**Изучаем Python**

**Марк Лутц 2011, издание O’ Reilly**

**Том 1**

***Часть 1. ВВедение***

Т.к. в учебнике по Питону сразу написано о прикладном программировании, то сразу же поясним этот момент, что такое и с чем его едят.

# Прикладное программирование

Прикладная программа или приложение — программа, предназначенная для выполнения определённых задач и рассчитанная на непосредственное взаимодействие с пользователем. В большинстве операционных систем прикладные программы не могут обращаться к ресурсам компьютера напрямую, а взаимодействуют с оборудованием и другими программами посредством операционной системы. Также на простом языке — вспомогательные программы.

К прикладному программному обеспечению относятся компьютерные программы, написанные для пользователей или самими пользователями для задания компьютеру конкретной работы. Программы обработки заказов или создания списков рассылки — пример прикладного программного обеспечения. Программистов, которые пишут прикладное программное обеспечение, называют прикладными программистами.

Подразделения на группы прикладных программ:

программные средства общего назначения

Текстовые редакторы

Текстовые процессоры

Системы компьютерной вёрстки

Графические редакторы

СУБД

Электронные таблицы

Веб-браузеры

программные средства развлекательного назначения

Медиаплееры

Компьютерные игры

программные средства специального назначения

Экспертные системы

Трансляторы

Мультимедиа-приложения (медиаплееры, программы для создания и редактирования видео, звука, text-to-speech и пр.)

Гипертекстовые системы (электронные словари, энциклопедии, справочные системы)

Системы управления содержимым

профессиональные программные средства

САПР

АРМ

АСУ

АСУ ТП

АСНИ

Геоинформационные системы

Биллинговые системы

CRM

CTRM/ETRM — системы управления складом

SRM (Supplier Relationship Management) — системы управления взаимоотношениями с поставщиками

BI (Business Intelligence) — аналитические системы

DMS (Document Management System) — Система управления документами/Система автоматизации документооборота (системы электронного документооборота)

CMS (Content Management System) — системы управления содержимым (контентом)

WMS (Warehouse Management System) — системы управления складом (СУС)

ERP-системы — системы планирования ресурсов предприятия

EAM-системы — системы управления основными фондами предприятия

MRM-системы — системы управления маркетинговыми ресурсами

MES-системы — системы оперативного (цехового) управления производством и ремонтами

Интеграционные шины данных (ESB)

По этому списку становится более ясно и понятно, что такое прикладное программирование и прикладные программы - приложения (не заметил в самом начале это понятного слова).

Стр. 45

Автор упоминает о том, что одним из минусов Питона является скорость выполнения приложений, она не так высока как у С/С++, т.к. Питон это интерпретируемый ЯП.

Но вместе с тем Питон можно использовать для "быстрый" приложений, это достигается комбинированием компилированных и оптимизированных библиотек расширения. Т.е. для ускоренных вычислений мы должны выделить из приложения компоненты, требующие скорости, в виде компилированных расширений и связать их системой сценариев на Питоне.

Как это делается пока не известно, но нужно держать такие моменты в голове, нужно будет изучить далее.

стр. 47

Список компаний и областей где активно используется Питон.

стр. 48

Области применения Питон:

- Системное программирование. Питон применяется для создания переносимых программ и утилит системного администрирования (так называемыми инструментами командной оболочки).

- Графический интерфейс. Тут всё понятно, пишется GUI для разных ПО.

- Веб-сценарии. Например пакет mod\_python предназначен для запуска сценариев на языке Питон по управлением веб-сервера Апач (что очень нам подходит, т.к. у нас как раз Апач и стоит).

Также существуют пакеты веб-разработки, один из популярных это - Django и пр. (в т.ч. технология веб - MVC).

- Интеграция компонентов. Позволяет интегрировать код на Питоне в различные ЯП и приложения различных ОС.

- Приложения баз данных. В Питоне имеются интерфейся для работы со всеми основными реляционными БД.

- Быстрое создание прототипов. Информация об объединении нескольких модулей на разных ЯП, пока не понятно для меня, надеюсь дальше станет яснее.

- Программирование математических вычеслений. Расширение NumPy для математических вычислений, за счет интеграции с математическими библиотеками, написанными на компилирующих языках программирования - превращает Питон в сложный, но удобный инструмент программирования математических вычислений.

- Игры, изображения, искусственный интеллект, XML, управление роботами и многое другое.

стр. 53

- Питон объектно-ориентированный язык. Но при этом ООП не является обязательным, Питон поддерживает и процедурный стиль программирования.

- Питон свободный ЯП.

- Питон переносим, т.е. на разных ОС один и тот же код, использующий стандартные библиотеки, будет выполняться интерпретатором без проблем (должен выполняться).

- Питон мощный ЯП. Динамическая типизация (привет Руби!); автоматическое управление памятью; модульное программирование (можно создавать модули, классы, исключения); встроенные типы объектов (строки, списки, словари); встроенные инструменты (конкатенация, сортировка, и т.д.); библиотеки утилит; утилиты сторонних разработчиков.

- Питон соединыемый. программы на Питоне могут с легкостью "склеиваться" с компонентами, написанными на других ЯП. Например пишем программу на Питоне, а потом по частям перенести на С (для скорости, например).

- Питон удобен.

- Питон прост в изучении. Хотя эта книга занимает почти 1300 страниц...

- Питон назван в честь комедийного сериала "Монти Пайтон". Может нужно произносить Пайтон, а не Питон, но буду писать здесь - Питон, потом переучимся, если что.

Интерпретатор - это такой модуль, который исполняет другие программы. Когда мы пишем код на Питоне, интерпретатор читает нашу программу и выполняет её инструкции. По сути, интерпретатор - это слой программной логики между нашим программным кодом и аппаратурой нашего ПК.

Схема выполнения программ на Питоне:

Исходный текст (m.py) -- Байт-код (m.pyc) -- Выполнение (PVM)

Интерпретатор транслирует текст программы (инструкции) в байт-код - низкоуровневое, платформонезависимое представление исходного текста программы.

Как только программа скомпилируется в байт-код он передается виртуальной машине Питон (PVM) - привет Java! Формально, PVM это последняя составляющая того, что называют интерпретатором Питон.

Интерпретатор Jython (первоначальное - JPython) - альтернативная реализация Питона с тесной интеграцией с Java. В ней код на Питоне компилируется в байт-код Java, а затем передается JVM (вирутальной машине Java).

IronPython - реализация Питона предназначена для обеспечения интеграции программ с приложениями, созданными для работы в среде Microsoft .NET Framework и Mono - открытом эквиваленте для ОС Linux.

Существует еще динамический компилятор Psyco, который во время компиляции переводит некоторые части байт-кода в двоичный код, для ускорения.

Shedsckin C++ транслирует код Питона в код на С++, который затем можно скомпилировать в машинный код. На момент создания книги он был в стадии тестирования и доработки, может уже в промышленных масштабах используетс, нужно проверить.

Также программы на Питоне можно переводить в настоящие исполняемые файлы (по типу .exe в Windows), которые объединяют в себя байт-код программы, PVM (интерпретатор) и файлы поддержки, необходимые программе.

В среде Питон файлы, в которых сохранен код, называют - модулями. Часто их называют программами на языке Питон. Файлы модулей, которые запускаются на выполнение непосредственно, иногда называют - сценариями, этим неофициальным термином обозначаются файлы программ верхнего уровня. Термин модуль зарезрвирован для обозначения файлов, которые могут импортироваться другими файлами.

В Unix системах возможно понадобится вводить путь к самому Питону, что-то типа /usr/bin/pythhon чтобы запускать программы (сценарии).

**Глава 3. Запускаем программы**

! - Ознакомимся с инструкциями import и from. И изучим запуск программ, модулей, сценариев.

# Первый сценарий на Питоне

import sys

print(sys.platform)

print(2\*8)

x = 'Spam'

print(x\*8)

- импортирует модуль Python (библиотеку дополнительных инструментов), чтобы позднее получить название платформы

- sys.platform это просто строковая переменная, содержимое которой идентифицирует тип компьютера, на котором выполняется сценарий.

- трижды вызывает функцию print и отображает результаты

- создает переменную x, которая создается в момент присваивания ей строкового объекта

Заметим, что мы сохранили этот сценарий под именем script1.py хотя можно было и без расширения, но расширение \*.py подразумевает, что сценарий можно будет в дальнейшем импортировать. Поэтому указываем его, т.к. этим действием мы обеспечим возможность дальнейшего использования кода.

Данные вывода можно передать в файл (например текстовый)командой:

script1.py > info.txt

- т.о. мы наблюдаем т.н. "Перенаправление потоков"

В Unix можно указывать специальный комментарий, который сможет запускать сценарий без указания пути к Питону.

Т.н. "hash bang":

#!/usr/bin/python

print("Hello men")

стр. 90 - о команде env, которая позволит запустить Питон если вы не знаете его расположение на компьютере.

Кстати напомним, что если в конце сценария добавить функцию input() то после вывода данных программы командная строка не закроется сама (как это может быть если мы запустим программу в Виндовс нажатием на ярлык файла).

В Питоне 3.0 функция input() заменила собой функцию raw\_input() из Питона 2.6

**! - О модулях.**

Итак, каждый файл с исходным текстом на Питоне, имя которого оканчивается расширением .py является модулем.

Другие файлы могут обращаться к программным компонентам, объявляемым модулем, импортируя этот модуль.

По сути, инструкция import (здесь впервые мы делаем акцент на различиях инструкций и функций в Питоне, далее будем пытаться понять в чем они различаются и вообще понимать суть обоих терминов)

В итоге, после импорта, содержимое модуля становится доступным внешнему миру через его атрибуты.

Модульная система является центральной идеей Питона.

Крупные программы обычно организованы в виде множества файлов модулей, которые импортируют и испольщуют функциональные возможности из другших модулей. При этом один из модулей определяется как основной файл верхнего уровня, который запускате всю программу.

Отметим, что операция импорта, на заключительном этапе, приводит к выполнению программного кода загружаемого файла. Т.о. импорт файла является еще одним способом запустить его.

-- Поработаем с импортом модулей (файлов).

скопируем наш script1.py в папку с Питоном (C:\Python34) и попробуем его импортировать.

>>> import script1

win32

256

SpamSpamSpamSpamSpamSpamSpamSpam

- видим, что мы запустили выполнение инструкций в этом модуле/файле (см. выше о запуске).

>>> import script1

>>> import script1

- а новая попытка импорта ни к чему не привела, интерпретатор не выполняет никаких действий и не выводит ничего. Даже если мы изменим теперь script1.py все равно ничего не будет выводиться.

Однократный запуск модуля при импорте сделан специально - т.к. эта операция требует слишком больших затрат вычислительных ресурсов, чтобы выоплнять её несколько раз в ходе выполнения программы.

- маленькое отступление - В ходе выполнения импорта, производится поиск файлов, компиляция их в байт-код и выполнение этого байт-кода. Вот почему операция затратная.

Для многократного запуска файла в одном сеансе можно воспользоваться встроенной функцией reload, доступной в модуле imp из стандартной библиотеки. В Питоне 2.6 эта функция была обычной встроенной, но в Питоне 3.0 она была перенесена в модуль imp.

>>> from imp import reload

>>> reload(script1)

win32

256

SpamSpamSpamSpamSpamSpamSpamSpam

<module 'script1' from 'C:\\Python34\\script1.py'>

- Инструкция from в этом примере просто копирует имя функции из модуля. Функция reload загружает и запускает текущую версию программного кода в файле.

Даже если файл был изменен в другом окне reload также выводит и изменения (изменили 2\*8 на 2\*10):

>>> reload(script1)

win32

1024

SpamSpamSpamSpamSpamSpamSpamSpam

<module 'script1' from 'C:\\Python34\\script1.py'>

- только не забываем изменять модуль (script1) котоырй находится именно в папке с Питоном, а не где-то еще откуда оне был скопирован - наступил на эти грабли, изменял script1.py, который был в папке D:\Job\python и удивлялся - почему не видно изменений в интерпретаторе.

Еще одно замечание - функция reload ожидает получить имя уже загруженного модуля, поэтому модуль нужно перед этим импортировать (что мы и сделали вначале).

Заметим, что reload - это функция, которая вызывается (поэтому нужны скобки вокруг имени модуля), import - это инструкция, и скобок не нужно когда что-то импортируем.

<module 'script1' from 'C:\\Python34\\script1.py'> - это дополнительная строка, представление результата, возвращаемого функцией reload после перезагрузки модуля. Более подробно обсудим её позднее.

Можно импортировать модуль imp (import imp) и затем использовать синтаксис - imp.reload(M) или так как мы обсудили выше. Этот новый синтаксис формата - module.attribute

Особенность - имена, загруженные с помощью инструкции from, не обновляются вызовом функции reload, а имена, загруженные инструкцией import - обновляются, как мы видели из примеров выше.

**О модулях. Атрибуты.** стр. 97

В широком понимании модули играют роль библиотек инструментов. Модуль - это, главным образом, всего лишь пакет имен переменных, известный как - пространство имен. Имена внутри этого пакета называются атрибутами, т.е. атрибут - это имя переменной, которая связана с определенным объектом (таким как модуль).

В типичном случае импортирующий программный код получает доступ ко всем именам верхнего уровня, определяемым в файле модуля.

Эти имена обычно связаны с функциональными возможнстями, экспортируемыми модулем - функциями, классами, переменными и т.д., которые предназначены для использования в других файлах и программах.

Снаружи доступ к именам в файле модуля можно получить с помощью двух инструкций, import и from, а также с помощью функции reload.

например, файл myfile.py содержит код:

title = "To beer or not to beer"

При импортировании этого модуля выполняется его программный код, котоый создает атрибут модуля. В нашем случае инструкция присваивания создает атрибут с именем title.

Доступ к этому атрибуту можно получить двумя способами.

Первый - загрузить модуль инструкцией import и обратиться к атрибуту по его имени, уточнив его именем модуля:

>>> import myfile

>>> print(myfile.title)

To beer or not to beer

Синтаксис object.attribute позволяет получить доступ к любому атрибуту в любом объекте.

Второй способ - воспользоваться инструкцией from:

>>> from myfile import title

>>> print(title)

To beer or not to beer

Инструкция from во многом подобна инструкции import, которая выполняет присваивание имен в импортирующем компоненте.

Инструкция from копирует атрибуты модуля так, что они становятся простыми переменными в программном коде, выполняющем импорт, благодаря чем мы и обращаемся к атрибутам без указания модуля (в отличие от синтаксиса object.attribute).

Т.о. мы видим, что импортирующий компонент получает доступ к именам, определенным в файле на верхнем уровне. Это могут быт ьне только переменные, но и функции и классы и т.п., что очень важно для языка Питон.

Рассмотрим пример модуля с тремя переменными.

a = 'Billy'

b = 'Villy'

c = 'Dilly'

print(a, b, c)

>>> import myfile

Billy Villy Dilly

>>> myfile.a, myfile.b

('Billy', 'Villy')

>>> from myfile import a, b, c

>>> a, b

('Billy', 'Villy')

Итак, видим, что инструкция import получает модуль со всеми его атрибутами, а from получает копии имен из этого модуля.

*Модули в Питоне являются самой крупной структурной единицей в программах этого ЯП.*

Вообще программы на Питоне состоят из множества модулей свзязанных инструкциями import. Т.о. модули позволяют уменьшить вероятность конфликтов имен в программном коде, даже если они одинаковые (модули то разные), можно и запутаться в модулях, конечно, но это уже личное дело разработчика.

***Не забудьте****,* что инструкция from может затереть переменные в импортирующем файле, если в импортируемом модуле есть переменные с такими же именами. Будьте бдительны и осторожны, следите за именами в обеих файлах.

Еще одна любопытная функция exec. Позволяет выполнять файлы практически не импортируя модуль. Причем след. вызов exec будет выполнять текущую версию файла и даже при изменении его выводить всё правильно.

>>> exec(open('script1.py').read())

win32

1024

SpamSpamSpamSpamSpamSpamSpamSpam

>>> exec(open('script1.py').read())

win32

16384

SpamSpamSpam

>>>

**Предупреждение!** Использование функции exec затирает существущие переменные, пример:

>>> x = 99

>>> exec(open('script1.py').read())

win32

16384

SpamSpamSpam

>>> x

'Spam'

***Часть 2. Типы и операции*** стр. 119

**Глава 4. Введение в типы объектов Питон**. стр. 121.

Данные в Питон представлены в форме объектов - либо встроенных, либо созданных разработчиком.

Объекты - это области памяти со значениями и ассоциированными с ними наборами операций.

1. Программы делятся на модули.

2. Модули содержат инструкции.

3. Инсрукции состоят из выражений.

4. Выражения создают и обрабатывают объекты.

Как мы вскоре узнаем, значительная доля работы приходится на реализацию объектов, известных также как структуры данных, которые предназначены для представления составляющих предметной области.

Таблица 4.1 Базовые типы данных. стр. 123.

Числа

Строки

Списки (типо массивы) []

Словари (типо хеш) {}

Кортежи (ни то ни се) ()

Строки, списки, кортежи - это, так называемые, последовательности в языке Питон, упорядоченные коллекции объектов. Естественно, строки попроще последовательности, другие посложнее, но в целом все они собой являют тип объектов - последовательности.

Со строками сложнее, они нам видятся цельными объектами, но по сути являются последовательностями односимвольных строк (т.е. символов).

Такая вот особенность этих базовых типов в Питон.

Далее мы узнаем, что структурные элементы программ, такие как функции, модули и классы, также являются объектами в Питоне.

Когда выполняется код:

>>> 'spam'

- то, говоря техническим языком, выполняется выражение-литерал, которое генерирует и возвращает новый строковый объект. Подобным образом создаются объекты других типов.

Отметим, что как только создается объект, он будет ассоциирован со своим собственным набором операций на протяжении всего времени существования. Т.е. мы можем выполнить над объектом только те операции, которые применимы к его типу.

**Методы в Питоне**, это - функции, присоединенные к объекту, которые запускаются выражением вызова.

Например, относительно строк, метод find возвращает смещение переданной ему подстроки или -1 если поиск не увенчался успехом, а replace проводит глобальный поиск с заменой.

Но при этом объект, к которому мы применяли методы (наша строка-испытатель 'spam'), не изменяется, а создается новая строка.

Т.е. отметим, что выражение вида "s = (s.replace('pa', 'XYZ'))" строку 'spam' изменяет, а "s.replace('pa', 'XYZ')" не изменяет, просто метод возвращает новую строку, а прежняя остается собой. Вот такие вот особенности.

split() - разбивает строку по разделителю и и создает список строк

line = 'aaa, sss, bbb, cccccc, dd'

line.split(',')

['aaa', ' sss', ' bbb', ' cccccc', ' dd']

upper() - преобразует символы в верхний регистр

isalpha() - проверка содержимого: isalpha, isdigit и т.д.

s = 'spam'

>>> s.isalpha()

True

>>> s.isdigit()

False

Существуют также методы подстановки:

>>> '%s, eggs, and %s' % ('spam','SPAM')

'spam, eggs, and SPAM'

>>> '{0}, eggs, and {1}'.format('spam','SPAM!')

'spam, eggs, and SPAM!'

Следует знать, что инструментальные стредства Питон делятся на несколько уровней:

- универсальные операции, которые могут применяться к нескольким типам, реализованы в виде встроенных функций и выражений (типа len(x), x[0] и др.);

- а операции, специфичные для определенного типа, реализованы в виде методов (см. выше методы для строк).

Помощь по методам мы можем получить применив команду - dir(s), а т.к. мы знаем, что s у нас по прежнему строка, то получим все методы относящиеся к строкам.

Точнее сказать, функция dir() возвращает нам список всех атрибутов, определенных внутри модуля.

Имена вида \_\_len\_\_ представляют собой реализацию строкового объекта и доступны для поддержки специализации. В общем случае двойное подчеркивание вначале и конце имени используются интерпретатором Питон для обозначения особенностей внутренней реализации.

А имена без подчеркиваний обозначают методы строковых объектов.

Так же мы можем получим специализированные имена и имена методов для любого другого вида объектов.

Справку по конкретному методу получаем командой - help(s.upper)

Пробовал по функциям с подчеркиваниями смотерть справку, но там непонятно, вводил например - help(len(s)) - т.е. по типу как используется эта функция, и получил кучу вывода непонятного, с командой - help(iter(s)) вывод получился короче, т.е. справку я смотрю вроде правильно, но описание непонятное.

**Надо учить английский !!!!!!**

Использование шаблонов. Как я понял шаблоны в Питоне это т.н. регулярные выражения.

>>> import re

>>> match = re.match('Hello[ \t]\*(.\*)world', 'Hello Python world')

>>> match.group(1)

' Python '

В этом примере выполняется поиск строки, начинающейся со слова 'Hello', вслед за которым следуют ноль или более символов табуляции или пробелов, за которыми могут следовать произвольные символы, которые будут сохранены, как группа совпадения, и завершающая словом "world".

Если такое совпадение будет найдено, части этой подстроки будут доступны в виде групп. Например следующий шаблон возвращает три группы, разделенные символами слеша:

>>> match = re.match('/(.\*)/(.\*)/(.\*)', '/usr/home/userlib')

>>> match.groups()

('usr', 'home', 'userlib')

**Списки.**  стр. 133

Списки - это самое общее представление последовательностей, реализованных в языке Питон.

Списки - это упорядоченные по местоположению коллекции объектов произвольных типов, размер которых не ограничен.

В отличие от строк списки являются изменяемыми объектами.

>>> l = [123, 'spam', 1.23]

>>> len(l)

3

>>> l[0]

123

>>> l[:-1]

[123, 'spam']

>>> l + [4,5,6]

[123, 'spam', 1.23, 4, 5, 6]

>>> l

[123, 'spam', 1.23]

Как видим, со списками мы можем производить те же операции, что и со строками, только результатом операций являются списки, а не строки.

Списки в Питоне являются аналогами массивов в других ЯП, но обладают более широкими возможностями.

>>> l.append('NI')

>>> l

[123, 'spam', 1.23, 'NI']

>>> l.pop(2) # аналог этой команды - del l[2]

1.23

>>> l

[123, 'spam', 'NI']

>>> m = ['bb', 'aa','cc']

>>> m.sort()

>>> m

['aa', 'bb', 'cc']

>>> m.reverse()

>>> m

['cc', 'bb', 'aa']

>>>

Как видим, т.к. списки являются изменяемыми, большинство методов списков не создают новый список, а изменяют оригинал.

При обращении за выход границ массива или попытке присвоения значения элементу за границей массива (l[99]=1) происходит ошибка и программа выдает соответствующее сообщение.

**Вложенные списки.**

>>> m = [[1,2,3],[4,5,6],[7,8,9]]

>>> m

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> m[1] # получить вторую строку (второй список)

[4, 5, 6]

>>> m[1][2] # из второй строки получить элемент с индексом 2

6

Вместе с тем этот вложенный список можно представтиь в виде матрицы.

И при обработке матрицы можно эффективно воспользоваться - выражениями генераторов списков.

>>> col = [row[1] for row in m]

>>> col

[2, 5, 8]

>>> m

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

- вот такой командой мы выбрали элементы второго столбца, при этом оригинальная матрица не изменилась.

Команда переводится так - "Получить элементы row[1] из каждой строки матрицы m и создать из них новый список".

>>> [row[1] + 1 for row in m]

[3, 6, 9]

>>> m

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

- добавить 1 к каждому элементу в столбце 2

>>> [row[1] for row in m if row[1] % 2==0]

[2, 8]

>>> m

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

- отфильтровать в выводе результата нечетные значения

Генераторы списков возвращают новые списки, но могут использоваться и для любых других объектов, допускающих выполнение итераций.Смотрим пару примеров.

Например выберем элементы по диагонали матрицы:

>>> diag = [m[i][i] for i in[0,1,2]]

>>> diag

[1, 5, 9]

Или дублирование символов в строке:

>>> doubl = [c \* 2 for c in 'spam']

>>> doubl

['ss', 'pp', 'aa', 'mm']

Можно использовать круглые скобки для создания генераторов, которые воспроизводят результаты по требованию.

>>> g = (sum(row) for row in m)

>>> g

>>> next(g)

6

>>> next(g)

15

>>> next(g)

24

>>> next(g)

- если далее продолжить команду next(g) - получаем сообщение об ошибке.

Можно еще обернуть вызов функции в список и она вернет все значения сразу:

>>> list(map(sum, m))

[6, 15, 24]

С помощью генераторов можно создавать множества и словари:

>>> {sum(row) for row in m} # создадим множество сумм строк

{24, 6, 15}

>>> {i:sum(m[i]) for i in range(3)} # Таблица пар ключ\значение сумм строк

{0: 6, 1: 15, 2: 24}

>>> [ord(x) for x in 'spaam'] # список кодов символов

[115, 112, 97, 97, 109]

>>> {ord(x) for x in 'spaam'} # множество ликвидирует дубликаты

{112, 97, 115, 109}

>>> {x: ord(x) for x in 'spaam'} # ключи словарей являются уникальными

{'a': 97, 'm': 109, 's': 115, 'p': 112}

**Словари.** стр. 137

Словари в Питоне - известны как отображения, как-бы не являются последовательностями.

Отображения - это коллекция объектов, но доступ к ним осуществляется не по определенным смещениям от начала коллекции, а по ключам.

Отображения не предусматривают какого-либо упорядочения элементов по их позиции, они просто отображают ключи на связанные с ними позиции.

Словари - единственный тип отображений в Питоне. Они также изменяемые, как списки.

Определим словарь как литерал, обрамляется он в фигурные скобки, как хеш в Руби:

>>> d = {'food': 'spam', 'quant':4, 'color':'red'}

>>> d

{'quant': 4, 'food': 'spam', 'color': 'red'}

>>> d['food']

'spam'

>>> d['quant'] +=1 # здесь мы прибавили 1 к значению ключа quant

>>> d

{'quant': 5, 'food': 'spam', 'color': 'red'}

Есть другие способы создания словарей:

>>> d={}

>>> d['name']='Bob'

>>> d['job']='dev'

>>> d['age']=40

>>> d

{'age': 40, 'name': 'Bob', 'job': 'dev'}

Любопытная особенность, маленькое различие в выводе отдельного элемента и при использовании меода print:

>>> print(d['name'])

Bob

>>> d['name']

'Bob' # видим, что в этом случае имя в кавычках

**Вложенность словарей.**

Допустим нам нужно описать человека более подробно, для этого мы несколько усложним структуру словаря, на примере будет всё понятно:

# создаем словарь с вложениями

>>> rec = {'name':{'first':'Bob', 'last':'Smith'}, 'job':['dev','mgr'], 'age':40.5}

# выборку можно осуществлять стандартно по ключу, смотрим вложенный словарь

>>> rec['name']

{'last': 'Smith', 'first': 'Bob'}

# выборка по вложенным элементам

>>> rec['name']['last']

'Smith'

# смотрим вложенный список

>>> rec['job']

['dev', 'mgr']

>>> rec['job'][-1]

'mgr'

>>> rec['job'].append('janitor')

>>> rec

{'age': 40.5, 'name': {'last': 'Smith', 'first': 'Bob'}, 'job': ['dev', 'mgr', 'janitor']}

Здесь мы видим как сильно похожи и взаимосвязаны способы работы со списками и словарями.

К тому же эти примеры демонстрируют нам гибкость языка Питон, в сравнении с другими более строгими ЯП.

К тому же можно освободить память вручную, например просто присвоив значение rec = 0, хотя в Питоне память освобождается атвоматически (т.н. "сборка мусора"). При этом не забывайте проверить не находится ли объект в базе данных или это может быть файл.

**Сортировка. Цикл for**

Порядок вывода ключей словаря может не совпадать с порядком их определения, т.е. например:

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

>>> d

{'a': 1, 'c': 3, 'b': 2}

>>> d['d']=4

>>> d

{'a': 1, 'd': 4, 'b': 2, 'c': 3}

Отсортируем по ключам, методами keys sort и циклом for

>>> ks = list(d.keys())

>>> ks

['a', 'd', 'b', 'c']

>>> ks.sort()

>>> ks

['a', 'b', 'c', 'd']

>>> for key in ks:

print(key, '=>', d[key])

a => 1

b => 2

c => 3

d => 4

Или можно воспользоваться новой функцией - sorted и не нужно будет трех этапов, причем переменную key мы можем назвать хоть mmm - оказывается это не специальное обозначение ключа (как я вначале подумал), а просто переменная относящаяся к ключу.

Я забыл, что ks - это **уже отсортированный список ключей**, а не весь наш словарь. А во втором случае, когда используем sorted, всё равно всё работает даже если вместо key писать bbb, такая вот крута функция.

>>> d

{'a': 1, 'd': 4, 'b': 2, 'c': 3}

>>> for key in sorted(d):

print(key, '=>', d[key])

a => 1

b => 2

c => 3

d => 4

Вот мы и познакомились с циклом for.

Переменная цикла, определяемая пользователем (в нашем случае это key), служит для ссылки на текущий элемент.

Еще один пример цикла for, в котором происходит обход всех символов и перевод в верхний регистр:

>>> for c in 'spam':

print(c.upper())

S

P

A

M

Пример цикла while:

>>> x = 4

>>> while x>0:

print('spam!'\*x)

x -=1

spam!spam!spam!spam!

spam!spam!spam!

spam!spam!

spam!

Цикл for и выражения-генераторы фактически представляют собой универсальное средство выполнения итераций. Обе констркции способны работать с любыми объектами, которые поддерживают протокол итераций.

Любой инструмент Питона, сканирующий объект слева направо использует протокол итераций. Пример - функция sorted.

Любой генератор списков, например:

>>> squar = [x\*\*2 for x in [1,2,3,4,5]]

>>> squar

[1, 4, 9, 16, 25]

можно представит ьв виде эквивалентного цикла for:

>>> squar = []

>>> for x in [1,2,3,4,5]:

squar.append(x\*\*2)

>>> squar

[1, 4, 9, 16, 25]

Следует отметить, что генераторы и родственные им функции map и filter обычно выполняются быстре ечем цикл for.

Но о скорости лучше не беспокоиться, потому что главное правило Питона - это удобный код и простота, а оптимизацию можно оставить на потом.

**Проверка с помощью оператора if.**

>>> d

{'a': 1, 'b': 2, 'c': 3}

>>> d['e']=99

>>> d

{'a': 1, 'b': 2, 'e': 99, 'c': 3}

Если мы захотим проверить значение ключа d['f'] то получим сообщение об ошибке. Во избежание таких казусов можно использовать проверку.

>>> 'f' in d # работает только на ключах, по значениям выдает false

False

>>> 2 in d

False

>>> if not 'f' in d: # здесь мы использовали оператор if

print('missing')

missing

if мы рассмотрим позднее.

**Кортежи.** стр. 144

Объект-кортеж (tuple - "тьюпл" или "тьюпел"), в общих чертах напоминает список, который невозможно изменить - кортежи это последовательности, как списки, но они неизменяемые, как строки.

Кортеж заключается в круглые скобки. Поддерживает включение объектов различных типов и операции типичные для последовательностей.

>>> t = (1,2,3,4)

>>> len(t) # Длина

4

>>> t + (5,6) # Конкатенация

(1, 2, 3, 4, 5, 6)

>>> t[0] # Извлечение элемента или среза и т.п.

1

>>> t.index(4) # Значение 4 нах-ся в позиции 3

3

>>>t.count(4) # Значение 4 присутствует в единственном экземпляре

1

>>> t[0]=2

Traceback (most recent call last):

File "<pyshell#149>", line 1, in <module>

t[0]=2

TypeError: 'tuple' object does not support item assignment

- получаем ошибку, **кортежи неизменяемые.**

>>> t=('spam', 3.0, [11,22,33])

>>> t

('spam', 3.0, [11, 22, 33])

>>> t[1]

3.0

>>> t[2][1]

22

>>> t.append(4)

Traceback (most recent call last):

File "<pyshell#154>", line 1, in <module>

t.append(4)

AttributeError: 'tuple' object has no attribute 'append'

Главное достоинство кортежей - их неизменяемость, и в основном их используют для обеспечения целостности коллекций, передаваемых между компонентами программы.

**Файлы.** стр. 145

Файлы являются одним из базовых типов, но для них отсутствует возможность создания объектов в виде литералов (это естественно, т.к. файлы достаточно сложный и своеобразный объект).

Поэтому чтобы создать файл, нужно вызвать функцю open, передав ей имя файла и строку режима доступа к файлу (привет Руби!). Смотрим:

>>> f = open('data.txt', 'w')

>>> f.write('Hello\n')

6

>>> f.write('World\n')

6

>>> f.close()

- открывая файл для создания, указываем имя и 'w', **важно!!!** - если создаем файл не в текущем каталоге, а где-то в другом месте, то указываем полный путь к нему!

Затем записываем в него пару слов (при этом Питон показывает нам кол-во записанных байтов). И закрываем файл.

Теперь мы прочитаем содержимое файла и в строку и отобразим её:

>>> f = open('data.txt') # по умолчанию режим 'r' поэтому можно не указывать

>>> text = f.read()

>>> text

'Hello\nWorld\n'

>>> print(text)

Hello

World

>>> text.split() # Содержимое файла всегда явл-ся строкой

['Hello', 'World']

Запомните, что самый лучший способ чтения файлов состоит в том, чтобы не читать его содержимое целиком - файлы предоставляют итераторы, которые обеспечивают автоматическое построчное чтение содержимого файла в циклах for и в других контекстах.

Кстати, в Питоне текстовые файлы представляются в виде строк и автоматически кодируются и декодируются в Юникод, а двоичные файлы представляются в виде строк специального типа bytes, при этом никаких автоматических преобразований не происходит.

**Другие базовые типы.** стр. 147

**Множества** - это неупорядоченные коллекции уникальных и неизменяемых объектов. Создаются они функцией set или с помощью определения литералов и генераторов множеств. Множества напоминают словари, в которых ключи не имеют значений.

>>> x = set('spam') # множества можно создавать из последовательностей

>>> y = {'h', 'a', 'm'} # можно определять литералы множеств

>>> x, y

({'s', 'p', 'a', 'm'}, {'m', 'a', 'h'})

>>> x&y # Пересечение

{'m', 'a'}

>>> x|y # Объединение

{'s', 'p', 'a', 'm', 'h'}

>>> x-y # Разность

{'s', 'p'}

>>> y-x

{'h'}

>>> {x\*\*2 for x in [1,2,3,4]} # генератор множеств

{16, 1, 9, 4}

Кроме того появились в Питоне вещественные числа с фиксированной точностью и рациональные числа (т.е. дроби).

>>> 1/3 # вещественное число

0.3333333333333333

>>> (2/3) + (1/2)

1.1666666666666665

>>> import decimal # вещественные числа с фиксированной точностью

>>> d = decimal.Decimal('3.141')

>>> d+1

Decimal('4.141')

>>> decimal.getcontext().prec = 2

>>> decimal.Decimal('1.00')/decimal.Decimal('3.00')

Decimal('0.33')

>>> from fractions import Fraction # Рациональные числа, т.е. дроби

>>> f = Fraction(2, 3)

>>> f

Fraction(2, 3)

>>> f + 1

Fraction(5, 3)

>>> f + Fraction(1,2)

Fraction(7, 6)

>>> from ff import Fraction # fractions - это модуль оказывается

Traceback (most recent call last):

File "<pyshell#36>", line 1, in <module>

from ff import Fraction

ImportError: No module named 'ff'

Кроме того в Питон имеется логический тип данных - **True** и **False** (по сути 1 и 0). Кроме того существует объект **None**, обычно используемый для инициализации объекто и переменных:

>>> 1 > 2, 1 < 2

(False, True)

>>> bool('spam')

True

>>> x = None

>>> x

>>> print(x)

None

>>> l = [None] \* 20

>>> l

[None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None]

Отметим некоторые особенности программного кода в Питон.

Тип объекта, возвращаемый встроенной функцией type, в свою очередь сам является объектом. Причем в версиях Питона 3.х типы объектов были объеденены с классами.

Пример:

>>> type(l) # Переменная l представляет объект типа list, типы являются

<class 'list'> классами и наоборот

>>> type(type(l)) # Даже сами типы являются объектами

<class 'type'>

Исследовать типы объектов можно не только в интерактивной оболочке, но и в сценариях:

>>> if type(l) == type([]): # Проверка типа

print('yes')

yes

>>> if type(l == list): # С использованием имени типа

print('yes')

yes

>>> if isinstance(l, list): # Проверка в ООП стиле

print('yea')

yea

- Но использвание таких проверок практически всегда является неверным решением, т.к. наличие таких проверок отрицательно сказывается на гибкости кода, потомоу что такими проверками мы ограничиваем наш код в типах данных.

Программный код на Питоне ориентируется на использование интерфейсов объектов (набором поддерживаемых операций), а не их типов.

Это тесно связано с идеей полиморфизма, которую пропагандирует Питон, полиморфизм является одной из ключевых концепций этого ЯП.

**Классы, определяемые пользователем.** стр. 149

ООП рассмотрим позже, но пока познакомимся с классами.

Классы позволяют нам определять новые типы объектов.

Например создадим тип объектов, который моделирует сотрудников:

>>> class Worker:

def \_\_init\_\_(self, name, pay): # инициализация при создании

self.name = name # self - это сам объект

self.pay = pay

def lastName(self):

return self.name.split()[-1] # разбить строку по символам пробела

def giveRaise(self, percent):

self.pay \*= (1.0 + percent) # обновить сумму выплат

- кстати все def, в этом случае, должны быть на одном уровне в файле сценария (как здесь) иначе lastName и giveRaise будут воприниматься не как методы, а как атрибуты объекта класса Worker и в коде ниже будет ошибка.

Смотрим дальше и всё поймем.

Данный класс определяет новый тип объектов, которые обладают атрибутами name и pay (иногда атрибуты называют - информацией о состоянии), а также двумя описаниями поведения, оформленными в виде функций (которые обычно называют методами).

Обращение к имени класса как к функции приводит к созданию экземпляра нового типа, а методы класса автоматически получают ссылку на текущий экземпляр, обрабатываемый этими методами (аргумент self):

>>> bob = Worker('Bob Smith', 50000)

>>> sue = Worker('Sue Jones', 60000)

>>> bob.lastName()

'Smith'

>>> sue.lastName()

'Jones'

>>> sue.giveRaise(.10)

>>> sue.pay

66000.0

В этой модели ООП присутствует подразумеваемый объект "self" - внутри функций, определяемых в классе, всегда присутствует подразумеваемый объект, и в нашем случае это self.

В некотором смысле типы, основанные на классах, просто создаются на базе основных типов и используют их функциональные возможности.

В данном случае пользовательский класс Worker - это всего лишь коллекция, состоящая из строки и числа (name и pay соответсвенно), плюс функции, выполняющие обработку этих двух встроенных объектов.

По сути в Питоне все данные, которые обрабатываются сценариями, являются объектами. Объекты с которыми мы познакомились действительно являются объектами, но для работы с ними не требуется обязательно использовать ООП подход.

Теперь, поскольку объекты представляют самое фундаментальное понятие для программирования на языке Питон, все наше внимание в первую очередь будет сосредоточено на объектах.

Поэтому мы приступаем к более детальному изучению концепции типов данных.

**Глава 5. Числа.** стр. 153

**Числовые типы** в Питон:

- Целые и вещественные числа;

- Комплексные числа;

- Числа фиксированной точности;

- Рациональные числа;

- Множества;

- Логические значения;

- Целые числа неограниченной точности;

- Различные встроенные функции и модули для работы с числами.

В Питон 2.6 целые числа были обычные и длинные, в версии 3.х целые числа стали одного типа.

Шестнадцатиричные записываются 0х или 0Х и дальше число, восьмеричные 0о или 0О (число 0 и символ "о") и дальше восьмеричное число, двоичные начинаются с 0b или 0B.

Комплексные числа записываются в виде - действительная\_часть + мнимая\_часть, где мнимая\_часть завершается символом j или J. Мнимая\_часть может указываться без действительной части, т.к. действительная может быть ен важно (с технич точки зрения).

Операторы выражений:

+ , - , \* , / , >> , \*\* , & и другие.

Встроенные математические функции:

pow, abs, round, int, hex, bin и др.

Вспомогательные модули:

random, math и др.

**Операторы выражений.**

Одной из самых фундаментальных возможностей обработки чисел являются - выражения - комбинации чисел и операторов, которые выозвращают значения при выполнении интерпретатором Питон.

Таблица 5.2 Операторы выражений и правила определения старшинства. стр. 157-159

**Смешанные операторы и определение старшинства.**

Чем выше приоритет оператора тем ниже он стоит в таблице 5.2 и тем раньше он выполняется в смешанных выражениях.

Чтобы не было ошибо при чтении выражений можно группировать подвыражения круглыми скобками - (А+В)\*С например так.

**Смешивание типов и их преобразование.**

Пусть нам нужно сложить целое и вещественное число (40 + 3.14), как же Питон вычисляет тип результата?

В выражениях, где участвуют значения различных типов, интерпретатор сначала выполняет преобразование типов операндов к типу самого сложного операнда, а затем прмиеняет математику, специфиную для этого типа.

В Питоне можно принудительно преобразовывать тип, но это не обязательно:

>>> int(3.1415)

3

>>> float(3)

3.0

>>> 40 + 3.14 # вот пример автоматического приведения типа

43.14

Все эти автоматические преобразования производятся только при смешивании числовых типов в выражении.

Питон не выполняет автоматическое преобразование других типов, т.е. нельзя сложить строку и число без ручного преобразования типов (с этим мы познакомимся позднее).

>>> a = 'spam'

>>> b = 7

>>> a+b

...

TypeError: Can't convert 'int' object to str implicitly

>>> print(a+b) # даже функция print не работает в этом случае

...

TypeError: Can't convert 'int' object to str implicitly

Применимость математических операндов к разным типам объектов характеризуется полиморфизмом, т.е. выполняемая операция зависит от типов объектов-операндов, над которыми она выполняется.

Например + это сложение с числами и + это конкатенация со строками и списками.

**Числа в действии.** стр. 162

Займемся программированием! Так мы лучше поймем действия с числами.

Для начала присвоим двум переменным a и b целочисленные значения. Переменные - это всего лишь имена, создаваемые в языке Питон, которые используются для обозначения информации в программе. О переменных подробно изучим позже, сейчас несколько моментов:

- Переменные создаются с помощью операции присваивания;

- При вычислении имена переменных замещаются их значениями;

- Прежде чем переменная сможет участвовать в выражениях, ей должно быть присвоено значение

- Переменные являются ссылками на объекты и никогда не объявляются заранее.

>>> a = 3 # Создается имя

>>> b = 4

>>> a + 1, a - 1 # Сложение, вычитание

(4, 2)

>>> b \* 3, b / 2 # Умножение, деление

(12, 2.0)

>>> a % 2, b \*\* 2 # Деление по модулю (остаток), возведение в степень

(1, 16)

>>> 2 + 4.0, 2.0 \*\* b # Смешивание типов, преобразование

(6.0, 16.0)

С технической точки зрения результатами этих выражений являются - кортежи, состоящие из двух значений, т.к. мы вводили строки, которые содержали по два выражения, разделенных запятыми. Именно по этой причине результат отображается в круглых скобках.

И опять же напоминаем - перед использованием переменных им были присвоены значения.

Именно поэтому при использовании счетчиков вначале они должны инициализироваться хотя бы нулевым значением, или перед использованием списка инициализировать его пустым значением и т.п.

>>> b / 2 + a # Примеры автоматического порядка и ручного выполнения операторов

5.0

>>> print(b / (2.0 + a))

0.8

- кстати если бы хотели использовать целочисленное деление, то нужно было записать выражение так: b // 2 + a но об операции деления поговорим далее.

**Форматы отображения чисел.**

Обратимся к последнему прмиеру и посмотрим разницу:

>>> b / (2.0 + a) # в примере в книге это выглядит так

0.80000000004

*- но на моем ПК оба выражения одинаковы, это зависит от архитектуры компьютера*

>>> b / (2.0 + a)

0.8

>>> print(b / (2.0 + a))

0.8

>>> num = 1 / 3.0

>>> num

0.3333333333333333

>>> print(num)

0.3333333333333333

- вот опять у меня обычный выывод и с помощью print одинаковые, но нужно запомнить, что обычный вывод зачастую может быть длиннее чем вывод функции print.

>>> "%e" % num # Вывод с использованием выражения форматирования строк

'3.333333e-01'

>>> '%4.2f' % num # Альтернативный формат представления вещественных чисел

'0.33'

>>> '{0:4.2f}'.format(num) # Метод форматирования строк

'0.33'

- функция repr(), также как и функция автоматического вывода в интерактивной оболочке, выводит результаты в том виде, в каком они были указаны в программном коде (например 0.333333333333333);

- функция str(), как и операция print, обычно выполняет преобразование в более дружественное значение.

Некоторые объекты имеют оба варианта строкового представления.

Одно замечание - на моем ПК обе функции показывают одинаковое значение, это либо особенность архитектуры или влияние ОС на Питон.

**Операции сравнения, простые и составные.**

>>> 1 < 2

True

>>> 2.0 >= 1 # Числов 1 преобразуется в 1.0

True

>>> 2.0 == 2.0

True

>>> 2.0 == 2 # снова преобразование 2 в 2.0

True

>>> 2.0 != 2.0 # Проверка на неравенство, false потомоу что 2.0 равно 2.0

False

Можно комбинировать операторы сравнения, например для проверки вхождения числа в диапазон:

>>> x < y < z # выражения идентичны, но в первом случае оно короче и быстрее

True

>>> x < y and y < z

True

>>> x < y > z # более сложные цепочки

False

>>> x < y and y > z

False

>>> 1 < 2 < 3.0 < 4

True

>>> 1 > 2 > 3.0 > 4

False

Рассмотрим пример с комбинацией операторов сравнения:

>>> 1 == 2 < 3 # то же, что и 1 == 2 and 2 < 3

False

- в этом выражении интерпретатор не сравнивает значение False (результат операции 1 == 2) с числом 3 - с технической точки зрения это соответствовало бы выражению 0 < 3, которое должно было бы вернуть True (т.к. True и False - это всего лишь числа 1 и 0, расширенные приписанными им свойствами), а в нашем случае мы получили значение выражения соответствующего "1 == 2 and 2 < 3".

***Запомним*** эту особенность, т.к. это важно.

**Деление, классические, с округлением вниз и истинное.**

X / Y - в Питон 3.х этот оператор выполняет операцию истинного деления, которая всегда сохраняет дробную часть независимо от типов операндов.

X // Y - Деление с округлением вниз. Он всегда отсекает дробную часть, округляя результат до ближайшего наименьшего целого независимо от типов операндов.

>>> 10 / 4

2.5

>>> 10 // 4

2

>>> 10 / 4.0

2.5

>>> 10 // 4.0

2.0

- тип результата // зависит от типов операндов, если хотя один из них вещественное число, то и результат будет вещественным числом.

Оператор // используется в программах гораздо чаще чем мы могли подумать, например когда нужен целочисленный результат при деление целочисленных операндов.

**Округление вниз и усечение дробной части.**

Оператор // также называется оператором деления с округлением вниз - т.е. он округляет результат до ближайшего меньшего целого значения.

Посмотрим несколько примеров:

>>> import math

>>> math.floor(2.5)

2

>>> math.floor(-2.5)

-3

>>> math.trunc(2.5)

2

>>> math.trunc(-2.5)

-2

>>> 5 / 2, 5 / -2

(2.5, -2.5)

>>> 5 // 2, 5 // -2

(2, -3)

>>> 5 / 2.0, 5 / -2.0

(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0

(2.0, -3.0)

**Почему усечение так важно?**

>>> (5/2), (5/2.0), (5/-2.0), (5/-2) # Классическое деление

(2.5, 2.5, -2.5, -2.5)

>>> (5//2), (5//2.0), (5//-2.0), (5//-2) # Деление с округлением вниз

(2, 2.0, -3.0, -3)

>>> (9/3), (9.0/3), (9//3), (9//3.0) # Оба оператора

(3.0, 3.0, 3, 3.0)

- мы видим различие действия операторов и опыт программистов подсказывает нам, что иногда нам нужны будут не просто математические значения, а логические, целочисленные результаты делений для работы программ, поэтмоу нужно внимательно следить за операциями деления и уметь правильно прмиенять оба оператора.

**Комплексные числа.** стр. 171

Шестнадцатеричная, восьмеричная и двоичная фомы записи чисел.

>>> 0o1, 0o20, 0o377 # восьмеричные литералы

(1, 16, 255)

>>> 0x01, 0x10, 0xff # шестнадцатеричные литералы

(1, 16, 255)

>>> 0b1, 0b10000, 0b11111111 # двоичные литералы

(1, 16, 255)

>>> oct(64), hex(64), bin(64) # встроенные функции по переводу в системы счисления

('0o100', '0x40', '0b1000000')

функция int напротив - переводит числа из системы счисления в десятичную:

>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)

(64, 64, 64, 64)

Функция eval интерпретирует строку во входном аргументе как программный код на Питоне. Правда с ней нужно обращаться осторожно, иначе можно выполнить какой-то случайный код, например код подсунутый злоумышленниками и получит ьпустой жесткий диск например.

>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')

(64, 64, 64, 64)

Строковые методы форматирования чисел:

>>> '{0:o}, {1:x}, {2:b}'.format(64,64,64)

'100, 40, 1000000'

>>> '%o, %x, %X' % (64, 255, 255)

'100, ff, FF'

**Битовые операции.** стр. 174

- если будет нужно, то ознакомимся прямо в книге, здесь описывать не будем.

**Другие встроенные стредства.**

Рассмотрим модуль math:

>>> import math

>>> math.pi, math.e # константы

(3.141592653589793, 2.718281828459045)

>>> math.sin(2\*math.pi/180) # Синус, тангенс, косинус

0.03489949670250097

>>> math.sqrt(144), math.sqrt(2) # Квадратный корень

(12.0, 1.4142135623730951)

>>> pow(2, 4), 2\*\*4 # Возведение в степень

(16, 16)

>>> abs(-42.0), sum((1,2,3,4)) # Абсолютное значение, сумма

(42.0, 10)

>>> min(3,1,2,4), max(3,1,2,4) # Минимум, максимум

(1, 4)

>>> math.floor(2.567), math.floor(-2.567)

(2, -3)

- округление вниз (до ближайшего наименьшего целого)

>>> math.trunc(2.567), math.trunc(-2.567)

(2, -2)

- усечение (отбрасывание дробной части)

>>> int(2.567), int(-2.567)

(2, -2)

- усечение (преобразование в целое число)

>>> round(2.567), round(2.467), round(2.567, 2)

(3, 2, 2.57)

>>> (1/3), round(1/3, 2)

(0.3333333333333333, 0.33)

- округление (в Питоне 3.х)

Еще примеры вычисления корня:

>>> import math

>>> math.sqrt(144)

12.0

>>> 144 \*\* .5

12.0

>>> pow(144, .5)

12.0

Из всех этих примеров выделим одно замечание, модули - это внешние компоненты, а встроенные функции постоянно располагаются в пространстве имен, которое используется интерпретатором Питон по умолчанию для поиска имен, используемых программой.

Сейчас для нас актуально правило: если видишь модуль, то сразу вспоминай - импорт.

Пример еще одного импорта - модуль random:

>>> import random

>>> random.random()

0.6384048535018463

>>> random.random()

0.8669699558489125

>>> random.randint(1, 10)

1

>>> random.randint(1, 10)

10

>>> random.choice(['Hi bro', 'Guten tag', 'Hello love!'])

'Hi bro'

>>> random.choice(['Hi bro', 'Guten tag', 'Hello love!'])

'Guten tag'

**Другие числовые типы.** стр. 177

**Числа с фиксированнйо точностью.**

Известны как числа типа - Decimal, по сутия являются вещественными, с фиксированным количеством знаков после запятой.

Причем можно указывать будут ли лишние числа усекаться или округляться, очень удобно для бухгалтерии, например.

>>> 0.1 + 0.1 + 0.1 - 0.3

5.551115123125783e-17

- это пример проблемы с ограниченным количеством битовв представлении вещественных чисел.

>>> from decimal import Decimal

>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')

Decimal('0.0')

- вот так решается проблема с битами, с помощью функции Decimal из модуля decimal.

- если одно из чисел будет с двумя знаками (0.01) а остальные с одним (0.1), то рещульатт будет выведен с двумя знаками, Питон автоматически выбирает наибольшую точность.

**Настройка точности.**

Настройка в модуле decimal позволяет настраивать точность и режим округления (вверх, вниз и т.д.):

>>> import decimal

>>> decimal.Decimal(1)/decimal.Decimal(7)

Decimal('0.1428571428571428571428571429')

>>> decimal.getcontext().prec = 4

>>> decimal.Decimal(1)/decimal.Decimal(7)

Decimal('0.1429')

>>> 1999 + 1.33 # в книге, в этой строке ответ был 2000.32999999999

2000.33 # мой ПК решает всё правильно, странно...

>>> decimal.getcontext().prec = 2

>>> pay = decimal.Decimal(str(1999 + 1.33))

>>> pay

Decimal('2000.33')

**Менеджер конекста объектов класса Decimal.**

В Питоне имеется возможность временно переопределять точность с помощью инструкции with менеджера конекстов. После выхода за пределы инструкции настройки восстанавливаются:

>>> import decimal

>>> decimal.Decimal('1.00')/decimal.Decimal('3.00')

Decimal('0.33')

>>> with decimal.localcontext() as ctx:

ctx.prec = 1

decimal.Decimal('1.00')/decimal.Decimal('3.00')

Decimal('0.3')

>>> decimal.Decimal('1.00')/decimal.Decimal('3.00')

Decimal('0.33')

**Рациональные числа.** стр. 179

По сути это дроби с числителем и знаменателем, числовой тип называется - Fraction.

Decimal и Fraction, похожи между собой по поведению и некоторым свойствам. Fraction так же находится в модуле.

Чтобы создать объект этого типа, необходимо импортировать модуль и вызвать конструктор класса, передав ему числитель и знаменатель:

>>> from fractions import Fraction

>>> x = Fraction(1, 3)

>>> y = Fraction(4, 6) # автоматически будет упрощено функцией gcd до 2/3

>>> x

Fraction(1, 3)

>>> y

Fraction(2, 3)

>>> print(y)

2/3

Математические операции:

>>> x + y

Fraction(1, 1)

>>> x - y

Fraction(-1, 3)

>>> x \* y

Fraction(2, 9)

Создадим рациональные числа другими способами:

>>> Fraction('.25')

Fraction(1, 4)

>>> Fraction('1.25')

Fraction(5, 4)

>>> Fraction('.25') + Fraction('1.25')

Fraction(3, 2)

**Точность.**

>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)

Fraction(0, 1)

- как и в случае с Decimal мы получим точный вывод операции, по сравнению с числом выше - 5.551115123125783e-17

>>> 1/3

0.3333333333333333

>>> Fraction(1, 3)

Fraction(1, 3)

>>> decimal.getcontext().prec = 2

>>> decimal.Decimal(1)/ decimal.Decimal(3)

Decimal('0.33')

- и снова мы видим, что фактически рациональные числа сохраняют точность и автоматически упрощают результат.

>>> (1/3) + (6/12)

0.8333333333333333

>>> Fraction(1, 3) + Fraction(6, 12)

Fraction(5, 6)

>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))

Decimal('0.83')

**Преобразование и смешивание в выражениях значений разных типов.**

Для преобразования в рациональные числа был реализован метод as\_integer\_ratio, объекты рациональных чисел обладают методом from\_float.

- в примерах ниже символ \* - это специальный синтаксис распаковывания кортежа в отдельные аргументы, подробнее рассмотрим позднее.

Рассмотрим некоторые способы преобразований как в одну так и другую сторону:

>>> (2.5).as\_integer\_ratio()

(5, 2)

>>> f = 2.5

>>> z = Fraction(\*f.as\_integer\_ratio())

>>> z

Fraction(5, 2)

>>> x

Fraction(1, 3)

>>> x + z

Fraction(17, 6)

>>> float(x)

0.3333333333333333

>>> float(z)

2.5

>>> float(x + z)

2.8333333333333335

>>> 17/6

2.8333333333333335

>>> Fraction.from\_float(1.75)

Fraction(7, 4)

>>> Fraction(\*(1.75).as\_integer\_ratio())

Fraction(7, 4)

При смешивании некоторых типов в выражениях иногда, чтобы сохранить точность, треубется вручную выполнить преобразование в тип Fraction.

>>> x

Fraction(1, 3)

>>> x + 2

Fraction(7, 3)

>>> x + 2.0

2.3333333333333335

>>> x + (1/3)

0.6666666666666666

>>> x + (1./3)

0.6666666666666666

>>> x + (4./3)

1.6666666666666665

>>> x + Fraction(4, 3)

Fraction(5, 3)

- В некоторых случаях при преобразовании вещественного числа в рациональное происходит потеря точности, потмоу что в своем первоначальном виде вещ. число может быть неточным.

- Поэтому нужно иногда вручную упростить число.

>>> 4.0 / 3

1.3333333333333333

>>> (4.0 / 3).as\_integer\_ratio() # произойдет потеря точности

(6004799503160661, 4503599627370496)

>>> x

Fraction(1, 3)

>>> a = x + Fraction(\*(4.0/3).as\_integer\_ratio())

>>> a

Fraction(22517998136852479, 13510798882111488)

>>> 22517998136852479 / 13510798882111488 # 5/3 (или близкое к нему!)

1.6666666666666665

>>> a.limit\_denominator(10) # упростить до ближайшего рационального

Fraction(5, 3)

**Множества.** стр. 183

По определению, каждый элемент может присутствовать в множестве только в единственном экземпляре, не смотря на на то сколько раз он будет добавлен.

Множества применяются чаще в приложениях обработки числовых данных и при работе с БД.

**Основы множеств.**

Создать объект множества можно функцией set, передав ей последовательность или др. объект, поддерживающий итерации:

>>> x = set('abcde')

>>> y = set('bdxyz')

>>> x

{'a', 'c', 'b', 'd', 'e'}

>>> 'e' in x # Проверка вхождения в множество

True

>>> x - y # Разность множеств

{'a', 'c', 'e'}

>>> x | y # Объединение

{'a', 'z', 'x', 'y', 'c', 'b', 'd', 'e'}

>>> x & y # Пересечение

{'b', 'd'}

>>> x ^ y # Симметрическая разность (XOR)

{'a', 'z', 'x', 'y', 'c', 'e'}

>>> x > y, x < y # Надмножество, подмножество

(False, False)

**Методы** (весь перечень смотрите по функции dir):

add - вставляет новый элемент в множество;

update - выполняет объединение;

remove - удаляет элемент по его значению.

>>> z = x.intersection(y)

>>> z

{'d', 'b'}

>>> z.add('spam')

>>> z

{'d', 'spam', 'b'}

>>> z.update(set(['x', 'y']))

>>> z

{'y', 'd', 'x', 'spam', 'b'}

>>> z.remove('b')

>>> z

{'y', 'd', 'x', 'spam'}

Операторы, родственные упомянутым выше, могут работать с любыми объектами итерируемых типов:

>>> s = set([1, 2, 3])

>>> s | set([3, 4])

{1, 2, 3, 4}

>>> s | [3, 4]

Traceback (most recent call last):

File "<pyshell#121>", line 1, in <module>

s | [3, 4]

TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> s.union([3, 4])

{1, 2, 3, 4}

>>> s.intersection((1, 3 , 5))

{1, 3}

>>> s.issubset(range(-5, 5))

True

**Изменения в Питон 3.х**

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

{1, 2, 3, 4}

>>> {1, 2, 3, 4} # Питон 3.х

{1, 2, 3, 4}

- эти два способа идентичны.

В Питон 3.х встроенная функция set всё еще необходима для создания пустых множеств и конструировании множеств на основе итерируемых объектов.

>>> set('spam')

{'p', 'm', 's', 'a'}

>>> s = {'s', 'p', 'a', 'm'}

>>> s.add('alot')

>>> s

{'alot', 'm', 'p', 'a', 's'}

Обратим внимание, что конструкция {} по-прежнему создает пустой словарь.

>>> s1 = {1, 2, 3, 4}

>>> s1 - {1, 2, 3, 4} # вывод пустого множества

set()

>>> type({}) # литерал {} обозначает пустой словарь

<class 'dict'>

>>> s = set() # инициализация пустого множества

>>> s.add(1.23)

>>> s

{1.23}

**Ограничения, связанные с неизменяемостью и фиксированные множества.**

Множества могут включать объекты только неизменяемфых (так называемый "хешируемых") типов.

Отсюда следует, что списки и словари не могут добавляться в множество, однако можно исполььзовать кортежи, если необходимо сохранить составные значения.

>>> s

{1.23}

>>> s.add([1, 2, 3])

TypeError: unhashable type: 'list'

>>> s.add({'a':1})

TypeError: unhashable type: 'dict'

>>> s.add((1, 2, 3))

>>> s

{1.23, (1, 2, 3)}

>>> s | {(4, 5, 6), (1, 2, 3)}

{1.23, (4, 5, 6), (1, 2, 3)}

>>> (1, 2, 3) in s

True

>>> (1, 4, 3) in s

False

Кортежи в множествах могут использоваться, например, для хранения дат, записей, IP-адресов и т.д.

Сами по себе множества нельзя добавлять в другие множества, но с помощью функции frozenset можно, она которая создает неизменяемое множество, которое невозможно изменить, и это дает возможность добавлять.

**Генераторы множеств.**

>>> {x \*\* 2 for x in [1, 2, 3, 4]}

{16, 1, 9, 4}

Мы видим, что генераторы множеств, как и в случае с генератором списков выполняет свою работу так - "Вернуть множество, содержащее квадраты значений х, для каждого х из списка".

>>> {x for x in 'spam'}

{'p', 'm', 's', 'a'}

>>> {c \* 4 for c in 'spam'}

{'ssss', 'pppp', 'aaaa', 'mmmm'}

>>> {c \* 4 for c in 'spamham'}

{'ssss', 'hhhh', 'pppp', 'aaaa', 'mmmm'}

>>> s = {c \* 4 for c in 'spam'}

>>> s | {'mmmm', 'xxxx'}

{'xxxx', 'mmmm', 'ssss', 'pppp', 'aaaa'}

>>> s & {'mmmm', 'xxxx'}

{'mmmm'}

**Где могут использоваться множества?**

Помимо математических вычислений множества могут использоваться в ряде прикладных областей.

Используя уникальность множеств можно использовать их для фильтрации повторов в других коллекциях.

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

>>> set(l)

{1, 2, 3, 4, 5}

>>> l = list(set(l))

>>> l

[1, 2, 3, 4, 5]

Можно также использовать множества для хранения пунктов, которые уже были посещены в процессе обхода графа или другой циклической структуры.

Множества удобно использовать при работе с большими массивами данных (например запросы к БД) - операция пересечения двух множеств позволит получить объекты, присутствующие сразу в обеих категориях, а объединение - все объекты, присутствующие в любом из множеств.

>>> engin = {'bob', 'sue', 'ann', 'vic'}

>>> manag = {'tom', 'sue'}

>>> 'bob' in engin # bob инженер?

True

>>> engin & manag # Кто одновременно инженер и менеджер?

{'sue'}

>>> engin | manag # Все сотрудники из обеих категорий

{'tom', 'vic', 'bob', 'ann', 'sue'}

>>> manag - engin # Менеджеры не являющиеся инженерами

{'tom'}

>>> engin - manag # Инженеры не являющиеся менеджерами

{'bob', 'ann', 'vic'}

>>> engin > manag # Все менеджеры являются инженерами? (надмножество)

False

>>> {'bob', 'sue'} < engin # Оба сотрудники инженеры? (подмножество)

True

>>> (manag | engin) > manag # Множество всех сотрудников является

True надмножеством менеджеров

>>> manag ^ engin # Сотрудники принадлежащие какой-то одной категории

{'bob', 'ann', 'tom', 'vic'}

>>> (manag | engin) - (manag ^ engin) # Пересечение!

{'sue'}

**Логические значения.**

Официально в Питон имеется логический тип bool, с двумя предопределенными значениями True и False. Эти значения являются экземплярами класса bool, который в свою очередь является всего лишь подклассом (в ООП смысле) целочисленного типа int.

True и False ведут себя как целочисленные значения 1 и 0, за исключением, что выводятся на экран они по-другому.

Нужно внимательно следить, т.к. из-за особенности этого типа можно получить интересные результаты:

>>> type(True)

<class 'bool'>

>>> isinstance(True, int)

True

>>> True == 1

True

>>> True is 1

False

>>> True or False # то же, что и 1 or 0

True

>>> True + 4 # а вот и обещанный прикол, получилось как-будто 1 + 4

5

**Числовые расширения.**

Поскольку числовая обработка данных является популярным применением Питон вы найдете массу дополнительных инструментов.

Напрмиер расширение - NumPy (Numeric Python). Это расширение нужно устанавливать отдельно, оно не идет с Питоном вместе.

**Глава 6. Интерлюдия о динамической типизации.** стр. 194

В этой главе мы детально ознакомимся с самой фундаментальной идеей языка Питон - динамической типизацией и полиморфизмом, которые составляют основу гибкости этого языка программирования.

Простым языком - полиморфизм, это когда одна функция (оператор, метод) могут по-разному выполняться для разных типов объектов, как например оператор + по-разному действует для чисел и строк.

**Остутствие инструкций объявления.**

Типы данных в Питон определяются автоматически во время выполнения, а не в результате объявления в программном коде (в отличие от С++ или Java).

**Переменные, объекты и ссылки.**

Отметим, что когда выполняется операция присваивания, типо a = 3, интерпретатор выполняет её, хотя перед этим ему нигде не сообщалось, что а - это имя переменной и тип - целочисленный объект.

В Питоне это происходит естественным способом:

- Создание переменной - переменная (т.е. имя), такая как a, создается автоматически когда в программном коде ей впервые присваивается значение.

Если переменная а уже есть то она просто получает новое значение

- Типы переменных - переменные не имеют никакой информации о типе или ограничениях, связанных с ним. Понятие типа присуще объектам, а не именам. Т.е. у нашей переменной "а" тип имеется лишь у объекта - 3 (целочисленный тип), а не у самой переменной "а".

Переменные универсальны, они являются лишь ссылками на конкретные объекты в конкретные моменты времени.

- Использование переменной - когда переменная учатсвует в выражении её имя замещается объектом на который она ссылается в настоящий момент.

Конечно перед использованием переменную нужно инициализировать, иначе будет вызвана ошибка.

Рассмотрим команду а = 3, она выполняется в 3 этапа:

- создается объект, представляющий число 3;

- создается переменная а, если она еще отсутствует;

- в переменную а записывается ссылка на вновь созданный объект, представляющий число 3.

***Имя - ссылка - объект***

Кстати объекты могут ссылаться на другие объекты, например список ссылается на объекты из своего списка.

А переменные не могут ссылаться на другие переменные, только на объекты.

Ссылки на объекты это, своего рода, ассоциативные связи, реализованные в виде указателя на область памяти.

*Переменные* - это записи в системной таблице, где предусмотрено место для хранения ссылок, на объекты.

*Объекты* - это области памяти с объемом, достаточным для представления значений этих объектов.

*Ссылки* - это автоматически разыменовываемые указатели на объекты.

С технической точки зрения объекты не так просты. Они имеют два поля: описатель типа - хранит информацию о типе объекта и счетчик ссылок - используемый для определения когда память, занимаемая объектом, может быть освобождена.

**Информация о типе хранится в объекте, но не в переменной.**

>>> a = 3 # тип - целое число

>>> a = 'spam' # тип - строка

>>> a = 1.23 # тип - вещественное число

- здесь у нас изменяется ссылка на объект, а не изменение типа переменной с числа на строку и обратно. Т.е. мы видим как переменная ссылается на конкретный объект в конкретный момент времени.

Итак, типы в Питон - это свойство объекта, а не переменной. Объект сам знает какого он типа, переменной эта информация не нужна.

**Объекты уничтожаются автоматически.**

>>> a = 3

>>> a = 'spam'

Каждый раз когда имя ассоциируется с новым объектом Питон освобождает память, занимаемую предыдущим объектом (если на него не ссылается какое-либо другое имя или объект).

Это называется - сбокрой мусора (garbage collection).

x = 42

x = 'spam'

x = 3.1415

x = [1, 2, 3]

- так, с условием, что на объекты не ссылаются другие переменные или объекты, с каждым новым присвоением х уничтожается старый объект.

Это происходит благодаря счетчику ссылок, с помощью которого интерпретатор следит за ссылками на объект. Как только счетчик равен 0 объект уничтожается.

**Разделяемые ссылки.** стр. 199

>>> a = 3

>>> b = a

В итоге получается такая схема - имена a, b ссылаются своиими ссылками на один и тот же объект.

Это называется - разделяемая ссылка, несколько имен ссылаются на один и тот же объект.

Если мы теперь изменим а, то ситуация будет следующей:

>>> a = 3

>>> b = a

>>> a = 'spam'

>>> b

3

Т.е. мы видим, что ссылка у переменной b осталась на объект - 3, а ссылка переменной а изменилась, она стала на объект - 'spam'.

Теперь оба имени ссылаются на разные объекты.

И наоборот - при изменении b ссылка a не изменилась бы, а как и раньше ссылалась бы на объект - 3.

>>> a = 5

>>> b = a

>>> a = a + 6

>>> b

5

- даже такая сложная схема по-прежнему иллюстрирует, что оба имени ссылаются на разные объекты в конце выражений.

**Разделяемые ссылки и изменяемые объекты.**

Не забываем, что в Питон существуют объекты и операции, которые изменяют сами объекты. Например изменения элемента списка изменяют сам список.

>>> l1 = [1, 2, 3]

>>> l2 = l1

- применим одну из операций к спискам:

>>> l1[0] = 24

>>> l1

[24, 2, 3]

>>> l2

[24, 2, 3]

- здесь мы изменяем, не сам объект l1, а компонент объекта, и в итоге изменяется сам объект и получаем также изменение в переменной l2!

! - Такое поведение объектов нужно запомнить и внимательно следить за разработкой ваших программ, ведь можно не заметить как изменение одного объекта может привести к изменению другого и вы получите результат, которого совсем не ожидали. Тщательно проверяйте весь код.

Избежать такого поведения языка Питон можно выполнять копирование объектов, вместо создания ссылок.

Это можно сделать с помощью функции list и модуля copy из стандартной библиотеки.

Однако самый стандартный способ копирования это получение среза от начала до конца списка:

>>> l1 = [2, 3, 4]

>>> l2 = l1[:]

>>> l1[0] = 24

>>> l1

[24, 3, 4]

>>> l2

[2, 3, 4]

Словари нужно копировать другим способом, т.к. они не являются последовательностями. Чтобы скопировать словарь необходимо воспользоваться методом X.copy().

>>> import copy

>>> x = copy.copy(y) # Создание "поверхностной" копии любого объекта y

>>> x = copy.deepcopy(y) # Создание полной копии: копируются все вложенные ссылки

- В число объектов, подверженных описанным выше изменениям входят такие объекты как - списки, словари и некоторые объекты, объявленные с помощью инструкции class.

**Разделяемые ссылки и равенство.**

>>> x = 42

>>> x = 'spam'

Т.к. интерпретатор Питона кэширует и повторно использует малые целые числа и небольшие строки, объект 42 скорее всего не будет уничтожен. Он останется в системной таблице для повторного использования (вдруг нам снова понадобится какой-нибудь объект равный 42).

Но это относится только к малому числу объектов, например к таким как - малые целые числа и небольшие строки.

**Равенства.**

>>> l = [1, 2, 3]

>>> m = l # ссылка на один и тот же объект

>>> l == m # Одно и то же значение

True

>>> l is m # Один и тот же объект

True

- т.е. оператор == сравнивает значения объектов, а оператор - is проверяет идентичность объектов, т.е. ссылаются ли оба имени на один и тот же объект.

Если точнее, то is сравнивает указатели, которые реализуют ссылки и, т.о., может использоваться для выявления разделяемых ссылок в программном коде.

Для проверки простого равенства достаточно == (он чаще и используется).

>>> l = [1 ,2 , 3]

>>> m = [1, 2, 3]

>>> l == m

True

>>> l is m # значения одинаковы, а объкты разные

False

- видим наглядную разницу операторов.

Еще один интересный пример:

>>> x = 42

>>> y = 42

>>> x == y

True

>>> x is y

True

- здесь мы видим как переплетаются операторы сравнения и особенности кэширования в Питон, в результате видим, что обе переменные ссылаются на один объект, хотя, теоретически, должны ссылаться на разные объекты.

Мы можем запрашивать у интерпретатора количество ссылок на объект. Функция getrefcount из модуля sys возвращает значения поля счетчика ссылок в объекте.

>>> import sys

>>> sys.getrefcount(x)

11

>>> sys.getrefcount(1)

1156

>>> sys.getrefcount(y)

11

>>> sys.getrefcount(42)

12

>>> sys.getrefcount(1)

1156

- большая часть ссылок была создана системным программным кодом самой IDLE (например число 1).

Т.к. строки и числа не могут изменяться совершенно не важно сколько ссылок будет указывать на один и тот же объект, главное нужно запомнить, что в Питоне есть автоматическая сборка мусора и она реально работает.

**Глава 7. Строки.** стр. 207

Строки - упорядочненные последовательности символов, используемые для хранени и представления текстовой информации.

С функциональной точки зрения строки могут использоваться для представления всего, что может быть выражено в текстовой форме: символы и слова, содержимое текстовых файлов, загруженных в память, адреса в Интернете, программы на Питоне и т.п. И также для хранения двоичных значений байтов и символов Юникода.

Строки в Питоне играют ту же роль, что и массивы символов в С.

Фактически, строки - это первый представитель большого класса объектов, называемых последовательностями.

Строки неизменяемы, т.е. символы, которые они содержат, имеют определенный порядок следования слева направо и сами строки невозможно изменить.

**Таблица "Наиболее типичные литералы строк и операции".** стр. 208

**Литералы строк.**

Строки в апострофах - 'spa"m'

Строки в кавычках - "spa'm"

Строки в тройных кавычках - '''...spam...''', """...spam..."""

Экранированные последовательности - "s\tp\na\0m"

Неформатированные строки - r"C:\new\test.spm"

Строки байтов в версии 3.0 - b'sp\x01am'

**Строки в апострофах и кавычках – это одно и то же.**

>>> 'spam', "spam" # строки в кавычках и апострофах идентичны

('spam', 'spam')

Такая транскрипция написания строк нужна для того, чтобы вставлять символы апострофа и кавычек в строки без использования слеша:

>>> "knight's", 'knight"s' # вот такой простой пример использования обоих способов

("knight's", 'knight"s')

- можно и простой слеш использовать:

>>> "knight\"s", 'knight\'s' # при этом меняется строка вывода, заметьте

('knight"s', "knight's") # в выводе в строке нет набора символов одного типа

Питон автоматически объединяет последовательность строковых литералов внутри выражения, хотя можно бы и добавить оператор + но Питон автоматически проводит конкатенацию строк:

>>> title = "Peace " 'of' " sheet"

>>> title

'Peace of sheet'

- а если поставим запятые, то получим кортеж, а не строку:

>>> title = "Peace ", 'of', " sheet"

>>> title

('Peace ', 'of', ' sheet')

**Экранированные последовательности представляют служебные символы.**

С помощью слеша вставляются специальные символы, известные как - экранированные последовательности.

Экранированные последовательности удобно использовать для вставки в строки служебных символов.

>>> s = 'a\nb\tc'

\n - байт, содержащий двоичное значение кода символа новой строки

\t - символ табуляции

>>> s

'a\nb\t\c'

>>> print(s)

a

b c

>>> len(s)

5

- длина этой строки 5 байтов, хотя в выводе функции print мы видим всего 3 символа.

**Таблица "Экранированные последовательности".** стр. 212

>>> s = 'a\0b\0c' # в этой строке два нулевых байта

>>> s

'a\x00b\x00c'

>>> print(s)

a'null'b'null'c

>>> len(s)

5

В Питоне нулевой байт (символ null) не является признаком завершения строки, как в языке С.

Еще пример, запишем строку, в которой указаны двоичные значения 1 и 2 (записанные в восьмеричной форме), и далее следует двоичное значение 3 (записанное в шестнадцатеричной форме).

>>> s = '\001\002\x03'

>>> s

'\x01\x02\x03'

- мы видим, что интерпретатор Питон отображает непечатаемые символы в шестнадцатеричном представлении, независимо от того, в каком виде они были указаны внутри литерала.

>>> s = "s\tp\na\x00m"

>>> s

's\tp\na\x00m' # \x00 - это нулевой символ в шестнадцатеричном представлении

>>> len(s)

7

>>> print(s)

s p

a m

Если интерпретор не понимает символ после слеша как корректный служебный символ, он просто соатвляет символ обратного слеша в строке:

>>> x = "C:\py\code"

>>> x

'C:\\py\\code'

>>> print(x)

C:\py\code

Еще одно замечание, если нам требуется работать с двоичными данными, то открывать файлы с ними нужно в режиме двоичного доступа (добавляя к флагу режима открытия флаг b, например - 'rb', 'wb' b т.д.).

В Питоне 3.0 содержимое двоичных файлов интерпретируется как коллекция строк типа bytes, которые по своим возможностям напоминают обычные строки.

**Неформатированные строки подавляют экранирование.**

Обратим внимание на команду, с помощью которой мы хотим открыть файл text.txt из папки C:\new -

mytest = open('C:\new\text.txt', 'w')

- здесь мы видим в чем будет заключаться подводные камни: последовательность \n интерпретируется как символ новой строки, а последовательность \t замещается символом табуляции, и в итоге мы получим ошибку:

>>> mytest = open('C:\new\text.txt', 'w')

Traceback (most recent call last):

File "<pyshell#9>", line 1, in <module>

mytest = open('C:\new\text.txt', 'w')

OSError: [Errno 22] Invalid argument: 'C:\new\text.txt'

Именно в таких случаях удобно использовать неформатированные строки, для этого перед кавычками ставим символ r (в верхнем или нижнем регистре), он отключает механизм экранирования.

>>> mytest = open(r'C:\new\text.txt', 'w')

>>> mytest

<\_io.TextIOWrapper name='C:\\new\\text.txt' mode='w' encoding='cp1251'>

Ну или, конечно, можно просто продублировать символы слеша:

>>> mytest = open('C:\\new\\text.txt', 'w')

Сам интерпретатор Питон, в определенных случаях, использует удваивание обратного слеша при выводе строк, обратный слеш:

>>> path = r'C:\new\text.txt'

>>> path

'C:\\new\\text.txt' # вот так интерпретатор представляет эту строку

>>> print(path)

C:\new\text.txt # более дружественный формат представленя

>>> len(path)

15

Так же как и с числами, при выводе результатов в интерактивной оболочке по умолчанию используется такой формат представления, как если бы это был программный код, отсюда и экранирование символов обратного слеша.

А инструкция print обеспечивает более дружественный формат.

Помимо хранения имен каталогов в Windows, неформатированные строки обычно используются для регулярных выражений (поиск по шаблону, благодаря модулю re).

Следует отметить, что в сценариях Питона, в строках с именами каталогов систем Windows и Unix можно использовать простые символы слеша, потому что Питон стремится поддерживать переносимость для путей к файлам.

И все же для каталогов в стиле Windows удобнее использовать неформатированные строки.

***Замечание.***

Даже неформатированная строка не должна заканчивать символом обратного слеша, иначе он заэкранирует собой закрывающую кавычку:

r"...\" - недопустимый строковый литерал!

Если такая необходимость есть, можно добавить два символа и затем удалить второй из них (r'1\nb\tc\\'[:-1]), добавить один символ вручную (r'1\nb\tc'+'\\') или использовать обычный синтаксис и дублировать все символы обратного слеша ('1\\nb\\tc\\')

**Тройные кавычки, многострочные блоки текста.**

Этот формат иногда называют блочной строкой, который удобно использовать для определения многострочных блоков текста в программном коде.

Синтаксис состоит из трех символов кавыек (или апострофов), которые закрываются такими же тремя кавычками.

Внутри такой строки могут находиться и кавычки и апострофы, при этом их не нужно экранировать.

Примеры:

>>> mantra = """Always look

on the dright

side of life."""

>>> mantra

'Always look\n on the dright\n side of life.'

>>> print(mantra)

Always look

on the dright

side of life.

>>> man = """One

two

three."""

>>> man

'One\n two\n three.'

>>> man = """One # здесь я сознательно после перевода строки удалил пробел перед three

two

three.""" # оказывается, после перевода строки, в таком формате,

автоматически ставится пробел перед след. строкой

>>> man

'One\n two\nthree.' # как мы и видим, пробела нет, и далее print нам всё покажет еще раз

>>> print(man)

One

two

three.

>>> mantra = """ Always look

on the dright

side of life.""" # вот я перед словом Always поставил пробел и все строки

встали на одном уровне, как видно после вывода инструкции print

>>> print(mantra)

Always look

on the dright

side of life.

Таким нехитрым способом мы можем пользоваться если нам необходимо ввести в программе многострочный текст, например сообщение об ошибке или код разметки на HTML или XML.

Еще одно применение тройных кавычек, это хакерский способ отключения строк программного кода во время разработки.

Например, если нам требуется отключить большой участок кода это довольно неудобно ведь в таком случае в сотне строк кода нужно ставить знак комментариев в каждой строке.

Но можно воспользоваться такой "хитростью":

x = 1

"""

import os

print(os.getcwd())

"""

y = 2

**Строки в действии.** стр. 217

**Базовые операции.**

>>> len('abc') # Длина: число элементов

3

>>> 'abc' + 'def' # Конкатенация (сложение строк): новая строка

'abcdef'

>>> 'NIL!' \* 4 # Повторение: подобно 'NIL!'+'NIL!'+'NIL!'

'NIL!NIL!NIL!NIL!'

>>> print('--------------------') # пример удобства базовых операций, на примере повторения

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

>>> print('-' \* 20)

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

- Обратим внимание, здесь происходит перегрузка операторов, + и \*, которые используются для операций сложения и умножения с числами.

**! -** Но складывать строки и числа **нельзя**! Это приведет к ошибке.

Выполним обход элементов строки в цикле, используя инструкцию for, и проверим вхождение подстроки в строку с помощью оператора выражений in, который, по сути, выполняет операцию поиска.

В случае поиска подстрок, оператор in напоминает метод str.find(), который мы рассмотрим далее, но in возвращает логический результат, а не позицию подстроки в строке:

>>> myjob = "hacker"

>>> for c in myjob: print(c, end=' ') # символ : обязателен, без него выдает ошибку

хотя вся команда в одну строку

h a c k e r

>>> for c in myjob: print(c, end='\\')

h\a\c\k\e\r\

>>> 'k' in myjob

True

>>> 'z' in myjob

False

>>> 'spam' in 'abcspamdef'

True

- Оператор цикла for присваивает переменной очередной элемент последовательности (в нашем случае - строки) и для каждого элемента выполняет одну или более инструкций.

В результате переменная 'c' превращается в своего рода курсор, который постепенно перемещается по строке.

Далее в книге мы более подробнее рассмотрим средства выполнения итераций.

**Доступ по индексам и извлечение подстроки.**

Т.к. строки это у нас упорядоченные коллекции символов, то обращаться к элементам строки можно по номерам позиций.

Индексирование - операция, с указанием числового смещения в квадратных скобках после имени строки.

**рис. 7.1 Смещения и срезы. Наглядно демонстрируется индексация.** стр. 219

Индексирование в Питон начинается с 0. Т.е. первый символ имеет индекс 0, второй 1 и т.д. Индексация с конца начинается с -1 (т.к. отрицательного 0 мы не знаем), в принципе всё это понятно и по рисунку 7.1.

>>> s = 'spam'

>>> s[0], s[-2]

('s', 'a')

- s[0] возвращает нам элемент со смещением 0, это, естественно первый символ,

- s[-2] возвращает второй символ с конца, это эквивалентно записи (4+(-2)) от начала.

>>> s[1:3], s[1:], s[:-1]

('pa', 'pam', 'spa')

- эти операции демонстрируют нам операции - извлечения среза, в некотором роде это форма синтаксического анализа. В случае среза Питон возвращает новый объект, содержащий непрерывную область, определяемую парой смещений.

Левая граница смещения (нижняя) - включительна, а правая (верхняя) - не входит в срез.

Если левая или правая или обе границы опущены, по умолчанию принимаются значения раные 0 (см. пример выше).

***Краткая справка.***

Операция индексирования (s[i]) извлекает компоненты по их смещениям:

- Первый элемент имеет смещение 0.

- Отрицательные индексы определяют смещение в обратном порядке.

- s[0] извлекает первый элемент.

- s[-2] извлекает второй с конца элемент (так же, как и выражение s[len(s)-2]).

Операция извлечения подстроки (s[i:j]) извлекает непрерывный раздел последовательности:

- Элемент с индексом, равным верхней границе, не включается в срез.

- Если границы не указаны, по умолчанию они берутся равными 0 (по одной, или обе сразу, не важно).

- s[1:3] извлекает элементы со смещением от 1 до 3 (не включая элемент со смещением 3).

- s[1:] извлекает элементы, начиная со смещения 1 и до конца.

- s[:3] извлекает элементы, начиная со смещения 0 и до 3 (не включая его).

- s[:] извлекает элементы, начиная со смещения 0 и до конца - это эффективный способ создать поверхностную копию последовательности s.

Последний пункт списка - это самый обычный трюк, с его помщью создается полная поверхностная копия объекта, т.е. объект с тем же значением, но расположенный в другой области памяти.

Этот прием не очень полезен с неизменяемыми объектами (строки), но его ценность возрастает при работе с изменяемми объектами (списки).

**Расширенная операция извлечения подстроки: третий предел.**

Вообще, в операции среза имеется еще и третий, необязательный, индекс. Т.е. полная форма записи выглядит как - s[I:J:K], где К - это шаг среза. По умолчанию шаг равен 1, поэтмоу при обычной операции извлечения мы его не указываем.

Например выражение Х[1:10:2] вернет каждый второй элемент последовательности Х в диапазоне смещений от 1 до 9 (т.е. 1, 3, 5, 7, 9).

>>> s = 'abcdefghijklmnop'

>>> s[1:10:2]

'bdfhj'

>>> s[::2]

'acegikmo'

Можно также использовать отрицательное значение шага, например:

>>> s = 'hello'

>>> s[::-1]

'olleh'

- в результате получаем перевернутую последовательность.

Выражение s[5:1:-1] извлечет элемент со 2 по 5 в обратном порядке (5, 4, 3, 2):

>>> s = 'abcdefg'

>>> s[5:1:-1]

'fedc'

Далее в книге мы еще встретимся с срезами с тремя пределами когда будем изучать цикл for.

Также познакомимся с тем, что операция извлечения среза эквивалентна операции индексирования, в которй в качестве индекса используется специальный объект среза, что может быть важно для разработчиков классов, которым требуется реализовать поддержку обеих операций:

>>> 'spam'[1:3] # Синтаксис извлечения среза

'pa'

>>> 'spam'[slice(1, 3)] # Использование объекта среза

'pa'

>>> 'spam'[::-1]

'maps'

>>> 'spam'[slice(None, None, -1)]

'maps'

**Примеры для изучения.**

Ознакомимся с примером, в котором аргументы командной строки, переданные сценарию Питон при запуске, доступны в виде атрибута argv встроенного модуля sys:

#File echo.py

import sys

print(sys.argv)

% echo.py -a -b -c

['C:\\Python34\\myfile.py', '-a', '-b', '-c']

Из этого вывода, с помощью операции среза, мы можем получить список аргументов, без имени программы, делается это все с помощью небольшого изменения:

#File echo.py

import sys

print(sys.argv[1:])

% echo.py -a -b -c # простое решение, если нам нужно получить только аргументы из консоли

['-a', '-b', '-c']

Также операция среза позволяет нам удалять лишние элементы из строк, например перевод строки можно удалить срезом line[:-1] хотя предпочтительнее использовать метод line.rstrip

**Инструменты преобразования строк.** стр. 223

Один из девизов Питона - не поддаваться искушению делать предположения о том, что имелось в виду.

Например случай когда Питон не дает сложить строку с числом, даже если строка выглядит как число:

'42' + 1 # будет сообщение об ошибке

Т.к. в этом случае суть оператора + не ясна для интерпретатора (сложение или конкатенация), он воспринимает такую инструкцию как ошибочную.

Вообще в Питоне отвергается любая магия, которая может осложнить жизнь программиста.

В подобных случаях можно использовать инструменты преобразования, например:

>>> int("42"), str(42) # преобразование из/в строку

(42, '42')

>>> repr(42) # преобразование в строку, как если бы она была лдитералом в программном коде

'42'

- функция repr преобразует объект в строковое представление, но возвращает объект в виде строки программного кода, который можно выполнить, чтобы воссоздать объект.

>>> print(str('spam'), repr('spam'))

spam 'spam'

>>> print('spam') # добавил это выражение сам, чтобы яснее было выражение выше

spam

Итак, мы можем вручную выполнять необходимые преобразования операндов перед операциями:

>>> s = '42'

>>> i = 1

>>> s + i

...

TypeError: Can't convert 'int' object to str implicitly

>>>

>>> int(s) + i # выполняем операцию сложения

43

>>> s + str(i) # выполняем операцию конкатенации

'421'

И с вещественными числами:

>>> str(3.1415), float("1.5")

('3.1415', 1.5)

>>>

>>> text = "1.234E-10"

>>> float(text)

1.234e-10

Далее в книге мы познакомимся со встроенной функцией eval - она выполняет строку, содержащую программный код на Питоне, и поэтому может выполнять преобразование строки в объект любого вида.

Функции int и float преобразуют только числа, но они и работают быстрее и безопаснее, потому что не принимают программный код произвольного выражения.

**Преобразование кодов символов.**

Имеется также возможность выполнить преобразование одиночного символа в его целочисленный код ASCII, для чего нужно передать этот символ функции ord - она возвращает фактическое числовое значение соотвествующего байта в памяти.

Обратное преобразование выполняется функцией chr, она получает код ASCII и преобразует его в символ:

>>> ord('s')

115

>>> chr(115)

's'

Пример некоторой символьной математики:

>>> s = '5'

>>> s = chr(ord(s) +1)

>>> s

'6'

>>> s = chr(ord(s)+1)

>>> s

'7'

Пример преобразования альтернативный встроенной функции int:

>>> int('5')

5

>>> ord('5') - ord('0')

5

Такие преобразования могут использоваться с операторами цикла, для получения целочисленных значений из строковых представлений двоичных чисел.

В каждой итерации текущее значение умножается на 2 и затем к нему прибавляется числовое значение следующей цифры:

>>> b = '1101' # двоичные цифры преобразуются в числа с помощью функции ord

>>> i = 0

>>> while b != '':

i = i\*2 +(ord(b[0]) - ord('0'))

b = b[1:]

>>> i

13

Операция побитового сдвига влево (i << 1) могла бы дать тот же эффект, что и операция умножения на 2. А также с помощью функций int и bin:

>>> int('1101', 2)

13

>>> bin(13)

'0b1101'

**Изменение строк.** стр. 225

Итак, строки - неизменяемая последовательность. Неизменяемая означает, что мы не можем изменить элемент строки, выполнив присваиваниме по индексу:

>>> s = 'spam'

>>> s[0] = "x"

...

TypeError: 'str' object does not support item assignment

Чтобы изменить строку, нужно создать новую (с помощью операций извлечения подстроки и конкатенации) и, при необходимости, присвоить результат первоначальному имени:

>>> s = s + 'SPAM' # можно назвать как "изменение" первоначальной строки

>>> s

'spamSPAM'

>>> s = s[:4] + 'Burger' + '!' # здесь мы замещаем символы таким нехитрым способом

>>> s

'spamBurger!'

- замешение можно провести методом replace:

>>> s = 'splot'

>>> s = s.replace('pl', 'pamal')

>>> s

'spamalot'

Т.о. мы видим, что строковые методы созадют новые строковые объекты, при этом мы можем присвоить их переменной (хоть новой хоть старой переменной).

К тому же создание нового объекта для каждого изменения это не столь неэффектвиная операция как может показаться, на самом деле при создании новых объектов они используют те же области памяти, что и прежние.

Интерпретатор Питон работает намного быстрее, чем можно было бы ожидать.

Ниже приведем два примера подстановки значений объектов в строку (при этом объекты преобразуются в строку) при помощи т.н. спецификаторов формата:

>>> 'That is %d %s bird!' % (1, 'dead') # Выражение форматирования

'That is 1 dead bird!'

>>> 'That is {0} {1} bird!'.format(1, 'dead') # Метод форматирования, применяем сразу к строке

'That is 1 dead bird!'

- Замечание! Во втором примере я попытался вместо 0 и 1 вставить 12 и 13 и интерпретатор выдал ошибку.

Насколько я понимаю, 0 и 1 это индексированные номера параметров передаваемых методом - т.е. 0 это индекс параметра 1 (из метода format), а 1 - это индекс параметра 'dead' (из метода format).

Далее мы сначала познакомимся с методами строк, а затем уже перейдем к приемам форматирования строк.

*Примечание:*

- В Питоне 3.0 появился новый строковый тип bytearray, который относится к категории изменяемых. В действительности же, объекты типа bytearray не являются строками - это последовательности 8-битных целых чисел. При отображении они выводятся как строки символов ASCII. Объекты этого типа обеспечивают возможность хранения больших объемов текста, который требуется изменять достаточно часто.

И далее мы увидим, что ord и chr могут работать с символами Юникода.

**Строковые методы.** стр. 227

*Методы* - это просто функции, которые связаны с определенными объектами. Формально они являются атрибутами, присоединенными к объектам, которые ссылаются на функции.

В Питоне выражения и встроенные функции могут работать с наборами типов, но методы - являются специфичными для типов объектов (т.е. строковые методы работают только с строками и т.п.).

В Питоне 3.0 некоторые методы могут относиться к разным типам (напр. count), но при этом функциональность такого метода может быть разной для разных типов.

*Функции* - это пакеты программного кода, а вызовы методов объединяют в себе выполнение двух операций (извлечение атрибута и вызов функций).

*Извлечение атрибута* - выражение вида object.attribute означает: "Извлечь значение атрибута attribute из объекта object".

*Вызов функции* - выражение вида function(arguments) означает: "Вызвать программный код функции function, передав ему ноль или более объектов-аргументов arguments, разделенных запятыми, и вернуть значение функции".

Объединение этих двух действий позволяет вызвать метод объекта.

Выражение вызова метода object.method(arguments) вычисляется слева направо, т.е. интерпретатор сначала извлекает метод объекта, а затем вызывает его, передавая ему входные аргументы.

Если метод возвращает какой-либо результат, он становится результатом всего выражения вызова метода.

Как мы увидим далее большинство объектов обладает методами, которые можно вызвать, и все они доступны с использованием одного и того же синтаксиса вызова метода.

**Таблица. 7.3 Строковые методы.** стр. 228

**Примеры использования строковых методов: изменение строк.** стр. 229

Изменим два символа в середине строки:

>>> s = 'spammy'

>>> s = s[:3] + 'xx' + s[5:]

>>> s

'spaxxy'

Также можно воспользоваться методом replace (в этом случае даже логичнее это метод):

>>> s = 'spammy'

>>> s = s.replace('mm', 'xx')

>>> s

'spaxxy'

Метод repalce является более универсальным чем кажется. Он принимает в качестве аргумента оригинальную подстроку (любой длины) строку замены (также любой длины), и выполняет глобальный поиск с заменой:

>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')

'aaSPAMbbSPAMccSPAMdd'

- видм, что в подобном случае метод replace может использоваться как инструмент реализации поиска с заменой по шаблону.

Если нужно заменить подстроку фиксир. размера, которая может быть в любом месте, можно также выполнить операцию замены или найти подстроку с помощью метода find:

>>> s = 'xxxxSPAMxxxxSPAMxxxx'

>>> where = s.find('SPAM') # поиск позиции

>>> where # подстрока найдена со смещением 4

4

>>> s = s[:where] + 'EGGS' + s[(where+4):]

>>> s

'xxxxEGGSxxxxSPAMxxxx'

- если подстрока не найдена метод find возвращает -1

Еще один вариант сипользования метода replace заключается в передаче третьего аргумента, который определяет число производимых замен:

>>> s = 'xxxxSPAMxxxxSPAMxxxx'

>>> s.replace('SPAM', 'EGGS') # заменить все найденные подстроки

'xxxxEGGSxxxxEGGSxxxx'

>>>

>>> s.replace('SPAM', 'EGGS', 1) # заменить одну подстроку

'xxxxEGGSxxxxSPAMxxxx'

- заметьте! Несмотря на то, что метод называется replace (заменить), в действительности он создает новую строку, т.е. методы, на самом деле, никогда не изменяют строки, даже если у них и громкое название как у replace.

Такое создание новых строк может оказаться недостатком использования методов. Если в сценарии много изменений длинных строк (при этом ведь создается такое же количество новых строк), можно повысить производительность сценария, преобразовав строку в изменяемый объект:

>>> s = 'spammy'

>>> l = list(s)

>>> l

['s', 'p', 'a', 'm', 'm', 'y']

Встроенная функция list (или функция-конструктор объекта) создает новый список из элементов любой послед-ти.

>>> l[3] = 'x'

>>> l[4] = 'x'

>>> l

['s', 'p', 'a', 'x', 'x', 'y']

И теперь можно преобразовать обратно:

>>> s

'spammy'

>>> s = ''.join(l)

>>> s

'spaxxy'

join - является строк. методом (не списка), вызывается с указанием строки-разделителя (в нашем случае - пустой строки).

Можно использовать произвольную строку-разделитель:

>>> 'SPAM'.join(['aaa', 'bbb', 'ccc']) # это пример из книги

'aaaSPAMbbbSPAMccc'

- теперь покажу пример , который я сам решил попробовать:

>>> 'SPAM'.join([l])

...

TypeError: sequence item 0: expected str instance, list found

>>>

>>> l

['s', 'p', 'a', 'x', 'x', 'y']

>>> a = 'SPAM'.join(l)

>>> a

'sSPAMpSPAMaSPAMxSPAMxSPAMy'

- видим, что просто так не получилось со списком, пришлось присвоить значение выражения переменной. Нужно это запомнить и обдумать в дальнейшем!

Строковый тип bytearray (благодаря, что объекты этого типа изменяются непосредственно) представляет отличную альтернативу комбинации функций list/join, особенно когда текст приходится часто менять.

**Примеры методов строк: разбор текста.**

Разбор текста - анализ структуры и извлечение подстрок.

>>> line = 'aaa bbb ccc'

>>> col1 = line[:3]

>>> col3 = line[8:]

>>> col1, col3

('aaa', 'ccc')

Конечно этот прием может использоваться если известны позиции искомых подстрок.

Пример избавления от разделителя подстрок (например, пробел - у нас):

>>> line = 'aaa bbb ccc'

>>> cols = line.split()

>>> cols

['aaa', 'bbb', 'ccc']

- мы не указали разделитель потомоу что по-умолчанию, в качестве строки-разделителя, использ. пробельные символы - символы пробела, табуляция или перевод строки.

>>> line = 'aaa\tbbb\tccc'

>>> cols = line.split()

>>> cols

['aaa', 'bbb', 'ccc']

Еще пример разделителей:

>>> line = 'bob,hacker,40'

>>> cols = line.split(',')

>>> cols

['bob', 'hacker', '40']

>>> line = "I'mSPAMBobhackerSPAM40"

>>> cols = line.split('SPAM')

>>> cols

["I'm", 'Bobhacker', '40']

**Другие часто используемые методы строк в действии.** стр. 232

Например, есть методы - удаляют символ пробела в конце строки, преобразуют регистр символов, проверяют характер содержимого строки и наличие подстроки вначале или в конце и т.п.:

>>> line = "The khights who say Ni!\n"

>>> line.rstrip()

'The khights who say Ni!'

>>> line.upper()

'THE KHIGHTS WHO SAY NI!\n'

>>> line.isalpha()

False

>>> line.endswith('Ni!\n')

True

>>> line.startswith('The')

True

А есть и альтернативные пути:

>>> line = "The khights who say Ni!\n"

>>> line.find('Ni') != -1 # Поиск с использованием вызова метода или выражения

True

>>> line.find('Ni')

20

>>> 'Ni' in line

True

>>> sub = 'Ni!\n'

>>> line.endswith(sub) # Проверка наличия подстроки в конце строки с пом. метода или среза

True

>>> line[-len(sub):] == sub

True

Нужно обратить внимание, что строк. методы не поддерживают - шаблоны. Для обработи текста с использ. шаблонов нужно использ. моджль re. Несмотря на это ограничение, строковые методы иногда бывают эффективнее чем функции модуля re.

**Оригинальный модуль string (был удален из Питон 3.0)**

Ранее, в Питон 2.6

Пусть переменная х ссылается на строк. объект, тогда вызов метода объекта будет таким:

x.method(arguments)

что эквивалентно вызову аналогичной функции из модуля string:

string.method(x, arguments)

Пример:

s = 'a + b + c'

x = s.replace('+', 'spam')

x

'aspambspamc'

и эта же операция с помощью модуля string:

import string

y = string.replace(s, '+', 'spam')

y

'aspambspamc'

**Выражения форматирования строк.**  стр. 234

Операции форматирования строк позволяют подставить в строки значения различных типов за одно действие. Их удобно использовать, например, когда требуется отформатироват ьтекст для вывода на экран.

Есть два способа форматирования строк:

- *Выражения форматирования строк* - этот способ основан на модели функции printf из языка С;

- *Метод форматирования строк* - более уникальный для Питона, возможности которого в знач. степени пересекаются с возможностями выражений форматирования.

В языке Питон имеется двухместный оператор % предназначенный для работы со строками, он обеспечивает простой способ форматирования значений, согласно заданной строке формата.

Оператор % обеспечивает компактную запись кода с множественной подстановкой.

Порядок форматирования такой:

1. Слева от оператора % указать строку формата, с одним или более спецификаторов формата, каждый из которых начинается с символа % (пр. %d);

2. Справа от оператора % указать объект (или объекты, в виде кортежа), знач. которого должно быть подставлено на место спецификатора (или спецификаторов) в левой части выражения.

Вспомним пример выше и применим новые знания:

>>> 'That is %d %s bird!' % (1, 'dead') # выражение форматирования

'That is 1 dead bird!'

>>> 'That is %s %d bird!' % (1, 'dead') # также видим кортеж справа, т.к. неск-ко знач.

...

TypeError: %d format: a number is required, not str

>>> 'That is %s %d bird!' % ('dead', 1)

'That is dead 1 bird!'

>>> 'That is %s %s bird!' % ('dead', 1)

'That is dead 1 bird!'

Операции форматирования имеют мощные возможности, рассмотрим еще примеры:

>>> ex = "Ni"

>>> "The knights who say %s!" % ex

'The knights who say Ni!'

>>>

>>> "%d %s %d you" % (1, 'spam', 4)

'1 spam 4 you'

>>>

>>> "%s -- %s -- %s" % (42, 3.14159, [1,2,3])

'42 -- 3.14159 -- [1, 2, 3]'

Объекты любого типа могут быть преобразованы в строку (напр. пр ивыводе на экран это происходит), поэтому для любого объекта может быть указан спецификатор %s.

Также запомните - выражение форматирования создает новую строку.

**Таблица 7.4 Спецификаторы формата.** стр. 236

Спецификаторы в левой части поддерживают целый набор операций, с сложным синтаксисом.

Синтаксис использования спецификатора формата:

%[(name)][flags][width][.precision]code

Символ спецификатора формата (code) из табл. 7.4 располагается в самом конце.

Между символом % и символом спецификатора можно добавлять след. информацию: ключ в словаре (name); список флагов (flags), могут определять, например, выравнивание (-), знак числа (+); ширину поля (width) и число знаков после десятичной точки (precision) и мн. др.

width и precision могут принимать значения \*, чтобы показать, что их значения должны извлекаться из след. элемента в списке входных значений.

Пример форматирования целого числа с параметрами по умолчанию, а затем целое число выведем в поле шириной 6 символов, выравнивание по левому краю и с дополнением ведущими нулями:

>>> x = 1234

>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)

>>> res

'integers: ...1234...1234 ...001234'

Спецификаторы e, f, g отображают вещественные числа:

>>> x = 1.23456789

>>> x

1.23456789

>>> '%e | %f | %g' % (x, x, x)

'1.234568e+00 | 1.234568 | 1.23457'

Для вещественных чисел можно доп. указать: выравнивание по левому краю, дополнение вед. нулями, знак числа, ширину поля и число знаков после десятичной точки.

Для простых задач можно использовать функцию str:

>>> '%-6.2f | %05.2f | %+06f' % (x, x, x)

'1.23 | 01.23 | +1.234568'

>>> "%s" % x, str(x)

('1.23456789', '1.23456789')

Если ширина поля и количество знаков после точки не известны, можно указать их при выполнении, а в строке формата использовать символ \*, чтобы указать интерпретатору, что эти значения должны извлекаться из очередного элемента в списке входных значений, справа от оператора %.

Пример, число 4 в кортеже определяет кол-во знаков после точки:

>>> '%f, %.2f, %.\*f' % (1/3.0, 1/3.0, 4, 1/3.0)

'0.333333, 0.33, 0.3333'

**Форматирование строк из словаря**. стр. 238.

В спецификаторах формата можно такжде использовать ключи словаря, который будет указываться в правой части:

>>> "%(n)d %(x)s" % {'n':1, 'x':'spam'}

'1 spam'

Такое извлечение значения по ключу (с спецификаторами) часто используется в программах, создающих код разметки HTML или XML.

Рассмотрим интересный способ, в котором мы отдельн оподготовим строку для форматирования и отдельно блок для форматирования, который указывается справа от оператора % (в данном случае это будет словарь):

>>> reply = """

Greetings...

Hello %(name)s!

Your age squared is %(age)s

"""

>>> values = {'name': 'Bob', 'age': 40} # Подготовка фактических значений

>>> print(reply % values) # Подстановка значений

Greetings...

Hello Bob!

Your age squared is 40

>>>

- Необычный ход, не знал, что так тоже можно. Как видим - способ интересный, что-то уже новое для нас. Можно использовать для поиска и вывода шаблонов, или подстановки, очень удобно.

Встр. функция vars() - возвращает словарь, содержащий все переменные, существующие на момент её вызова:

>>> food = 'spam'

>>> age = 400

>>> vars()

{'\_\_loader\_\_': <class '\_frozen\_importlib.BuiltinImporter'>, 'food': 'spam', 'reply': '\nGreetings...\nHello %(name)s!\nYour age squared is %(age)s\n', '\_\_doc\_\_': None, 'age': 400, 'values': {'age': 40, 'name': 'Bob'}, '\_\_name\_\_': '\_\_main\_\_', '\_\_package\_\_': None, '\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_spec\_\_': None}

- vars() выдает нам, в выводе, словарь с переменными и их значениями; если внимательно поискать можем найти и наши ранние переменные (reply, values) и новые (food, age = 400) и др.

Если задействовать vars() в правой части оператора форматирования, можно отформатировать значения, по именам переменных (т.е. по ключам словаря):

>>> "%(age)d %(food)s" % vars()

'400 spam'

**Метод форматирования строк.** стр. 239.

В Питон 2.6 и 3.0 появился новый способ форматирования строк, более близкий по духу к Питону. Новый способ, по-прежнему, опирается на некоторые концепции функции printf языка С и имеет более подробный и явный синтаксис.

Оба способа форматирования значительно пересекаются по возможностям и вследствии этого программистам трудно выбрать какой использовать. Вообще - используйте какой понравится по синтаксису.

**Основы.**

Метод format использует строку, относительно которой он вызывается, как шаблон и принимает произвольное количество аргументов, представляющих значения для подстановки.

Спецификаторы и их параметры окружаются фигурными скобками {}, они могут определять порядковый номер {1} или имена именованных аргументов {food}.

Также аргументы могут передаваться функциям и методам в виде позиционных и именованных аргументов.

>>> templ = '{0}, {1} and {2}' # используем порядковые номера аргументов

>>> templ.format('spam', 'ham', 'eggs')

'spam, ham and eggs'

>>> templ = '{0}, {2} and {1}' # меняем местами номера аргументов, всё работает

>>> templ.format('spam', 'ham', 'eggs')

'spam, eggs and ham'

>>> templ = '{motto}, {pork}, and {food}' # Имена именованных аргументов

>>> templ.format(motto = 'spam', pork = 'ham', food = 'eggs')

'spam, ham, and eggs'

>>> templ = '{motto}, {0}, and {food}' # комбинация вариантов спецификаторов

>>> templ.format('ham', motto = 'spam', food = 'eggs')

'spam, ham, and eggs'

>>> '{motto}, {0}, and {food}'.format('ham', motto = 'spam', food = 'eggs')

'spam, ham, and eggs'

- метод можно вызывать прямо к объекту, не обязательно сначала строку присваивать переменной.

>>> '{motto}, {0}, and {food}'.format(42, motto = 3.14, food = [1, 2])

'3.14, 42, and [1, 2]'

- format также предоставляет возможность подстановки значений объектов любых типов.

Метод format создает и возвращает новый объект строки, и его можно использовать в дальнейшем:

>>> x = '{motto}, {0}, and {food}'.format(42, motto = 3.14, food = [1, 2])

>>> x

'3.14, 42, and [1, 2]'

>>> x.split(' and ') # мы получили два элемента списка '3.14, 42,' и '[1, 2]'

['3.14, 42,', '[1, 2]']

>>> y = x.replace('and', 'but no milk') # замена подстроки

>>> y

'3.14, 42, but no milk [1, 2]'

**Использование ключей, атрибутов и смещений.**

При использовании format (как и %), в строках формата можно ссылаться на имена атрибутов объекта и ключи словарей, [] - означает ключи словаря, а точка . - прим-ся для доступа к атрибутам объектов, на которые ссылаются спецификаторы.

Примеры:

первый - подстановка ключа 'spam' словаря и атрибута значение 'platform' объекта модуля sys (позиционные аргументы);

второй - то же самое, но объекты для подстановки передаются в виде именованных аргументов.

>>> 'My {1[spam]} runs {0.platform}'.format(sys, {'spam': 'laptop'})

'My laptop runs win32'

В скобках [] в строках формата можно также указывать смещение от начала списка (или любой последовательности), но тольк оположительное смещение.

Чтобы обратиться по отриц. смещению, получить срез или вставить результат произвольного выражения, данные нужно подготовить отдельно, за пределами строки формата:

>>> somelist = list('SPAM')

>>> somelist

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

>>> 'first = {0[0]}, third = {0[2]}'.format(somelist)

'first = S, third = A'

>>> 'first = {0[0]}, third = {0[-1]}'.format(somelist)

...

TypeError: list indices must be integers, not str

- как видим, если [-1] использовать внутри строки формата, это приведет к ошибке!

>>> 'first = {0}, last = {1}'.format(somelist[0], somelist[-1])

'first = S, last = M'

>>> parts = somelist[0], somelist[-1], somelist[1:3]

>>> 'first = {0}, last = {1}, middle = {2}'.format(\*parts)

"first = S, last = M, middle = ['P', 'A']"

- также, если использовать [1:3] внутри строки формата, это приведет к ошибке!

- еще одно замечание, в последнем примере, в части .format(\*parts), символ \* обязателен, без него возникает ошибка:

>>> parts = somelist[0], somelist[-1], somelist[1:3]

>>> 'first = {0}, last = {1}, middle = {2}'.format(parts)

...

IndexError: tuple index out of range

**Специальные приемы форматирования.**

Метод format имеет возможности более точного форматирования, вслед за идентификатором символа подстановки, через двоеточие, можно указать спецификатор формата, поределяющий ширину поля вывода, выравнивание и код типа значения.

Синтаксис спецификатора формата:

*{fieldname! conversionflag:formatspec}*

Поля спецификатора:

- *fieldname* - порядковый номер или имя именованного аргумента, за которым может следовать необязательное имя ".name" атрибута или индекс "[index]" элементов.

- *conversionflag* - может быть r, s или a, которые определяют применение к значению встреонной функции repr, str или ascii соответственно.

- *formatspec* - определяет способ представления значения, описывает такие характеристики представления, как ширина поля вывода, выравнивание, дополнение, количество знаков после десятичной точки и т.д., и завершается необязательным кодом типа значения.

Поле formatspec, следующее за двоеточием, имеет синтаксис (квадратные скобки указывают необязательные поля):

*[[fill]align][sign][#][0][width][.precision][typecode]*

- в поле align может указываться символ <, >, = или ~, обозначающие выравнивание по левому/правому краю, дополнение знака числа или выравнивание по центру.

Также formatspec может содержать вложенные {} строки форматирования с именами полей, чтобы извлекать значения динамически.

Дополнительно метод format позволяет использовать код 'b' для отображения целых чисел в двоичном формате, код % означает символ процента, а для десятичных целых чисел используется код 'd'.

*Пример:*

Спецификатор {0:10} предписывает вывести значение первого позиционного аргумента в поле шириной 10 символов, {1:<10} предписывает вывести значение второго позиционного аргумента в поле шириной 10 символов, с выравниванием по левому краю, а спецификатор {0.platform:>10} предписывает вывести значение атрибута platform первого позиционного аргумента в поле шириной 10 символов, с выравниванием по правому краю:

>>> '{0:10} = {1:10}'.format('spam', 123.4567)

'spam = 123.4567'

>>> '{0:>10} = {1:<10}'.format('spam', 123.4567)

' spam = 123.4567 '

>>> '{0.platform:>10} = {1[item]:<10}'.format(sys, dict(item = 'laptop'))

' win32 = laptop '

format с вещественными числами:

>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159)

'3.141590e+00, 3.142e+00, 3.14159'

>>> '{0:f}, {1:.2f}, {2:06.2f}'.format(3.14159, 3.14159, 3.14159)

'3.141590, 3.14, 003.14'

{2:g} предписывает вывести третий аргумент, отформатированный в соответствии с представленным вещественных чисел по умолчанию, предусмотренным кодо 'g'.

{1:.2f} предписывает использовать формат 'f' представления вещественных чисел с двумя знаками после десятичной точки.

{2:06.2f} ограничивает ширину поля вывода 6 символами и предписывает дополнить число нулями слева.

Вывод чисел в восьмеричном, шестнадцатеричном и двоичном форматах:

>>> '{0:X}, {1:o}, {2:b}'.format(255, 255, 255)

'FF, 377, 11111111'

- и сравним с алтернативными методами:

>>> bin(255), int('11111111', 2), 0b11111111

('0b11111111', 255, 255)

>>> hex(255), int('FF', 16), 0xFF

('0xff', 255, 255)

>>> oct(255), int('377', 8), 0o377

('0o377', 255, 255)

Параметры форматирования можно указывать непосредственно в строке форматирования:

>>> '{0:.2f}'.format(1/3.0)

'0.33'

>>> '%.2f' % (1/3.0)

'0.33'

- или динамически извлекать из списка аргументов:

>>> '{0:.{1}f}'.format(1/3.0, 4)

'0.3333'

>>> '%.\*f' % (4, 1/3.0)

'0.3333'

В Питон 2.6 и 3.0 появилась встроенная функция format, которая может использоваться для форматирования одиночных значений. С тех. точки зрения, функция format вызывает метод format\_\_ объекта, который в свою очередь вызывает метод str.format каждого форматируемого элемента.

**Сравнение с оператором форматирования %** стр. 243

В общем случае, применение оператора форматирования выглядит проще, чем вызов метода format:

>>> print(' %s = %s ' % ('spam', 42))

spam = 42

>>> print(' {0} = {1} '.format('spam', 42))

spam = 42

В более сложных случаях, оба способа имеют почти одинаковую сложность.

Не будем здесь приводить подробные различия, в случае необходимости почитаем их потом, или дополним.

**Сравнения обоих методов стр. 246**

**Общие категории типов.** стр. 249

Сделаем паузу после исследования строк. Рассмотрим некоторые общие концепции большинства типов, которые будем рассматривать далее.

**Типы одной категории имеют общий набор операций.**

Формально в Питоне существуют три категории типов (и, соответственно, операций):

- Числа (целые, вещественные, с фиксированной точностью, рациональные и др.);

- Последовательности (строки, списки, кортежи);

- Отображения (словари).

Множества - образуют отдельную категорию типов (не отображают ключи в значения, не являются упорядоченными последовательностями).

Т.е., например, для любых объектов последовательностей X и Y справедливо следующее:

X + Y создает новый объект последовательности, включающий содержимое обоих операндов;

X \* N создает новый объект последовательности, включающий N копий операнда X.

- и так для всех остальных категорий типов и операций, т.е. операции последовательностей одинаково действуют на любые виды последовательностей и это же правило сохраняется для категории чисел и отображений.

**Изменяемые типы допускают непосредственное изменение.**

*Неизменяемые* (числа, строки, кортежи, фиксированные множества):

- объекты неизменяемых типов не поддерживают непосредственное изменение значения объекта, однако можно создават ьновые объекты с помощью выражений и присваивать их требуемым переменным.

*Изменяемые* (списки, словари, множества):

- изменяемые типы наоборот, всегда могут изменяться непосредственно, с помощью операций, которые не создают новые объекты. Изменяемые объекты могут быть скопированы, но поддерживают и возможности непосредственного изменения.

**Глава 8. Списки и словари.** стр. 253

Оба типа объектов изменяемые и являются коллекциями других объектов.

**Списки.** стр. 253

Список - гибкий тип упорядоченных коллекций. Могут содержать объекты любых типов: числа, строки и даже другие списки.

Ну и, естественно, списки могут изменяться непосредственно, они - изменяемые объекты.

Основные свойства списков. Списки это:

- Упорядоченные коллекции объектов произвольных типов. Т.о. списки это лишь место, в котором собраны др. объекты, поэтому их можно рассматривать как группы. Имеется позиционное упорядочивание элементов слева направо (как у последовательностей);

- Доступ к элементам по смещению. Есть операции извлечения элемента по индексу смещения, срезы, конкатенация;

- Переменная длина, гетерогенность и произвольное число уровней вложенности. Списки могут содержать не только односимвольные строки, но и любые др. объекты (гетерогенность) - вернее сказать, могут содержать, одновременно, разные типы объектов (это и есть гетерогенность). Могут изменять длину (непосредственное изменение). Имеют возможность создания произвольного числа уровней вложенности.

- Относятся к категории изменяемых объектов. Могут изменяться непосредственно и поддерживают все операции над последоватльностями (как и у строк). Естественно после операций (конкатенация и срез) над списком возвращается новый список, а не строки. Естественно списки поддерживают операции, неподдерживаемые строками (непосредственное изменение элементов, удаление, по длине и т.п.).

- Массивы ссылок на объекты. Формально списки могут содержать ноль или более сылок на объекты. Они чем-то напоминают массивы указателей (адресов). Списки - это самые настоящие массивы языка С, реализованные в интерпретаторе Питон, а не связанные структурой данных.

И запомним, всякий раз, когда выполняется присваивание объекта элементу какой-либо структуры или имени переменной, интерпретатор Питона сохраняет ссылку на этот объект, а не его копию (кроме случаев явного запроса операции копирования.)

**Таблица 8.1 Литералы списков и операции.** стр. 255

Литералы:

L = [] # Пустой список

L = [0, 1, 2, 3] # Четыре элемента

L = ['abc', ['def', 'ghi']] # Вложенные списки

**Списки в действии.** стр. 256

**Базовые операции над списками.**

>>> len([1, 2, 3]) # Длина

3

>>> [1, 2, 3] + [4, 5, 6] # Конкатенация

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

>>> ['Ni!'] \* 4 # Повторение

['Ni!', 'Ni!', 'Ni!', 'Ni!']

Также понимаем, что прис операции сложения, справа и слева должны быть последовательности одного типа, нельзя конкатенировать списки и строки, нужно их преобразовывать:

>>> str([1, 2]) + "34" # то же, что и "[1, 2]" + "34"

'[1, 2]34'

>>> [1, 2] + list("34") # то же, что и [1, 2] + ["3", "4"]

[1, 2, '3', '4']

**Итерации по спискам и генераторы списков.**

>>> 3 in [1, 2, 3] # Проверка на вхождение

True

>>> for x in [1, 2, 3]:

print(x, end=' ') # Итерации

1 2 3

- цикл for выбирает элементы последовательности в порядке слева направо и выполняет одну или более инструкций для каждого.

>>> res = [c \* 4 for c in 'SPAM'] # Генератор списков

>>> res

['SSSS', 'PPPP', 'AAAA', 'MMMM']

Эквивалент генератора списков, цикл for:

>>> res = []

>>> for c in 'SPAM':

res.append(c \* 4)

>>> res

['SSSS', 'PPPP', 'AAAA', 'MMMM']

Функция map действуе тпохоже, но применяет к элементам последовательности не выражение, а функцию и создает новый список:

>>> list(map(abs, [-1, -2, 0, 1, 2])) # вероятно abs это - абсолютное значение

[1, 2, 0, 1, 2]

list

**Индексы, срезы и матрицы.** стр. 258

>>> L = ['spam', 'Spam', 'SPAM!']

>>> L[2] # Отчет смещений начинается с нуля

'SPAM!'

>>> L[-2] # Отрицательное смещение

'Spam'

>>> L[1:] # Операция среза возвращает список

['Spam', 'SPAM!']

Рассмотрим простейшую матрицу:

>>> matrix = [[1, 2, 3],[4, 5, 6],[7 , 8 , 9]] # лишние пробелы в [7 , 8 , 9]

не влияют на правильную работу

>>> matrix

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> matrix[1]

[4, 5, 6]

>>> matrix[1][1]

5

>>> matrix[2][0]

7

>>> mat = [[1, 2, 3], # есть и такой способ записи матрицы, неск-ко строк

[4, 5, 6],

[7, 8, 9]]

>>> mat

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> mat[1][1]

5

**Изменение списка.** стр. 259

**Присваивание по индексам и срезам.**

>>> s = ['spam', 'Spam', 'SPAM!']

>>> s[1] = 'eggs'

>>> s

['spam', 'eggs', 'SPAM!']

>>>

>>> s[0:2] = ['eat', 'more']

>>> s

['eat', 'more', 'SPAM!']

- происходит непосредственное изменение списка.

Прсиваивание работает как в С, интерпретатор замещает старую ссылку на объект в указанном смещении на новую.

Присваивание срезу, выполняется за два действия:

- *Удаление*. Раздел списка, определяемый слева от оператора =, удаляется.

- *Вставка*. Новые элементы, справа от оператора =, вставляются в список, начиная с левого края, где находился прежний, удаленный срез.

При этом мы видим, что число удаляемых элементов не обязательно должно быть равно количеству вставляемых элементов (можно удалить больше чем вставить и наоборот, вставить больше чем удалить).

Обычно используются более простые способы замены, вставки, и удаления (напр. конкатенация, методы insert, pop, remove).

**Методы списков.**

>>> s

['eat', 'more', 'SPAM!']

>>> s.append('please') # метод добавляет элемент в конец списка

>>> s

['eat', 'more', 'SPAM!', 'please']

>>> s.sort() # Сортировка элементов списка ('S' < 's')

>>> s

['SPAM!', 'eat', 'more', 'please']

*Методы* - это функции (в действительности - атрибуты, ссылающиеся на функции), которые связаны с определенным типом объектов.

Метод append один из самых популярных методов, добавляет элемент (ссылку на объект) в конец списка.

append принимает только один аргумент (т.е. один объект, а не список).

Метод sort - сортирует, упорядочивает элементы. Можно изменять порядок сортировки с помощью именованных аргументов - специальных синтаксических конструкций вида "name=value", которые используются в вызовах функций для передачи параметров настройки по их именам.

Пример - аргумент key метода sort, а также reverse:

>>> d = ['abc', 'ABD', 'abe'] # Сортировка с учетом регистра символов

>>> d.sort()

>>> d

['ABD', 'abc', 'abe']

>>>

>>> d = ['abc', 'ABD', 'abe']

>>> d.sort(key=str.lower) # Приведение символов к нижнему регистру

>>> d

['abc', 'ABD', 'abe'] # сами элементы не изменилсь после действия аргумента key

>>>

>>> d = ['abc', 'ABD', 'abe']

>>> d.sort(key=str.lower, reverse=True) # Изменяет направление сортировки

>>> d

['abe', 'ABD', 'abc'] # сами элементы не изменились после

действия аргументов

**Замечание!** В Питоне 3.х нельзя сортировать список с различными типами объектов:

>>> t = [3, 2, 1, 'sss']

>>> t.sort()

...

TypeError: unorderable types: str() < int()

Отметим, append и sort изменяют сам объект списка (а возвращают None). Т.е. при использовании этих методов нет необходимости в присвоении результата.

Также в последних версиях Питона сортировку можно выполнить с помощью встроенной функции, при этом взвращая новый список, не изменяя старыый:

>>> d = ['abc', 'ABD', 'abe']

>>> sorted(d, key=str.lower, reverse=True) # встроенная функция сортировки

['abe', 'ABD', 'abc']

>>> d

['abc', 'ABD', 'abe']

>>> d = ['abc', 'ABD', 'abe']

>>> sorted([x.lower() for x in d], reverse=True) # Элементы, предварительно изменяются

['abe', 'abd', 'abc']

>>> d

['abc', 'ABD', 'abe']

- во втором примере выполняется сортировка не оригинального, а уже измененного (временного) списка, получившегося с помощью генератора списков.

Иногда функция sorted, может оказаться удобнее метода sort.

reverse - изменяет порядок элементов на обратный

extend и pop - вставляют несколько элементов в конец списка и удаляют элементы из конца списка соответственно.

reversed - во многом напоминает функцию sorted, но её необходимо обертывать в вызов функции list, потому что она возвращает итератор:

>>> l = [1, 2]

>>> l.extend([3, 4, 5]) # Добавляет элементы в конец списка

>>> l в отличие от append, где добавляется только один объект

[1, 2, 3, 4, 5]

>>> l.pop() # Удаляет и возвращает последний элемент

5

>>> l

[1, 2, 3, 4]

>>> l.reverse()

>>> l

[4, 3, 2, 1]

>>> list(reversed(l))

[1, 2, 3, 4]

В некоторых программах pop и append реализуют структуру данных типа стек - "последний пришел, первый вышел" (Last-In-First-Out, LIFO). Конец списка - вершина стека:

>>> l = []

>>> l.append(1) # Втолкнуть в стек

>>> l.append(2)

>>> l

[1, 2]

>>> l.pop() # Вытолкнуть из стека

2

>>> l

[1]

pop - может принимать необязательное смещение эелемента, который удаляется из списка и возвращается (по умолчанию - это последний элемент)

remove - позволяет удалять элементы с определенными значениями

insert - позволяет вставлять элементы в определенную позицию

index - позволяет отыскивать смещение элемента по заданноу значению

>>> l = ['spam', 'eggs', 'ham']

>>> l.index('eggs') # Узнаем индекс объекта

1

>>> l.insert(1, 'toast') # Вставка в требуемую позицию

>>> l

['spam', 'toast', 'eggs', 'ham']

>>> l.remove('eggs') # Удаление элемента с определенным значением

>>> l

['spam', 'toast', 'ham']

>>> l.pop(1) # Удаление элемента в указанной позиции

'toast'

>>> l

['spam', 'ham']

>>> l = ['spam', 'eggs', 'ham', 'toast']

>>> l

['spam', 'eggs', 'ham', 'toast']

>>> l.pop(-3) # Причем можно указать и отрицательное смещение

'eggs'

>>> l

['spam', 'ham', 'toast']

- Кстати! Наконец-то понятно, что такое метод, который что-то возвращает (как - pop, к примеру) и метод, который не возвращает (т.е. просто что-то делает, ну как - remove, например).

Очень полезное знание!

**Прочие часто используемые операции над списками.**

Инструкция del для удаления элемента или среза:

>>> l = ['SPAM', 'eggs', 'ham', 'toast']

>>> del l[0] # Удаление одного элемента из списка

>>> l

['eggs', 'ham', 'toast']

>>> del l[1:] # Удаление среза

>>> l # То же, что и l[1:] = []

['eggs']

В последнем примере нужно оговориться, что операция присваивания пустого списка по индексу элемента приводит к сохранению ссылки на пустой список в этом элементе, а не к его удалению:

>>> l = ['eggs', 'ham', 'toast']

>>> l[1:] = []

>>> l

['eggs']

>>> l[0] = []

>>> l

[[]] # Вот и подтверждение пустого списка вместо удаления элемента

>>> del l[0]

>>> l

[]

>>> l = ['SPAM', 'eggs', 'ham', 'toast']

>>> del(l[0])

>>> l

['eggs', 'ham', 'toast']

>>> del l[0]

>>> l

['ham', 'toast']

- Замечание! Как видим из этого примера - del, в моем интерпретаторe IDLE, одинаково работает как без скобок так и со скобками, или у меня глюк или это особенности работы инструкций в Питоне (не даром сказано ни метод, ни функция, а именно инструкция) - еще одна особенность, нужно бы запомнить и изучить.

**Словари.** стр. 264

После списков словари являются, пожалуй, самым гибким из встроенных типов данных в Питон.

В отличие от списков, словари не являются упорядоченными коллекциями, в них элементы извлекаются по ключу, а не по смещению.

Основные характеристики словарей:

- Доступ к элементам по ключу, а не по индексу. Иногда словари называют ассоциативными массивами, или хешами. Для получения элементов применяется не смещение, а ключ, играя т.о. роль индекса элемента.

- Неупорядоченные коллекции произвольных объектов. Элементы словарей хранятся в произвольном порядке, это было сделано для более быстрого поиска. Ключи описывают символическое (не физическое) местоположение элементов в словаре.

- Переменная длина, гетерогенность и произвольное число уровней вложенности. Словари могут увеличиваться и уменьшаться (непосредственно, без создания копий). Могут содержать объекты любых типов и поддерживают произвольное число уровней вложенности.

- Относятся к категории "изменяемых отображений". Словари явл-ся изменяемыми объектами, но не поддерживают операции над последовательностями как строки и списки. Словари - это неупорядоченные коллекции. Словари - единственный встроенный представитель объектов-отображений (объекты, которые отображают ключи на значения).

- Таблицы ссылок на объекты (хеш-таблицы). Если списки - это массивы ссылок на объекты, с возможностью доступа к элементам по их позиции, то словари - это неупорядоченные таблицы ссылок на объекты, которые поддерживают доступ к элементам по ключу. Внутри словари реализованы как хеш-таблицы (структуры данных, которые обеспечивают очень высокую скорость поиска). Интерпретатор Питон использует оптимизированные алгоритмы хеширования для обеспечения максимально высокой скорости поиска ключей. Словари хранят ссылки на объекты (а не их копии).

**Таблица 8.2 Литералы словарей и операции.** стр. 266

d = {} # Пустой словарь

d = {'spam' : 2, "eggs" : 3} # Словарь из двух элементов

d = {'spam' : {'ham' : 1, 'eggs' : 2} # Вложение

**Словари в действии.** стр. 266

**Базовые операции над словарями.**

>>> d = {'spam':2, 'ham':1, 'eggs':3} # Создание словаря

>>> d

{'spam': 2, 'eggs': 3, 'ham': 1} # Вот оно - случайный порядок следования

>>> d['spam'] # Извлечение значения по ключу

2

функция len - возвращает число элементов в словаре, или, то же самое, длину ключей;

оператор in - позволяет проверить наличие ключа, и только ключа! попытка проверки наличия определенного элемента выдает - false, видимо разработчики Питона решили, что и так сойдет, находите ключ, а потом по нему смотрите значение.

метод keys - возвращает все ключи, имеющиеся в словаре, в виде списка;

>>> d = {'spam':2, 'ham':1, 'eggs':3}

>>> len(d) # Число элементов словаря

3

>>> 'ham' in d # Начинаем испытание проверки на вхождение

True

>>> 3 in d

False

>>> '3' in d

False

>>> d = {'spam':'pam', 'ham':'tam', 'eggs':'dam'}

>>> 'spam' in d

True

>>> 'pam' in d

False

>>>

>>> d = {'spam':2, 'ham':1, 'eggs':3}

>>> list(d.keys())

['spam', 'eggs', 'ham'] # Создает новый список ключей

>>>

- конечно метод keys пришлось обернуть в функцию list, т.к. keys возвращает итератор, а не список.

А вызов функции list принудительно выполняет обход всех значений итератора.

**Изменение словарей.**

В примерах ниже ознакомимся с операциями изменения словарей, кстати в Питоне все типы-коллекции могут вкладываться друг в друга в произвольном порядке.

>>> d = {'eggs':3, 'ham':1, 'spam':2}

>>> d

{'ham': 1, 'eggs': 3, 'spam': 2}

>>>

>>> d['ham'] = ['grill', 'bake', 'fry'] # Изменение элемента

>>> d

{'ham': ['grill', 'bake', 'fry'], 'eggs': 3, 'spam': 2} # Вложение получилось

>>>

>>> del d['eggs'] # Удаление элемента

>>> d

{'ham': ['grill', 'bake', 'fry'], 'spam': 2}

>>>

>>> d['brumch'] = 'Bacon' # Добавление нового элемента

>>> d

{'ham': ['grill', 'bake', 'fry'], 'brumch': 'Bacon', 'spam': 2}

Операция присваивания по существующему ключу словаря приводит к замене предыдущего значения, ассоциируемого с этим ключом.

В отличие от списков, словари допускают присваивание по новому ключу (которого не было в словаре), в этом случае просто создается новый элемент словаря (пример с ключом 'brumch').

**Дополнительные методы словарей.**

метод values - возвращает список значений элементов словаря:

>>> d = {'spam':2, 'ham':1, 'eggs':3}

>>> list(d.values())

[1, 3, 2]

метод items - возвращает кортежи пар (key, value), вернее даже список кортежей пар:

>>> d = {'spam':2, 'ham':1, 'eggs':3}

>>> list(d.items())

[('ham', 1), ('eggs', 3), ('spam', 2)]

При попытке вызвать несуществующий элемент словаря мы получим ошибку, но метод get позволяет вернуть значение по умолчанию или None:

>>> d.get('spam')

2

>>> print(d.get('toast'))

None

>>> d.get('toast', 88)

88

Метод update - своего рода метод конкатенации словарей, он объединяет ключи и значения одного словаря с ключами и значениями другого, просто перезаписывая значения с одинаковыми ключами:

>>> d

{'ham': 1, 'eggs': 3, 'spam': 2}

>>> d2 = {'toast':4, 'muffin':5}

>>> d.update(d2)

>>> d

{'ham': 1, 'toast': 4, 'muffin': 5, 'eggs': 3, 'spam': 2}

>>> d = {'spam':2, 'ham':1, 'eggs':3} # А теперь посмотрим на перезапись элементов

>>> d2 = {'toast':4, 'spam':5} # Пусть в d2 будет тоже ключ 'spam' как и в d

>>> d.update(d2)

>>> d # update d по d2

{'ham': 1, 'toast': 4, 'eggs': 3, 'spam': 5}

>>> d2

{'spam': 5, 'toast': 4}

>>> d = {'spam':2, 'ham':1, 'eggs':3}

>>> d2 = {'toast':4, 'spam':5}

>>> d2.update(d) # update d2 по d

>>> d

{'ham': 1, 'eggs': 3, 'spam': 2}

>>> d2

{'ham': 1, 'spam': 2, 'eggs': 3, 'toast': 4}

- как видим разница очевидна, порядок конкатенации имеет значение.

Метод pop удаляет ключ из словаря и возвращает его значение

>>> d

{'ham': 1, 'toast': 4, 'muffin': 5, 'eggs': 3, 'spam': 2}

>>> d.pop('muffin')

5

>>> d.pop('toast') # Удаляет ключ и возвращает значение заданного ключом элемента

4 # или проще - значение заданного ключа (пары ключ:значение)

# возвращает не индекс! а значение!!! не путать!

>>> d

{'ham': 1, 'eggs': 3, 'spam': 2}

Сравним метод pop с работой ег о со списками, где он также удаляет элементы и возвращает их значения, используя при этом номер позиции, а не ключ как pop для словарей:

>>> L = ['aa', 'bb', 'cc', 'dd']

>>> L.pop()

'dd'

>>> L

['aa', 'bb', 'cc']

>>> L.pop(1)

'bb'

>>> L

['aa', 'cc']

Имеется также метод copy, который будет рассмотрен позднее в этой книге.

**Таблица языков.**

Рассмотрим применение словаря на примере таблицы языков программирования и их создателей:

>>> table = {'Python':'Guido Van Rossum',

'Perl':'Larry Wall',

'Tcl':'John Ousterhout'}

>>> language = 'Python'

>>> cerator = table[language]

>>> cerator

'Guido Van Rossum'

>>>

>>> for lang in table: # То же, что и - for lang in table.keys()

print(lang, '\t', table[lang])

Tcl John Ousterhout

Python Guido Van Rossum

Perl Larry Wall

>>> for a in table: # Просто проверяю lang ключевое слово или можно любое

print(a, '\t', table[a])

Tcl John Ousterhout

Python Guido Van Rossum

Perl Larry Wall

- приведенная команда с циклом for просто выполняет обход всех ключей в таблице и выводит список ключей и их значений, разделенных символом табуляции.

Словари не являются последовательностями, как списки и строки, но если необходимо выполнить обход элементов словаря, нет ничего сложного - это легко сделать методом keys, возвращающим все ключи словаря, которые можно обойти в цикле for, а внутри цикла можно получить значения ключей, как было сделано у нас в примере выше.

Для любого словаря D цикл можно оформить как for key in D, что равносильно полной форме записи - for key in D.keys(), пример:

>>> for a in table.keys():

print(a, '\t', table[a])

Tcl John Ousterhout

Python Guido Van Rossum

Perl Larry Wall

**Замечания по использованию словарей.** стр. 271.

- Операции над последовательностями неприменимы к словарям. Словари - это отображения, а не последовательности. Операции конкатенация (упорядоченное объединение) и извлечение среза (непрерывного блока элементов) просто неприменимы, интерпретатор даже выдаст ошибку.

- Присваивание по несуществующему индексу приводит к созданию нового элемента. Т.е. можем ключи создать сразу, литералом нового словаря или потом добавить их путем присваивания значения новому ключу.

- Ключи не обязательно должны быть строками. Ключами могут быть любые неизменяемые объекты (строки, числа). Например целые числа-ключи могут служить индексами к элементам (как у списков), можно использовать и кортежи в качестве ключей (составные ключи). Экземпляры классов могут быть ключами, если у них будут методы, которые сообщат интерпретатору, что он имеет дело с неизменямыми объектами.

**Использование словарей для имитации гибких списков.**

Помним, что списки не допускают присваивание по индексам, находящимся за пределами списков:

>>> L = []

>>> L[99] = 'spam'

...

IndexError: list assignment index out of range

Построим подобный пример со словарем:

>>> d = {}

>>> d[99] = 'spam'

>>> d[99]

'spam'

>>> d

{99: 'spam'}

- вот такой вот гибкий эквивалент списков, где целочисленные ключи словаря играют роль индексов в списках.

**Использование словарей для структур разреженных данных.**

Рассмотрим структуры разреженных данных, такие как многомерные массивы, где всег онесколько элементов имеют определенные значения:

>>> mat = {}

>>> mat[(2, 3, 4)] = 88

>>> mat[(7, 8, 9)] = 99

>>>

>>> x = 2; y = 3; z = 4 # Символ ; отделяет инструкции

>>> mat[(x, y, z)]

88

>>> mat

{(2, 3, 4): 88, (7, 8, 9): 99}

Здесь словарь представляет трехмерный массив, в котором только два элемента, (2, 3, 4) и (7, 8, 9), имеют определенные значения. Ключами словаря являются кортежи.

Т.о. вместо создания практически пустой терхмерной матрицы мы сделали более эффективное представление.

При попытке обратиться к пустому элементу получим ошибку:

>>> mat[(2, 3, 6)]

...

KeyError: (2, 3, 6)

**Как избежать появления ошибок обращения к несуществующему ключу.**

Есть, минимум, три способа получить значение по умолчанию вместо ошибки:

- предварительная проверка ключа с помощью if;

- конструкция try, для перехвата и обработки исключения;

- метод get, для явного возвращения значяения по умолчанию.

>>> mat

{(2, 3, 4): 88, (7, 8, 9): 99}

>>>

>>> if (2,3,6) in mat:

print(mat[(2,3,6)])

else:

print(0)

0

- наверное else должен быть на одном уровне с if, но мне пришлось его написать в самом начале строки в интерпретаторе IDLE, иначе правое расположение else возбуждало ошибку.

Мы ведь помним про уровни в Питоне? - Операторы и методы должны быть на соответствующи отступах в тексте сценария.

>>> mat

{(2, 3, 4): 88, (7, 8, 9): 99}

>>>

>>> try:

print(mat[2,3,6])

except KeyError: # В случае возникновения ошибки перехватить

print(0) исключение и обработать

… # то же самое с отступами как и выше

0

>>> mat

{(2, 3, 4): 88, (7, 8, 9): 99}

>>>

>>> mat.get((2,3,4), 0) # Существующий ключ, извлекается и возвращается

88

>>> mat.get((2,3,6), 0) # Отсутствует, используется аргумент default

0

**Использование словарей в качестве "записей".**

Словари могут представлять самые разные типы структурированной информации. Например один из многих способов описания свойств элементов в программах, т.е. играть роль "структур" и "записей" в других ЯП.

Пример, заполнение словаря путем присваивания значений новым ключам в виде нескольких инструкций:

>>> rec = {}

>>> rec['name'] = 'mel'

>>> rec['age'] = 45

>>> rec['job'] = 'trainer/writer'

>>>

>>> print(rec['name'])

mel

След. пример. Заполнение в единственной инструкции, присутствуют вложенный словарь и список:

>>> mel = {'name':'Mark',

'jobs':['trainer','writer'],

'web':'www.rmi.net/`lutz',

'home':{'state':'C0', 'zip':80513}}

>>> mel

{'name': 'Mark', 'web': 'www.rmi.net/`lutz', 'jobs': ['trainer', 'writer'], 'home': {'zip': 80513, 'state': 'C0'}}

- чтобы извлечь компоненты объектов, достаточно объединить цепочку операции индексирования:

>>> mel['name']

'Mark'

>>> mel['jobs']

['trainer', 'writer']

>>> mel['jobs'][1]

'writer'

>>> mel['home']['zip']

80513

В простых случаях тиакие схемы работают хорошо, просто и удобно. В более сложных случаях используются, конечно, классы.

***Интерфейсы словарей.***

Некоторые расширения Питон предоставляют интерфейсы, которые выглядят и действуют как словари. Например, обращение к индексированным файлам данных в формате DBM во многом напоминает обращение к словарю, котоырй сначала требуется открыть. Строки сохраняются и извлекаются индексированием по ключу:

import anydbm

file = anydbm.open('filename') # Ссылка на файл

file['key'] = 'data' # Сохранение данных по ключу

data = file['key'] # Извлечение данных по ключу

Далее мы узнаем каким способом сохранять целые объекты Питона - достаточно лишь заменит ьимя модуля anydbm на shelve (shelves(хранилища) - это БД с доступом к информации по ключу, предназначенные для хранения объектов Питон).

Кстати! У меня команда import anydbm - выдала ошибку, что такого модуля нет, а import shelve прошла нормально. Возможно anydbm это не модуль, а имя любого модуля, связанного с БД, отсюда и ошибка, т.к. я не знаю модулей Питона, которые связаны с БД. Или у меня нет модуля anydbm (может он нестандартный или его уже нет в Питоне 3.4).

Поддержка CGI- сценариев в Питоне также обеспечивает интерфейс, напоминающий словарь. Вызов метода cgi.FieldStorage() возвращает объект, по характеристикам напоминающий словарь - с одной записью для каждого поля ввода, находящегося на клиентской веб-странице.

import cgi

form = cgi.FieldStorage() # Анализирует данные формы

if 'name' in form:

showReply('Hello, ' + form['name'].value)

Все эти объекты являются примерами отображений. Далее мы убедимся, что словарные интерфейсы имеют отношение ко множеству встроенных инструментво Питона.

**Другие способы создания словарей.** стр. 275

В Питоне 2.3 и выше последние два вызова конструктора dict (в действительности - имени типа) в примере ниже, имеют тот же эффект, что и литералы и форма присваивания по отдельным ключам в примерах выше:

>>> {'name':'mel', 'age':45} # Традиционное литеральное выражение

{'name': 'mel', 'age': 45}

>>> d = {} # Динамическое присваивание по ключам

>>> d['name'] = 'mel'

>>> d['age'] = 45

>>>

>>> d

{'name': 'mel', 'age': 45}

>>> dict(name = 'mel', age = 45) # Форма именованных аргументов

{'name': 'mel', 'age': 45}

>>> dict([('name', 'mel'), ('age', 45)]) # Кортежи ключ/значение

{'name': 'mel', 'age': 45}

- Первый вариант. Удобен, если содержимое словаря известно заранее;

- Второй вариант. Удобно использовать, когда необходимо динамически создавать словарь по одному полю за раз;

- Третий вариант. Даже компактнее чем литералы, но требует, чтобы все ключи были строками.

- Четвертый вариант. Удобен когда ключи и значения во время выполнения программы необходимо хранить в виде последовательностей.

Если значения ключей остаются все время одними и теми же, можно просто передать список ключей и начальное значение (по умолчанию оно равно None):

>>> dict.fromkeys(['a', 'b'], 0)

{'b': 0, 'a': 0}

>>> dict.fromkeys([1, 2], 'a')

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

>>> dict.fromkeys([1, 2], 0)

{1: 0, 2: 0}

**Замечание!** - Методы d.keys, d.values и d.items возвращают итерируемые представления вместо списков.

**Генераторы словарей.**

Генераторы выполняют цикл, отбирают пары ключ/значение в каждой итерации и заполняют ими новый словарь.

>>> list(zip(['a','b','c'], [1,2,3]))

[('a', 1), ('b', 2), ('c', 3)]

- здесь мы объединяем ключи и значения, новая функцияя - zip у нас, которая и объединяет.

>>> d = dict(zip(['a','b','c'], [1,2,3]))

>>> d

{'b': 2, 'a': 1, 'c': 3}

- а здесь мы уже создаем словарь используя результат вызова функции zip.

А теперь посмотрим на работу генератора словарей, также с помощью функции zip:

>>> D = {k:v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}

>>> D

{'b': 2, 'a': 1, 'c': 3}

Генераторы универсальны и могут даже вычислять ключи с помощью выражений:

>>> D = {x: x\*\*2 for x in [1, 2, 3, 4]}

>>> D

{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c\*4 for c in 'SPAM'} # Цикл через итерируемый объект

>>> D

{'P': 'PPPP', 'M': 'MMMM', 'A': 'AAAA', 'S': 'SSSS'}

>>> D = {c.lower(): c+'!' for c in ['SPAM', 'EGGS', 'HAM']}

>>> D

{'ham': 'HAM!', 'eggs': 'EGGS!', 'spam': 'SPAM!'}

Генераторы словарей удобно использовать для инициализации словарей из списков ключей, почти как метод fromkeys из предыдущего раздела:

>>> D = dict.fromkeys(['a', 'b', 'c'], 0)

>>> D

{'b': 0, 'a': 0, 'c': 0}

- инициализация списком ключей

>>> D = {k:0 for k in ['a', 'b', 'c']}

>>> D

{'b': 0, 'a': 0, 'c': 0}

- То же самое, но с помощью генератора словаря

>>> D = dict.fromkeys('spam')

>>> D

{'m': None, 'a': None, 'p': None, 's': None}

- из другого итерируемого объекта, используются значения по умолчанию

>>> D = {k:None for k in 'spam'}

>>> D

{'m': None, 'a': None, 'p': None, 's': None}

**Представления словарей.**

Методы keys, values, items возвращают объекты представлений.

Объекты представлений - это итерируемые объекты, т.е. объекты, которые вместо всего списка значений возвращают по одному значению за одно обращение.

Представления словарей также сохраняют оригинальный порядок следования компонентов словаря, отражают результаты операций, которые выполняются над словарем и поддерживают операции над множествами.

Но они не являются списками и не поддерживают обращения к элементам по индексам и метод sort, и не отображают значения своих элементов при выводе.

Естественно мы обертываем вызовы этих трех методов в функцию list.

>>> D = dict(a=1, b=2, c=3)

>>> D

{'b': 2, 'a': 1, 'c': 3}

>>> K = D.keys() # Создает объект представления

>>> K

dict\_keys(['b', 'a', 'c'])

>>> list(K) # Принудительное создание списка

['b', 'a', 'c']

>>> V = D.values() # То же самое и для значений элементов

>>> V

dict\_values([2, 1, 3])

>>> list(V)

[2, 1, 3]

>>> list(D.items())

[('b', 2), ('a', 1), ('c', 3)]

>>> K[0] # Ошибка при попытке выполнить операцию над списком

...

TypeError: 'dict\_keys' object does not support indexing

>>> list(K)[0]

'b'

Но эту особенность мы будем редко замечать, т.к. в Питон конструкции обхода элементов в цикле автоматически заставляют итерируемые объекты возвращать по одному результату в каждой итерации:

>>> for k in D.keys():

print(k)

b

a

c

Кроме того словари в итерации последовательно возвращают ключи, что исключает необходимость метода keys:

>>> for key in D:

print(key)

b

a

c

Также представления способны динамически отражать все последующие изменения в словарях, выполненные уже после создания объекта отображения:

>>> D

{'b': 2, 'a': 1, 'c': 3}

>>> K = D.keys()

>>> V = D.values()

>>> list(K) # Представления сохраняют оригинальный

['b', 'a', 'c'] # порядок следования ключей в словаре

>>> list(V)

[2, 1, 3]

>>> del D['b'] # Изменяет словарь непосредственно

>>> D

{'a': 1, 'c': 3}

>>> list(K) # Изменения в словаре отражаются на объектах представлений

['a', 'c']

>>> list(V)

[1, 3]

**Представления словарей и множества.** стр. 279.

Объекты представлений, возвращаемые методом keys, похожи на множества и поддерживают операции множеств, такие как пересечение и объединение.

Объекты представлений метода values, такой особенностью не обладают, потмоу что они не являются уникальными.

Объекты представлений метода items такой способностью обладают, если пары (key, value) являются уникальными и хешируемыми.

Мы помним, что множества подобны словарям, ключи которых не имеют значений. Множества, к тому же, заключаются в фигурные скобки, как словари. Подобно ключам словарей, элементы множеств неупорядочены, уникальны и неизменяемы.

Ниже пример, как интерпретируются списки ключей, когда они используются в операциях над множествами. В таких операциях объекты представлений могут смешиваться с другими представлениями, множествами и словарями (в этом случае словари интерпретируются как представления):

>>> K | {'x':4} # Представления ключей (и иногда элементов) похожи на множества

{'c', 'a', 'x'}

>>> V & {'x':4}

TypeError: unsupported operand type(s) for &: 'dict\_values' and 'dict'

>>> V & {'x':4}.values()

TypeError: unsupported operand type(s) for %: 'dict\_values' and 'dict\_values'

>>> V & list({'x':4}.values())

TypeError: unsupported operand type(s) for &: 'dict\_values' and 'list'

>>> V | list({'x':4}.values())

TypeError: unsupported operand type(s) for |: 'dict\_values' and 'list'

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

>>>

>>> D.keys() & D.keys() # Пересечение представлений ключей

{'b', 'c', 'a'}

>>> D.keys() & {'b'} # Пересечение представления ключей и множества

{'b'}

>>> D.keys() & {'b':1} # Пересечение представления ключей и словаря

{'b'}

>>> D.keys() | {'b', 'c', 'd'} # Объединение представления ключей и множества

{'b', 'c', 'd', 'a'}

Представления элементов словарей также могут обладать свойствами множеств, если они допускают возможность хеширования, т.е. - если они содержат неизменяемые объекты:

>>> D = {'a':1}

>>> list(D.items())

[('a', 1)]

>>> D.items() | D.keys()

{('a', 1), 'a'}

>>> D.items() | D

{('a', 1), 'a'}

>>> D.items() | {('c', 3), ('d', 4)}

{('a', 1), ('c', 3), ('d', 4)}

>>> dict(D.items() | {('c', 3), ('d', 4)})

{'d': 4, 'c': 3, 'a': 1}

**Сортировка ключей словаря.**

Итак, в версиях 3.х для сортировки ключей нужно либо преобразовать объект представления ключей в список, либо воспользоваться фуекцией sorted, применив её к объекту, возвращаемому методу keys или к самому словарю:

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

>>> D

{'b': 2, 'c': 3, 'a': 1}

>>>

>>> Ks = D.keys()

>>> Ks.sort() # Сортировка объекта представления выдает ошибку

...

AttributeError: 'dict\_keys' object has no attribute 'sort'

>>> Ks = list(Ks) # Преобразовать в список и потом отсортировать

>>> Ks.sort()

>>> for k in Ks:

print(k, D[k])

a 1

b 2

c 3

>>> D

{'b': 2, 'c': 3, 'a': 1}

>>>

>>> Ks = D.keys() # Вызвать функцию sorted() с результатом вызова keys

>>> for k in sorted(Ks): # soreted() принимает итерируемые объекты и возвращает

print(k, D[k]) # результат

a 1

b 2

c 3

>>> D

{'b': 2, 'c': 3, 'a': 1} # Еще лучше отсортировать сам словарь

>>>

>>> for k in sorted(D): # Итератор словаря возвращает ключи

print(k, D[k])

a 1

b 2

c 3

**Сравнивание словарей в Питон 3.х больше не поддерживается.**

Можно лишь сравнить отсортированные списки ключей вручную:

sorted(D1.items()) < sorted(D2.items()) # То же, что и D1 < D2 в версии 2.6/12

Метод has\_key умер, да здравствует has\_key!

Вместо метода has\_key в Питон 3.0 рекомендуется использовать оператор in или проверять результат вызова метода get на равенство значения по умолчанию (in предпочтительнее):

>>> D

{'b': 2, 'c': 3, 'a': 1}

>>> D.has\_key('c') # Работает только в версии 2.Х (True или False)

...

AttributeError: 'dict' object has no attribute 'has\_key'

>>> 'c' in D

True

>>> 'x' in D

False

>>> if 'c' in D: # Предпочтительнее в версии 3.0

print('present', D['c'])

present 3

>>> print(D.get('c'))

3

>>> print(D.get('x'))

None

>>> if D.get('c') != None: # Вариант с get

print('present', D['c'])

present 3

**Глава 9. Кортежи, файлы и все остальное.** стр. 284

Завершаем детальное исследование базовых типов объектов Питона. Рассмотрим кортежи - коллекции объектов, которые не могут изменяться; Файлы - интерфейсы к внешним файлам на компьютере. Узнаем, что кортежи - это относительно простые объекты, с операциями как у строк и списков. Объекты-файлы - широко используемые многофункциональные инструменты для работы с файлами.

**Кортежи.**

Кортежи - простые группы оъектов. Действуют как списки, но только неизменяемые (непосредственно нельзя) и литералы - последовательность элементов в круглых скобках (а не [] как списки).

Свойства кортежей:

- Это упорядоченные коллекции объектов произвольных типов. Кортежи - это коллекции упорядоченных объектов (как строки и списки), и могут содержать объекты любых типов.

- Обеспечивают доступ к элементам по смещению. Доступ по смещению (как строки и списки), не по ключу. Конечно поддерживают все операции со смещением, индексирование и извлечение среза и т.п.

- Относятся к категории неизменяемых последовательностей. Не изменяются непосредственно как и строки, хотя и поддерживают операции над последовательностями. Но не поддерживают операции списков, такие как метод append, например.

- Имеют фиксированную длину, гетерогенны и поддерживают произвольное число уровней вложенности. Могут хранить списки, словари и другие кортежи.

- Это массивы ссылок на объекты. Как и списки, кортежди проще представлять как массивы ссылок на объекты - кортежи хранят указатели (ссылки) на другие объекты, а операция индексирования над кортежами выполняется очень быстро.

**Таблица 9.1 Литералы кортежей и операции.** стр. 285

**Кортежи в действии.** стр. 286

>>> (1, 2) + (3, 4) # Конкатенация

(1, 2, 3, 4)

>>>

>>> (1, 2) \* 4 # Повторение

(1, 2, 1, 2, 1, 2, 1, 2)

>>>

>>> T = (1, 2, 3, 4) # Индексирование, извлечение среза

>>> T[0], T[1:3]

(1, (2, 3))

**Особенности синтаксиса определения кортежей: запятые и круглые скобки.**

Поскольку круглые скобки применяются и в выражениях, интерпретатору нужно как-то показать, что единственный объект в скобках - это кортеж, а не выражение. Делается это очень просто - нужно просто добавить запятую после элемента:

>>> x = (40) # Целое число

>>> x

40

>>> y = (40,) # Кортеж, содержащий целое число

>>> y

(40,)

Иногда можно даже не ставить скобки при определении кортежа:

>>> T = 1, 2, 'Ni!', 1.34

>>> T

(1, 2, 'Ni!', 1.34)

Но если будем передавать кортежи функциям в виде литералов или что-то подобное где скобки важны, то они там нужны обязательно. Поэтому все-таки - лучше ставить скобки, чтобы ненароком ничего не напутать.

**Преобразования, методы и неизменяемость.**

Итак, кортежи имеют операции идентичные строковых и списков. Возвращают они, конечно, же, кортежи. А также кортежи имеют ограниченный набор методов.

Если необходима сортировка, то кортеж сначала нужно преобразовать в список и применить сортировку или функцию sorted:

>>> T = ('aa', 'cc', 'dd', 'bb')

>>> T

('aa', 'cc', 'dd', 'bb')

>>> tmp = list(T) # Создать список из элементов кортежа

>>>

>>> tmp.sort() # Отсортировать список

>>> tmp

['aa', 'bb', 'cc', 'dd']

>>> T = tuple(tmp) # Создать кортеж из элементов списка

>>> T

('aa', 'bb', 'cc', 'dd')

>>> sorted(T) # Сортировка встроенной функцией sorted

['aa', 'bb', 'cc', 'dd']

В действительности функции list и tuple создают новый объект (т.е. объект преобразования не изменяется), но благодаря им создается эффект преобразования.

Использование генератора списков и кортежа:

>>> T = (1, 2, 3, 4)

>>>

>>> L = [x+20 for x in T]

>>> L

[21, 22, 23, 24]

Генераторы списков могут использоваться для обхода содержимого любых объектов последовательностей. И даже могут читать файлы, которые поддерживают чтение строки за строкой.

Методы index и count, подобно одноименным методам списков:

>>> T = (1, 2, 3, 2, 4, 2)

>>> T.index(2) # Первое вхождение 2 находится по индексу 1

1

>>> T.index(2, 2) # Следующее вхождение 2 за позицией 2 (за индексом 2)

3

>>> T = (1, 2, 2, 4, 5, 2) # Следующее вхождение 2 за позицией 2 (новый кортеж)

>>> T.index(2, 2)

2

>>> T = (1, 2, 3, 4, 5, 2) # Следующее вхождение 2 за позицией 2 (еще другой кортеж)

>>> T.index(2, 2)

5

>>> T.count(2) # Определить количество двоек в кортеже

3

Кстати, неизменяемость относится только к самому кортежу, но не к объектам внутри него:

>>> T = (1, [2, 3], 4)

>>> T[1] = 'spam' # Нельзя изменить сам кортеж

...

TypeError: 'tuple' object does not support item assignment

>>> T[1][0] = 'spam' # Зато можно изменить вложенный изменяемый объект внутри кортежа

>>>

>>> T

(1, ['spam', 3], 4)

**Зачем нужны кортежи если есть списки.**

Автор языка Питон, математик, рассматривал кортежи как простые ассоциации объектов, а списки - как изменяемые структуры данных.

Сейчас мы видим, что неизменяемость кортежей обеспечивает своего рода поддержку целостности.

Тем самым кортежи играют роль объявлений "констант", которые есть у других ЯП.

К тмоу же кортежи можно использовать в качестве ключей словаря (а списки нет), например матрицы из главы 8.

Некоторые встроенные операции могут требовать и предполагать использование кортежей, а не списков.

**Файлы.** стр. 289.

Файл - именованная область постоянной памяти в вашем компьютере, коорым управляет операционная система.

Последний всртоенный тип объектов Питона, котоырй мы исследуем, обеспечивает возможность доступа к этим файлам из программ на языке Питон.

Встроенная функция open() создает объект файла, который обеспечивает связь с файлом размещенным на ПК. После вызова функции open можно выполнять операции чтения и записи во внешний файл.

Объекты файлов выглядят необычно, они не числа, ни последовательности, ни отображения - для задач работы с файлами они предлставляют только методы.

Большинство методов файлов связаны с выполнением операций ввода-вывода во внешние файлы, ассоциированные с объектом. Существуют также методы, которые позволяют переходить на позицию в файле, выталкивать на диск буферывывода и т.п.

**Таблица 9.2 Часто используемые операции над файлами.** стр. 289

**Открытие файлов.**

Чтобы открыть файл программ должна вызвать функцию open, передав ей имя внешнего файла и режим работы.

Обычно используется режим 'r' (по умолчанию), 'w' - когда файл открывается для записи, 'a' - когда файл открывается на запись в конец.

- Добавление символа b в строку режима означает работу с двоичными данными (при этом Питон 3.0 отключается интерпретация символов конца строки и кодирование символов Юникода).

- Добавление символа + означает, что файл открывается для чтения и для записи (т.е. мы можем читать и записывать данные в один и тот же файл).

Оба аргумента функции open должны быть строками. Крмое того open может принимать третий аргумент, управляющий буферизацией выводимых данных - 0 означает, что вы. инф. не будет буферизироваться (т.е. будет записываться во внеш. файл сразу, в момен твызова метода записи).

Имя файла может включать платформозависимые префиксы абсолютного или относительного пути к файлу.

**Использование файлов.**

Содержимое файла в программах Питон принимает форму строк.

- Для чтения строк лучше использовать итераторы файлов. Это делается итератором (он есть у файлов), который автоматически читает строку за строкой в контексте цикла for.

- Содежимое файлов находится в строках, а не объектах. Обратим внимание, что данные получаемые из файла, всегда попадают в сценарий в виде строки, поэтому нам необходимо будет выполнять преобразование данных в другие типы Питона, если эт форма записи нам не будет подходить в каком-то сценарии программы. Также, при записи данных в файл, нам нужно будет передавать методам уже сформированные строки (интерпретатор сам не преобразует). Поэтому, при работе с файлами нам пригодятся инстурменты преобразования данных из строк в числовое и наоборот (int, float, str, выражения форматирования строк и метод format). Кроме того в Питон есть доп. библиотечные инструменты, для работы с универсальным объектом хранилища данных (напр. модуль pickle) и обработки упакованных двоичных данных в файлах (напр. модуль struct).

- Вызов метода close является необязательным. Можно и не вызывать close, но в крупных сценариях его рекомендуется использовать. И вообще, привычку использовать close вручную можно только поприветствовать.

- Файлы обеспечивают буферизацию ввода-вывода и позволяют производить позиционирование в файле. В момент явного закрытия файлов, освобождаются ресурсы ОС и выталкиваются выходные буферы. Ввыод в файлы делается через промежуточные буферы. Можно отключить буферизацию, но это может привести к снижению производительности операций ввода-вывода. Метод seek - позволяет сценариям управлять позицией чтения и записи.

**Файлы в действии.** стр. 292

Рассмотрим примеры, открываем новый файл для записи, записываем две строки и закрываем; далее открываем файл и читаем строки. Затем обратим внимание на метод readline:

>>> file = open('file.txt', 'w') # Открывает файл (создает/очищает)

>>> file.write('Hello text file\n') # Записываем первую строку текста

16

>>> file.write('Goodbye text file\n') # Вторая строка

18

>>> file.close() # Выталкивает выходные буферы на диск

>>> file = open('file.txt') # Открывает файл: 'r' - по умолчанию

>>> file.readline() # Читает строку

'Hello text file\n'

>>> file.readline()

'Goodbye text file\n'

>>> file.readline() # Пустая строка: конец файла

''

Обратим внимание на метод readline - в последней строке сценария он возвратил нам пустую строку - именно так методы файлов в Питоне сообщают о том, что достигнут конец файла. Действительно пустая строка в файле (напр. пустая строка между абзацами) будет содержать единственный символ новой строки.

Метод write возвращает количество записанных символов. К тому же методы записи добавляют символ конца строки, поэтому приходится добавлять его самим вручную.

Если нам нужно правильно интерпретировать символы конца строки, то используем метод:

>>> open('file.txt').read() # Прочитать файл целиком в строку

'Hello text file\nGoodbye text file\n'

>>>

>>> print(open('file.txt').read()) # Более дружественная форма отображения

Hello text file

Goodbye text file

Чтобы просмотреть файл строка за строкой, лучший выбор - итератор:

>>> for line in open('file.txt'): # Используем итератор

print(line, end='') # Вместо line можно любое слово или даже символ

Hello text file

Goodbye text file

- Итератор это компактный и эффективный способ.

**Текстовые и двоичные файлы в Питон 3.0**

Сейчас мы работали с текстовыми файлами. В Питоне 3.0 тип файла определяется вторым аргументом функции open - символ 'b' означает binary (двоичный).

- Текстовые файлы идут в виде обычных строк типа str, с автоматическим кодированием/декодированием символов Юникода, с интерпретацией символов конца строки;

- Двоичные файлы идут в виде строк типа bytes и передаются программе без изменений.

Мы не можем открывать двоичные данные в текстовом режиме, скорее всего это завершится ошибкой.

Когда выполняется операция чтения двоичных данных, она возвращает объект типа bytes - последовательность коротких целых чисел, представляющие абсолютные значения байтов (которые могут соотв-ть символам, а могут и не соотв-ть), которые очень близко напоминают обычную строку.

>>> data = open('text.bin', 'rb').read() # Открываем двоичный файл для чтения

>>> data # Строка байт хранит двоичные данные

b'spam\n'

>>> data[1:3] # Ведет себя как строка

b'pa'

>>> data[1:3][0] # Но в действительности хранит 8-битные целые числа

112

>>> bin(data[1:3][0]) # Функция bin() в Питон 3.0

'0b1110000'

- у меня не было настоящего двоичного файла пришлось самому создать файл с расширением \*.bin и записать в него строку - spam.

Двоичные файлы не выполняют преобразование символов конца строки.

**Сохранение и интерпретация объектов языка Питон в файлах.**

Рассмотрим запись различных объектов в текстовый файл, с преобразованием в строки, конечно же:

>>> X, Y, Z = 43, 44, 45 # Объекты Питон должны записываться в файл только в виде строк

>>> S = 'Spam'

>>> D = {'a':1, 'b':2}

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

>>>

>>> F = open('datafile.txt', 'w') # Создаем файл для записи

>>>

>>> F.write(S + '\n') # Завершаем строки символом \n

5

>>> F.write('%s,%s,%s\n' % (X, Y, Z)) # Преобразуем числа в строки

9

>>> F.write(str(L) + '$' + str(D) + '\n') # Преобразуем и разделяем символом $

27

>>> F.close()

А теперь посмотрим содержимое нашего нового файла:

>>> chars = open('datafile.txt').read() # Отображение строки в неформатированном виде

>>> chars

"Spam\n43,44,45\n[1, 2, 3]${'b': 2, 'a': 1}\n"

>>> print(chars) # Удобочитаемое представление файла

Spam

43,44,45

[1, 2, 3]${'b': 2, 'a': 1}

Теперь нам нужно выполнить обратные преобразования, т.к. интерпретатор Питона не преобразует строки в другие типы автоматически:

>>> F = open('datafile.txt') # Открываем файл снова

>>> line = F.readline() # Прочитаем одну строку

>>> line

'Spam\n'

>>> line.rstrip() # Удалим символ конца строки

'Spam'

- rstrip можно заменить срезом line[:-1] но только в том случае если мы точно знаем, что строки в нашем файле заканчиваются символом \n (иногда его может и не быть, особенно в последней строке).

>>> line = F.readline() # Читаем следующую строку из файла

>>> line # Это строка

'43,44,45\n'

>>> parts = line.split(',') # Разбиваем строку line на подстроки по запятым

>>> parts

['43', '44', '45\n']

- split разбивает строку на части по символами-разделителям (у нас это запятая), в результате получаем список строк.

Теперь нам нужно преобразовать эти строки в целые числа (т.к. они изначально были числами), чтобы можно было прмиенять к ним математические операции:

>>> int(parts[1]) # Преобразовать строку в целое число

44

>>>

>>> numbers = [int(P) for P in parts] # Преобразуем весь список

>>> numbers

[43, 44, 45]

- Заметим! Мы не удаляли символ '\n', потому что int и некоторые другие функции преобразования просто игнорируют символы-разделители окружающие цифры.

Для преобразования списка и словаря можно воспользоваться функцией eval():

>>> line = F.readline()

>>> line

"[1, 2, 3]${'a': 1, 'b': 2}\n"

>>> parts = line.split('$') # Разбить на строки по символу $

>>> parts

['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]

>>>

>>> eval(parts[0]) # Преобразовать строку в объект

[1, 2, 3]

>>> object = [eval(P) for P in parts] # То же самое для всех строк в списке

>>> object

[[1, 2, 3], {'a': 1, 'b': 2}]

**Сохранение объектов Питон с помощью модуля pickle.**

Функция eval, в предыдущем примере, это мощный инструмент. Иногда даже слишком мощный - ведь таким способом злоумышленник может запустить вредоносный код, передав его функции eval.

Если вам нужно извлечь объекты Питон из файлов лучшим решением является модуль pickle из стандартной библиотеки.

Он напоминает суперуниверсальную утилиту форматирования и преобразования.

>>> D = {'a':1, 'b':2}

>>> F = open('datafile.pkl', 'wb')

>>> import pickle

>>> pickle.dump(D, F) # Модуль pickle запишет в файл любой объект

>>> F.close()

>>> F = open('datafile.pkl', 'rb')

>>> E = pickle.load(F) # Загружает любые объекты из файла

>>> E

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

Модуль pickle занимается т.н. сериализацией объектов - преобразование объектов в строку байтов и обратно, почти не требуя от нас никих доп. действий.

В действительности pickle преобразует наш объект в строку и потом обратно, и если мы применим другой режим, то увидим кое-что интересное:

>>> open('datafile.pkl', 'rb').read() # Формат может измениться

b'\x80\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02K\x02u.'

>>> F = open('datafile.pkl', 'rb')

>>> E = pickle.load(F)

>>> E

{'a': 1, 'b': 2} # Словарь, по-прежнему, доступен без изменений

**Замечание!** Мы открываем файл, где хранится сериализованный объект, в двоичном режиме. Это потому что модуль pickle создает и использует объекты типа bytes.

**Сохранение и интерпретация упакованных двоичных данных в файлах.**

В некоторых приложениях приходится работать с упакованными двоичными данными, созданными, напр. программами на С.

Стандартная библиотека Питон включает инструмент для этого - модуль struct, который сохраняет и упаковывает двоичные данные.

Чтобы создать файл с упакованными двоич. данными откроем его в режиме 'wb' (write binary - запись двоичных данных) и передадим модулю struct строку формата и некотоырй объект Питон.

Пример: строка формата, котоаря определяет пакет данных - 4-байтовое целое число, 4-символьную строку и 2-байтовое целое число, причем все представлены в формате big-endian - в порядке следования байтов "от старшего к младшему" (есть и другие спецификаторы форматов):

>>> F = open('data.bin', 'wb') # Открываем файл для записи в двоичном режиме

>>> import struct

>>> data = struct.pack('>i4sh', 7, b'spam', 8) # Пакет двоичных данных

>>> data

b'\x00\x00\x00\x07spam\x00\x08'

>>> F.write(data) # Записываем строку байтов

10

>>> F.close()

**- Внимание!** В книге строка данных выглядела так - data = struct.pack('>i4sh', 7, 'spam', 8) - в ней НЕ БЫЛО тегов 'b' - это была строка для Питон 2.х, а для Питон 3.4 нам нужно было наши текстовые строки обернуть в теги 'b'.

В книге упустили этот момент.

- И, я так понимаю, '>i4sh' - это спецификатор формата.

Ну а теперь нам нужно прочитать строку bytes (у нас получилаьс экранированная последовательность шестнадцатеричных значений), которую создал Питон:

>>> F = open('data.bin', 'rb')

>>> F

<\_io.BufferedReader name='data.bin'>

>>> data = F.read()

>>> data

b'\x00\x00\x00\x07spam\x00\x08'

>>> value = struct.unpack('>i4sh', data)

>>> value

(7, b'spam', 8)

Режимы доступа к двоичным файлам 'wb' и 'rb' могут использоваться для обработки простейших двоичных файлов, таких как изображения или аудиофайлы, без необходимости выполнять распаковку их содержимого.

**Менеджеры контекста файлов.**

Менеджеры контекста, в основном, применяются для обработки исключений, и также могут обертывать программный код, выполняющий операции с файлами, доп. слоем логики, чтобы, при выходе за блок инструкций менеджера, файл был закрыт автоматически:

>>> with open(r'C:\misc\data.txt') as myfile:

for line in myfile:

...операции над строкой line...

Аналогочиную функциональность предоставляет конструкция try/finally, но с более избыточным кодом:

>>> myfile = open(r'C:\misc\data.txt')

try:

for line in myfile:

...операции над строкой line...

finally:

myfile.close()

**Другие инструменты для работы с файлами.**

Функция seek переустанавливает текущую позицию в файле (для след. операции чтения и записи), функция flush принудительно выталкивает содержимое выходных буферов на диск и т.д.

Функция open и объекты файлов явл-ся основным интерфейсом к внешним файлам. Назовем и другие инструменты:

- Стандартные потоки ввода-вывода. Это объекты уже открытых файлов в модуле sys, такие как sys.stdout (см. далее в главах "Инструкция print").

- Дескрипторы файлов в модуле os. Целочисленные дескрипторы файлов, обеспечивающие поддержку низкоуровневых операций, таких как блокировка файлов.

Дескриптор файла - это целое число без знака, с помощью которого процесс обращается к открытому файлу.

Ко всем потокам ввода-вывода (которые могут быть связаны как с файлами, так и с каталогами, сокетами и FIFO) можно получить доступ через так называемые файловые дескрипторы. Файловый дескриптор — это неотрицательное целое число. Когда создается новый поток ввода-вывода, ядро возвращает процессу, создавшему поток ввода-вывода, его файловый дескриптор.

В операционной системе Windows для получения дескриптора файла его необходимо открыть функцией CreateFile. Затем полученный дескриптор (или handle) можно использовать для работы с этим файлом. Если это драйвер, то дескриптор будет использоваться в функции DeviceIoControl.

По умолчанию Unix-оболочки связывают файловый дескриптор 0 с потоком стандартного ввода процесса (терминал), файловый дескриптор 1 — с потоком стандартного вывода (терминал), и файловый дескриптор 2 — с потоком диагностики (куда обычно выводятся сообщения об ошибках). Это соглашение соблюдается многими Unix-оболочками и многими приложениями — и ни в коем случае не является составной частью ядра.

- Сокеты, каналы и очереди (FIFO). Объекты, по своим характеристикам напоминающие файлы, используемые для синхронизации процессов или организации взаимодействия по сети.

- Файлы с доступом по ключу, известные как хранилища ("shelves"). Используются для хранения объектов языка Питон по ключу.

- Потоки командной оболочки. Такие инструменты, как os.popen и subprocess.Popen, которые поддерживают возможность запуска дочерних процессов и выполнения операций с их стандартными потоками ввода-вывода.

Классы, которые могут использоваться в качестве родительских, для реализации собственных типов файлов, находятся в модуле io.

**Пересмотренный перечень категорий типов.** стр. 301

- Над объектами одной категории можно выполнять одинаковые операции, напр. строки, списки и кортежи - имеют операции для последовательностей (конкатенация и определение длины и т.п.).

- Непосредственное изменение только у изменяемых объектов (списки, словари и множества) - мы не можем изменить непосредственно числа, строки и кортежи.

- Файлы экспортируют только методы, поэтому изменяемость к ним напрямую неприменима - они могут изменяться при записи, но к изменяемости типов Питон это не относится.

- В категорию "Числа" входят все числовые типы: целые, вещественные, комплексные, фиксированной точности и рациональные.

- В категорию "Строки" входят типы str, bytes и unicode (в 2.6). Строковый тип bytearray относится к изменяемым.

- Множества напоминают словари, в которых ключи не имеют значений. Элементы множеств не отображаются на значения и неупорядочены, поэтому их нельзя отнести ни к отображениям, ни к последовательностям.

**Таблица 9.3 Классификация объектов.** стр. 301

***Перегрузка операторов.***

В 6-й части книги мы узнаем, что объекты, реализованные с помощью классов, могут свободно попадать в любую из этих категорий.

Напр. нам нужен новый тип последовательностей, который был бы совместим со встроенным типом последовательностей, в таком случае мы можем описать класс, который перегружает операторы индексирования и конкатенации:

>>> class MySequence:

def \_\_getitem\_\_(self, index):

# вызывается при выполнении операции self[index]

def \_\_add\_\_(self, other):

# вызывается при выполнении операции self + other

и так далее.

Или для создания нового типа, реализовав методы операций непосредственного изменения (например, для выполнения операций присваивания self[index] = value вызывается метод \_\_setitem\_\_).

Или можно реализовать новые объекты на других ЯП, таких как C, в виде расширений. Чтобы определить выбор между множествами операций над числами, последовательностями необходимо будет подставить указатели на функции С в ячейки структуры.

**Гибкость объектов.**

- Списки, словари и кортежи могут хранить объекты любых типов.

- Списки, словари и кортежи допускают произвольную вложенность.

- Списки и словари могут динамически увеличиваться и уменьшаться.

Составные типы объектов в Питон прекрасно подохдят для данных со сложной структурой. Напр. элементами словарей могут быть списки, которые могут содержать кортежи, в которых могут быть словари и т.д. Вложенность может быть произвольной глубины.

Пример, стр. 303. **Рисунок 9.1**

- На этом примере показана схема многоуровневой вложенности на примере списка:

>>> L = ['abc', [(1,2), ([3],4)], 5]

>>>

>>> L[1]

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

>>> L[1][1]

([3], 4)

>>> L[1][1][0]

[3]

>>> L[1][1][0][0]

3

**Ссылки и копии.**

Как мы уже знаем, при выполнении присваивания всегда сохраняется ссылка на объект, а не копия самого объекта. Обычно это нам и нужно, но в случае перекрестных ссылок такой подход может быть нежелательным.

Пример, с **рисунком 9.2** на стр. 304:

>>> x = [1,2,3]

>>> l = ['a', x, 'b']

>>> d = {'x':x, 'y':2}

- Таким нехитрым способом у нас три ссылки на один объект.

Такая ситуация мождет быть не только запутанной, но и опасной для двух вторых объектов, например:

>>> x[1] = 'surprise' # Эта операция изменяет все три ссылки!

>>>

>>> l

['a', [1, 'surprise', 3], 'b']

>>> d

{'y': 2, 'x': [1, 'surprise', 3]}

Ссылки в Питоне - это более высокоуровневый аналог указателей в других ЯП.

Если нам нужна, все-таки, копия объекта, а не ссылки, то мы можем запросить её:

- Выражение извлечения среза с пустыми пределами (L[:]) создает копию последовательности.

- Метод словарей и множеств copy создает копию словаря (D.copy()).

- Некоторые встроенные функции, как list, создают копию списка (list(L)).

- Модуль copy, из стандартной библиотеки, создает полные копии объектов.

Пример, у нас есть список и словарь:

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

>>> D = {'a':1, 'b':2}

- мы можем свзяать с другими переменными копии этих объектов:

>>> A = L[:] # Вместо A = L (или list(L))

>>> B = D.copy() # Вместо B = D (то же относится и к множествам)

>>> A[1] = 'Ni'

>>> B['c'] = 'spam'

>>>

>>> L, D

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

>>>

>>> A, B

([1, 'Ni', 3], {'a': 1, 'c': 'spam', 'b': 2})

- изменяются копии, а не оригиналы!

Также можем теперь правильно написать первый пример:

>>> x = [1,2,3]

>>> l = ['a', x[:], 'b']

>>> d = {'x':x[:], 'y':2}

-теперь l и d не зависят от изменений списка x.

Одно "но" - извлечения среза с пустыми значениями пределов и метод словаря copy создают "поверхностные копии" - они не копируют вложенные структуры данных.

Если нужна полная копия нужно использовать стандартный модуль copy, нужно импортировать - import copy, а затем вставить выражение X = copy.deepcopy(Y), котоырй создаст полную копию объекта Y.

**Сравнение, равенство и истина.** стр. 306.

Любые объекты Питона поддерживают эти операции.

Операция сравнения проверяет все части составных объектов, при вложенных объектах выполняется обход элементов вглубину, рекурсивно, слева направо. Первое же различие определяет результат.

>>> L1 = [1, ('a', 3)]

>>> L2 = [1, ('a', 3)]

>>>

>>> L1 == L2, L1 is L2

(True, False) # Значения объектов равны, но это не один и тот же объект

- Оператор == проверяет равенство значений. Рекурсивное сравнивание всех вложенных объектов.

- Оператор is проверяет идентичность объектов. Проверяет являются ли сравниваемые объекты одним и тем же объектом (т.е. расположены ли они по одному и тому же адресу).

>>> S1 = 'spam'

>>> S2 = 'spam'

>>>

>>> S1 == S2, S1 is S2

(True, True)

- А здесь мы видим как интерпретатор, с целью оптимизации, кэширует и повторно использует короткие строки.

Использования кэширования оправданно, т.к. строки неизменяемые объекты и случайно повлиять на такие объекты мы не можем.

Увеличим длину строки для примера:

>>> S1 = 'spam in string'

>>> S2 = 'spam in string'

>>>

>>> S1 == S2, S1 is S2

(True, False)

- вот теперь всё правильно работает.

Операторы отношений к вложенным структурам также применяются рекурсивно:

>>> L1 = [1, ('a', 3)]

>>> L2 = [1, ('a', 2)]

>>>

>>> L1 < L2, L1 == L2, L1 > L2

(False, False, True) # Кортеж результатов: меньше, равно, больше

- L1 > L2 т.к. вложенное число 3 больше чем 2.

Сравнение типов, некоторые свойства:

- Числа сравниваются по величине;

- Строки сравниваются лексикографически, символ за символом ('abc' < 'ac');

- При сравнении списков и кортежей сравниваются все компоненты слева направо;

- Словари сравниваются как отсортированные списки ( пары ключ/значение) в Питон 2.6, а в Питон 3.0 не поддерживают операторы отношений;

- Попытка сравнить несомоставимые нечисловые типы (1 < 'spam') приведет к ошибке.

**Сравнение словарей в Питон 3.0**

>>> D1 = {'a':1, 'b':2}

>>> D2 = {'a':1, 'b':3}

>>>

>>> D1 == D2

False

>>> D1 < D2

...

TypeError: unorderable types: dict() < dict()

Чтобы сравнить пары значений нам придется вручную стравнить каждую пару используя метод list и функцию sorted:

>>> list(D1.items())

[('a', 1), ('b', 2)]

>>> sorted(D1.items())

[('a', 1), ('b', 2)]

>>> sorted(D1.items()) < sorted(D2.items())

True

>>> sorted(D1.items()) > sorted(D2.items())

False

Впрочем, на практике, для сравнивания словарей используются более эффективные процедуры чем данный способ.

**Смысл понятий "Истина" и "Ложь" в языке Питон.**

True - Истина и False - Ложь.

В Питоне "ложь" представлена целочисленным значением 0, а "истина" - целочисленным значением 1.

Вообще интерпреатор распознает лубую пустую структуру данных как "ложь", а непустую - как "истина".

В широком понимании истина и ложь - это свойства, присущие всем объектам в Питон.

- Числа отличные от 0 являются "истиной";

- Другие объекты явл-ся "истиной" если они не пустые;

- None это тоже "ложь".

**Таблица 9.4. Примеры значений истинности объектов.** стр. 309

Т.о. выражение if X означает if X != ''

**Объект None.**

None всегда расценивается как "ложь". Он похож на NULL в С.

Помните ранее - спискам нельзя присвоить значения отсутствующим элементам. Эту особенность избегают, создавая элементы со значением None:

>>> L = [None] \* 20

>>>

>>> L

[None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None, None]

- можно записать и нули, но предпочтительнее None.

Запомните: None это не значит "неопределенный", это не "ничто", None - это настоящий объект, занимающий определенную область памяти, с зарезервированным именем.

Кроме того этот объект явл-ся значением, возвращаемым функциями по умолчанию.

**Тип bool.** стр. 310.

Тип bool просто усиливает понятия "истина" и "ложь" в языке Питон.

True и False явл-ся просто разновиднстями целых чисел 1 и 0.

- При явном использовании True, False в операциях проверки на истинность они интерпретируются как "истина" и "ложь", которые, в действительности, являются спец. версиями целых чисел 1 и 0.

- Результаты операций проверок также выводятся как слова True, False, а не как значения 1 и 0.

К тому же нам не обязательно использовать только логические типы в инструкциях типа if, - любые объекты могут быть истинными или ложными.

Кроме того в Питон есть встр. функция bool, которая может использ. для проверки логического значения объекта (напр. для проверки истинности объекта, чтобы убедиться, что объект не явл-ся пустым или не равен нулю):

>>> bool(1)

True

>>> bool('spam')

True

>>> bool({})

False

Хотя на практике мы будем редко пользоваться переменными типа bool, т.к. логич. рез-ты автоматич. использ. инструкциями if и др.

**Иерархии типов данных в языке Питон.** стр. 310

В языке Питон ВСЕ элементы являются объектами и могут быть использованы в ваших программах. Например можно передать класс в функцию, присвоить его переменной, заполнить им список или словарь и т.д.

Даже сами типы - это объекты! Тип любого объекта - это объект типа "type".

**Рис. 9.3 Основные типы Питон и их иерархия**. стр. 311

**Объекты типов.**

Сами типы - это разновидности объектов: они являются объектами типа "type".

Т.е. вызов встроенной функции type(X) возвращает объект типа объекта Х. Объекты типов могут использ. для сравнения типов вручную в инструкции if, хотя ручная проверка не приветствуется в Питон.

В Питон 2.2 у каждого базового типа появилось новое встроенное имя (dict, list, str, tuple и др.). Эти имена представляют собой не просто функции преобразования, а настоящие конструкторы объектов.

Модуль types, предоставляет доп. имена типов, не являющиеся встроенными и в нем есмть возможность проверять тип объекта функцией isinstance:

>>> type([1]) == type([]) # Сравнивание с типом другого списка

True

>>> type([1]) == list # Сравнивание с именем типа

True

>>> isinstance([1], list) # Список или объект класса, производного от list

True

>>> import types # В модуле types определены имена других типов

>>> def f(): pass

>>> type(f) == types.FunctionType

True

Т.к. в совр. Питон от типов можно порождать другие дочерние классы, рекомендуется не пренебрегать функцией isinstance.

Далее в книге мы исследуем как функция type(X) и операции проверки типов применяются к экземплярам пользовательских классов.

В Питон 3.0 и для классов "нового стиля" в 2.6 типом экземпляра класса явл-ся сам класс, на основе которого был создан экземпляр.

**Другие типы в Питон.**

Помимо встроенных типов и элементов программ (функции, модули, классы), в составе Питон бывают и десятки других типов объектов, в виде расширений на С или в виде классов Питона - объекты регулярных выражений, файлы DBM, компоненты графического интерфейса, сетевые сокеты и т.д.

Отличие этих типов в том, что встроенные мы создаем спец. синтаксисом (1 - цифра, [1, 2] список, def и lambda для создания функций и т.п.), а другие типы, обычно доступны в модулях стандартной библиотеки, которые нужно импортировать перед использованием (регулярные выражения - модуль re, напр. и т.п.).

**Ловушки встроенных типов.**

В этой части главы мы подводим итоги изучению базовых типов данных.

**Операция присваивания создает ссылку, а не копию.**

Мы должны еще раз запомнить - не забывайте об опасности разделяемых ссылок:

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

>>> M = ['x', L, 'y'] # Встраиваем ссылку из L

>>>

>>> M

['x', [1, 2, 3], 'y']

>>>

>>> L[1] = 0 # В этот момент список М также изменяется!

>>>

>>> M

['x', [1, 0, 3], 'y']

Конечно же мы помним - чтобы избежать опасности таких ссылок, делайте копию объекта и используйте её вместо оригинального объекта:

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

>>> M = ['x', L[:], 'y']

>>>

>>> L[1] = 0

>>> L

[1, 0, 3]

>>> M

['x', [1, 2, 3], 'y']

**Операция повторения добавляет один уровень вложенности.**

Повторение последовательности добавляет её саму к себе заданное число раз, но с вложеннми последовательностями эффект может быть не таким как ожидали:

>>> L = [4, 5, 6]

>>> X = L \* 4 # То же, что и [4, 5, 6] + [4, 5, 6] + ...

>>> Y = [L] \* 4 # То же, что и [L] + [L] + ... = [L, L, ...]

>>>

>>> X

[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]

>>> Y

[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]

- мы видим разницу между двумя, казалось бы, идентичными повторениями. Т.к. во втором случае L является вложенным списком, то в Y попадают ссылки на оригинальный список.

И здесь появляется связь с предыдущим разделом:

>>> L[1] = 0 # Воздействует на Y, но не на Х

>>>

>>> X

[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]

>>> Y

[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]

- в таких случаях можно воспользоваться созданием копии нужного объекта, как в прошлом разделе.

**Избегайте создания циклических структур данных.**

Если объект-коллекция содержит ссылку на себя, он называется - циклическим объектом.

>>> L = ['spam']

>>> L.append(L)

>>> L

['spam', [...]]

- когда интепретатор находит циклическую ссылку он выводит [...], чтобы не попасть в бесконечный цикл.

Старайтесь избегать циклических ссылок, если же они очень нужны в коде, то предусматривайте их обработку, чтобы не уронить программу.

**Неизменяемые типы не могут изменяться непосредственно.**

Изменить такой объект можно только создав новый с помощью операций среза, конкатенации или др., если нужно то присвойте это значение прежней переменной:

>>> T = (1, 2, 3)

>>>

>>> T[2] = 4

...

TypeError: 'tuple' object does not support item assignment

>>>

>>> T = T[:2] + (4,)

>>>

>>> T

(1, 2, 4)

**Часть 3. Инструкции и синтаксис.** стр. 321

**Глава 10. Введение в инструкции языка Питон.** стр. 323

Инструкции - это то, что вы пишите, чтобы сообщить интерпретатору, какие действия должна выполнять ваша программа.

Если программа "выполняет какие-то действия", то инструкции - это способ указать, какие именно действия должна выполнять программа.

Питон - это процедурный язык программирования, основанный на использовании инструкций, комбинируя инструкции, вы задаете процедуру, которую выполняет интерпретатор в соответствии с целями программы.

**Структура программы на языке Питон.**

Иерархия:

1. Программы делятся на модули.

2. Модули содержат инструкции.

3. Инструкции состоят из выражений.

4. Выражения создают и обрабатывают объекты.

**Инструкции в языке Питон.**

**Таблица 10.1 Инструкции языка Питон.** стр. 324

Особенности инструкций таблицы в Питон 3.0:

- Инструкции присваивания принимают разные синтаксические формы, которые будут описаны далее;

- В версии 3.0 print не явл-ся ни зарез-м словом, ни инструкцией - это функция. Практически всегда выполняется как инструкция (т.е. занимает отдельную стркоу в программе);

- Начиная с версии 2.5 yield явл-ся выражением, а не инструкцией. Обычно оно занимает отдельную строку и поэтмоу включено в таблицу. Иногда в сценариях вып-ся присваивание этой инструкции или извлечение рез-та из нее. В отличие от print, имя yield явл-ся зарезервированным словом.

Особенности в Питон 2.6:

- В 2.6 инструкция nonlocal недоступна. Её действие выполняется другими способами;

- В 2.6 print не функция, а инструкция со своим синтаксисом;

- exec в 2.6 также явл-ся инструкцией со своим синтаксисом;

- В 2.5 инструкции try/except и try/finally были объеденены, теперь мы можем использовать предложения except и finally одновременно, в одной инструкции try;

- В 2.5 инструкция with/as была недоступна, если в программный код не включалась инструкция from \_\_future\_\_import with\_statement.

**История о двух if.**

Для начала посмотрим на синтаксис Питона - как НЕ надо писать программный код.

Рассмотрим инструкцию на С:

if (x > y) {

x = 1;

y = 2;

}

- это могли быть и C, C++, Java, JavaScript или Perl.

Теперь взглянем на эквивалент Питона:

if x > y:

x = 1

y = 2

Первое - инструкция на Питоне компактнее, точнее в ней меньше синтаксических элементов. Питон - язык сценариев, так, что это свойство закономерно, краткость - сестра таланта.

Если точнее, то Питон один элемент добавляет, а три ликвидирует.

**Что добавляет язык Питон.**

Один из основных синтаксических элементов Питона - символ двоеточия (:). Все составные инструкции в Питоне (т.е. инструкции с вложенными инструкциями) записываются по одному шаблону:

Основная инструкция:

Вложенный блок инструкций

- двоеточие здесь - обязательный элемент (отсутствие двоеточия - частая ошибка начинающих программистов).

**Что Питон устраняет.**

- **Круглые скобки не обязательны**. Поясним.

Первый элемент:

if (x > y)

В языке Питон можно опустить круглые скобки, на работу инструкции это не повлияет:

if x > y

- конечно скобки можно и оставить, ошибки не будет, но старайтесь писать код так, чтобы не ставить их везде без разбору - ведь синтаксис Питона как раз и направлен на упрощения писания кода.

- **Конец строки является концом инструкции**. Поясним.

Пример в С:

x = 1;

В Питоне конец строки автоматически считается концом инструкции, стоящей в этой строке:

x = 1

- и снова - вы можете ставить точку с запятой, ошибки в коде не будет, но старайтесь так не писать код, Питон настроен как раз на то, чтобы не пользоваться точкой с запятой.

- **Конец отступа - это конец блока**. Поясним.

Мы не вводим ничего специально в код, чтобы синтаксически пометить начало и конец вложенного блока кода. Нам не нужно вставлять begin/end, then/endif или фигурные скобки вокруг вложенных блоков, как это делается в С-подобных языках:

if (x > y) {

x = 1;

y = 2;

}

- от автора - известная программистская шутка - это не bug, это feature (это не баг, это фича).

Для этих целей, в Питоне, используются отступы, все инструкции из одного блока оформляются с одинаковыми отступами:

if x > y:

x = 1

y = 2

- Отступы можно выделять пробелом или табуляцией, интерпретатору не важно, и можно любую величину отступов. Важно лишь, чтобы отступы инструкций одного блока были одинаковы (ну это и так понятно).

**Почему отступы являются частью синтаксиса?**

Это сделано для однородности и удобочитаемости кода. Мы должны выстраивать свой код вертикально, выравнивая его в соответствии с логической струтурой.

Удобочитаемость способствует простоте сопровожедния кода. К тому же отступы важны для читаемости кода и в других ЯП, просто в Питоне отступы стали частью синтаксиса.

Представьте себе программу на С, над которой трудились несколько программистов и каждый любил свои размеры отступов (и все типы отступов есть в одном тексте программы), а при этом еще есть несколько вариантов вставки фигурных скобок.

Не позавидуешь программисту, который будет это все разбирать...

Еще пример на С:

if (x)

if (y)

statement1

else

statement2

- как думаете, к какому if относится else? Оказывается к вложенному оператору if (y) хотя выглядит наоборот. Это классическая ловушка языка С.

Пример на Питоне:

if x:

if y:

statement1

else

statement2

- Здесь мы ясно понимаем, что else относится к if x.

Питон - это язык подобный типу WYSIWYG (What You See Is What You Get - что видишь, то и получешь), т.е. порядок оформлени кода определяет порядок его выполнения.

Даже если ЯП не требует точных отступов важно понимать какое значение имеет выравнивание для удобочитаемости и высого качества программного кода.

Поэтому такой синтаксис Питона выглядит как достоинство этого языка.

Для оформления отступов используется или пробелы или табуляцию, старайтесь не совмещать их вместе, т.к. в Питон 3.0 это является даже ошибкой (впрочем придерживайтесь этого правила и в других ЯП).

**Несколько специальных случаев.** стр. 332

- Конец строки явл-ся концом инструкции, расположенной в этой строке (точка с запятой не обязательны);

- Вложенные инструкции объединяются в блоки по величине отступов (без фигурных скобок).

Однако есть и специальные правила.

- В одной строке можно писать более одной инструкции, разделив их точкой с запятой:

a = 1; b = 2; print(a + b)

>>> a = '11'

>>> b = 'abc'

>>> a.isdigit(); b.isdigit()

True

False

- в этом случае точка с запятой это - разделители инструкций.

И конечно же составные инструкции нужно писать раздельно (представляете целую программу из одной строки... это был бы ужас).

- Обратное правило - допускается записывать одну инструкцию в нескольких строках. В таких случаяз можно использовать скобки () квадратные скобки [] b и фигурные {}, интерпретатор будет читать код пока не дойдет до закрывающей скобки.

mlist = [111,

222,

333]

- Так можно записывать многие типы объектов в Питон (помните, так мы создавали матрицу, можно и кортежи и словари и т.п.).

Круглыми скобками можно также выделять выражения:

X = (A + B +

C + D)

Такой прием можно применять и к составным инструкциям:

if (A == 1 and

B == 2 and

c == 3):

print('spam' \* 3)

Также можем использовать символ обратного слеша:

X = A + B + \

C + D

- но это уже устаревший метод, так писать код не рекомендуется.

**Специальный случай оформления блока.**

Тело составной инструкции может располагаться в той же строке, что и основная инструкция, после двоеточия:

if x > y: print(x)

Но это правило работает только если после двоеточия идет простая инструкция (и желательно одна) и нет других составный инструкций в теле основной инструкции.

Крупные инструкции, по-прежнему, должны записываться в отдельных строках.

Далее мы рассмотрим некоторые короткие примеры.

**Короткий пример: интерактивные циклы.**

В моей домашней папке создал файл test.py в нем у нас будет цикл, выполняющий операции чтения/вычисления/вывода:

while True:

reply = input('Enter text:')

if reply == 'stop': break

print(reply.upper())

- **Замечание!** в Питон 2.6 он не работает, нужно было использовать инструкцию raw\_input вместо input, пришлось запустить Питон 3.4, тогда всё пошло как надо:

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

Enter text:hello

HELLO

Enter text:1

1

Enter text:123

123

Enter text:alex

ALEX

Enter text:stop

- работа цикла понятна, рассмотрим подробности.

- Цикл while, наиболее универсальный, эа этим словом следует условное выражение, результат которого интерпретируется как истина или ложь. Далее следует блок программного кода, который продолжает выполняться пока условие истинно (а у нас оно всегда истино, потмоу что оно True - само "истина");

- Функция input используется здесь как универсальное средство получения ввода с клавиатуры - и также она выводит подсказку (это необязательный строковй аргумент) и возвращает введнный пользователем ответ в виде строки;

- Однострочная инструкция if, можно и в несколько строк, но т.к. как у нас короткая простая инструкция в теле if мы смогли сэкономить одну строку;

- И наконец, для немедленного выхода из цикла используется инструкция break - она выполняет выход за пределы инструкции цикла и программа продолжает работу с первой инструкции, которая расположена за циклом. Без break наш цикл работал бы вечно, т.к. наше условие всегда истинно (True).

Также отметим, что все строки цикла while нах-ся на одном отступе, т.е. сразу видно тело цикла.

**Математическая обработка данных пользователя.**

Займемся математическими исследованиями языка Питон:

>>> reply = '20'

>>> reply \*\* 2

...

TypeError: unsupported operand type(s) for \*\* or pow(): 'str' and 'int'

Получили ошибку, потому что ввод пользователя всегда передается сценарию в виде строки, а преобразование типов в выражениях выполняется только если объекты являются числами.

Т.о. мы не можем возвести строку цифр в степень, не преобразовав её в целое число вручную:

>>> int(reply) \*\* 2

400

А теперь переделаем наш прошлый цикл для выполнения математических действий, создадим новый файл - tnt.py:

while True:

reply = input('Enter text:')

if reply == 'stop': break

print(int(reply) \*\* 2)

print('Bye!')

- результаты выполнения этой программы:

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

Enter text:2

4

Enter text:11

121

Enter text:stop

Bye!

user@home:~$

Работа этого сценария похожа на предыдущий, кроме того, что он выполняет математические операции и в конце теперь нам выводится "Bye!" в момент завершения работы сценария.

**Обработка ошибок проверкой ввода.**

А теперь посмотрим на работу прошлого сценария, если пользователь вдруг введет не число, а, например, строку:

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

Enter text:2

4

Enter text:11

121

Enter text:xxx

...

ValueError: invalid literal for int() with base 10: 'xxx'

user@home:~$

Функция int возбуждает исключения, когда возникает ошибка и сценарий автоматически завершается.

**Особенность!** Во время такой ошибки я не заметил, что сценарий завершился и набрал в терминале "stop" и у меня выключился компьютер! И закрыл все открытые приложения!!!

Не повторять таких ошибок, всегда смотреть внимательно в терминал!

Итак, если нам нужно обеспечить устойчивость сценария нужно проверить содержимое строки с помощью строкового метода isdigit:

>>> S = '123'

>>> T = 'xxx'

>>> S.isdigit(), T.isdigit()

(True, False)

И теперь мы можем написать устойчивый сценарий, с нужными проверками:

while True:

reply = input('Enter text:')

if reply == 'stop':

break

elif not reply.isdigit():

print('Bad!' \* 5)

else:

print(int(reply) \*\* 2)

print('Bye!')

И как видим по его работе - всё выполняется правильно:

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

Enter text:2

4

Enter text:11

121

Enter text:abc

Bad!Bad!Bad!Bad!Bad!

Enter text:qwerty

Bad!Bad!Bad!Bad!Bad!

Enter text:13

169

Enter text:stop

Bye!

user@home:~$

- Отлично!

Как видим, в своей полной форме за if следует выражение проверки условия и вложенный блок кода, один или более необязательных проверок elif ("else if") и соответствующих им вложенных блоков кода и необязательная часть else, со своим блоком кода, который выполняется при несоблюдении условия if.

Интерпретатор выполняет первый блок кода, для которого проверка дает в результате истину, проходя инструкцию сверху вниз, либо часть else, если все проверки дали в результате ложь.

**Обработка ошибок с помощью инструкции try. стр. 338**

Инструкция try - более универсальный способ перехватывания и обработки ошибок. Ознакомимся предварительно:

while True:

reply = input('Enter text:')

if reply == 'stop': break

try:

num = int(reply)

except:

print('Bad!' \* 5)

else:

print(int(reply) \*\* 2)

print('Bye!')

Наш цикл работает в прежнем режиме:

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

Enter text:2

4

Enter text:11

121

Enter text:abc

Bad!Bad!Bad!Bad!Bad!

Enter text:3

9

Enter text:stop

Bye!

user@home:~$

- Наличие ошибки проверяет код, который предполагает, что преобразование будет выполнено и выполнит обработку исключения, если такое преобразование невозможно.

Схема try:

- за try идет основной блок кода (действие, которое мы пытаемся выполнить) - у нас это выражение num = int(reply), которое выполняется если reply это число и, которое выдает ошибку, если reply не число, а, например, строка;

- далее идет except, где располагается программный код обработки исключений;

- далее идет else, с кодом, который выполняется если в try исключений не возникло.

Кстати, обратите внимание на все отступы, здесь else явл-ся частью инструкции try, а не if.

**Три уровня вложенности программного кода.**

Представим, что мы хотим ввести проверку на величину введенного числа:

while True:

reply = input('Enter text:')

if reply == 'stop':

break

elif not reply.isdigit():

print('Bad!' \* 5)

else:

num = int(reply)

if num < 20:

print('low')

else:

print(num \*\* 2)

print('Bye!')

И работает наш код точно так же как и прежде:

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

Enter text:25

625

Enter text:33

1089

Enter text:12

low

Enter text:15

low

Enter text:abc

Bad!Bad!Bad!Bad!Bad!

Enter text:stop

Bye!

user@home:~$

- Заметим, что здесь у нас добавился вложенный if в выражении else, которое относится к другому if. Все отмечено соответствующими отступами. Ну и конечно числа проверяются на величину, меньше 20 отсекаются ("low").

**Глава 11. Присваивание, выражения и print.** стр. 342

Теперь мы приступаем к более подробному изучению конкретных инструкций Питона.

**Инструкции присваивания.**

Ранее мы использовали эту инструкцию для назначения имен объектам. В канонической форме Цель инструкции присваивания записывается слева от знака равно, а Объект, который присваивается - справа.

Особенности присваивания:

- Инструкция присваивания создает ссылку на объект. Как уже говорилось - в Питон присваивание сохраняет ссылки на объекты в переменных или в элементах структур данных. Они всегда создают ссылки на объекты и никогда не создают копии. Переменные больше напоминают указатели, чем области хранения данных;

- Переменные создаются при первом присваивании. Поэтому в Питон не нужно заранее объявлять переменные как в других ЯП. Иногда, при прсиваивании создаются элементы структур данных (словарей, атрибуты объектов). Когда имя переменной встречается в выражении оно замещается объектом, на который ссылается переменная;

- Прежде чем использовать переменную, ей должно быть присвоено значение. Нельзя использовать переменную, которой не присвоено значение - возникает ошибка.

- Некоторые инструкции неявно тоже выполняют операцию присваивания. Напр. импорт модуля, определение функции или класса, переменная в цикле for и передача аргументов функции - это неявное присваивание.

**Формы инструкции присваивания.**

**Таблица 11.1 Формы инструкции присваивания.**

**Присваивание кортежей и списков:**

>>> spam, ham = 'yum', 'YUM'

>>> spam

'yum'

>>> ham

'YUM'

- Внутри интерпретатор Питон сначала создает элементы кортежа справа, поэтому часто эта операция называется - распаковыванием кортежа.

>>> [spam, ham] = ['YUM', 'yum']

>>> spam

'YUM'

>>> [spam]

['YUM']

>>> [ham]

['yum']

- как видим теперь у нас изменились значения элементов spam и ham, они переопределились второй инструкций [spam, ham] = ['YUM', 'yum'], теперь по этим итменам следуют элементы списка.

**Присваивание последовательностей:**

Любая последовательность имен может быть связана с любой последовательностью значений. даже можно смешивать разные типы, пример ниже.

>>> a, b, c, d = 'spam'

>>> a

's'

>>> b

'p'

>>> c

'a'

>>> d

'm'

**Расширенное распаковывание последовательностей:**

Питон 3.0 позволяет более гибко выбирать присваиваемые фрагменты, пример ниже.

>>> a, \*b = 'spam'

>>> a

's'

>>> b

['p', 'a', 'm']

- а присвоен первый символ из строки справа, а b - оставшаяся часть строки. Напоминает извлечение среза.

**Групповое присваивание одного значения:**

>>> spam = ham = 'lunch' # То же, что: ham = 'lunch' и spam = ham

>>> spam

'lunch'

>>> ham

'lunch'

>>> spam is ham

True

- результат тот же как если бы были выполнены две инструкции, поскольку ham здесь интерпретируется как оригинальный объект-строка (т.е. не отдельная копия этого объекта):

>>> spam = 'pum'

>>> ham

'lunch'

>>> spam

'pum'

>>> ham = 'luntik'

>>> spam

'pum'

>>> ham

'luntik'

Комбинированное присваивание - это краткая форма, котоаря объединяет в себе выражение и присваивание:

>>> spam += 42 # То же, что: spam = spam + 42

- в Питон для каждого двухместного оператора есть своя комбинировання инструкция присваивания.

**Присваивание последовательностей.**

>>> nudge = 1

>>> wink = 2

>>> A, B = nudge, wink # Присваивание кортежей

>>> A, B

(1, 2)

>>> [C, D] = [nudge, wink] # Присваивание списков

>>> C, D

(1, 2)

- Заметим, что в третьей строке у нас присутствуют два кортежа, просто мы опустили круглые скобки и поэтому не так заметно. Интерпретатор сопоставляет значения элементов кортежа справа с переменными кортежа слева и вып-т присваивание.

Т.к. в процессе выполнения инструкции интерпретатор создает временный кортеж, где хранятся оригинальные значения переменных справа, данная форма присваивания может исп-ся для реализации Обмена значений переменных без создания временной переменной - кортеж справа автоматически запоминает пред. знач. переменных:

>>> nud = 1

>>> win = 2

>>> nud, win = win, nud # Кортежи: обмен значениями

>>> nud, win # То же, что и T = nud; nud = win; win = T

(2, 1)

Допускается присваивать кортеж кортеж значений списку переменных, стркои символов - кортежу переменных и т.д. В случае элементы свяжутся согласно их позиции:

>>> [a, b, c] = (1, 2, 3)

>>> a, c

(1, 3)

>>> (a, b, c) = "ABC"

>>> a, c

('A', 'C')

- Справа можно указывать не только последовательности, но и любые объекты, которые имеют итерацию по элементам.

**Дополнительные варианты инструкций присваивания последовательностей.** стр. 345

Последовательности должны иметь одно и то же число элементов иначе будет ошибка.

В Питон 3.0 допускается более обощенная форма присваивания, но в 2.х количество элементов справа/слева должно быть равно:

>>> string = 'SPAM'

>>> a, b, c, d = string # Одинаковое число элементов слева/справа

>>> a, d

('S', 'M')

>>>

>>> a, b, c = string # А здесь возникает ошибка

...

ValueError: too many values to unpack (expected 3)

В общем случае нам нужно получить срез, рассмотрим неск-ко способов:

>>> a, b, c = string[0], string[1], string[2:] # Элементы и срез

>>> a, b, c

('S', 'P', 'AM')

>>> a, b, c = list(string[:2]) + [string[2:]] # Срезы и конкатенация

>>> a, b, c

('S', 'P', 'AM')

>>> a, b, = string[:2] # То же самое, только проще

>>> c = string[2:]

>>> a, b, c

('S', 'P', 'AM')

>>> (a, b), c = string[:2], string[2:] # Вложенные последовательности

>>> a, b, c

('S', 'P', 'AM')

Как видим мы можем присваивать даже вложенные последовательности.

>>> ((a, b), c) = ('SP', 'AM')

>>> a, b, c

('S', 'P', 'AM')

- здесь мы присваиваем кортеж из двух строк,

Присваивание вложенных последовательностей используется редко на практике, в частности можно в циклах for:

for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ... # Простое присваивание кортежей

for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ... # Присваивание вложенных кортежей

В Питон 2.6 присв-е влож. кортежей можно использ. в списках аргуемнтов функций (но не в Питон 3.0):

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

f(((1, 2), 3))

Распространенный оборот в Питон - присваивание последовательности целых чисел множеству переменных:

>>> red, green, blue = range(3)

>>> red, blue

(0, 2)

- это эквивалент перечислимых типов в других ЯП.

Функция range() генерирует непрерывный список последовательных чисел (в Питон 3.0 это list(range())):

>>> list(range(3))

[0, 1, 2]

Еще пример с кортежами, - разделение последовательности на начальную и остальную части в циклах:

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

>>> while L:

front, L = L[0], L[1:]

print(front, L)

1 [2, 3, 4]

2 [3, 4]

3 [4]

4 []

# Это пример для Питон 2.6, не забываем!!!

Присваивание кортежа в цикле, можно заменить двумя строка, но удобнее одной:

front = L[0]

L = L[1:]

- кстати, теперь работа цикла стала понятнее. А условие цикла становится False когда наш список становистя пустым.

**Расширенная операция распаковывания последовательностей в Питон 3.0**

В Питон 3.0 у нас появляется возможность написать \*X слева от оператора = и этому имени со звездочкой будут присвоены все элементы, не присвоенные другим переменным, т.н. реализация операций "начало" и "остаток".

**Расширенная операция распаковывания в действии.**

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

>>> a, b, c, d = spam

>>> print(a, b , c, d)

1 2 3 4

>>> a, b = spam # Получаем ошибку

...

ValueError: too many values to unpack (expected 2)

Но теперь, в Питон 3.0, у нас есть волшебная звездочка \*

>>> a, \*b = spam

>>> a

1

>>> b

[2, 3, 4]

>>> \*a, b = spam

>>> a

[1, 2, 3]

>>> b

4

Рассмотрим пример с звездочкой посредине:

>>> a, \*b, c = spam

>>> a

1

>>> b

[2, 3]

>>> c

4

- переменной со звездочкой присваиваются все элементы последовательности справа, которые остаются после присваивания всем элементам без звездочек.

В любом случае, где бы ни была звездочка ей присваиваются все эелементы, которые остаются после других переменных:

>>> a, b, \*c = spam

>>> a

1

>>> b

2

>>> c

[3, 4]

Посмотрим на распаковывание строки:

>>> a, \*b = 'spam'

>>> a, b

('s', ['p', 'a', 'm'])

>>> a, \*b, c = 'spam'

>>> a, b, c

('s', ['p', 'a'], 'm')

Инструкция присваивания последовательностей всегда возвращает список с множеством элементов, тогда как операция извлечения среза возвращает последовательность того же типа, что и последовательность, из которой извлекается срез:

>>> S = 'spam'

>>> S[0], S[1:]

('s', 'pam')

>>> S[0], S[1:3], S[3]

('s', 'pa', 'm')

Разделение последовательности на начальную и остальную части в циклах в Питон 3.0:

>>> while L:

front, \*L = L

print(front, L)

1 [2, 3, 4]

2 [3, 4]

3 [4]

4 []

**Граничные случаи.**

Первое, - переменной со звездочкой может соответствовать единственный элемент, но ей всегда присваивается список:

>>> spam

[1, 2, 3, 4]

>>> a, b, c, \*d = spam

>>> print(a, b, c, d)

1 2 3 [4]

Второе, - если переменной со звездочкой не остается элементов (все уже присвоены другим переменным), то ей присв-ся пустой список:

>>> a, b, c, d, \*e = spam

>>> print(a, b, c, d, e)

1 2 3 4 []

>>> a, b, \*e, c, d = spam

>>> print(a, b, c, d, e)

1 2 3 4 []

>>> a, b, \*c, e, d = spam

>>> print(a, b, c, d, e)

1 2 [] 4 3

Третье, - ошибкой будет считаться, если указать несколько переменных со звездочкой; если значений недостаточно, а слева не окажется переменной со звездочкой (как и ранее) и если переменная со звездочкой окажется единственной вне послед-ти:

>>> a, \*b, c, \*d = spam

SyntaxError: two starred expressions in assignment

>>> a, b = spam

...

ValueError: too many values to unpack (expected 2)

>>> \*a = spam

SyntaxError: starred assignment target must be in a list or tuple

>>> \*a, = spam

>>> a

[1, 2, 3, 4]

**Полезное удобство.**

Заметим, что расширенная операция присваивания - этовсего лишь удобство, можно сделать то же самое более простыми средставми.

Разбиение последовательности "первый, остаток":

>>> spam

[1, 2, 3, 4]

>>> a, \*b = spam # Первый, остаток

>>> a, b

(1, [2, 3, 4])

>>> a, b = spam[0], spam[1:] # Первый, остаток: традиционная реализация

>>> a, b

(1, [2, 3, 4])

Разбиение последовательности "остаток, последний":

>>> \*a, b = spam

>>> a, b

([1, 2, 3], 4)

>>> a, b = spam[:-1], spam[-1] # Традиционная реализация

>>> a, b

([1, 2, 3], 4)

В общем выбирайте какая реализация вам больше нравится (расширенная распаковка выглядит конечно компактнее и естественнее).

**Использование в циклах for.**

for (a, \*b, c) in [(1,2,3,4), (5,6,7,8)]: ...

- на первом проходе цикла мы получим выражение:

a, \*b, c = (1,2,3,4)

и b получит значение [2,3].

Можно и просто присваивать кортежи:

for (a, b, c) in [(1,2,3), (5,6,7)]: ...

- то же, что и: a, b, c = (1,2,3)

**Групповое присваивание.** стр. 352

Все переменные ,при групп. присваивании, указывают на один и тот же объект. Поэтому такое присваивание лучше всего подходит для неизменяемых объектов:

>>> a = b = 42

>>> b = b + 1

>>> a, b # Числа неизменемые объекты, переменная а не изменилась

(42, 43)

А вот изменяемые представляют опасность в таком случае:

>>> a = b = []

>>> b.append(42)

>>> a, b

([42], [42])

>>> a = b = []

>>> a.append(44)

>>> a, b

([44], [44])

- в обоих случаях изменение переменной влечет за собой изменение второй, вот оно действие разделяемых ссылок.

Избежать можно только отдельным присваиванием:

>>> a = []

>>> b = []

>>> b.append(32)

>>> a, b

([], [32])

**Комбинированные инструкции присваивания.** стр. 353

**Таблица 11.2 Комбинированные инструкции присваивания.**

Такие инструкции существуют для любого двухместного оператора (т.е. оператора, слева и справа от которого располагаются значения участвующие в операции).

Пример:

>>> x = 1

>>> x = x + 1 # Традиционная форма записи

>>> x

2

>>>

>>> x += 1 # Комбинированная

>>> x

3

Такая операция со строками будет конкатенацией:

>>> S = 'spam'

>>> S += 'SPAM'

>>> S

'spamSPAM'

Преимущества комбинированных инструкций присваивания:

- Меньше писать кода.

- Левая часть должна быть получена всего один раз. Нарп. если в инструкции X += Y переменная X это сложное выражение, то в инструкции вида X = X + Y выражение X будет вычисляться два раза.

- Автоматически выбирается оптимальный алгоритм выполнения. Т.е. операции выполняются на самих объектах, без какого-либо копирования.

Еще момент, - комбинированные инструкции могут оптимизировать операции.

Пример с одним элементом списка и методом append():

>>> L = [1, 2]

>>> L = L + [3] # Конкатенация, более медленная операция

>>> L

[1, 2, 3]

>>> L.append(4) # Более быстрая, но изменяет сам объект

>>> L

[1, 2, 3, 4]

И пример с добавлением нескольких элементов и методом extend():

>>> L = L + [5, 6] # Конкатенация, более медленная операция

>>> L

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

>>> L.extend([7, 8]) # Более быстрая, но изменяет сам объект

>>> L

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

Конечно конкатенация менее опасна при работе с разделяемыми ссылками, но методы работают быстрее.

Комбинированные инструкции также работают быстрее, т.к. автоматически вызывается метод extend:

>>> L += [9, 10] # То же, что L.extend([9, 10])

>>> L

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

**Комбинированные инструкции присваивания и разделяемые ссылки.**

Учитывайте, что со списками операция += изменяет сам объект, а конкатенация + создает новый. Последствия могут быть неожиданными, или даже неприятными:

>>> L = [1, 2]

>>> M = L

>>> L = L + [3, 4] # Конкатенация создает новый объект

>>> L, M # Меняется L, но не M

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

>>> L = [1, 2]

>>> M = L

>>> L += [3, 4] # Операция += предполагает вызов метода extend

>>> L, M # Изменяются оба значения

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

- конечно все это имеет значение только для изменяемых обектов (напр. списки и словари).

**Правила именования переменных.** стр. 354

- Синтаксис: (символ подчеркивания или алфавитный символ) + (любое число символов, цифр или символов подчеркивания). Допустимые имена: \_spam, spam и Spam\_1. Недопустимые имена: 1\_Spam, spam$ и @#!.

- Регистр символов в именах имеет значение. SPAM и spam - это две разные переменные.

- Запрещено использовать зарезервированные слова. Если мы захотим создать переменную с именем class, то возникнет ошибка, но можно использовать klass или Class.

**Таблица 11.3 Зарезервированные слова в Питон 3.0** стр. 356

В Питон 2.6 print зарезервированное слово, также как и exec, а nonlocal не является зарезервированным.

Также нежелательно создавать файлы с именами зарезервированных слов, т.к. интерпретатор будет воспринимать их как переменные и выдаст ошибку.

**Соглашения по именованию.**

Соглашения - правила, которые не обязательны, но им обычно следуют на практике. Напр. имена с двумя подчеркиваниями в начале и конце (\_\_name\_\_) имеют особе значения для интерпретатора, поэтому лучше не пользоваться таким наименованием своих переменных.

Список соглашений:

- Имена, начинающиеся с одного символа подчеркивания (\_Х), не импортируются инструкцией from module import \*

- Имена, имеющие два символа подчеркивания в начале и конце (\_\_Х\_\_), являются системными;

- Имена, начинающие с двух символов подчеркивания (\_\_Х), являются локальными ("искаженными") для объемлющего класса;

- Имя, состоящее из единственного символа подчеркивания (\_), хранит результат последнего выражения при работе в интерактивной оболочке.

Еще неск-ко негласных правил, которые обычно выполняют программисты:

- Имена классов обычно начинаются с заглавного символа, а имена модулей - со строчного;

- Имя self, хотя и не зарезервированно, играет особую роль в классах;

- В главе 17 мы познакомимся с крупной категорией имен, называемых - встроенными, т.е. предопределенными, но не зарезервированными (вследствие чего они допускают переопределение, напр. инструкция open = 42 явл-ся вполне допустимой, но иногда можно пожалеть о таких переопределениях).

**Имена не имеют типа, тип - это характеристика объектов.**

Важно четко понимать различие между именами и объектами в Питон.

Объекты - имеют тип (число, строка, список), могут изменяться или нет.

Имена (они же переменные) - всегда явл-ся лишь ссылками на объекты, они не знают об изменяемости или о типе объекта, на который ссылаются в данный момент

>>> x = 0 # Сейчас имя х связывается с целочисленным объектом

>>> x = 'Hello' # А теперь оно представляет строку

>>> x = [1, 2, 3] # А теперь список

Далее мы узнаем, что имена находятся внутри области видимости, которая определяет где эти имена могут использоваться, - место, где выполняется присваивание, определяет, где это имя будет видимо.

Python PEP 8 - официальное руководство по оформлению кода, в нем стандарты оформления кода. Но знайте, что этот документ сложный и субъективный, и не всеми выполняется. Но тем не менее он содержит полезные сведения и можно ознакомиться с его рекомендациями (помните, чт оони не обязательны, а просто рекомендуются).

**Инструкции выражений**. стр. 360

В Питон выражения могут быть инструкциями. Но т.к. они не сохраняют результат вычислений их нужно использовать только, если выражение выполняет полезные действия в виде побочного эффекта.

Обычно используются в двух ситуациях:

- Для вызова функций и методов. Некоторые функции и методы выпол-т огромный объем работ, но не возвращают значения. В других ЯП такие функции иногда наз-ся процедурами. Для сохранения таких значений можно вызывать эти функции в инструкциях выражений;

- Для вывода значений в интерактивной оболочке. Выводы результатов в интерпретаторе IDLE это тоже инструкции выражений, которые играют роль скоращенной версии инструкции print.

**Таблица 11.4 Наиболее часто используемые в Питон инструкции.** стр. 360

spam(eggs, ham) # Вызов функции

spam.ham(eggs) # Вызов метода

spam # Вывод значения переменной в интерактивной оболочке IDLE

print(a, b, c, sep='') # Операция вывода в Питон 3.0

yield x \*\* 2 # Инструкция выражения yield

Инструкция print была преобразована в функцию, которая вызывается в отдельной строке; yield также часто оформляется как инструкция. Действительно, оба случая это инструкции выражений.

Хоть print и вызывается в отдельной строке (как инструкция выражений), тем не менее она возвращает значение, как и любая функция. Она возвращает None - как все функции, которые явно не возвращают какого-либо значимого значения:

>>> x = print('spam') # print - это выражение вызова функции в Питно 3.0

spam

>>> print(x) # может использоваться как инструкция выражения.

None

Хоть выражения могут быть инструкциями сами инструкции не могут быть выражениями. Напр. в Питон не допускается встраивание инструкции присваивания (=) в выражения, это сделано, чтобы избежать ошибок.

**Инструкции выражений и непосредственное изменение объектов.**

Инструкции выражений иногда приводят к ошибкам.

Инст. выр. часто использ. для вызова методов списка, которые изменяют сам список:

>>> L = [1, 2]

>>> L.append(3)

>>> L

[1, 2, 3]

Однако начинающие программисты часто пишут такие операции в виде присваиваний:

>>> L = L.append(4) # append возвращает значение None, но не список

>>> print(L) # и мы теряем наш список L

None

Такие методы списка как append, sort, reverse всегда изменяют сам список, но не возвращают список, который был изменен.

В действительности они возвращают объект None. - Не забывайте о таких ньюансах!

**Операция print.**

В Питон print - это удобный, для кодеров, интерфейс к стандартному потоку вывода.

С технич. стороны эта инструкция преобразует объекты в текстовое представление и, либо посылает рез-т в поток стандартного вывода, или передает другому объекту похожему на файл. Эта инструкция тесно связана с понятиями файлов и потоков в Питон.

- Методы объектов файлов.

В главе 9 мы уже знакомились с методами файлов (вывод текста - file.write(str)). print чем-то похожа, но более специализирована: по умолчанию записывает объекты в поток stdout (с форматированием), а методы файлов записывают строки в произвольные файлы. И print не требует преобразовывать объекты в строковое представление.

- Поток стандартного вывода.

Наз-ся stdout - это просто объект, куда программы выводят текст. Как и потоки стандарт. ввода и стандарт. вывода сообщений об ошибках, поток стандарт. вывода создаются в момент запуска сценария. Поток вывода обычно отображается на окно терминала, где запущена программа Питона, если явно не перенаправлен вывод в файл или конвейер системной командной оболочки.

Т.к. поток stdout доступен из встроен. модуля sys (sys.stdout), можно имитировать поведение print с помощью методов записи в файл, хотя использование print выглядит проще.

В Питон 3.х print стала функцией, которая принимает аргументы, определяющие спец. режимы ввода.

В Питон 2.х print это инструкция, со своим синтаксисом (записывается без круглых скобок).

**Функция print в Питон 3.0** стр. 363

В 3.0 print это инструкция выражения.

Обычно ей вызов оформляется в отдельной строке, потому что она возвращает None. Т.к. в 3.0 print - обычная функция, её вызов сделан в виде стандарт. вызова функций, а спец. режимы - вызовом аргументов.

В 2.6 print не позволяет строки-разделители, приходится конструировать строки заранее.

**Формат вызова (все еще 3.0).**

print([object, ...][, sep=' '][, ens='\n'][, file=sys.stdout])

- Здесь элементы в квадр. скобках явл-ся необязательными, а значения после =, определяют значения по умолчанию.

print - выводит в файл stream один или более объектов в текстовом представлении, разделенных строкой sep и завершает вывод строкой end

Параметры sep, end, file, если нужны, должны передаваться не в виде позиционных, а в виде именвоанных аргументов, их синтаксис "имя=значение" и после объектов, предназначенных для вывода.

**Параметры вывода:**

- sep - строка, которая должна вставляться между объектами при выводе. По умолчанию - один пробел. Чтобы убрать строку-разделитель вообще - передать в аргументе пустую строку;

- end - строка, добавляемая в конец выводимого текста. По умолчанию - символ конца строки \n. Если в этом аргументе передать пустую строку след. вызов print начнется там где закончился вывод текущей строки;

- file - объект файла, стандарт. поток или др. объект, похожий на файл, куда будет выводиться текст. По умолчанию - это sys.stdout, объект стандарт. потока вывода. В этом аргументе можно передать любой объект, поддерживающий метод файлов write(string); если передается настоящий объект файла, он должен быть уже открыт для записи.

Текстовое представление объекта, для вывода, print с помощью функции str (вернее не самой функции, а её точной реализации).

**Функция print в действии.**

Вывод значений с аргументами по умолчанию:

>>> print() # Выведет пустую строку

>>> x = 'spam'

>>> y = 99

>>> z = ['eggs']

>>>

>>> print(x, y, z) # Выведет три объекта

spam 99 ['eggs']

Особые условия вывода:

>>> print(x, y, z, sep='') # Строка-разделитель выводиться не будет

spam99['eggs']

>>> print(x, y, z, sep=', ') # Нестандартная строка разделитель

spam, 99, ['eggs']

Изменяем конец вывода:

>>> print(x, y, z, end='') # Подавляем символ конца строки

spam 99 ['eggs']

- у меня не так как в книге результат, а теперь еще испытания:

>>> print(x, y, z, end='END')

spam 99 ['eggs']END

>>> print(x, y, z, end=''), print('The end')

spam 99 ['eggs']The end

(None, None)

>>> print(x, y, z), print('The end')

spam 99 ['eggs']

The end

(None, None)

- откуда-то взялось это (None, None) после вывода двух инструкций, но одна инструкция по-прежнему нормально работает:

>>> print(x, y, z, sep='')

spam99['eggs']

>>> print('The end')

The end

**- Нужно будет обмозговать.** Кстати если точку с запятой использовать, то все нормально отображается:

>>> print(x, y, z, end=''); print('The end')

spam 99 ['eggs']The end

- кажется все дело в синтаксисе... Когда буду перечитывать, вернусь к прежним главам, почитаю о запятых и точках с запятой в строке инструкций.

Ну а мы продолжаем исследования:

>>> print(x, y, z, end=''); print(x, y, z)

spam 99 ['eggs']spam 99 ['eggs']

- а вот в книге сами и показали как надо писать, значит точно все дело в запятой, вернее в точке с запятой!!!

>>> print(x, y, z, end='...\n') # Нестандартный разделитель строк

spam 99 ['eggs']...

Можно одновременно использовать именованные аргументы, но только после объектов, которые нужно вывести:

>>> print(x, y, z, sep='...', end='!\n') # Несколько именованных аргументов

spam...99...['eggs']!

>>> print(x, y, z, end='!\n', sep='...') # Порядок аргументов не имеет значения

spam...99...['eggs']!

Использование аргумента file:

>>> print(x, y, z, sep='...', file=open('data.txt', 'w')) # Вывод в файл

>>> print(x, y, z) # Вывод в поток stdout

spam 99 ['eggs']

>>> print(open('data.txt').read()) # Вывод содержимого текстового файла

spam...99...['eggs']

- кстати файл data.txt (в моей домашней папке, в Убунте) действительно содержит строку spam...99...['eggs']

При необходимости получения более сложного вывода и форматирования строки, можно сконструировать строку заранее или прибегнуть к помощи инструментов форматирования:

>>> text = '%s: %-.4f, %05d' % ('Result', 3.14159, 42)

>>> print(text)

Result: 3.1416, 00042

>>> print('%s: %-.4f, %05d' % ('Result', 3.14159, 42))

Result: 3.1416, 00042

**Инструкция print в Питон 2.6**

В 2.6 свой синтаксис, не вида встроенной функции, без круглых скобок.

Формы инструкции.

print в Питон 2.6 print в Питон 3.0 Интерпретация

print x, y print(x, y) Вывод объектов в stdout;

добавляет пробел между объектами и символ

конца строки

print x, y, print(x, y, end='') То же самое, только символ конца строки

не добавляется

print >> afile, x, y print(x, y, file=afile) Текст передается методу myfile.write,

а не sys.stdout

- как видим, запятая в print в 2.6 имеет важное значение, а не просто разделитель объектов в инструкции.

**Инструкция print в действии:**

>>> x = 'a'

>>> y = 'b'

>>> print x, y

a b

- это вывод инструкции по умолчанию.

А теперь уберем символ конца строки, завершив инструкцию запятой:

>>> print x, y,; print x, y

a b a b

Чтобы подавить пробел нужно выполнить конкатенацию или форматирование строки:

>>> print x + y

ab

>>> print '%s...%s' % (x, y)

a...b

**Перенаправление вывода print (здесь и далее обсуждаются сразу обе версии: 2.6 и 3.0).**

>>> print('Hello World') # Вызов строкового объекта в 3.0

Hello World

>>> print 'Hello world' # Вызов строкового объекта в 2.6

Hello world

Можно даже не использовать print в оболочке:

>>> 'Hello world' # Одинаково работает как в 2.6 так и в 3.0

'Hello world'

Конечно это не пример мастерства кодинга, но наглядная демонстрация поведения операции вывода.

>>> import sys

>>> sys.stdout.write('Hello World\n') # А это пример более сложного способа вывода

Hello World

>>> sys.stdout.write('Hello World\n') # Вывод команды в Питон 3.0

Hello World

12

**Перенаправление потока вывода вручную.**

print(X, Y) то же, что print X, Y в Питон 2.6

- и более длинный эквивалент:

import sys

sys.stdout.write(str(X) + ' ' + str(Y) + '\n')

- здесь мы вручную преобразуем объект в строку и добавляем символ конца строки и вызываем метод write потока вывода.

Более длинная инструкция это точный эквивалент print, потому что можно переназначить вывод sys.stdout на что-то другое, не на стандарт. поток вывода, и т.о. мы переназначим вывод print:

import sys

sys.stdout = open('log.txt', 'a') # Перенаправить вывод в файл, режим добавления в файл

...

print(x, y, z) # Текст появится в файле log.txt

- теперь все инструкции print в программе будут выводить текст в конец файла log.txt, а не в поток вывода. Т.к. инструкции print продолжают вызывать метод write объекта sys.stdout, а модуль sys у нас один для каждого процесса, то перенаправление будет действовать на все инструкции print в программе.

Такой ход полезен когда нам нужно перенаправить все выводы в какой-нибудь файл. Если вывод основан на print, то мы как раз можем воспользоваться такой особенностью.

Кстати, выводить таким способом можно и не в файл, а в любой объект, поддерживающий данный интерфейс - метод write, принимающий строковый аргумент. Этот объект может быть, например, классом, в котором реализована обработка вывода.

**Автоматическое перенаправление потоков.** стр. 369

Прием с выводом в файл в объекте sys.stdout часто используется на практике. Но есть проблема - отсутствует прямой способ восстановления первонач. потока вывода, если вдруг после вывода в файл нужн овернуться к выводу на экран.

Но, т.к. sys.stdout явл-ся объектом мы можем сохранить его и восстановить позднее:

>>> import sys

>>> temp = sys.stdout # Сохранить для последующего восстановления

>>> sys.stdout = open('log.txt', 'a') # Перенаправить вывод в файл сейчас файл log.txt

# уже создан

>>> print('spam') # Вывод идет в файл, а не на экран

>>> print(1, 2, 3)

>>> sys.stdout.close() # Вытолкнуть буферы на диск, в этот момент

# выводы инструкции print записываются в log.txt

>>> sys.stdout = temp # Восстановить первоначальный поток

>>>

>>> print('black here') # Теперь вывод идет на экран

black here

>>> print(open('log.txt').read()) # А вот то, что было выведено в файл

spam

1 2 3

- такое перенаправление используется довольно часто.

Конечно, если нам нужно отдельные инструкции print выводить в файл, а отдельные в стандартный поток, то применяются точечные перенаправления:

log = open('log.txt', 'a') # 3.0

print(x, y, z, file=log) # Вывод в объект, напоминающий файл

print(a, b, c) # Вывод в оригинальный поток вывода

log = open('log.txt', 'a') # 2.6

print >> log, x, y, z # Вывод в объект, напоминающий файл

print a, b, c # Вывод в оригинальный поток вывода

- Главное не забывать предварительно создать объект-файл (log) и передавать инструкции этот объект, а не имя файла:

>>> log = open('log.txt', 'a')

>>> print(1, 2, 3, file=log) # в 2.6 print >> log, 1, 2, 3

>>> print(4, 5, 6, file=log)

>>> log.close()

>>>

>>> print(7, 8, 9) # в 2.6 print 7, 8, 9

7 8 9

>>> print(open('log.txt').read())

1 2 3

4 5 6

Эти расширенные инструкции нередко использ-ся для вывода сообщений об ошибках в поток sys.stderr, это можно делать с помощью метода write и форматировать вручную или использовать синтаксис перенаправления:

>>> import sys

>>> sys.stderr.write(('Bad!' \* 8) + '\n')

Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

33

>>> print('Bad!' \* 8, file=sys.stderr) # в 2.6 print >> sys.stderr, 'Bad!' \* 8

Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

Еще примеры выводов и перенаправлений:

>>> X = 1; Y = 2

>>> print(X, Y) # Вывод: простой способ

1 2

>>> import sys

>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n') # Вывод: сложный способ

1 2

4 4 - это число байтов

>>> print(X, Y, file=open('temp1', 'w')) # Перенаправление в файл

>>>

>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\n') # Запись в файл

4

>>> print(open('temp1', 'rb').read()) # Двоичный режим

b'1 2\r\n'

>>> print(open('temp2', 'rb').read())

b'1 2\r\n'

**Реализация вывода, не зависящая от версии интерпретатора.**

Первый способ - это использовать инструкции 2.6 и автоматически преобразовывать в 3.0 с помощью сценария 2to3 (он есть в документации к Питон 3.0).

Второй способ - использовать функцию print 3.0 в программах 2.6, включив её использование с помощью инструкции:

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

Мы можем просто использовать скобки в 2.6 и станет похоже на 3.0, но при этом, с несколькими объектами, будет выводиться кортеж:

>>> print('spam') # Одинаковое написание и одинаковый вывод как в 3.0 так и 2.6

spam

>>> print('spam', 'ham', 'eggs') # Вызов с несколькими объектами в 3.0

spam ham eggs

>>> print('spam', 'ham', 'eggs') # Вызов с несколькими объектами в 2.6

('spam', 'ham', 'eggs') # Получаем объект кортежа!

- последний пример нужно будет попробовть из дома, в Питон 2.6. Не забыть!!!

Чтобы по-настоящему сделать независимость от версии интерпретатора, нужно сначала сконструировать строку:

>>> print('%s, %s, %s' % ('spam', 'han', 'eggs'))

spam, han, eggs

>>> print('{0}, {1}, {2}'.format('spam', 'han', 'eggs'))

spam, han, eggs

Не забывайте о круглых скобках в выводе версии 2.6 (кортеж), либо опускайте скобки в вызове print, либо перепишите вызов согласно совместимости.

***Print и stdout.***

Эквивалентность print и метода write объекта sys.stdout имеет большое значение. Она позволяет перенавить sys.stdout в другой объект, который поддерживает тот же метод write, что и файлы.

Т.к. инструкция print всего лишь передает текст методу sys.stdout.write, можно перехватывать этот текст, перенаправив sys.stdout в объект, обладающий мтеодом write для обработки текста.

Напр. можно отправить текст в окно графич. интерфейса или в неск-ко мест сразу. В общих чертах:

class FileFaker:

def write(self, string):

# Выполнить какие-либо действия со строкой

import sys

sys.stdout = FileFaker()

print(someObjects) # Передает строку методу write класса

- это все балгодаря тому, что print явл-ся - полиморфной операцией, онане интересуется тем, что за объект sys.stdout, ей нужен лишь метод (т.е. интерфейс) с именем write.

А используя аргумент file в 3.0 или расширенную форму >> в 2.6 мы вообще не влияем на объект sys.stdout.

Встроенная функция input читает данные из файла sys.stdin, благодаря этому есть возможность перенаправлять ввод аналогичным образом, используя классы, реализующие метод read (см. пример input и цикла while в гл. 10).

Вывод текста идет в поток stdout - это обеспечивает возможность вывода документов HTML в CGI-сценариях.

И кроме того это позволяет перенаправить ввод и вывод прямо в командной строке ОС:

python script.py < inputfile > outputfile

python script.py | filterProgram

Механизм перенаправления операций вывода в Питон явл-ся альтернативой этим видам перенаправления средствами командной оболочки.

**Глава 12. Условная инструкция if и синтаксические правила.** стр. 376

**Условные инструкции if.**

Если по-простому, - в Питн инструкция if выбирает, какое действие нужно выполнить. Это основной инструмент выбора в Питон.

И это наша первая составная инструкция.

Как и все составные инструкции if может содержать другие инструкции, в т.ч. и другие if.

**Общая форма.**

if в Питоне похожа на if в других ЯП. Синтаксис: записывается if с условным выражением, дале могут следовать одна или более необязательных elif ("else if") с условными выражениями и, наконец, необязательная часть else.

if <test1>: # if с условным выражением test1

<statements1> # Ассоциированный блок

elif <test2>:

<statements2>

else:

<statements3>

**Простые примеры.**

>>> if 1: # Простейший пример цикла, все необязательные части опущены

print('True')

True

>>> if 1: # Не забываем, что число 1 явл-ся логической "истиной"

print('True')

else: # Обработаем возможный ложный результат

print('False')

True

**Множественное ветвление.**

>>> x = 'killer'

>>>

>>> if x == 'roger': # Это пример со всеми необязательными частями

print("how's jessica?")

elif x == 'bugs':

print("up doc?")

else:

print("Run away!")

Run away!

Обратите внимание, что if, elif, else - находятся на одном отступе.

В таком цикле выполняются те инструкции, которые при проверке дали "истину", но у нас все условия дали "ложь" и поэтому выполнился блок else.

Множественное ветвление в Питон (аналог switch или case) оформляется в виде последовательности проверок if/elif (как в предыдущем примере) либо индексированием словарей, либо поиском в списках:

>>> print({'spam':1.25,

'ham': 1.99,

'eggs': 0.99,

'bacon': 1.10}[cho])

1.99

Менее компактный эквивалент:

>>> if cho == 'spam':

print(11)

elif cho == 'ham':

print(1.99)

elif cho == 'eggs':

print(22)

elif cho == 'bacon':

print(33)

else: # Обрабатывает ситуации, когда не найдено

ни одного совпадения

print('Bad cho')

1.99

Пример с использованием get, для получения значения по умолчанию:

>>> print(bran.get('spam', 'Bad cho')) # ключ spam есть в словаре

11

>>> print(bran.get('bacon', 'Bad cho')) # Получаем знач. по умолчанию

Bad cho

А теперь используем in:

>>> cho = 'bacon'

>>>

>>> if cho in bran:

print(bran[cho])

else:

print('Bad cho')

Bad cho

Множественные ветвления на базе словаря удобно использовать в программах, которые имеют дело с динамическими данными, однако чаще используются множественные ветвления на if. Т.е. чаще выбирают более простой и понятный способ.

**Синтаксические правила языка Питон.** стр. 379

Познакомимся с допоплнительными правилами синтаксиса Питон:

- Инструкции выполняются последовательно, одна за другой, пока не будет предусмотрено что-то другое. Т.к. путь интерпретатора Питно через текст программы наз-ся потоком управления, такие инструкции как if (циклы и др.), часто называются - инструкциями управления потоком выполнения, т.к. такие инструкции могу тизменять направление выполнения.

- Границы блоков и инструкций определяются автоматически.

- Составные инструкции = "заголовок + ":" + инструкция с отступами". Все составные инструкции в Питон оформляются одинаково: заголовок завершается двоеточием, затем идут одна или более вложенных инструкций (блоки), обычно с отступом относительно заголовка. В инструкции if, напр., elif и else явл-ся не только частями инструкции if, но и сами явл-ся заголовками с собственными блоками. Вот такие вот хитрости.

- Пустые строки, пробелы и комментарии обычно игнорируются. Пустые строки игнорятся в файлах программ, в IDLE они завершают составные инструкции; пробелы используются только в строковых литералах и при оформлении отступов, а комментарии игнорируются всегда и простираются до конца строки.

- Строки документирования игнорируются, но сохраняются и отображаются спец. инструментами. Строки доументирования - это такая доп. форма комментариев. Они доступны во время выполнения и назначение их понятно.

Одним из важных фактов в сравнении с др. ЯП будет отсутствие фигурных скобок, ограничивающих блок, и точек с запятой, завершающих инструкции. Ну и отсутствие объявления типов переменных.

**Разделители блоков: правила оформления отступов.**

Отступы - это границы блоков кода. Инструкции в блоке выстроены вертикально. Интерпретатору не важна величина отступов, важно правильное расположение их.

Рассмотрим пример с несколькими уровнями блоков:

>>> x = 'spam'

>>> if 'rub' in 'rubbery':

print(x \* 8)

x += 'NI'

if x.endswith('NI'):

x \*= 2

print(x)

spamspamspamspamspamspamspamspam

spamNIspamNI

- можем привести неправильную версию оформления этого простого примера:

>>> x = 'spam' # Ошибка: отступ в первой строке

>>> if 'rub' in 'rubbery':

print(x \* 8)

x += 'NI' # Ошибка: неуместный отступ

if x.endswith('NI'):

x \*= 2

print(x) # Ошибка: непоследовательное оформление отступов

Отступы - это часть синтаксиса языка.

**Не смешивайте пробелы и символы табуляции: новый механизм проверки ошибок в Питон 3.0**

Используйте в отступах или пробелы или табуляцию, но не вместе. В Питон 2.6 запуск сценария с ключом -t (python -t main.py) будет предупреждать если в коде смешаны пробелы и табуляция (в Питно 2.6 можно смешивать эти символы), а с ключом -tt буде тсчитать ошибкой (аналог работы ошибки в Питон 3.0).

**Разделители инструкций: строки и многострочные инструкции.**

- Инструкции могут располагаться в нескольких строках, если они окружены иснтаксической парой скобок (), {} или []. Желательно выравнивать промежуточные строки, но не обязательно.

- Инструкции могут располагаться в нескольких строках, если они завершаются символом обратного слеша. Это несколько устаревшая форма, легче использовать просто круглые скобки, т.к. если вдруг забудете поставить слеш (а это легко пропустить), то можно получит ьсовсем не то, что ожидалось.

- Литералы строк в тройных кавычках могут располагаться в нескольких строках. С ними мы познакомились в главе 7.

- Другие правила. Инструкции можно завершать точкой с запятой (используется такое редко), чаще используется, чтобы разместить в одной строке неск-ко инструкций. Также в любом месте файла могут присутствовать пустые строки и комментарии.

**Несколько специальных случаев.**

Примеры:

>>> L = ['Good',

'Bad',

'Ugly']

>>> L

['Good', 'Bad', 'Ugly']

- этот прием можно использовать с круглыми и фигурными скобками (получим кортеж или словарь).

if a == b and c == d and \ # мало используемый прием разделения

d == e and f == g:

print('olde')

- лучше использовать круглые скобки, т.к. в них можно заключить любое выражение:

if (a == b and c == d and # рекомендуемый прием

d == e and f == g):

print('olde')

Несоставные инструкции в одной строке:

>>> x = 1; y = 2; print(x)

1

Еще многострочные примеры:

>>> S = """

aaaa

bbbb

cccc"""

>>> S

'\naaaa\nbbbb\ncccc' # Видим все символы, даже символ конца строки

>>> S = ('aaaa'

'bbbb'

'cccc')

>>> S

'aaaabbbbcccc' # А здесь у нас неявная операция конкатенации

Простая инструкция с единственным условием и действием, можно записывать в одну строку:

if 1: print('hello')

Всё это - специальные случаи, рекомендуется, все же, избегать таких написаний и располагать инструкции в разных строках и выравнивать блоки, исключением являются совсем простейшие случаи кода.

**Проверка истинности.** стр. 385

В Питон есть свои особенности по проверке истинности:

- Любое число, не равное нулю, или непустой объект интерпретируются как истина.

- Числа, равные нулю, пустые объекты и специальный объект None интерпретируются как ложь.

- Операции сравнения и проверки на равенство применяются к структурам данных рекурсивно.

- Операции сравнения и проверки на равенство возвращают True или False (которые 1 и 0).

- Логические операторы and и or возвращают истинный или ложный объект-операнд.

Логические операторы:

X and Y - Истина, если оба значения X и Y истинны.

X or Y - Истина, если любое из значений X или Y истинно.

not X - Истина, если значение X ложно (выражение возвращает True или False).

X, Y могут быть значениями истинности или выражениями, возвращающими истинность.

Кроме того, and и or возвращают истинный или ложный ОБЪЕКТ, а не значения True, False. Рассмотрим их работу ниже:

>>> 2 < 3, 3 < 2 # Операции сравнения возвращают True или False

(True, False) # True, False по сути 1 и 0, просто выводятся по особому

Оператор or проверяет слева направо и возвращает первый же имеющий истину, т.е. прекращает дальнейшее вычисление как тольк овстретит истину - обычно это наз-ся вычислением по короткой схеме:

>>> 2 or 3, 3 or 2 # Вернет левый (первый) операнд, если он истинный

(2, 3)

>>> [] or 3 # Правый, т.к. левый (первый) ложный

3

>>> [] or {} # Правый, т.к. левый (первый) ложный и интерпретатор

{} # просто вычисляет и возвращает объект справа

Оператор and также останавливается как только результат станет известен, однако он вычисляет операнды слева направо и возвращает первый объект имеющий ЛОЖНОЕ значение:

>>> 2 and 3, 3 and 2 # Возвращает правый операнд, т.е. левый истинный

(3, 2)

>>> [] and {} # Возвращает левый операнд, если он ложный

[]

>>> 3 and [] # Иначе возвращает правый операнд (истинный или ложный)

[]

Конечный результат - получаем значение, которое логически интепретируется как истина или ложь пр ииспользовании в инструкции if или while.

**Трехместное выражение if/else.**

Логические операторы, по сути, образовывают выражения, которые выполняются как инструкция if.

if X:

A = Y

else:

A = Z

- рассмотрим альтернативное выражение:

A = Y if X else Z

- здесь также интерпретатор выполняет приравнивание к Y, только если X истинное, а Z выполняется, только если Х ложное.

Пример:

>>> A = 't' if 'spam' else 'f' # Непустая строка - это "истина"

>>> A

't'

>>> A = 't' if '' else 'f' # Пустая тсрока - это "ложь"

>>> A

'f'

Дополнительно. Т.к. функция bool преобразуте Х в соотв. число 1 или 0, получим выбор требуемого значения из списка:

A = [Z, Y][bool(X)]

Например:

>>> ['f', 't'][bool('')] # Рез-т bool "ложь" - получаем элемент с индексом 0

'f'

>>> ['f', 't'][bool('spam')] # Рез-т bool "истина" - получаем элемент с индексом 1

't'

Но лучше использовать более простое if/else и то, толькое если обе части выражения просты, в более сложных случаях лучше использ. обычную версию инструкции if.

***Логические значения.***

В Питон часто использ. прием выбора одного объекта из множества, основанный на поведение логич. операторов:

X = A or B or C or None

- эта инструкция присвоит Х первый истинный объект или None если все окажутся пустыми (ложными).

Также т.о. можно присваивать дефолтные значения:

X = A or default # A присваивается если оно истинно (непустое)

Важно знать как выполняются вычисления по сокращенной схеме, потмоу что справа от логического оператора может находиться функция, выполняющая важную работу или оказывающая побочное влияние, вызов которой не произойдет из-за действия правила вычисления по сокр. схеме:

if f1() or f2(): ...

- В данном случае, если функция f1() вернет истинное (непустое) значение, интерпретатор не вызовет функцию f2().

Чтобы гаранитровать вызов обоих можно вызвать их до применения оператора or:

tmp1, tmp2 = f1(), f2()

if tmp1 or tmp2: ...

**Глава 13. Циклы while и for.** стр. 392

**Циклы while.**

while - продолжает выполнять блок инструкций (обычно с отступами) до тех пор, пока условное выражение продолжает возвращать "истину".

Она наз-ся - циклом, потому что управление циклически возвращается к началу инструкции, пока условное выражение не вернет ложное значение. Как только проверка условия вернет "ложь", управление будет передано инструкции, следующей за вложенным блоком тела цикла while.

Т.е. тело цикла продолжает выполняться снова и снова, пока условное выражение возвращает истину.

**Общий формат.**

Строка заголовка с условием; тело цикла (с однйо или более инструкциями с отступами); необязательная часть else, на случай если управление передается за границы цикла без использования break:

while <test>: # Условное выражение test

<statements1> # Тело цикла

else:

<statements2> # else выполняется если нет инструкции break

Примеры:

>>> while True:

print('Cicle')

...

- это пример бесконечного цикла, т.к. условное выражение всегда будет истинно.

Следующий пример - цикл вырезает из стркои первый символ, пока она не опустеет и т.о. не станет "ложной", т.е. пустой.

>>> x = 'spam'

>>> while x: # То же, что (while x != ''), но гораздо компактнее

print(x, end=' ')

x = x[1:]

spam pam am m

Пример: перебираем значения от a до b, не включая b:

>>> a = 0; b = 10

>>> while a < b:

print(a, end=' ') # end=' ' заменяет символ конца строки на пробел, не забываем

a +=1

0 1 2 3 4 5 6 7 8 9

- т.н. цикл пересечения, далее в книге мы реализуем его по-другому.

В Питон нет цикла "do until", его можно имитировать добавив в тело цикла условную инструкцию и инструкцию break:

while True:

... тело цикла ...

if exitTest(): break

- чтобы понять как это работает перейдем к след. разделу.

**break, continue, pass и else.**

В языке Питон:

- break - производит переход за пределы объемлющего цикла (всей инструкции цикла).

- continue - производит переход в начало цикла (в строку заголовка).

- pass - ничего не делает: это пустая инструкция, используемая как заполнитель.

- блок else - выполняется, только если цикл завершился обычным образом (без использования инструкции break).

**Общий формат цикла с учетом новых инструкций:**

while <test1>:

<statements1>

if <test2>: break # Выйти из цикла, пропустив часть else

if <test3>: continue # Перейти в начало цикла, к выражению tets1

else:

<statements2> # Выполняется, если не была использована break

- break и continue могут появляться в любом месте внутри цикла while (или for), но обычно они используются в условных выражениях if.

**Описание и примеры инструкции pass.**

pass не вып-ет действий, она использ. там где синтаксисом обязательная инструкция, но действий в этой точке программы выполнять нельзя.

Напр. бесконечный цикл:

while 1: pass

- интерпретатор зависает в этой точке, потмоу что циклически ничего не делает, такая вот хитрая инструкция.

pass в мире инструкций - это как None в мире объектов.

Кстати, видим, что цикл while тоже можно записывать в одну строку, но только простые инструкции!

Далее в книге мы рассмотрим норм. примеры, где pass использ-ся, чтобы игнорировать исключения в инструкции try, или для определения пустых классов, реализующих объекты, которые ведут себя подобно другим классам.

Иногда pass используется как заполнитель, вместо того, что "что будет написано позднее", и в качестве временного тела фиктивной функции:

def func1():

pass

def func2():

pass

- пустое тело функции вызовет ошибку, поэтмоу вставляется pass.

В Питон 3.0 вместо любого выражения можно писать три точки ... и это явл-ся альтернативой pass, т.е. пример выше можно переписать:

def func1():

...

def func2():

...

>>> x = ... # А это альтернатива объекту None

>>> x

Ellipsis

**Описание и примеры инструкции continue.**

continue - вызывает немедленный переход в начало цикла. Это позволяет избегать вложенных инструкций.

Пример, continue пропускает нечетные числа:

>>> while x:

x = x - 1 # или можно написать x -= 1

if x % 2 !=0: continue # Нечетное? - пропустить вывод print()

print(x, end=' ')

8 6 4 2 0

- помним, что 0 это "ложь", а оператор % вычисляет остаток от деления.

Заметим также, что continue напоминает инструкцию "goto" из других ЯП.

Перепишем наш пример с помощью if:

>>> x = 10

>>> while x:

x -= 1

if x % 2 ==0:

print(x, end=' ')

8 6 4 2 0

**Замечание!** - Оба варианта цикла while (т.е. цикл while вообще) - изменяют переемнную х, после завершения цикла она стала равной нулю!

**Описание и примеры инструкции break.**

break вызывает немедленный выход из цикла. И её также можно использовать чтобы избежать вложений.

Пример, итерактивный цикл, где ввод с помощью input и выход если будет введено "stop":

>>> while 1:

name = input('Enter name:')

if name == 'stop': break

age = input('Enter age:')

print('Hello', name, '=>', int(age) \*\* 2)

Enter name:mel

Enter age:11

Hello mel => 121

Enter name:bob

Enter age:30

Hello bob => 900

Enter name:stop

- Обратите внимание, мы преобразуем age в целое число с помощью функции int, т.к. функция input возвращает ввод пользователя в виде строки.

input может возбуждать исключение при получении символа конца файла (при комбинации Ctrl-Z или Ctrl-D), если это может быть, то нужно обернуть вызов input инструкцией try.

**Описание и примеры инструкции else.**

break и else позволяют не применять флаг состояния поиска.

Например, пример цикла, который определяет, явл-ся ли положительное целое число y простым числом, выполняя поиск делителей больше 1:

x = y // 2 # для значений y > 1

# // - это оператор деления, см. главу 4

while x > 1:

if y % x == 0:

print(y, 'has factor', x)

break # Перешагнуть блок else

x -= 1

else:

print(y, 'is prime')

- вставляем break в месте, где будет найден делитель, вместо установки спец. флага. В нашем цикле управление будет передано else, только если break не была выполнена, т.е. когда уверенно можно сказать, что число явл-ся простым.

- else также выполняется когда тело цикла ниразу не выполнилось.

- т.е. в нашем примере будет получено 'is prime' если x изначально меньше или равно 1 (т.е. когда y = 2).

**Еще о else.**

else - обеспечивает явный синтаксис частой ситуации - else позволяет обработать "другой" способ выхода из цикла, без установки и проверки флагов или условий.

Напр. мы создаем цикл поиска некоторого значения и после выхода из цикла нам нужно узнать - то ли это значение, которое мы искали:

found = False

while x and not found:

if match(x[0]):

print('Ni') # Искомое значение является первым?

found = True

else:

x = x[1:] # Вырезать первое значение и повторить

if not found:

print('not found')

- здесь мы инициализируем и проверям флаг, чтобы проверить поиск.

Но этот цикл можно переписать с использованием else:

while x: # Выйти, когда х опустеет

if match(x[0]):

print('Ni')

break # Выход, в обход блока else

x = x[1:]

else:

print('Not found') # Блок отработает, только если строка х исчерпана

- это более компактная и более удачная реализация такой задачи.

***Имитация циклов while языка С.***

Питон не предусматривает возможность выполнять присваивание там, где ожидается выражение. Т.е. шаблон на С в питоне не применим:

while ((x = next()) != Null) {... обработка х ...}

Т.е. в Питон невозможно вставить = там где подразумевается ==.

Но в циклах while мы можем сделать как в С.

- Можно поместить операцию присваивания в тело цикла с инструкцией break:

while True:

x = next()

if not x: break

...обработка х...

- Или вместе с инструкцией if:

x = True

while x:

x = next()

if x:

...обработка х...

- Или вынести первое присваивание за пределы цикла:

x = next()

while x:

...обработка х...

x = next()

- первый из этих циклов наиболее простой и часто используемый.

**Циклы for.** стр. 400

for - универсальный итератор последовательностей в Питон. Может работать со всеми типами объектов и классов, поддерживающих итерацию.

**Общий формат.**

Строка заголовка, где указана переменная для присваивания (или - цель), а также объект, обход которого выполняется; далее идет блок (с отступами) инструкций, которые нужно выполнить:

for <target> in <object>: # Связывает элементы объекта с переменной

<statements> # Повторяющееся тело цикла, использует переменную цикла

else:

<statements> # Если не попали на break

Цикл for один за другим присваивает элементы объекта последовательности переменной цикла и выполянет тело цикла для каждого из них.

Обращение к текущему элементу последовательности, в теле цикла, выполняется переменной цикла.

После выхода из цикла, обычно переменная цикла ссылается на последний элемент последовательности.

Полная форма цикла for, break и continue имеет вид:

for <target> in <object>:

<statements>

if <test>: break

if <test>: continue

else:

<statements>

**Примеры:**

**Типичные варианты использования.**

- поочередно, слева направо, прсивоим х каждый из трех элементов списка и выведем их:

>>> for x in ['spam', 'eggs', 'ham']:

print(x, end=' ')

spam eggs ham

- вычислим сумму и произведение всех элементов в списке:

>>> sum = 0

>>> for x in [1, 2, 3, 4]:

sum = sum + x

>>> sum

10

>>> x # переменная цикла - равна последнему элементу

4

>>> prod = 1

>>> for item in [1, 2, 3, 4]: prod \*= item # простое выражение можно и в одну строку

>>> prod

24

>>> item

4

**Другие типы данных.** стр. 401

Применение к строкам и кортежам:

>>> S = "Lumberjack"

>>> T = ("and", "T'm", "okay")

>>> for x in S: print(x, end=' ')

...

L u m b e r j a c k

>>> for x in T: print(x, end=' ')

...

and T'm okay '

- Ниже мы увидим, что for может применяться даже к объектам не явл-ся последовательностями, напр. к файлам и словарям!

**Присваивание кортежа в цикле for.**

Если ободить послед-ть кортежей, то переменная цикла сама будет кортежем:

>>> T = [(1, 2),(3, 4),(5, 6)] # Список кортежей

>>> for (a, b) in T: # Операция присваивания кортежа в действии

print(a, b)

1 2

3 4

5 6

Первый проход цикла подобен инструкции (a, b) = (1, 2) и т.д., в рез-те в каждой итерации вып-ся операция присв-ия кортежа.

Такое присваивание возможно благодаря функции zip. Эта форма часто использ-ся при работе с БД, когда ответ на запрос SQL возвращается в виде послед-ти последовательностей, таких как список в нашем примере - внешний список - это таблица БД, а вложенные кортежи - строки в таблице, т.о. циклом мы извлекаем нужные нам записи.

Циклом for можно обходить словари:

>>> D = {'a':1, 'b':2, 'c':3} # k - key - ключи словаря

>>>

>>> for k in D: # Информация в словарях ищется по ключам!

print(k, '=>', D[k]) # Значения находятся методом items, см. далее

b => 2

c => 3

a => 1

>>> list(D.items())

[('b', 2), ('c', 3), ('a', 1)]

>>> for (key, value) in D.items():

print(key, '=>', value) # Обход ключей и значений одновременно

b => 2

c => 3

a => 1

- синтаксически, после слова for выполняется присваивание переменной цикла любого вида (любого типа).

Мы всегда можем выполнить распаковку кортежа внутри цикла:

>>> T

[(1, 2), (3, 4), (5, 6)]

>>>

>>> for both in T:

a, b = both # Присваивание вручную

print(a, b)

1 2

3 4

5 6

- но использование кортежа в заголовке позволяет экономить одну операцию в теле цикла.

Автоматическое распаковывание вложенных структур в цикле for:

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

>>> a, b, c

(1, 2, 3)

- рассмотрим аналог с циклом for:

>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)

1 2 3

4 5 6

- и еще пример:

>>> for ((a, b), c) in [((1, 2), 3), ['XY', 6]]: print(a, b, c)

1 2 3

X Y 6

**Расширенная операция присваивания последовательностей в циклах for в версии Питон 3.0**

Рассмотрим пример:

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

>>> a, b, c

(1, 2, 3)

>>>

>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:

print(a, b, c)

1 2 3

4 5 6

- А теперь рассмотрим пример для извлечения фрагментов влож-х послед-тей в цикле for:

>>> a, \*b, c = (1, 2, 3, 4) # Расширенная инструкция

>>> a, b, c # распаковывания последовательностей

(1, [2, 3], 4)

>>>

>>> for (a, \*b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:

print(a, b, c)

1 [2, 3] 4

5 [6, 7] 8

На практике этот подход использ-ся для выборки нескольких полей из записи данных.

А теперь тот же пример в версии Питон 2.6:

>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:

a, b, c = all[0], all[1:3], all[3]

print(a, b, c)

1 (2, 3) 4

5 (6, 7) 8

**Вложенные циклы for.**

Пример: имея список объектов (items) и список ключей (tests), фрагмент пытается отыскать каждый ключ в списке объектов и сообщает о рез-тах.

>>> items = ["aaa", 111, (4, 5), 2.01] # Множество объектов

>>> tests = [(4, 5), 3.14] # Ключи, которые требуется искать

>>> for key in tests: # Для всех ключей, for1

for item in items: # Для всех элементов, for2

if item == key:

print(key, "was found") # Проверить совпадение

break

else:

print(key, "not found")

(4, 5) was found

3.14 not found

- Т.к. при совпадении if вызывает break, можно утверждать, что блок else будет выполняться, только если поиск завершился неудачей. else у нас относится к вложенному циклу for.

- Как видим, мы запускаем одновременно два цикла, внешний (for1) вып-т обход ключей, а внутренний (for2) - обход элементов, в поисках каждого ключа.

- Т.е. за один проход for1 выполняются все итерации for2, т.о. одна переменная for1 сравнивается со всеми элементами for2, а после завершения for2 в действие вступает второй элемент цикла for1 и действия for2 повторяются.

Кстати, этот пример можно переписать с оператором in:

>>> for key in tests:

if key in items:

print(key, 'was found')

else:

print(key, 'not found!')

(4, 5) was found

3.14 not found!

- ради компактности кода и скорости вычислений лучше переложить на интерпретатор большинство вычислений, как в этом примере с оператором in.

Пример, выборка одинаковых элементов из двух последовательностей:

>>> seq1 = 'spam'

>>> seq2 = 'spam'

>>> res = [] # Создаем пустой список

>>> for x in seq1: # Выполнить обход первой последовательности

if x in seq2: # Общий элемент?

res.append(x) # Если общий, то добавляем его в конец списка res

>>> res # Общие элементы записываются в список res

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

- Было бы замечательно, если бы этот цикл можно было бы сделать универсальным, чтобы использ не только для seq1 и seq2, а многократно.

- Эт амысль ведет нас к - функциям, которые мы будем изучать далее в книге.

***Сканирования файлов.***

Загрузить содержимое файла в строку одной инструкцией, методом read:

file = open('test.txt', 'r')

print(file.read())

Но для загрузки по частям испол-ся цикд while, завершающийся break, при достижении конца файла, или цикл for. Посимвольное чтение парой способов:

file = open('test.txt')

while True:

char = file.read(1) # Читать по одному символу

if not char: break

print(char)

for char in open('test.txt').read():

print(char)

Чтение строками или блоками циклом while:

file = open('test.txt')

while True:

line = file.readline() # Читать строку за строкой

if not line: break

print(line, end=' ') # Прочитанная строка уже содежит символ \n

file = open('test.txt')

while True:

chunk = file.read(10) # Читать блоками по 10 байтов

if not chunk: break

print(chunk)

Двоичные данные читаются блоками определенного размера. Однако, в случаетекстовых данных, чтение с помощью for проще и быстрее:

for line in open('test.txt').readlines(): # метод readlines() загружает файл в память целиком

print(line, end=' ')

1 2 3

4 5 6

Hello world

Bye!

for line in open('test.txt'): # Использование итератора:

print(line, end=' ') # Лучший способ чтения текста

1 2 3

4 5 6

Hello world

Bye!

- последний пример - наиболее предпочтительный способ рабоыт с текстовыми файлами, т.к. не загружает файл целиком в память.

**Приемы программирования циклов.** стр. 407

for - это счетынй цикл. Он проще и быстрее чем while, поэтому его нужно рассматривать в первую очередь, если нужно сделать обход последовательности.

Особые случаи обхода в цикле for:

- Встроенная функция range возвращает непрерывную послед-ть увеличивающихся целых чисел, которые можно использовать в качестве индексов внутри цикла for.

- Встроенная функция zip возвращает список кортежей, составленных из элементов входных списков с одинаковыми индексами, котоырй может использ-ся для одновременного обхода нескольких послед-тей в цикле for.

Везде, где только можно, используйте for.

**Счетные циклы: while и range.**

В 3.0 range возвращает итератор, кот. генерирует элементы по требованию, поэтому мы обернем её в list:

>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))

([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])

- range с одним аргументом, генерирует спсиок целых чисел в диапазоне от нуля до значения аргумента, без его включения (см. первый список).

- range c двумя аргументами, тогда первый аргумент будет определять нижнюю границу диапазона (см. второй список).

- range с третьим (необязательным) аргументом, - эот аргумент определяет шаг (по умолчанию шаг равен 1).

Можно использ-ть отриц. числа и в порядке убывания:

>>> list(range(-5, 5))

[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(range(5, -5, -1))

[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]

>>> list(range(5, -5)) # Если не передадим шаг -1 получим пустой список

[]

В циклах for range дает простой способ повторить действие определенное число раз.

Напр. использ range для вывода строки три раза.

**Кстати!** В Питон 3.0 for автоматически извлекает значения из итератора range, поэтому list не требуется:

>>> for i in range(3):

print(i, 'Pythons')

0 Pythons

1 Pythons

2 Pythons

Также range использ. для косвенного обхода последовательностей.

Самый простой способ обхода это цикл for, при этом основную работу вып-т интерпретатор:

>>> X = 'spam'

>>> for item in X: print(item, end=' ')

s p a m

Можно сделать то же самое с циклом while:

>>> i = 0

>>> while i < len(X):

print(X[i], end=' ')

i += 1

s p a m

А теперь посмотрим на работу for с функцией range:

>>> X = 'spam'

>>> len(X) # Длина строки

4

>>> list(range(len(X))) # Все допустимые смещения Х

[0, 1, 2, 3]

>>>

>>> for i in range(len(X)): print(X[i], end=' ') # Извлечение элементов вручную

s p a m

- в этом примере вып-ся обход списка смещений в стркое Х, а не фактических элементов строки.

**Обход части последовательности: range и срезы.**

Последний пример с range можно переписать в более лучшей форме:

for item in X: print(item)

Однако пример с range позволяет управлять обходом, напр. проупскать некоторые элементы:

>>> S = 'abcdefghijk'

>>> list(range(0, len(S), 2))

[0, 2, 4, 6, 8, 10]

>>>

>>> for i in range(0, len(S), 2): print(S[i], end=' ')

a c e g i k

- выбирается каждый второ элемент строки при обохде значений сгенерированных функцией range.

Но есть другой путь для решения этой задачи, более простой:

>>> S = 'abcdefghijk'

>>> for c in S[::2]: print(c, end=' ')

a c e g i k

**Изменение списков: range**

Пусть нам нужно прибавит 1 к каждому элементу списка:

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

>>> for x in L:

x += 1

>>> L

[1, 2, 3, 4, 5]

>>> x

6

- здесь у нас изменилась только переменная цикла, а не список.

Чтобы изменить список нам нужно выполнять присваивание по индексу:

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

>>> for i in range(len(L)):

L[i] += 1

>>> L

[2, 3, 4, 5, 6]

Можно написать эту же команду и с циклом while, но он будет чуть длиннее и будет выполняться медленнее.

>>> i = 0

>>> while i < len(L):

L[i] += 1

i += 1

>>> L

[3, 4, 5, 6, 7]

**Параллельный обход: zip и map.**

zip - позволяет выполнить обход нескольких последовательностей параллельно.

>>> L1 = [1, 2, 3, 4]

>>> L2 = [5, 6, 7, 8]

>>> zip(L1, L2)

<zip object at 0x00F6E300>

>>> list(zip(L1, L2))

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

- подобно range, zip также возвращает итерируемый объект, поэтому оборачиваем её в list.

- в рез-те нам возвращается список кортежей из пар элементов.

>>> for (x, y) in zip(L1, L2):

print(x, y, ' -- ', x+y)

1 5 -- 6

2 6 -- 8

3 7 -- 10

4 8 -- 12

- в этом примере мы обходим циклом for пары кортежей zip.

Функция zip принимает любые итерируемые объекты, даже файлы и позволяет указывать более двух аргументов.

Т.е. из N аргументов функция zip создает N-мерный кортеж:

>>> T1, T2, T3 = (1, 2, 3),(4, 5, 6),(7, 8, 9)

>>> T3

(7, 8, 9)

>>> list(zip(T1, T2, T3))

[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

Длина списка, возвращаемого zip, равна длине кратчайшей из последовательностей, если аргументы разной длины:

>>> S1 = 'abc'

>>> S2 = 'xyz123'

>>>

>>> list(zip(S1, S2))

[('a', 'x'), ('b', 'y'), ('c', 'z')]

>>> list(zip(S2, S1))

[('x', 'a'), ('y', 'b'), ('z', 'c')]

**Эквивалентная функция map в Питон 2.6**

>>> S1 = 'abc'

>>> S2 = 'xyz123'

>>> list(map(None, S1, S2))

[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, 1), (None, 2), (None, 3)]

>>> list(map(ord, 'spam'))

[115, 112, 97, 109]

Тот же рез-т с помощью цикла:

>>> res = []

>>> for c in 'spam': res.append(ord(c))

>>> res

[115, 112, 97, 109]

**Конструирование словаря с помощью zip.**

Словари можно создавать с помощью литералов или присваивая значения ключам.

Но как быть, если ключи и значения нах-ся в программе в списках:

>>> keys = ['spam', 'eggs', 'toast']

>>> vals = [1, 3, 5]

Применим функцию zip и цикл for для создания словаря:

>>> list(zip(keys, vals))

[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}

>>> for (k, v) in zip(keys, vals): D2[k] = v

>>> D2

{'spam': 1, 'eggs': 3, 'toast': 5}

Однако, начиная с Питон 2.2 можно обойтись без цикла for, передав рез-т zip конструктору dict:

>>> keys = ['spam', 'eggs', 'toast']

>>> vals = [1, 3, 5]

>>>

>>> D3 = dict(zip(keys, vals))

>>> D3

{'spam': 1, 'eggs': 3, 'toast': 5}

Встроенное имя dict, в действительности явл-ся именем типа. Этот вызов приводит к преорбазованию списка в словарь, н она самом деле это вызов конструктора объекта.

**Генерирование индексов и элементов: enumerate**

Для получения индекса элемента и самого элемента можно использовать цикл for:

>>> S = 'spam'

>>> offset = 0

>>> for item in S:

print(item, 'appears at offset', offset)

offset += 1

s appears at offset 0

p appears at offset 1

a appears at offset 2

m appears at offset 3

А теперь то же самое с помощью функции enumerate:

>>> S = 'spam'

>>> for (offset, item) in enumerate(S):

print(item, 'appears at offset', offset)

s appears at offset 0

p appears at offset 1

a appears at offset 2

m appears at offset 3

Функция enumerate возвращает объект-генератор. Пример присваивания:

>>> E = enumerate(S)

>>> E

<enumerate object at 0x016DD490>

>>> next(E)

(0, 's')

>>> next(E)

(1, 'p')

>>> next(E)

(2, 'a')

>>> [c \* i for (i, c) in enumerate(S)]

['', 'p', 'aa', 'mmm']

**Глава 14. Итерации и генераторы, часть 1.** стр. 417

Знания в этих главах могут показаться избыточными, но эти удобные инструменты часто используются и мы можем встретить их в чьем-то коде.

**Итераторы: первое знакомство.**

Мы знаем, что цикл for может работать с последовательностями любого типа (списки, кортежи, строки), фактически - он способен работать с любыми итерируемыми объектами (которые поддерживают итерации).

На самом деле это верно для всех стредств выполнения итераций (циклы for, генераторы списков, оператор in, функция map и др.).

Объект считается итерируемым, либо если он физически явл-ся последовательностью, либо объектом, который воспроизводит по одному результату за раз в контексте инструментов выполнения итераций, таких как цикл for.

**Протокол итераций: итераторы файлов.**

Объекты открытых файлов имеют метод с именем readline, котоырй читает по одной строке за одно обращение - каждый раз, вызывая метод readline, мы перемещаемся к след. строке.

В конце файла возвращается пустая строка, что может служить сигналом для выхода из цикла:

>>> f.readline()

'import sys\n'

>>> f.readline()

'print(sys.path)\n'

>>> f.readline()

'x = 2\n'

>>> f.readline()

'print(x \*\* 33)\n'

>>> f.readline()

''

Файлы имеют метод \_\_next\_\_, который производит почти тот же эффект - когда его вызывают возвращает строку. Единственно, при достижение конца файла \_\_next\_\_ возбуждает исключение:

>>> f = open('tnt.py')

>>> f.\_\_next\_\_()

'import sys\n'

>>> f.\_\_next\_\_()

'print(sys.path)\n'

>>> f.\_\_next\_\_()

'x = 2\n'

>>> f.\_\_next\_\_()

'print(x \*\* 33)\n'

>>> f.\_\_next\_\_()

Traceback (most recent call last):

File "<pyshell#34>", line 1, in <module>

f.\_\_next\_\_()

StopIteration

- любой такой объект доступен для сканирования с помощью цикла for или других итерационных инструментов.

Пример, читаем содержимое файла строку за строкой, переводит символы к верхнему регистру без обращения к методам файла:

>>> for line in open('tnt.py'):

print(line.upper(), end='')

IMPORT SYS

PRINT(SYS.PATH)

X = 2

PRINT(X \*\* 33)

- аргумент end='', подавляет вывод символа \n, потомоу что строки, прочитанные из файла, уже содержат этот символ (иначе у нас после каждой строки была бы еще пустая строка).

Можно читать и с помощью метода readlines, но тогда файл будет загружаться в память целиком, что не очень хорошо:

>>> for line in open('tnt.py').readlines():

print(line.upper(), end='')

Можно реализовать с помощью цикла while:

>>> f = open('tnt.py')

>>> while True:

line = f.readlines()

if not line: break

print(line.upper(), end='')

... вывод тот же

**Выполнение итераций вручную: iter и next.** стр. 420

Функция next, автоматически вызывает метод \_\_next\_\_ объекта.

Напр. у нас есть итерируемый объект X, тогда вызов next(X) будет равносилен вызову X.\_\_next\_\_(), но выглядит намного проще:

>>> f = open('script1.py')

>>> f.\_\_next\_\_()

'import sys\n'

>>> f.\_\_next\_\_()

'print(sys.platform)\n'

>>> f = open('script1.py')

>>> next(f)

'import sys\n'

>>> next(f)

'print(sys.platform)\n'

Вначале цикла, цикл for получает итератор из итерируемого объекта, передавая его встроенной функции iter, которая возвращает объект, имеющий требуемый метод \_\_next\_\_

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

>>> I = iter(L) # Получить объект-итератор

>>> I.\_\_next\_\_() # Вызвать \_\_next\_\_ чтобы перейти к следующему элементу

1

>>> I.\_\_next\_\_()

2

>>> I.\_\_next\_\_()

3

>>> I.\_\_next\_\_()

...

StopIteration

С файлами этот начальный этап не нужен, у файлов есть свой итератор, т.е. объекты файлов имеют свой метод \_\_next\_\_

>>> f = open('script1.py')

>>> iter(f) is f

True

>>> f.\_\_next\_\_()

'import sys\n'

Списки и др. объекты не имеют итераторов, поэтому нужно вызывать iter:

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

>>> iter(L) is L

False

>>> L.\_\_next\_\_()

...

AttributeError: 'list' object has no attribute '\_\_next\_\_'

>>> I = iter(L)

>>> I.\_\_next\_\_()

1

>>> next(I) # То же, что и вызов метода I.\_\_next\_\_()

2

Рассмотрим эквивалентность ручной и автоматической итерации:

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

>>> for X in L: # Автоматический способ выполнения итераций

print(X \*\* 2, end=' ') # Получает итератор, вызывает \_\_next\_\_

# обрабатывает исключение

1 4 9

>>> I = iter(L) # Ручная итерация, имитация цикла for

>>> while True:

try: # Инструкция try обрабатывает исключения

X = next(I) # Или I.\_\_next\_\_

except StopIteration:

break

print(X \*\* 2, end=' ')

1 4 9

- для понимания второго примера нужно знать, что try выполняет операцию и перехватывает исключения, которые могут возникнуть в ходе операции.

В Питон 2.6 метод итераций вызывается X.next(), а не X.\_\_next\_\_(), в 2.6 также доступна функция next(X).

**Другие итераторы встроенных типов.**

Пример обхода словаря, с запросом ключей:

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

>>> for key in D.keys():

print(key, D[key])

c 3

b 2

a 1

В последних версиях Питон есть итератор словарей, который возвращает по одному ключу за раз:

>>> I = iter(D)

>>> next(I)

'c'

>>> next(I)

'b'

>>> next(I)

'a'

>>> next(I)

...

StopIteration

Теперь можно не использовать метод keys:

>>> for key in D:

print(key, D[key])

c 3

b 2

a 1

Объекты разных типов поддерживают протокол итераций, напр. хранилища объектов (объекты файлов с доступом по ключу, использ-е дял хранения объектов Питон) и объекты, возвращаемые функцией os.popen (исп-ся для чтения вывода команд системной оболочки):

>>> import os

>>> P = os.popen('dir')

>>> P.\_\_next\_\_()

' ’®¬ ў гбва®©бвўҐ C \xadҐ Ё¬ҐҐв ¬ҐвЄЁ.\n' # интерпретатор не понимает русские названия

>>> P.\_\_next\_\_()

' ‘ҐаЁ©\xadл© \xad®¬Ґа в®¬\xa0: A04A-5E2E\n'

>>> P.\_\_next\_\_()

'\n'

>>> P.\_\_next\_\_()

' ‘®¤Ґа¦Ё¬®Ґ Ї\xa0ЇЄЁ C:\\Python34\n'

- Заметим, что объекты, возвращаемые функцией popen, в 2.6 поддерживают метод P.next(); а в 3.0 поддерживают метод P.\_\_next\_\_(), но не поддерживают функцию next(P), но это только проблема ручных итераций, а автоматические можно сделать с помощью цикла for.

Поддержка протокола итераций, - это причина по которой мы обертываем в list некоторые результаты:

>>> R = range(5)

>>> R # Диапазоны в весрии 3.0 - это итерируемые объекты

range(0, 5)

>>> I = iter(R) # Используйте протокол итераций для обхода элементов

>>> next(I)

0

>>> next(I)

1

>>> list(range(5)) # Или функцию list для получения всех элементов сразу

[0, 1, 2, 3, 4]

Теперь мы, зная о протоколе итераций, должны суметь объяснить следующее поведение функции enumerate:

>>> E = enumerate('spam') # enumerate возвращает итерирумеый объект

>>> E

<enumerate object at 0x02287828>

>>> I = iter(E)

>>> next(I) # Получить результаты с помощью протокола итераций

(0, 's')

>>> next(I)

(1, 'p')

>>> list(enumerate('spam')) # или с помощью list

[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]

- Обычно весь этот механизм скрыт от нас, т.к. автоматически сипольз-ся циклом for.

Фактически любые инструменты, выполняющие обход объектов слева направо, используют протокол итераций.

**Генераторы списков первое знакомство.** стр. 425

Мы узнали, что функция range изменяет список в ходе итераций:

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

>>> for i in range(len(L)):

L[i] += 10

>>> L

[11, 12, 13, 14]

В след. примере цикл был замещен одним выражением:

>>> L = [x + 10 for x in L]

>>> L

[21, 22, 23, 24]

**- Запомним**, - генераторы списков создают **НОВЫЕ** объекты списков.

**Основы генераторов списков.**

Многие считают, что генераторы списков в Питон похожи на цикл for, записанный задом наперед.

Разберем прошлый пример генератора:

>>> L = [x + 10 for x in L]

Генераторы списков записываются в квадртаных скобках, т.к. они генерируют списки. Гнератор начинается некоего выражения с переменной цикла (х + 10) далее идет загловок цикла for, в котором объявляется переенная цикла и итерируемый объект.

Т.о. обходится список L, присваивая х очередной элемент и собирает рез-ты, пропуская эдементы через выражение слева.

Технически, можно обойтись и без генератора списков:

>>> res = []

>>> for x in L:

res.append(x + 10)

>>> res

[31, 32, 33, 34]

- фактически, это и есть точное представление внутреннего механизма генератора списков.

Но генераторы списков запсиываются компактнее и удобнее. Более того они быстрее чем for.

**Использование генераторов списков для работы с файлами.**

Мы помним, что у объекта файла есть метод readlines, котоырй загружает файл целиком с список строк:

>>> f = open('script1.py')

>>> lines = f.readlines()

>>> lines

['import sys\n', 'print(sys.platform)\n', "x = 'Spam'\n", 'print(x\*3)']

- но каждый элемент списка заканчивается символом \n, что очень не нравится многим программам.

Было бы неплохо эти символы удалить.

Воспользуемся функцией rstrip для этого:

>>> lines = [line.rstrip() for line in lines]

>>> lines

['import sys', 'print(sys.platform)', "x = 'Spam'", 'print(x\*3)']

Мы даже можем открыть файл внутри выражения, т.к. генератор списков будет использ-ть итерационный протокол, который будет читать по одной строке вызовом метода \_\_next\_\_ и выполнять дальнейшие преобразования:

>>> lines = [line.rstrip() for line in open('script1.py')]

>>> lines

['import sys', 'print(sys.platform)', "x = 'Spam'", 'print(x\*3)']

- это наиболее выгодный вариант, т.к. большая часть действий выполняется в интерпретаторе Питон. С большими файлами это может быть большим плюсом.

Рассмотрим другие преобразования:

>>> [line.upper() for line in open('script1.py')]

['IMPORT SYS\n', 'PRINT(SYS.PLATFORM)\n', "X = 'SPAM'\n", 'PRINT(X\*3)']

>>> [line.rstrip() for line in open('script1.py')]

['import sys', 'print(sys.platform)', "x = 'Spam'", 'print(x\*3)']

>>> [line.split() for line in open('script1.py')]

[['import', 'sys'], ['print(sys.platform)'], ['x', '=', "'Spam'"], ['print(x\*3)']]

>>> [line.replace(' ', '!') for line in open('script1.py')]

['import!sys\n', 'print(sys.platform)\n', "x!=!'Spam'\n", 'print(x\*3)']

>>> [('sys' in line, line[0]) for line in open('script1.py')]

[(True, 'i'), (True, 'p'), (False, 'x'), (False, 'p')]

>>> lines

['import sys', 'print(sys.platform)', "x = 'Spam'", 'print(x\*3)']

>>> f = open('script1.py')

>>> lines = f.readlines()

>>> lines

['import sys\n', 'print(sys.platform)\n', "x = 'Spam'\n", 'print(x\*3)']

- с файлом ничего не случилось, ну а lines мы снова изменили и список стал с символами \n.

**Расширенный синтаксис генераторов списков.**

Можем усложнить генератор, добавив в for оператор if.

Например, повторим предыдущий пример, но отберем строки, начинающиеся с символа p:

>>> lines = [line.rstrip() for line in open('script1.py') if line[0] == 'p']

>>> lines

['print(sys.platform)', 'print(x\*3)']

Для понятности перепишем этот пример на базе простого цикла for:

>>> res = []

>>> for line in open('script1.py'):

if line[0] == 'p':

res.append(line.rstrip())

>>> res

['print(sys.platform)', 'print(x\*3)']

- эта инструкция for выполняет то же самое, но медленнее и занимает 4 строки кода.

Также генераторы списков могу тсодержать произвольное количество вложенных циклов for.

Пример: создаем список результатов операции конкатенации x+y для всех x в одной строке и для всех y - в другой. Получаем сочетания символов:

>>> [x + y for x in 'abc' for y in 'lmn']

['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

Это выражение также можно описать классическим способом:

>>> res = []

>>> for x in 'abc':

for y in 'lmn':

res.append(x + y)

>>> res

['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']

Вообще генераторы используются в простых случаях, в сложных решениях лучше будет простая инструкция for.

Вообще, если в программе что-то становится сложно, то, возможно, это не самое лучшее решение - такова идея Питона.

**Другие контексты итераций**. стр. 429

До этого момента мы рассматривали итераторы в контексте инструкции for. Имейте в виду, каждый инструмент, выполняющий обход слева направо, использует итерационный протокол.

Генераторы списков, оператор in, функция map и функции sorted и zip, также основываются на применении итерационного протокола.

Применим их к файлу:

>>> uppers = [line.upper() for line in open('tnt.py')]

>>> uppers

['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X \*\* 33)\n']

>>> map(str.upper, open('tnt.py')) # В 3.0 map возвращает итератор

<map object at 0xb52ddcec>

>>> list(map(str.upper, open('tnt.py')))

['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(X \*\* 33)\n']

>>> 'y = 2\n' in open('tnt.py')

False

>>> 'x = 2\n' in open('tnt.py')

True

- В этом примере функция map вызывает заданную функцию (str.upper) для каждого элемента итерируемого объекта. У map ест ьограничение - ей можно передать только функцию, а не произвольное значение.

Функция sorted сортирует объекты, zip объединяет элементы итерируемых объектов, enumerate создает пары из элементов и их позиций, filter отбирает элементы, для которых указанная функция возвращает "истину", а reduce вып-ет указанную операцию, объединяя все элементы в объекте:

>>> sorted(open('tnt.py'))

['import sys\n', 'print(sys.path)\n', 'print(x \*\* 33)\n', 'x = 2\n']

>>> list(zip(open('tnt.py'), open('tnt.py')))

[('import sys\n', 'import sys\n'), ('print(sys.path)\n', 'print(sys.path)\n'), ('x = 2\n', 'x = 2\n'), ('print(x \*\* 33)\n', 'print(x \*\* 33)\n')]

>>> list(enumerate(open('tnt.py')))

[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'), (3, 'print(x \*\* 33)\n')]

>>> list(filter(bool, open('tnt.py')))

['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x \*\* 33)\n']

>>> import functools, operator

>>> functools.reduce(operator.add, open('tnt.py'))

'import sys\nprint(sys.path)\nx = 2\nprint(x \*\* 33)\n'

- Функция sorted, может принимать любой итерируемый объект, и в отличие от остальных, возвращает новый список, а не итератор.

Функция sum вычисляет сумму чисел в итерируемом объекте (работает только с числами), any и all возвращают True, если любой (any) или все (all) элементы явл-ся истинными значениями. max и min возвращают максимальный и минимальный элемент:

>>> sum([3, 2, 4, 1, 5, 0])

15

>>> any(['spam', '', 'ni'])

True

>>> all(['spam', '', 'ni'])

False

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

5

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

1

max и min могут применяться к файлам, они находят наибольшие наименьшие строки:

>>> max(open('tnt.py'))

'x = 2\n'

>>> min(open('tnt.py'))

'import sys\n'

- применяйте их на свое усмотрение, автор не пишет хорошо или плохо такое применение, наверное не очень.

Многие инструменты используют итерационных протокол. Напр. функции list и tuple, строковый метод join (вставляет подстроку между строками в итерируемом объекте) и даже операция присваивания последовательностей.

>>> list(open('tnt.py'))

['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x \*\* 33)\n'] # Список

>>> tuple(open('tnt.py'))

('import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(x \*\* 33)\n') # Кортеж

>>> '&&'.join(open('tnt.py'))

'import sys\n&&print(sys.path)\n&&x = 2\n&&print(x \*\* 33)\n'

>>> a, b, c, d = open('tnt.py')

>>> a, d

('import sys\n', 'print(x \*\* 33)\n')

>>> a, \*b = open('tnt.py')

>>> a, b

('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(x \*\* 33)\n'])

Ранее мы ознакомились, что функция dict может принимать итераторы, возвращаемые функцией zip. То же самое относится к функции set и новым генераторам множеств и словарей в Питон 3.0:

>>> set(open('tnt.py'))

{'import sys\n', 'x = 2\n', 'print(sys.path)\n', 'print(x \*\* 33)\n'}

>>> {line for line in open('tnt.py')}

{'import sys\n', 'x = 2\n', 'print(sys.path)\n', 'print(x \*\* 33)\n'}

>>> {ix: line for ix, line in enumerate(open('tnt.py'))}

{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(x \*\* 33)\n'}

>>>

Генератор множеств и словарей поддерживают расширенный синтаксис генераторов списков, включаю проверку if:

>>> {line for line in open('tnt.py') if line[0] == 'p'}

{'print(sys.path)\n', 'print(x \*\* 33)\n'}

>>> {ix: line for (ix, line) in enumerate(open('tnt.py')) if line[0] == 'p'}

{1: 'print(sys.path)\n', 3: 'print(x \*\* 33)\n'}

Далее в книге мы будем рассматривать спец. форму аргументов \*arg, которая исп-ся в вызовах функций для распаковывания значений коллекций в отдельные аргументы. Этой форме конструкции можно передать любой итерируемый объект, даже файлы:

>>> def f(a, b, c, d): print(a, b, c, d, sep='&')

...

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

1&2&3&4

>>> f(\*[1, 2, 3, 4]) # Распаковывание списка в аргументы

1&2&3&4

>>> f(\*open('tnt.py')) # Можно даже выполнить обход стрко в файле!

import sys

&print(sys.path)

&x = 2

&print(x \*\* 33)

В виде распаковывания аргументов в вызовах функций можно передавать любые итерируемые объекты. Поэтому zip можно исп-ть для распаковывания упакованных кортежей, передавая ей рез-т другого вызова zip:

>>> x = (1, 2)

>>> y = (3, 4)

>>>

>>> list(zip(x, y))

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

>>>

>>> a, b = zip(\*zip(x, y))

>>> a

(1, 2)

>>> b

(3, 4)

**Новые итерируемые объекты в Питон 3.0** стр. 433

В 3.0 в отличие от 2.6 делается сильный акцент на использование итераторов.

Помимо ассоциации итераторов со многими встроенными типами данных, в 3.0 методы keys, values, items также возвращают итерируемые объекты, как и функции range, map, zip, filter.

В 2.6 не нужно обертывать zip в list, т.к. zip возвращает список результатов, но в 3.0 zip возвращает итератор.

Однако в крупных программах такие особенности 3.0 позволяют экономить память и время на немедленное вычисление результатов.

**Итератор range.**

>>> R = range(10) # Возвращает итератор, а не список

>>> R

range(0, 10)

>>> I = iter(R) # Вернет итератор для дивпазона

>>> next(I) # Переход к след. результату

0 # То же происходит в циулах for, генераторах и пр.

>>> next(I)

1

>>> next(I)

2

>>> list(range(10)) # Если нужно, можно принудительно сгенерировать список

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

В Питон 2.х range возвращает список, а в 3.0 объект возвращаемый range поддерживает только итерации, обращение по индексам и функцию len. Если нужна другая операция, то нужно оборачивать в list:

>>> R[0]

0

>>> R[-1]

9

>>> next(I) # Выборка продолжается с того места где выше мы остановили её

3

>>> I.\_\_next\_\_() # метод работает, но лучше использовать функцию next()

4

**Итераторы map, zip и filter.** стр. 434

Возвращают итераторы и принимают итераторы в виде аргументов. После однократного получения отдельного результата этот результат исчезает.

>>> M = map(abs, (-1, 0, 1)) # возвращает итератор

>>> M

<map object at 0x0171F830>

>>> next(M) # результаты исчерпываются безвозвратно

1

>>> next(M) # len() и индексирование не поддерживается

0

>>> next(M)

1

>>> next(M)

...

StopIteration

>>> for x in M: print(x) # Теперь итератор map пуст:

... # возможен только один проход

>>> M = map(abs, (-1, 0, 1)) # Чтоб второй проход выполнить:

>>> # снова создаем итератор

>>> for x in M: print(x)

1

0

1

>>> list(map(abs, (-1, 0, 1)))

[1, 0, 1]

zip возвращает итератор, который действует точно так же:

>>> Z = zip((1, 2, 3), (10, 20, 30))

>>> Z

<zip object at 0x022880D0>

>>>

>>> list(Z)

[(1, 10), (2, 20), (3, 30)]

>>>

>>> for pair in Z: print(pair)

...

>>> Z = zip((1, 2, 3), (10, 20, 30))

>>> for pair in Z: print(pair)

(1, 10)

(2, 20)

(3, 30)

>>> Z = zip((1, 2, 3), (10, 20, 30))

>>> next(Z)

(1, 10)

>>> next(Z)

(2, 20)

Функция filter возвращает только те элементы итерируемых объектов, для которых указанная функция возвращает значение True:

>>> filter(bool, ['spam', '', 'ni']) # Также возвращает итератор

<filter object at 0x02285910>

>>>

>>> list(filter(bool, ['spam', '', 'ni']))

['spam', 'ni']

**Поддержка множественных и единственных итераторов.**

Различия между результатом range и рез-ми других функций. Рез-т range поддерживает len() и индексирование, но сам не явл-ся итератором (итератор можно получить вызвав функцию iter), кроме того он поддерживает применение к рез-ту несколких итераторов, которые действуют независимо друг от друга:

>>> R = range(3)

>>> next(R)

...

TypeError: 'range' object is not an iterator

>>> I1 = iter(R)

>>> next(I1)

0

>>> next(I1)

1

>>> I2 = iter(R) # Два итератора для одного диапазона

>>> next(I2)

0

>>> next(I1) # I1 находится в другой позиции, не совпадающей с I2

2

zip, map, filter не поддерживают получение различных итераторов для одного и того же рез-та:

>>> Z = zip((1, 2, 3), (10, 20, 30))

>>> I1 = iter(Z)

>>> I2 = iter(Z)

>>> next(I1)

(1, 10)

>>> next(I1)

(2, 20)

>>> next(I2)

(3, 30)

>>> M = map(abs, (-1, 0, 1))

>>> I1 = iter(M); I2 = iter(M)

>>> print(next(I1), next(I1), next(I1))

1 0 1

>>> next(I2)

...

StopIteration

- а для объекта диапазона можно получить множество итераторов:

>>> R = range(3)

>>> I1, I2 = iter(R), iter(R)

>>> [next(I1), next(I1), next(I1)]

[0, 1, 2]

>>> next(I2)

0

**Итераторы представлений словарей.**

Как мы знаем методы словарей keys, values, items возвращают итерируемые объекты представлений, которые возвращают по одному рез-ту за раз, вместо полного списка.

>>> D = dict(a = 1, b = 2, c = 3)

>>> D

{'c': 3, 'a': 1, 'b': 2}

>>> K = D.keys() # Объект представления в 3.0 не явл-ся списком

>>> K

dict\_keys(['c', 'a', 'b'])

>>> next(K) # Представления не явл-ся итераторами

...

TypeError: 'dict\_keys' object is not an iterator

>>> I = iter(K) # Можно получить итератор вручную, но он не поддерж. len()

>>> next(I) # и доступ по индексу

'c'

>>> next(I)

'a'

>>> for k in D.keys(): print(k, end=' ')

...

c a b

Можно вывести весь список с помощью list, он на практике такого обычно не требуется, тольк ов интерактивном сеансе или применить операции над списками, доступ по индексу напр.:

>>> K = D.keys()

>>> list(K)

['c', 'a', 'b']

>>> V = D.values()

>>> V

dict\_values([3, 1, 2])

>>> list(V)

[3, 1, 2]

>>> list(D.items())

[('c', 3), ('a', 1), ('b', 2)]

>>>

>>> for (k, v) in D.items(): print(k, v, end=' ')

c 3 a 1 b 2

В 3.0 словари поддерживают собственные итераторы, которые возвращают последовательности ключей. Т.е. можно не вызывать метод keys:

>>> D

{'c': 3, 'a': 1, 'b': 2}

>>> I = iter(D)

>>> next(I)

'c'

>>> next(I)

'a'

>>> for key in D: print(key, end=' ')

...

c a b

В 3.0 для сортировки нужно преобразовать представление ключей в список с помощью list или вызвать sorted:

>>> D

{'c': 3, 'a': 1, 'b': 2}

>>> for k in sorted(D.keys()): print(k, D[k], end=' ')

...

a 1 b 2 c 3

>>> D

{'c': 3, 'a': 1, 'b': 2}

>>> for k in sorted(D): print(k, D[k], end=' ') # Лучший способ сортировки ключей

...

a 1 b 2 c 3

**Позднее мы узнаем, что:**

- С помощью yield пользовательские функции можно превратить в итерируемые функции-генераторы.

- Генераторы списков можно трансф-ть в итерируемые выражения-генераторы, заключив их в круглые скобки.

- В пользовательские классы можно добавить поддержку итераций с помощью методов перегрузки операторов \_\_iter\_\_ и \_\_getitem\_\_

**Глава 15. Документация.** стр. 441

**Источники документации в Питон.**

**Таблица 15.1 Источники документации в языке Питон.** стр. 442

**Комментарии #**

- Это самый элементарный способ документирования программного кода. Интерпретатор игнорирует текст после символа решетки, только если он не внутри строкового литерала. В наст. момент считается, что стркои документирования лучше подходят для создания функционального описания ("Мой файл делает то-то и то-то"), а комментарии больше подходят для особенностей программного кода ("Это выражение делает то-то и то-то").

**Функция dir.**

- dir это простой способ получить все атрибуты объекта (т.е. методы и элементы данных). Напр. узнаем, что есть в стандарт. библиотечном модуле sys:

>>> import sys

>>> dir(sys)

['\_\_displayhook\_\_', '\_\_doc\_\_', '\_\_excepthook\_\_', '\_\_interactivehook\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', '\_\_stderr\_\_', '\_\_stdin\_\_', '\_\_stdout\_\_', '\_clear\_type\_cache', '\_current\_frames', '\_debugmallocstats', '\_getframe', '\_home', '\_mercurial', '\_xoptions', 'api\_version', 'argv', 'base\_exec\_prefix', 'base\_prefix', 'builtin\_module\_names', 'byteorder', 'call\_tracing', 'callstats', 'copyright', 'displayhook',

…часть текста опущена…

Чтобы узнать атрибуты встроенных типов, нужно передать функции литерал или объект типа, например посмотрим атрибуты списов и строк:

>>> dir([])

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_',

…часть текста опущена…

>>> dir('')

['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mod\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_rmod\_\_', '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format\_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

Рез-ты dir для любого встроенного типа включают набор атрибутов, которые имеют отношение к реализации этого типа, они начин-ся и заканч-ся двумя подчерками, пока можно их игнорировать.

Можно передавать имя типа вместо литерала:

>>> dir(str) == dir('')

True

>>> dir(list) == dir([])

True

- Это работает, т.к имена функций преобразования (str, list), в Питон фактически явл-ся именами типов - вызов любого из этих конструкторов приводи тк созданию экземпляра этого типа.

dir предоставляет список имен атрибутов, но ничего не сообщает о том, что они означают.

**Строки документирования: \_\_doc\_\_**

- Эта документация автоматически присоединяется к объектам и доступна во время выполнения. Синтаксически, такие строки распол-ся в начале файлов модулей, функций и классов, перед кодом. Интерпретатор автоматически помещает строки документирования в атрибут \_\_doc\_\_ соответствующего объекта.

Строки документирования, определяемые пользователем.

Пример будет файл - docstring.py (мы создадим его сами). Инструкции def и class можно просто игнорировать пока:

"""

Module documentation

Words Go Here

"""

spam = 40

def square(x):

"""

function domentation

can we have your liver then?

"""

return x \*\* 2

class Employee:

"class documentation"

pass

print(square(4))

print(square.\_\_doc\_\_)

Самое главное в протоколе документирования, то, что комментарии доступны для просмотра в виде атрибутов \_\_doc\_\_ после того как файл импортирован. Примеры вывода нужной документации:

>>> import docstring

16

function domentation

can we have your liver then?

>>> print(docstring.\_\_doc\_\_)

Module documentation

Words Go Here

>>> print(docstring.square.\_\_doc\_\_)

function domentation

can we have your liver then?

>>> print(docstring.Employee.\_\_doc\_\_)

class documentation

Заметим, что нужно использовать print, иначе будет выводиться единственная строка со встроенными символами новой строки:

>>> docstring.square.\_\_doc\_\_

'\n\tfunction domentation\n\tcan we have your liver then?\n\t'

Чтобы извлечь строку с описанием метода класса, внутри модуля, нужно **использ**. след. синтаксис:

module.class.method.\_\_doc\_\_

**Стандарты оформления строк документирования.**

Стандартов по сути нет, автор прост настоятельно рекомендует делать документацию и комментарии.

**Встроенные строки документирования.**

Пример, документация встроен. модуля sys:

>>> import sys

>>> print(sys.\_\_doc\_\_)

This module provides access to some objects used or maintained by the

interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known

path -- module search path; path[0] is the script directory, else ''

modules -- dictionary of loaded modules

...остальной текст пропущен, его там много дальше...

Можно смотреть описание функций, классов, методов по их атрибутам:

>>> print(sys.getrefcount.\_\_doc\_\_)

getrefcount(object) -> integer

Return the reference count of object. The count returned is generally

one higher than you might expect, because it includes the (temporary)

reference as an argument to getrefcount().

>>> print(int.\_\_doc\_\_)

int(x=0) -> integer

int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments

are given. If x is a number, return x.\_\_int\_\_().

>>> print(map.\_\_doc\_\_)

map(func, \*iterables) --> map object

Make an iterator that computes the function using arguments from

each of the iterables. Stops when the shortest iterable is exhausted.

**PyDoc: функция help.** стр. 446

Инструмент Питона PyDoc умеет извлекать строки документирования вместе с информацией о структуре программных компонентов и формирует отчет.

Два, самых заметных интерфейса в PyDoc, это функция help и графический интерфейс для отчетов в формате HTML.

>>> import sys

>>> help(sys.getrefcount)

Help on built-in function getrefcount in module sys:

getrefcount(...)

getrefcount(object) -> integer

Return the reference count of object. The count returned is generally

one higher than you might expect, because it includes the (temporary)

reference as an argument to getrefcount().

- мы импортировали модуль sys, чтобы выяснить информацию по модулю, функция help работает и без импорта модуля.

Еще пример:

>>> help(sys)

Help on built-in module sys:

NAME

sys

MODULE REFERENCE

http://docs.python.org/3.4/library/sys

The following documentation is automatically generated from the Python

source files. It may be incomplete, incorrect or include features that

are considered implementation detail and may vary between Python

implementations. When in doubt, consult the module reference at the

location listed above.

DESCRIPTION

This module provides access to some objects used or maintained by the

interpreter and to functions that interact strongly with the interpreter.

...остальной текст опущен...

FUNCTIONS

\_\_displayhook\_\_ = displayhook(...)

displayhook(object) -> None

Print an object to sys.stdout and also save it in builtins.\_

...остальной текст опущен...

DATA

\_\_stderr\_\_ = None

\_\_stdin\_\_ = None

\_\_stdout\_\_ = None

...остальной текст опущен, там очень много дальше...

Также можно получить информацию о встроенном типе, передав функции имя типа (dict - для словарей, str - для строк, list - для списков и т.д.):

>>> help(dict)

Help on class dict in module builtins:

class dict(object)

| dict() -> new empty dictionary

| dict(mapping) -> new dictionary initialized from a mapping object's

...далее текст опущен...

>>> help(str.replace)

Help on method\_descriptor:

replace(...)

S.replace(old, new[, count]) -> str

Return a copy of S with all occurrences of substring

old replaced by new. If the optional argument count is

given, only the first count occurrences are replaced.

>>> help(ord)

Help on built-in function ord in module builtins:

ord(...)

ord(c) -> integer

Return the integer ordinal of a one-character string.

Также help может извлекать информацию не только из встроенных модулей, напр. из нашего docstring.py:

>>> import docstring

16

function domentation

can we have your liver then?

>>> help(docstring.square)

Help on function square in module docstring:

square(x)

function domentation

can we have your liver then?

>>> help(docstring.Employee)

Help on class Employee in module docstring:

class Employee(builtins.object)

| class documentation

|

| Data descriptors defined here:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

>>> help(docstring)

Help on module docstring:

NAME

docstring

DESCRIPTION

Module documentation

Words Go Here

CLASSES

builtins.object

Employee

class Employee(builtins.object)

| class documentation

|

| Data descriptors defined here:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

FUNCTIONS

square(x)

function domentation

can we have your liver then?

DATA

spam = 40

FILE

c:\python34\docstring.py

>>>

**PyDoc: отчеты в формате HTML.**

Сценарий Python/tkinter создает отчеты в формате HTML, для просомтра в браузере, причем как локально так и удаленно.

Запускаем поисковый механизм, с графич. интерфейсом из меню Пуск, Документация модулей (у меня это было Docs Server) или запустить pydocgui.pyw из каталога Tools/Scripts.

Ищем в поиске нужный нам модуль (напр. glob), и в найденных результатах жмем на нужный нам. Далее откроется описание модуля, с ссылками на импортированные модули и др.

Можно посмотреть и наш docstring, набрав в поиске его имя.

Также обращайтесь ко всем возможным средствам поиска информации будь-то веб или печатные издания или документация Питона.

**Типичные ошибки программирования.**

- **Не забывайте про двоеточия**. Обязательно ставьте двоеточия, не забывайте, будьте внимательны.

- **Начинайте с первой позиции в строке**. Иначе получите сообщение об ошибке, в первой позиции никаих пробелов и отступов не нужно.

- **Пустые строки имеют особый смысл в интерактивной оболочке**. Внутри файлов модулей пустые строки просто игнорируются, но в интерактивной оболочке это специальные строки в составных инструкциях.

- **Используйте отступы непротиворечивым способом**. Не смешивайте пробелы и символы табуляции при оформлении отступов. Это справедливо для любого ЯП. Используйте или пробелы или табуляцию, но НЕ одновременно вместе.

- **Не пишите на языке С**. Не нужно заключать в скобки условные выражения if и while, можно, но это лишнее в Питон. Не нужно точек с запятой, можно, но в Питон это, опять же, лишнее. И помните - не встраивайте инструкции присваивания в условные выражения циклов while и не заключайте блоки в фигурные скобки.

- **Вместо циклов while и функции range старайтесь использовать простые циклы for**. Простые циклы for почти всегда быстрее и проще, т.к. в основном нагружен сам интерпретатор. И избегайте искушения считать что-либо в циклах на языке Питон.

- **Будьте внимательны, выполняя присваивание изменяемых объектов**. Напр. (a = b = [])или (a += [1, 2]) и прочих. Помните о том, что возможно изменение других переменных при изменении одной.

- **Не ожидайте получения результатов от функций, выполняющих непосредственное изменение в объектах**. Напр. методы append или sort, не возвращают ничего (кроме None), поэтмоу их нужно вызыват ьбез присвоения значения. Ошибки: mylist = mylist.append(X) - в переменную mylist запишется ссылка на объект None, а не на измененный список, в итоге список мы потеряем совсем.

- **Всегда используйте круглые скобки при вызове функций**. Даже елси аргументв онет все равно вызываем в видле function(). Например если мы попытаемся закрыть файл file.close то ошибки не будет (т.к. обращение к методу может быть без скобок в Питоне), но файл не закроется!

- **Не используйте расширения имен файлов в инструкциях import и reload**. Синтаксис определения пути зависит от типа платформы и определяется настройками параметра пути поиска модулей, а не инструкцией import, поэтому указание расширений будет не только ошибкой, но оно и не нужно.

**Упражнения к третьей части книги. стр. 457.**

Часть. 4. Функции. стр. 459

Глава 16. Основы функций. стр. 461