**Изучаем Python**

**Марк Лутц 2011, издание O’ Reilly**

**Том 2**

***Часть. 4. Функции.*** *стр. 459*

**Глава 16. Основы функций.** стр. 461

Функция - это средство, позволяющее группировать наборы инструкций так, что в программе они могут запускаться неоднократно.

Они могут вычислять результат и иметь входные параметры. Оформление операций в виде функций - очень удобный инструмент, который мы можем использовать в самых разных ситуациях.

Т.о. функции устраняют необходимость вставлять в программу избыточные копии блоков одного и того же программного кода, он заменяется одной единственной функцией. Кстати, при изменении программы достаточно изменить только функцию, а не все повторяющиеся блоки.

Функции - самые основные программные структуры в ЯП Питон, обеспечивающие многократное использование программного кода и уменьшающие его избыточность. Также это средство проеткирования, разбивающее сложжную систему на простые и легкие части.

**Таблица 16.1 Инструкции и выражения, имеющие отношения к функциям.** стр. 461

**Зачем нужны функции?**

Это универсальное средство структурирования программы, в других ЯП функции могут называться - подпрограммами или процедурами.

Две основные роли:

- *Максимизировать многократное использование программного кода и минимизировать его избыточность.* Функции в Питон являют простейший способ упаковки логики выполнения, которая может использоваться в разных местах программы и более чем один раз. Ф-ции позволяют обощать и группировать код, который потом можно использ. много раз. Т.к. ф-ции позволяют поместить реализацию в одно место и использовать её в разных местах, они явл-ся основным инструментом структуризации - дают возможность уменьшить избыточность кода.

- *Процедурная декомпозиция.* Ф-ция помогает разбить сложную систему на части. Ведь гораздо проще создать решение малых задач по отдельности, чем реализовать большую и сложную задачу целиком и разом. Вообще ф-ции описывают "как делать", а не "зачем делать".

**Создание функций.** стр. 463

Ранее мы уже пользовались функциями, напр. создавали объект с помощью ф-ции open, использовали len и мн. др.

В этой главе мы узнаем как создавать новые функции. Написанные нами функции ведут себя так же ка ки встроенные - они могут вызываться в выражениях, получать значения и возвращать результаты.

Кстати, ф-ции в Питон ведут себя не так как в С.

Краткое описание основных концепций и понятий функций:

- **def - это испольняемый программный код**. Ф-ции в Питон создаются с помощью инструкции def. В отличие от компилирующихся ЯП, типа С, def относится к классу исполняемых инструкций - ф-ция не существует, пока интерпретатор не доберется до инструкции def и не выполнит ее. Допустимо (иногда и полезно) вкладывать def внутрь инструкций if, циклов while и даже других инструкций def. В самом типичном случае def вставляется внутрь модулей и генерируют функции во время первой операции импортирования модуля.

- **def создает объект и присваивает ему имя.** Интерпретатор встречает и выполняет def, при это создавая новый объект-функцию и связывая его с именем ф-ции. Т.е. имя становится ссылкой на объект-функцию. Объект-функция может быть свзязано с неск-ми именами, может сохраняться в списке и т.д. Функциям можно прикреплять атрибуты, с данными.

- **Выражение lambda создает объект и возвращает его в виде результата.** Ф-ции могут создаваться с пом. выражения lambda. Это дает нам создавать встроенные определения ф-ций там, где синтаксис не позволяет использ. инструкцию def.

- **return - передает объект результата вызывающей программе.** Когда ф-ция вызывается, вызывающая программа приостанавливает работу, пока ф-ция не завершится и не вернет управление. Ф-ции вычисляющие значения возвпращают ег ос помощью return - возвращаемое значение становится рез-том вызова ф-ции.

- **yield - передает объект результата вызывающей программе и запоминает, где был произвдеен возврат.** Ф-ции, называемые генераторами, могут использовать yield и сохранять состояние так, чтобы работа ф-ции могла быть возобновлена позднее.

- **Аргументы передаются посредством присваивания (в виде ссылок на объекты).** В Питон аргуменыт передаются ф-циям посредством операции присваивания (т.е. в виде ссылок на объекты). В Питон и вызывающая программа и и ф-ция совместно использ. ссылку на объект, но здесь нет никакого совмещения имен. Изменение имени аргумента не меняет имени в вызывающ. программе, но модификация изменямых объектоввнутри ф-ции может приводить к изменению объектов в вызывающей программе.

- **global - объявляет переменные, глобальные для модуля, без присваивания им значений.** По умолч. все имена, прсиваивание которым произв-ся внутри ф-ции, явл-ся локальными для этих ф-ций и существуют только во время выполнения этих ф-ций. Чтобы прсивоить значение имени в объемлющем модуле, ф-ция должна объявить его с помощью инструкции global. Поиск имен всегда производится в некоторой области видимости - там, где хранятся переменные, - а операция присваивания связывает имена с областями видимости.

- **nonlocal - объявляет переменные, находящиеся в в области видимости объемлющей функции, без присваивания им значений.** Это позволяет использ. объемлющие ф-ции, как место хранения информации о состоянии - информация восстанавливается в момент вызова ф-ции, при этом отпадает необходимость использ-ть глобальные переменные.

- **Аргументы получают свои значения (ссылки на объекты) в результате выполнения операции присваивания.** При передаче аргументво ф-ции выполняется операция прсиваивания значений (т.е. ссылок на объекты). Передача объектов в функции производится по ссылкам, но это не означает, что созд-ся псевдонимы имен. Изменение аргуменат фнутри ф-ции не влечет за собйо изменение соответствующего имени в вызывающей программе, но изменения в изменемых объектах отразятся в вызывающей программе.

- **Аргументы, возвращаемые значения и переменные не объявляются.** Как и во всем Питоне, на ф-ции не накладываются ограничения по типу. Фактически, никакие элементы ф-ций не требуют предварительного объявления, мы можем передавать аргументы любых типов, возвращать из ф-ции объекты любого типа и т.д. Т.о. одна и та же ф-ция может применяться к объектам различных типов. Допустимыми явл-ся любые объекты, поддерживающие совместимые интерфейсы (методы и выражения), независимо от конкретного типа.

**Инструкция def.**

def создает объект ф-ции и связывает его с именем. Общий вид таков:

def <name>(arg1, arg2, ... argN):

<statements>

Мы видим, что def явл-ся типичной составной инструкцией, с заголовком и блоком инструкций, т.н. - телом ф-ции, т.е. программным кодом, который выполняется интерпретатором всякий раз, когда ф-ция вызывается.

В заголовке определяется имя ф-ции и список из нуля и более аргументов (др. навазние - параметры). Имена аргументов в заголовке будут связаны с объектами, передаваемыми в ф-цию, в точке вызова.

Тело ф-ции часто содержит инструкцию return:

def <name>(arg1, arg2, ... argN):

...

return <value>

- return может располагаться в любом месте в теле ф-ции - она завершает работу ф-ции и передает результат вызывающей программе.

Инструкция return явл-ся необязательной, если ее нет ф-ция завершается когда достигается конец тела ф-ции. С технич. точки зрения ф-ция без return автоматически возвращает объект None, однако это значение обычно просто игнорируется.

**Инструкции def исполняются во время выполнения.**

def - исполняемая инструкция, когда она выполняется, она создает новый объект ф-ции и присваивает этот объект имени.

**Не забываем!** - Все, что есть в Питон относится к времени выполнения, здесь нет понятия времени компиляции.

Т.к. def инструкция, то можеть быть везде где могут быть инструкции - даже внутри других инструкций:

if test:

def func(): # Определяет функцию таким способом

...

else:

def func(): # Или таким способом

...

...

func() # Вызов выбранной версии

Чтобы понять этот фрагмент обратим внимание, что def напоминает инструкцию присваивания = т.к. она просто выполняет присваивание во время выполнения.

В отличие от С ф-ции в Питон не должны быть полностью поределены к моменту запуска программы. Т.е. инструкции def не интерпретируются, пока не будут достигнуты и выполнены потоком выполнения, а код внутри def не выполняется, пока ф-ция не будет вызвана позднее.

Важно не имя функции, а объект, на который ссылается имя:

othername = func # Связывание объекта функции с именем

othername() # Вызов функции

- мы связали нашу функцию с другим именем и вызвали с использованием нового имени.

Ф-ция - это обычные объекты, они явно записываются в память во время выполнения программы.

Ф-ции позволяют присвоединять любые атрибуты, с информацией, напр. для сохранения инф-ии:

def func(): ... # Создает объект функции

func() # Вызывает объект

func.attr = value # Присоединяет атрибут к объекту

**Первый пример: определения и вызовы.** стр. 466

Напишем первый пример функции. Как видим ф-ция имеет две стороны: определение (инструкция def, которая создает ф-цию) и вызов (выражение, которое предписывает интерпретатору выполнить тело функции).

**Определение.**

Ф-ция times, которая возвращает рез-т обработки двух аргументов:

>>> def times(x, y):

return x \* y

...

>>>

- когда интерпретатор достигает инструкции def и выполняет ее, он создает объект функции, в который упакует программнфый код ф-ции и свяжет объект с именем times. Как правило такие ф-ции размещаются в файлах модулей и выполняются по время импортирования, но наша небольшая и можно создать ее прямо в интерактивной оболочке.

**Вызов.**

После выполнения def появл-ся возможность вызвать ф-цию, добавив круглые скобки после ее имени. В скобках указ-ся аргументы:

>>> times(2, 4) # В круглых скобках аргументы: 2 и 4

8

- имени x присваивается значение 2, а имени y - значение 4, после чего запускается тело ф-ции. В нашем случае значение возвращается инструкцией return (собственно она и есть тело нашей ф-ции).

Теперь мы можем указать, что тело функции выполняется в момент ВЫЗОВА функции.

Результат ф-ции можно присвоить переменной:

>>> x = times(3.14, 4)

>>> x

12.56

А теперь передадим ф-ции объекты разных типов:

>>> times('Ni', 4) # Получилась операция повторения

'NiNiNiNi'

- т.о. мы видим, что смысл ф-ции times и тип возвращаемого значения определяется аргументами, которые ей передаются.

**Полиморфизм в языке Питон.**

Как мы уже подчеркнули, смысл выражения x\*y полностью зависит от типа объектов x и y - т.е. это может быть как умножение, так и повторение.

В Питон именно объекты определяют синтаксический смысл операции.

Такого рода зависимость от типов известна как полиморфизм - термин, означающий, что смысл операции зависит от типов обрабатываемых объектов. Фактически все операции в Питон явл-ся полиморфическими.

Т.о. одна функция может автоматически применяться к целой категории типов объектов. Если объект поддерживает ожидаемые методы и операторы выражений, он будет совместим с логикой функции.

Напр. наша ф-ция times, - любые два объекта, поддерживающие оператор \*, смогут обрабатываться ф-цией times и не важно что это за объекты и когда были созданы.

Если же будут переданы объекты, НЕ поддерживающие ожидаемый интерфейс, интерпретатор автоматически возбудит исключение.

Вообще интерфейс - это набор методов и операций, которые используются функцией.

Т.о. мы видим отличие Питона от статической типизации - программный код на Питоне не делает предположений о конкретных типах данных. Вообще, при программировании на Питон во внимание принимаются интерфейсы объектов, а не типы данных.

Конечно такая модель полиморфизма предполагает необходимость тестирования кода на наличие ошибок, напр. на соответствие типов и интерфейсов и т.п. Но тестирование оно много где применяется в больших масштабах, так что это не совсем даже и недостаток.

**Второй пример: пересечение последовательностей.** стр. 469

Вспомним цикл for, который выбирал элементы, общие для двух строк. Мы его усовершенствуем, сделав функцию.

**Определение.**

Плюсы оформления нашего цикла в виде функции:

- Сделав код в виде ф-ции, появится возможность использовать его столько раз сколько нужно;

- Т.к. вызывающая программа может передавать ф-ции произвольные аргументы, ф-ция может использ-ся с любыми двумя последовательностями (или итерируемыми объектами) для полученяи их пересечения;

- Когда логика работы в виде ф-ции достаточно изменить код всего в одном месте, чтобы изменить способ полученяи пересечения;

- Поместив ф-цию в файл модуля, её можно импортировать и использовать в любой программе на нашем ПК.

Итак, наш код, обернутый в функцию, превращается в утилиту нахождения пересечения:

def intersect(seq1, seq2):

res = []

for x in seq1:

if x in seq2:

res.append(x)

return res

- все понятно: имя, аргументы, тело ф-ции, return. Двигаемся дальше.

**Вызов.**

Мы поместили нашу ф-цию в файл и для её создания импортируем наш модуль. Как только def будет выполнена, можно будет вызывать ф-цию и передавать ей два объекта последовательностей в круглых скобках:

>>> def inter(seq1, seq2):

res = []

for x in seq1:

if x in seq2:

res.append(x)

return res

>>> inter(s1, s2)

['S', 'A', 'M']

- Как видим я набрал функцию прямо в IDLE, хотя в книге написано, что можно создать её в модуле и импортировать его, но тут начинается самое интересное...

- Я создал файл tnt.py и импортировал его, вызвав dir(tnt) я увидел, что атрибут inter присутствует у модуля tnt, НО - при попытке вызвать ф-цию inter:

>>>inter(s1, s2)

...

NameError: name 'inter' is not defined

- я получал сообщение об ошибке "NameError: name 'inter' is not defined", не представляю с чем это связано, возможно с тем, что tnt.py расположен в моей домашней директории.

- Самое печальное то, что я не знаю как с этим бороться.

*Замечание! –* Я все-таки разобрался с этой проблемой!!! см. далее

**Разобрался!!!!!**

Мы импортируем модуль, а потом указывая имя модуля и его аргументы мы их используем. А в нашем случае ф-ция inter() как раз явл-ся аргументом модуля tnt (как мы уже писали выше о проверке dir()) и в конечном итоге всё работает:

>>> import tnt

>>> s1 = 'spam'

>>> s2 = 'scam'

>>> tnt.inter(s1, s2)

['s', 'a', 'm']

- это **рекомендованный** способ использования модулей.

Или, чтобы не писать каждый раз имя модуля, используем конструкцию from import:

>>> s1 = 'spam'

>>> s2 = 'scam'

>>> from tnt import inter

>>> inter(s1, s2)

['s', 'a', 'm']

- Вот такие вот подводные камни. Но всё-таки рекомендуется использовать просто import.

Все-таки нужно повторить первую часть моих записок, а то вот уже забыл правила использования атрибутов модулей. А если бы всё это помнил, не пришлось бы целый час сидеть в инете и искать ответ на "загадочную" ошибку с импортированным модулем.

Кстати наша функция, если честно, работает слишком медленно и в принципе не особо нужна, её можно заменить единственным выражением генератора списков:

>>> [x for x in s1 if x in s2]

['S', 'A', 'M']

- но всё же это простой пример применения функций к различному диапазону типов (различных типов последовательностей).

**Еще о полиморфизме.**

Как и другие ф-ции, наша inter тоже полиморфна, пример:

>>> import tnt

>>> x = tnt.inter([1, 2, 3], (1, 4)) # Смешивание типов

>>> x # Объект с результатом

[1]

- мы передали ф-ции список и кортеж, два разных типа - и она все равно сработала!

Т.е. inter будет выполнять итерацию по объектам послетовательностей любых типов, если они поддерживают ожидаемый интерфейс.

В нашем случае для inter, это означает, что первый объект должен поддерживать циклы for, а второй - поддерживать оператор in, выполняющий проверку на вхождение.

Так наша ф-ция поддерживает любые итераторы с ожидаемымми интерфейсами, даже файлы, словари и объекты на основе классов с перегрузкой операторов.

Опять же - если объекты не поддерживают ожидаемые интерфейсы, то будет возбуждено исключение.

**Локальные переменные.**

Рассмотрим переменные в нашем примере, и соотв. функции.

Переменная res внутри ф-ции inter - это то, что в Питон называется - локальной переменной, - имя, которое доступно только программному коду внутри инструкции def и существует только во время выполнения ф-ции.

Фактически, любые имена, которым были присвоены какие-либо значения внутри ф-ции, по умолчанию классифицируются как локальные переменные.

Почти все имена в ф-ции inter явл-ся локальными переменными:

- Переменная res явно участвует в операции присваивания, - поэтому она локальная;

- Аргументы передаются через присваивание, - поэтому seq1 и seq2 тоже локальные переменные;

- Цикл for присваивает элементы переменной, поэтому имя х также явл-ся локальным.

Все эти локальные переменные появл-ся в момент вызова ф-ции и исчезают, когда ф-ция возвращает управление - return возвращает объект результата, а имя res исчезает.

**Глава 17. Области видимости.** стр. 474

**Области видимости в языке Питон.**

Что же такое имена в Питон. Каждый раз когда в программе используется некоторое имя, интерпретатор создает, изменяет или отыскивает это имя в пространстве имен - в области, где находятся имена.

Поиск имени применительно к программному коду, под термином область видимости подразумевается пространство имен, - т.е. место в коде, где имени было присвоено значение, определяет област ьвидимости этого имени для программного кода.

Почти все, что относится к именам (в т.ч. классификацию области видимости), в Питон связано с операциями присваивания. Т.е. мы видим, что имена появляются в тот момент когда им впервые присваивается некоторое значение, и прежде чем использовать имена им нужно присвоить значение.

Т.к. имена не объявл-ся заранее, интерпретатор, по месту операции присваивания, связывает имя с конкретным пространством имен. Т.е. место, где выполняется присваивание, определяет пространство имен, в котором будет наход-ся имя, а следовательно, и область его видимости.

Также, все имена, значения которым присв-ся внутри ф-ции, ассоциируются с пространством имен этой ф-ции. Это означает что:

- Имена, определяемые внутри инструкции def, видны только коду внутри def. К этим именам нельзя обратиться за пределами ф-ции;

- Имена, определяемые внутри def, не вступают в конфликт за пределами def, даже если и там и там одинаковые имена. Например имя Х за пределами def (в др. конструкции def или на верхнем уровне модуля), полностью отлично от имени Х, которому присвоено значение внутри инструкции def.

В любом случае, область видимости переменной (где она может использ-ся) всегда определяется местом, где ей было присвоено значение, и никакого отношения не имеет к месту, откуда была вызвана функция.

Как мы узнаем далее значения переменным могут быть присвоены в трех разных местах:

- Если присваивание переменной вып-ся внутри инструкции def, переменная явл-ся локальной для этой функции;

- Если присваивание производится в пределах объемлющей инструкции def, переменная явл-ся нелокальной для этой функции;

- Если присваивание произв-ся за пределами всех инструкций def, она явл-ся глобальной для всего файла (модуля, класса и т.п.).

Это называется лексической областью видимости, т.к. видимость опред-ся местоположением переменных в текстах программы, а не местом откуда вызывается ф-ция.

Пример, где инструкция Х = 99 создает глобальную переменную Х (видима в любом месте файла), а инструкция Х = 88 создает локальную (видима тольк овнутир def):

X = 99

def func():

X = 88

- и хоть имена одинаковы, конфликта не будет, т.к. область видимости делает их различными.

**Правила видимости имен.**

Ф-ции образуют вложенные пространства имен (области видимости), которые ограничивают доступ к используемым в них именам, т.о. имена внутри ф-ций не конфликтуют с именами за их пределами (внутри модуля или внутри др. ф-ций).

Еще раз повторим, - ф-ции образуют локальную область видимости, а модули - глобальную. Они связаны м/у стобой:

- **Объемлющий модуль - это глобальная область видимости**. Глобальные переменные, для внешнего мира становятся атрибутами объекта модуля, но внутри модуля исп-ся как простые переменные.

- **Глобальная область видимости охватывает единственный файл.** Не заблуждайтесь насчет слвоа "глобальный" - имена на верхнем уровне файла явл-ся глобальными только для программного кода в этом файле. В Питон нет понятия глобальной области видимости для всех файлов. Имена всегда относятся к какому-нибудь модулю и всегда нужно явно импортировать модуль. Если слышите "глобальный", подразумевайте "модуль".

- **Каждый вызов функции создает новую локальную область видимости.** Каждую инструкцию def (и lambda) можно представить как определение новой локальной области видимости. Т.к. Питон позволяет ф-циям вызывать самих себя в цикле (наз-ся - рекурсия), локальная область видимости, с технич. точки зрения соответствует вызову ф-ции - т.е. каждый вызов создает новое локальное пространство имен. Рекурсия может исп-ся для обработки данных, структура которых заранее не известна.

- **Операция присваивания создает локальные имена, если они не были объявлены глобальными или нелокальными.** По умолч. все имена, с присваиванием внутри ф-ции, помещаются в локальную область видимости. Если нужно присвоить значение имени верхнего уровня в модуле, который вмещает ф-цию, это имя нужно объявить внутри ф-ции глобальным с помощью инструкции global. Если нужно присвоить значение имени, наход-ся в объемлющей инструкции def, это имя нужно объявит ьвнутри ф-ции с помощью инструкции nonlocal.

- **Все остальные имена явл-ся локальными в области видимости объемлющей функции, глоабльными или встроенными.** Преполагается, что имена, которым не присвоены значения внутри определения ф-ции, находятся в объемлющей локальной области (внутри объемлющей инструкции def), глобальной (в пространстве имен модуля) или встроенно (предопределнные имена в модуле builtins).

Код вводимый в интерактив. оболочке подчиняется всем этим правилам, этот код находится на уровне модуля \_\_main\_\_ - этот модуль действует также как любой другой модуль. Т.о. имена создаваемые в оболочке также нах-ся внутри модуля.

Любые операции присваивания, выполняемые внутри ф-ции, классифицируют имена как локальные: инструкции =, import, def, передача аргументов и т.д.

Операции непосредственного изменения объектов не рассматривают имена как локальные - это свойственно только операциям присваивания.

Например, если имени L присвоен список, определенный на верхнем уровне в модуле, то такая инструкция как L.append(Х), внутри ф-ции не будет классифицировать имя L как локальное, тогда как инструкция L = X - будет. В первом случае происходит изменение объекта на которое указывает L, а не самого имени L - список будет найден в глобальной области видимости, как обычно, и будет изменен, без объявления имени global (или nonlocal).

Этот пример показывает различие м/у именами и объектами - т.е. видим, что операция, изменяющая объект, совсем не то, что операция присваивания объекта имени.

**Разрешение имен: правило LEGB.**

Три правила для def:

- Поиск имен ведется, самое большое, в четырех областях видимости: локальной, затем в объемлющей ф-ции (если она есть), затем в глобальной и, наконец, во встроенной.

- По умолч. операция присваивания создает локальные имена.

- Объявления global и nonlocal отображают имена на область видимости вмещающего модуля и ф-ции соответственно.

Все имена, присваемые значениями внутри def (или lambda), по умолч. явл-ся локальными; функции могут использ. имена в лексически объемлющих ф-циях и глобальной области, но чтобы изменять их, они должны быть объявлены нелокальными и глобальными.

Схема разрешения имен в Питон иногда наз-ся LEGB (по первым буквам областей видимости):

- Когда фнутри ф-ции идет обращение к имени, интерпретатор сначала ищет имя в локальной области (local, L), затем в локальной области объемлющей инструкции def (enclosing, E) или выражении lambda, затем в глобальной (global, G), и, наконец, во встроенной (built-in, B). Если имя так и не найдется, то появится сообщение об ошибке.

- Когда внутри ф-ции выполняется операция присваивания (а не обращение к имени внутри выражения), интерпретатор всегда создает или изменяет имя в локальной области, если в этой ф-ции оно не было объявлено глобальным или нелокальным.

- Когда выполняется присваивание имени за пределеами ф-ции (на уровне модуля или в интерактив. оболочке), локальная область совпадает с глобальной - с пространством имен модуля.

**Рисунок 17.1** Правило LEGB поиска имен в областях видимости. стр. 478

Отметим, что эти правила применимы только к простым переменным, типа spam. Далее в книге мы увидим, что полные имена атрибутов (как напр. object.spam) принадлежат определенным объектам и к ним применяются другие правила поиска. Поиск производится в одном или более объектах, а не областях видимости, что связано с механизмом под названием "наследование".

**Пример области видимости.**

>>> x = 99 # Глобальная область видимости

>>> # Х и func() определены в модуле: глобальная область

>>> def func(y): # Y и Z определены в функции: локальная область

# Локальная область видимости

z = x + y # Х - глобальная переменная

return z

>>> func(1) # func в модуле: вернет число 100

100

- ***Глобальные имена: Х и func.*** Х - глобальное, т.к. объявлено на верхнем уровне модуля, к нему можно обращаться внутри ф-ции без дополнительных объявлений. func - глобальное по тем же причинам. Инструкция def связывает объект ф-ции с именем func на верхнем уровне модуля.

- ***Локальные имена: Y и Z.*** Имена Y и Z явл-ся локальными (и сущ-ют только во время выполнения ф-ции), потому что присваивание им осущ-ся внутри определения ф-ции.

Суть разделения в том, что локальные переменные играют роль временных имен, на время вызова ф-ции.

Разделение имен на глобал. и локал. облегчает понимание ф-ций, т.к. большинство имен, используемых в ф-ции, появл-ся непосредственно в самой ф-ции, а не в произвольном месте модуля. Также мы будем уверены, что локал. имена не изменятся где-то в другом месте программы.

**Встроенная область видимости.**

Встроенная область видимости - это всего лишь встроенный модуль с именем builtins, но чтобы использовать это имя необходимо импортировать этот модуль, потому что само имя builtins не явл-ся встроенным.

Такой вот парадокс, встроенная область реализована как модул ьстандартной библиотеки с именем builtins, но само имя не находится во встроенной области, поэтому чтобы исследовать его нужно импортировать.

>>> import builtins

>>> dir(builtins)

['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',

... множество других имен пропущено, их очень много...

, 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

>>>

Имена в этом списке стоставляют встроенную област ьвидимости языка Питон. Примерно первая половина - это встроенные исключения, а вторая - встроенные функции.

Согласно правилу LEGB интерпретатор выполняет поиск в этом модуле в последнюю очередь. Все имена из этого списка мы получаем в свое распоряжение по умолч., чтобы их использ. не требуется импортировать какие-то модули.

Поэтому ест ьдва способа вызова встроенной функции:

>>> zip # Обычный способ

<class 'zip'>

>>> import builtins # Более сложный способ

>>> builtins.zip

<class 'zip'>

- второй способ иногда удобен при выполнении сложных действий.

Согласно правилу LEGB, как только будет найдено подходящее имя поиск прекращается, отсюда следует, что имена в локальной области могут переопределять переменные с такими же именами как в глобальной, так и во встроенной области, а глоабльные имена могут переопределять имена во встроенной области.

Напр. внутри ф-ции можно создать переменую open:

def hider():

open = 'spam' # Локальная переменная, переопределяет встроенное имя

...

open('data.txt') # В этой области видимости файл не будет открыт

Однако, в результате этого встроенная ф-ция с именем open, которая располагается во встроенной (внешней) области видимости, окажется скрытой.

Обычно это считается ошибкой, но неприятность в том, что Питон не сообщит об ошибке, ведь технически переопределение возможно.

Таким же способом ф-ции могут переопределять имена глобальных переменных:

X = 88 # Глобальная переменная Х

def func():

X = 99 # Локальная переменная Х: переопределяет глобальную

func()

print(X) # Выведет 88: значение не изменилось

- без объявления global (или nonlocal) в инструкции def, нельзя изменить переменную, расположенную за пределами ф-ции.

**Примечание.**

Касательно различий м/у 2.6 и 3.0. В 2.6 builtins наз-ся \_\_builtin\_\_. В большинстве глобальных областей присутствует имя \_\_builtins\_\_ ссылающееся на имя builtins.

В 3.0 выражение \_\_builtins\_\_ is builtins вернет True, а в 2.6 значение True вернет выражение \_\_builtins\_\_ is \_\_builtin\_\_. Т.е. мы можем исследовать встроенную область с помощью вызова dir(\_\_builtins\_\_) в обеих версиях, не импортируя модуль.

Однако в программах 3.0 рекомендуется использ. модуль builtins.

**Как испортить себе жизнь в Питон 2.6**

В 2.6 имена True и False всего лишь переменные во встроенной области. Их можно переопределить инструкцией True = False. Эта инструкция всего лишь переопределит True в единственной области видимости, остальные области будут использовать оригинал из встроенной (норм. вида).

А можно написать \_\_builtins\_\_.True = False и истина станет ложью для всей программы! В 3.0 такие фокусы ликвидировали, там эти слова зарезервированы.

**Инструкция global.**

global и nonlocal - единственные инструкции в Питон, напоминающие инструкции объявления. Они не объявляют тип или размер - они объявляют пространства имен.

global сообщает интерпретатору, что ф-ция будет изменять одно или более глобал. имен.

- Глобальные имена - имена, котороые определены на верхнем уровне вмещающего модуля.

- Глобальные имена должны объявляться, только если им будут присваиваться значения внутри ф-ции.

- Обращаться к глобальным именам внутри ф-ции можно и без объявления их глобальными.

global изменяет переменные, нах-ся на верхнем уровен модуля, за пределами def. nonlocal применяется к именам в локальных областях объемлющих инструкций def.

Записывается ключевое слово global и за ним одно или более имен, через запятую, которые будут отображены на област ьвидимости вмещающего модуля при обращении к ним или при присваивании внутри ф-ции:

X = 88 # Глобальная переменная Х

def func():

global X

X = 99 # Глобальная переменная Х: за пределами инструкции def

func()

print(X) # Выведет 99: значение не изменилось

- после объявления global имя Х внутри инструкции def ссылается на переменную Х за её пределами. Т.е. теперь оба имени ссылаются на одну переменную.

Более сложный пример:

y, z = 1, 2 # Глобальные переменные в модуле

def all\_global():

global x # Объявляется глобальной для присваивания

x = y + z # Объявлять y и z не нужно: применяется правило LEGB

- Здесь x, y, z используемые внутри all\_global() явл-ся глобальными. y, z считаются глобальными потому, что внутри ф-ции им не присваивается значение. х глоабльная потому что перечислена в инструкции global. Без инструкции global переменная х была бы локальной.

Внимание! y и z не были объявлены как глобальные, но следуя правилу LEGB, интерпретатор автоматически отыщет их в области видимости модуля.

Кроме того переменная х может не существовать в модуле в момент вызова ф-ции, присваивание создаст эту переменную.

**Минимизируйте количество глобальных переменных.**

Использование локальных переменных - это наиболее правильное решение. Изменение же глобальных переменных может привести к известным проблемам: когда значения переменных зависят от порядка, в котором вызываются функции, что осложняет отладку программы (или поиск причины ошибки).

Пример:

X = 99

def func1():

global X

X = 88

def func2():

global X

X = 77

- При применении этих двух функций значение Х будет зависеть от выбранного момента времени, т.к. будет зависеть от того какая ф-ция вызывалась последней.

Т.е. 88 или 77 - придется смотреть ВЕСЬ ход выполнения программы и при изменении модуля также придется держать в голове ВСЮ программу.

Это значительная проблема глобальных переменных.

С другой стороны, кроме случаев ООП и классов, глобал. переменные это один из самых удобных способов хранить информацию о состоянии (информация, которую нужно хранить между вызовами ф-ций), т.к. локал. переменные исчезают, а глобальные нет.

Можно использ. другие приемы для этого, но глобал. переменные для этого гораздо проще.

Также глобальные переменные можно хранить в одном глобальном модуле и использовать в многопоточных режимах, в роли памяти.

А пока лучше избегать глобальных переменных, старайтесь организовывать обмен данными через параметры и возвращаемые значения.

**Минимизируйте количество изменений в соседних файлах. стр. 484**

Несмотря на то, что можно непосредственно изменять переменные в другом файле, этого следует избегать.

Рассмотрим два модуля:

# first.py

X = 99 # Этот программный код ничего не знает о существовании second.py

# second.py

import first

print(first.X) # Ничего плохого нет, чтобы обратиться к имени в др. файле

first.X = 88 # Но изменение может привести к сложностям

- первый модуль определяет Х=99, второй импортирует первый модуль, чтобы использовать его переменную и затем присвоить ей новое значение.

Импортирование модуля для использ-я его переменных - это обязательное условие, т.к. каждый модуль обладает своим персональным пространством имен.

Т.о. глобальная облась модуля, после импорта, превращается в пространство имен атрибутов объекта модуля - импортирующий модуль при этом получает доступ к глобал. импортируемого модуля.

В целом нет ничего плохого такого обращения к переменным, тем более технич. возможно оно. Но такая тесная зависимость модулей, в лучшем случае приводит к снижению гибкости программы, а в худшем - к ошибкам.

Достаточно трудно держать такие свзяи в голове, а если модули кодируют разные программисты то вообще будет путаница.

В таких случаях свзяь м/у модулями лучше делать через вызовы функций, передавая им аргументы и получая возвращаемые значения:

# first.py

X = 99 # Этот программный код ничег оне знает о существовании second.py

def setX(new):

global X

X = new

# second.py

import first

first.setX(88)

Даже тот кто будет в первый раз читать код, поймет, что ф-ция setX - это часть интерфейса модуля, и поймет, что переменная Х может изменяться.

Эта ф-ция устраняет элемент неожиданности, который был в первом примере.

Мы не можем совсем отказаться от изменений в соседних файлах, но нужно свести их к минимуму и тщательно следить за этим.

**Другие способы доступа к глобальным переменным.**

Перевод глобал. переменных в атрибуты объекта модуля позволяет имитировать инструкцию global, импортируя вмещающий модуль и выполняя присваивание его атрибутам.

Пример: код нашего файла в одном случае импортирует вмещающий модуль по имени, в другом использ. таблицу загруженных модулей sys.modules:

#tnt.py

var = 99 # Глобальная переменная == атрибут модуля

def local():

var = 0 # Здесь изменяется локальная переменная

def glob1():

global var # Глобальное объявление (обычное объявление)

var += 1 # Изменяем глоабльную переменную var, теперь var = 100

def glob2():

var = 0 # Здесь изменяется локальная переменная

import tnt # Модуль tnt импортирует сам себя

tnt.var += 1 # Благодаря импорту мы можем изменить глобал. переменную, var = 101

def glob3():

var = 0 # Здесь изменяется локальная переменная

import sys # Импорт системной таблицы

glob = sys.modules['tnt'] # Получить объект модуля (или использ. \_\_name\_\_)

glob.var += 1 # Изменяется глобальная переменная, теперь var=102

def test():

print(var)

local(); glob1(); glob2(); glob3()

print(var)

Рассмотрим работу наших ф-ций и затем сделаем важные замечания:

>>> import tnt

>>> tnt.test()

99

102

>>> tnt.var

102

- все данные правильные, как в книге.

**Замечание!!!** Опишу выявленные мной особенности этого примера.

В книге, в ф-ции glob2 последняя строка была другая:

def glob2():

var = 0

import tnt

glob.var += 1

- при этом я получал вывод 99 и сразу сообщение об ошибке:

>>> import tnt

>>> tnt.test()

99

Traceback (most recent call last):

File "<pyshell#4>", line 1, in <module>

tnt.test()

File "C:\Python34\tnt.py", line 26, in test

local(); glob1(); glob2(); glob3()

File "C:\Python34\tnt.py", line 15, in glob2

glob.var += 1

NameError: name 'glob' is not defined

Почитав код я заменил glob на tnt и всё заработало! Удивительно, но факт. И либо в программе ошибка, либо у меня всё работает, но как-то по-другому чем у автора книги.

Но думаю, что я всё правильно исправил, т.к. вывод работы программы верный.

В этом примере мы написали много кода, такие способы дают больше кода чем могло быть при использ. global.

Т.о. знаем, что global дает возможность изменения переменных в модуле из ф-ций. nonlocal обеспечивает изменение переменных в объемлющих ф-циях и для её изучения мы приступим к вложенным ф-циям.

**Области видимости и вложенные функции.** стр. 487

С вложенными областями сталкиваются достаточно редко, но мы изучим их.

Иногда объемлющие области видимости называют статически вложенными областями видимости. На самом деле вложение явл-ся лексическими - вложенные области соотв-т физически вложенным блокам программного кода в исходных текстах программы.

**Вложенные области видимости.**

Внутри функции:

- Поиск имени Х сначала идет в локальной области (внутри самой ф-ции); затем в локальных областях всех объемлющих ф-ций (изнутри наружу); затем в текущей глоабльной области (в модуле), и, наконец, во встроенной области (модуль builtins). Поиск имен, объявленных в global, начинается сразу в глоабльной области видимости.

- Присваивание (X = value) по умолч. создает или изменяет имя Х в текущей локал. области. Если Х объявлено глоабльным внутир ф-ции, присв-ие создает или изменяет Х в области видимости объемлющего модуля. Если Х объявлено нелокальным внутри ф-ции, присв-ие создает или изменяет имя Х в ближ. области видимости объемлющей ф-ции.

global отображает имена в область видимости объемлющего модуля. Если есть влож. ф-ции можно получить значения переменных в объемл. ф-циях, но чтобы их изменить, переменные должны быть указаны в объявлении nonlocal.

**Примеры вложенных областей видимости.**

X = 99 # Имя в глоабл. области видимости: не используется

def f1():

X = 88 # Локальное имя в объемлющей ф-ции

def f2():

print(X) # Обращение к переменной во вложенной ф-ции

f2()

f1() # Выведет 88: локальная переменная в объемлющей ф-ции

- В этом примере вложенная инструкция def исполняется в момент вызова ф-ции f1 - она создает ф-цию и связывает её с именем f2, которое явл-ся локалбным и размещается в локал. области видимости ф-ции f1.

Неким образом f2 - это временная ф-ция, которая существует только во время работы (и видима только для программного кода) объемлющей ф-ции f1.

Внутри f2 - когда происходит вывод Х, она ссылается на переменную Х в локал. области объемлющей ф-ции f1. Ф-ции могут обращаться к именам, котореы физически располагаются в любых объемлющих инструкциях def, и имя Х в f2 автоматом отображается на имя Х в f1, по правилам LEGB.

Это правило работает даже если объемлющая ф-ция уже вернула управление:

def f1():

X = 88

def f2():

print(X) # Сохраняет значение Х в объемлющей области

return f2 # Возвращает f2, но не вызывает её

action = f1() # Создает и возвращает ф-цию

action() # Вызов этой ф-ции: выведет 88

- при вызове action запускается ф-ция, созданная во время выполнения ф-ции f1. Ф-ция f2 помнит переменную Х в области объемлющей ф-ции f1, которая уже неактивна.

**Фабричные функции.**

Замыкание или фабричная функция - под этим подразумевается объект ф-ции, который сохраняет значения в объемлющих областях видимости, даже когда эти области прекращают сущ-ие.

Классы лучше подходят для сохранения состояния, а эти функции просто альтернатива.

Пример, фабрич. ф-ции использ-ся когда нужно создавать обработчики событий прямо в процессе выполнения в соответствии с условиями (например нужно запретить пользователю вводить данные).

Пример:

>>> def maker(N):

def action(X): # Создать и вернуть функцию

return X \*\* N # Ф-ция action запоминает значение n в объемлющей

return action # области видимости

- здесь определяется внешняя ф-ция, которая создает и возвращает вложенную, не вызывая её.

>>> f = maker(2)

>>> f

<function maker.<locals>.action at 0x0227A618>

- если вызвать maker она просто вернет ссылку на вложенную ф-цию.

А теперь вызовем то, что было получено от maker:

>>> f(3) # Запишет 3 в X, в N по-прежнему хранится 2

9

>>> f(4)

16

- при этом была вызвана вложенная внутри maker ф-ция action.

Интересно то, что вложенная ф-ция хранит число 2, значение N в maker, даже при том, что ф-ция maker уже завершила свою работу.

На самом деле имя N из объемлющей локальной области сохр-ся как инф. о состоянии, присоединенная к action, и мы получаем обратно значение аргумента, возведенное в степень N (в нашем случае - в квадрат).

Если снова вызвать maker (внеш. ф-цию), получим новую фложенную ф-цию уже с др. инф. о состоянии, присоединенной к ней - и будет вычислятьтся куб, а ранее сохраненная ф-ция по-прежнему будет возвращать квадрат аргумента:

>>> g = maker(3) # ф-ция g хранит число 3, а f - число 2

>>> g(3) # куб аргумента

27

>>> f(3) # квадрат аргумента

9

>>>

Этот прием распространен среди программистов работающих с функциональными ЯП.

Объемлющие области мы встретим позднее, в выражениях lambda. Также прием вложения ф-ций обычно использ. при разработке декораторов.

Вообще, классы лучше подходят на роль "памяти". Также есть еще глобальные переменные, объемлющие области (текущая часть главы), и аргументы по умолчанию.

**Сохранение состояния объемлющей области видимости с помощью аргументов по умолчанию.**

def f1():

x = 88

def f2(x = x): # Сохраняет значение переменной х в объемлющей области

print(x) # в виде аргумента

f2()

f1() # Выведет 88

Конструкция arg = val в заголовке инструкции def означает, что аргумент arg по умолч. будет иметь значение val, если ф-ции не передается какого-либо другого значения.

Т.о. в нашем примере х по умолч. будет равен х в объемлющей области видимости.

всё это сложно и зависит от вычисления значения по умолч. Поэтому сейчас в Питон автоматическое сохранение любых значений в объемлющей области, для использ. во вложенных def.

Наилучшая рекомендация, это, конечно, стараться избегать вложенных инструкций def, если это возможно.

Рассмотрим пример:

>>> def f1():

x = 88

f2(x)

>>> def f2(x):

print(x)

>>> f1()

88

- Допустимо вызывать ф-цию (f2), определение которой в тексте программы нах-ся ниже ф-ции откуда производится вызов (f1), при условии, что вторая инструкция def (f2) будет исполнена до того, как первая ф-ция попытается вызвать её.

**Вложенные области видимости и lambda-выражения.**

lambda-выражение генерирует новую ф-цию, которая будет вызываться позднее, и оно похоже на инструкцию def.

Т.к. lambda это выражение оно может использоваться там где нельзя использ. def, например в литералах списков и словарей.

lambda также порождает локальную область видимости (как и def). lambda могут обращаться ко всем переменным, которые присутствуют в ф-циях, где находятся эти выражения.

def func():

x = 4

action = (lambda n: x \*\* n) # запоминается х из объемлющей инструкции def

return action

x = func()

print(x(2)) # Выведет 16 (4 \*\* 2)

Аналог этого примера:

def func():

x = 4

action = (lambda n, x=x: x \*\* n) # Передача х вручную

return action

Т.к. lambda это выражение, то они обычно вкладываются в инструкции def.

**Области видимости и значения по умолчанию применительно к переменным цикла.**

Исключение: если lambda-выражение или def вложены в цикл внутри др. ф-ции и влож. ф-ция ссылается на переменную из объемлющей области видимости, которая изм-ся в цикле, все ф-ции, созданные в этом цикле, будут иметь одно и то же значение - значение, которое имела переменная на последней итерации.

Пример, пытемся создать список ф-ций, каждая из которых запоминает текущее значение переменной i и объемлющей области:

>>> def makeActions():

acts = []

for i in range(5): # Сохранить каждое значение i

acts.append(lambda x: i \*\* x) # Все запомнят последнее значение i

return acts

>>> acts = makeActions()

>>> acts[0]

<function makeActions.<locals>.<lambda> at 0x01707AE0>

- Но желаемый результат мы не получим, потому что поиск переменной в объемлющей области производится позднее, при вызове влож. ф-ций, в рез-те все они получат одно и то же значение (значение, которое переменная имела на последней итерации). Т.е. каждая ф-ция в списке будет возвращать 4 во второй степени, потому что во всех них переменная i имеет одно значение:

>>> acts[0](2) # Все возвращают 4 \*\* 2, последнее значение i

16

>>> acts[2](2) # хотя должно быть 2 \*\* 2

16

>>> acts[3](2) # или 3 \*\* 2

16

В таком случае нужно явно сохранять значение из объемлющей области в виде аргумента со значением по умолч. Т.е. нужно передать текущее значение из объемлющей области в виде значения по умолчанию, которое вычисл-ся в момент создания вложенной ф-ции (а не когда она вызывается):

>>> def makeActions():

acts = []

for i in range(5): # использовать значения по умолчанию

acts.append(lambda x, i=i: i \*\* x) # Сохранить текущее значение i

return acts

>>> acts = makeActions()

>>> acts[0](2)

0

>>> acts[2](2)

4

>>> acts[3](2)

9

С подобным случаем можно столкнуться на практике, напр. в коде который генерирует функции-обработчики событий для элементов управления в графич. интерфейсе (нажатие кнопок напр.).

**Произвольное вложение областей видимости.**

Области видимости могут вкладываться произвольно, но поиск будет идти только в объемлющих ф-циях:

>>> def f1():

x = 99

def f2():

def f3():

print(x) # Будет найдена в области видимости f1 !

f3()

f2()

>>> f1()

99

Интерпретатор будет искать переменную в локальных областях всех объемлющих инструкций def, от внутр. к внеш., выше локал. и ниже глобал. области модуля.

Но на практике считается, что плоское лучше вложенного - лучше будет если вы сведете к минимуму количество вложенных определений ф-ций.

**Инструкция nonlocal.** стр. 494

nonlocal позволяет влож. ф-циям не только читать, но и изменять значения переменных в областях видимости объемлющих ф-ций.

nonlocal объявляет имена, которые будут изменяться в теле ф-ции и которые находятся в объемлющей области видимости. Отличие в том, что nonlocal применяется только к областям объемлющих ф-ций и не трогает глобал. область модуля.

Имена, перечисленные в nonlocal, должны фактическ исуществовать в области видимости, вмещающей ф-цию, где встречается это объявление - они могут сущ-ть только в объемлющей области и не могут быть созданы первой инструкцией присваивания влож. ф-ции.

Т.е. nonlocal позволяет присваивать значения переменным в объемлющих областях и ограничивает поиск таких имен областями видимости объемлющих ф-ций.

**Основы использования инструкции nonlocal.**

def func():

nonlocal name1, name2, ...

- nonlocal дает влож. ф-ции возможность присваивать именам (name1, name2) новые значения.

Т.о. для влож. ф-ций есть возможность поддерживать доступную для изменения информацию о состоянии, которая восстанавливается при последующих вызовах влож. ф-ции.

Также nonlocal вынуждает интерпретатор начинать поиск с областей видимости объемлющих ф-ций, пропуская локальную область видимости ф-ций. Т.е. nonlocal означает: - пропустить локальную область при поиске имен.

На практике, имена, которые в nonlocal, должны быть определены в объемлющих ф-циях к моменту, когда поток управления достигнет nonlocal, иначе буде тошибка.

nonlocal более строгая чем global, - она ограничивает область поиска тольк ообластями видимости объемлющих ф-ций.

Основное назначение nonlocal - не тольк ополучать, но и изменять значения переменных в объемлющих областях.

- global вынуждает интерпретатор начинать поиск с области объемлющего модуля и позволяет присваивать переменным новые значения. Поиск идет вплоть до встроенной области видимости.

- nonlocal ограничивает поиск областями видимости объемлющих ф-ций, и требует, чтобы имена, которые в nonlocal, уже существовали и может присваивать значения. В область поиска не входят глобал. и встроенная области видимости.

В Питон 2.6 нельзя присвоить новые значения, для этого использ-ся классы.

**Инструкция nonlocal в действии.**

Ф-ция tester создает и возвращает влож. ф-цию nested. Обращение к переменной state из влож. ф-ции отображается на локал. област ьвидимости ф-ции tester, с применением обычных правил поиска:

>>> def tester(start):

state = start # Обращение к нелокальным переменным

def nested(label): # действует как обычно

print(label, state) # Извлекает значение state из области

return nested # видимости объемлющей ф-ции

>>> F = tester(0)

>>> F('spam')

spam 0

>>> F('ham')

ham 0

По умолч. изменение перемененной в объемлющей области видимости не допускается:

>>> def tester(start):

state = start

def nested(label):

print(label, state)

state += 1 # По умолчанию не изменяется

return nested

>>> F = tester(0)

>>> F('spam')

...

UnboundLocalError: local variable 'state' referenced before assignment

**Использование nonlocal для изменения переменных.**

Если теперь state, локальную для tester, объявить в nested с помощью nonlocal, мы сможем изменять её внутри nested.

>>> def tester(start):

state = start # В каждом вызове сохраняется своё значение state

def nested(label):

nonlocal state # Объект state находится в объемлющей области

print(label, state)

state += 1 # Изменит значение переменной, объявленной как nonlocal

return nested

>>> F = tester(0) # start = state = 0

>>> F('spam') # label = 'spam' и будет увеличивать state при каждом вызове

spam 0

>>> F('spam')

spam 1

>>> F('spam')

spam 2

При каждом отдельном вызове tester будет создаваться отдельная копия переменной state.

Т.о. объект state, фактич. прикрепляется к возвращаемому объекту ф-ции nested - каждый вызов tester создает новый, независимый объект state, т.е. изменение state в одной ф-ции не будет вляить на другие. Пример ниже:

>>> G = tester(42) # новая функция с новой переменной state

>>> G('spam')

spam 42

>>> G('eggs')

eggs 43

>>> F('bacon') # но в ф-ции F state прежнее

bacon 3

**Граничные случаи.**

Первый момент - в отличии от имен в global, имена в nonlocal к моменту объявления уже должны существовать в области видимости объемлющей ф-ции, иначе - ошибка:

>>> def tester(start):

def nested(label):

nonlocal state # Нелокальные переменные должны существовать!

print(label, state)

state += 1

return nested

...

SyntaxError: no binding for nonlocal 'state' found

>>>

>>> def tester(start):

def nested(label):

global state # Глобальные переменные могут отсутствовать

state = 0 # Создаст переменную в области видимости модуля

print(label, state)

return nested

...

>>>

>>> F = tester(0)

>>> F('abc')

abc 0

>>> state

0

Второй момент - nonlocal огранивает область поиска имен только областями видимости объемлющих ф-ций - поиск нелокальных переменных не производится за пределами инструкции def ни в глобал. области модуля, ни во встроенной области, даже если переменные с такими именами там есть:

>>> spam = 99

>>> def tester():

def nested():

nonlocal spam # Переменная должна быть внутри def, а не в модуле!

print('Current=', spam)

spam += 1

return nested

...

SyntaxError: no binding for nonlocal 'spam' found

Именно из-за подобных случаев, во избежание путаницы, интерпретатор вынужден определять местоположение нелокальных имен в момент создания ф-ции, а не в момент её вызова.

**Как следует использовать инструкцию nonlocal?**

Основное - это решение проблем с сохранением информации с переменными в областях видимости объемлющих функций, напр. между вызовами ф-ций и методов.

Помним пример, где вызов ф-ции tester создает самостоятельный пакет информации, доступной для изменения, и где нет конфликта имен:

>>> def tester(start):

state = start

def nested(label):

nonlocal state

print(label, state)

state += 1

return nested

>>> F = tester(0)

>>> F('spam')

spam 0

Этот код будет работать только в Питон 3.0, для 2.6 нужны другие способы.

**Сохранение информации в глобальных переменных.**

В 2.6 и ранее, для предыдущего эффекта достаточно переместить переменную state в глобал. область видимости (область видимости модуля):

>>> def tester(start):

global state # Переместить в область видимости модуля

state = start

def nested(label):

global state

print(label, state)

state += 1

return nested

>>> F = tester(0)

>>> F('spam')

spam 0

>>> F('eggs')

eggs 1

Но здесь возникают некоторые ограничения. Глобальные переменные могут привести к конфликтам имен (это одна из ошибок при использ-ии глобал. переменных), также этот способ создает только одну копию информации о состоянии - если вызовем tester еще раз, то прежнее значение state будет сброшено в исходное состояние, т.е. предыдущие изменения будут затерты:

>>> G = tester(42) # Сбрасывает значение единств. копии state в глобал. области

>>> G('toast')

toast 42

>>> G('bacon')

bacon 43

>>> F('ham') # Значение state = 0 для F было затерто!

ham 44

**Сохранение информации с помощью классов (предварительное знакомство).**

Взглянем на комбинацию ф-ций tester/nested как на класс - информация о состоянии может быть записана в объекты явно, после их создания.

Нужно знать, что инструкция def внутри инструкции class действует точно так же, как и за её пределами. За исключением того, что ф-ция, определяемая внутри класса, автоматически получает аргумент self, ссылающийся на объект, относительно которого был сделан вызов (экземпляр класса, или объект, создается обращением к имени самого класса как к ф-ции):

>>> class tester:

def \_\_init\_\_(self, start): # Конструктор объекта,

self.state = start # сохранение информации в новом объекте

def nested(self, label):

print(label, self.state) # Явное обращение к информации

self.state += 1 # Изменения всегда допустимы

>>> F = tester(0) # Создаст экземпляр класса, вызовет \_\_init\_\_

>>> F.nested('spam') # Ссылка на F будет передана в аргументе self

spam 0

>>> F.nested('ham')

ham 1

>>> G = tester(42) # Каждый экземпляр получает свою копию информации

>>> G.nested('toast') # Изменения в одном объекте не сказываются на других

toast 42

>>> G.nested('bacon')

bacon 43

>>> F.nested('eggs') # В объекте F сохранилась прежняя информация

eggs 2

>>> F.state # Информация может быть получена за пределами класса

3

Можно перегрузить операторы. Если к экземпляру класса обратиться как к функции, то будет вызван метод \_\_call\_\_. Т.о. мы уберем необходимость вызова именованного метода:

>>> class tester:

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

self.state = start

def \_\_call\_\_(self, label): # Вызывается при вызове экземпляра

print(label, self.state) # поэтому нет необходимости в методе .nested()

self.state += 1

>>> H = tester(99)

>>> H('juice')

juice 99

>>> H('capkuke')

capkuke 100

Несмотря на то, что классы отличный способ хранения информации, они могут оказаться слишком тяжеловесными для простых случаев. А вложенные инструкции def, в некоторых случаях, могут быть даже быстрее классов.

**Сохранение информации в атрибутах функций.**

Добьемся эффекта nonlocal с помощью атрибутов функции - эти атрибуты, определяемые пользователем, присоединяются непосредственно к функции.

>>> def tester(start):

def nested(label):

print(label, nested.state) # nested - объемлющая область видимости

nested.state += 1 # Изменит атрибут, а не значение имени nested

nested.state = start # Инициализация после создания ф-ции

return nested

>>> F = tester(0)

>>> F('spam') # F - это функция 'nested'

spam 0 # с присоединенным атрибутом state

>>> F('ham')

ham 1

>>> F.state # Атрибут state доступен за пределами ф-ции

2

>>>

>>> G = tester(42) # G имеет собственный атрибут state,

>>> G('eggs') # отличный от одноименного атрибута ф-ции F

eggs 42

>>> F('ham')

ham 2

Этот пример опирается на тот факт, что имя nested явл-ся локал. переменной в области видимости ф-ции tester, включающей имя nested, - на это имя можно ссылаться и внутри ф-ции nested.

Также здесь использ-ся факт, что изменение самого объекта не явл-ся операцией присваивания.

**Задания по главе.** стр. 502

**Глава 18. Аргументы.** стр. 505

Здесь мы рассмотрим концепции Питон, связанные с передачей аргументов - способом передачи объект в функции.

**Передача аргументов.**

Передача аргументов производится посредством операции присваивания. Приведем нескольк оважных замечаний:

- Аргументы передаются через автоматическое присваивание объектов локальным переменным. Аргументы ф-ции - ссылки на объекты, которые (возможно) использ-ся совместно с вызывающей программой, - это разновидность присваивания. Объекты, которые перед-ся в виде аргументов, никогда не копируются автоматически.

- Операция присваивания именам аргументов внутри функции не оказывает влияния на вызывающую программу. При вызове ф-ции, имена аргументов становятся локал. именами в области видимости ф-ции.

- Изменение внутри ф-ции аргумента, который явл-ся изменяемым объектов, может оказывать влияние на вызывающую программу. Т.е. ф-ции могут воздействовать на полученные, в качестве аргументов, изменяемые объекты.

Т.о. всё, что касается ссылок относится и к аргументам.

Схема передачи аргументов в Питон (похожа на С):

- Неизменяемые объекты передаются "по значению". Напр. строки, числа, передаются в виде ссылок на объекты, а не в виде копий объектов. Но это очень напоминает копирование.

- Изменяемые объекты передаются "по указателю". Напр. списки, словари также передаются в виде ссылок на объекты. Изменяемые объекты могу тподвергаться изменению внутри ф-ций.

**Аргументы и разделяемые ссылки.**

>>> def f(a): # Имени а присваивается переданный объект

a = 99 # Изменяется только локальная переменная

...

>>> b = 88

>>> f(b) # Первоначально, имена a и b ссылаются на одно и то же число 88

>>> print(b) # Переменная b не изменилась

88

- в момент вызова f(b) перем. а присваивается объект 88, но а сущ-ет только внутри вызванной ф-ции и, изменение а внутри ф-ции, не оказывает влияние на коружение откуда вызвана ф-ция.

Этот пример и есть - "не является совмещением имен", а = 99 внутри ф-ции, не изменяет переменные, находящиеся в области видимости программного кода, вызывающего ф-цию.

Рассмотрим случай, когда аргументам передаются изменяемые объекты, с ними всё сложнее, здесь может быть изменение в вызывающей программе:

>>> def changer(a, b):

a = 2 # Изменяется только значение локального имени

b[0] = 'spam' # Изменяется непосредственно разделяемый объект

...

>>> X = 1

>>> L = [1, 2] # Вызывающая программа

>>> changer(X, L) # Передаются изменяемый и неизменяемый объект

>>> X, L

(1, ['spam', 2]) # Как видим Х не изменился, L - изменился

>>>

- первая операция присваивания изменяет только локальную переменную а, записывая в неё ссылку на совершенно другой объект, и не изменяет Х в вызывающей программе;

- b тоже локальная переменая, но в ней передается изменяемый объект, и поскольку, в этом случае, операция присваивания воздействует непосредственно на сам объект, то изменяется объект в вызывающей программе.

На самом деле изменяется не сам объект b, а часть объекта, на который ссылается аргумент b. И этот объхект изменяется потмоу что он продолжает существовать даже после завершения ф-ции.

**Рисунок 8.1** Иллюстрирует связи имя/объект, которые имеют место непосредственно сразу после вызова ф-ции, но перед тем как будет запущено её тело.

Вообще аргументы и ссылки, изученные нами ранее, тесно связаны.

>>> x = 1

>>> a = x # Разделяют один и тот же объект

>>> a = 2 # Изменяется только 'а', 'x' остается равным 1

>>> print(x)

1

>>> L = [1, 2]

>>> b = L # Разделяют один и тот же объект

>>> b[0] = 'spam' # Изменяется и сам L

>>> print(L)

['spam', 2]

**Как избежать воздействий на изменяемые аргументы.**

В Питон аргументы передаются в ф-ции по ссылкам, это и нужно, чаще всего. Т.е. так мы можем не создавать многие копии объектов.

А для избегания воздействий на вызывающую программу, как в предыдущих примерах, мы можем просто явно копировать объекты (см. гл. 6):

L = [1, 2]

changer(X, L[:]) # Передается копия, поэтому L не изменится

Моно сделать передачу копии прямо в ф-ции:

def changer(a, b):

b = b[:] # Входной список копируется, что исключает воздействие

# на вызывающую программу

a = 2

b[0] = 'spam' # Изменится только копия списка

Чтобы действительно предотвратить изменения можно преобразовать изменяемые объекты в неизменяемые:

L = [1, 2]

changer(X, tuple(L)) # Передается кортеж (преобразуется объект последовательности в кортеж),

# попытка изменения кортежа возбудит исключение

- однако такое решение накладывает на ф-цию ограничения (вдруг мы хотим не изменить список ,а применить другие методы, которые неприменимы к кортежам).

И вообще, такая особенность ф-ций не всегда плохо или ошибка, возможно мы специально хотим изменять изменяемые объекты и такой способ удобен.

Просто помните о том, что ф-ции могут изменять изменяемые объекты в вызывающей программе и следите за своим кодом внимательно.

**Имитация выходных параметров.**

Инструкция return может возвращать сразу неск-ко значений, упаковав их в кортеж или др. коллекцию. В Питон мы можем имитировать такое понятие из др. ЯП как - "передача аргументов по ссылке":

>>> def multiplex(x, y):

x = 2 # Изменяется только локальное имя

y = [3, 4]

return x, y # Новые значения возвр-ся в виде кортежа

>>> X = 1

>>> L = [1, 2]

>>> X, L = multiplex(X, L) # Результаты присваиваются именам

>>> X, L # в вызывающей программе

(2, [3, 4])

Выглядит как-будто ф-ция возвращает два значения, а на самом деле это один кортеж из двух элементов.

**Распаковывание аргументов в Питон 2.Х**

Ф-ции, объявленной как:

def f((a, (b, c))) :

- можно передать кортежи, соответствующие ожидаемой структуре. Т.е. вызвать f((1, (2, 3))) и a, b, c присваиваются значения 1, 2, 3.

В Питон 3.0 такая форма не поддерживается. Нужно делать так:

def f(T): (a, (b, c)) = T

**Специальные режимы сопоставления аргументов.** стр. 511

Аргументы всегда передаются через присваивание. Однако есть доп. возможности для создания дополнительных способов, которыми объекты аргументов сопоставляются с именами аргументов в заголовке ф-ции.

По умолч. сопоставление происходит в соотв-ии с позициями аргументов, слева направо, и аргументов должно быть столько сколько имен указано в заголовке ф-ции.

И сейчас мы познакомимся с другими способами.

**Основы.**

Эти спец. режимы не явл-ся обязательными и имеют отношение только к сопоставлению объектов и имен, основной механизм передачи аргументов по-прежнему это присваивание.

Фактически некоторые режимы нужны для разработчиков библиотек, а не приложений, но мы должны с ними ознакомиться:

- *Спосоставление по позиции: значения и имена ставятся в соответствие по порядку, слева направо. Вообще это режим по умолчанию, с ним мы знакомы.*

- *Споставление по именам: соответствие определяется по указанным именам аргументов.* Вызывающая программа указывает соотвествие м/у аргументами и их значениями в момент вызова ф-ции, синтаксис: name = value.

- *Значения по умолчанию: указываются значения аргументов, которые могут не передаваться.* Это использ. если вдруг передается неполное кол-во аргументов, синтаксис также: name = value.

- *Переменное число аргументов: прием произвольного числа аргументов, позиционных или именованных.* Можно использ. спец. аргумент, перед именем которго стоит один или два символа \*, для объединения произвол. кол-ва аргуметов в коллекцию (часто наз-ся varargs).

- *Переменное число аргументов: передача произвольного числа аргументов, позиционных и именованных.* Вызывающая программа также может использ. \* для распаковки коллекции в отдельнеы аргументы.

- *Только именованные аргументы: аргументы, которые должны передаваться только по имени.* В 3.0 (но не в 2.6) могут определяться аргументы, передаваемые по имени, а не по позиции.

**Синтаксис сопоставления.**

**Таблица 18.1 Виды сопоставления аргументов функций.** стр. 512

Эти режимы делятся на случаи вызова функции и определения функции:

- В инструкции вызова ф-ции при использ. простых значений соотв-ие именам аргументов опред-ся по позиции, а с формой name = value соотв-ие опред-ся по именам аргументов - наз-ся передачей именованных аргументов. Формы \*sequence и \*\*dict в вызоывх ф-ций дают передавать произвольное число объектов по позиции или по именам в виде последовательностей или словарей.

- В заголовке ф-ции с простыми значениями соответствие идет по позиции и по имени, а с формой name = value определяются значения по умолчанию. С формой \*name все доп. позиционные аргументы объединяются в кортеж, а с формой \*\*name все доп. именованные аргументы объединяются в словарь. В 3.0 и выше обычные аргументы и аргументы со знач. по умолчанию, следующие за формой \*name изи за символом \*, явл-ся именованными аргументами, которые, при вызове ф-ции, передаются по имени.

Наиболее часто используется форма передачи именованных аргументов и аргументов со значениями по умолчанию. Мы встречались с ними ранее:

- Именованные аргументы использ-сь для передачи необязательных параметров ф-ции print, именованные аргументы позволяют указывать значения аргументов вместе с их именами, что дает больше смысла вызову ф-ции;

- Значения по умолчанию встречались когда мы изучали передачу значений из объемлющей области видимости, это позволяет определять необязательные аргументы и указывать значения по умолчанию в определении ф-ции.

**Тонкости сопоставления.**

Несколько правил:

- В вызове ф-ции порядок аргументов таков: любые позиционные аргументы (значения), за которыми могут следовать любые именованные аргументы (name = value) и аргументы в форме \*sequence, за которым могут следовать аргументы в форме \*\*dict.

- В заголовке ф-ции порядок аргументов таков: любые обычные аргументы (name), за которыми могут следовать аргументы со значениями по умолчанию (name = value), за которым следуют аргументы в форме \*name (или \* в 3.0), если имеются, за которыми могут следовать любые имена или пары name = value аргументов, которые передаются только по имени, за которыми могут следовать аргументы в форме \*\*name.

В обоих случаях форма \*\*arg должна следовать последней.

Также отметим, что порядок аргументов важен и если его нарушить будет вызвана ошибка.

Ход действий интерпретатора:

1. Сопоставление неименованных аргументов по позициям.

2. Сопоставление именованных аргументов по именам.

3. Сопоставление дополнительных неименованных аргументов с кортежем \*name.

4. Сопоставление дополнительных именованных аргументов со словарем \*\*name.

5. Сопоставление значений по умолчанию с отсутствующими именованными аргументами.

- Далее интепретатор убеждается, что каждому аргументу соотв-ет только одно значение (иначе возбуждается исключение!) и связывает имена аргументво с полученными объектами.

В Питон 3.0 имена аргументов в загловке ф-ции могут также снабжаться аннотациями в форме name:value (или name:value=default, если имеется значение по умолчанию).

**Примеры использования именованных аргументов и значений по умолчанию.** стр. 515

По умолчанию сопоставление идет слева направо, по позиции:

>>> def f(a, b, c): print(a, b, c)

>>> f(1, 2, 3)

1 2 3

- в этом примере все просто: а соотв-ет 1, b - 2, и т.д.

**Именованные аргументы.**

Определяют соответствие по именам, а не по позициям:

>>> f(c=3, b=1, a=2)

2 1 3

- b=3 здесь означает, что значение 2 передается ф-ции в аргументе с именем b, т.е. интерпретатор сопоставляет имя b в вызове ф-ции с именем аргумента b в заголовке определения ф-ции и затем передает значение 2 в этот аргумент.

Обратив внимание, что порядок следования именованных аргументов не имеет значения.

Можно объединить передачу по позициям и по именам аргументов:

>>> f(1, c=3, b=2)

1 2 3

Самые используемые роли именованных аргументов такие: делают вызовы ф-ций более описательными (если имена будут более звучные чем a, b, c: - напр. name='Bob', age=40 и т.п.) и вторая роль - они используются вместе со значениями по умолчания, о которых мы поговорим далее.

**Значения по умолчанию.**

Приведем краткий наглядный пример:

>>> def f(a, b=22, c=33): print(a, b, c)

...

>>> f(11)

11 22 33

>>> f(a=12)

12 22 33

При таком вызове мы обязаны передать значение аргумента а (по позиции или по имени), а значения b и c можно опустить.

Можно передать и два аргумента, или все три:

>>> f(10, 14)

10 14 33

>>> f(3, 4, 5)

3 4 5

- в последнем случае оба значения по умолч. просто не будут использованы.

Еще пример, с комбинированием режимов передачи значений и т.н. "перепрыгиванием" через аргументы:

>>> f(7, c=55)

7 22 55

- здесь у нас 7 - аргумент по позиции, 55 - по имени, а 22 - по умолчанию.

Главное не путайте конструкции name=value в заголовке ф-ции и в вызове. Это разные синатксические конструкции.

**Комбинирование именованных аргументов и значений по умолчанию.**

Пример, вызывающая программа всегда должна передавать ф-ции как минимум два аргумента (spam и eggs), два др. аргумента необязательны, в случае отсутствия они принимаются по умолчанию:

>>> def func(spam, eggs, toast=0, ham=0):

print(spam, eggs, toast, ham)

...

>>> func(1, 2)

1 2 0 0

>>> func(1, ham=1, eggs=0)

1 0 0 1

>>> func(spam=1, eggs=0)

1 0 0 0

>>> func(toast=1, eggs=2, spam=3)

3 2 1 0

>>> func(1, 2, 3, 4)

1 2 3 4

- Еще раз обратите внимание, что порядок следования именованных аргументов не важен. Важно, чтобы были значения для spam и eggs, а сопоставление может быть и по именам и по позициям. Форма name=value имеет разный смысл в вызове ф-ции и в инструкции def (именованный аргумент - в вызове и значение по умолчанию - в заголовке).

**Примеры произвольного числа аргументов.** стр. 518

\* и \*\* пердназначены для передачи произвольного числа аргументов ф-циям. Оба варианта могут появл-ся как в определениях ф-ций так и в вызовах, в обоих случаях у них сходные назначения.

**Сбор аргументов в коллекцию.**

Пример, выполняется сборка лишних позиционных аргументов в кортеж:

>>> def f(\*args): print(args)

...

При вызове ф-ции интерпретатор соберет все позиционные аргументы в новый кортеж и присвоит его переменной args, с этим кортежем можно работать как с обычным объектом кортежа:

>>> f()

()

>>> f(1)

(1,)

>>> f(1, 2, 3, 4)

(1, 2, 3, 4)

Комбинация \*\* применяется при передаче именованных аргументов - в этом случае аргументы будут собраны в новый словарь, с ним также можно работать как с обычным словарем.

Форма \*\* позволяет преобразовать аргументы, передаваемые по именам, в словари, которые можно обойти с помощью метода keys и т.п. итераторами:

>>> def f(\*\*args): print(args)

...

>>> f()

{}

>>> f(a=1, b=2)

{'b': 2, 'a': 1}

Пример, в заголовках ф-ций можно комбинировать \* и \*\*, здесь у нас число 1 передается как позиционный аргумент, 2 и 3 объединяются в кортеж pargs с позициями, а x и y помещаются в словарь kargs с именами:

>>> def f(a, \*pargs, \*\*kargs): print(a, pargs, kargs)

>>> f(1, 2, 3, x=1, y=2)

1 (2, 3) {'y': 2, 'x': 1}

Все эти формы можно комбинировать в еще более сложные формы, а сейчас обратимся к формам \* и \*\* в вызовах ф-ций.

**Извлечение аргументов из коллекции.**

Форма \* в вызовах ф-ций распаковывает, а не создает коллекцию аргументов. Напр. передаем четыре аргумента в виде кортежа и интерпретатор их распаковывает:

>>> def func(a, b, c, d): print(a, b, c, d)

...

>>> args = (1, 2)

>>> args += (3, 4)

>>> func(\*args)

1 2 3 4

Форма \*\* в вызовах распаковывает словари пар ключ/значение в отдельные аргументы, которы епередаются по ключу:

>>> args = {'a':1, 'b':2, 'c':3}

>>> args['d'] = 4

>>> func(\*\*args)

1 2 3 4

Можно комбинировать позиционные и именованные аргументы:

>>> func(\*(1, 2), \*\*{'d':4, 'c':4})

1 2 4 4

>>> func(1, \*(2, 3), \*\*{'d':4})

1 2 3 4

>>> func(1, c=3, \*(2,), \*\*{'d':4})

1 2 3 4

>>> func(1, \*(2, 3), d=4)

1 2 3 4

>>> func(1, \*(2,), c=3, \*\*{'d':4})

1 2 3 4

Такие способы использ. когда мы заранее не знаем сколько аргументов может быть передано ф-ции.

И также не путайте конструкции \*/\*\* в заголовках ф-ции и в их вызовах (сборка лишних аргументов в коллекцию и распаковка коллекции соответственно).

В вызовах ф-ций в форме \*pargs можно передавать любые итерирумеые объекты, даже файл - func(\*open('fname')).

**Обобщенные способы вызова ф-ций.**

Разные способы получения аргументов помогают тем, что нам заранее не нужно знать кол-во обязательных аргументов в ф-ции.

Например, нам в программе можно использовать if для выбора из множества ф-ций и списков аргументов и вызывать любую из них единообразным способом:

if <test>:

action, args = func1, (1,) # Вызвать func1 с 1 аргументом

else:

action, args = func2, (1, 2, 3) # Вызвать func2 с 3 аргументами

...

action(\*args) # Фактический вызов универсальным способом

Вообще передачу произвольного числа аргументов удобно использовать всегда, когда заранее не известен перечень аргументов.

Напр. пользователь выбирает ф-цию с помощью польз-го интерфейса, может быть невозможно нам записать явный вызов ф-ции в програм. коде. Для решения сделать список операциями над списками и вызвать ф-цию, с пом. синтаксиса произвольного числа аргументов:

>>> args = (2, 3)

>>> args += (4,)

>>> args

(2, 3, 4)

>>> func(\*args)

...

TypeError: func() missing 1 required positional argument: 'd'

>>> args += (5,)

>>> func(\*args)

2 3 4 5

Список аргументов программа может создать во время выполнения. Пример, реализация вызова произвольных ф-ций с любым кол-вом аргументов, передавая их все:

def tracer(func, \*pargs, \*\*kargs): # Принимает произвольные аргументы

print('calling:', func.\_\_name\_\_)

return func(\*pargs, \*\*kargs) # Передает все полученные аргументы

def func(a, b, c, d):

return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4))

При вызове tracer все аргументы будут собарны в коллекцию и переданы требуемой ф-ции:

calling: func

10

**Исчезнувшая встроенная функция apply (Питон 2.6).**

В 2.6 роль форм \* и \*\* выполняла ф-ция apply (в 3.0 не работает), т.е. след. два примера эквивалентны:

func(\*pargs, \*\*kargs) # Новый синтаксис вызова

apply(func, pargs, kargs) # Устаревшая функция

Пример:

>>> def echo(\*args, \*\*kargs): print(args, kargs)

...

>>> echo(1, 2, a=3, b=4)

(1, 2) {'b': 4, 'a': 3}

В 2.6 этот пример выглядел бы так:

>>> pargs = (1, 2)

>>> kargs = {'a':3, 'b':4}

>>> apply(echo, pargs, kargs)

(1, 2) {'b': 4, 'a': 3}

>>> echo(\*pargs, \*\*kargs)

(1, 2) {'b': 4, 'a': 3}

>>> echo(0, c=5, \*pargs, \*\*kargs) # Новый синтаксис 3.0 более универсален к тому же

(0, 1, 2) {'b': 4, 'a': 3, 'c':5}

**Питон 3.0: аргументы, которые могут передаваться только по именам.** стр. 522

В 3.0 мы можем определять аргументы, которые могут передаваться только в форме именованных аргументов и никогда позиционных.

Такие особенные аргументы оформляются в виде обычных именованных, следующих за формой \*args в списке аргументов.

Пример, а - может быть как именованным так и позиционным, b - собираются все дополнительные аргументы, а с - только как именованный:

>>> def kvon(a, \*b, c):

print(a, b, c)

...

>>> kvon(1, 2, c=3)

1 (2,) 3

>>> kvon(a=1, c=3)

1 () 3

>>> kvon(1, 2, 3)

...

TypeError: kvon() missing 1 required keyword-only argument: 'c'

Чтобы показать, что ф-ция не принимает списки аргументов, можно использ-ть одиночный символ \*, при все аргументы после \* будут передавать по именам:

>>> def kvon(a, \*, b, c):

print(a, b, c)

...

>>> kvon(1, c=3, b=2)

1 2 3

>>> kvon(c=3, b=2, a=1)

1 2 3

>>> kvon(1, 2, 3)

...

TypeError: kvon() takes 1 positional argument but 3 were given

>>> kvon(1)

...

TypeError: kvon() missing 2 required keyword-only arguments: 'b' and 'c'

Для именованных аргументов после \* можно также указывать значения по умолчанию:

>>> def kvon(a, \*, b='spam', c='ham'):

print(a, b, c)

...

>>> kvon(1)

1 spam ham

>>> kvon(1, c=3)

1 spam 3

>>> kvon(a=1)

1 spam ham

>>> kvon(c=3, b=2, a=1)

1 2 3

>>> kvon(1, 2)

...

TypeError: kvon() takes 1 positional argument but 2 were given

А теперь посмотрим как именованные аргументы комбинируются, по умолч. также остаются необязательными:

>>> def kvon(a, \*, b, c='ham'):

print(a, b, c)

...

>>> kvon(1, b='eggs')

1 eggs ham

>>> kvon(1, c='eggs')

...

TypeError: kvon() missing 1 required keyword-only argument: 'b'

>>> kvon(1, 2)

...

TypeError: kvon() takes 1 positional argument but 2 were given

>>> def kvon(a, \*, b=1, c, d=2):

print(a, b, c, d)

...

>>> kvon(3, c=4)

3 1 4 2

>>> kvon(3, c=4, b=5)

3 5 4 2

>>> kvon(3)

...

TypeError: kvon() missing 1 required keyword-only argument: 'c'

>>> kvon(1, 2, 3)

...

TypeError: kvon() takes 1 positional argument but 3 were given

**Правила, определяющие порядок следования.**

Одно замечание, аргументы с обязательной передачей по именам должны идти после одной звездочки \*, но не после двух \*\*, и два символа \*\* не должны быть пусты:

>>> def kvon(a, \*\*pargs, b, c)

SyntaxError: invalid syntax

>>> def kvon(a, \*\*, b, c)

SyntaxError: invalid syntax

Т.о. аргументы только по имени должны идти после \* но перед \*\*

>>> def f(a, \*b, \*\*d, c=6): print(a, b, c, d) # Ошибка, после \* должн быть

SyntaxError: invalid syntax # хоть один аргумент, а не \*\*

>>> def f(a, \*b, c=6, \*\*d): print(a, b, c, d) # Кооллекции аргументов

# в заголовке

>>> f(1, 2, 3, x=4, y=5) # Используется значение по умолч.

1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, 2, 3, x=4, y=5, c=7) # Переопределение знач. по умолч.

1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, 2, 3, c=7, x=4, y=5) # Среди именованных аргументов

1 (2, 3) 7 {'y': 5, 'x': 4}

>>> def f(a, c=6, \*b, \*\*d): print(a, b, c, d) # c - не явл-ся только именованным аргументом!

...

>>> f(1, 2, 3, x=4)

1 (3,) 2 {'x': 4}

В вызовах ф-ций похожие правила порядка аргументов. Только именованные аргументы должны располагаться перед \*\*. При этом аргументы, которые могут передаваться только по именам, могут располагаться как перед \* так и после неё, а также могут включаться в словарь \*\*:

>>> def f(a, \*b, c=6, \*\*d): print(a, b, c, d) # Только именованные аргументы

# м/у \* и \*\*

>>> f(1, \*(2, 3), \*\*dict(x=4, y=5)) # Распаковывание аргументов при вызове

1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, \*(2, 3), \*\*dict(x=4, y=5), c=7) # Именованные аргументы после \*\* !

SyntaxError: invalid syntax

>>> f(1, \*(2, 3), c=7, \*\*dict(x=4, y=5)) # Переопределение знач. по умолч.

1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, c=7, \*(2, 3), \*\*dict(x=4, y=5)) # Перед \* или после нее

1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, \*(2, 3), \*\*dict(x=4, y=5, c=7)) # Только именованные аргументы внутри \*\*

1 (2, 3) 7 {'y': 5, 'x': 4}

**Когда используются аргументы, которые могут передаваться только по именам?**

Они упрощают создание ф-ций, которые принимают произвольное кол-во позиционных аргументов и параметры настройки, передаваемые в виде именованных аргументов. Можно и без них, но тогда больше кода и работы.

Допустим есть ф-ция, которая принимает много аргументво и еще флаг трассировки:

process(X, Y, Z) # Используется значение флага по умолчанию

process(X, Y, notify=True) # значение флага определяется явно

Без аргумента, который передается только по имени, нам пришлось бы задействовать и форму \*args и \*\*args и вручную проверять аргументы.

Следующее поределение ф-ции гарантирует, что ни один позиционный аргумент не будет сопоставлен с notify и передает его по имени:

def process(\*args, notify=False): ...

**Функция поиска минимума.** стр. 525

Теперь к практике.

Допустим нам нужна ф-ция способная находить минимальное значение из множества (ноль и более) аргументов с разными типами данных.

Мы можем собрать аргументы в кортеж и выполнить их обход с помощью for. Все типы поддерживают операцию сравнения и нам даже не нужно учитывать тип аргументов (полиморфизм, однако) - т.е. мы можем просто сравнить объекты и дать интерпретатору самому выбрать корректную операцию сравнения.

**Основное задание.**

Три способа реализации:

- Первая реализация: извлекает первый аргумент (args - кортеж) и обходит остальную часть коллекции, отсекая первый элемент (т.к. нам не нужно сравнивать этот элемент сам с собой);

- Вторая реализация: интерпретатор сам выбирает первый аргумент и остаток, благодаря чему отпадает необходимость извлекать первый аргумент и получать срез;

- Третья реализация: преобразует кортеж в список с помощью встроенной ф-ции list и использует метод списка sort.

sort может обеспечивать высокую производительность, но линейный характер сканирования обеспечивает первым двум вариантам более высокую скорость.

Запишем все три варианта в файл mins.py:

def min1(\*args):

res = args[0]

for arg in args[1:]:

if arg < res:

res = arg

return res

def min2(first, \*rest):

for arg in rest:

if arg < first:

first = arg

return first

def min3(\*args):

tmp = list(args)

tmp.sort()

return tmp[0]

print(min1(3, 4, 1, 2))

print(min2('bb', 'aa'))

print(min3([2, 2], [1, 1], [3, 3]))

>>> import mins

1

aa

[1, 1]

Отметим, что ни один вариант не проверяет - вдруг ф-ции не передается ни одного аргумента; такую проверку можно сделать, но интерпретатор сам выдаст исключение если не будет ни одного аргумента.

В первом случае исключение будет при попытке получить нулевой элемент, во втором - когда обнаружится несоответствие списка аргументов, в третьем - когда ф-ция попытается вернуть нулевой элемент.

**Дополнительные баллы.**

А если нам теперь нужны максимальные значения?

В первых двух версиях достаточно изменить < на >, а третья версия должна возвращать не tmp[0], а tmp[-1]. Ну и плюсом будет изменить имя ф-ции на max (это немного юмора).

К тому же можно обощить ф-цию так, что она будет искать либо минимум либо максимум, определяя отношение элементов за счет интерпретации строки выражения с помощью таких средств, как ф-ция eval, или передавая произвольную ф-цию сравнения:

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 getthan(x, y): return x > y

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

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

>>> import minmax

1

6

Конечно это мы делали просто упражнение, в Питно ф-ции min и max уже есть. Эти примеры могут быть распространены и на другие случаи.

**Универсальные функции для работы с множествами.** стр. 528

Ранее мы писали ф-цию, которая возвращает пересечение двух последовательностей (отбирает общие элементы).

Ниже мы напишем ф-цию, которая возвращает пересечение произвольного числа последовательностей, где использ. \*args. Все аргументы передаются в тело ф-ции в составе кортежа, поэтмоу использ. просто цикл for.

Также напишм ф-цию union, возвращающую объединению, которая также принимает произвольное число аргументов и собирает вместе все элементы, имеющиеся в любом операнде:

Сохраняем наши ф-ции в файл inter.py. Обе ф-ции принимают аргументы в виде кортежа и работают со всеми видами последовательностей.

def intersect(\*args):

res = []

for x in args[0]: # Сканировать первую послед-ть

for other in args[1:]: # Во всех остальных аругментах

if x not in other: break # Общий элемент?

else: # Нет: прервать цикл

res.append(x) # Да: добавить элемент в конец

return res

def union(\*args):

res = []

for seq in args: # Для всех аргументов

for x in seq: # Для всех элементов

if not x in res:

res.append(x) # Добавить новый элемент в результат

return res

>>> from inter import intersect, union

>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"

>>> intersect(s1, s2), union(s1, s2)

(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])

>>> intersect([1, 2, 3], (1, 4))

[1]

>>> intersect(s1, s2, s3)

['S', 'A', 'M']

>>> union(s1, s2, s3)

['S', 'P', 'A', 'M', 'C', 'L']

Автор пишет, что появление множеств в Питон делает эти ф-ции ненужными в общем-то. Хотя почему не нужны я не знаю.. Наверное потому что множества можно сравнивать между собой на пересечения, а другие типы последовательностей преобразовывать в множества для этого.

**Имитация функции print в Питон 3.0** стр. 530

Код, который мы напишем здесь, будет работать в Питон 2.6 и ниже (ну а нам придется проверять его в 3.0, или потом проверим в 2.6 из дома) для имитация интерфейса ф-ции print в Питон 3.0.

На самом деле в 2.6 можно включить поддержку ф-ции print, выполнив инструкцию:

from \_\_future\_\_ import print\_function

Но для практической работы мы напишем нашу ф-цию print для 2.6. Опишем реализацию нашей ф-ции print30 (30 потомоу что print в 2.6 зарезервированное слово) в файле print30.py:

"""

Имитация ф-ции print в 3.0

Сигнатура вызова: print30(\*args, sep=' ', end='\n', file=None)

"""

import sys

def print30(\*args, \*\*kargs):

sep = kargs.get('sep', ' ')

end = kargs.get('end', '\n')

file = kargs.get('file', sys.stdout)

output = ''

first = True

for arg in args:

output += ('' if first else sep) + str(arg)

first = False

file.write(output + end)

Замечание! Пришлось преобразовать код в кодировку utf-8 без BOM (не знаю почему я выбрал " без BOM", просто так делали в одном из скринкастов по фреймворку kohana и я решил сделать также). Без преобразования я получал ошибку:

>>> from print30 import print30

Traceback (most recent call last):

File "<pyshell#16>", line 1, in <module>

from print30 import print30

File "C:\Python34\print30.py", line 4

"""

^

SyntaxError: (unicode error) 'utf-8' codec can't decode byte 0xc8 in position 0: invalid continuation byt

"""

И теперь (когда всё с кодировками нормально) проверим её работу, для этого запишем тест в файл testprint30.py:

from print30 import print30 # Эта строка должна быть в самом файле testprint30

# если ввести её просто в интерпретаторе

# - все равно не будет работать, нужно именно в файле

print30(1, 2, 3)

print30(1, 2, 3, sep='') # Подавляем вывод разделителя

print30(1, 2, 3, sep='...')

print30(1, [2], (3,), sep='...') # Вывод объектов различ. типов

print30(4, 5, 6, sep='', end='') # Подавить вывод символа новой строки

print30(7, 8, 9)

print30() # Добавить новую строку

import sys

print30(1, 2, 3, sep='??', end='.\n', file=sys.stderr) # Перенаправить в файл

И запускаем на проверку, мне было лень открывать командную строку и поэтому запустил тест в IDLE с помощью import:

>>> import testprint30

1 2 3

123

1...2...3

1...[2]...(3,)

4567 8 9

1??2??3.

**Использование аргументов, которые могут передаваться только по имени.**

Пример ф-ции выше можно реализовать в 3.0 с применением аргументов, которые могут передаваться только по имени - для автоматич. проверки доп. параметров настройки. Запишем её в файл print33.py:

# Использоване аргументов, которые могут передаваться только по имени

import sys

def print33(\*args, sep=' ', end='\n', file=sys.stdout):

output = ''

first = True

for arg in args:

output += ('' if first else sep) + str(arg)

first = False

file.write(output + end)

Эта версия print, в отличие от оригинальной, не сможет игнорировать все лишние именованные аругменты, а вызовет исключение:

>>> from print33 import print33

>>> print33(99, name='bob')

...

TypeError: print33() got an unexpected keyword argument 'name'

Чтобы выявить лишние именованные аргументы вручную, мы могли извлекать допустимые параметры с помощью метода dict.pop() и по окончании проверять размер словаря.

Приведем измененную версию нашей ф-ции print30:

# Удаляет допустимые именованные аргументы со значениями по умолчанию

import sys # Этой строки не было в книге, а без неё работает не так программа

def print30(\*args, \*\*kargs):

sep = kargs.pop('sep', ' ')

end = kargs.pop('end', '\n')

file = kargs.pop('file', sys.stdout)

if kargs: raise TypeError('extra keywords: %s' % kargs)

output = ''

first = True

for arg in args:

output += ('' if first else sep) + str(arg)

first = False

file.write(output + end)

>>> from print30 import print30

>>>

>>> print30(99, name='Bob')

...

TypeError: extra keywords: {'name': 'Bob'}

Конечно версия ф-ции с именованными аргументами имеет меньше кода (print33), но она работает только в 3.0 (а там уже есть нормальная ф-ция print), поэтому в 2.6 придется использовать более громоздкий код (print30), если это нам понадобится когда-либо.

**Именованные аргументы.**

Специальные режимы сопоставления не обязательны в общем-то. Однако они все-таки использ-ся в некоторых инструментах Питон.

Имен. арг-ты играют важную роль в модуле tkinter (разработка графич. интерфейса в Питон), напр. вызов:

from tkinter import \*

widget = Button(text="Press me", command=someFunction)

- создает новую кнопку и определяет текст на ней и функцию обратного вызова, с помощью аргументов text и command.

Т.к. у графич. компонентов может быть много параметров, именованные аргументы позволяют указывать только необходимые параметры (иначе прийдется указать все).

Многие ф-ции имеют ожидают именованные аргументы для режима работы. Напр. ф-ция sorted:

sorted(iterable, key=None, reverse=False)

- ожидает итерируемы объект для сортировки и доп. имен. аргументы, определяющие ключ словаря (для сортировки) и сортировку в обратном порядке, которые по умолч. отключены.

**Контрольные вопросы.** стр. 534

**Глава 19. Расширенные возможности функций.** стр. 536

**Концепции проектирования функций.**

У начинающих кодеров часто возникают проблемы выбора: как лучше связать элементы м/у собой, напр. как разложить задачу на ф-ции (связность), как должны взаимодействовать ф-ции (взаимодействие) и т.д.

Мы должны учитывать и размер ф-ций (удобство сопровождения кода).

Кратко рассмотрим некоторые основные правила для тех, кто начинает осваивать Питон:

- **Взаимодействие: для передачи значений функции используйте аргументы, для возврата результатов - инструкцию return.** Это ограничит внешнее воздействие на ф-цию небольшим числом известных мест в программе.

- **Взаимодействие: используйте глобальные переменные, только если это действительно необходимо.** Старайтесь по минимуму использ. глобал. переменные, идеально - вообще обходиться без них.

- **Взаимодействие: не воздействуйте на изменяемые аргументы, если вызывающая программа не предполагает этого.** Использование и воздействие на изменяемые аргументы делает ф-цию слишком специфичной и неустойчивой, т.к. появл-ся счлишком тесная связб м/у ф-цией и вызывающей программой.

- **Связность: каждая функция должна иметь единственное назначение.** Ф-ция должна решать одну задачу (но никак не "эта ф-ция выполняет всю программу" или "эта ф-ция делает то-то И то-то"). Разбивайте сложные ф-ции на простые. Иначе повторное использ. ф-ции будет невозможно (а это один из паттернов Питон).

- **Размер: каждая функция должна иметь относительно небольшой размер.** Пишите ф-ции простыми и короткими, не нужно писать на неск-ко экранов.

- **Взаимодействие: избегайте непосредственного изменения переменных в другом модуле.** Такое изменение делает свзяь м/у модулями очень тесной (также как и изменение глобал. переменных из функции), что влечет за собой высокую сложность понимания и многократного использования. Вместо таких изменений модулей используйте функции доступа.

**Рисунок 19.1** Схема организации взаимодействий ф-ций с внешним миром. стр. 537

Конечно из всех правил есть исключения и мы познакомимся с ними в книге далее.

В любом случае нужно стремиться минимизировать внешние зависимости функций и других компонентов программ.

**Рекурсивные функции.** стр. 538

Рекурсивные - это функции, которые могут вызывать сами себя, прямо или косвенно, образуя цикл.

Рекурсия это сложная тема, но тем не менее достаточно полезна, она даже явл-ся альтернативой простым циклам и итерациям, хотя и не всегда более проста и быстра.

**Вычисление суммы с применением рекурсии.**

Сумму можно вычислить с помощью sum или написать свою ф-цию:

>>> def mysum(L):

if not L:

return 0

else:

return L[0] + mysum(L[1:])

...

>>> mysum([1, 2, 3, 4, 5])

15

На каждом уровне рекурсии эта функция вызывает саму себя, чтобы получить сумму остатка списка, которая складывается с первым элементом. Рекурсивный цикл заканчивается и возвращается ноль, когда ф-ция получит пустой список. При этом на каждом уровне создается собственная переменная L.

Добавим вывод L на каждом уровне рекурсии, для наглядности:

>>> def mysum(L):

print(L)

if not L:

return 0

else:

return L[0] + mysum(L[1:])

...

>>> mysum([1, 2, 3, 4, 5])

[1, 2, 3, 4, 5]

[2, 3, 4, 5]

[3, 4, 5]

[4, 5]

[5]

[]

15

- т.о. видим, что список становится всё меньше, а сумма вычисляется уже в процессе обратного раскручивания рекурсии.

>>> def mysum(L):

print(L)

if not L:

return 0

else:

summ = (L[0] + mysum(L[1:]))

print(summ)

return summ

...

>>> mysum([1, 2, 3, 4, 5])

[1, 2, 3, 4, 5]

[2, 3, 4, 5]

[3, 4, 5]

[4, 5]

[5]

[]

5

9

12

14

15

15

**Альтернативные решения.**

def mysum(L):

return 0 if not L else L[0] + mysum(L[1:]) # Трехместный оператор

def mysum(L):

return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # Суммирует любые типы,

# предполагает наличие

# хотя бы одного значения

def mysum(L):

first, \*rest = L # Использует расширенную

return first if not rest else first + mysum(rest) # операцию присваивания

# последовательностей в 3.0

Последние две ф-ции будут завершаться ошибкой, при получении пустого списка, но зато находят сумму любых типов объектов:

>>> mysum([1])

1

>>> mysum([])

...

IndexError: list index out of range

>>> mysum([1, 2, 3, 4, 5])

15

>>> mysum(['spam', 'ham', 'eggs'])

'spamhameggs'

Кстати, третья версия может работать с любыми итерируемыми объектами, включая открытые для чтения файлы. Перва я вторая такого не могут, т.к. опираются на операцию индексирования.

Обратим внимание, что рекурсия может быть прямой (как у нас) и косвенной (ф-ция вызывает др. ф-цию, которая вызывает ф-цию вызвавшую её), т.о. выполняется два вызова вместо одного:

>>> def mysum(L):

if not L: return 0

return nonempty(L) # Вызов функции, которая вызовет эту функцию

>>> def nonempty(L):

return L[0] + mysum(L[1:]) # Косвенная рекурсия

>>> mysum([1.1, 2.2, 3.3, 4.4])

11.0

**Инструкции циклов вместо рекурсии.**

Цикл while, для наших примеров может быть более простым:

>>> L = [1, 2, 3, 4, 5]

>>> sum = 0

>>> while L:

sum += L[0]

L = L[1:]

>>> sum

15

Инструкция for, с точки зрения использования памяти и скорости выполнения, также более подходит нам чем рекурсия:

>>> L = [1, 2, 3, 4, 5]

>>> sum = 0

>>> for x in L: sum += x

>>> sum

15

**Обработка произвольных структур данных.**

С др. стороны, рекурсия (или алгоритмы на основе стека) могут реализовывать обход структур данных с произвольной организацией.

Например посчитать сумму из структуры с вложенными списками:

[1, [2, [3, 4], 5], 6, [7, 8]]

Простые циклы здесь не подойдут, вложенные циклы также не подойдут, а поможет рекурсия:

def sumtree(L):

tot = 0

for x in L: # Обход элементов одного уровня

if not isinstance(x, list):

tot += x # Числа суммируются непосредственно

else:

tot += sumtree(x) # Списки обрабатываются рекурсивными вызовами

return tot

L = [1, [2, [3, 4], 5], 6, [7, 8]]

print(sumtree(L)) # Выведет 36

Далее в книге мы увидим более практичные примеры:

- глава 24, сценарий reloadall.py, выполняет обход цепочек импортирования.

- глава 28, сценарий classtree.py, выполняыет обход деревьев наследования классов.

- глава 30, сценарий lister.py, также выполняет обход деревьев наследования классов

И запомним, что в случае линейных итераций нужно использ. циклы, но иногда рекурсия бывает незаменима.

**Функции - это объекты: атрибуты и аннотации.** стр. 542

Ф-ции в Питон явл-ся больше чем блок инструкций, - они явл-ся настоящими объектами. Могут передаваться между частями программы и вызываются косвенно.

**Косвенный вызов функций.**

Мы можем писать программы, которые будут работать с ф-циями как с обычными объектами. У них даже есть спец. операции: они могут вызываться перечислением аргументов в круглых скобках, следующих сразу же за выражением ф-ции.

Имя, в инструкции def, это просто переменная в текущей области видимости. Когда def выполнена, имя ф-ции становится всего лишь ссылкой на объект - ее можно присвоить да. именам и вызывать ф-цию по любому из них:

>>> def echo(message): # Имени echo присваивается объект ф-ции

print(message)

...

>>> echo('Direct call') # Вызов объекта ф-ции по оригинальному имени

Direct call

>>> x = echo # Теперь на ф-цию ссылается еще и имя х

>>> x('Indirect call') # Вызов той же самой ф-ции по новому имени х

Indirect call

Ф-ции можно передавать другим ф-циям в виде аргументов:

>>> def indirect(func, arg): # Наша ф-ция (func) окажется в аргументах др. ф-ции

func(arg)

...

>>> indirect(echo, "Argument call!") # Пример понятен

Argument call!

Есть возможность наполнять структуры данных ф-циями, как если бы они были простыми числами или строками:

>>> schedul = [(echo, 'Spam!'), (echo, 'Ham!')]

>>>

>>> for (func, arg) in schedul:

func(arg) # Вызов ф-ции, сохраненной в контейнере

...

Spam!

Ham!

- в этом примере просто вып-ся обход списка shedul и вызывается ф-ция echo с одним аргументом, таже обратим внимание на присваивание кортежа в заголовке for.

Ф-ции могут создаваться и возвращаться другими ф-циями:

>>> def make(label):

def echo(message):

print(label + ':' +message)

return echo

>>> F = make('Spam')

>>> F('Ham!')

Spam:Ham!

>>> F('Eggs!')

Spam:Eggs!

**Интроспекция функций.**

Т.к. ф-ции обычные объекты, мы можем оперировать с ними обычными операциями. Например создание и вызов:

>>> def func(a):

b = 'spam'

return b \* a

>>> func(8)

'spamspamspamspamspamspamspamspam'

Можем получить базовый доступ к атрибутам ф-ций:

>>> func.\_\_name\_\_

'func'

>>> dir(func)

['\_\_annotations\_\_', '\_\_call\_\_', '\_\_class\_\_', '\_\_closure\_\_', '\_\_code\_\_', '\_\_defaults\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_get\_\_', '\_\_getattribute\_\_', '\_\_globals\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_kwdefaults\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_name\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_qualname\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_']

Механизмы интроспекции дают исследовать реализацию ф-ции, например каждая ф-ция имеет присоединенный объект с программным кодом, т.о. можно получить список локальных переменых и аргументов:

>>> func.\_\_code\_\_

<code object func at 0x022507A0, file "<pyshell#35>", line 1>

>>> dir(func.\_\_code\_\_)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'co\_argcount', 'co\_cellvars', 'co\_code', 'co\_consts', 'co\_filename', 'co\_firstlineno', 'co\_flags', 'co\_freevars', 'co\_kwonlyargcount', 'co\_lnotab', 'co\_name', 'co\_names', 'co\_nlocals', 'co\_stacksize', 'co\_varnames']

>>> func.\_\_code\_\_.co\_varnames

('a', 'b')

>>> func.\_\_code\_\_.co\_argcount

1

**Атрибуты функций.**

К атрибутам полученым выше (вернее в ф-циям, имеющим эти атрибуты) можно присоединять и свои атрибуты:

>>> func

<function func at 0x0227B588>

>>> func.count = 0

>>> func.count += 1

>>> func.count

1

>>> func.handles = 'Button-Press'

>>> func.handles

'Button-Press'

>>>

>>> dir(func)

['\_\_annotations\_\_', '\_\_call\_\_', '\_\_class\_\_', '\_\_closure\_\_', '\_\_code\_\_', '\_\_defaults\_\_',

...сотальные имена опущены...

'\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'count', 'handles']

Такие атрибуты (свои) можно использовать для хранения информации о состоянии непосредственно в объекте ф-ции.

В некотором роде атрибуты можно рассматривать как имитацию "статистических локальных переменных", имеющихся в др. ЯП, - переменных, которые явл-ся локальными для ф-ции, но сохраняют свои значения после выхода из ф-ции. Атрибуты связаны с объектами, а не с областями видимости, но эффект использования такой же как и областей.

**Аннотации функций в Питон 3.0**

В 3.0 (но не 2.6) можно присоединять к объектам ф-ций краткое описание (аннотацию) - произвольные данные об аргументах ф-ции и о возвращаемом значении.

Аннотации не обязательны, но если прсиутствуют, то просто сохр-ся в атрибутах \_annotations\_\_ объектов ф-ций и могут использ-ся.

Пример неаннотированной ф-ции с 3 аргументами:

>>> def func(a, b, c):

return a + b + c

>>> func(1, 2, 3)

6

Синтаксически, аннотации нах-ся в заголовках инструкций def, ассоциированные с аргументами и возвращаемыми значениями. После аргументов ставятся двоеточия и затем аннотация, а для возвр. значения - после символов -> за списком аргументов, см. пример:

>>> def func(a: 'spam', b: (1, 10), c: float) -> int:

return a + b + c

>>> func(1, 2, 3)

6

Если аннотации есть, интерпретатор соберет их в словарь и присоединит к объекту ф-ции, ключи и значения будут понятны из примера:

>>> func.\_\_annotations\_\_

{'b': (1, 10), 'return': <class 'int'>, 'c': <class 'float'>, 'a': 'spam'}

Т.к. аннотация это всего лишь объект, присоед-й к др. объекту, её можно обрабатывать. Пример, аннотированы два аргумента из трех и затем выполняем обход аннотаций:

>>> def func(a: 'spam', b, c: 99):

return a + b + c

>>> func(1, 2 ,3)

6

>>> func.\_\_annotations\_\_

{'c': 99, 'a': 'spam'}

>>> for arg in func.\_\_annotations\_\_:

print(arg, '=>', func.\_\_annotations\_\_[arg])

c => 99

a => spam

Обратите внимание на моменты: в аннотированных аргументах можно указывать и значения по умолч., при этом аннотация и символ : нах-ся перед = и знач. по умолч.:

>>> def func(a: 'spam' = 4, b: (1, 10) = 5, c: float = 6)-> int:

return a + b + c

# Все аргументы получили знач. по умолч. и аннотации

>>> func(1, 2, 3)

6

>>> func() # Все аргументы получают знач. по умолч. 4 + 5 + 6 = 15

15

>>> func(1, c=10) # 1 + 5 + 10

16

>>> func.\_\_annotations\_\_

{'b': (1, 10), 'return': <class 'int'>, 'c': <class 'float'>, 'a': 'spam'}

Причем пробелы в заголовке ф-ции не обязательны, они добавлены для удобочитаемости кода:

>>> def func(a:'spam'=4, b:(1,10)=5, c:float=6)->int:

return a + b + c

>>> func(1, 2) # 1 + 2 + 6

9

>>> func.\_\_annotations\_\_

{'b': (1, 10), 'return': <class 'int'>, 'c': <class 'float'>, 'a': 'spam'}

Аннотации могут применяться различными способами, напр. для наложения ограничений на типы или знач. аргументов - что использ-ся для регистрации информации об интерфейсе.

Аннотации рассматриваются как альтернатива аргументам декораторов функций (когда информация располагается за пределами заголовка ф-ции и не ограничичвается единственной ролью).

Обратив внимание - аннотации могут указываться только в инструкциях def - они не могут использ. в lambda-выражениях, потому что синтаксис lambda уже огранич-т область использ-я ф-ций, определяемых таким образом.

**Анонимные функции: lambda.** стр. 548

Подобно def выражение lambda создает ф-цию, которая будет вызываться позднее, но в отличие от def, выражение возвращает ф-цию, а не связывает её с именем.

Поэтому lambda иногда наз-т анонимными (безымянными) ф-циями. Они част оиспольз. чтобы получить встроенную ф-цию или отложить выполнение фрагмента программного кода.

**Основы lambda-выражений.**

Синтаксис: lambda, затем один или более аргументов и далее выражение (почти как def):

lambda argument1, argument2 ... argumentN : выражение, использующее аргументы

Отличия lambda от ф-ций создаваемых инструкцией def:

- **lambda - это выражение, а не инструкция.** Т.е. lambda может появляться там, где синтаксис Питона не позволяет использовать def - внутри литералов или в вызовах ф-ций. Также lambda возвращает значение (ф-цию), которое можно присвоить переменной, тогда как def связывает ф-цию с именем в заголовке, а не возвращает в виде рез-та.

- **Тело lambda - это не блок инструкций, а единственное выражение.** Тело lambda сродни тому, что мы помещаем в return внутри def. Т.о. в теле lambda может быть реализована только логика, не использующая такие инструкции как if, это сделано чтобы lambda были простыми, т.к. для сложных случаев у нас есть def.

Пример работы def и lambda:

>>> def func(x, y, z): return x + y + z

...

>>> func(2, 3, 4)

9

>>> f = lambda x, y, z: x + y + z

>>> f(2, 3, 4)

9

- f присваивается объект ф-ции, созданный lambda-выражением.

В lambda можно указывать аргументы по умолч.:

>>> x = (lambda a ="fee", b="fie", c="foe": a + b + c)

>>> x("wee")

'weefiefoe'

>>> x()

'feefiefoe'

Правила поиска переменных для lambda такие же как и для def:

>>> def kan():

title = 'Sir'

action = (lambda x: title + ' ' + x) # Заголовок в объемлющей def

return action # Возвращает ф-цию

>>> act = kan()

>>> act('Robin')

'Sir Robin'

**Когда можно использовать lambda-выражения?**

Они удобны для создания очень маленьких ф-ций, также позволяют встраивать определения ф-ций в программный код, который их использует. Тем самым упрощают сценарии где нужен небольшой код.

Также эти выражения используются для создания таблиц переходов, т.е. списков или словарей действий, выполняемых по требованию:

L = [lambda x: x\*\*2 # Встроенные определения функций

lambda x: x\*\*3

lambda x: x\*\*4] # Список из трех ф-ций

for f in L:

print(f(2)) # выведет 4, 8, 16

print(L[0](3)) # выведет 9

Также lambda наиболее полезны в качестве краткого варианта инструкции def, чтобы использовать там где def неприменимо.

Чтобы повторить пример выше нам пришлось бы создать три ф-ции с помощью def:

def f1(x): return x\*\*2 # Определение именованных функций

def f2(x): return x\*\*3

def f3(x): return x\*\*4

L = [f1, f2, f3] # Ссылка по имени

for f in L:

print(f(2)) # выведет 4, 8, 16

print(L[0](3)) # выведет 9

Подобные таблицы можно создавать с помощью словарей и др. структур данных:

>>> key = 'got'

>>> {'already': (lambda: 2 + 2),

'got': (lambda: 2 \* 4),

'one': (lambda: 2 \*\* 6)}[key]()

8

- здесь, в момент когда интерпретатор создает словарь, каждое из влож. lambda генерирует и оставляет после себя ф-цию для последующего использ. - обращение по ключу извлекает одну из этих ф-ций, а круглые скобки обеспечивают вызов извлеченной ф-ции.

Чтобы такое реализовать без lambda, нужно написать три отдельные инструкции def за пределами словаря и ссылаться на них по именам:

>>> def f1(): return 2 + 2

>>> def f2(): return 2 \* 4

>>> def f3(): return 2 \*\* 6

>>> key = 'one'

>>> {'already': f1, 'got': f2, 'one': f3}[key]()

64

Близость программного кода, которую дают lambda-выражения, особенно полезна, когда ф-ции использ. в единственном месте (и не использ. где-то еще в коде) - как в нашем примере.

lambda удобно использ. в списках аргументов ф-ций - для определения временных ф-ций, которые больше нигде не использ. в сценарии.

**Как (не) запутать программный код на Питон.**

Большую часть инструкций в Питон можно представить в виде выражений. Это помогает обходить ограничение lambda на одно выражение.

Например нужно вывести информацию из тела lambda-выражения, можно просто записать sys.stdout.write(str(x)+'\n') вместо print(x).

Таким же способом в lambda можно заложить логику в виде трехместного выражения if/else или эквивалент (более слож.) - комбинацию операторов and/or.

Инструкцию:

if a:

b

else:

c

Можно представить двумя разными выражениями:

b if a else c

((a and b) or c)

И эти выражения можно помещать внутрь lambda:

>>> lower = (lambda x, y: x if x < y else y)

>>> lower('bb', 'aa')

'aa'

>>> lower('aa', 'bb')

'aa'

А если внутри lambda нужно выполнить цикл, то их можно заменить ф-цией map и генераторами списков:

>>> import sys

>>> all = lambda x: list(map(sys.stdout.write, x))

>>> t = all(['spam\n', 'toast\n', 'eggs\n'])

spam

toast

eggs

>>> all = lambda x: [sys.stdout.write(line) for line in x]

>>> t = all(('bright\n', 'side\n', 'of\n', 'life\n'))

bright

side

of

life

Используйте подобные хитрости очень внимательно, т.к. они усложняют понимание кода. Просто и понятно - старайтесь писать код так, для сложного использ. def. Но умеренное использ. lambda может быть даже полезно.

**Вложенные lambda-выражения и области видимости.**

Пример, lambda находится внутри инструкции def - и потому получает значение имени х из области видимости объемлющей ф-ции (поиск в области видимости влож. ф-ции):

>>> def action(x):

return (lambda y: x + y ) # Создать и вернуть ф-цию, запомнить х

>>> act = action(99)

>>> act

<function action.<locals>.<lambda> at 0x0228A348>

>>> act(2) # Вызвать ф-цию, созданную ф-цией action

101

Для понимания такой работы поиска lambda представим предыдущую инструкцию def в виде lambda:

>>> action = (lambda x: (lambda y: x + y))

>>> act = action(99)

>>> act(3)

102

>>> ((lambda x: (lambda y: x + y))(99))(4)

103

Эта структура lambda-выражений создает ф-цию, которая при вызове создает др. ф-цию. Выглядит замысловато и лучше избегат ьвложений lambda друг в друга.

**Функции обратного вызова.**

lambda применяются в определении ф-ций обратного вызова для tkinter GUI API.

Пример, создаем кнопку, нажатие на которую выводи тсообщение в консоль (модуль tkinter должен быть доступен на компьютере):

import sys

from tkinter import Button, mainloop

x = Button(

text = 'Press me'

command = (lambda: sys.stdout.write('Spam\n')))

x.pack()

mainloop()

Здесь, в качестве обработчика события, регистрируется ф-ция, сгенерированная lambda. Преимущество перед def в том, что обработчик события нажатия находится прямо здесь, в вызове ф-ции, создающей кнопку.

lambda откладывает выполнение обработчика, пока не произойдет событие - write произойдет когда кнопка будет нажата, а не когда создана.

Т.к. правила областей видимости применяются и к lambda их удобно применять пр иобращении к спец. аргументу экземпляра self, который явл-ся локальной переменной в объемлющих методах классов:

class MyGui:

def makewidget(self):

Button(command = (lambda: self.onPress("spam")))

def onPress(self, message):

...текст сообщения...

**Отображение функций на последовательности: map** стр. 554

Одна из частых задач, это применение некоторой операции к каждому элементу списка или др. последовательности.

Пример, обновление всех счетчиков в списке с помощью for:

>>> counters = [1, 2, 3, 4]

>>>

>>> update = []

>>>

>>> for x in counters:

update.append(x+10)

>>> update

[11, 12, 13, 14]

Ф-ция map упрощает этот пример, она применяет указанную ф-цию (inc) к каждому элементу послед-ти (counters) и возвращает итерируемый объект (в Питон 3.0, в 2.6 возвращает список), поэтому оборачиваем в list:

>>> def inc(x): return x + 10

>>> list(map(inc, counters))

[11, 12, 13, 14]

Также в map можно встретить lambda-выражения:

>>> list(map((lambda x: x+3), counters))

[4, 5, 6, 7]

Такой вариант использ-я map можно представить в виде цикла for, поэтому мы можем написать эквивалент этой утилиты:

>>> def mymap(func, seq):

res = []

for x in seq: res.append(func(x))

return res

А теперь проверим работу ф-ции inc, с map и нашей mymap:

>>> list(map(inc, [1, 2, 3])) # Встроенная map возвращает итератор

[11, 12, 13]

>>> mymap(inc, [1, 2, 3]) # А наша mymap возвращает список

[11, 12, 13]

Но преимущество map в том, что она встроенная и доступна всегда и всегда работает одинаково.

К том же есть специфические ситуации, напр. имеется неск-ко аргументов с последовательностями, а map извлекает их параллельно и передает как отдельные аргументы в ф-цию:

>>> pow(3, 4) # 3\*\*4

81

>>> list(map(pow, [1, 2, 3], [2, 3, 4])) # 1\*\*2, 2\*\*3, 3\*\*4

[1, 8, 81]

При передаче неск-ких последовательностей ф-ция map предполагает, что ей будет передана ф-ция, принимающая N аргументов для N последовательностей.

Здесь ф-ция pow при каждом вызове принимает от map два аргумента - по одному для каждой послед-ти.

Отличие map от генераторов списков в том, что map применяет к каждому элементу послед-ти не произвольное выражение, а ф-цию. Вследствии этого она обладает меньшей гибкостью. Однако иногда и более высокой произодительностью чем генераторы.

**Средства функционального программирования: filter и reduce.**

map - это простейший представитель класса встроенных ф-ций в Питон, используемых в функциональном программировании, т.е. ф-ций, которые применяют др. ф-ции к последовательностям и др. итераторам.

Родственные ей ф-ции filter - фильтрует элементы с помощью ф-ций, и применяют ф-ции к парам элементов - reduce:

>>> list(range(-5, 5)) # Итератор в 3.0

[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(filter((lambda x: x > 0), range(-5, 5))) # Итератор в 3.0

[1, 2, 3, 4]

Элементы, для которых ф-ция возвращает истину, добавл-ся в список (итератор) результатов.

filter некий аналог for, но она встроенная ф-ция и более быстрая:

>>> res = []

>>> for x in range(-5, 5):

if x > 0:

res.append(x)

>>> res

[1, 2, 3, 4]

Ф-ция reduce в 3.0 была перемещена в модуль functools и принимает итератор, но сама возвращает не итератор, а одиночный объект.

Пример где reduce помогает вычислить сумму и произведение элементов списка:

>>> from functools import reduce

>>>

>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])

10

>>> reduce((lambda x, y: x \* y), [1, 2, 3, 4])

24

На каждом шаге reduce передает текущую сумму или произведение со след. элементом списка lambda-функции. По умолч. первый элемент послед-ти принимается в качестве начального знач.

Эквивалент вычисления суммы с циклом for:

>>> L = [1, 2, 3, 4]

>>> res = L[0]

>>> for x in L[1:]:

res = res + x

>>> res

10

Напишем свою имитацию ф-ции reduce:

>>> def myreduce(func, seq):

tally = seq[0]

for next in seq[1:]:

tally = func(tally, next)

return tally

>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])

15

>>> myreduce((lambda x, y: x \* y), [1, 2, 3, 4, 5])

120

Кроме всего этого reduce может принимать необязательный третий аргумент, которыый использ. в качестве начального значения и служит знач. по умолч., когда послед-ть не содержит ни одного элемента.

>>> reduce((lambda x, y: x + y), [], 55)

55

Загляните в модуль operator, который содержит ф-ции, соотв-щие встроенным выражениям, которые могут пригодиться при сипольз-ии некоторых функциональных инструментов:

>>> import operator, functools

>>>

>>> functools.reduce(operator.add, [2, 4, 6]) # Оператор сложения в виде функции

12

>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])

12

Мощные приемы функционального программирования с ф-циями map, filter, reduce вы можете дополнить lambda-выражениями и генераторами списков.

**Глава 20. Итераторы и генераторы.** стр. 560

**Еще раз о генераторах списков: функциональные инструменты.**

Отображение операций на последовательности и сбор результатов (ф-ции map и filter, напр.) стало настолько популярной задачей в Питон, что в нем появилась новая особенность - генераторы списков.

Генераторы - применяют к элементам произвольные выражения, вместо ф-ций.

Мы уже знакомились ранее с генераторами, здесь мы изучим их проведя аналогии с альтернативами, основанными на применении ф-ций.

**Генераторы списков и ф-ция map.**

Вспомним ф-цию ord, возвращающую целочисленный код ASCII единственного символа (обратной ей явл-ся ф-ция chr - возвращает символ, соответствующий коду ASCII):

>>> ord('s')

115

А теперь мыхотим получить коды всех символов в строке, для этого нам поможет for:

>>> res = []

>>>

>>> for x in 'spam':

res.append(ord(x))

>>> res

[115, 112, 97, 109]

Но тоже самое мы можем сделать и с mаp, причем короче:

>>> res = map(ord, 'spam') # Применить ф-цию к послед-ти

>>> list(res)

[115, 112, 97, 109]

То же самое можно сделать и с помощью генератора, только он отображает на послед-ть выражение, а не ф-цию:

>>> res = [ord(x) for x in 'spam']

>>> res

[115, 112, 97, 109]

Заметим, что генераторы возвращают список, а не итератор как map. Синтаксически генераторы заключаются в квадратные скобки. Самый простой вид генератора мы опсиали выше.

Еще пример:

>>> [x \*\* 2 for x in range(10)]

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

- создается список квадратов от 0 до 9. Чтобы получить тот же результат с map нам пришлось бы написать ф-цию, напр. используя lambda:

>>> list(map((lambda x: x \*\* 2), range(10)))

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

**Добавление проверок и вложенных циклов: функция filter.**

Как мы знаем, после for можно добавлять if для реализации логики.

Генераторы с if можно представить как аналог filter, т.е. пропускаются элементы, для которых if возвращает "ложь".

Пример, три версии выбора четных чисел в диапазоне от 0 до 4:

>>> [x for x in range(5) if x % 2 == 0]

[0, 2, 4]

>>> list(filter((lambda x: x % 2 == 0), range(5)))

[0, 2, 4]

>>> res = []

>>> for x in range(5):

if x % 2 == 0:

res.append(x)

>>> res

[0, 2, 4]

Во всех примерах есть оператор деления по модулю (остаток от деления) %, так мы вычисляем четные числа.

Генератор позволяет объединить if и выражение, получается эффект filter или map в одной строке:

>>> [x \*\* 2 for x in range(10) if x % 2 == 0]

[0, 4, 16, 36, 64]

Эквивалент с map, будет сложнее и длиннее:

>>> list(map((lambda x: x\*\*2), filter((lambda x: x % 2 == 0), range(10))))

[0, 4, 16, 36, 64]

Генераторы списков дают возможность запрограммировать любое число вложенных циклов for, каждый из которых может сопровождаться собственным оператором if с условным выражением.

[ expression for target1 in sequence1 [if condition]

for target2 in sequence2 [if condition] ...

for targetN in sequenceN [if condition] ]

Вложенные операторы for в генераторах действуют точно так же, как влож. инструкции for.

Пример:

>>> res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]

>>> res

[100, 200, 300, 101, 201, 301, 102, 202, 302]

Пример более объемного эквивалента:

>>> res = []

>>> for x in [0, 1, 2]:

for y in [100, 200, 300]:

res.append(x + y)

>>> res

[100, 200, 300, 101, 201, 301, 102, 202, 302]

Итерации в генераторах могут по любым послед-м и итерируемым объектам. Напр. обход строк и конкатенация:

>>> [x + y for x in 'spam' for y in 'SPAM']

['sS', 'sP', 'sA', 'sM', 'pS', 'pP', 'pA', 'pM', 'aS', 'aP', 'aA', 'aM', 'mS', 'mP', 'mA', 'mM']

Еще пример, иллюстрация действия оператора if, присоединенного к вложенному оператору for:

>>> [(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]

[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]

Эквивалент предыдущего примера на базе инструкций:

>>> res = []

>>> for x in range(5):

if x % 2 == 0:

for y in range(5):

if y % 2 == 1:

res.append((x, y))

>>> res

[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]

**Генераторы списков и матрицы.**

Основной способ реализации матриц - вложенные списки. Пример:

>>> M = [[1, 2, 3],

[4, 5, 6],

[7, 8, 9]]

>>>

>>> N = [[2, 2, 2],

[3, 3, 3],

[4, 4, 4]]

Обращение по индексу:

>>> M[1]

[4, 5, 6]

>>> M[1][2]

6

А теперь извлеком второй столбец обходя строки матрицы или обход позиций в строках:

>>> [row[1] for row in M]

[2, 5, 8]

>>> [M[row][1] for row in (0, 1, 2)]

[2, 5, 8]

Также можем извлечь диагональ элементов, range создает список смещений для индексирования строк и столбцов (т.е. M[0][0], M[1][1] и т.д.), если у нас равномерная матрица, конечно:

>>> [M[i][i] for i in range(len(M))]

[1, 5, 9]

И еще генераторы можно использ-ть для объединения матриц. Примеры, результаты умножения соответствующиъ элементов двух матриц и второй - структура вложенных списков с теми же значениями:

>>> [ M[row][col] \* N[row][col] for row in range(3) for col in range(3) ]

[2, 4, 6, 12, 15, 18, 28, 32, 36]

>>> [[ M[row][col] \* N[row][col] for col in range(3)] for row in range(3) ]

[[2, 4, 6], [12, 15, 18], [28, 32, 36]]

В последнем выражении итерации по строкам выполняются во внешнем цикле: для каждой строки запускается итерация по столбцам, которая создает одну строку в матрице с результатами.

Эквивалент послед. выражения:

>>> res = []

>>> for row in range(3):

tmp = []

for col in range(3):

tmp.append(M[row][col] \* N[row][col])

res.append(tmp)

>>> res

[[2, 4, 6], [12, 15, 18], [28, 32, 36]]

Видим, что строка генератора списков гораздо короче и выполняется быстрее, если матрицы большие, НО она и сложнее.

**Понимание генераторов списков.**

При такой гибкости (сложности) генераторов, начинающим кодерам рекомендуется использовать циклы for и ф-цию mаp, а генераторы - в отдельных случаях (если они простые).

Сейчас уже ф-ция map быстрее чем for, а генераторы немного быстрее чем map (зависит от вида задачи).

for делаю логику программы более явной. Но map и генераторы тоже могут быть полезными, к тому же они могут использ-ся там где нет места инструкции for (в теле lambda, в литералах списков и словарей и мн. др.). Т.е. мы должны писать простые ф-ции map и генераторы списков, а в слож. случаях использовать полные инструкции.

**Генераторы списков и map.**

Вспомним, что метод readlines возвращает строки с символом конца строки в конце (\n):

>>> open('tnt.py').readlines() # В книге они пишут название без расширения файла

['aaa\n', 'bbb\n', 'ccc\n'] # а без расширения у меня выдает ошибку

Удалим символы конца строки с помощью map и генераторов:

>>> [line.rstrip() for line in open('tnt.py').readlines()]

['aaa', 'bbb', 'ccc']

>>> [line.rstrip() for line in open('tnt.py')] # Генератор списков

['aaa', 'bbb', 'ccc']

>>> list(map((lambda line: line.rstrip()), open('tnt.py'))) # Функция map

['aaa', 'bbb', 'ccc']

В последних двух случаях используются файловые итераторы (т.е. нам не нужно вызывать метод, который будет читать строки из файла).

Генераторы могут играть роль извлечения столбца. Интерфейс доступа к БД Питона возвращает результат в виде списка кортежей. Список - это таблица, кортежи - строки, а элементы кортежей - это значения столбцов:

>>> [age for (name, age, job) in listbd] # Операция присваивания кортежа

[35, 40] # для извлечения значения в список

>>> list(map((lambda (name, age, job): age), listbd)) # В 3.0 не работает эта схема

[35, 40] # только В 2.6

**Еще раз об итераторах: генераторы.** стр. 567

В Питон есть две языковые конструкции, откладывающие создание результатов когда это возможно, т.е. по требованию, а не всех сразу:

- Функции-генераторы - как обычные инструкции def, но для возврата результатов по одному значению за раз используют инструкцию yield, которая приостанавливает выполнение функции.

- Выражения-генераторы - как генераторы списков, но не конструируют список с результатами, а возвращают объект, который будет воспроизводить рез-ты по требованию.

Обе конструкции реализуют возврат рез-ов по требованию за счет реализации протокола итераций.

**Функции-генераторы: инструкция yield вместо return.**

Можно создавать ф-цию, которая может возвращать значение, а позднее продолжить свою работу с того места, где она была приостановлена. Наз-ся они функции-генераторы, потому что они генерируют последовательность значений с течением времени.

Созд-ся с помощью инструкции def. Автоматически поддерживают протокол итераций и могут использ. в конексте итераций.

**Замораживание состояния.**

Ф-ции-генераторы автоматически приостанавливают и возобновляют свое выполнение, при этом сохраняя информацию, необходимую для генерации значений. Т.о. это альтернатива вычислению всей серии значений заранее, с ручным сохранением и восстановлением состояния в классах.

Эти ф-ции сохраняют информацию о своем состоянии, т.е. всей локальной области видимости, со всеми локал. переменными.

Главное их отличие в том, что они поставляют значение, а не возвращают его - yield приостанавливает работу ф-ции и передает значение вызывающей программе, с сохранением инф-ции о состоянии.

Когда ф-ция-генератор возобновит работу, её выполнение продолжается с первой инструкции, следующей за yield.

**Интеграция с протоколом итераций.**

Ф-ции-генераторы тесно связаны с протоколом итераций. Как мы знаем итерируемые объекты определяют метод \_\_next\_\_, который либо возвращает след. элемент в итерации, либо воздуждает исключение StopIteration по окончании итераций.

Доступ к итератору можно получить с помощью ф-ции iter.

Этот протокл используют все итерационные маханизмы в Питон (for и др.), если протокол не поддерживается for возвращается к операции индексирования послед-ти.

Чтобы поддерживать такой протокол ф-ции с yield компилируются именно как генераторы. При вызове такой ф-ции она возвращает объект генератора, с поддержкой итераций (\_\_next\_\_). Ф-ции-генераторы также могту включать return.

Метод \_\_next\_\_ генератора возобновляет выполнение ф-ции, работа которой продолжается, пока она не встретит следующую инструкцию yield или не возбудит StopIteration.

**Пример функции-генератора.** стр. 569

Пример ф-ции, которая генерирует серии квадратов чисел:

>>> def gensquar(N):

for i in range(N):

yield i\*\*2 # Позднее продолжить работу с этого места

Эта ф-ция поставляет значение и возвращает управление вызывающей программе на каждой итерации цикла - когда она возобновляет работу, восстанавливается её предыдущее состояние и управление передается непосредственно в точку, находящуюся сразу же за инструкцией yield.

Напр. при использовании ф-ции в заголовке цикkа for, управление возвращается ф-ции на каждой итерации в точку, сразу за yield:

>>> for i in gensquar(5):

print(i, end = ':')

0:1:4:9:16:

Если хотим узнать, что происходит внутри цикла for, вызовите ф-цию-генератор напрямую:

>>> x = gensquar(4)

>>> x

<generator object gensquar at 0xb52e257c>

В Питон была создана ф-ция next вызов которой next(X) производит вызлов метода X.\_\_next\_\_ объекта:

>>> next(x)

0

>>> next(x)

1

>>> next(x)

4

>>> next(x)

9

>>> next(x)

...

StopIteration

Как мы знаем, циклы for и др. итераторы работают с генераторами точно так же - вызывают метод \_\_next\_\_ в цикле,пока не получат исключение.

В нашем примере мы могли создать список всех значений:

>>> def bsquar(n):

res = []

for i in range(n): res.append(i\*\*2)

return res

>>> for x in bsquar(5): print(x, end=' : ')

0 : 1 : 4 : 9 : 16 :

>>> bsquar(5) # Пример просто для сравнения, сам добавил

[0, 1, 4, 9, 16]

Вообще можно было использ. и map и генератор списко:

>>> for x in [n\*\*2 for n in range(5)]:

print(x, end=' : ')

0 : 1 : 4 : 9 : 16 :

>>> for x in map((lambda x: x\*\*2), range(5)):

print(x, end=' : ')

0 : 1 : 4 : 9 : 16 :

Однако ф-ции-генераторы лучше используют память и более быстры. Распределяют время по отдельным итерациям, и позволяют избежать всей работы сразу, если вдруг объем большой или вычисления продолжительны.

**Расширенный протокол функций-генераторов: send и next.**

send действует как next, но к тому же обеспечивает, для вызывающей программы, способ взаимодействия с генератором, влияя на его работу.

С технич. точки зрения, yield в наст. время явл-ся не инструкцией, а выражением, которое возвращает элемент, передаваемый методу send.

Когда выражение yield располагается справа от оператора =, оно должно заключаться в круглые скобки (кроме случая когда оно не явл-ся составной частью более крупного выражения). Нарп. правильно будет писать X = yield Y, а также X = (yield Y) + 42.

Расширенный протокол - значения передаются генератору G вызовом метода G.send(value). После этого код генератора возобнавляет работу, и выражение yield возвращает значение, полученное от send.

Когда вызывается метод G.\_\_next\_\_ или next(G), выраежине yield возвращает None.

>>> def gen():

for i in range(10):

x = yield i

print(x)

>>> G = gen()

>>> next(G) # Чтобы запустить генератор, сначала нужно вызвать next()

0

>>> G.send(77) # Переход к след. знач. и передача знач. выражению yield

77

1

>>> G.send(88)

88

2

>>> next(G) # видим, что next() и X.\_\_next\_\_ передают значение None

None

3

Метод send можно использ. чтобы реализовать генератор, который можно будет завершать из вызывающей программы или переустанавливать в нем текущую позицию в результатах.

**Выражения-генраторы: итераторы и генераторы списков.** стр. 573

Понятия итератора и генератора списков были объединены в одну конструкцию - выражения-генераторы.

Они как обычные генераторы только в круглых скобках:

>>> [x\*\*2 for x in range(4)] # Генератор списков: создает список

[0, 1, 4, 9]

>>> (x\*\*2 for x in range(4)) # Выражение-генератор:

<generator object <genexpr> at 0x00F6E300> # создает итерируемый объект

Т.о. генератор списков явл-ся эквивалентом генератора-выраежния, обернутого в list:

>>> list((x\*\*2 for x in range(4)))

[0, 1, 4, 9]

С др. стороны выражения-генераторы возвращают объект-генератор, поддерживающий протокол итераций:

>>> G = (x\*\*2 for x in range(4))

>>> next(G)

0

>>> next(G)

1

>>> next(G)

4

>>> next(G)

9

>>> next(G)

...

StopIteration

Обычно нам не приходится наблюдать итерационную механику действий выражений-генераторов (как в последнем примере), т.к. for делают это автоматически:

>>> for num in (x\*\*2 for x in range(4)):

print('%s, %s' % (num, num/2.0))

0, 0.0

1, 0.5

4, 2.0

9, 4.5

Именно таким способом работает любой итерационный контекст, мы помним.

Круглые скобки можно опустить, если выражение-генератор явл-ся единственным элементом, заключенным в др. скобки, напр. в вызове ф-ции, кроме второго случая с sorted, в примере ниже:

>>> sum(x\*\*2 for x in range(4))

14

>>> sorted(x\*\*2 for x in range(4))

[0, 1, 4, 9]

>>> sorted((x\*\*2 for x in range(4)), reverse=True)

[9, 4, 1, 0]

>>> import math

>>> list(map(math.sqrt, (x\*\*2 for x in range(4))))

[0.0, 1.0, 2.0, 3.0]

Выражения-генераторы оптимизируют использ-е памяти, на практике они работают медленнее, поэтмоу их лучше использ-ть когда объем результатов очень велик.

**Функции-генераторы и выражения-генераторы.**

Одни и те же итерации можно реализовать как ф-цией так и выражением. Например четыре повторения каждого символа в исходной строке:

>>> G = (c \* 4 for c in 'SPAM') # Выражение-генератор

>>> list(G)

['SSSS', 'PPPP', 'AAAA', 'MMMM']

>>> def tfour(S): # Функция-генератор

for c in S:

yield c\*4

>>> G = tfour('spam')

>>> list(G)

['ssss', 'pppp', 'aaaa', 'mmmm']

- ф-ция-генератор, конечно длиннее, но может вместить больше логики.

Оба вида генераторов могут выполнять автоматическое и ручное управление итерациями. В предыдущ. примерах list дает автоматич. выполнение итераций.

Рассмотрим выполнение вручную:

>>> G = (c\*4 for c in 'SPAM')

>>> I = iter(G)

>>> next(I)

'SSSS'

>>> next(I)

'PPPP'

>>> G = tfour('spam')

>>> I = iter(G)

>>> next(I)

'ssss'

>>> next(I)

'pppp'

**Генераторы - это объекты итераторов однократного применения.**

И ф-ции-генераторы и выражения итераторы имеют собственные итераторы и поэтому поддерживают лишь однократное выполнение итераций. Напр. для выражения-ген из пред. примера итератором явл-ся сам генератор:

>>> G = (c \* 4 for c in 'SPAM')

>>> iter(G) is G # Итератором генератора явл-ся сам генератор:

True # G имеет метод \_\_next\_\_

>>> G = (c \* 4 for c in 'SPAM')

>>> I1 = iter(G) # Выполняем итерации вручную

>>> next(I1)

'SSSS'

>>> next(I1)

'PPPP'

>>> I2 = iter(G) # Второй итератор ссылается на ту же позицию!

>>> next(I2)

'AAAA'

Чтобы выполнить повторный обход нужно создавать новый генератор:

>>> list(I1) # остатки результатов в I1

['MMMM']

>>> next(I2) # Другие итераторы тоже исчерпаны

...

StopIteration

>>> I3 = iter(G) # исчерпан и вновь созданные итераторы

>>> next(I3)

...

StopIteration

>>> I3 = iter(c \* 4 for c in 'SPAM') # Чтобы выполнить повторный обход рез-ов

>>> next(I3) # создаем новый генератор

'SSSS'

То же самое и с функциями:

>>> def tfour(S):

for c in S:

yield c\*4

>>> G = tfour('spam')

>>> iter(G) is G

True

>>> I1, I2 = iter(G), iter(G)

>>> next(I1)

'ssss'

>>> next(I1)

'pppp'

>>> next(I2)

'aaaa'

Некоторые встроенные типы имеют отличное поведение, поддерживая создание множества независимых итераторов:

>>> L = [1, 2, 3, 4]

>>> I1, I2 = iter(L), iter(L)

>>> next(I1)

1

>>> next(I1)

2

>>> next(I2) # Списки поддерживают множество независимых итераторов

1

>>> del L[2:] # Изменения отражаются на всех итераторах

>>> next(I1)

...

StopIteration

**Имитация функций zip и map с помощью инструментов итераций.**

Как мы знаем map отображает заданную ф-цию на элементы послед-ти, так же как zip объединяет их:

>>> s1 = 'abc'

>>> s2 = 'xyz123'

>>> list(zip(s1, s2))

[('a', 'x'), ('b', 'y'), ('c', 'z')]

- zip объединяет элементы, усекая результирующую последовательность, по длине кратчайшей исходной последовательности.

>>> list(zip([-2, -1, 0, 1, 2])) # Единственная послед-ть:

[(-2,), (-1,), (0,), (1,), (2,)] # одномерные кортежи

>>> list(zip([1, 2, 3], [2, 3, 4, 5])) # N последовательностей:

[(1, 2), (2, 3), (3, 4)] # N-мерные кортежи

map передает объединенные элементы последовательностей указанной ф-ции, усекая результирующую послед-ть по длине кратчайшей исходной послед-ти:

>>> list(map(abs, [-2, -1, 0, 1, 2])) # Единственная послед-ть:

[2, 1, 0, 1, 2] # одномерная ф-ция

>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5])) # N последовательностей:

[1, 8, 81] # N-мерная ф-ция

**Создание собственной версии функции map(func, ...).**

Пример нашей версии map, запишем её в tnt.py:

def mymap(func, \*seq):

res = []

for args in zip(\*seq):

res.append(func(\*args))

return res

print(mymap(abs, [-2, -1, 0, 1, 2]))

print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))

Ф-ция получает множество аргументов-послед-тей (итерируемых объектов), распаковывает их в аргументы ф-ции zip, для последующего объединения, и затем распаковывает рез-ты вызова zip в аргументы указанной ф-ции func.

В работе этой ф-ции мы используем тот факт, что объединение элементов последовательностей, по сути, явл-ся промежуточной операцией при отображении.

Проверка показывает результат:

>>> import tnt

[2, 1, 0, 1, 2]

[1, 8, 81]

На самом деле наша ф-ция похожа на шаблон генератора списков, т.е. мы можем её переписать:

def mymap(func, \*seq):

return [func(\*args) for args in zip(\*seq)] # Используем генератор списков

print(mymap(abs, [-2, -1, 0, 1, 2]))

print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))

>>> import tnt

[2, 1, 0, 1, 2]

[1, 8, 81]

Используя наши знания мы можем создать ф-цию-генератор или выражение-генератор, для выдачи рез-ов по требованию (напр. если вдруг объем данных очень большой):

def mymap1(func, \*seq):

res = []

for args in zip(\*seq):

yield func(\*args)

def mymap2(func, \*seq):

return (func(\*args) for args in zip(\*seq))

print(list(mymap1(abs, [-2, -1, 0, 1, 2])))

print(list(mymap1(pow, [1, 2, 3], [2, 3, 4, 5])))

print(list(mymap2(abs, [-2, -1, 0, 1, 2])))

print(list(mymap2(pow, [1, 2, 3], [2, 3, 4, 5])))

>>> import tnt

[2, 1, 0, 1, 2]

[1, 8, 81]

[2, 1, 0, 1, 2]

[1, 8, 81]

Обернув выводы в list мы получаем сразу список результатов, но никакой работы не выполняется пока мы не запустим list. Т.е. генераторы в этом примере выдают результаты только по требованию.

**Создание собственных версий функций zip(...) и map(None, ...).**

Сейчас мы будем имитировать ф-ции zip и map в Питон 2.6, в этой версии языка ф-ции могли принимать и обрабатывать все элементы неравномерных последовательностей:

def myzip(\*seq):

seq = [list(s) for s in seq]

res = []

while all(seq):

res.append(tuple(s.pop(0) for s in seq))

return res

def mymap(\*seq, pad=None):

seq = [list(s) for s in seq]

res = []

while any(seq):

res.append(tuple((s.pop(0) if s else pad) for s in seq))

return res

s1, s2 = 'abc', 'xyz123'

print(myzip(s1, s2))

print(mymap(s1, s2))

print(mymap(s1, s2, pad=99))

Обе эти ф-ции работают с любыми итерируемыми объектами. Ф-ции all и any возвращают True если все или хотя бы один элемент объекта имеет истинное значение, они позволяют остановить итерации, когда один или все аргументы итераций превращаются в пустые списки. Также обратите на именованный аргумент pad:

А вот рез-т работы нашего сценария:

>>> import tnt

[('a', 'x'), ('b', 'y'), ('c', 'z')]

[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]

[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]

Сейчас наши ф-ции возвращают весь список результатов сразу и их также можно переделать в генераторы, поставляющие по одному элементу за раз, и мы снова используем ф-цию list - для обертки результатов:

def myzip(\*seq):

seq = [list(s) for s in seq]

while all(seq):

yield tuple(s.pop(0) for s in seq)

def mymap(\*seq, pad=None):

seq = [list(s) for s in seq]

while any(seq):

yield tuple((s.pop(0) if s else pad) for s in seq)

s1, s2 = 'abc', 'xyz123'

print(list(myzip(s1, s2)))

print(list(mymap(s1, s2)))

print(list(mymap(s1, s2, pad=99)))

>>> import tnt

[('a', 'x'), ('b', 'y'), ('c', 'z')]

[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]

[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]

И наконец ниже мы приведем версии zip и map, которые не удаляют элементы с помощью метода pop, а выполняют работу за счет нахождения аргументов с минимальной и максимальной длиной. Этим можно реализовать вложенные генераторы списков для обхода диапазонов индексов в аргументах:

def myzip(\*seq):

minlen = min(len(s) for s in seq)

return [tuple(s[i] for s in seq) for i in range(minlen)]

def mymap(\*seq, pad=None):

maxlen = max(len(s) for s in seq)

index = range(maxlen)

return [tuple((s[i] if len(s)>i else pad) for s in seq) for i in index]

s1, s2 = 'abc', 'xyz123'

print(myzip(s1, s2))

print(mymap(s1, s2))

print(mymap(s1, s2, pad=99))

>>> import tnt

[('a', 'x'), ('b', 'y'), ('c', 'z')]

[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]

[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]

Т.к. в этих версиях есть ф-ция len и индексирование, то они уже не предполагают что аргументы могут быть произвольными итерирумыми объектами. Внеш. генератор списков обходит диапазон индексов в аргументе, а внутренний (передается ф-ции tuple) обходит полученные ф-цией послед-ти, извлекая из каждой по одному элементу.

Аргументы min и max явл-ся выражениями-генераторами, которые заканчивают итерации до того, как влож. генератор списков присутпит к итерациям.

На самом деле, пока поток управления не достигнет квадрат. скобок генератора списков, не воспроизв-ся никаких результатов - в этом месте идет запуск генераторов списка и выражений-генераторов.

Чтобы возвращались не списки, а генераторы нужно вместо квадрат. скобок использовать круглые:

def myzip(\*seq):

minlen = min(len(s) for s in seq)

return (tuple(s[i] for s in seq) for i in range(minlen))

s1, s2 = 'abc', 'xyz123'

print(list(myzip(s1, s2)))

>>> import tnt

[('a', 'x'), ('b', 'y'), ('c', 'z')]

**Однократные итерации.**

Помним, что некоторые ф-ции (напр. map) возвращают итераторы, и имеют только однократный обход элементов, опустошаясь по достижении конца итераций. Рассмотрим теперь их значений на практике.

Рассмотрим альтернативу ф-ции zip:

>>> def myzip(\*args):

iters = map(iter, args)

while iters:

res = [next(i) for i in iters]

yield tuple(res)

Т.к. здесь мы сипользуем iter и next, наша ф-ция принимает любые итерируемые объекты.

В 2.6 эта ф-ция прекрасно работает, но в 3.0 впадает в бесконечный цикл, т.к. ф-ция map в 3.0 возвращает объект-итератор однократного использования, а не список как в 2.6.

В 3.0, как только внутри цикла будет выполнен генератор списков, переменная iters навсегда останется пустой (и res будет ссылаться) на пустой список []).

Чтобы ф-ция работала в 3.0 нужн ос помощью list создать объект, который поддерживает многократные итерации:

>>> def myzip(\*args):

iters = list(map(iter, args)) # Всего лишь одно изменение

while iters:

res = [next(i) for i in iters]

yield tuple(res)

>>> list(myzip('abc', 'limpop'))

[('a', 'l'), ('b', 'i'), ('c', 'm')]

Главное запомним! - Обертывать вызовы ф-ции map в list в версии 3.0 приходится не только для отображения списков результатов в IDLE!

**Генерирование значений во встроенных типах и классах.**

Словари, как мы помним, имеют свои итераторы, которые воспроизводят ключи:

>>> D = {'a':1, 'b':2, 'c':3 }

>>> x = iter(D)

>>> next(x)

'b'

>>> next(x)

'c'

Ключи можно обойти вручную или с помощью цикла for, ф-ции map, генераторов списков и др.:

>>> for key in D:

print(key, D[key])

b 2

c 3

a 1

Знаем также, что интерпретатор протсо загружает строки из файла по мере необходимости:

>>> for line in open('data.txt'):

print(line, end='')

Hello

World

Итераторы встроенных типов используют те же концепции, что и наши выражения- и функции-генераторы, только работают со своим типом данных.

Есть возможность реализовать объекты-генераторы с помощью классов. Но это тема дальнейших глав.

**Краткая сводка по синтаксису генераторов в 3.0.** стр. 583

Напомним, что у нас еще есть генераторы множеств и словарей:

- С множествами литеральная форма {1, 3, 2} эквивалентна вызову set([1, 3, 2]), а новый иснтаксис генераторов множеств {f(x) for x in S if P(x)} напоминает синтаксис выражений-генераторов set(f(x) for x in S if P(x)), где f(x) - произвольное выражение.

- Новая конструкция генераторов словарей {key: val for (key, val) in zip(keys, vals)} действует так же, как dict(zip(keys, vals)), and {x: f(x) for x in items} и напоминает выражение-генератор

dict((x, f(x)) for x in items).

>>> [x \* x for x in range(10)] # Генератор списков: работает подобно list

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>>

>>> (x \* x for x in range(10)) # Выражение-генератор: воспроизводи тэлементы

<generator object <genexpr> at 0x022812D8>

>>>

>>> {x \* x for x in range(10)} # генератор множеств. {x, y} - литерал множеств в 3.0

{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

>>>

>>> {x: x \* x for x in range(10)} # генератор словарей

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

- последние два примера доступны только в 3.0 и недоступны в 2.6.

**Понимание генераторов множеств и словарей.**

Генераторы множеств и словарей это особые выражения-генераторы, которые передаются конструкторам этих типов данных. И они принимают любые итерируемые объекты:

>>> {x \* x for x in range(10)} # Генератор

{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

>>> set(x \* x for x in range(10)) # Генератор и конструктор типа

{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

>>> {x: x \* x for x in range(10)}

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

>>> dict((x, x \* x) for x in range(10))

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

Мы всегда можем конструировать объекты результата вручную, напр. возьмем два последних генератора:

>>> res = set()

>>> for x in range(10): # Эквивалент генератора множеств

res.add(x \* x)

>>> res

{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

>>> res = {}

>>> for x in range(10): # Эквивалент генератора словарей

res[x] = x \* x

>>> res

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

И хотя обе формы принимают итераторы, они не воспроизводят рез-ты по требованию - выдают рез-т целиком. Если нужно ключи и знач. по запросу используйте выражение-генератор:

>>> G = ((x, x\*x) for x in range(10))

>>> next(G)

(0, 0)

>>> next(G)

(1, 1)

**Расширенный синтаксис генераторов множеств и словарей.**

Генераторы множеств и словарей поддерживают инструкции if, прим. воспроизведем квадраты четных чисел в заданном диапазоне (фильтрация будет):

>>> [x \* x for x in range(10) if x % 2 == 0] # Списки упорядочены

[0, 4, 16, 36, 64]

>>> {x \* x for x in range(10) if x % 2 == 0} # А множества - нет

{0, 16, 64, 4, 36}

>>> {x: x \* x for x in range(10) if x % 2 == 0} # Как и ключи словаря

{0: 0, 8: 64, 2: 4, 4: 16, 6: 36}

Вложенные for также явл-ся допустимыми, хотя неупорядоченная природа обоих типов может осложнить интерпретацию рез-ов:

>>> [x + y for x in [1, 2, 3] for y in [4, 5, 6]] # Списки сохраняют дубликаты

[5, 6, 7, 6, 7, 8, 7, 8, 9]

>>>

>>> {x + y for x in [1, 2, 3] for y in [4, 5, 6]} # А множества - нет

{8, 9, 5, 6, 7}

>>>

>>> {x: y for x in [1, 2, 3] for y in [4, 5, 6]} # Как и ключи словарей

{1: 6, 2: 6, 3: 6}

Также генераторы словарей и множеств способны выполнять итерации по итераторам любых типов, поддерживающих протокол итераций:

>>> {x + y for x in 'ab' for y in 'cd'}

{'bd', 'ad', 'bc', 'ac'}

>>> {x + y: (ord(x), ord(y)) for x in 'ab' for y in 'cd'}

{'bd': (98, 100), 'ad': (97, 100), 'bc': (98, 99), 'ac': (97, 99)}

>>> {k \* 2 for k in ['spam', 'ham', 'sausage'] if k[0] == 's'}

{'sausagesausage', 'spamspam'}

>>> {k.upper(): k \* 2 for k in ['spam', 'ham', 'sausage'] if k[0] == 's'}

{'SPAM': 'spamspam', 'SAUSAGE': 'sausagesausage'}

**Хронометраж итерационных альтернатив.** стр. 586

В этом разделе мы подведем некоторые итоги изучению итераторов и ф-ций. Мы знаем, например, что генераторы списков более быстры чем циклы for, а скорость ф-ции map выше или ниже в зависимости от задачи.

Выражения-генераторы, обычно немного медленнее, чем генераторы списков, но при этом минимизируют объем требуемой памяти.

Все это постоянно оптимизируется и поэтому вычислять нужно непосредственно на практике.

**Модуль time.**

Рассмотрим утилиту mytimer, которую сохраним в файле, чтобы использ. для изучения производительности различных итераций:

# файл mytimer.py

import time

reps = 1000

repslist = range(reps)

def timer(func, \*pargs, \*\*kargs):

start = time.clock()

for i in repslist:

ret = func(\*pargs, \*\*kargs)

elapsed = time.clock() - start

return (elapsed, ret)

Эта утилита может вызвать любую ф-цию с любым кол-вом позиционных и именованных аргументов. Она фиксирует начальное время, вызывает ф-цию фмиксированное число раз и вычитает время начала из времени конца.

Отметим следующее:

- Модуль time и зстандарт. библиотеки, позволяет получить текущее время, с точностью зависящей от платформы. В Windows, должен уметь с точностю до микросекунд, т.е. достаточно точно.

- Вызов ф-ции range вынесен за пределы цикла измерения времени, поэтому время конструирования диапазона не накладывается на получаемые рез-ты в Питон 2.6. В 3.0 ф-ция range возвращает итератор, поэтому в 3.0 данный шаг можно считать излишним (хотя он и не мешает) - шаг, я так понимаю, это вынесение ф-ции range за фикл, т.е. в 3.0 можно было бы и не выносить (это моя личная догадка).

- Счетчик reps оформлен как глобальная переменная, поэтому может изменяться импортирующим модулем, если нужно: mytimer.reps = N

По окончании измерений возвращается время выполнения всех вызовов внутри кортежа и последнее значение, полученное от исследуемой ф-ции (чтобы провреить работу самой ф-ции).

**Сценарий хронометража.**

Запустим сценарий, использующий наш mytimer, и который измерит скорость выполнения различных приемов конструирвоания списков:

# файл timeseq.py

import sys, mytimer # Импортирует функцию timer

reps = 10000

repslist = range(reps) # Вызов range вынесен за пределы цикла в 2.6

def forLoop():

res = []

for x in repslist:

res.append(abs(x))

return res

def listComp():

return [abs(x) for x in repslist]

def mapCall():

return list(map(abs, repslist)) # вызов list нужен только в 3.0

def genExp():

return list(abs(x) for x in repslist) # list возвращает сразу все рез-ты

def genFunc():

def gen():

for x in repslist:

yield abs(x)

return list(gen())

print(sys.version)

for test in (forLoop, listComp, mapCall, genExp, genFunc):

elapsed, result = mytimer.timer(test)

print('-'\*33)

print('%-9s: %.5f => [%s...%s]' % (test.\_\_name\_\_, elapsed, result[0], result[-1]))

Этот сценарий тестирует пять способов создания списков и выполняет по 10 миллионов итераций каждым способом, т.е. каждый тест создает список из 10000 элементов 1000 раз.

Заметим, что некоторые вызовы мы обернули в list (для 3.0), и заметим атрибут \_\_name\_\_ - он возвращает имя ф-ции.

**Результаты хронометража.**

Запустим из командной строки и проверим:

C:\Python34>python timeseq.py

3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600 32 bit (Intel)]

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

forLoop : 2.64343 => [0...9999] # цикл for

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

listComp : 1.52439 => [0...9999] # генератор списков

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

mapCall : 1.21337 => [0...9999] # ф-ция map

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

genExp : 1.96060 => [0...9999] # выражение-генератор

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

genFunc : 1.93480 => [0...9999] # функция-генератор

Время у меня разное с автором, но в целом рез-ты одинаковы: map оказалась быстрее чем генератор списков; оба они оказались существенно быстрее инструкции цикла for, а ф-ция-генератор и выраждение-генратор заняли промежуточное положение.

Заметим, что выражения-генераторы медленее генераторов списков. Т.е. внутренняя реализация их различается (хотя конструктивно они похожи).

А теперь изменим методы нашего сценария так, чтобы выполнялась настоящая операция (напр. сложение), а не abs:

# файл timeseq.py

import sys, mytimer

reps = 10000

repslist = range(reps)

def forLoop():

res = []

for x in repslist:

res.append(x+10)

return res

def listComp():

return [x+10 for x in repslist]

def mapCall():

return list(map((lambda x: x+10), repslist))

def genExp():

return list(x+10 for x in repslist)

def genFunc():

def gen():

for x in repslist:

yield x+10

return list(gen())

print(sys.version)

for test in (forLoop, listComp, mapCall, genExp, genFunc):

elapsed, result = mytimer.timer(test)

print('-'\*33)

print('%-9s: %.5f => [%s...%s]' % (test.\_\_name\_\_, elapsed, result[0], result[-1]))

Тестируем измененный сценарий в 3.0:

C:\Python34>python timeseq.py

3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600 32 bit (Intel)]

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

forLoop : 2.59981 => [10...10009]

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

listComp : 1.42804 => [10...10009]

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

mapCall : 2.83584 => [10...10009]

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

genExp : 1.80817 => [10...10009]

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

genFunc : 1.79342 => [10...10009]

Мы видим как изменилсь рез-ты: вызов map стал более медленным, чем цикл for.

На самом деле невозможно заранее утверждать какой метод лучше, можно лишь провести хронометраж своего кода.

И даже в таком случае единственное, что мы знаем - это то ,что использование пользовательской ф-ции в вызове map, приводит к снижению скорости её выполнения в 2 раза.

И все-таки основное внимание кодеров Питона должно относиться не к производительности (не только к ней), а к удобочитаемости и простоте кода! А потом уже можно заниматься его оптимизацией, если нужно.

**Альтернативные реализации модуля хронометража.** стр. 590

Наш модуль хронометража имеет ряд не лучших особенностей:

- У нас всегда использ-ся ф-ция time.clock. Для Виндовс она лучше, а в Unix лучшая точность у ф-ции time.time (Запомнить!).

- Для изменения кол-ва повторений нужно менять глобал. переменную модуля - это не идеальное решение, если ф-ция timer импортируется и одновременно использ. в неск-их модулях.

- Ф-ция timer выполняет тестовую ф-цию большое число раз. Чтобы учесть случайности, вызванные разными уровнями нагрузки на систему, можно было бы отбирать наилучшие рез-ты из серии тестов вместо того, чтобы рассчитывать общее время выполнения.

Ниже приводится новая реализация модуля mytimer: выбор ф-ции определения времени, в зависимости от платформы; счетчик повторений передается ф-ции timer в виде именованного аргумента \_reps и дополнительно предоставл-ся ф-ция хронометража, возвращающая лучший рез-т из серии по N испытаний:

# файл mytimer.py

"""

timer(spam, 1, 2, a=3, b=4, \_reps=1000) вызывает и измеряет время работы ф-ции spam(1, 2, a=3)

\_reps раз? и возвращает общее время, затраченное на все вызовы, с результатом вызова испытуемой функции;

best(spam, 1, 2, a=3, b=4, \_reps=50) многократно вызывает ф-цию timer, чтобы исключить влияние флуктуаций в нагрузке на систему, и возвращает лучший результат из серии по \_reps испытаниям

"""

import time, sys

if sys.platform[:3] == 'win':

timefunc = time.clock

else:

timefunc = time.time

def trace(\*args): pass

def timer(func, \*pargs, \*\*kargs):

\_reps = kargs.pop('\_reps', 1000)

trace(func, pargs, kargs, \_reps)

repslist = range(\_reps)

start = timefunc()

for i in repslist:

ret = func(\*pargs, \*\*kargs)

elapsed = timefunc() - start

return (elapsed, ret)

def best(func, \*pargs, \*\*kargs):

\_reps = kargs.pop('\_reps', 50)

best = 2\*\*32

for i in range(\_reps):

(time, ret) = timer(func, \*pargs, \_reps=1, \*\*kargs)

if time < best: best = time

return (best, ret)

Строка документирования описывает порядок его исполь-ия. Метод pop удаляет аргумент \_reps из списка аргументов, предназначенных для испытуемой ф-ции, чтобы подставить значение по умолчанию если нужно.

Крмое этого модуль позволяет выводить значения аргументво на этапе отладки, для этого нужно изменить вызов trace на print.

Также изменим немного сценарий тестового модуля:

# файл timeseq.py

import sys, mytimer

reps = 10000

repslist = range(reps)

def forLoop():

res = []

for x in repslist:

res.append(x+10)

return res

def listComp():

return [x+10 for x in repslist]

def mapCall():

return list(map((lambda x: x+10), repslist))

def genExp():

return list(x+10 for x in repslist)

def genFunc():

def gen():

for x in repslist:

yield x+10

return list(gen())

print(sys.version)

for tester in (mytimer.timer, mytimer.best):

print('<%s>' % tester.\_\_name\_\_)

for test in (forLoop, listComp, mapCall, genExp, genFunc):

elapsed, result = tester(test)

print('-'\*33)

print('%-9s: %.5f => [%s...%s]' % (test.\_\_name\_\_, elapsed, result[0], result[-1]))

И запускаем тест:

C:\Python34>python timeseq.py

3.4.4 (v3.4.4:737efcadf5a6, Dec 20 201

<timer>

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

forLoop : 2.60389 => [10...10009]

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

listComp : 1.38774 => [10...10009]

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

mapCall : 2.85636 => [10...10009]

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

genExp : 1.80662 => [10...10009]

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

genFunc : 1.85520 => [10...10009]

<best>

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

forLoop : 0.00243 => [10...10009]

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

listComp : 0.00122 => [10...10009]

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

mapCall : 0.00267 => [10...10009]

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

genExp : 0.00166 => [10...10009]

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

genFunc : 0.00165 => [10...10009]

- Значения best могут возрасти при выполнении итераций по большим массивам данных.

В целом видим, что генераторы списков показывают лучшую производительность - ф-ция map оказывается быстрее лишь при использовании встроенных ф-ций (напр. наша первая версия с встроенной ф-цией abs).

**Использование аргументов, которые могут передаваться только по именам.**

Эти аргументы мы можем использовать в нашем модуле mytimer. Они идеально подходят для передачи параметров, таких как аргумент \_reps, например.

Приведем альтернативную версию нашего модуля (именованные аргументы только в Питон 3.0):

# файл mytimer.py

"""

Вместо формы \*\* и метода pop словарей используются аргументы, которые могут передаваться только по именам, появившиеся в 3.0

В 3.0 нет необходимости выносить вызов range() за пределы цикла, так как эта функция возвращает генератор, а не список

"""

import time, sys

trace = lambda \*args: None # or print

timefunc = time.clock if sys.platform == 'win32' else time.time

def timer(func, \*pargs, \_reps=1000, \*\*kargs):

trace(func, pargs, kargs, \_reps)

start = timefunc()

for i in range(\_reps):

ret = func(\*pargs, \*\*kargs)

elapsed = timefunc() - start

return (elapsed, ret)

def best(func, \*pargs, \_reps=50, \*\*kargs):

best = 2\*\*32

for i in range(\_reps):

(time, ret) = timer(func, \*pargs, \_reps=1, \*\*kargs)

if time < best: best = time

return (best, ret)

C:\Python34>python timeseq.py

3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600 32 bit (Intel)]

<timer>

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

forLoop : 2.60013 => [10...10009]

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

listComp : 1.40457 => [10...10009]

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

mapCall : 2.86075 => [10...10009]

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

genExp : 1.84090 => [10...10009]

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

genFunc : 1.81671 => [10...10009]

<best>

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

forLoop : 0.00243 => [10...10009]

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

listComp : 0.00122 => [10...10009]

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

mapCall : 0.00265 => [10...10009]

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

genExp : 0.00165 => [10...10009]

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

genFunc : 0.00165 => [10...10009]

А теперь опробуем наш модуль в IDLE:

>>> from mytimer import timer, best

>>>

>>> def power(x, y): return x \*\* y # Испытуемая ф-ция

>>> timer(power, 2, 32) # Общее время, последний рез-т

(0.0007838985122410809, 4294967296)

>>> timer(power, 2, 32, \_reps=1000000) # Переопределить кол-во повтором

(0.7994840126328917, 4294967296)

>>> timer(power, 2, 100000)[0] # 2 \*\* 100000: общее время для 1000 повторов

1.1828567597278408

>>> best(power, 2, 32) # Лучшее время, последний рез-т

(2.234920913224414e-06, 4294967296)

>>> best(power, 2, 100000)[0] # 2 \*\* 100000: лучшее время

0.0011783620543894813

>>> best(power, 2, 100000, \_reps=500)[0] # Переопределить кол-во повторов

0.0011775239590434694

Результаты хронометража могут оказаться более осмысленными при тестировании длительных операций, например - операция 2 \*\* 1000000 выполняется на порядок (в 10 раз) дольше чем предыдущая 2 \*\* 100000:

>>> timer(power, 2, 1000000, \_reps=1)[0] # 2 \*\* 1000000: общее время

0.017655316527680043

>>> timer(power, 2, 1000000, \_reps=10)[0]

0.1749920730148915

>>> best(power, 2, 1000000, \_reps=1)[0] # 2 \* 1000000: лучшее время

0.017450821263594207

>>> best(power, 2, 1000000, \_reps=10)[0] # Иногда 10 ничуть не хуже 50

0.017288789497001744

>>> best(power, 2, 1000000, \_reps=50)[0] # Лучшее разрешение

0.017180116467329754

**Другие предложения.** стр. 595

Изучить и испытать выкладки в этом разделе самостоятельно.

**Типичные ошибки при работе с функциями.** стр. 596

**Локальные имена определяются статически.**

Как мы знаем, имена, которым выполняется присваивание внутри ф-ции, рассматриваются как локальные - они располагаются в области видимости ф-ции и существуют только во время работы ф-ции.

Но мы пока не знали, что локал. переменные определяются статически, во время компиляции кода в инструкции def, а не в соответствии с операциями присваивания, во время выполнения.

Обычно, если внутри ф-ции имени не присвоено значение, его поиск будет идти в объемлющем модуле:

>>> x = 99

>>> def selector(): # внутри ф-ции переменной х не присв-ся никакого значения

print(x) # эта переменная х будет найдена в глобал. области

>>> selector()

99

А теперь немного изменим нашу ф-цию:

>>> def selector():

print(x) # Переменная х еще не существует!

x = 88 # х - классифицируется как локал. переменная

>>> selector()

...

UnboundLocalError: local variable 'x' referenced before assignment

- Получим сообщение о том, что переменная не определена. Причем мы его получим если даже, как в прошлом примере, присвоим в модуле значение переменной х, а потом определим нашу ф-цию и попытаемся выполнить её.

Этот код компилируется интерпретатором во время ввода в интерактивной оболочке или во время импорта модуля. Во время компиляции Питно обнаруживает операцию присваивания Х и делает вывод, что Х - это локальное имя везде в теле ф-ции.

Но во время выполнения ф-ции, из-за того, что к моменту вызова инструкции print операция присваивания еще не проводилась, интерпретатор сообщает о том, что имя не определено.

Т.е. мы сталкиваемся с тем, чтообращение к переменной Х произошло до того как ей было присвоено значение.

Если же мы хотим вывести значение глоабл. переменной Х, нужно объвить её глоабльной с помощью global:

>>> def selector():

global x

print(x)

x = 88

>>> selector()

99

- При этом наша ф-ция изменяет глобальную переменную х, еще раз вызовем ф-цию:

>>> selector()

88

>>> x

88

- Или сделаем так как в книге:

>>> x = 99

>>> def selector():

import \_\_main\_\_ # Импортировать вмещающий модуль

print(\_\_main\_\_.x) # Квалифицированное обращение к глобальной версии имени

x = 88 # Неквалифицированное локальное имя Х

print(x) # Вывести локальную версию имени

>>> selector()

99

88

Обращение по квалифицированному имени приводит к извлечению значения из пространства имен объекта (у нас это модуль с именем \_\_main\_\_).

**Значения по умолчанию и изменяемые объекты.**

Знач. по умолч. для аргументов вычилсяются и запоминаются в момент выполнения инструкции def, а не в момент вызова ф-ции.

Т.к. знач. по умолч. сохраняются м/у вызовами ф-ции следует быть внимательным с изменяемыми значениями, напр. со списками. Пусть наша ф-ция использует пустой список в качестве знач. по умолч., а при каждом вызове его изменяет:

>>> def saver(x=[]): # Объект списка сохраняется!

x.append(1) # При каждом вызове изменяется один и тот же объект

print(x)

>>> saver([2]) # Значение по умолч. не используется

[2, 1]

>>> saver() # Значение по умолч. используется

[1]

>>> saver() # Список растет при каждом вызове!

[1, 1]

>>> saver()

[1, 1, 1]

Если же такое поведение не нужно, можно просто создавать копию аргуменат по умолч. в начале тела ф-ции или переместить выражение, возвращающее знач. по умолч., в тело ф-ции.

>>> def saver(x=None):

if x is None: # Аргумент отсутствует?

x = [] # Создать новый список

x.append(1) # Изменить объект списка

print(x)

>>> saver([2])

[2, 1]

>>> saver() # Список больше не растет

[1]

>>> saver()

[1]

И рассмотрим еще один способ определения нашей ф-ции:

>>> def saver():

saver.x.append(1)

print(saver.x)

>>> saver.x = []

>>> saver()

[1]

>>> saver()

[1, 1]

>>> saver()

[1, 1, 1]

Имя ф-ции явл-ся глобальным для самой ф-ции, но объявлять его глоабльным внутри ф-ции не требуется. Такой прием использования атрибутов, присоединенных к объекту ф-ции, оказывается более явным.

**Функции, не возвращающие результат.**

Ф-ции могут не иметь инструкцию return (или yield). Они не возвращают управление явно, выход из неё происходит, когда поток управления достигает конца тела ф-ции. С технич. точки зрения все ф-ции возвращают значение - если нет return ф-ция возвращает объект None:

>>> def proc(x):

print(x)

>>> x = proc('test 123...')

test 123...

>>> print(x)

None

Такие ф-ции, не имеющие return, явл-ся эквивалентами того, что в других ЯП называется "процедурами". Они вызываются как инструкции, а возвращаемое None игнорируется, т.к. они выполняют работу не вычисляя рез-т.

Помните об этом, т.к. интерпретатор не выдаст ошибку если мы попытаемся присвоить рез-т такой ф-ции чему либо.

Пример, присваивание рез-та метода списков append, получим None, а не список:

>>> list = [1, 2, 3, 4]

>>> list

[1, 2, 3, 4]

>>> list = list.append(4)

>>> print(list)

None

>>> list # Ничего не выводится, списка больше нет

>>>

И еще один момент, будьте внимательны при использовании переменных в области видимости объемлющей ф-ции, которые изменяются объемлющим циклом - все ссылки на эту переменную будут запоминать значение, которое будет иметь переменная в последней итерации цикла (см. гла. 17).

Чтобы сохранить значения переменной цикла в каждой итерации, используйте аргументы со значениями по умолчанию.

**Упражнения к 4 части.** стр. 602