим как много условий у нас здесь, мы должны не только не забывать перезагружать модуль после импорта, но и не забывать повторно запускать инструкции from после перезагрузки модуля. К тому же не забывать импортирвоать саму ф-цию reload (для Питон 3.0). Это достаточно сложно и требует внимания.

Лучше конечно не объединять reload и from, используйте ф-цию reload в паре с инструкцией import или запускайте программы др. способами.

**Рекурсивный импорт с инструкцией from может не работать.**

В ходе импорта инструкции в файле вып-ся от начала и до конца, и нужно быть внимательным когда использ-ся модули импортирующие друг друга (рекурсивный импорт).

Если использовать import, то мы можем и не заметить проблем в таком случае (ведь мы сипользуем полные имена), а если мы используем from, то будьте внимательны - следите за доступом к именам, вдруг они еще не определены.

Пример. Модули recur1 и recur2. Модуль recur1 создает имя Х и затем импортирует recur2 до того как присвоит значение имени Y. В этом месте модуль recur2 может импортировать модуль recur1 целиком с пом. import (он уже существует во внутр. таблице модулей интерпретатора), но если использ. from, ей будет доступно только имя Х, а имя Y еще не существует и будет ошибка:

# recur1.py

X = 1

import recur2 # Запустить recur2, если он еще не был импортирован

Y = 2

# recur2.py

from recur1 import X # ОК: "X" уже имеет значение

from recur1 import Y # Ошибка: "Y" еще не существует

>>> import recur1

... # Ошибка: "Y" еще не существует

ImportError: cannot import name Y

При рекурсивном импорте модуля recur1 и модуля recur2 интерпретатор не будет повторно выполнять инструкции модуля recur1 (иначе получим бесконечный цикл), но пространство имен модуля recur1 еще не заполнено до конца к моменту, когда он импортируется модулем recur2.

Решение? - Не используйте from в операции рекурсивного импорта. Иначе попадете в зависимость от порядка следования инструкций в модулях.

Два решения проблемы:

- Ликвидировать рекурсивный импорт, правильно проектируя модули: увеличить согласованность внутри моудля и уменьшить взаимозависимость между модулями.

- Если не можем избавться от циклов, попробуйте отсрочить обращение к именам модуля, используя инструкции import и полные имена (вместо from), или поместите from внутрь ф-ций, либо ближе к конце файла.

**Упражнения к 5 части.** стр. 689