

# Погружение в Python 3

1. Что нового в «Погружении в Python 3»
2. Установка Python
3. Ваша первая программа на Python
4. Встроенные типы данных
5. Генераторы
6. Строки
7. Регулярные выражения
8. Замыкания и генераторы
9. Классы и итераторы
10.      Подробнее об итераторах
11.      Тестирование
12.      Рефакторинг
13.      Файлы
14.      XML
15.      Сериализация объектов Python
16.      HTTP и веб-сервисы
17.      Пример: перенос chardet на Python 3
18.      Создание пакетов библиотек
19.      Перенос кода на Python 3 с помощью 2to3
20.      Особые названия методов
21.      Куда пойти
22.      Устранение проблем
23.      О книге
24.      О переводе
25.      Выходные данные

---

**Автор:** Марк Пилигрим (Mark Pilgrim)

**Перевод:** Инициативная группа

**Источник оригинала:** [ru.wikisource.org](http://ru.wikisource.org)

**Лицензия:** данный материал распространяется по лицензии GNU FDL.

# Что нового в «Погружении в Python 3»

Isn't this where we came in?

*Pink Floyd, The Wall*

*или «минус первый уровень»*

Вы прочитали «Погружение в Python» и, может быть, даже купили бумажную версию. (Спасибо!) Вы уже неплохо знаете Python 2. Вы готовы окунуться с головой в Python 3... Если всё это про вас, читайте дальше. (Если что-либо из этого неверно, вам следует начать с начала.)

Вместе с Python 3 поставляется скрипт под названием `2to3`. Изучите его. Полюбите его. Используйте его. «Перенос кода на Python 3 с помощью `2to3`» — справочник по всем тем изменениям, которые `2to3` может проделать автоматически. Поскольку многое из этого — изменения в синтаксисе, лучше всего начать именно с них. (`print` теперь функция, ``x`` не работает и т. д.)

«Пример: перенос `chardet` на Python 3» рассказывает о моей (в конце концов, успешной) попытке перенести одну нетривиальную библиотеку с Python 2 на Python 3. Это может вам помочь. А может и не помочь. Кривая обучения получается довольно крутой из-за того, что вам сначала необходимо хоть немного разобраться в этой библиотеке, чтобы понимать, что именно в ней поломалось и как я это исправил. Часто программы ломаются на строках (strings). Кстати, о строках...

Строки... Ох... С чего бы начать?... В Python 2 были *строки* (`str`) и *юникоднЫе строки* (`unicode`). В Python 3 — *байты* (`bytes`) и *строки* (`string`). То есть все строки стали теперь юникодными, а когда хотите работать с кучей байтов, вы используете новый тип `bytes`. Python 3 *никогда* автоматически не преобразовывает строки в байты или наоборот, поэтому, если вы не уверены, что из них вы имеете в какой-то определённый момент, ваш код почти наверняка поломаётся. В главе о строках эта ситуация разбирается подробнее.

Байты и строки ещё не раз будут появляться в книге.

- В главе «Файлы» вы узнаете разницу между чтением файлов в «двоичном» и «текстовом» режимах. Чтение (и запись!) файлов в текстовом режиме требует указание параметра `encoding` (кодировка). Одни файловые методы при работе с текстовыми файлами подсчитывают символы, другие считают байты. Если в вашей программе предполагается, что один символ == одному байту, она *будет* падать на многобайтовых символах.
- В главе «HTTP и веб-сервисы» модуль `httplib2` принимает заголовки и данные по HTTP. Заголовки HTTP возвращаются как строки, а тело ответа HTTP возвращается в виде байтов.
- В главе «Сериализация объектов Python» вы узнаете, почему модуль `pickle` в Python 3 определяет новый формат данных, несовместимый с Python 2. (Подсказка: это из-за различий между байтами и строками.) Ещё есть JSON, который вообще не поддерживает тип `bytes`. Я покажу вам, как справиться с этим ограничением.
- Глава «Пример: перенос `chardet` на Python 3» — просто кровавое месиво из байтов и строк.

Даже если вы не задумываетесь о Юникоде (рано или поздно всё равно придётся), вы захотите прочитать о форматировании строк в Python 3, оно совершенно отличается от такового в Python 2.

Итераторы в Python 3 везде, и сейчас я разбираюсь в них намного лучше, чем пять лет назад, когда я писал «Погружение в Python». Вам тоже необходимо разобраться в них, потому что множество функций, которые в Python 2 возвращали списки, теперь в Python 3 возвращают итераторы. Как минимум, следует прочитать вторую половину главы «Итераторы» и вторую половину главы «Подробнее об итераторах».

По многочисленным просьбам я добавил приложение «Особые названия методов», которое немного похоже на главу «Модель данных» документации по Python, только с приколами.

Когда я писал «Погружение в Python», все доступные на тот момент библиотеки для работы с XML были жутким отстоем. Впоследствии Фредрик

Лунд (Fredrik Lundh) написал ElementTree, которая уже совсем не отстой. Питоновские боги, показав свою мудрость, внедрили ElementTree в стандартную библиотеку, и теперь она составляет основу моей новой главы про XML. Старые способы разбора XML всё ещё доступны, но их надо избегать, потому что они отстойные!

Ещё из нового в Python — не в языке, а в сообществе — появление репозитория в виде The Python Package Index (PyPI). К Python прилагаются утилиты для пакетирования вашего кода в стандартных форматах и дальнейшего его распространения через PyPI. Подробнее в главе «Пакетирование библиотек Python».

---

# Установка Python

Tempora mutantur nos et mutamur in illis. ~ Меняются времена, и мы меняемся вместе с ними.

*древнеримская поговорка*

## Погружение

Добро пожаловать в Python 3. В этой главе вы установите Python 3, ту его версию, которая подходит именно вам.

## Какой Python вам подходит?

Первое, что вам необходимо проделать с Python, — установить его. Или это уже сделано?

Если вы собираетесь работать с Python на удалённом сервере, ваш хостинг-провайдер, возможно, уже установил Python 3. Если у вас домашний компьютер с Linux, Python 3 тоже может быть уже установлен. В большинстве популярных дистрибутивах GNU/Linux по умолчанию установлен Python 2, немногие (но их число растёт) также включают Python 3. Mac OS X включает консольную версию Python 2, но до сих пор не включает Python 3. В Microsoft Windows не входит никакая версия Python. Но не отчаивайтесь! Python можно установить в несколько кликов, независимо от вашей операционной системы.

Простейший способ проверить, установлен ли Python 3 в вашем Linux или Mac OS X, — это открыть командную строку. В Linux поищите программу **«Терминал»** (**«Terminal»**) в меню приложений (**«Applications»**). Она может находиться в подменю **«Стандартные»** (**«Accessories»**) или **«Системные утилиты»** (**«System»**). В Mac OS X в папке `/Application/Utilities/` должно быть приложение **«Terminal.app»**.

Получив приглашение командной строки, просто введите `python3` (строчными буквами, без пробелов) и посмотрите, что произойдёт. На моей домашней Linux-системе Python 3 уже установлен, и эта команда запускает *интерактивную оболочку Python*.

```
mark@atlantis:~$ python3
Python 3.0.1+ (r301:69556, Apr 15 2009, 17:25:52)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(Чтобы выйти из интерактивной оболочки Python, введите `exit()` и нажмите `↵ Enter`.)

Мой хостинг-провайдер тоже предоставляет Linux с доступом к командной строке, но Python 3 на сервере не установлен. (Фу!)

```
mark@manganese:~$ python3
bash: python3: command not found
```

Итак, вернёмся к вопросу, с которого начался этот раздел: «Какой Python вам подходит?» Любой, который работает на вашем компьютере.

---

*Читайте далее инструкции по установке на Windows или перейдите к [установке на Mac OS X](#), [на Ubuntu Linux](#) или на [другие платформы](#).*

## Установка на Microsoft Windows

Сегодня существуют Windows для двух архитектур: 32-битной и 64-битной. Конечно, существуют и разные *версии* Windows — XP, Vista, Windows 7 — Python работает на всех них. Важнее различие между 32-битной и 64-битной архитектурами. Если вы не знаете, какая архитектура вашего компьютера, это, вероятно, 32 бита.

Посетите [python.org/download/](http://python.org/download/) и скачайте соответствующий установочный пакет Python 3 для вашей архитектуры Windows. Ваш выбор будет примерно таким:

- **Python 3.1 Windows installer** (Windows binary — does not include source)
- **Python 3.1 Windows AMD64 installer** (Windows AMD64 binary — does not include source)

Я не хочу публиковать здесь прямые ссылки, потому что в Python постоянно происходят мелкие обновления, и я не хочу оказаться крайним, если вы вдруг пропустите важные обновления. Устанавливайте самую свежую версию Python 3.x, если только у вас нет каких-либо объективных причин поступить иначе.

1. Диалог Windows: предупреждение о безопасности при открытии файла  
По окончании загрузки откройте .msi-файл. Windows покажет предупреждение о безопасности, потому что вы пытаетесь запустить исполнимый код. Официальный установочный пакет Python имеет цифровую подпись Python Software Foundation, некоммерческой организации, курирующей разработку Python. Опасайтесь подделок!

Нажмите кнопку «Запустить» («Run»), чтобы запустить программу установки Python 3.

2. Программа установки Python: выбор типа установки Python 3.1 Первый вопрос, который задаёт программа установки: установить Python 3 для всех пользователей компьютера или только для вас. По умолчанию выбран ответ «установить для всех пользователей», и если у вас нет причин выбирать другой вариант, его следует оставить. (Одна из возможных причин установки «только для себя» — это установка на рабочий компьютер, где ваша учётная запись не имеет административных полномочий. Но в таком случае почему вы устанавливаете Python без разрешения системного администратора? Не впутывайте меня в неприятности!)

Нажмите кнопку «Далее» («Next»), чтобы подтвердить выбор типа установки.

3. Программа установки Python: выбор директории назначения Затем программа установки предложит выбрать директорию назначения. По умолчанию все версии Python 3.1.x предлагают установку в директорию `C:\Python31\`, для большинства пользователей это должно работать, если у вас нет особых причин изменить этот параметр, не меняйте его. Если у вас отдельный логический диск для установки приложений, вы можете выбрать его, пользуясь встроенными инструментами, или просто вписать путь в соответствующем поле ввода. Python можно установить не только на диск `C:`, но на любой диск, в любую папку.

Нажмите кнопку «Далее» («Next»), чтобы подтвердить выбор директории назначения.

4. Программа установки Python: выбор компонентов Python 3.1 Следующая страница выглядит сложно, но на самом деле это не так. Как во многих других программах установки, у вас есть возможность отказаться от



установки любого из компонентов Python 3. Если свободного пространства на диске совсем мало, вы можете исключить некоторые компоненты.

- Опция **Регистрировать расширения (Register Extensions)** позволяет вам запускать скрипты Python (файлы с расширением `.py`) двойным кликом по иконке. Рекомендуется, но не обязательно. (Эта опция не занимает места на диске, поэтому нет особого смысла в её исключении.)
- **Tcl/Tk** — это графическая библиотека, используемая оболочкой Python, которая будет использоваться на протяжении всей книги. Я настоятельно рекомендую оставить эту опцию.
- Опция **Документация (Documentation)** устанавливает файл справки, содержащий значительную часть информации с `docs.python.org`. Рекомендуется, если у вас dial-up или ограниченный доступ к Интернету.
- **Полезные скрипты (Utility Scripts)** включают скрипт `2to3.py`, подробнее о котором вы узнаете ниже. Необходимо, если вы хотите узнать о переносе на Python 3 существующего кода, написанного на Python 2. Если у вас нет существующего кода на Python 2, можете убрать эту опцию.
- **Тестовый набор (Test Suite)** — это коллекция скриптов, используемых для тестирования самого Python. В этой книге мы их использовать не будем, да и я никогда не использовал при программировании на Python. Совершенно необязательно.

5. Программа установки Python: требования к дисковому пространству Если вы не знаете точно, сколько у вас дискового пространства, нажмите кнопку «Использование диска» («Disk Usage»). Программа установки покажет список логических дисков, посчитает, сколько пространства доступно на каждом из них и сколько останется после установки.

Нажмите кнопку «ОК», чтобы вернуться на страницу выбора компонентов.

6. Программа установки Python: отключение опции «Тестовый набор» экономит 7908 Кбайт на жёстком диске Если вы решите отключить опцию, нажмите кнопку перед ней и в выпавшем меню выберите «Компонент полностью недоступен» («Entire feature will be unavailable»). Например, исключение тестового набора сэкономит вам 7908 Кбайт дискового пространства.

Нажмите кнопку «Далее» («Next»), чтобы подтвердить выбор опций.

7. Программа установки Python: индикатор прогресса Программа установки копирует все необходимые файлы в выбранную директорию назначения. (Это происходит так быстро, что скриншот удалось сделать только с третьей попытки!)



8. [[Программа установки Python: установка завершена. Выражаю особую виндовую благодарность Марку Хэммонду (Mark Hammond), щедро посвятившему много лет работе на Windows-направлению, без него Python для Windows был бы всё ещё Python для DOS.]] Нажмите кнопку «Готово» («Finish»), чтобы закрыть программу установки. [war-robin.com]
9. Оболочка Python для Windows, графическая интерактивная оболочка для Python В меню «Пуск» должен появиться новый пункт под названием Python 3.1. Внутри него будет программа IDLE. Кликните на ней, чтобы запустить интерактивную оболочку Python.

---

*Перейти к [использованию оболочки Python](#).*

## Установка на Mac OS X

Все современные компьютеры Macintosh используют процессоры Intel (как и большинство компьютеров с Windows). Старые Mac-и использовали процессоры PowerPC. Вам не обязательно понимать разницу между ними, потому что для всех Mac-ов есть один установочный пакет.

Посетите [python.org/download/](http://python.org/download/) и загрузите установочный пакет для Macintosh. Он будет называться примерно так: **Python 3.1 Mac Installer Disk Image**, номер версии может быть другим. Загружайте именно версию 3.x, а не 2.x.

1. Содержимое установочного образа Python Ваш браузер должен автоматически смонтировать образ диска и открыть окно Finder, чтобы показать вам его содержимое. (Если это не произошло, вам необходимо найти образ диска в папке загрузок и смонтировать его, кликнув на нём дважды. Он будет называться примерно так: `python-3.1.dmg`.) Образ диска содержит несколько текстовых файлов (`Build.txt`, `License.txt`, `ReadMe.txt`) и собственно установочный пакет `Python.mpkg`.

Дважды кликните на установочном пакете `Python.mpkg`, чтобы запустить программу установки Python.

2. Программа установки Python: экран приветствия Первая страница программы установки даёт краткое описание и отсылает к файлу `ReadMe.txt` (который вы не читали, ведь так?) за более подробными сведениями.

Нажмите кнопку «Продолжить» («Continue») для продолжения установки.

3. Программа установки Python: сведения о поддерживаемых архитектурах, дисковом пространстве и допустимых папок назначения Следующая страница содержит действительно важные сведения: для Python требуется Mac OS X 10.3 или более поздняя версия. Если вы всё ещё

используете Mac OS X 10.2, вам действительно надо обновиться. Apple перестала выпускать обновления безопасности для вашей операционной системы, и компьютер находится под возможной угрозой, даже когда просто подключается к Интернету. Кроме того, на ней не работает Python 3.

Нажмите кнопку «Продолжить» («Continue»), чтобы идти дальше.

4. Программа установки Python: лицензионное соглашение Как все порядочные программы установки, программа установки Python показывает лицензионное соглашение об использовании программного обеспечения. Python — это открытое программное обеспечение, и его лицензия одобрена организацией Open Source Initiative. На протяжении истории Python у него были разные владельцы и спонсоры, каждый из которых оставил свой след в лицензии. Но конечный результат таков: исходный код Python открыт, и его можно использовать на любой платформе, для любых целей, без платы и обязательств.

Нажмите кнопку «Продолжить» («Continue») ещё раз.

5. Программа установки Python: диалог принятия лицензионного соглашения Из-за особенностей стандартного механизма установки Apple вы должны «согласиться» с лицензией, чтобы выполнить установку. Поскольку Python — открытое программное обеспечение, «согласие» с лицензией скорее расширяет ваши права, нежели ограничивает их.

Нажмите кнопку «Согласен» («Agree») для продолжения.

6. Программа установки Python: стандартный экран установки Следующий экран позволяет изменить место установки. Python **обязательно** надо устанавливать на системный диск, но из-за ограничений программы установки это не проверяется. По правде говоря, мне никогда не приходилось изменять место установки.

В этом экране можно также уточнить список устанавливаемых компонентов, выбрав или исключив некоторые из них. Если вы хотите это сделать, нажмите кнопку «Компоненты» («Customize»), в противном случае нажмите «Установить» («Install»).

7. Программа установки Python: экран выборочной установки Если вы хотите произвести выборочную установку, программа установки покажет следующий список компонентов:
  - **Фреймворк Python (Python Framework).** Это основная часть Python, она всегда выбрана и неактивна, потому что должна быть обязательно установлена.

- **Графические приложения (GUI Applications)** включают IDLE — графическую оболочку Python, которую вы будете использовать на протяжении всей книги. Я настоятельно рекомендую оставить эту опцию включённой.
- **Инструменты командной строки UNIX (UNIX command-line tools)** включают приложение командной строки `python3`. Эту опцию я тоже настоятельно рекомендую оставить.
- **Документация по Python (Python Documentation)** содержит значительную часть информации с `docs.python.org`. Рекомендуется, если у вас dial-up или ограниченный доступ к Интернету.
- **Инструмент обновления профиля оболочки (Shell profile updater)** управляет обновлением вашего профиля оболочки (используемого в «Terminal.app») и обеспечивает нахождение данной версии Python в путях поиска программ вашей оболочки<sup>[1]</sup>. Вероятно, вам не потребуется изменять этот пункт.
- Опцию **Исправить системный Python (Fix system Python)** изменять не нужно. (Она заставляет ваш «мак» использовать Python 3 как интерпретатор по умолчанию для всех скриптов на Python, включая встроенные системные скрипты от Apple. Будет очень плохо, потому что большинство скриптов написаны на Python 2, и они перестанут правильно работать под Python 3.)

Нажмите кнопку «Установить» («Install») для продолжения.

8. Программа установки Python: диалог ввода пароля администратора Для того, чтобы установить системные фреймворки и библиотеки в `/usr/local/bin/`, программа установки спросит у вас пароль администратора. Без привилегий администратора установить Python на Mac нельзя.

Нажмите кнопку «ОК», чтобы начать установку.

9. Программа установки Python: индикатор прогресса Программа установки будет показывать индикатор прогресса во время установки выбранных компонентов.
10. Программа установки Python: установка выполнена Если всё пройдет правильно, программа установки покажет большую зеленую галку, означающую, что установка завершена успешно.

Нажмите кнопку «Закрыть» («Close»), чтобы выйти из программы установки.

11. Содержимое папки `/Applications/Python 3.1/` Если вы не меняли место установки, свежее установленные файлы будут располагаться в

папке **Python 3.1** внутри папки **/Applications**. Наиболее важная её часть — **IDLE**, графическая оболочка Python.

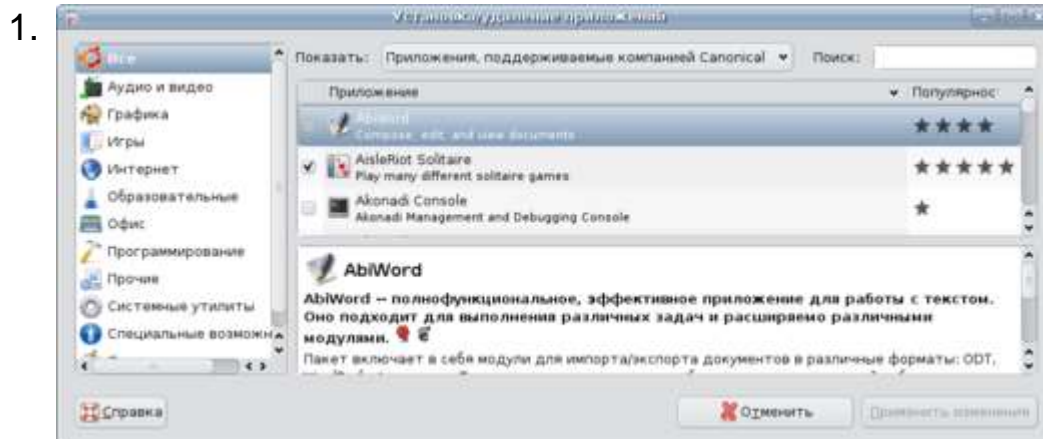
Дважды кликните по **IDLE**, чтобы запустить оболочку Python.

12. Графическая интерактивная оболочка Python на Mac Оболочка Python — это то место, где вы проведёте большую часть времени, исследуя Python. Во всех примерах в этой книге предполагается, что знаете, как найти оболочку Python.

-----  
*Перейти к [использованию оболочки Python](#).*

## Установка на Ubuntu Linux

Современные дистрибутивы Linux подкреплены обширными репозиториями предкомпилированных приложений (пакетов), готовых к установке. Точные сведения могут отличаться от дистрибутива к дистрибутиву. В Ubuntu Linux самый простой способ установить Python 3 — через приложение «Установка/удаление» («Add/Remove») в меню «Приложения» («Applications»).

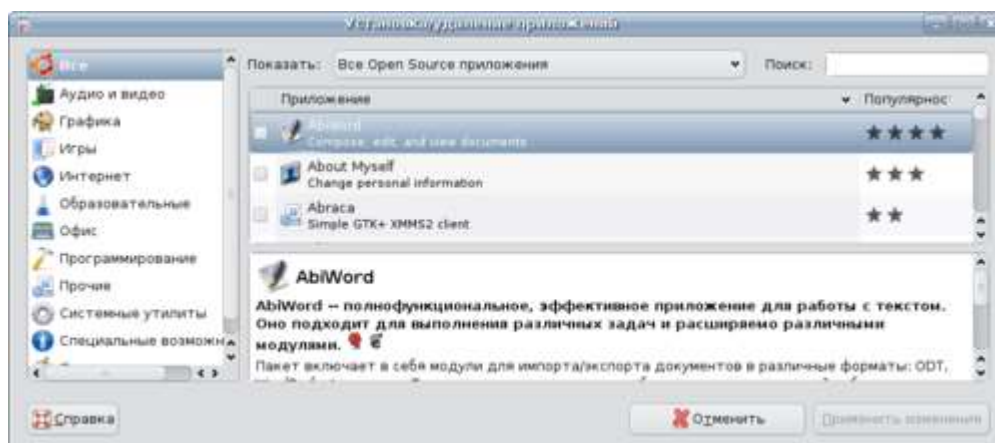


Установка/удаление: приложения, поддерживаемые компанией Canonical

Когда вы впервые запускаете «Установку/удаление», отображается список приложений по категориям. Некоторые из них уже установлены, но большая часть — нет. Репозиторий содержит более 10 000 приложений, поэтому вы можете применить различные фильтры, чтобы просмотреть меньшие части репозитория. Фильтр по умолчанию — «Приложения, поддерживаемые компанией Canonical» («Canonical-maintained applications») — показывает небольшое подмножество из общего числа приложений, только те, что официально поддерживаются

компанией Canonical, создающей и поддерживающей Ubuntu Linux.

2.



Установка/удаление: все Open Source приложения

Python 3 не поддерживается Canonical, поэтому сначала выберите из выпадающего меню фильтров «Все Open Source приложения» («All Open Source applications»).

3.



Установка/удаление: поиск «python 3»

После переключения фильтра на отображение всех открытых приложений сразу же воспользуйтесь строкой поиска, чтобы найти «python 3».



Установка/удаление: выбор пакета Python 3.0

Теперь список приложений сократился до тех, которые соответствуют запросу «python 3». Нужно отметить два пакета. Первый — «Python (v3.0)». Он содержит собственно интерпретатор Python.



Установка/удаление: Выбор пакета IDLE для Python 3.0

Второй пакет, который вам нужен, находится непосредственно над первым — «IDLE (using Python-3.0)». Это графическая оболочка Python, которую вы будете использовать на протяжении всей книги.

После того, как вы отметите эти два пакета, нажмите кнопку «Применить изменения» («Apply Changes») для продолжения.



6.



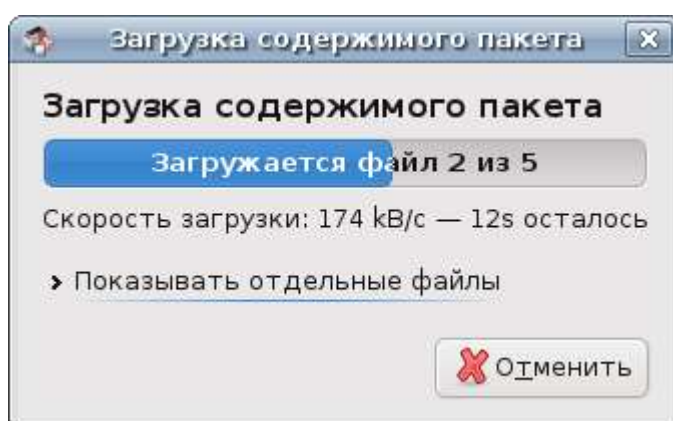
Установка/удаление:  
применение изменений

Программа управления пакетами попросит подтвердить, что вы хотите установить два пакета — «IDLE (using Python-3.0)» и «Python (v3.0)».

Нажмите кнопку «Применить» («Apply»)

для продолжения.

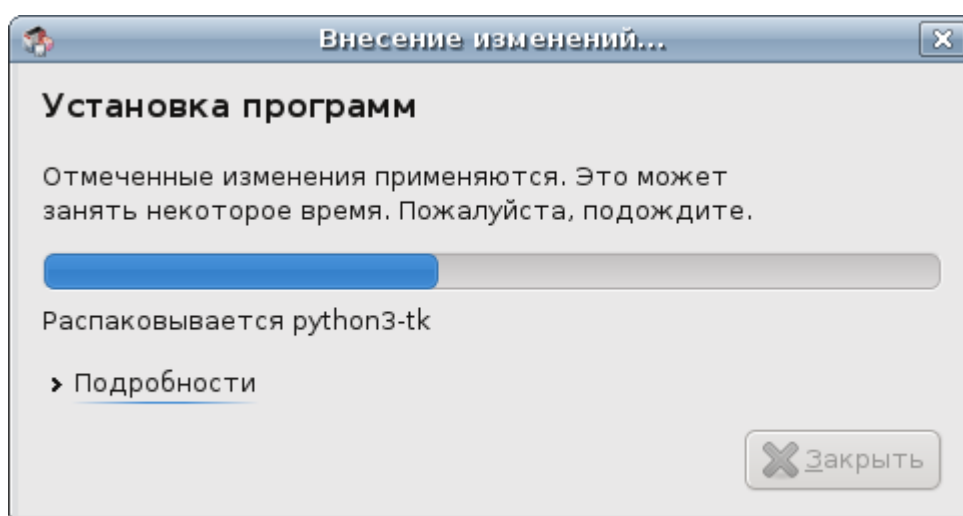
7.



Установка/удаление: индикатор выполнения загрузки

Программа управления пакетами будет показывать индикатор выполнения во время загрузки необходимых пакетов из Интернет-репозитория Canonical.

8.



Установка/удаление: индикатор выполнения установки



После загрузки пакетов программа управления пакетами автоматически начнёт устанавливать их.

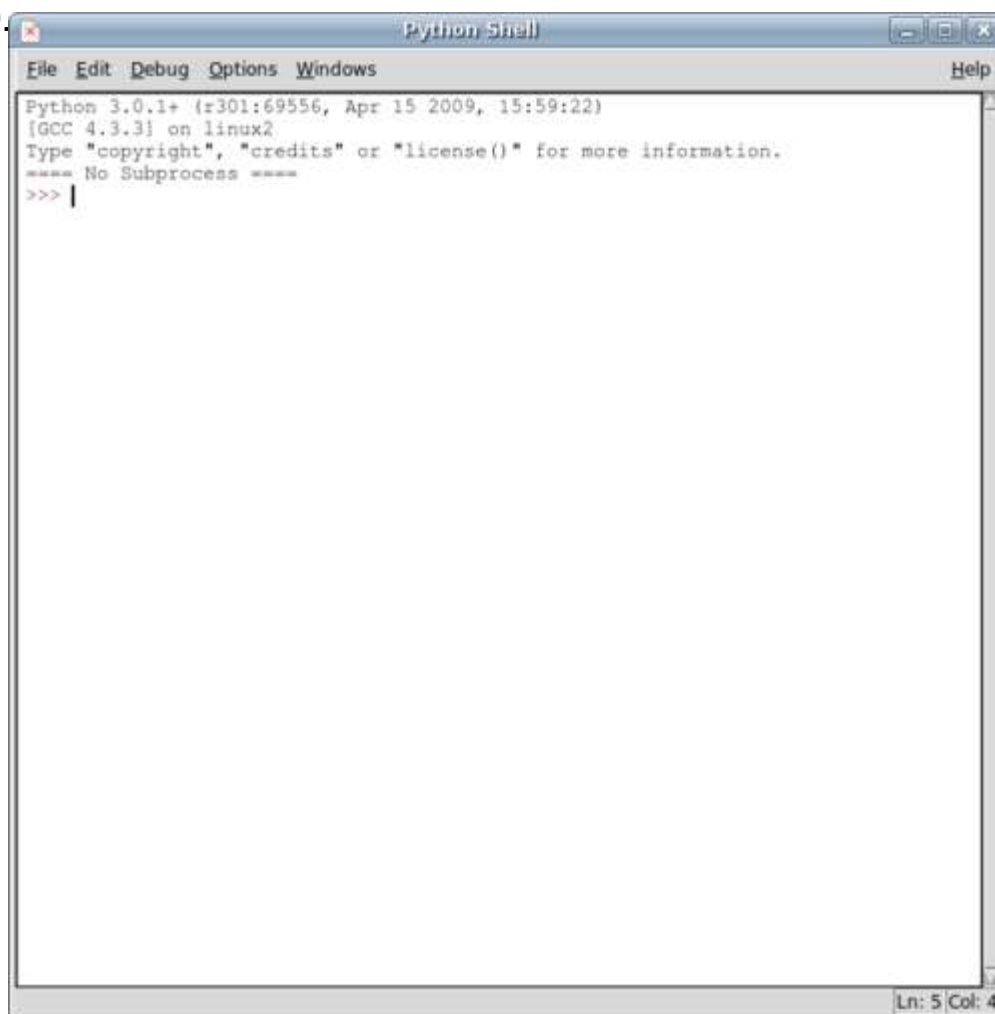


Установка/удаление: новые приложения установлены

Если всё прошло хорошо, программа управления пакетами подтвердит, что оба пакета были успешно установлены. Отсюда вы можете запустить оболочку Python, дважды кликнув по пункту «IDLE», или, нажав кнопку «Закрыть» («Close»), выйти из программы управления пакетами.

Вы всегда сможете запустить оболочку Python, из меню «Приложения» («Applications»), подменю «Программирование» («Programming»), выбрав пункт «IDLE».

10.



Графическая  
интерактивная  
оболочка  
Python для  
Linux

Оболочка  
Python — это  
то место, где  
вы проведёте  
большую часть  
времени,  
исследуя  
Python. Во всех  
примерах в  
этой книге

предполагается, что знаете, как найти оболочку Python.

---

*Перейти к [использованию оболочки Python](#).*

## Установка на другие платформы

Python 3 доступен на множестве разнообразных платформ. В частности, он доступен почти в любом дистрибутиве Linux, BSD и Solaris. Например, RedHat Linux использует программу управления пакетами yum; у FreeBSD свои порты и коллекции пакетов; у Solaris — pkgadd со своими друзьями. Поиск в вебе по словам «Python 3» + название вашей операционной системы быстро покажет, имеется ли соответствующий пакет Python 3 и как его установить.

## Использование оболочки Python

Оболочка Python — это то место, где можно исследовать синтаксис Python, получать интерактивную справку по командам и отлаживать небольшие программы. Графическая оболочка Python — IDLE — включает, кроме того,

неплохой текстовый редактор, поддерживающий подсветку синтаксиса Python. Если у вас пока нет любимого текстового редактора, стоит попробовать IDLE.

Перво-наперво, сама по себе оболочка Python — замечательная интерактивная площадка для игр с языком. На протяжении всей книги вы будете встречать примеры наподобие этого:

```
>>> 1 + 1
2
```

Первые три угловых скобки — `>>>` — обозначают приглашение оболочки Python. Его вводить не надо. Это только для того, чтобы показать вам, что этот пример должен выполняться в оболочке Python.

`1 + 1` — это, то, что вы вводите. В оболочке вы можете ввести любое корректное выражение или команду языка Python. Не стесняйтесь, она не укусит! Худшее, что может случиться, — это сообщение об ошибке. Команды выполняются сразу (как только вы нажмёте `↵ Enter`), выражения вычисляются тоже немедленно, и оболочка печатает результат.

`2` — результат вычисления этого выражения. Как ожидалось, `1 + 1` является корректным выражением на Python. Результат, конечно же, `2`.

Теперь попробуем другой пример.

```
>>> print('Hello world!')
Hello world!
```

Довольно просто, правда? Но в оболочке Python можно сделать гораздо больше разных вещей! Если вы где-нибудь застрянете, вдруг забудете команду или какие аргументы нужно передавать какой-то функции, в оболочке Python вы всегда можете вызвать интерактивную справку. Просто введите `help` и нажмите `↵ Enter`.

### Перевод сообщения оболочки

```
>>> help
```

Type `help()` for interactive help, or `help(object)` for help about object.

Введите `help()` для входа в режим интерактивной справки или `help(объект)` для получения справки о конкретном объекте.

Есть два режима встроенной справки. Можно получить справку по конкретному объекту, при этом просто выведется документация и вы вернётесь к приглашению оболочки Python. Ещё можно войти в справочный режим, в котором не вычисляются выражения Python, а вы просто вводите ключевые слова и названия команд, а в ответ выводится всё, что известно о данной команде.

Чтобы войти в интерактивный справочный режим, введите `help()` и нажмите `Enter`.

### Перевод сообщений оболочки

```
>>> help()
```

Welcome to Python 3.0! This is the online help utility.

If this is your first time using Python, you should definitely check out the tutorial on the Internet at <http://docs.python.org/tutorial/>.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

```
help>
```

Добро пожаловать в Python 3.0! Вы находитесь в режиме оперативной справки.

Если вы используете Python впервые, вам определённо следует ознакомиться с обучающим Интернет-курсом на <http://docs.python.org/tutorial/><sup>[2]</sup>.

Введите название модуля, ключевое слово или тему, чтобы получить справку по написанию программ на Python и использованию модулей. Чтобы выйти из справочного режима и вернуться в интерпретатор, просто введите quit.

Чтобы просмотреть список доступных модулей, ключевых слов и тем справки, введите modules, keywords или topics. У каждого модуля есть краткое описание его назначения; чтобы получить список модулей, в описании которых встречается определённое слово, например, слово «spam», введите modules spam.

Обратите внимание, что приглашение изменилось с `>>>` на `help>`. Это значит, что вы находитесь в режиме интерактивной справки. Здесь вы можете ввести любое ключевое слово, команду, название модуля или функции — что угодно, что может понять Python — и прочитать документацию по нему.

## Перевод сообщений оболочки

help> print

1.

Help on built-in function print  
in module builtins:

Справка по встроенной функции print из  
модуля builtins:

print(...)

print(value, ..., sep=' ',  
end='\n', file=sys.stdout)

Prints the values to a stream,  
or to sys.stdout by default.

Optional keyword arguments:  
file: a file-like object (stream);

defaults to the current  
sys.stdout.

sep: string inserted between  
values, default a space.

end: string appended after  
the last value, default a  
newline.

Печатает значения в указанный поток или в  
sys.stdout (по умолчанию).

Необязательные именованные аргументы:

- **file** — файлоподобный объект (поток), по умолчанию sys.stdout;
- **sep** — строка, вставляемая между значениями, по умолчанию пробел;
- **end** — строка, дописываемая после последнего значения, по умолчанию символ новой строки.

help> PapayaWhip

2.

no Python documentation  
found for 'PapayaWhip'

в Python не найдена документация по  
«PapayaWhip»<sup>[3]</sup>

help> quit

3.

You are now leaving help and  
returning to the Python  
interpreter.

If you want to ask for help on  
a particular object directly  
from the

interpreter, you can type  
"help(object)". Executing  
"help('string')"

has the same effect as typing  
a particular string at the help>

Вы покидаете режим справки и возвращаетесь  
в интерпретатор Python. Если вы хотите  
получить справку о некотором объекте прямо  
из интерпретатора, можете ввести  
help(объект). Выполнение help('строка')  
работает так же, как ввод этой строки в  
приглашение help>.

prompt.

>>>

4.

1. Чтобы получить документацию по функции `print()`, просто введите `print` и нажмите `↵ Enter`. Интерактивная справка покажет нечто вроде ман-страницы: имя функции, краткое описание, аргументы их значения по умолчанию и так далее. Если документация выглядит не очень понятно, не пугайтесь. В ближайших главах вы получите более полное представление обо всём этом.
2. Конечно, интерактивная справка не всё знает. Если вы введёте что-то, что не является командой Python, модулем, функцией или другим встроенным ключевым словом, интерактивная справка лишь пожмёт своими виртуальными плечами.
3. Чтобы выйти из интерактивной справки, введите `quit` и нажмите `↵ Enter`.
4. Приглашение снова стало `>>>`, чтобы показать, что вы вышли из режима интерактивной справки и вернулись в оболочку Python.

IDLE, графическая оболочка Python, включает ещё и текстовый редактор с расцветкой кода Python.

## Редакторы и IDE для Python

IDLE — не лучший вариант, когда дело доходит до написания программ на Python. Поскольку программирование полезнее начинать изучать с освоения самого языка, многие разработчики предпочитают другие текстовые редакторы и интегрированные среды разработки (Integrated Development Environment, IDE). Я не буду здесь о них подробно рассказывать, но сообщество Python имеет список поддерживающих Python редакторов, покрывающий широкий спектр платформ и лицензий.

Вы также можете взглянуть на список IDE, поддерживающих Python, правда, пока немногие из них поддерживают Python 3. Один из них — PyDev, плагин для Eclipse [1], превращающий его в полноценную среду разработки на Python. И Eclipse, и PyDev кроссплатформенные и открытые.

На коммерческом фронте есть Komodo IDE от ActiveState. Его нужно лицензировать для каждого пользователя, но студентам дают скидки, а также есть возможность бесплатно ознакомиться с продуктом в течение ограниченного периода.

Я пишу на Python девять лет, и делаю это в GNU Emacs [2], а отлаживаю в оболочке Python в командной строке. В разработке на Python нет более

правильных или менее правильных способов. Делайте то, что считаете правильным, то, что работает для вас.

## Примечания

1. Написано немного путано, но так оно и есть. По теме можно почитать `w:en:PATH (variable)`. — *Прим. пер.*
  2. Обучающий курс на английском языке. В Викиучебнике доступен его перевод на русский — Учебник Python 3.1. — *Прим. пер.*
  3. Papa ya whip (англ.) — мусс из папайи. — *Прим. пер.*
-



# Ваша первая программа на Python

Не убегайте от проблем, не осуждайте себя и не несите своё бремя в праведном безмолвии. У вас есть проблема? Прекрасно! Это пойдёт на пользу! Радуйтесь: погрузитесь в неё и исследуйте!

*Досточтимый Хенепола Гунаратана*

## Погружение

Обычно книги о программировании начинаются с кучи скучных глав о базовых вещах постепенно переходят к созданию чего-нибудь полезного. Давайте всё это пропустим. Вот вам готовая, работающая программа на Python. Возможно, вы ровным счётом ничего в ней не поймёте. Не беспокойтесь об этом, скоро мы разберём её строчка за строчкой. Но сначала прочитайте код и посмотрите, что вы сможете извлечь из него.

[[humansize.py](#)]

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    """Преобразует размер файла в удобочитаемую для человека форму.
```

Ключевые аргументы:

size -- размер файла в байтах

`a_kilobyte_is_1024_bytes` -- если True (по умолчанию), используются степени 1024

если False, используются степени 1000

Возвращает: текстовую строку (string)

```
'''
if size < 0:
    raise ValueError('число должно быть неотрицательным')

multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
for suffix in SUFFIXES[multiple]:
    size /= multiple
    if size < multiple:
        return '{0:.1f} {1}'.format(size, suffix)

raise ValueError('число слишком большое')

if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

Теперь давайте запустим эту программу из командной строки. В Windows это будет выглядеть примерно так:

```
c:\home\diveintopython3\examples> c:\python31\python.exe humansize.py
```

1.0 TB

931.3 GiB

В Mac OS X и Linux, будет почти то же самое:

```
you@localhost:~/diveintopython3/examples$ python3 humansize.py
```

1.0 TB

931.3 GiB

Что сейчас произошло? Вы выполнили свою первую программу на Python. Вы вызвали интерпретатор Python в командной строке и передали ему имя скрипта, который хотели выполнить. В скрипте определена функция `approximate_size()`, которая принимает точный размер файла в байтах и вычисляет «красивый» (но приблизительный) размер. (Возможно, вы видели это в Проводнике Windows, в Mac OS X Finder, в Nautilus, Dolphin или Thunar в Linux. Если отобразить папку с документами в виде таблицы, файловый менеджер в каждой её строке покажет иконку, название документа, размер, тип, дату последнего изменения и т. д. Если в папке есть 1093-байтовый файл с названием «TODO», файловый менеджер не покажет «TODO 1093 байта»; вместо этого он скажет что-то типа «TODO 1 КБ». Именно это и делает функция `approximate_size()`.)

Посмотрите на последние строки скрипта, вы увидите два вызова `print(approximate_size(аргументы))`. Это вызовы функций. Сначала вызывается `approximate_size()`, которой передаётся несколько аргументов, затем берётся возвращённое ею значение и передаётся прямо в функцию `print()`. Функция `print()` встроенная, вы нигде не найдёте её явного объявления. Её можно только использовать, где угодно и когда угодно. (Есть множество встроенных функций, и ещё больше функций, которые выделены в отдельные *модули*. Терпение, непоседа.)

Итак, почему при выполнении скрипта в командной строке, всегда получается один и тот же результат? Мы ещё дойдём до этого. Но сначала давайте посмотрим на функцию `approximate_size()`.

## Объявление функций

В Python есть функции, как и в большинстве других языков, но нет ни отдельных заголовочных файлов, как в C++, ни конструкций `interface/implementation`, как в Паскале. Когда вам нужна функция, просто объявите её, например, так:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

*Когда вам нужна функция, просто объявите её.*

Объявление начинается с ключевого слова `def`, затем следует имя функции, а за ним аргументы в скобках. Если аргументов несколько, они разделяются запятыми.

К тому же, стоит заметить, что в объявлении функции не задаётся тип возвращаемых данных. Функции в Python не определяют тип возвращаемых ими значений; они даже не указывают, существует ли возвращаемое значение вообще. (На самом деле, любая функция в Python возвращает значение; если в функции выполняется инструкция `return`, она возвращает указанное в этой инструкции значение, если нет — возвращает `None` — специальное нулевое значение.)



В некоторых языках программирования функции (возвращающие значение) объявляются ключевым словом `function`, а подпрограммы (не возвращающие значений) — ключевым словом `sub`. В Python же подпрограмм нет. Все функции возвращают значение (даже если оно `None`), и всегда объявляются ключевым словом `def`.

Функция `approximate_size()` принимает два аргумента: `size` и `kilobyte_is_1024_bytes`, но ни один из них не имеет типа. В Python тип переменных никогда не задаётся явно. Python вычисляет тип переменной и следит за ним самостоятельно.



В Java и других языках со статической типизацией необходимо указывать тип возвращаемого значения и каждого аргумента функции. В Python явно указывать тип, для чего-либо, не нужно. Python самостоятельно отслеживает типы переменных на основе присваиваемых им значений.

## Необязательные и именованные аргументы

В Python аргументы функций могут иметь значения по умолчанию; если функция вызывается без аргумента, то он принимает своё значение по умолчанию. К тому же, аргументы можно указывать в любом порядке, задавая их имена.

Давайте ещё раз посмотрим на объявление функции `approximate_size()`:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

Второй аргумент — `a_kilobyte_is_1024_bytes` — записывается со значением по умолчанию `True`. Это означает, что этот аргумент необязательный; можно вызвать функцию без него, а Python будет действовать так, как будто она вызвана с `True` в качестве второго параметра.

Теперь взглянем на последние строки скрипта:

```
if __name__ == '__main__':
    print(approximate_size(1000000000000, False)) ①
    print(approximate_size(1000000000000))        ②
```

- ① Функция `approximate_size()` вызывается с двумя аргументами. Внутри функции `approximate_size()` переменная `a_kilobyte_is_1024_bytes` будет `False`, поскольку `False` передаётся явно во втором аргументе.
- ② Функция `approximate_size()` вызывается только с одним аргументом. Но всё в порядке, потому что второй аргумент необязателен! Поскольку второй аргумент не указан, он принимает значение по умолчанию `True`, как определено в объявлении функции.

А ещё можно передавать значения в функцию по имени.

```
>>> from humansize import approximate_size
>>> approximate_size(4000, a_kilobyte_is_1024_bytes=False) ①
'4.0 KB'
>>> approximate_size(size=4000, a_kilobyte_is_1024_bytes=False) ②
'4.0 KB'
>>> approximate_size(a_kilobyte_is_1024_bytes=False, size=4000) ③
```

'4.0 KB'

```
>>> approximate_size(a_kilobyte_is_1024_bytes=False, 4000) ④
```

File "<stdin>", line 1

SyntaxError: non-keyword arg after keyword arg

```
>>> approximate_size(size=4000, False) ⑤
```

File "<stdin>", line 1

SyntaxError: non-keyword arg after keyword arg

*Перевод сообщений оболочки:*

Файл "<stdin>", строка 1

SyntaxError: неименованный аргумент после именованного

- ① Функция `approximate_size()` вызывается со значением 4000 в первом аргументе и `False` в аргументе по имени `a_kilobyte_is_1024_bytes`. (Он стоит на втором месте, но это не важно, как вы скоро увидите.)
- ② Функция `approximate_size()` вызывается со значением 4000 в аргументе по имени `size` и `False` в аргументе по имени `a_kilobyte_is_1024_bytes`. (Эти именованные аргументы стоят в том же порядке, в каком они перечислены в объявлении функции, но это тоже не важно.)
- ③ Функция `approximate_size()` вызывается с `False` в аргументе по имени `a_kilobyte_is_1024_bytes` и 4000 в аргументе по имени `size`. (Видите? Я же говорил, что порядок не важен.)
- ④ Этот вызов не работает, потому что за именованным аргументом следует неименованный (позиционный). Если читать список аргументов слева направо, то как только встречается именованный аргумент, все следующие за ним аргументы тоже должны быть именованными.
- ⑤ Этот вызов тоже не работает, по той же причине, что и предыдущий. Удивительно? Ведь сначала передаётся 4000 в аргументе по имени `size`, затем, «очевидно», можно ожидать, что `False` станет аргументом по имени `a_kilobyte_is_1024_bytes`. Но в Python это не работает. Раз есть именованный аргумент, все аргументы справа от него тоже должны быть именованными.

## Написание читаемого кода

Не буду растопыривать перед вами пальцы и мучить длинной лекцией о важности документирования кода. Просто знайте, что код пишется один раз, а читается многократно, и самый важный читатель вашего кода — это вы сами, через полгода после написания (т. е. всё уже забыто, и вдруг понадобилось

что-то починить). В Python писать читаемый код просто. Используйте это его преимущество и через полгода вы скажете мне «спасибо».

## Строки документации

Функции в Python можно документировать, снабжая их строками документации (англ. *documentation string*, сокращённо *docstring*). В нашей программе у функции `approximate_size()` есть строка документации:

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    """Преобразует размер файла в удобочитаемую для человека форму.

    Ключевые аргументы:
    size -- размер файла в байтах
    a_kilobyte_is_1024_bytes -- если True (по умолчанию), используются степени
    1024
                               если False, используются степени 1000

    Возвращает: текстовую строку (string)

    """
```

*Каждая функция заслуживает хорошую документацию.*

Тройные кавычки<sup>[1]</sup> используются для задания строк<sup>[2]</sup> содержащих многострочный текст. Всё, что находится между начальными и конечными кавычками, является частью одной строки данных, включая переводы строк, пробелы в начале каждой строки текста, и другие кавычки. Вы можете использовать их где угодно, но чаще всего будете встречать их в определениях строк документации.



Тройные кавычки — это ещё и простой способ определить строку, содержащую одинарные (апострофы) и двойные кавычки, подобно `qq/.../` в Perl 5.

Всё, что находится в тройных кавычках, — это строка документации функции, описывающая, что делает эта функция. Строка документации, если она есть, должна начинать тело функции, т. е. находится на следующей строчке сразу под объявлением функции. Строго говоря, вы не обязаны писать документацию к каждой своей функции, но всегда желательно это делать. Я знаю, что вам уже все уши прожужжали про документирование кода, но Python даёт вам дополнительный стимул — строки документации доступны во время выполнения как атрибут функции.



Многие IDE для Python используют строки документации для отображения контекстной справки, и когда вы набираете название функции, её документация появляется во всплывающей подсказке. Это может быть невероятно полезно, но это всего лишь строки документации, которые вы сами пишете.

## Путь поиска оператора `import`

Перед тем, как идти дальше, я хочу вкратце рассказать о путях поиска библиотек. Когда вы пытаетесь импортировать модуль (с помощью оператора `import`), Python ищет его в нескольких местах. В частности, он ищет во всех директориях, перечисленных в `sys.path`. Это просто список, который можно легко просматривать и изменять при помощи стандартных списочных методов. (Вы узнаете больше о списках в главе Встроенные типы данных.)

```
>>> import sys                                ①
>>> sys.path                                  ②
['',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
>>> sys                                       ③
<module 'sys' (built-in)>
>>> sys.path.insert(0, '/home/mark/diveintopython3/examples') ④
>>> sys.path                                  ⑤
['/home/mark/diveintopython3/examples',
 '',
 '/usr/lib/python31.zip',
 '/usr/lib/python3.1',
 '/usr/lib/python3.1/plat-linux2@EXTRAMACHDEPPATH@',
 '/usr/lib/python3.1/lib-dynload',
 '/usr/lib/python3.1/dist-packages',
 '/usr/local/lib/python3.1/dist-packages']
```

- ① Импортирование модуля `sys` делает доступными все его функции и атрибуты.
- ② `sys.path` — список имён директорий, определяющий текущий путь поиска. (Ваш будет выглядеть иначе, в зависимости от вашей операционной системы, от используемой версии Python и от того, куда он был установлен.) Python будет искать в этих директориях (в заданном порядке)



файл с расширением «.py», имя которого совпадает с тем, что вы пытаетесь импортировать.

- ③ Вообще-то я вас обманул; истинное положение дел немного сложнее, потому что не все модули лежат в файлах с расширением «.py». Некоторые из них, как, например, модуль `sys`, являются встроенными; они впаены в сам Python. Встроенные модули ведут себя точно так же, как обычные, но их исходный код недоступен, потому что они не были написаны на Python! (Модуль `sys` написан на Си.)
- ④ Можно добавить новую директорию в путь поиска, добавив имя директории в список `sys.path`, во время выполнения Python, и тогда Python будет просматривать её наравне с остальными, как только вы попытаетесь импортировать модуль. Новый путь поиска будет действителен в течение всего сеанса работы Python.
- ⑤ Выполнив команду `sys.path.insert(0, новый_путь)`, вы вставили новую директорию на первое место в список `sys.path`, и, следовательно, в начало пути поиска модулей. Почти всегда, именно это вам и нужно. В случае конфликта имён (например, если Python поставляется со 2-й версией некоторой библиотеки, а вы хотите использовать версию 3) этот приём гарантирует, что будут найдены и использованы ваши модули, а не те, которые идут в комплекте с Python.

## Всё является объектом

Если вы вдруг пропустили, я только что сказал, что функции в Python имеют атрибуты, и эти атрибуты доступны во время выполнения. Функция, как и всё остальное в Python, является объектом.

Запустите интерактивную оболочку Python и повторяйте за мной:

```
>>> import humanize ①
>>> print(humanize.approximate_size(4096, True)) ②
4.0 KiB
>>> print(humanize.approximate_size.__doc__) ③
Преобразует размер файла в удобочитаемую для человека форму.
```

Ключевые аргументы:

`size` -- размер файла в байтах

`a_kilobyte_is_1024_bytes` -- если `True` (по умолчанию), используются степени 1024

если `False`, используются степени 1000

Возвращает: текстовую строку (`string`)

- ① Первая строка импортирует программу `humansize` в качестве модуля — фрагмента кода, который можно использовать интерактивно или из другой Python-программы. После того, как модуль был импортирован, можно обращаться ко всем его публичным функциям, классам и атрибутам. Импорт применяется как в модулях, для доступа к функциональности других модулей, так и в интерактивной оболочке Python. Это очень важная идея, и вы ещё не раз встретите её на страницах этой книги.
- ② Когда вы хотите использовать функцию, определённую в импортированном модуле, нужно дописать к её имени название модуля. То есть вы не можете использовать просто `approximate_size`, обязательно `humansize.approximate_size`. Если вы использовали классы в Java, то для вас это должно быть знакомо.
- ③ Вместо того, чтобы вызвать функцию (как вы, возможно, ожидали), вы запросили один из её атрибутов — `__doc__`.



Оператор `import` в Python похож на `require` из Perl. После `import` в Python, вы обращаетесь к функциям модуля как `модуль.функция`; после `require` в Perl, для обращения к функциям модуля используется имя `модуль::функция`.

## Что такое объект?

В языке Python всё является объектом, и у любого объекта могут быть атрибуты и методы. Все функции имеют стандартный атрибут `__doc__`, содержащий строку документации, определённую в исходном коде функции. Модуль `sys` — тоже объект, имеющий (кроме прочего) атрибут под названием `path`. И так далее.

Но мы так и не получили ответ на главный вопрос: что такое объект? Разные языки программирования определяют «объект» по-разному. В одних считается, что все объекты должны иметь атрибуты и методы. В других, что объекты могут порождать подклассы. В Python определение ещё менее чёткое. Некоторые объекты не имеют ни атрибутов, ни методов, хотя и могли бы их иметь. Не все объекты порождают подклассы. Но всё является объектом в том смысле, что может быть присвоено переменной или передано функции в качестве аргумента.

Возможно, вы встречали термин «объект первого класса» в других книгах о программировании. В Python функции — *объекты первого класса*. Функцию можно передать в качестве аргумента другой функции. Модули — *объекты первого класса*. Весь модуль целиком можно передать в качестве аргумента функции. Классы — объекты первого класса, и отдельные их экземпляры — тоже объекты первого класса.

Это очень важно, поэтому я повторю это, на случай если вы пропустили первые несколько раз: *всё в Python является объектом*. Строки — это объекты. Списки — объекты. Функции — объекты. Классы — объекты. Экземпляры классов — объекты. И даже модули являются объектами.

## Отступы

Функции в Python не имеют ни явных указаний `begin` и `end`, ни фигурных скобок, которые бы показывали, где код функции начинается, а где заканчивается. Разделители — только двоеточие (`:`) и отступы самого кода.

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True): ①
    if size < 0: ②
        raise ValueError('число должно быть неотрицательным') ③
    ④
    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]: ⑤
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)

    raise ValueError('число слишком большое')
```

- ① Блоки кода определяются по их отступам. Под «блоками кода» я подразумеваю функции, блоки `if`, циклы `for` и `while` и т. д. Увеличение отступа начинает блок, а уменьшение — заканчивает. Ни скобок, ни ключевых слов. Это означает, что пробелы имеют важное значение, и их количество тоже. В этом примере код функции отбит четырьмя пробелами. Не обязательно здесь должны быть именно четыре пробела, просто их число должно быть постоянным. Первая встретившаяся строка без отступа будет означать конец функции.
- ② За оператором `if` должен следовать блок кода. Если в результате вычисления условного выражения оно окажется истинным, то выполнится блок, выделенный отступом, в противном случае произойдёт переход к блоку `else` (если он есть). Обратите внимание, что вокруг выражения скобки не стоят.
- ③ Эта строка находится внутри блока `if`. Оператор `raise` вызывает исключение (типа `ValueError`), но только если `size < 0`.
- ④ Это ещё *не* конец функции. Совсем пустые строки не считаются. Они могут повысить читаемость кода, но не могут служить разделителями блоков кода. Блок кода функции продолжается на следующей строке.

- ⑤ Оператор цикла `for` тоже начинает блок кода. Блоки кода могут содержать несколько строк, а именно — столько, сколько строк имеют такую же величину отступа. Этот цикл `for` содержит три строки кода. Других синтаксических конструкций для описания многострочных блоков кода нет. Просто делайте отступы, и будет вам счастье!

После того, как вы переборете внутренние противоречия и проведёте пару ехидных аналогий с Фортраном, вы подружитесь с отступами и начнёте видеть их преимущества. Одно из главных преимуществ — то, что все программы на Python выглядят примерно одинаково, поскольку отступы — требование языка, а не вопрос стиля. Благодаря этому становится проще читать и понимать код на Python, написанный другими людьми.



В Python используются символы возврата каретки для разделения операторов, а также двоеточие и отступы для разделения блоков кода. В C++ и Java используются точки с запятой для разделения операторов и фигурные скобки для блоков кода.

## Исключения

Исключения (англ. *exceptions* — нештатные, исключительные ситуации, требующие специальной обработки) используются повсюду в Python. Буквально каждый модуль в стандартной библиотеке Python использует их, да и сам Python вызывает их во многих ситуациях. Вы ещё неоднократно встретите их на страницах этой книги.

Что такое исключение? Обычно это ошибка, признак того, что что-то пошло не так. (Не все исключения являются ошибками, но пока это не важно.) В некоторых языках программирования принято возвращать код ошибки, который вы потом *проверяете*. В Python принято использовать исключения, которые вы *обрабатываете*.

Когда происходит ошибка в оболочке Python, она выводит кое-какие подробности об исключении и о том, как это случилось, и всё. Это называется *необработанным* исключением. Когда это исключение было вызвано, не нашлось никакого программного кода, чтобы заметить его и обработать должным образом, поэтому оно всплыло на самый верхний уровень — в оболочку Python, которая вывела немного отладочной информации и успокоилась. В оболочке это не так уж страшно, однако если это произойдёт во время работы настоящей программы, то вся программа с грохотом упадёт, если исключение не будет обработано. Может быть это то, что вам нужно, а может, и нет.



В отличие от Java, функции в Python не содержат объявлений о том, какие исключения они могут вызывать. Вам решать, какие из возможных

исключений необходимо отлавливать.

Результат исключения — это не всегда полный крах программы. Исключения можно *обработать*. Иногда исключения возникают из-за настоящих ошибок в вашем коде (например, доступ к переменной, которая не существует), но порой исключение — это нечто, что вы можете предвидеть. Если вы открываете файл, он может не существовать. Если вы импортируете модуль, он может быть не установлен. Если вы подключаетесь к базе данных, она может быть недоступна или у вас может быть недостаточно прав для доступа к ней. Если вы знаете, что какая-то строка кода может вызвать исключение, то его следует обработать с помощью блока `try...except`.



Python использует блоки `try...except` для обработки исключений и оператор `raise` для их генерации. Java и C++ используют блоки `try...catch` для обработки исключений и оператор `throw` для их генерации.

Функция `approximate_size()` вызывает исключение в двух разных случаях: если переданный ей размер (`size`) больше, чем функция может обработать, или если он меньше нуля.

```
if size < 0:
```

```
    raise ValueError('число должно быть неотрицательным')
```

Синтаксис вызова исключений достаточно прост. Надо написать оператор `raise`, за ним название исключения и опционально, поясняющую строку для отладки. Синтаксис напоминает вызов функции. (На самом деле, исключения реализованы как классы, и оператор `raise` просто создаёт экземпляр класса `ValueError` и передаёт в его метод инициализации строку 'число должно быть неотрицательным'. Но мы забегаем вперёд!)



Нет необходимости обрабатывать исключение в той функции, которая его вызвала. Если одна функция не обрабатывает его, исключение передаётся в функцию, вызвавшую эту, затем в функцию, вызвавшую вызвавшую, и т. д. «вверх по стеку». Если исключение нигде не будет обработано, то программа упадёт, а Python выведет «раскрутку стека» (англ. *traceback*) в стандартный поток ошибок — и на этом конец. Повторяю, возможно, это именно то, что вам нужно, — это зависит от того, что делает ваша программа.

## Отлов ошибок импорта

Одно из встроенных исключений Python — `ImportError` (ошибка импорта), которое вызывается, если не удастся импортировать модуль. Это может случиться по нескольким причинам, самая простая из которых — отсутствие модуля в пути поиска, оператора `import`. Что можно использовать для включения в программу опциональных возможностей. Например, библиотека `chardet`

предоставляет возможность автоматического определения кодировки символов. Предположим, ваша программа хочет использовать эту библиотеку в том случае, если она есть, или спокойно продолжить работу, если пользователь не установил её. Можно сделать это с помощью блока `try...except`.

```
try:
    import chardet
except ImportError:
    chardet = None
```

После этого можно проверять наличие модуля `chardet` простым `if`:

```
if chardet:
    # что-то сделать
else:
    # продолжить дальше
```

Другое частое применение исключения `ImportError` — выбор из двух модулей, предоставляющих одинаковый интерфейс (API), причём применение одного из них предпочтительнее другого (может, он быстрее работает или требует меньше памяти). Для этого можно попытаться импортировать сначала один модуль, и если это не удалось, то импортировать другой. К примеру, в главе XML рассказывается о двух модулях, реализующих один и тот же API, так называемый `ElementTree API`. Первый — `lxml` — сторонний модуль, который необходимо скачивать и устанавливать самостоятельно. Второй — `xml.etree.ElementTree` — медленнее, но входит в стандартную библиотеку Python 3.

```
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree
```

При выполнении этого блока `try...except` будет импортирован один из двух модулей под именем `etree`. Поскольку оба модуля реализуют один и тот же API, то в последующем коде нет необходимости проверять, какой из этих модулей был импортирован. И раз импортированный модуль в любом случае именуется как `etree`, то не придётся вставлять лишние `if` для обращения к разноимённым модулям.

## Несвязанные переменные

Взглянем ещё раз на вот эту строчку функции `approximate_size()`:

```
multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

Мы нигде не объявляли переменную `multiple` (множитель), мы только присвоили ей значение. Всё в порядке, Python позволяет так делать. Что он не



позволит сделать, так это обратиться к переменной, которой не было присвоено значение. Если попытаться так сделать, возникнет исключение `NameError` (ошибка в имени).

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>> x = 1
>>> x
1
```

*Перевод сообщения оболочки:*

```
Раскрытие стека (список последних вызовов):
  Файл "<stdin>", строка 1, <модуль>
```

```
NameError: имя 'x' не определено
```

Однажды вы скажете Python «спасибо» за это.

## Всё чувствительно к регистру

Все имена в Python чувствительны к регистру — имена переменных, функций, классов, модулей, исключений. Всё, что можно прочитать, записать, вызвать, создать или импортировать, чувствительно к регистру.

```
>>> an_integer = 1
>>> an_integer
1
>>> AN_INTEGER
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'AN_INTEGER' is not defined
>>> An_Integer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'An_Integer' is not defined
>>> an_inteGer
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'an_inteGer' is not defined
```

*Перевод сообщений оболочки:*

```
Раскрытие стека (список последних вызовов):
  Файл "<stdin>", строка 1, <модуль>
```

```
NameError: имя '<имя>' не определено
```



И так далее.

## Запуск скриптов

### *В Python всё является объектом.*

Модули Python — это объекты, имеющие несколько полезных атрибутов. И это обстоятельство можно использовать для простого тестирования модулей, при их написании, путём включения особого блока кода, который будет исполняться при запуске файла из командной строки. Взгляните на последние строки `humansize.py`:

```
if __name__ == '__main__':
    print(approximate_size(10000000000000, False))
    print(approximate_size(10000000000000))
```



Как и в C, в Python используется оператор `==` для проверки на равенство и оператор `=` для присваивания. Но в отличие от C, Python не поддерживает присваивание внутри другого выражения, поэтому у вас не получится случайно присвоить значение вместо проверки на равенство.

Итак, что же делает этот блок `if` особенным? У всех модулей, как у объектов, есть встроенный атрибут `__name__` (имя). И значение этого атрибута зависит от того, как модуль используется. Если модуль импортируется, то `__name__` принимает значение равное имени файла модуля, без расширения и пути к каталогу.

```
>>> import humansize
>>> humansize.__name__
'humansize'
```

Но модуль можно запустить и напрямую, как самостоятельную программу, в этом случае `__name__` примет особое значение — `__main__`. Python вычислит значение условного выражения в операторе `if`, определит его истинность, и выполнит блок кода `if`. В данном случае, будут напечатаны два значения.

```
c:\home\diveintopython3> c:\python31\python.exe humansize.py
1.0 TB
931.3 GiB
```

И это ваша первая программа на Python!

## Материалы для дальнейшего чтения

- PEP 257: Docstring Conventions объясняет, чем отличается хорошая строка документации от великолепной.

- Python Tutorial: Documentation Strings также касается данного вопроса.
- PEP 8: Style Guide for Python Code обсуждает хороший стиль расстановки отступов.
- *Python Reference Manual* объясняет, что означают слова «в Python всё является объектом», потому что некоторые люди — педанты, которые любят длиннющие обсуждения вещей такого рода.

## Примечания

1. В английском языке апострофы, обрамляющие текст, — это уже одинарные кавычки. — *Прим. пер.*
  2. Имеется ввиду тип данных `string` (строка). — *Прим. пер.*
-

# Встроенные ТИПЫ ДАННЫХ

В начале всяческой философии лежит удивление, изучение движет его вперёд, невежество убивает его.

*Мишель де Монтень*

## Погружение

Отложите на минутку вашу первую программу на Python и давайте поговорим о типах данных. В Python у каждого значения есть тип, но нет необходимости явно указывать типы переменных. Как это работает? Основываясь на первом присвоении значения переменной, Python определяет её тип и в дальнейшем отслеживает его самостоятельно.

В Python имеется множество встроенных типов данных. Вот наиболее важные из них:

1. **Логический**, может принимать одно из двух значений — True (истина) или False (ложь).
2. **Числа**, могут быть целыми (1 и 2), с плавающей точкой (1.1 и 1.2), дробными ( $1/2$  и  $2/3$ ), и даже комплексными.
3. **Строки** — последовательности символов Юникода, например, HTML-документ.
4. **Байты** и **массивы байтов**, например, файл изображения в формате JPEG.
5. **Списки** — упорядоченные последовательности значений.
6. **Кортежи** — упорядоченные неизменяемые последовательности значений.
7. **Множества** — неупорядоченные наборы значений.
8. **Словари** — неупорядоченные наборы пар вида ключ-значение.

Конечно, существуют и многие другие типы данных. В языке Python всё является объектом, поэтому в нём имеются также и такие типы, как *модуль*, *функция*, *класс*, *метод*, *файл*, и даже *скомпилированный код*. Некоторые из них вы уже встречали: у модулей есть имена, функции имеют строки документации, и т. д. С классами вы познакомитесь в главе «Классы и итераторы»; с файлами — в главе «Файлы».

Строки и байты несколько сложны, настолько же и важны, поэтому им отведена отдельная глава. Но сначала давайте познакомимся с остальными типами.

## Логические значения

*Практически любое выражение можно использовать в логическом контексте.*

Логический тип данных может принимать одно из двух значений: истина или ложь. В Python имеются две константы с понятными именами `True` (от англ. *true* — истина) и `False` (от англ. *false* — ложь), которые можно использовать для непосредственного присвоения логических значений. Результатом вычисления выражений также может быть логическое значение. В определенных местах (например, в операторе `if`), Python ожидает, что результатом вычисления выражения будет логическое значение. Такие места называют *логическим контекстом*. Практически любое выражение можно использовать в логическом контексте, Python в любом случае попытается определить его истинность. Для этого имеются отдельные наборы правил, для различных типов данных, указывающие на то, какие из их значений считать истинными, а какие ложными в логическом контексте. (Эта идея станет более понятна по мере ознакомления с конкретными примерами далее в этой главе.)

К примеру, рассмотрим следующий отрывок из программы `humansize.py`:

```
if size < 0:
    raise ValueError('число должно быть неотрицательным')
```

Здесь переменная `size` и значение `0` имеют тип целого числа, а знак `<` между ними является числовым оператором. Результатом же вычисления выражения `size < 0` всегда будет логическое значение. Вы можете самостоятельно в этом убедиться с помощью интерактивной оболочки Python:

```
>>> size = 1
>>> size < 0
False
>>> size = 0
>>> size < 0
False
```

```
>>> size = -1
>>> size < 0
True
```

Из-за некоторых обстоятельств, связанных с наследием оставшимся от Python 2, логические значения могут трактоваться как числа. True как 1, и False как 0.

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

*Перевод сообщения оболочки:*

Раскрутка стека (список последних вызовов) :

Файл "<stdin>", строка 1, <модуль>

ZeroDivisionError: целочисленное деление на ноль или остаток по модулю ноль

Ой-ой-ой! Не делайте так! Забудьте даже, что я упоминал об этом.

## Числа

Числа — это потрясающая штука. Их так много, всегда есть, из чего выбрать. Python поддерживает как целые числа, так и с плавающей точкой. И нет необходимости объявлять тип для их различия; Python определяет его по наличию или отсутствию десятичной точки.

```
>>> type(1)           ①
<class 'int'>
>>> isinstance(1, int) ②
True
>>> 1 + 1             ③
2
>>> 1 + 1.0           ④
2.0
>>> type(2.0)
<class 'float'>
```

① Можно использовать функцию `type()` для проверки типа любого значения

или переменной. Как и ожидалось, число 1 имеет тип `int` (целое).

- ② Функцию `isinstance()` тоже можно использовать для проверки принадлежности значения или переменной определенному типу.
- ③ Сложение двух значений типа `int` дает в результате тот же `int`.
- ④ Сложение значений типа `int` и `float` дает в результате `float`. Для выполнения операции сложения Python преобразует значение типа `int` в значение типа `float`, и в результате возвращает `float`.

## Преобразование целых чисел в десятичные дроби и наоборот

Как вы только что видели, некоторые операции (как, например, сложение), при необходимости, преобразуют целые числа в числа с плавающей точкой. Это преобразование можно выполнить и самостоятельно.

```
>>> float(2)           ①
2.0
>>> int(2.0)           ②
2
>>> int(2.5)           ③
2
>>> int(-2.5)          ④
-2
>>> 1.12345678901234567890 ⑤
1.1234567890123457
>>> type(10000000000000000) ⑥
<class 'int'>
```

- ① Можно явно преобразовать значение типа `int` в тип `float`, вызвав функцию `float()`.
- ② Так же нет ничего удивительного в том, что можно преобразовать значение типа `float` в значение типа `int`, с помощью функции `int()`.
- ③ Функция `int()` отбрасывает дробную часть числа, а не округляет его.
- ④ Функция `int()` «округляет» отрицательные числа в сторону увеличения. Она не возвращает целую часть числа, как делает функция «пол» (англ. *floor*), а просто отбрасывает дробную часть.
- ⑤ Точность чисел с плавающей точкой равна 15 десятичным знакам в дробной части.
- ⑥ Целые числа могут быть сколь угодно большими.



Python 2 имел отдельные типы целых чисел: `int` и `long`. Тип `int` был ограничен значением `sys.maxint`, которое менялось в зависимости от платформы, но обычно было равно  $2^{32}-1$ . Python 3 же имеет только один целочисленный тип, который в большинстве случаев ведёт себя как тип `long` в Python 2. См. [PEP 237](#).

## Основные операции с числами

Над числами можно выполнять различные операции.

```
>>> 11 / 2    ①
5.5
>>> 11 // 2   ②
5
>>> -11 // 2  ③
-6
>>> 11.0 // 2 ④
5.0
>>> 11 ** 2   ⑤
121
>>> 11 % 2    ⑥
1
```

- ① Оператор `/` выполняет деление чисел с плавающей точкой. Он возвращает значение типа `float`, даже если и делимое, и делитель, имеют тип `int`.
- ② Оператор `//` выполняет целочисленное деление необычного вида. Когда результат положительный, можете считать, что он просто отбрасывает (не округляет) дробную часть, но будьте осторожны с этим.
- ③ Когда выполняется целочисленное деление отрицательных чисел, оператор `//` округляет результат до ближайшего целого в «большую» сторону. С математической точки зрения, это конечно же округление в меньшую сторону, т. к.  $-6$  меньше чем  $-5$ ; но это может сбить вас с толку, и вы будете ожидать, что результат будет «округлён» до  $-5$ .
- ④ Оператор `//` не всегда возвращает целое число. Если хотя бы один из операндов — делимое или делитель — будет типа `float`, то хотя результат и будет округлён до ближайшего целого, в действительности он также будет иметь тип `float`.
- ⑤ Оператор `**` выполняет возведение в степень.  $11^2$  — это 121.
- ⑥ Оператор `%` возвращает остаток от целочисленного деления. 11, делённое на 2, даёт 5 и 1 в остатке, поэтому здесь результат — 1.





В Python 2, оператор / обычно означает целочисленное деление, но добавив в код специальную директиву можно заставить его выполнять деление с плавающей точкой. В Python 3, оператор / всегда означает деление с плавающей точкой. См. [PEP 238](#).

## Дроби

Python не ограничен целыми числами и числами с плавающей точкой. Он также может выполнять все те забавные вещи, которые вы изучали в школе на уроках математики и затем благополучно забыли.

```
>>> import fractions ①
>>> x = fractions.Fraction(1, 3) ②
>>> x
Fraction(1, 3)
>>> x * 2 ③
Fraction(2, 3)
>>> fractions.Fraction(6, 4) ④
Fraction(3, 2)
>>> fractions.Fraction(0, 0) ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fractions.py", line 96, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(0, 0)
```

*Перевод сообщения оболочки:*

Раскрытие стека (список последних вызовов):

Файл "<stdin>", строка 1, в <модуль>

Файл "fractions.py", строка 96, в \_\_new\_\_

raise ZeroDivisionError('Дробь(%s, 0)' % числитель)

ZeroDivisionError: Дробь(0, 0)

- ① Перед началом использования дробей, импортируйте модуль fractions.
- ② Чтобы определить дробь, создайте объект класса Fraction и передайте ему числитель и знаменатель.
- ③ С дробями можно выполнять все обычные математические операции. Все они возвращают новый объект класса Fraction.  $2 \cdot \frac{1}{3} = \frac{2}{3}$

- ④ Объект Fraction автоматически сократит дроби.  $\frac{6}{4} = \frac{3}{2}$
- ⑤ У Python хватает здравого смысла, чтобы не создавать дроби с нулевым знаменателем.

## Тригонометрия

Ещё в Python можно работать с основными тригонометрическими функциями.

```
>>> import math
>>> math.pi ①
3.1415926535897931
>>> math.sin(math.pi / 2) ②
1.0
>>> math.tan(math.pi / 4) ③
0.9999999999999999
```

- ① Модуль math содержит константу  $\pi$  — отношение длины окружности к её диаметру.
- ② Модуль math содержит все основные тригонометрические функции, включая `sin()`, `cos()`, `tan()`, и их варианты наподобие `asin()`.
- ③ Заметьте, однако, что точность расчетов в Python не бесконечна. Выражение  $\tan\left(\frac{\pi}{4}\right)$  должно возвращать значение 1.0, а не 0.9999999999999999.

## Числа в логическом контексте

*Нулевые значения — ложь, ненулевые значения — истина.*

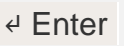
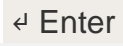
Вы можете использовать числа в логическом контексте, например, в операторе `if`. Нулевые значения — ложь, ненулевые значения — истина.

```
>>> def is_it_true(anything): ①
...     if anything:
...         print("да, это истина")
...     else:
...         print("нет, это ложь")
...
>>> is_it_true(1) ②
да, это истина
>>> is_it_true(-1)
```

```

да, это истина
>>> is_it_true(0)
нет, это ложь
>>> is_it_true(0.1) ③
да, это истина
>>> is_it_true(0.0)
нет, это ложь
>>> import fractions
>>> is_it_true(fractions.Fraction(1, 2)) ④
да, это истина
>>> is_it_true(fractions.Fraction(0, 1))
нет, это ложь

```

- ① Вы знали, что можно определять свои собственные функции в интерактивной оболочке Python? Просто нажимайте клавишу  в конце каждой строки, а чтобы закончить ввод нажмите клавишу  на пустой строке.
- ② В логическом контексте, ненулевые целые числа — истина; значение 0 — ложь.
- ③ Ненулевые числа с плавающей точкой — истина; значение 0.0 — ложь. Будьте осторожны с этим! Если имеется малейшая ошибка округления (как вы могли видеть в предыдущем разделе, это вполне возможно), то Python будет проверять значение 0.000000000000001 вместо 0.0 и соответственно вернёт логическое значение True.
- ④ Дроби тоже могут быть использованы в логическом контексте. Fraction(0, n) — ложь для всех значений n. Все остальные дроби — истина.

## Списки

Списки — рабочая лошадка Python. Когда я говорю «список», вы, наверное, думаете: «это массив, чей размер я должен задать заранее и который может хранить элементы только одного типа» и т. п., но это не так. Списки намного интереснее.



Списки в Python похожи на массивы в Perl 5. Там переменные, содержащие массивы, всегда начинаются с символа @; в Python переменные могут называться как угодно, язык следит за типом самостоятельно.



В Python список — это нечто большее, чем массив в Java (хотя список можно использовать и как массив, если это действительно то, чего вы хотите от жизни). Точнее будет аналогия с Java-классом `ArrayList`, который может хранить произвольные объекты и динамически расширяться по мере добавления новых элементов.

## Создание списка

Создать список легко: впишите все значения, через запятую, в квадратных скобках.

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'example'] ①
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[0] ②
'a'
>>> a_list[4] ③
'example'
>>> a_list[-1] ④
'example'
>>> a_list[-3] ⑤
'mpilgrim'
```

- ① Сначала вы определили список из пяти элементов. Обратите внимание, они сохраняют свой первоначальный порядок. Это не случайно. Список — это упорядоченный набор элементов.
- ② Список можно использовать как массив с нумерацией от нуля. Первый элемент не пустого списка будет всегда `a_list[0]`.
- ③ Последним элементом этого пятиэлементного списка будет `a_list[4]`, потому что нумерация элементов в списке всегда начинается с нуля.
- ④ Используя отрицательный индекс, можно обратиться к элементам по их номеру от конца списка. Последний элемент не пустого списка будет всегда `a_list[-1]`.
- ⑤ Если вас сбивают с толку отрицательные индексы, то просто думайте о них следующим образом: `a_list[-n] == a_list[len(a_list) - n]`. В нашем примере `a_list[-3] == a_list[5 - 3] == a_list[2]`.

## Разрезание списка

*`a_list[0]` — первый элемент списка `a_list`.*

После того, как список создан, можно получить любую его часть в виде списка. Это называется «*slicing*» — *срез списка*.

```
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[1:3]      ①
['b', 'mpilgrim']
>>> a_list[1:-1]     ②
['b', 'mpilgrim', 'z']
>>> a_list[0:3]      ③
['a', 'b', 'mpilgrim']
>>> a_list[:3]       ④
['a', 'b', 'mpilgrim']
>>> a_list[3:]       ⑤
['z', 'example']
>>> a_list[:]        ⑥
['a', 'b', 'mpilgrim', 'z', 'example']
```

- ① Вы можете получить часть списка, называемую «срезом», указав два индекса. В результате получается новый список, включающий в себя элементы исходного в том же порядке, начиная с первого индекса среза (в данном случае `a_list[1]`), до последнего, но не включая его (в данном случае `a_list[3]`).
- ② Срез работает, даже если один или оба индекса отрицательны. Если вам это поможет, можете думать об этом так: список читается слева направо, первый индекс среза определяет первый нужный вам элемент, а второй индекс определяет первый элемент, который вам не нужен. Возвращаемое значение всегда находится между ними.
- ③ Нумерация списков начинается с нуля, поэтому `a_list[0:3]` возвращает первые три элемента списка, начиная с `a_list[0]`, заканчивая на (но не включая) `a_list[3]`.
- ④ Если левый индекс среза — 0, вы можете опустить его, 0 будет подразумеваться. Так, `a_list[:3]` — это то же самое, что и `a_list[0:3]`, потому что начальный 0 подразумевается.
- ⑤ Аналогично, если правый индекс среза является длиной списка, вы можете его опустить. Так, `a_list[3:]` — это то же самое, что и `a_list[3:5]`, потому что этот список содержит пять элементов. Здесь прослеживается явная симметрия. В этом пятиэлементном списке `a_list[:3]` возвращает первые 3 элемента, а `a_list[3:]` возвращает последние два элемента. На самом деле, `a_list[:n]` всегда будет возвращать первые `n` элементов, а `a_list[n:]` будет возвращать все остальные, независимо от длины списка.

- ⑥ Если оба индекса списка опущены, включаются все элементы списка. Но это не то же самое, что первоначальная переменная `a_list`. Это новый список, включающий все элементы исходного. Запись `a_list[:]` представляет собой простейший способ получения полной копии списка.


## Добавление элементов в список

Существует четыре способа добавить элементы в список.

```
>>> a_list = ['a']
>>> a_list = a_list + [2.0, 3] ①
>>> a_list ②
['a', 2.0, 3]
>>> a_list.append(True) ③
>>> a_list
['a', 2.0, 3, True]
>>> a_list.extend(['four', 'Ω']) ④
>>> a_list
['a', 2.0, 3, True, 'four', 'Ω']
>>> a_list.insert(0, 'Ω') ⑤
>>> a_list
['Ω', 'a', 2.0, 3, True, 'four', 'Ω']
```

- ① Оператор `+` соединяет списки, создавая новый список. Список может содержать любое число элементов; ограничений размера не существует (пока есть доступная память). Однако, если вы заботитесь о памяти, знайте, что сложение списков создает ещё один список в памяти. В данном случае, этот новый список немедленно присваивается существующей переменной `a_list`. Так что эта строка кода, на самом деле, реализует двухэтапный процесс — сложение, а затем присвоение — который может (временно) потребовать много памяти, если вы имеете дело с большими списками.
- ② Список может содержать элементы любых типов, и элементы одного списка не обязательно должны быть одного и того же типа. Здесь мы видим список, содержащий строку, число с плавающей точкой и целое число.
- ③ Метод `append()` добавляет один элемент в конец списка. (Теперь у нас в списке присутствуют *четыре* различных типа данных!)
- ④ Списки реализованы как классы. «Создание» списка — это фактически создание экземпляра класса. Т. о., список имеет методы, которые работают с ним. Метод `extend()` принимает один аргумент — список, и добавляет каждый его элемент к исходному списку.

- ⑤ Метод `insert()` вставляет элемент в список. Первым аргументом является индекс первого элемента в списке, который будет сдвинут новым элементом со своей позиции. Элементы списка не обязаны быть уникальными; например, теперь у нас есть два различных элемента со значением 'Ω': первый элемент `a_list[0]` и последний элемент `a_list[6]`.

 В Python конструкция `a_list.insert(0, value)` действует как функция `unshift()` в Perl. Она добавляет элемент в начало списка, а все другие элементы увеличивают свой индекс на единицу, чтобы освободить пространство.

Давайте подробнее рассмотрим разницу между `append()` и `extend()`.

```
>>> a_list = ['a', 'b', 'c']
>>> a_list.extend(['d', 'e', 'f']) ①
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(a_list) ②
6
>>> a_list[-1]
'f'
>>> a_list.append(['g', 'h', 'i']) ③
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]
>>> len(a_list) ④
7
>>> a_list[-1]
['g', 'h', 'i']
```

- ① Метод `extend()` принимает один аргумент, который всегда является списком, и добавляет каждый элемент этого списка к `a_list`.
- ② Если вы возьмёте список из трёх элементов и расширите его списком из ещё трёх элементов, в итоге получится список из шести элементов.
- ③ С другой стороны, метод `append()` получает единственный аргумент, который может быть любого типа. Здесь мы вызываем метод `append()`, передавая ему список из трёх элементов.
- ④ Если вы возьмёте список из шести элементов и добавите к нему список, в итоге вы получите... список из семи элементов. Почему семь? Потому что последний элемент (который мы только что добавили) является списком. Списки могут содержать любые типы данных, включая другие списки. Возможно, это то, что вам нужно, возможно, нет. Но это то, что вы просили, и это то, что вы получили.



## Поиск значений в списке

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list.count('new')    ①
2
>>> 'new' in a_list        ②
True
>>> 'c' in a_list
False
>>> a_list.index('mpilgrim') ③
3
>>> a_list.index('new')      ④
2
>>> a_list.index('c')        ⑤
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
```

*Перевод сообщения оболочки:*

```
Раскрытие стека (от внешних к внутренним):
  Файл "<интерактивный ввод>", строка 1, позиция ?
ValueError: list.index(x): x не в списке
```

- ① Как вы наверное ожидаете, метод `count` возвращает количество вхождений указанного значения в список.
- ② Если всё, что вам нужно — это узнать, присутствует ли значение в списке или нет, тогда оператор `in` намного быстрее, чем метод `count()`. Оператор `in` всегда возвращает `True` или `False`; он не сообщает, сколько именно в списке данных значений.
- ③ Если вам необходимо точно знать, на каком месте в списке находится какое-либо значение, то используйте метод `index()`. По умолчанию, он просматривает весь список, но вы можете указать вторым аргументом индекс (отсчитываемый от нуля), с которого необходимо начать поиск, и даже третий аргумент — индекс, на котором необходимо остановить поиск.
- ④ Метод `index()` находит только первое вхождение значения в списке. В данном случае `'new'` встречается дважды в списке: в `a_list[2]` и `a_list[4]`, но метод `index()` вернёт только индекс первого вхождения.
- ⑤ Вопреки вашим ожиданиям, если значение не найдено в списке, то метод `index()` возбудит исключение.[\[wap-robin.com\]](http://wap-robin.com)

Постойте, что? Да, верно: метод `index()` возбуждает исключение, если не может найти значение в списке. Вы наверное заметили отличие от

большинства других языков, которые возвращают какой-нибудь неверный индекс (например, -1). Если на первый взгляд это может немного раздражать, то, я думаю, в будущем вы примете этот подход. Это означает, что ваша программа будет аварийно завершена в том месте, где возникла проблема, вместо того чтобы тихо перестать работать в каком-нибудь другом месте. Запомните, -1 тоже является подходящим индексом для списков. Если бы метод `index()` возвращал -1, вас могли ожидать невесёлые вечера, потраченные на поиск ошибок в коде!

## Удаление элементов из списка

### *Списки никогда не содержат разрывов.*

Списки могут увеличиваться и сокращаться автоматически. Вы уже видели как они могут увеличиваться. Также существует несколько разных способов удалить элементы из списка.

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list[1]
'b'
>>> del a_list[1]      ①
>>> a_list
['a', 'new', 'mpilgrim', 'new']
>>> a_list[1]          ②
'new'
```

- ① Можно использовать выражение `del` для удаления определенного элемента из списка.
- ② Если после удаления элемента с индексом 1 опять попытаться прочитать значение списка с индексом 1, это не вызовет ошибки. Все элементы после удаления сдвигают свои индексы, чтобы «заполнить пробел», возникший после удаления элемента.

Не знаете индекс? Не беда — можно удалить элемент по значению.

```
>>> a_list.remove('new') ①
>>> a_list
['a', 'mpilgrim', 'new']
>>> a_list.remove('new') ②
>>> a_list
['a', 'mpilgrim']
>>> a_list.remove('new')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

*Перевод сообщений оболочки:*

Раскрутка стека (список последних вызовов):

Файл "<stdin>", строка 1, <модуль>

ValueError: list.remove(x): x не в списке

- ① Можно удалить элемент из списка при помощи метода `remove()`. Метод `remove()` в качестве параметра принимает значение и удаляет первое вхождение этого значения из списка. Кроме того, индексы всех элементов, следующих за удалённым, будут сдвинуты, чтобы «заполнить пробел». Списки никогда не содержат разрывов.
- ② Можно вызывать метод `remove()` столько, сколько хотите, однако если попытаться удалить значение, которого нет в списке, будет порождено исключение.

### Удаление элементов из списка: дополнительный раунд

Другой интересный метод списков — `pop()`. Метод `pop()` — это еще один способ удалить элементы из списка, но с одной особенностью.

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim']
```

```
>>> a_list.pop() ①
```

```
'mpilgrim'
```

```
>>> a_list
```

```
['a', 'b', 'new']
```

```
>>> a_list.pop(1) ②
```

```
'b'
```

```
>>> a_list
```

```
['a', 'new']
```

```
>>> a_list.pop()
```

```
'new'
```

```
>>> a_list.pop()
```

```
'a'
```

```
>>> a_list.pop() ③
```

```
Traceback (most recent call last):
```

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

```
IndexError: pop from empty list
```

*Перевод сообщения оболочки:*

Раскрутка стека (список последних вызовов):


Файл "<stdin>", строка 1, <модуль>

IndexError: pop из пустого списка

- ① Если вызвать `pop()` без аргументов, он удалит последний элемент списка и вернет удаленное значение.

② С помощью метода `pop()` можно удалить любой элемент списка. Просто вызовите метод с индексом элемента. Этот элемент будет удалён, а все элементы после него сместятся, чтобы «заполнить пробел». Метод возвращает удалённое из списка значение.

③ Метод `pop()` для пустого списка возбуждает исключение.

 Вызов метода `pop()` без аргументов эквивалентен вызову функции `pop()` в Perl. Он удаляет последний элемент из списка и возвращает удалённое значение. В языке программирования Perl есть также функция `shift()`, которая удаляет первый элемент и возвращает его значение. В Python это эквивалентно `a_list.pop(0)`.

## Списки в логическом контексте

*Пустые списки — ложь, все остальные — истина.*

Вы также можете использовать список в логическом контексте, например, в операторе `if`:

```
>>> def is_it_true(anything):
...     if anything:
...         print("да, это истина")
...     else:
...         print("нет, это ложь")
...
>>> is_it_true([])           ①
нет, это ложь
>>> is_it_true(['a'])       ②
да, это истина
>>> is_it_true([False])     ③
да, это истина
```

① В логическом контексте пустой список — ложь.

② Любой список, состоящий хотя бы из одного элемента, — истина.

③ Любой список, состоящий хотя бы из одного элемента, — истина. Значения элементов не важны.

## Кортежи

Кортеж — это неизменяемый список. Кортеж не может быть изменён никаким способом после его создания.

```
>>> a_tuple = ("a", "b", "mpilgrim", "z", "example") ①
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple[0] ②
'a'
>>> a_tuple[-1] ③
'example'
>>> a_tuple[1:3] ④
('b', 'mpilgrim')
```

- ① Кортеж определяется так же, как список, за исключением того, что набор элементов заключается в круглые скобки, а не в квадратные.
- ② Элементы кортежа заданы в определённом порядке, как и в списке. Элементы кортежа индексируются с нуля, как и элементы списка, таким образом первый элемент не пустого кортежа — это всегда `a_tuple[0]`.
- ③ Отрицательные значения индекса отсчитываются от конца кортежа, как и в списке.
- ④ Создание среза кортежа («slicing») аналогично созданию среза списка. Когда создаётся срез списка, получается новый список; когда создаётся срез кортежа, получается новый кортеж.

Основное отличие между кортежами и списками состоит в том, что кортежи не могут быть изменены. Говоря техническим языком, кортеж — неизменяемый объект. На практике это означает, что у них нет методов, которые бы позволили их изменить. У списков есть такие методы, как `append()`, `extend()`, `insert()`, `remove()`, и `pop()`. У кортежей ни одного из этих методов нет. Можно взять срез от кортежа (так как при этом создастся новый кортеж), можно проверить, содержит ли кортеж элемент с конкретным значением (так как это действие не изменит кортеж), и... собственно, всё.

# продолжение предыдущего примера

```
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple.append("new") ①
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> a_tuple.remove("z") ②
Traceback (innermost last):
```

```
File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> a_tuple.index("example")      ③
4
>>> "z" in a_tuple                 ④
True
```

### Перевод сообщений оболочки:

Раскрытие стека (от внешних к внутренним):

Файл "<интерактивный ввод>", строка 1, позиция ?

AttributeError: у объекта 'tuple' нет атрибута '<атрибут>'

- ① Вы не можете добавить элементы к кортежу. Кортежи не имеют методов `append()` или `extend()`.
- ② Вы не можете удалять элементы из кортежа. Кортежи не имеют методов `remove()` или `pop()`.
- ③ Вы можете искать элементы в кортеже, поскольку это не изменяет кортеж.
- ④ Вы также можете использовать оператор `in`, чтобы проверить существует ли элемент в кортеже.

Так где же могут пригодиться кортежи?

- Кортежи быстрее, чем списки. Если вы определяете неизменяемый набор значений и всё, что вы собираетесь с ним делать — итерировать по нему, используйте кортеж вместо списка.
- Кортежи делают код безопаснее в том случае, если у вас есть «защищенные от записи» данные, которые не должны изменяться. Использование кортежей вместо списков избавит вас от необходимости использовать оператор `assert`, дающий понять, что данные неизменяемы, и что нужно приложить особые усилия (и особую функцию), чтобы это обойти.
- Некоторые кортежи могут использоваться в качестве ключей словаря (конкретно, кортежи, содержащие *неизменяемые* значения, например, строки, числа и другие кортежи). Списки никогда не могут использоваться в качестве ключей словаря, потому что списки — изменяемые объекты.



Кортежи могут быть преобразованы в списки и наоборот. Встроенная функция `tuple()` принимает список и возвращает кортеж из всех его элементов, функция `list()` принимает кортеж и возвращает список. По сути дела, `tuple()` замораживает список, а `list()` размораживает кортеж.

## Кортежи в логическом контексте

Вы можете использовать кортежи в логическом контексте, например, в операторе if.

```
>>> def is_it_true(anything):
...     if anything:
...         print("да, это истина")
...     else:
...         print("нет, это ложь")
...
>>> is_it_true(())           ①
нет, это ложь
>>> is_it_true(('a', 'b'))  ②
да, это истина
>>> is_it_true((False,))    ③
да, это истина
>>> type((False))           ④
<class 'bool'>
>>> type((False,))
<class 'tuple'>
```

- ① В логическом контексте пустой кортеж является ложью.
- ② Любой кортеж состоящий по крайней мере из одного элемента — истина.
- ③ Любой кортеж состоящий по крайней мере из одного элемента — истина. Значения элементов не важны. Но что делает здесь эта запятая?
- ④ Чтобы создать кортеж из одного элемента, необходимо после него поставить запятую. Без запятой Python предполагает, что вы просто добавили еще одну пару скобок, что не делает ничего плохого, но и не создает кортеж.

## Присваивание нескольких значений за раз

Вот крутой программмерский прием: в Python можно использовать кортежи, чтобы присваивать значение нескольким переменным сразу.

```
>>> v = ('a', 2, True)
>>> (x, y, z) = v           ①
>>> x
'a'
>>> y
2
>>> z
True
```



- ①  $v$  — это кортеж из трех элементов, а  $(x, y, z)$  — кортеж из трёх переменных. Присвоение одного другому приводит к присвоению каждого значения из  $v$  каждой переменной в указанном порядке.

Это не единственный способ использования. Предположим, что вы хотите присвоить имена диапазону значений. Вы можете использовать встроенную функцию `range()` для быстрого присвоения сразу нескольких последовательных значений.

```
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
SUNDAY) = range(7) ①
>>> MONDAY ②
0
>>> TUESDAY
1
>>> SUNDAY
6
```

- ① Встроенная функция `range()` создаёт последовательность целых чисел. (Строго говоря, функция `range()` возвращает итератор, а не список или кортеж, но вы узнаете разницу чуть позже.) `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, и `SUNDAY` — определяемые переменные. (Этот пример заимствован из модуля `calendar`, небольшого забавного модуля, который отображает календари, примерно как программа `cal` из UNIX. В этом модуле определяются константы целого типа для дней недели.)
- ② Теперь каждой переменной присвоено конкретное значение: `MONDAY` — это 0, `TUESDAY` — 1, и так далее.

Вы также можете использовать присвоение значений нескольким переменным сразу, чтобы создавать функции, возвращающие несколько значений, для этого достаточно просто вернуть кортеж, содержащий эти значения. В том месте программы, где была вызвана функция, возвращаемое значение можно использовать как кортеж целиком, или присвоить значения нескольких отдельных переменных. Этот приём используется во многих стандартных библиотеках Python, включая и модуль `os`, о котором вы узнаете в следующей главе.

## Множества

Множество — это «мешок», содержащий неупорядоченные уникальные значения. Одно множество может содержать значения любых типов. Если у вас есть два множества, вы можете совершать над ними любые стандартные операции, например, объединение, пересечение и разность.

## Создание множества

Начнём с самого начала. Создать множество очень легко.

```
>>> a_set = {1} ①
>>> a_set
{1}
>>> type(a_set) ②
<class 'set'>
>>> a_set = {1, 2} ③
>>> a_set
{1, 2}
```

- ① Чтобы создать множество с одним значением, поместите его в фигурные скобки (`{}`).
- ② Множества, вообще-то, реализуются как классы, но пока не беспокойтесь об этом.
- ③ Чтобы создать множество с несколькими значениями, отделите их друг от друга запятыми и поместите внутрь фигурных скобок.

Также вы можете создать множество из списка.

```
>>> a_list = ['a', 'b', 'mpilgrim', True, False, 42]
>>> a_set = set(a_list) ①
>>> a_set ②
{'a', False, 'b', True, 'mpilgrim', 42}
>>> a_list ③
['a', 'b', 'mpilgrim', True, False, 42]
```

- ① Чтобы создать множество из списка, воспользуйтесь функцией `set()`. (Педанты, которые знают как реализованы множества, отметят, что на самом деле это создание экземпляра класса, а не вызов функции. Я *обещаю*, вы узнаете в чём разница далее в этой книге. Сейчас просто знайте, что `set()` ведет себя как функция и возвращает множество.)
- ② Как я упоминал ранее, множество может содержать значения любых типов. И, как я упоминал ранее, множества *неупорядочены*. Это множество не помнит первоначальный порядок списка, из которого оно было создано. Если вы добавляете элементы в множество, оно не запоминает, в каком порядке они добавлялись.
- ③ Исходный список не изменился.

У вас ещё нет значений? Нет проблем. Можно создать пустое множество.

```
>>> a_set = set() ①
>>> a_set ②
```

```

set()
>>> type(a_set)    ③
<class 'set'>
>>> len(a_set)     ④
0
>>> not_sure = {}  ⑤
>>> type(not_sure)
<class 'dict'>

```

- ① Чтобы создать пустое множество, вызовите `set()` без аргументов.
- ② Напечатанное представление пустого множества выглядит немного странно. Вы, наверное, ожидали увидеть `{}`? Это означало бы пустой словарь, а не пустое множество. О словарях вы узнаете далее в этой главе.
- ③ Несмотря на странное печатное представление, это *действительно* множество...
- ④ ...и это множество не содержит ни одного элемента.
- ⑤ В силу исторических причуд, пришедших из Python 2, нельзя создать пустое множество с помощью двух фигурных скобок. На самом деле, они создают пустой словарь, а не множество.

## Изменение множества

Есть два способа добавить элементы в существующее множество: метод `add()` и метод `update()`.

```

>>> a_set = {1, 2}
>>> a_set.add(4)  ①
>>> a_set
{1, 2, 4}
>>> len(a_set)    ②
3
>>> a_set.add(1)  ③
>>> a_set
{1, 2, 4}
>>> len(a_set)    ④
3

```

- ① Метод `add()` принимает один аргумент, который может быть любого типа, и добавляет данное значение в множество.
- ② Теперь множество содержит 3 элемента.

③ Множества — мешки *уникальных значений*. Если попытаться добавить значение, которое уже присутствует в множестве, ничего не произойдет. Это не приведет к возникновению ошибки; просто нулевое действие.

④ Это множество *все ещё* состоит из 3 элементов.

```
>>> a_set = {1, 2, 3}
>>> a_set
{1, 2, 3}
>>> a_set.update({2, 4, 6})
>>> a_set
{1, 2, 3, 4, 6}
>>> a_set.update({3, 6, 9}, {1, 2, 3, 5, 8, 13})
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 13}
>>> a_set.update([10, 20, 30])
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 20, 30}
```

① Метод `update()` принимает один аргумент — множество, и добавляет все его элементы к исходному множеству. Так, как если бы вы вызывали метод `add()` и по очереди передавали ему все элементы множества.

② Повторяющиеся значения игнорируются, поскольку множество не может содержать дубликаты.

③ Вообще-то, вы можете вызвать метод `update()` с любым количеством параметров. Когда он вызывается с двумя множествами, метод `update()` добавляет все элементы обоих множеств в исходное множество (пропуская повторяющиеся).

④ Метод `update()` может принимать объекты различных типов, включая списки. Когда ему передается список, он добавляет все его элементы в исходное множество.

## Удаление элементов множества

Существуют три способа удаления отдельных значений из множества. Первые два, `discard()` и `remove()`, немного различаются.

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set
{1, 3, 36, 6, 10, 45, 15, 21, 28}
>>> a_set.discard(10)
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.discard(10)
```

```

>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.remove(21)           ③
>>> a_set
{1, 3, 36, 6, 45, 15, 28}
>>> a_set.remove(21)           ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 21

```

*Перевод сообщения оболочки:*

```

Раскрытие стека (список последних вызовов):
  Файл "<stdin>", строка 1, <модуль>

KeyError: 21

```

- ① Метод `discard()` принимает в качестве аргумента одиночное значение и удаляет это значение из множества.
- ② Если вы вызвали метод `discard()` передав ему значение, которого нет в множестве, ничего не произойдет, просто нулевое действие.
- ③ Метод `remove()` также принимает в качестве аргумента одиночное значение, и также удаляет его из множества.
- ④ Вот в чём отличие: если значения нет в множестве, метод `remove()` породит исключение `KeyError`.

Подобно спискам, множества имеют метод `pop()`.

```

>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set.pop()                 ①
1
>>> a_set.pop()
3
>>> a_set.pop()
36
>>> a_set
{6, 10, 45, 15, 21, 28}
>>> a_set.clear()               ②
>>> a_set
set()
>>> a_set.pop()                 ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'

```

*Перевод сообщения оболочки:*

Раскрутка стека (список последних вызовов) :

Файл "<stdin>", строка 1, <модуль>

KeyError: 'pop из пустого множества'

- ① Метод `pop()` удаляет один элемент из множества и возвращает его значение. Однако, поскольку множества неупорядочены, это не «последний» элемент в множестве, поэтому невозможно проконтролировать какое значение было удалено. Удаляется произвольный элемент.
- ② Метод `clear()` удаляет все элементы множества, оставляя вас с пустым множеством. Это эквивалентно записи `a_set = set()`, которая создаст новое пустое множество и перезапишет предыдущее значение переменной `a_set`.
- ③ Попытка извлечения (`pop`) элемента из пустого множества породит исключение `KeyError`.

## Основные операции с множествами

Тип `set` в Python поддерживает несколько основных операций над множествами.

```
>>> a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
>>> 30 in a_set                                ①
True
>>> 31 in a_set
False
>>> b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}
>>> a_set.union(b_set)                          ②
{1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}
>>> a_set.intersection(b_set)                  ③
{9, 2, 12, 5, 21}
>>> a_set.difference(b_set)                     ④
{195, 4, 76, 51, 30, 127}
>>> a_set.symmetric_difference(b_set)           ⑤
{1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}
```

- ① Чтобы проверить, принадлежит ли значение множеству, используйте оператор `in`. Он работает так же, как и для списков.
- ② Метод `union()` (объединение) возвращает новое множество, содержащее все элементы каждого из множеств.
- ③ Метод `intersection()` (пересечение) возвращает новое множество, содержащее все элементы, которые есть и в первом множестве, и во втором.

- ④ Метод `difference()` (разность) возвращает новое множество, содержащее все элементы, которые есть в множестве `a_set`, но которых нет в множестве `b_set`.
- ⑤ Метод `symmetric_difference()` (симметрическая разность) возвращает новое множество, которое содержит только уникальные элементы обоих множеств.

Три из этих методов симметричны.

# продолжение предыдущего примера

```
>>> b_set.symmetric_difference(a_set) ①
{3, 1, 195, 4, 6, 8, 76, 15, 17, 18, 51, 30, 127}
>>> b_set.symmetric_difference(a_set) == a_set.symmetric_difference(b_set) ②
True
>>> b_set.union(a_set) == a_set.union(b_set) ③
True
>>> b_set.intersection(a_set) == a_set.intersection(b_set) ④
True
>>> b_set.difference(a_set) == a_set.difference(b_set) ⑤
False
```

- ① Симметрическая разность множеств `a_set` и `b_set` *выглядит* не так, как симметрическая разность множеств `b_set` и `a_set`, но вспомните, множества неупорядочены. Любые два множества, все (без исключения) значения которых одинаковы, считаются равными.
- ② Именно это здесь и произошло. Глядя на печатное представление этих множеств, созданное оболочкой Python, не обманывайтесь. Значения элементов этих множеств одинаковы, поэтому они равны.
- ③ Объединение двух множеств также симметрично.
- ④ Пересечение двух множеств также симметрично.
- ⑤ Разность двух множеств несимметрична. По смыслу, данная операция аналогична вычитанию одного числа из другого. Порядок операндов имеет значение.

Наконец, есть ещё несколько вопросов по множествам, которые вы можете задать.

```
>>> a_set = {1, 2, 3}
>>> b_set = {1, 2, 3, 4}
>>> a_set.issubset(b_set) ①
True
>>> b_set.issuperset(a_set) ②
True
```



```
>>> a_set.add(5) ③
>>> a_set.issubset(b_set)
False
>>> b_set.issuperset(a_set)
False
```

- ① Множество `a_set` является подмножеством `b_set` — все элементы `a_set` также являются элементами `b_set`.
- ② И наоборот, `b_set` является надмножеством `a_set`, потому что все элементы `a_set` также являются элементами `b_set`.
- ③ Поскольку вы добавили элемент в `a_set`, но не добавили в `b_set`, обе проверки вернут значение `False`.

## Множества в логическом контексте

Вы можете использовать множества в логическом контексте, например, в операторе `if`.

```
>>> def is_it_true(anything):
...     if anything:
...         print("да, это истина")
...     else:
...         print("нет, это ложь")
...
>>> is_it_true(set()) ①
нет, это ложь
>>> is_it_true({'a'}) ②
да, это истина
>>> is_it_true({False}) ③
да, это истина
```

- ① В логическом контексте пустое множество — ложь.
- ② Любое множество, содержащее хотя бы один элемент — истина.
- ③ Любое множество, содержащее хотя бы один элемент — истина. Значения элементов не важны.

## Словари

Словарь — это неупорядоченное множество пар ключ—значение. Когда вы добавляете ключ в словарь, вы также должны добавить и значение для этого ключа. (Значение всегда можно изменить позже.) Словари в Python оптимизированы для получения значения по известному ключу, но не для других целей.



Словарь в Python аналогичен хэшу в Perl 5. В Perl 5 переменные, хранящие хэши, всегда начинаются с символа %. В Python переменные могут быть названы как угодно, язык сам отслеживает типы данных.

## Создание словаря

Создать словарь очень просто. Синтаксис похож на синтаксис создания множеств, но вместо элементов, используются пары ключ-значение. Если у вас есть словарь, вы можете просматривать значения по их ключу.

```
>>> a_dict = {'server': 'db.diveintopython3.org', 'database': 'mysql'} ①
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['server'] ②
'db.diveintopython3.org'
>>> a_dict['database'] ③
'mysql'
>>> a_dict['db.diveintopython3.org'] ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'db.diveintopython3.org'
```

*Перевод сообщения оболочки:*

```
Раскрытие стека (список последних вызовов):
  Файл "<stdin>", строка 1, <модуль>
KeyError: 'db.diveintopython3.org'
```

- ① Сначала вы создаёте новый словарь с двумя элементами и присваиваете его переменной `a_dict`. Каждый элемент является парой ключ—значение, а весь набор элементов заключён в фигурные скобки.
- ② `'server'` является ключом, и он связан со значением, обращение к которому с помощью `a_dict['server']` даст нам `'db.diveintopython3.org'`.
- ③ `'database'` является ключом, и он связан со значением, обращение к которому с помощью `a_dict['database']` даст нам `'mysql'`.
- ④ Можно получить значение по ключу, но нельзя получить ключи по значению. Так `a_dict['server']` — это `'db.diveintopython3.org'`, но `a_dict['db.diveintopython3.org']` породит исключение, потому что `'db.diveintopython3.org'` не является ключом.

## Изменение словаря

Словари не имеют какого-либо предопределенного ограничения размера. Когда угодно можно добавлять новые пары ключ—значение в словарь или

изменять значение, соответствующее существующему ключу. Продолжим предыдущий пример:

```
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['database'] = 'blog' ①
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'blog'}
>>> a_dict['user'] = 'mark' ②
>>> a_dict ③
{'server': 'db.diveintopython3.org', 'user': 'mark', 'database': 'blog'}
>>> a_dict['user'] = 'dora' ④
>>> a_dict
{'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}
>>> a_dict['User'] = 'mark' ⑤
>>> a_dict
{'User': 'mark', 'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}
```

- ① Ваш словарь не может содержать одинаковые ключи. Присвоение значения существующему ключу уничтожит старое значение.
- ② Можно добавлять новые пары ключ—значение когда угодно. Данный синтаксис идентичен синтаксису модифицирования существующих значений.
- ③ Кажется, что новый элемент словаря (ключ 'user', значение 'mark') попал в середину. На самом деле, это всего лишь совпадение, что элементы кажутся расположенными по порядку в первом примере; такое же совпадение, что теперь они выглядят расположенными не по порядку.
- ④ Присвоение значения существующему ключу просто заменяет старое значение новым.
- ⑤ Изменится ли значение ключа 'user' обратно на "mark"? Нет! Посмотрите на него внимательнее — ключ "User" написан с заглавной буквы. Ключи словаря регистрозависимы, поэтому это выражение создаст новую пару ключ—значение, а не перезапишет существующую. Вам кажется, что ключи похожи, а с точки зрения Python они абсолютно разные.

## Словари со смешанными значениями

Словари могут состоять не только из строк. Значения словарей могут быть любого типа, включая целые, логические, произвольные объекты, или даже другие словари. И значения в одном словаре не обязаны быть одного и того же типа; можно смешивать и сочетать их, как вам необходимо. Ключи словаря более ограничены, но они могут быть строками, целыми числами и

некоторыми другими типами. Ключи разных типов тоже можно смешивать и сочетать в одном словаре.

На самом деле вы уже видели словарь с не строковыми ключами и значениями в вашей первой программе на Python.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

Давайте вытащим эту переменную из нашей программы и поработаем с ней в интерактивной оболочке Python.

```
>>> SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
...             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
>>> len(SUFFIXES)    ①
2
>>> 1000 in SUFFIXES ②
True
>>> SUFFIXES[1000]   ③
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>> SUFFIXES[1024]   ④
['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']
>>> SUFFIXES[1000][3] ⑤
'TB'
```

- ① Так же, как для [списков](#) и [множеств](#), функция `len()` возвращает количество элементов словаря.
- ② И так же, как со списками и множествами, можно использовать оператор `in`, чтобы проверить, определён ли конкретный ключ в словаре.
- ③ 1000 *является* ключом в словаре `SUFFIXES`; его значение — список из восьми элементов (восемью строкам, если быть точным).
- ④ Аналогично, 1024 — ключ словаря `SUFFIXES`; и его значение также является списком из восьми элементов.
- ⑤ Так как `SUFFIXES[1000]` является списком, то можно обратиться к отдельным элементам списка по их порядковым номерам, которые индексируются с нуля.

## Словари в логическом контексте

*Пустые словари — ложь, все остальные — истина.*

Вы можете использовать словари в логическом контексте, например в операторе if.

```
>>> def is_it_true(anything):
...     if anything:
...         print("да, это истина")
...     else:
...         print("нет, это ложь")
...
>>> is_it_true({})           ①
нет, это ложь
>>> is_it_true({'a': 1})     ②
да, это истина
```

① В логическом контексте пустой словарь ложен.

② Любой словарь с хотя бы одной парой ключ—значение истинен.

## Константа **None**

None — это специальная константа в Python. Она обозначает пустое значение. None — это не то же самое, что False. None также и не 0. None даже не пустая строка. Если сравнивать None с другими типами данных, то результатом всегда будет False.

None — это просто пустое значение. None имеет свой собственный тип (NoneType). Вы можете присвоить None любой переменной, но вы не можете создать других объектов типа NoneType. Все переменные, значение которых None равны друг другу.

```
>>> type(None)
<class 'NoneType'>
>>> None == False
False
>>> None == 0
False
>>> None == ""
False
>>> None == None
True
>>> x = None
>>> x == None
True
>>> y = None
>>> x == y
True
```

## None в логическом контексте

В логическом контексте None всегда является ложью, а not None — истиной.

```
>>> def is_it_true(anything):
...     if anything:
...         print("да, это истина")
...     else:
...         print("нет, это ложь")
...
>>> is_it_true(None)
нет, это ложь
>>> is_it_true(not None)
да, это истина
```

## Материалы для дальнейшего чтения

- Логические операции
  - Численные типы
  - Типы-последовательности
  - Типы-множества
  - Типы-отображения
  - Модуль `fractions`
  - Модуль `math`
  - PEP 237: Унификация длинных целых и целых
  - PEP 238: Изменение оператора деления
-

# Генераторы

Нам приходится сильнее напрягать свое воображение не для того, чтобы, как в художественной литературе, представить себе то, чего нет на самом деле, а для того, чтобы постичь то, что действительно происходит.

*Ричард Фейнман*

## Погружение

В каждом языке программирования есть одна такая особенность, сложно устроенная, но специально упрощённая штука. Если вы раньше писали на другом языке, можете и не обратить на это внимания, поскольку ваш старый язык не так сильно упрощал эту штуку (потому что он был занят тем, что сильно упрощал какую-нибудь другую штуку). В этой главе вы изучите генераторы списков, словарей и множеств — три взаимосвязанные концепции, сконцентрированные вокруг одной очень мощной технологии. Но сначала я хочу немного отклониться от нашего повествования, чтобы рассказать вам о двух модулях, которые помогут вам передвигаться по вашей локальной файловой системе.

## Работа с файлами и каталогами

Python 3 поставляется с модулем `os`, что означает «операционная система». Модуль `os` содержит множество функций для получения информации о локальных каталогах, файлах, процессах и переменных окружения (а в некоторых случаях, и для манипулирования ими). Python предлагает очень хороший унифицированный программный интерфейс для всех поддерживаемых операционных систем, так что ваши программы можно запускать на любом компьютере с минимальным количеством платформо-зависимого кода.

## Текущий рабочий каталог

Когда ваше знакомство с Python только начинается, вы много времени проводите в интерактивной оболочке Python. На протяжении всей этой книги вы будете видеть примеры, выглядящие следующим образом:



1. Импортрование какого-либо модуля из папки примеров
2. Вызов функции из этого модуля
3. Объяснение результата

## Всегда есть текущий рабочий каталог.

Если вы ничего не знаете о текущем рабочем каталоге, то, возможно, шаг 1 окажется неудачным и будет порождено исключение типа `ImportError`. Почему? Потому что Python будет искать указанный модуль в пути поиска оператора `import`, но не найдёт его, потому что каталог `examples` не содержится в путях поиска. Чтобы исправить это, вы можете сделать одно из двух:

- либо добавить папку `examples` в путь поиска оператора `import`;
- либо сделать текущим рабочим каталогом папку `examples`.

Текущий рабочий каталог является неявным параметром, который Python постоянно хранит в памяти. Текущий рабочий каталог есть всегда, когда вы работаете в интерактивной оболочке Python, запускаете свой сценарий из командной строки или CGI-сценарий где-то на веб-сервере.

Модуль `os` содержит две функции для работы с текущим рабочим каталогом.

```
>>> import os [1]
>>> print(os.getcwd()) [2]
C:\Python31
>>> os.chdir('/Users/pilgrim/diveintopython3/examples') [3]
>>> print(os.getcwd()) [4]
C:\Users\pilgrim\diveintopython3\examples
```

1. ↑ Модуль `os` поставляется вместе с Python; вы можете импортировать его когда угодно и где угодно.
2. ↑ Используйте функцию `os.getcwd()` для получения значения текущего рабочего каталога. Когда вы работаете в графической оболочке Python, текущим рабочим каталогом является каталог из которого она была запущена. В Windows это зависит от того, куда вы установили Python; каталог по умолчанию `c:\Python31`. Если оболочка Python запущена из командной строки, текущим рабочим каталогом считается тот, в котором вы находились, когда запускали её.
3. ↑ Используйте функцию `os.chdir()` чтобы сменить текущий рабочий каталог.
4. ↑ Когда я вызывал функцию `os.chdir()`, я использовал путь в стиле Linux (косая черта, нет буквы диска) даже если на самом деле работал в

Windows. Это одно из тех мест, где Python пытается стирать различия между операционными системами.

## Работа с именами файлов и каталогов

Раз зашла речь о каталогах, я хочу обратить ваше внимание на модуль `os.path`. Он содержит функции для работы с именами файлов и каталогов.

```
>>> import os
```

```
>>> print(os.path.join('/Users/pilgrim/diveintopython3/examples/', 'humansize.py')) [1]
/Users/pilgrim/diveintopython3/examples/humansize.py
```

```
>>> print(os.path.join('/Users/pilgrim/diveintopython3/examples', 'humansize.py')) [2]
/Users/pilgrim/diveintopython3/examples\humansize.py
```

```
>>> print(os.path.expanduser('~')) [3]
c:\Users\pilgrim
```

```
>>> print(os.path.join(os.path.expanduser('~'), 'diveintopython3', 'examples', [4]
'humansize.py'))
c:\Users\pilgrim\diveintopython3\examples\humansize.py
```

1. ↑ Функция `os.path.join()` составляет путь к каталогу из одного или нескольких частичных путей. В данном случае она просто соединяет строки.
2. ↑ Это уже менее тривиальный случай. Функция `join` добавит дополнительную косую черту (slash) к имени папки перед тем как дописать имя файла. В данном случае Python добавляет обратную косую черту (backslash) вместо обыкновенной, потому что я запустил этот пример в Windows. Если вы введёте данную команду в Linux или Mac OS X, вы увидите простую косую черту. Python может обратиться к файлу независимо от того, какой разделитель используется в пути к файлу.
3. ↑ Функция `os.path.expanduser()` раскрывает путь, в котором используется символ `~` для обозначения домашнего каталога текущего пользователя. Функция работает на любой платформе, где у пользователя есть домашний каталог, включая Linux, Mac OS X и Windows. Функция возвращает путь без косой черты в конце, но для функции `os.path.join()` это не имеет значения.
4. ↑ Комбинируя эти две функции, вы можете легко строить файловые пути для папок и файлов в домашнем каталоге пользователя. Функция `os.path.join()` принимает любое количество аргументов. Я получил огромное удовольствие, когда обнаружил это, так как в других языках при разработке инструментальных средств мне приходилось постоянно

писать глупую маленькую функцию `addSlashIfNecessary()`. В языке программирования Python умные люди уже позаботились об этом.

Модуль `os.path` также содержит функции для разбиения файловых путей, имён папок и файлов на их составные части.

```
>>> pathname = '/Users/pilgrim/diveintopython3/examples/humansize.py'
>>> os.path.split(pathname) [1]
('/Users/pilgrim/diveintopython3/examples', 'humansize.py')
>>> (dirname, filename) = os.path.split(pathname) [2]
>>> dirname [3]
'/Users/pilgrim/diveintopython3/examples'
>>> filename [4]
'humansize.py'
>>> (shortname, extension) = os.path.splitext(filename) [5]
>>> shortname
'humansize'
>>> extension
'.py'
```

1. ↑ Функция `split` дробит полный путь и возвращает кортеж, содержащий отдельно путь до каталога и имя файла.
2. ↑ Помните, я рассказывал про то, как присваивать несколько значений за раз и как вернуть одновременно несколько значений из функции? Функция `os.path.split()` действует именно так. Можно присвоить возвращаемое из функции `split` значение кортежу из двух переменных. Каждая из переменных примет значение соответствующего элемента результирующего кортежа.
3. ↑ Первая переменная — `dirname` — получит значение первого элемента кортежа, возвращаемого функцией `os.path.split()`, а именно путь до каталога.
4. ↑ Вторая переменная — `filename` — примет значение второго элемента кортежа, возвращаемого функцией `os.path.split()`, а именно имя файла.
5. ↑ Модуль `os.path` также содержит функцию `os.path.splitext()`, которая дробит имя файла и возвращает кортеж, содержащий отдельно имя и отдельно расширение файла. Можно использовать ту же технику, что и ранее для присваивания каждого из интересующих значений отдельным переменным.

## Получение содержимого каталога

*Модуль `glob` понимает символы-джокеры, использующиеся в командных оболочках.*

Модуль `glob` — это ещё один инструмент из стандартной библиотеки Python. Это простой способ программно получить содержимое папки, а также он умеет использовать символы-джокеры, с которыми вы наверняка знакомы, если работали в командной строке.

```
>>> os.chdir('/Users/pilgrim/diveintopython3/')
>>> import glob
```

```
>>> glob.glob('examples/*.xml') [1]
['examples\\feed-broken.xml',
 'examples\\feed-ns0.xml',
 'examples\\feed.xml']
```

```
>>> os.chdir('examples/') [2]
```

```
>>> glob.glob('*test*.py') [3]
['alphameticstest.py',
 'pluraltest1.py',
 'pluraltest2.py',
 'pluraltest3.py',
 'pluraltest4.py',
 'pluraltest5.py',
 'pluraltest6.py',
 'romantest1.py',
 'romantest10.py',
 'romantest2.py',
 'romantest3.py',
 'romantest4.py',
 'romantest5.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

1. ↑ Модуль `glob` принимает шаблон, содержащий символы-джокеры, и возвращает пути всех файлов и каталогов, соответствующих ему. В этом примере шаблон содержит путь к каталогу и `"*.xml"`, которому будут соответствовать все `xml`-файлы в каталоге `examples`.

2. ↑ Теперь сделаем текущим рабочим каталог `examples`. Функция `os.chdir()` может принимать и относительные пути.
3. ↑ Вы можете использовать несколько символов-джокеров в своём шаблоне. Этот пример находит все файлы в текущем рабочем каталоге, заканчивающиеся на `.py` и содержащие слово `test` где-нибудь в имени файла.

## Получение сведений о файле

Любая современная операционная система хранит сведения о каждом файле (метаданные): дата создания, дата последней модификации, размер файла и т. д. Python предоставляет единый программный интерфейс для доступа к этим метаданным. Вам не надо открывать файл; всё, что требуется — имя файла.

```
>>> import os
>>> print(os.getcwd())
c:\Users\pilgrim\diveintopython3\examples
>>> metadata = os.stat('feed.xml')
>>> metadata.st_mtime
1247520344.9537716
>>> import time
>>> time.localtime(metadata.st_mtime)
time.struct_time(tm_year=2009, tm_mon=7, tm_mday=13, tm_hour=17,
tm_min=25,
tm_sec=44, tm_wday=0, tm_yday=194, tm_isdst=1)
```

1. ↑ Текущий рабочий каталог — папка с примерами.
2. ↑ `feed.xml` — файл в папке с примерами. Вызов функции `os.stat()` возвращает объект, содержащий различные метаданные о файле.
3. ↑ `st_mtime` — время изменения файла, но записано оно в ужасно неудобном формате. (Фактически это количество секунд, прошедших с начала «эры UNIX», начавшейся в первую секунду 1 января 1970 года. Серьёзно.)
4. ↑ Модуль `time` является частью стандартной библиотеки Python. Он содержит функции для преобразований между различными форматами представления времени и часовыми поясами, для преобразования их в строки (`str`) и др.
5. ↑ Функция `time.localtime()` преобразует время из формата «секунды с начала эры» (поле `st_mtime`, возвращённое функцией `os.stat()`) в более удобную структуру, содержащую год, месяц, день, час, минуту, секунду

и т. д. Этот файл в последний раз изменялся 13 июля 2009 года, примерно в 17 часов, 25 минут.

```
# продолжение предыдущего примера
>>> metadata.st_size [1]
3070
>>> import humansize
>>> humansize.approximate_size(metadata.st_size) [2]
'3.0 KiB'
```

1. ↑ Функция `os.stat()` также возвращает размер файла в свойстве `st_size`. Размер файла `feed.xml` — 3070 байт.
2. ↑ Вы можете передать свойство `st_size` в функцию `approximate_size()`.

## Получение абсолютных путей

В предыдущем разделе функция `glob.glob()` возвращала список относительных путей. В первом примере пути имели вид `'examples/feed.xml'`, а во втором относительные пути были даже короче, например, `'romantest1.py'`. Пока вы остаётесь в текущем рабочем каталоге, по этим относительным путям можно будет открывать файлы или получать их метаданные. Но если вы захотите получить абсолютный путь — то есть тот, который включает все имена каталогов до корневого или до буквы диска, вам понадобится функция `os.path.realpath()`.

```
>>> import os
>>> print(os.getcwd())
c:\Users\pilgrim\diveintopython3\examples
>>> print(os.path.realpath('feed.xml'))
c:\Users\pilgrim\diveintopython3\examples\feed.xml
```

## Генераторы списков

В генераторах списков можно использовать любые выражения Python.

С помощью генераторов списков можно легко отобразить один список в другой, применив некоторую функцию к каждому элементу.

```
>>> a_list = [1, 9, 8, 4]
>>> [elem * 2 for elem in a_list] [1]
[2, 18, 16, 8]
>>> a_list [2]
[1, 9, 8, 4]
```

```
>>> a_list = [elem * 2 for elem in a_list][3]
>>> a_list
[2, 18, 16, 8]
```

1. ↑ Чтобы понять, что здесь происходит, прочитайте генератор справа налево. `a_list` — отображаемый список. Python последовательно перебирает элементы списка `a_list`, временно присваивая значение каждого элемента переменной `elem`. Затем применяет функцию `elem * 2` и добавляет результат в возвращаемый список.
2. ↑ Генератор создаёт новый список, не изменяя исходный.
3. ↑ Можно присвоить результат работы генератора списка отображаемой переменной. Python создаст новый список в памяти и, когда результат работы генератора будет получен, присвоит его исходной переменной.

В генераторах списков можно использовать любые выражения Python, включая функции модуля `os`, применяемые для работы с файлами и каталогами.

```
>>> import os, glob
>>> glob.glob('*.*xml')[1]
['feed-broken.xml', 'feed-ns0.xml', 'feed.xml']

>>> [os.path.realpath(f) for f in glob.glob('*.*xml')][2]
['c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml']
```

1. ↑ Это выражение возвращает список всех `.xml`-файлов в текущем рабочем каталоге.
2. ↑ Этот генератор принимает список всех `.xml`-файлов и преобразует его в список полных путей.

При генерировании списков можно также фильтровать элементы, чтобы отбросить некоторые значения.

```
>>> import os, glob

>>> [f for f in glob.glob('*.*py') if os.stat(f).st_size > 6000][1]
['pluraltest6.py',
 'romantest10.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```



1. ↑ Чтобы профильтровать список, добавьте оператор `if` в конце генератора списка. Выражение, стоящее после оператора `if`, будет вычислено для каждого элемента списка. Если это выражение будет истинно, данный элемент будет обработан и включён в генерируемый список. В данной строке генерируется список всех `.py`-файлов в текущей директории, а оператор `if` фильтрует этот список, оставляя только файлы размером больше 6000 байт. Таких файлов только шесть, поэтому будет сгенерирован список из шести имён файлов.

Все рассмотренные примеры генераторов списков использовали простые выражения: умножение числа на константу, вызов одной функции или просто возврат элемента списка без изменений (после фильтрации).

Но при генерации списков можно использовать выражения любой сложности.

```
>>> import os, glob
```

```
>>> [(os.stat(f).st_size, os.path.realpath(f)) for f in glob.glob('*.xml')]
[(3074, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml'),
 (3386, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml'),
 (3070, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml')]
[1]
```

```
>>> import humanize
```

```
>>> [(humanize.approximate_size(os.stat(f).st_size), f) for f in glob.glob('*.xml')]
[('3.0 KiB', 'feed-broken.xml'),
 ('3.3 KiB', 'feed-ns0.xml'),
 ('3.0 KiB', 'feed.xml')]
[2]
```

1. ↑ Этот генератор ищет все `.xml`-файлы в текущем рабочем каталоге, получает размер каждого файла (вызывая функцию `os.stat()`), и создает кортеж из размера файла и абсолютного пути каждого файла (вызывая функцию `os.path.realpath()`).
2. ↑ Этот генератор, основанный на предыдущем, вызывает функцию `approximate_size()`, передавая ей размер каждого `.xml`-файла.

## Генераторы словарей

Генератор словаря похож на генератор списка, но вместо списка он создает словарь.

```
>>> import os, glob
```

```
>>> metadata = {(f, os.stat(f)) for f in glob.glob('*test*.py')}
[1]
```

```
>>> metadata[0]
('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0, st_dev=0,
st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
[2]
```

```

st_mtime=1247520344, st_ctime=1247520344))

>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*test*.py')} [3]

>>> type(metadata_dict) [4]
<class 'dict'>

>>> list(metadata_dict.keys()) [5]
['romantest8.py', 'pluraltest1.py', 'pluraltest2.py', 'pluraltest5.py',
'pluraltest6.py', 'romantest7.py', 'romantest10.py', 'romantest4.py',
'romantest9.py', 'pluraltest3.py', 'romantest1.py', 'romantest2.py',
'romantest3.py', 'romantest5.py', 'romantest6.py', 'alphameticstest.py',
'pluraltest4.py']

>>> metadata_dict['alphameticstest.py'].st_size [6]
2509

```

1. ↑ Это не генератор словаря, это генератор списка. Он находит все файлы с расширением .py, проверяет их имена, а затем создает кортеж из имени файла и метаданных файла (вызывая функцию `os.stat()`).
2. ↑ Каждый элемент результирующего списка — кортеж.
3. ↑ Это генератор словаря. Синтаксис подобен синтаксису генератора списка, но с двумя отличиями. Во-первых, он заключён в фигурные скобки, а не в квадратные. Во-вторых, вместо одного выражения для каждого элемента он содержит два, разделённые двоеточием. Выражение слева от двоеточия (в нашем примере `f`) является ключом словаря; выражение справа от двоеточия (в нашем примере `os.stat(f)`) — значением.
4. ↑ Генератор словаря возвращает словарь.
5. ↑ Ключи данного словаря — это просто имена файлов, полученные с помощью `glob.glob('*test*.py')`.
6. ↑ Значение, связанное с каждым ключом, получено с помощью функции `os.stat()`. Это означает, что в этом словаре мы можем по имени файла получить его метаданные. Один из элементов метаданных (`st_size`) — это размер файла. Размер файла `alphameticstest.py` — 2509 байт.

Также, как и в генераторах списков, вы можете включать в генераторы словарей условие `if`, чтобы отфильтровать входную последовательность с помощью выражения-условия, вычисляющегося для каждого элемента.

```

>>> import os, glob, humansize

>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*')} [1]

>>> humansize_dict = [2]
{os.path.splitext(f)[0]:humansize.approximate_size(meta.st_size) \

```

```

...         for f, meta in metadata_dict.items() if meta.st_size > 6000}

>>> list(humansize_dict.keys())
['romantest9', 'romantest8', 'romantest7', 'romantest6', 'romantest10', 'pluraltest6']

>>> humansize_dict['romantest9']
'6.5 KiB'

```

1. ↑ В этом выражении берётся список файлов в текущей директории (`glob.glob('*')`), для каждого файла определяются его метаданные (`os.stat(f)`) и строится словарь, ключами которого выступают имена файлов, а значениями — метаданные каждого файла.
2. ↑ Этот генератор строится на основе предыдущего. Отфильтровываются файлы меньше 6000 байт (`if meta.st_size > 6000`). Отобранные элементы используются для построения словаря, ключами которого являются имена файлов без расширения (`os.path.splitext(f)[0]`), а значениями — приблизительный размер каждого файла (`humansize.approximate_size(meta.st_size)`).
3. ↑ Как вам уже известно из предыдущего примера, всего имеется шесть таких файлов, следовательно, в этом словаре шесть элементов.
4. ↑ Значение для каждого ключа представляется из себя строку, полученную вызовом функции `approximate_size()`.

## Другие интересные штуки, которые можно делать с помощью генераторов словарей

Вот трюк с генераторами словарей, который когда-нибудь может оказаться полезным: перестановка местами ключей и значений словаря.

```

>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> {value:key for key, value in a_dict.items()}
{1: 'a', 2: 'b', 3: 'c'}

```

Конечно же, это сработает, только если значения элементов словаря неизменяемы, как, например, строки или кортежи.

```

>>> a_dict = {'a': [1, 2, 3], 'b': 4, 'c': 5}
>>> {value:key for key, value in a_dict.items()}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <dictcomp>
TypeError: unhashable type: 'list'

```

## Генераторы множеств

Нельзя оставить за бортом и множества, они тоже могут создаваться с помощью генераторов. Единственное отличие — вместо пар ключ:значение, они строятся на основе одних значений.

```
>>> a_set = set(range(10))
>>> a_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

>>> {x ** 2 for x in a_set} [1]
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> {x for x in a_set if x % 2 == 0} [2]
{0, 8, 2, 4, 6}

>>> {2**x for x in range(10)} [3]
{32, 1, 2, 4, 8, 64, 128, 256, 16, 512}
```

1. ↑ В качестве входных данных генераторы множеств могут получать другие множества. Этот генератор рассчитывает квадраты множества чисел в диапазоне от 0 до 9.
2. ↑ Подобно генераторам списков и словарей, генераторы множеств могут содержать условие `if` для проверки каждого элемента перед включением его в результирующее множество.
3. ↑ На вход генераторы множеств могут принимать не только множества, но и любые другие последовательности.

## Материалы для дальнейшего чтения

- Модуль `os`
  - `os` — доступ к особым возможностям операционных систем
  - Модуль `os.path`
  - `os.path` — манипуляции с именами файлов, независимые от платформы
  - Модуль `glob`
  - `glob` — сравнение имён файлов с шаблонами
  - Модуль `time`
  - `time` — Функции для манипулирования временем
  - Генераторы списков
  - Вложенные генераторы списков
  - Техника циклов
-

# Строки

Немного скучных вещей, которые вам необходимо знать перед погружением

Знаете ли вы, что у народа острова Бугенвиль самый короткий алфавит в мире? Алфавит языка ротокас состоит всего из 12 букв: A, E, G, I, K, O, P, R, S, T, U, и V. На другом конце этой своеобразной числовой оси расположились такие языки, как китайский, японский и корейский, насчитывающие тысячи символов. Английский, конечно, содержит всего 26 букв — 52, если считать буквы и верхнего, и нижнего регистров — плюс горстка знаков пунктуации !@#\$%&.

Когда люди говорят «текст», они подразумевают «буквы и символы на экране компьютера». Но компьютеры не работают с буквами и символами; они работают с битами и байтами. Каждый фрагмент текста, который вы когда-либо видели на экране компьютера, на самом деле хранится в определенной кодировке. Грубо говоря, кодировка символов обеспечивает соответствие того, что вы видите на экране и того, что на самом деле хранится в памяти или на диске. Существует много различных кодировок символов, некоторые из них оптимизированы для конкретных языков, например русского, китайского или английского, другие могут быть использованы сразу для нескольких языков.

В действительности, все гораздо сложнее. Многие символы являются общими для нескольких кодировок, но каждая кодировка может использовать свою последовательность байтов для хранения их в памяти или на диске. Вы можете думать о кодировке символов, как о разновидности криптографического ключа. Всякий раз, когда вам передают последовательность байтов — файл, веб-страницу, все равно — и утверждают, что это «текст», вам необходимо понять, какая кодировка использовалась. Зная кодировку, вы сможете декодировать байты в символы. Если вам дают неправильный ключ или не дают ключа вовсе, вам не остается ничего, кроме как попытаться взломать код самостоятельно. Скорее всего, в результате вы получите кучу крякозябров (gibberish — тарабарщина, невнятная речь, но так понятнее, прим. перев.). Все, что вы знали о строках — неверно.

*Все, что вы знали о строках — неверно.*

Несомненно, вам приходилось видеть такие веб-страницы, со странными вопросительными знаками на месте апострофов. Обычно это означает, что автор страницы неправильно указал их кодировку, вашему браузеру осталось просто угадать ее, а результатом стала смесь ожидаемых и совершенно неожиданных символов. В английском языке это просто раздражает; в других языках результат может стать совершенно нечитаемым.

Существуют кодировки для всех основных мировых языков. Но, поскольку, языки существенно отличаются друг от друга, а память и дисковое пространство раньше были дорогими, каждая кодировка оптимизирована для конкретного языка. Под этим я подразумеваю то, что для представления символов своего языка, все кодировки используют один и тот же диапазон чисел (0-255). Например, вы вероятно знакомы с кодировкой ASCII, которая хранит символы английского языка в виде чисел от 0 до 127. (65 — заглавная «А», 97 — строчная «а» и т.д.) Английский алфавит очень простой и может быть представлен менее, чем 128 числами. Если вам известна двоичная система счисления, вы понимаете, что в байте задействовано всего 7 битов из 8.

Западноевропейские языки, такие как французский, испанский и немецкий содержат больше символов, чем английский. Точнее, они содержат символы с различными диакритическими знаками, например, испанский символ ñ. Самая распространенная кодировка для этих языков — CP-1252, также известная как «windows-1252», что связано с широким использованием ее в операционной системе Microsoft Windows. В кодировке CP-1252 символы с номерами от 0 до 127 такие же, как и в ASCII, а остальной диапазон используется для таких символов как п-с-тильдой-сверху (241), и-с-двумя-точками-сверху (252) и т.д. Однако, это все еще однобайтная кодировка; максимально возможный номер 255 еще помещается в один байт.

А еще существуют такие языки, как китайский, японский и корейский, которые имеют так много символов, что они требуют многобайтовых кодировок. Это означает, что каждый «символ» представляется двухбайтовым числом от 0 до 65535. Но различные многобайтные кодировки всё равно имеют ту же проблему, что и различные однобайтные кодировки: каждая кодировка использует одинаковые числа для обозначения разных вещей. Отличие лишь в том, что диапазон чисел больше, потому что нужно кодировать намного больше символов.

Это было вполне нормально в несетевом мире, где вы набирали «текст» для себя и иногда распечатывали его. В этом мире не было ничего кроме «обычного текста» (не знаю, м.б. быть вообще не переводить “plain text” — прим. перев.). Исходный код был в кодировке ASCII, а для всего остального использовались текстовые процессоры, которые определяли свои собственные (нетекстовые) форматы, в которых, наряду с информацией о форматировании, отслеживалась и информация о кодировке. Люди читают эти документы с помощью такого же текстового процессора, какой использовался и для их создания, так что, все более или менее работало.

Теперь подумайте о распространении глобальных сетей, таких как e-mail и web. Множество «обычного текста» перемещается вокруг планеты, создаётся на одном компьютере, передаётся через второй и принимается и отображается третьим компьютером. Компьютеры видят только числа, но числа могут иметь различное значение. О нет! Что же делать? Системы были спроектированы таким образом, чтобы передавать информацию о кодировке вместе с каждым отрывком «обычного текста». Вспомните, это криптографический ключ, устанавливающий соответствие между числами и понятными человеку символами. Потерянный ключ означает искаженный текст или кракозябры, если не хуже.

Теперь подумайте о задаче хранения различных отрывков текста в одном месте, например в одной таблице базы данных, хранящей все когда-либо полученные вами email сообщения. Вам всё ещё нужно хранить кодировку вместе с каждым отрывком текста, чтобы иметь возможность прочитать его. Думаете, это трудно? Попробуйте реализовать поиск в этой базе данных, с преобразованием на лету между множеством кодировок. Разве это не забавно?

Теперь подумайте о возможности многоязычных документов, где символы из нескольких языков находятся рядом в одном документе. (Подсказка: программы, которые пытаются делать это, обычно используют коды смены алфавита для переключения «режимов». Бац, и вы в русском режиме KOI8-R, и 241 означает Я; бац, и теперь вы в греческом режиме для Macintosh, и 241 означает ώ.) И конечно вы так же захотите осуществлять поиск по этим документам.

Теперь плачьте, т.к. все, что вы знали о строках — неверно, и нет такого понятия «обычный текст».

## Юникод

### *Введение в Юникод.*

Юникод спроектирован для представления системой каждого символа любого языка. Юникод представляет каждую букву, символ или идеографию как 4-х байтное число. Каждое число представляет уникальный символ, используемый по крайней мере в одном из языков в мире. (используются больше чем 65535 из них, таким образом 2 байта не были бы достаточны.) У символов, которые используются в разных языках, один код, если нет хорошей этимологической причины. Независимо от всего, есть точно 1 код соответствующий символу, и точно 1 символ соответствующий числовому коду. Каждый код всегда означает только один символ; нет никаких «режимов». U+0041 всегда соответствует 'A', даже если в вашем языке нету символа 'A'.

На первый взгляд, это великолепная идея. Одна кодировка для всего. Множество языков в одном документе. Не надо больше никаких «переключений режимов» для смены кодировок. Но у вас должен возникнуть



очевидный вопрос. Четыре байта? На каждый символ? Это кажется ужасно расточительным, особенно для таких языков, как английский или испанский, в которых нужно меньше одного байта (256 чисел) для представления любого возможного символа. На самом деле, это расточительно даже для иероглифических языков (таких как китайский), в которых никогда не нужно больше, чем два байта на символ.

Существует кодировка Юникод, которая использует четыре байта на символ. Она называется UTF-32, так как 32 бита = 4 байтам. UTF-32 — прямолинейная кодировка; каждому символу Юникод (4-байтовому числу) соответствует символ с определенным номером. Это имеет свои преимущества, самое важное из которых заключается в том, что вы можете найти N-ый символ в строке за постоянное время, так как N-ый символ начинается в  $4 \cdot N$  байте. Но эта кодировка также имеет и недостатки, самый очевидный из которых — для хранения каждого символа требуется четыре байта.

Хотя в Юникоде существует огромное количество символов, на самом деле большинство людей никогда не используют те, номера которых выше 65535 ( $2^{16}$ ). Поэтому существует другая кодировка Юникода, называемая UTF-16 (очевидно что 16 бит = 2 байтам). UTF-16 кодирует каждый символ в номерах 0–65535; для представления же редко используемых "запредельных" символов с номерами выше 65535 приходится прибегать к некоторым уловкам. Самое очевидное преимущество: UTF-16 дважды эффективнее по потреблению памяти, нежели UTF-32, так как каждый символ требует 2 байта вместо 4-х (кроме случаев с теми самыми "запредельными" символами). И, как и в случае с UTF-32, можно легко отыскать нужный N-ый символ в строке за постоянное время, если вы уверены, что текст не содержит "запредельных" символов; и всё хорошо, если это действительно так.

Тем не менее существуют неочевидные недостатки, как UTF-32, так и UTF-16. На различных компьютерных платформах отдельные байты хранятся по-разному. Это означает, что символ U+4E2D может быть сохранён в UTF-16 либо как 4E 2D, либо как 2D 4E (задом наперёд), в зависимости от используемого порядка байт: big-endian или little-endian. (Для UTF-32 видов порядков даже больше.) Пока вы храните свои документы исключительно у себя на компьютере, вы в безопасности — различные приложения на одном компьютере всегда используют один и тот же порядок. Но в тот момент, когда вы захотите передать документы на другой компьютер в Интернете или иной сети, вам понадобится способ пометки документа, какой у вас используется порядок байт. В противном случае, принимающая документ система не имеет понятия, что представляет последовательность байт "4E 2D": U+4E2D или U+2D4E.

Для решения этой проблемы многобайтовые кодировки Юникода имеют "отметку о порядке байт" (BOM), которая представляет собой специальный непечатаемый символ, который вы можете включить в начало документа для сохранения информации о используемом порядке байт. Для UTF-16, эта отметка имеет номер U+FEFF. Если вы принимаете документ с UTF-16,

который начинается с байт FF FE, — это однозначно оповещает о прямом порядке; если же начинается с байт FE FF, следовательно порядок обратный.

На самом деле, UTF-16 не идеален, особенно если вы имеете дело с большим количеством символов ASCII. Вы не думали о том, что даже китайская веб-страница может содержать большое количество символов ASCII — все элементы и атрибуты, окружающие печатные китайские символы (и на них тоже тратится 2 байта, хотя они и умещаются в один). Возможность искать N-ый символ в строке за постоянное время заманчива, однако до сих пор существует надоевшая всем проблема с теми "запредельными" символами, которая заключается в том, что вы не можете гарантировать, что каждый символ хранится точно в двух байтах, вследствие чего поиск за постоянное время также становится невозможным (если вы только не имеете отдельный индекс по символам). Открою вам секрет: и до сих пор в мире существует огромное число ASCII текстов...

Кое-кто до вас тоже задумывался над этой проблемой и пришёл вот к такому решению:

## UTF-8

UTF-8 это кодировка Юникода с переменным числом байт. Это означает, что различные символы занимают разное число байт. Для символов ASCII (A-Z, цифр и т.п.) UTF-8 использует только 1 байт на символ (действительно, а больше и не требуется). Причём и на деле для них зарезервированы точно те самые номера, как и в ASCII; первые 128 символов (0–127) таблицы UTF-8 неотличимы от той же части ASCII. "Расширенные" символы, такие как ñ и ö занимают два байта. (bytes are not simply the Unicode code point like they would be in UTF-16; there is some serious bit-twiddling involved.) Китайские символы, такие как 中 занимают три байта. Самые редко используемые символы — четыре.

Недостатки: так как каждый символ занимает различное число байт, поиск N-го символа обладает сложностью  $O(N)$ , что означает, что время поиска пропорционально длине строки. Кроме того, bit-twiddling, применяемый для кодирования символов в байты, также увеличивает время поиска. (прим. перев. в кодировке с фиксированным числом байт на символ время поиска составляет  $O(1)$ , то есть оно не зависит от длины строки).

Преимущества: крайне эффективное кодирование наиболее часто используемых символов ASCII. Не хуже, чем хранение расширенных символов в UTF-16. Лучше, чем UTF-32 для китайских символов. Также (не хочу грузить вас математикой, так что вам придётся поверить мне на слово), в связи с самой природой bit twiddling, проблемы с порядком байт просто не существует. Документ, закодированный в UTF-8, использует один и тот же порядок байт на любом компьютере!

## Погружение

В языке программирования Python 3 все строки представляют собой последовательность Unicode символов. В Python нет такого понятия, как строка в кодировке UTF-8, или строка в кодировке CP-1251. Некорректным является вопрос: "Это строка в UTF-8?" UTF-8 — это способ закодировать символы в последовательность байт. Если Вы хотите взять строку и превратить её в последовательность байт в какой-либо кодировке, то Python 3 может помочь Вам в этом. Если же Вы желаете превратить последовательность байт в строку, то и здесь Python 3 Вам пригодится. Байты — это не символы, байты — это байты. Символы — это абстракция. А строка — это последовательность таких абстракций.

```
>>> s = '深入 Python' ①
>>> len(s) ②
9
>>> s[0] ③
'深'
>>> s + ' 3' ④
'深入 Python 3'
```

- ① Чтобы создать строку окружите её кавычками. В Python строки можно создавать как с помощью одинарных ('), так и с помощью двойных кавычек (").
- ② Стандартная функция len() возвращает длину строки, т.е. количество символов в ней. Эта же функция используется для определения длины списков, кортежей, множеств и словарей. Строка в данном случае похожа на кортеж символов.
- ③ Так же как и со списками, Вы можете получить произвольный символ из строки, зная его индекс.
- ④ Так же как и со списками, Вы можете объединять строки, используя оператор +.

## Форматирование строк

Строки можно создавать как с помощью одинарных, так и с помощью двойных кавычек.

Давайте взглянем еще раз на `humansize.py` :

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'], ①
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

```
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

'''Convert a file size to human-readable form.

②

Keyword arguments:

size -- file size in bytes

a\_kilobyte\_is\_1024\_bytes -- if True (default), use multiples of 1024  
if False, use multiples of 1000

Returns: string

'''

③

if size < 0:

raise ValueError('number must be non-negative')

④

multiple = 1024 if a\_kilobyte\_is\_1024\_bytes else 1000

for suffix in SUFFIXES[multiple]:

size /= multiple

if size < multiple:

return '{0:.1f} {1}'.format(size, suffix)

⑤

raise ValueError('number too large')

- ① 'KB', 'MB', 'GB'... - это все строки.
- ② Комментарии к функции - это тоже строка. Комментарии к функции могут быть многострочными, поэтому используются тройные кавычки в начале и в конце строки.
- ③ Эти тройные кавычки заканчивают комментарии к функции.
- ④ Здесь еще одна строка, которая передается конструктору исключения как удобочитаемый текст ошибки.
- ⑤ Здесь ... ого, это ещё что такое?

Python 3 поддерживает форматирование значений в строки. Форматирование может включать очень сложные выражение. Самое простое использование - это вставка значения в поле подстановки строки.

```
>>> username = 'mark'
```

```
>>> password = 'PapayaWhip'
```

①

```
>>> "{0}'s password is {1}".format(username, password)
```

②

```
"mark's password is PapayaWhip"
```

- ① Вы же не думаете, что мой пароль действительно PapayaWhip
- ② Здесь много чего происходит. Во первых, вызывается метод format(...) для строки. Строки - это объекты, а у объектов есть методы. Во вторых, значением всего выражения будет строка. В третьих, {0} и {1} являются полями, которые заменяются аргументами, переданными методу format()

## Составные имена полей

Предыдущий пример показал простейший способ форматирования строк: поля в строке представляют из себя целые числа. Эти числа в фигурных скобках означают порядковые номера аргументов в списке параметров метода `format()`. Это означает, что `{0}` заменяется первым аргументом (в данном случае `username`), а `{1}` заменяется на второй аргумент (`password`), &c. Вы можете иметь столько номеров полей, сколько аргументов есть у метода `format()`. А аргументов может быть сколько угодно. Но имена полей гораздо более мощный инструмент, чем может показаться на первый взгляд.

```
>>> import humansize
>>> si_suffixes = humansize.SUFFIXES[1000]    ①
>>> si_suffixes
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>> '1000{0[0]} = 1{0[1]}'.format(si_suffixes) ②
'1000KB = 1MB'
```

- ① Вместо того, чтобы вызывать какие-либо функции модуля `humansize`, Вы просто используете один из словарей, которые в этом модуле определены: список суффиксов СИ (степени 1000)
- ② Этот кусок выглядит сложным, хотя это и не так. `{0}` ссылается на первый аргумент, переданный методу `format()` (переменная `si_suffixes`). Но `si_suffixes` - это список. Поэтому `{0[0]}` ссылается на первый элемент этого списка: 'KB'. В тоже время `{0[1]}` ссылается на второй элемент того же списка: 'MB'. Все, что находится за фигурными скобками - включая 1000, знак равенства, и пробелы - остается нетронутым. В результате мы получим строку '1000KB = 1MB'.

*{0} is replaced by the 1st format() argument.  
{1} is replaced by the 2nd.*

Этот пример показывает, что при форматировании в именах полей можно получить доступ к элементам и свойствам структур данных используя (почти) Python синтаксис. Это называется "составные имена полей". Следующие составные имена полей просто работают:

- Передать список и получить доступ к элементу списка по его индексу (как в предыдущем примере);
- Передать словарь и получить доступ к значению словаря по его ключу;
- Передать модуль и получить доступ к его переменным и функциям зная их имена;

- Передать экземпляр класса и получить доступ к его свойствам и методам по их именам;
- Любая комбинация выше перечисленных.

И чтобы взорвать ваш мозг, вот пример которые использует все вышеперечисленные возможности:

```
>>> import humansize
>>> import sys
>>> '1MB = 1000{0.modules[humansize].SUFFIXES[1000][0]}'.format(sys)
'1MB = 1000KB'
```

Вот как это работает:

Модуль `sys` содержит информацию об работающем интерпретаторе Python. Так как Вы его импортировали, то можете использовать в качестве аргумента метода `format()`. То есть поле `{0}` ссылается на модуль `sys`. `sys.modules` представляет из себя словарь со всеми модулями, которые на данный момент импортированы интерпретатором Python. Ключи этого словаря - это строки с именами модулей; значения - объекты, представляющие импортированные модули. Таким образом поле `{0.modules}` ссылается на словарь импортированных модулей. `sys.modules['humansize']` - это объект, представляющий собой модуль `humansize`, который Вы только что импортировали. Таким образом составное поле `{0.modules[humansize]}` ссылается на модуль `humansize`. Заметьте, что синтаксис здесь отличается. В синтаксисе Python ключи словаря `sys.modules` являются строками, и чтобы обратиться к значениям словаря необходимо окружить кавычками имя модуля (например `'humansize'`). Вот цитата из PEP 3101: Расширенное форматирование строк: "Правила для парсинга ключей очень простые. Если он начинается с цифры, то его нужно интерпретировать как число. Иначе - это строка". `sys.modules['humansize'].SUFFIXES` - это словарь, определенный в самом начале модуля `humansize`. Поле `{0.modules[humansize].SUFFIXES}` ссылается на этот словарь.

`sys.modules['humansize'].SUFFIXES[1000]` - это список суффиксов системы СИ: `['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']`. Таким образом поле `{0.modules[humansize].SUFFIXES[1000]}` ссылается на этот список. А `sys.modules['humansize'].SUFFIXES[1000][0]` - это первый элемент списка суффиксов: `'KB'`. Таким образом окончательное составное поле `{0.modules[humansize].SUFFIXES[1000][0]}` заменяется на строку из двух символов `KB`.

## Описатели формата

Постойте! Есть еще кое-что. Давайте взглянем на еще одну странную строку из `humansize.py`:

```
if size < multiple:
    return '{0:.1f} {1}'.format(size, suffix)
```

`{1}` заменяется на второй аргумент метода `format()`, то есть на значение переменной `suffix`. Но что означает `{0:.1f}`? Здесь две вещи: `{0}`, которой Вы уже знаете и `:.1f`, о которой Вы еще не слышали. Вторая половина (двоеточие и все что после него) описывает формат, который уточняет каким образом замещающее значение должно быть отформатировано.

✎ Описатель формата позволяет Вам модифицировать замещающий текст многими полезными способами, как функция `printf()` в языке программирования C. Вы можете добавить заполнение нулями или пробелами, горизонтальное выравнивание текста, контролировать десятичную точность и даже конвертировать числа в 16-ричную систему.

Внутри замещаемого поля символ двоеточие (`:`) и все что идет после него означает описатель формата. Описатель формата `".1"` означает "округлить до десятых" (то есть показывать только один знак после запятой). Описатель формата `"f"` означает "число с фиксированной запятой" (fixed-point number) (в отличие от экспоненциального или какого-либо другого представления десятичных чисел). Таким образом если переменная `size` имеет значение 698.24 а `suffix` - 'GB', форматированная строка получится '698.2 GB', потому что число 698.24 округлено до одного знака после запятой, и к нему добавлен суффикс.

```
>>> '{0:.1f} {1}'.format(698.24, 'GB')
'698.2 GB'
```

За всеми деталями описателей формата обратитесь в раздел "Format Specification Mini-Language" официальной документации Python 3.

## Другие общие методы строк

Помимо форматирования строки позволяют делать множество полезных трюков.

```
>>> s = "Finished files are the re- ①
... sult of years of scientif-
... ic study combined with the
... experience of years."
>>> s.splitlines() ②
['Finished files are the re-',
 'sult of years of scientif-',
 'ic study combined with the',
 'experience of years.']
```



```
>>> print(s.lower())           ③
finished files are the re-
sult of years of scientif-
ic study combined with the
experience of years.
>>> s.lower().count('f')      ④
6
```

- ① В интерактивной оболочке Python Вы можете вводить многострочный текст. Такой текст начинается с тройного символа кавычек. А когда Вы нажмете ENTER интерактивная оболочка предложит Вам продолжить вводить текст. Заканчиваться многострочный текст должен также тройным символом кавычек. Когда Вы нажмете ENTER интерактивная оболочка Python выполнит команду (запишет текст в переменную `s`).
- ② Метод `splitlines()` берет многострочный текст и возвращает список строк, по одной на каждую строку оригинального текста. Заметьте, что символы перевода строки не добавляются в результирующие строки.
- ③ Метод `lower()` переводит все символы строки в нижний регистр. (Аналогично метод `upper()` переводит строку в верхний регистр.)
- ④ Метод `count()` подсчитывает количество появлений подстроки. Да, в этом предложении 6 букв "f".

Вот еще один часто встречающийся случай. Пусть у Вас есть список пар ключ-значение в виде `key1=value1&key2=value2`, и Вы хотите разделить их и получить словарь в виде `{key1: value1, key2: value2}`.

```
>>> query = 'user=pilgrim&database=master&password=PapayaWhip'
>>> a_list = query.split('&')           ①
>>> a_list
['user=pilgrim', 'database=master', 'password=PapayaWhip']
>>> a_list_of_lists = [v.split('=', 1) for v in a_list] ②
>>> a_list_of_lists
[['user', 'pilgrim'], ['database', 'master'], ['password', 'PapayaWhip']]
>>> a_dict = dict(a_list_of_lists)      ③
>>> a_dict
{'password': 'PapayaWhip', 'user': 'pilgrim', 'database': 'master'}
```

- ① Метод `split()` принимает один аргумент, разделитель, и разбивает строку по разделителям на список строк. В данном случае разделителем выступает аперсанд (&), но разделитель может быть каким угодно.
- ② Теперь у Вас есть список строк, каждая из которых состоит из ключа, знака `=` и значения. Мы можем использовать генераторы списков чтобы пройти по всему списку и разбить каждую строку в месте первого знака `=` на две строки: ключ и значение. (Теоретически значение также может

содержать знак равенства. Если просто сделаем `'key=value=foo'.split('=')`, то получим список из трех элементов `['key', 'value', 'foo']`.)

- ③ Наконец Python может превратить этот список в словарь используя функцию `dict()`.

☞ Предыдущий пример похож на грамматический разбор параметров в URL, в реальной жизни такой разбор намного сложнее. Если Вам необходимо работать с параметрами URL, то лучше использовать функцию `urllib.parse.parse_qs()`, которая умеет обрабатывать некоторые неочевидные специфические случаи.

## Разрезание строк

Как только Вы создали строку, Вы можете получить любую её часть как новую строку. Это называется разрезание строк. Разрезание работает также как срезы для списков, что вполне логично, так как строки это те же последовательности символов.

```
>>> a_string = 'My alphabet starts where your alphabet ends.'
```

```
>>> a_string[3:11] ①
```

```
'alphabet'
```

```
>>> a_string[3:-3] ②
```

```
'alphabet starts where your alphabet en'
```

```
>>> a_string[0:2] ③
```

```
'My'
```

```
>>> a_string[:18] ④
```

```
'My alphabet starts'
```

```
>>> a_string[18:] ⑤
```

```
' where your alphabet ends.'
```

- ① Вы можете получить любую часть строки, так называемый "срез", указав два индекса. Возвращаемое значение представляет из себя новую строку, содержащую все символы оригинальной строки в том же порядке, начиная с первого указанного индекса.
- ② Как и при работе со срезами списков, индексы для срезов строк могут быть отрицательными.
- ③ Индексация символов в строке начинается с нуля, поэтому `a_string[0:2]` вернет первые два элемента строки, начиная с `a_string[0]` (включительно) и заканчивая (не включительно) `a_string[2]`.
- ④ Если срез начинается с индекса 0, то этот индекс можно опустить. Таким образом `a_string[:18]` - это тоже самое, что и `a_string[0:18]`.
- ⑤ Аналогично, если последний индекс - это длина строки, то его можно не ставить. То есть `a_string[18:]` означает тоже самое, что и `a_string[18:44]`, так как в строке 44 символа. Здесь наблюдается приятная симметрия. В нашем примере строка содержит 44 символа, `a_string[:18]` возвращает первые 18 символов, а `a_string[18:]` возвращает все кроме

первых 18 символов. Фактически `a_string[:n]` всегда возвращает первые `n` символов, а `a_string[n:]` возвращает оставшуюся часть, независимо от длины строки.

## Строки против последовательности байт

Байты - это байты; символы - это абстракция. Неизменяемая последовательность Unicode символов называется строкой (string). Неизменяемая последовательность чисел-от-0-до-255 называется объект `bytes`.

```
>>> by = b'abcd\x65' ①
```

```
>>> by
b'abcde'
```

```
>>> type(by) ②
```

```
<class 'bytes'>
```

```
>>> len(by) ③
```

```
5
```

```
>>> by += b'\xff' ④
```

```
>>> by
b'abcde\xff'
```

```
>>> len(by) ⑤
```

```
6
```

```
>>> by[0] ⑥
```

```
97
```

```
>>> by[0] = 102 ⑦
```

```
Traceback (most recent call last):
```

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

```
TypeError: 'bytes' object does not support item assignment
```

- ① Чтобы создать объект `bytes` используйте синтаксис "байтовых строк" `b`". Каждый байт в байтовой строке может быть либо ASCII символом, либо закодированным шестнадцатеричным числом от `\x00` до `\xff` (0-255).
- ② Тип байтовой строки - `bytes`.
- ③ По аналогии со списками и строками, Вы можете определить длину байтовой строки с помощью встроенной функции `len()`.
- ④ По аналогии со списками и строками, Вы можете объединять байтовые строки с помощью оператора `+`. Результат будет новым объектом с типом `bytes`.
- ⑤ Объединение 5-байтового и однобайтового объекта даст в результате 6-ти байтовый объект.
- ⑥ По аналогии со списками и строками, Вы можете получить конкретный байт из байтовой строки по его индексу. Элементами обычной строки

выступают строки, а элементами байтовой строки являются целые числа. Конкретно числа от 0 до 255.

- ⑦ Байтовая строка неизменяемая. Вы не можете изменять какие-либо байты в ней. Если у Вас возникла необходимость изменить отдельные байты, то Вы можете либо использовать оператор конкатенации (+), который действует так же, как и со строками, либо конвертировать объект bytes в объект bytearray.

```
>>> by = b'abcd\x65'
>>> barr = bytearray(by) ①
>>> barr
bytearray(b'abcde')
>>> len(barr) ②
5
>>> barr[0] = 102 ③
>>> barr
bytearray(b'fbcde')
```

- ① Для конвертирования объекта bytes в изменяемый объект bytearray используйте встроенную функцию bytearray().
- ② Все методы и операторы, которые Вы использовали с объектами типа bytes, также подходят к объектам bytearray.
- ③ Единственное отличие состоит в том, что Вы можете изменить значение отдельного байта при работе с объектом bytearray. Записываемое значение должно быть целым числом от 0 до 255.

Единственное, чего Вы не можете делать, это смешивать байты и строки.

```
>>> by = b'd'
>>> s = 'abcde'
>>> by + s ①
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't concat bytes to str
>>> s.count(by) ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'bytes' object to str implicitly
>>> s.count(by.decode('ascii')) ③
1
```

- ① нельзя соединить байты и строку. Эта два разных типа данных.

- ② Вы не можете подсчитать частоту встречаемости последовательности байтов в строке, потому что в строке вообще нет байтов. Строка - это последовательность символов. Возможно Вы имеете в виду "подсчитать количество вхождений строки, полученной декодированием последовательности байт из конкретной кодировки"? Тогда это необходимо указать точно. Python 3 не будет автоматически конвертировать байты в строки или строки в байты.
- ③ По случайному совпадению это строка кода означает "подсчитать количество вхождений строки, полученной декодированием последовательности байт из конкретной кодировки".

Здесь появляется связь между строками и байтами: объект типа `bytes` имеет метод `decode()`, аргументом которого является кодировка, и который возвращает строку. В свою очередь строка имеет метод `encode()`, аргументом которого является кодировка, и который возвращает объект `bytes`. В предыдущем примере декодирование было относительно простым: последовательность байт в кодировке ASCII преобразовывалась в строку. Но этот процесс подходит для любой кодировки, которая поддерживает символы строки, даже устаревшие (не-Unicode) кодировки.

```
>>> a_string = '深入 Python'      ①
>>> len(a_string)
9
>>> by = a_string.encode('utf-8')  ②
>>> by
b'\xe6\xb7\xb1\xe5\x85\xa5 Python'
>>> len(by)
13
>>> by = a_string.encode('gb18030') ③
>>> by
b'\xc9\xee\xc8\xeb Python'
>>> len(by)
11
>>> by = a_string.encode('big5')    ④
>>> by
b'\xb2\xa4J Python'
>>> len(by)
11
>>> roundtrip = by.decode('big5')  ⑤
>>> roundtrip
'深入 Python'
>>> a_string == roundtrip
True
```

- ① Это строки. В ней 9 символов.
- ② Это объект типа `bytes`. В нем 13 байт. Это последовательность байт, полученная кодирование строки `a_string` в кодировке UTF-8.
- ③ Это объект типа `bytes`. В нем 11 байт. Это последовательность байт, полученная кодирование строки `a_string` в кодировке GB18030.
- ④ Это объект типа `bytes`. В нем 11 байт. Это последовательность байт, полученная кодирование строки `a_string` в кодировке Big5.
- ⑤ Это строка. Она состоит из девяти символов. Она представляет из себя последовательность символов, которые Вы получите после декодирования `bu` используя алгоритм кодировки Big5. Полученная строка совпадает с первоначальной.

## P.S. Кодировка в исходном коде Python

Python 3 предполагает, что ваш исходный код — т.е. каждый файл `.py` — записан в кодировке UTF-8.

В Python 2, кодировкой по умолчанию для файлов `.py` была кодировка ASCII. В Python 3 кодировка по умолчанию — UTF-8.

Если вы желаете использовать другую кодировку в вашем коде, вы можете разместить объявление кодировки на первой строке каждого файла. Например, для кодировки windows-1252 объявление выглядит следующим образом:

```
# -*- coding: windows-1252 -*-
```

Объявление кодировки также может располагаться на второй строке файла, если первой строкой является путь к интерпретатору Python.

```
#!/usr/bin/python3
# -*- coding: windows-1252 -*-
```

За дополнительной информацией обращайтесь к PEP 263: Defining Python Source Code Encodings.

## Материалы для дальнейшего чтения

- On Unicode in Python:
- Python Unicode HOWTO
- What's New In Python 3: Text vs. Data Instead Of Unicode vs. 8-bit
- PEP 261 explains how Python handles astral characters outside of the Basic Multilingual Plane (i.e. characters whose ordinal value is greater than 65535)
- On Unicode in general:
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and \* Character Sets (No Excuses!)
- On the Goodness of Unicode

- On Character Strings
  - Characters vs. Bytes
  - On character encoding in other formats:
    - Character encoding in XML
    - Character encoding in HTML
  - On strings and string formatting:
    - string — Common string operations
    - Format String Syntax
    - Format Specification Mini-Language
    - PEP 3101: Advanced String Formatting
-



# Регулярные выражения

“Некоторые люди, во время решения одной проблемы думают: «Я знаю, я буду использовать регулярные выражения». Теперь у них две проблемы...” — Jamie Zawinski

## Погружение

Каждый новый язык программирования имеет встроенные функции для работы со строками. В Python, строки имеют методы для поиска и замены: `index()`, `find()`, `split()`, `count()`, `replace()` и т.д. Но эти методы ограничены для простейших случаев. Например метод `index()` ищет простую жёстко заданную часть строки и поиск всегда регистрозависимый. Чтобы выполнить регистронезависимый поиск по строке `s`, вы должны вызвать `s.lower()` или `s.upper()` для того чтобы быть уверенным что строка имеет соответствующий регистр для поиска. Методы `replace()` и `split()` имеют те же ограничения.

Если ваша задача может быть решена при помощи этих методов, лучше использовать их. Они простые и быстрые, легко читаемые, много может быть сказано о быстром, простом и удобочитаемом коде. Но если вы обнаружите что вы используете большое количество строковых функций с условиями `if` для обработки специальных случаев, или используете множество последовательных вызовов `split()` и `join()` чтобы нарезать на кусочки ваши строки, значит вы нуждаетесь в регулярных выражениях.

Регулярные выражения это мощный и (по большей части) стандартизированный способ для поиска, замены и парсинга текста при помощи комплексных шаблонов. Хотя синтаксис регулярных выражений довольно сложный и выглядит непохожим на нормальный код (прим. пер. «смахивает на perl»), конечный результат часто более удобочитаемый чем набор из последовательных функций для строк. Существует даже способ поместить комментарии внутрь регулярных выражений, таким образом вы можете включить небольшую документацию в регулярное выражение.



Если вы пользовались регулярными выражениями в других языках (таких как Perl, JavaScript, или PHP), синтаксис Python-а будет для вас достаточно привычным. Прочитайте обзор модуля `re` для того чтобы узнать о доступных функциях и их аргументах.

\*\*

## Учебный пример: Адрес Улицы

Эта серия примеров основана на реальных проблемах, которые появились в моей работе несколько лет назад, когда мне пришлось обработать и стандартизировать адреса улиц, экспортированных из устаревшей системы до того, как произвести импорт в новую систему. (Обратите внимание: это не придуманный пример, им всё ещё можно пользоваться). Этот пример показывает как я подошёл к проблеме:

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.') ①
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.') ②
'100 NORTH BRD. RD.'
>>> □UNIQae610d7ca506639d-nowiki-00000003-QINU□ ③
'100 NORTH BROAD RD.'
>>> import re ④
>>> re.sub('ROAD$', 'RD.', s) ⑤
'100 NORTH BROAD RD.'
```

- ① Моя задача стандартизировать адрес улицы, например 'ROAD' всегда выражается сокращением 'RD.'. На первый взгляд мне показалось, что это достаточно просто, и я могу использовать метод `replace()`. В конце концов, все данные уже в верхнем регистре и несовпадение регистра не составит проблемы. Строка поиска 'ROAD' являлась константой и обманчиво простой пример `s.replace()` вероятно работает.
- ② Жизнь же, напротив, полна противоречивых примеров, и я быстро обнаружил один из них. Проблема заключалась в том что 'ROAD' появилась в адресе дважды, один раз как 'ROAD', а во второй как часть названия улицы 'BROAD'. Метод `replace()` обнаруживал 2 вхождения и слепо заменял оба, разрушая таким образом правильный адрес.
- ③ Чтобы решить эту проблему вхождения более одной подстроки 'ROAD', вам необходимо прибегнуть к следующему: искать и заменять 'ROAD' в последних четырёх символах адреса (`s[-4:]`), оставляя строку отдельно (`s[:-4]`). Как вы могли заметить, это уже становится громоздким. К примеру, шаблон зависит от длины заменяемой строки. (Если вы

заменяли 'STREET' на 'ST.', вам придется использовать `s[:-6]` и `s[:-6].replace(...)`. Не хотели бы вы вернуться к этому коду через полгода для отладки? Я не хотел бы.

- ④ Пришло время перейти к регулярным выражениям. В Python все функции, связанные с регулярными выражениями содержится в модуле `re`.
- ⑤ Взглянем на первый параметр: 'ROAD\$'. Это простое регулярное выражение которое находит 'ROAD' только в конце строки. Знак \$ означает «конец строки». (Также существует символ ^, означающий «начало строки».) Используя функцию `re.sub()` вы ищите в строке `s` регулярное выражение 'ROAD\$' и заменяете на 'RD.'. Оно совпадает с 'ROAD' в конце строки `s`, но *не* совпадает с 'ROAD', являющимся частью названия 'BROAD', так как оно находится в середине строки `s`.

Продолжая историю про обработку адресов, я скоро обнаружил, что предыдущий пример совпадения 'ROAD' на конце адреса был недостаточно хорош, так как не все адреса включали в себя определение улицы. Некоторые адреса просто оканчивались названием улицы. Я избегал этого в большинстве случаев, но если название улицы было 'BROAD', тогда регулярное выражение совпадало с 'ROAD' на конце строки 'BROAD', чего я совершенно не хотел.

```
>>> s = '100 BROAD'
>>> re.sub('ROAD$', 'RD.', s)
'100 BRD.'
>>> re.sub('\bROAD$', 'RD.', s) ①
'100 BROAD'
>>> re.sub(r'\bROAD$', 'RD.', s) ②
'100 BROAD'
>>> s = '100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD$', 'RD.', s) ③
'100 BROAD ROAD APT. 3'
>>> re.sub(r'\bROAD\b', 'RD.', s) ④
'100 BROAD RD. APT 3'
```

- ① В *действительности* я хотел совпадения с 'ROAD' когда оно на конце строки *и* является самостоятельным словом (а не частью большего). Чтобы описать это в регулярном выражении необходимо использовать '`\b`', что означает «слово должно оказаться прямо тут.» В Python это сложно, так как '`\`' знак в строке должен быть экранирован. Иногда это называют как «бедствие бэкслэша» и это одна из причин почему регулярные выражения проще в Perl чем в Python. Однако недостаток Perl в том что регулярные выражения смешиваются с другим

синтаксисом, если у вас ошибка, достаточно сложно определить где она, в синтаксисе или в регулярном выражении.

- ② Чтобы обойти проблему «бедствие бэкслэша» вы можете использовать то, что называется *неформатированная строка* (*raw string*), путём применения префикса строки при помощи символа 'r'. Это скажет Python-у что ничего в этой строке не должно быть экранировано; '\t' это табулятор, но r'\t' это символ бэкслэша '\', а следом за ним буква 't'. Я рекомендую всегда использовать неформатированную строку, когда вы имеете дело с регулярными выражениями; с другой стороны всё становится достаточно путанным (несмотря на то что наше регулярное выражения уже достаточно запутано).
- ③ \*вздых\* К неудаче я скоро обнаружил больше причин противоречащих моей логике. В этом случае адрес улицы содержал в себе цельное отдельное слово 'ROAD' и оно не было на конце строки, так как адрес содержал номер квартиры после определения улицы. Так как слово 'ROAD' не находится в конце строки, регулярное выражение `re.sub()` его пропускало и мы получали на выходе ту же строку что и на входе, а это то чего вы не хотите.
- ④ Чтобы решить эту проблему я удалил символ '\$' и добавил ещё один '\b'. Теперь регулярное выражение совпадало с 'ROAD' если оно являлось цельным словом в любой части строки, на конце, в середине и в начале.

\*  
\*\*

## Учебный пример: Римские цифры

Скорее всего вы видели римские цифры, даже если вы в них не разбираетесь. Вы могли видеть их на копирайтах старых фильмов и ТВ-шоу («Copyright MCMXLVI» вместо «Copyright 1946»), или на стенах в библиотеках университетов («учреждено MDCCCLXXXVIII» вместо «учреждено 1888»). Вы могли видеть их в структуре библиографических ссылок. Эта система отображения цифр относится к древней Римской империи (отсюда и название).

В римских цифрах семь символов, которые повторяются в различных комбинациях для отображения цифр.

- I = 1
- V = 5
- X = 10
- L = 50
- C = 100
- D = 500

- $M = 1000$

Нижеследующие правила позволяют конструировать римские цифры:

- Иногда символы складываются. I это 1, II это 2, и III это 3. VI это 6 (посимвольно, «5 и 1»), VII это 7, и VIII это 8.
- Десятичные символы (I, X, C, и M) могут быть повторены до 3 раз. Для образования 4 вам необходимо отнять от следующего высшего символа пятёрки. Нельзя писать 4 как IIII; вместо этого, она записывается как IV («на 1 меньше 5»). 40 записывается как XL («на 10 меньше 50»), 41 как XLI, 42 как XLII, 43 как XLIII, и 44 как XLIV («на 10 меньше 50, и на 1 меньше 5»).
- Иногда символы... обратны сложению. Разместив определённые символы до других, вы вычитаете их от конечного значения. Например 9, вам необходимо отнять от следующего высшего символа десять: 8 это VIII, но 9 это IX («на 1 меньше 10»), не VIIII (так как символ I не может быть повторён 4 раза). 90 это XC, 900 это CM.
- Пятёрки не могут повторяться. 10 всегда отображается как X, никогда как VV. 100 всегда C, никогда LL.
- Римские цифры читаются слева направо, поэтому положение символа имеет большое значение. DC это 600; CD это совершенно другая цифра (400, «на 100 меньше 500»). CI это 101; IC это даже не является допустимым Римским числом (так как вы не можете вычитать 1 прямо из 100; вам необходимо записать это как XCIX, «на 10 меньше 100, и на 1 меньше 10»).

## Проверка на тысячи

Что необходимо сделать чтобы проверить что произвольная строка является допустимым римским числом? Давайте будем брать по одному символу за один раз. Так как римские числа всегда записываются от высшего к низшему, начнём с высшего: с тысячной позиции. Для чисел от 1000 и выше, используются символы M.

```
>>> import re
>>> pattern = '^M?M?M?$' ①
>>> re.search(pattern, 'M') ②
❑UNIQae610d7ca506639d-nowiki-00000039-QINU❑
>>> re.search(pattern, 'MM') ③
<_sre.SRE_Match object at 0106C290>
>>> re.search(pattern, 'MMM') ④
<_sre.SRE_Match object at 0106AA38>
>>> re.search(pattern, 'MMMM') ⑤
```

```
>>> re.search(pattern, "") ⑥
<_sre.SRE_Match object at 0106F4A8>
```

- ① Этот патерн состоит из трёх частей. `^` совпадает с началом строки. Если его не указать, патерн будет совпадать с `M` без учёта положения в строке, а это не то что нам надо. Вы должны быть уверены что символы `M` если присутствуют, то находятся в начале строки. `M?` Опционально совпадает с одним символом `M`. Так это повторяется три раза то патерн совпадёт от нуля до трёх раз с символом `M` в строке. И символ `$` совпадёт с концом строки. Когда комбинируется с символом `^` в начале, это означает что патерн должен совпасть с полной строкой, без других символов до и после символов `M`.
- ② Сущность модуля `re` это функция `search()`, которая использует патерн регулярного выражения (*pattern*) и строку ('`M`') и ищет совпадения в соответствии регулярному выражению. Если совпадение обнаружено, `search()` возвращает объект который имеет различные методы описания совпадения; если совпадения не обнаружено, `search()` возвращает `None`, в Python значение нуля (*null*). Всё о чём мы заботимся в данный момент, совпадёт ли патерн, это можно сказать глянув на значение возвращаемое функцией `search()`. '`M`' совпадает с этим регулярным выражением, так как первое опциональное `M` совпадает, а второе опциональное `M` и третье игнорируется.
- ③ '`MM`' совпадает так как первое и второе опциональное `M` совпадает а третье игнорируется
- ④ '`MMM`' совпадает полностью, так как все три символа `M` совпадают
- ⑤ '`MMMM`' не совпадает. Все три `M` совпадают, но регулярное выражение настаивает на конце строки, (так как требует символ `$`), а строка ещё не кончилась (из за четвёртого `M`). Поэтому `search()` возвращает `None`.
- ⑥ Занимательно то, что пустая строка также совпадает с регулярным выражением, так как все символы `M` опциональны.

## Проверка на сотни

### ? делает патерн необязательным

Расположение сотен более сложное чем тысяч, так как существует несколько взаимно исключающих путей записи и зависит от значения.

- 100 = C

- 200 = CC
- 300 = CCC
- 400 = CD
- 500 = D
- 600 = DC
- 700 = DCC
- 800 = DCCC
- 900 = CM

Таким образом есть четыре возможных паттерна:

- CM
- CD
- От нуля до трёх символов C (ноль если место сотен пусто)
- D, от последующих нулей до трёх символов C

Два последних паттерна комбинированные:

- опциональное D, за ним от нуля до трёх символов C

Этот пример показывает как проверить позицию сотни в римском числе.

```
>>> import re
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)$' ①
>>> re.search(pattern, 'MCM') ②
□UNIQae610d7ca506639d-nowiki-00000040-QINU□
>>> re.search(pattern, 'MD') ③
□UNIQae610d7ca506639d-nowiki-00000041-QINU□
>>> re.search(pattern, 'MMMCCC') ④
□UNIQae610d7ca506639d-nowiki-00000042-QINU□
>>> re.search(pattern, 'MCMC') ⑤
>>> re.search(pattern, '') ⑥
□UNIQae610d7ca506639d-nowiki-00000043-QINU□
```

- ① Этот паттерн стартует также как и предыдущий, проверяя сначала строки (^), потом тысячи (M?M?M?). Следом идёт новая часть в скобках, которая описывает три взаимоисключающих паттерна разделённых вертикальной линией: CM, CD и D?C?C?C? (который является опциональным D и следующими за ним от нуля до трёх опциональных символов C). Парсер регулярного выражения проверяет каждый из этих паттернов в последовательности от левого к правому, выбирая первый подходящий и игнорируя последующие.



- ② 'MCM' совпадает так как первый M совпадает, второй и третий символ M игнорируется, символы CM совпадают (и CD и D?C?C?C? патерны после этого не анализируются). MCM это римское представление числа 1900.
- ③ 'MD' совпадает так как первый M совпадает, второй и третий символ M игнорируется, и патерн D?C?C?C? Совпадает с D (три символа C опциональны и игнорируются). MD это римское представление числа 1500.
- ④ 'MMMCCC' совпадает так как первый M совпадает, и патерн D?C?C?C? совпадает с CCC (символ D опциональный и игнорируется). MMMCCC это римское представление числа 3300.
- ⑤ 'MCMC' не совпадает. Первый символ M совпадает, второй и третий символ M игнорируется, также совпадает CM, но патерн \$ не совпадает так как вы ещё не в конце строки (вы ещё имеете не совпадающий символ C). Символ C не совпадает как часть паттерна D?C?C?C?, так как исключаящий патерн CM уже совпал.
- ⑥ Занимательно то, что пустая строка всё ещё совпадает с регулярным выражением, так как все символы M опциональны и игнорируются и пустая строка совпадает с паттерном D?C?C?C? где все символы опциональны и игнорируются.

Опаньки! Вы заметили как быстро регулярные выражения становятся отвратительными? И пока что мы обработали только позиции тысяч и сотен в римском представлении чисел. Но если последуете далее, вы обнаружите что десятки и единицы описать будет легче, так как они имеют такой же патерн. Тем временем давайте рассмотрим другой путь описать этот патерн.

\*\*

## Использование синтаксиса {n, m}

*модификатор {1,4} совпадает с 1 до 4  
вхождением паттерна*

В предыдущей секции мы имели дело с паттерном где одинаковый символ может повториться до трёх раз. Существует другой путь записать это регулярное выражение, которое многие люди найдут более читаемым. Для начала взглянем на метод который мы уже использовали в предыдущем примере.

```
>>> import re
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'M') ①
□UNIQae610d7ca506639d-nowiki-00000045-QINU□
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MM') ②
□UNIQae610d7ca506639d-nowiki-00000046-QINU□
>>> pattern = '^M?M?M?$'
>>> re.search(pattern, 'MMM') ③
□UNIQae610d7ca506639d-nowiki-00000047-QINU□
>>> re.search(pattern, 'MMMM') ④
>>>
```

- ① Тут патерн совпадает с началом строки и первым опциональным M, но не со вторым и третьим (но это нормально так как они опциональны), а также с концом строки.
- ② Тут патерн совпадает с началом строки, с первым и вторым опциональным символом M, но не с третьим (это нормально так как он опционален) и с концом строки.
- ③ Тут патерн совпадает с началом строки и со всеми тремя опциональными символами M и также с концом строки.
- ④ Тут патерн совпадает с началом строки и со всеми тремя опциональными символами M, но не совпадает с концом строки (так как присутствует ещё одно M), таким образом патерн не совпадает и возвращает **None**.

```
>>> pattern = '^M{0,3}$' ①
>>> re.search(pattern, 'M') ②
□UNIQae610d7ca506639d-nowiki-00000049-QINU□
>>> re.search(pattern, 'MM') ③
<_sre.SRE_Match object at 0x008EE090>
>>> re.search(pattern, 'MMM') ④
<_sre.SRE_Match object at 0x008EEDA8>
>>> re.search(pattern, 'MMMM') ⑤
>>>
```

- ① Этот патерн говорит: «Совпасть с началом строки, потом с от нуля до трёх символов M находящимися где угодно, потом с концом строки». Символы 0 и 3 могут быть любыми цифрами, если вам необходимо совпадение с 1 и более символами M, необходимо записать M{1,3}.

- ② Тут патерн совпадает с началом строки, потом с одним из возможных трёх символов M, потом с концом строки.
- ③ Тут патерн совпадает с началом строки, потом с двумя из возможных трёх символов M, потом с концом строки.
- ④ Тут патерн совпадает с началом строки, потом с тремя из возможных трёх символов M, потом с концом строки.
- ⑤ Тут патерн совпадает с началом строки, потом с двумя из возможных трёх символов M, но *не совпадает* с концом строки.

Регулярное выражение позволяет до трёх символов M до конца строки, но у вас четыре, и патерн возвращает **None**.

### Проверка на десятки и единицы

Теперь давайте расширим регулярное выражение чтобы включить десятки и единицы. Этот пример показывает проверку на десятки.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?)(XC|XL|L?X?X?X?)$'
>>> re.search(pattern, 'MCMXL') ①
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCML') ②
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'MCMLX') ③
❑UNIQae610d7ca506639d-nowiki-0000004F-QINU❑
>>> re.search(pattern, 'MCMLXXX') ④
❑UNIQae610d7ca506639d-nowiki-00000050-QINU❑
>>> re.search(pattern, 'MCMLXXXX') ⑤
>>>
```

- ① Тут патерн совпадает с началом строки, потом с первым опциональным символом M, потом CM, потом XL, потом с концом строки. Вспомните что синтаксис (A|B|C) означает «совпасть только с одним из символов A, B или C» У нас совпадает XL, и мы игнорируем XC и L?X?X?X?, а после этого переходим к концу строки. MCMXL это римское представление числа 1940.
- ② Тут патерн совпадает с началом строки, потом с первым опциональным символом M, потом CM, потом с L?X?X?X?. Из L?X?X?X? Совпадает L и пропускает три опциональных символа X. После этого переходит к концу строки. MCML это римское представление числа 1950.

- ③ Тут патерн совпадает с началом строки, потом с первым опциональным символом М, потом СМ, потом с опциональным L и первым опциональным X, пропуская второй и третий опциональные символы X, после этого переходит к концу строки. MCMLX это римское представление числа 1960.
- ④ Тут патерн совпадает с началом строки, потом с первым опциональным символом М, потом СМ, потом с опциональным L и всеми тремя опциональными символами X, после этого переходит к концу строки. MCMLXXX это римское представление числа 1980.
- ⑤ Тут патерн совпадает с началом строки, потом с первым опциональным символом М, потом СМ, потом с опциональным L и всеми тремя опциональными символами X, после этого *не совпадает* с концом строки, так как есть ещё один символ X, таким образом патерн не срабатывает и возвращает **None**. MCMLXXXX это недопустимое римское число.

(A|B) совпадает либо с A либо с B.

Для описания единиц подходит тот же патерн. Я уменьшу детализацию и покажу конечный результат.

```
>>> pattern = '^M?M?M?(CM|CD|D?C?C?C?)(XC|XL|L?X?X?X?)(IX|IV|V?I?I?I?)$'
```

Итак как это будет выглядеть используя альтернативный синтаксис  $\{n,m\}$ ? Этот пример показывает новый синтаксис.

```
>>> pattern = '^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$'
```

```
>>> re.search(pattern, 'MDLV') ①
```

```
□UNIQae610d7ca506639d-nowiki-00000053-QINU□
```

```
>>> re.search(pattern, 'MMDCLXVI') ②
```

```
□UNIQae610d7ca506639d-nowiki-00000054-QINU□
```

```
>>> re.search(pattern, 'MMMDCCLXXXVIII') ③
```

```
□UNIQae610d7ca506639d-nowiki-00000055-QINU□
```

```
>>> re.search(pattern, 'I') ④
```

```
□UNIQae610d7ca506639d-nowiki-00000056-QINU□
```

- ① Тут патерн совпадает с началом строки, потом с одним из трёх возможных символов М, потом  $D?C\{0,3\}$ . Из них совпадает только опциональное D и ни один из опциональных C. Далее совпадает опциональное L из  $L?X\{0,3\}$  и ни один из трёх опциональных X. После совпадает с V из  $V?I\{0,3\}$  и ни с одним из трёх опциональных I и наконец с концом строки. MDLV это римское представление числа 1555.

- ② Тут патерн совпадает с началом строки, потом с двумя из трёх возможных символов M, потом D и один опциональный C из  $D?C\{0,3\}$ . Потом  $L?X\{0,3\}$  с L и один из трёх возможных X, потом  $V?I\{0,3\}$  с V и одним из трёх I, потом с концом строки. MMDCLXVI это римское представление числа 2666.
- ③ Тут патерн совпадает с началом строки, потом с тремя из трёх M, потом D и C из  $D?C\{0,3\}$ , потом  $L?X\{0,3\}$  с L и три из трёх X, потом  $V?I\{0,3\}$  с V и тремя из трёх I, потом конец строки. MMMDCCCLXXXVIIII это римское представление числа 3888, и это максимально длинное римское число которое можно записать без расширенного синтаксиса.
- ④ Смотрите внимательно. (Я чувствую себя магом, «Смотрите внимательно детки, сейчас кролик вылезет из моей шляпы ;)» Тут совпадает начало строки, ни один из трёх M, потом  $D?C\{0,3\}$  пропускает опциональный D и три опциональных C, потом  $L?X\{0,3\}$  пропускает опциональный L и три опциональных X, потом  $V?I\{0,3\}$  пропускает опциональный V и один из трёх опциональных I. Потом конец строки. Стоп, фуф.

Если вы следовали всему и поняли с первой попытки, значит у вас получается лучше чем у меня. Теперь представьте что вы пытаетесь разобраться в чьих то регулярных выражениях в важной функции в рамках огромной программы. Или например представьте что вы возвращаетесь к собственной программе через несколько месяцев. Я делал это и это не слишком приятное зрелище.

А сейчас давайте исследуем альтернативный синтаксис, который позволит легче выполнять поддержку ваших выражений.

\*\*

## Подробные регулярные выражения

До сих пор вы имели дело с тем что я называю «компактными» регулярными выражениями. Как вы могли заметить они трудны для прочтения, даже если вы понимаете что они делают. Нет гарантии что вы сможете разобраться в них спустя шесть месяцев. Что вам действительно необходимо так это вложенная документация

Python позволяет вам сделать это при помощи *подробных регулярных выражений*. Подробные регулярные выражения отличаются от компактных двумя способами:

- Пустые строки игнорируются, пробелы, табы и возвраты каретки не совпадают соответственно. Они вообще не совпадают. (Если вы хотите совпадения с пробелом в подробном регулярном выражении, вам необходимо поставить бэкслэш перед ним.)
- Комментарии игнорируются. Комментарий в подробном регулярном выражении такой же как и комментарий в коде Python: он начинается с символа `#` и действует до конца строки. В этом случае это комментарий это комментарий внутри многострочной строки, но он работает также как и простой.

Пример сделает это более понятным. Давайте перепроверим компактное регулярное выражение с которым мы работали и создадим подробное регулярное выражение. Этот пример показан ниже.

```
>>> pattern = '''
^                # начало строки
M{0,3}           # тысячи - 0 до 3 M
(CM|CD|D?C{0,3}) # сотни — 900 (CM), 400 (CD), 0-300 (0 до 3 C),
                  # или 500-800 (D, с последующими от 0 до 3 C)
(XC|XL|L?X{0,3}) # десятки - 90 (XC), 40 (XL), 0-30 (0 до 3 X),
                  # или 50-80 (L, с последующими от 0 до 3 X)
(IX|IV|V?I{0,3}) # единицы - 9 (IX), 4 (IV), 0-3 (0 до 3 I),
                  # или 5-8 (V, с последующими от 0 до 3 I)
$                # конец строки
'''
```

```
>>> re.search(pattern, 'M', re.VERBOSE) ①
□UNIQae610d7ca506639d-nowiki-00000058-QINU□
>>> re.search(pattern, 'MCMLXXXIX', re.VERBOSE) ②
□UNIQae610d7ca506639d-nowiki-00000059-QINU□
>>> re.search(pattern, 'MMMDCCLXXXVIII', re.VERBOSE) ③
<_sre.SRE_Match object at 0x008EEB48>
>>> re.search(pattern, 'M') ④
```

- ① Главное что надо запомнить, это то что необходимо добавлять экстра аргументы для работы с ними: `re.VERBOSE` это константа определённая в модуле `re` которая служит сигналом что патерн должен быть использован как подробное регулярное выражение. Как вы можете видеть, этот патерн содержит большое количество пустых строк. (и все они игнорируются), а также несколько комментариев (которые игнорируются также). Если мы игнорируем комментарии и пустые строки, то получается то же самое регулярное выражение что и в предыдущем примере, но в гораздо более читабельном виде.
- ② Здесь совпадает начало строки, потом одно и трёх возможных `M`, потом `CM`, потом `L` и три из возможных `X`, потом `IX`, потом конец строки.

- ③ Здесь совпадает начало строки, потом три из трёх возможных M, потом D и три из возможных трёх C, потом L и три из трёх возможных X, потом V и три из трёх возможных I, потом конец строки.
- ④ Тут не совпадает. Почему? Так как отсутствует флаг `re.VERBOSE` и функция `re.search` рассматривает патерн как компактное регулярное выражение, с значащими пробелами и символами `#`. Python не может автоматически определить является ли регулярное выражение подробным или нет. Python рассматривает каждое регулярное выражение как компактное до тех пор пока вы не укажете что оно подробное.

\*  
\*\*

## Учебный пример: Обработка телефонных номеров

*\d совпадает с любыми цифрами (0–9).  
\D совпадает со всем кроме цифр*

До сих пор вы были сконцентрированы на полных паттернах. Совпадает паттерн или не совпадает, но регулярные выражения могут быть гораздо мощнее этого. Когда регулярное выражение совпадает с чем либо, вы можете получить специально выделенную часть совпадения. Вы можете узнать что совпало и где.

Этот пример появился из ещё одной реальной проблемы которые я испытывал на предыдущей работе. Проблема была в обработке американских телефонных номеров. Клиент хочет ввести телефонный номер в простое поле (без разделителей), но потом также хочет сохранить индекс, магистраль, номер и опционально добавочную информацию в базе данных компании. Я поискал по интернет и нашёл много примеров регулярного выражения которое должно делать это, но к сожалению ни одно из решений не подошло.

Вот телефонные номера которые я должен был обработать:

- 800-555-1212
- 800 555 1212
- 800.555.1212
- (800) 555-1212
- 1-800-555-1212
- 800-555-1212-1234
- 800-555-1212x1234
- 800-555-1212 ext. 1234



- `work 1-(800) 555.1212 #1234`

Достаточно вариантов в любом из этих примеров. Мне необходимо было знать код 800, магистраль 555 и остаток номера 1212. Для тех что с расширениями, мне необходимо было знать что расширение 1234

Давайте займёмся разработкой решения для обработки телефонного номера. Этот пример показывает первый шаг:

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$') ①
>>> phonePattern.search('800-555-1212').groups()           ②
('800', '555', '1212')
>>> phonePattern.search('800-555-1212-1234')               ③
>>> phonePattern.search('800-555-1212-1234').groups()      ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```

- ① Всегда читайте регулярное выражение слева направо. Выражение совпадает с началом строки и потом с `(\d{3})`. Что такое `\d{3}`? Итак, `\d` значит «любая цифра» (от 0 до 9). `{3}` значит «совпадение с конкретно тремя цифрами»; это вариации на тему `{n, m}` синтаксиса который вы наблюдали ранее. Если заключить это выражение в круглые скобки, то это значит «совпасть должно точно три цифры и потом *запомнить их как группу которую я запрошу позже*». Потом выражение должно совпасть с дефисом. Потом совпасть с другой группой из трёх цифр, потом опять дефис. Потом ещё одна группа из четырёх цифр. И в конце совпадение с концом строки.
- ② Чтобы получить доступ к группам которые запомнил обработчик регулярного выражения, используйте метод `groups()` на объекте который возвращает метод `search()`. Он должен вернуть кортеж такого количества групп, которое было определено в регулярном выражении. В нашем случае определены три группы, одна с тремя цифрами, другая с тремя цифрами и третья с четырьмя цифрами.
- ③ Это регулярное выражение не окончательный ответ, так как оно не обрабатывает расширение после телефонного номера. Для этого вы должны расширить регулярное выражение.
- ④ Вот почему вы не должны использовать «цепочку» из методов `search()` и `groups()` в продакшн коде. Если метод `search()` не вернёт совпадения, то он вернёт `None`, это не стандартный объект регулярного

выражения. Вызов `None.groups()` генерирует очевидное исключение: `None` не имеет метода `groups()`. (Конечно же это немного менее очевидно, когда вы получаете это исключение из глубин вашего кода. Да, сейчас это говорит мой опыт.)

```
>>> phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})-(\d+)$') ①
>>> phonePattern.search('800-555-1212-1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800 555 1212 1234') ③
>>>
>>> phonePattern.search('800-555-1212') ④
>>>
```

- ① Это регулярное выражение почти идентично предыдущему. Так же как и до этого оно совпадает с началом строки, потом с запомненной группой из трёх цифр, потом дефис, потом запомненная группа из трёх цифр, потом дефис, потом запомненная группа из четырёх цифр. Что же нового? Это совпадение с другим дефисом и запоминаемая группа из одной и более цифры.
- ② Метод `groups()` теперь возвращает кортеж из четырёх элементов, а регулярное выражение теперь запоминает четыре группы.
- ③ К неудаче это регулярное выражение не является финальным ответом, так как оно подразумевает что различные части номера разделены дефисом. Что случится если они будут разделены пробелами, запятыми или точками?
- ④ Вам необходимо более общее решение для совпадения с различными типами разделителей.

Опаньки! То что делает это регулярное выражение это ещё не совсем то что вы хотите. В действительности это даже шаг назад, так как вы не можете обрабатывать телефонные номера без расширений. Это совершенно не то что вы хотели; если расширение есть, вы бы хотели знать какое оно, но если его нет, вы до сих пор хотите знать различные части телефонного номера.

Следующий пример показывает как регулярное выражение обрабатывает разделители между различными частями телефонного номера.

```
>>> phonePattern = re.compile(r'^(\d{3})\D+(\d{3})\D+(\d{4})\D+(\d+)$') ①
>>> phonePattern.search('800 555 1212 1234').groups() ②
```

```

('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212-1234').groups() ③
('800', '555', '1212', '1234')
>>> phonePattern.search('80055512121234') ④
>>>
>>> phonePattern.search('800-555-1212') ⑤
>>>

```

- ① Держите свою шляпу. У вас совпадает начало строки, потом группа из трёх цифр, потом `\D+`. Что это за чертовщина? Ок, `\D` совпадает с любым символом кроме цифр и также «+» означает «1 или более». Итак `\D+` означает один или более символом не являющихся цифрами. Это то что вы используете вместо символа дефиса «-» чтобы совпадало с любыми разделителями.
- ② Использование `\D+` вместо «-» значит, что теперь регулярное выражение совпадает с телефонным номером разделённым пробелами вместо дефисов.
- ③ Конечно телефонные номера разделенные дефисами тоже срабатывают.
- ④ К неудаче это ещё не окончательный ответ, так как он подразумевает наличие разделителя. Что если номер введён без всяких разделителей?
- ⑤ ОпцаЁ! И до сих пор не решена проблема расширения. Теперь у вас две проблемы, но вы можете справиться с ними используя ту же технику.

Следующий пример показывает регулярное выражение для обработки телефонных номеров без разделителей.

```

>>> phonePattern = re.compile(r'^(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('80055512121234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800.555.1212 x1234').groups() ③
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ④
('800', '555', '1212', '')
>>> phonePattern.search('(800)5551212 x1234') ⑤
>>>

```

- ① Только одно изменение, замена «+» на «\*». Вместо `\D+` между частями номера, теперь используется `\D*`. Помните что «+» означает «1 или более»? Ок, «\*» означает «ноль или более». Итак теперь вы можете обработать номер даже если он не содержит разделителей.
- ② Подумать только, это действительно работает. Почему? У вас совпадает начало строки, потом запоминается группа из трёх цифр (800), потом ноль или более нецифровых символов, потом запоминается группа из трёх цифр (555), потом ноль или более нецифровых символов, потом запоминается группа из четырёх цифр (1212), потом ноль или более нецифровых символов, потом запоминается группа из произвольного количества цифр (1234), потом конец строки.
- ③ Различные вариации также работают: точки вместо дефисов, и также пробелы или «x» перед расширением.
- ④ Наконец вы решили давнюю проблему: расширение снова опционально. Если не найдено расширения метод `groups()` всё ещё возвращает четыре элемента, но четвёртый элемент просто пустая строка.
- ⑤ Я ненавижу быть вестником плохих новостей, но вы ещё не закончили. Что же тут за проблема? Существуют дополнительные символы до «агеа» кода, но регулярное выражение думает что код города это первое что находится в начале строки. Нет проблем, вы можете использовать ту же технику «ноль или более нецифровых символов» чтобы пропустить начальные символы до кода города.

Следующий пример показывает как работать с символами до телефонного номера.

```
>>> phonePattern = re.compile(r'^\D*(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('(800)5551212 ext. 1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212').groups() ③
('800', '555', '1212', '')
>>> phonePattern.search('work 1-(800) 555.1212 #1234') ④
>>>
```

- ① Это то же самое что и в предыдущем примере, кроме `\D*`, ноль или более нецифровых символов, до первой запомненной группы (код города). Заметьте что вы не запоминаете те нецифровые символы до

кода города (они не в скобках). Если вы обнаружите их, вы просто пропустите их и запомните код города.

- ② Вы можете успешно обработать телефонный номер, даже со скобками до кода города. (Правая скобка также обрабатывается; как нецифровой символ и совпадает с `\D*` после первой запоминаемой группы.)
- ③ Простая проверка не поломали ли мы чего-то, что должно было работать. Так как лидирующие символы полностью опциональны, совпадает начало строки, ноль нецифровых символов, потом запоминается группа из трёх цифр (800), потом один нецифровой символ (дефис), потом группа из трёх цифр (555), потом один нецифровой (дефис), потом запоминается группа из четырёх цифр (1212), потом ноль нецифровых символов, потом группа цифр из нуля символов, потом конец строки.
- ④ Вот где регулярное выражение выколупывает мне глаза тупым предметом. Почему этот номер не совпал? Потому что 1 находится до кода города, но вы допускали что все лидирующие символы до кода города не цифры (`\D*`).

Давайте вернёмся назад на секунду. До сих пор регулярное выражение совпадало с началом строки. Но сейчас вы видите что в начале могут быть непредсказуемые символы которые мы хотели бы проигнорировать. Лучше не пытаться подобрать совпадение для них, а просто пропустить их все, давайте сделаем другое допущение: не пытаться совпадать с началом строки вообще. Этот подход показан в следующем примере.

```
>>> phonePattern = re.compile(r'(\d{3})\D*(\d{3})\D*(\d{4})\D*(\d*)$') ①
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups() ②
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212') ③
('800', '555', '1212', '')
>>> phonePattern.search('80055512121234') ④
('800', '555', '1212', '1234')
```

- ① Заметьте отсутствие `^` в регулярном выражении. Вы больше не совпадаете с началом строки. Ничего теперь не подсказывает как следует поступать с введёнными данными вашему регулярному выражению. Обработчик регулярного выражения будет выполнять тяжелую работу чтобы разобраться где же введённая строка начнёт совпадать.
- ② Теперь вы можете успешно обработать телефонный номер который включает лидирующие символы и цифры, плюс разделители любого типа между частями номера.

- ③ Простая проверка. Всё работает.
- ④ И даже это работает тоже.

Видите как быстро регулярное выражение выходит из под контроля? Бросим взгляд на предыдущие итерации. Можете ли вы объяснить разницу между одним и другим?

Пока вы понимаете финальный ответ (а это действительно он; если вы обнаружили ситуацию которую он не обрабатывает, я не желаю об этом знать), напомним подробное регулярное выражение, до тех пор пока вы не забыли почему вы сделали выбор который вы сделали.

```
>>> phonePattern = re.compile(r'''
    # don't match beginning of string, number can start anywhere
    (\d{3})    # area code is 3 digits (e.g. '800')
    \D*       # optional separator is any number of non-digits
    (\d{3})    # trunk is 3 digits (e.g. '555')
    \D*       # optional separator
    (\d{4})    # rest of number is 4 digits (e.g. '1212')
    \D*       # optional separator
    (\d*)     # extension is optional and can be any number of digits
    $        # end of string
''', re.VERBOSE)
>>> phonePattern.search('work 1-(800) 555.1212 #1234').groups() ①
('800', '555', '1212', '1234')
>>> phonePattern.search('800-555-1212')                          ②
('800', '555', '1212', '')
```

- ① Кроме того что оно разбито на множество строк, это совершенно такое же регулярное выражение как было в последнем шаге, и не будет сюрпризом что оно обрабатывает такие же входные данные.
- ② Финальная простая проверка. Да, всё ещё работает. Вы сделали это.

## Итоги

Это всего лишь верхушка айсберга того что могут делать регулярные выражения. Другими словами, даже если вы полностью ошеломлены ими сейчас, поверьте мне, вы ещё ничего не видели.

Вы должны быть сейчас умелыми в следующей технике:

- ^ совпадение с началом строки.
- \$ совпадение с концом строки.

$\backslash b$  совпадает с границей слова.

$\backslash d$  совпадает с цифрой.

$\backslash D$  совпадает с не цифрой.

$x?$  совпадает с опциональным символом  $x$  (другими словами ноль или один символов  $x$ ).

$x^*$  совпадает с ноль или более  $x$ .

$x^+$  совпадает с один или более  $x$ .

$x\{n, m\}$  совпадает с  $x$  не менее  $n$  раз, но не более  $m$  раз.

$(a|b|c)$  совпадает с  $a$  или  $b$  или  $c$ .

$(x)$  группа для запоминания. Вы можете получить значение используя метод `groups()` на объекте который возвращает `re.search`.



Регулярные выражения экстремально мощный инструмент, но они не всегда корректный способ для решения любой проблемы. Вы должны изучить побольше о них чтобы разобраться когда они являются подходящими для решения проблемы, так как иногда они могут добавить больше проблем чем решить

---



# Замыкания и генераторы

## Погружение

По причинам, превосходящим всяческое понимание, я всегда восхищался языками. Не языками программирования. Хотя да, ими, а также языками естественными. Возьмем, к примеру, английский. Английский язык — это шизофренический язык, который заимствует слова из немецкого, французского, испанского и латинского языков (не говоря уже об остальных). Откровенно говоря, «заимствует» — неуместное слово; он их скорее «ворует». Или, возможно, «ассимилирует» — как Борги. Да, хороший вариант.

« Мы Борги. Ваши лингвистические и этимологические особенности станут нашими. Сопротивление бесполезно. »

В этой главе вы узнаете о существительных во множественном числе. Также вы узнаете о функциях, которые возвращают другие функции, о сложных регулярных выражениях и генераторах. Но сначала давайте поговорим о том, как образуются существительные во множественном числе. (Если вы не читали раздел посвященный регулярным выражениям, сейчас — самое время. Материал этого раздела подразумевает, что вы понимаете основы регулярных выражений, и довольно быстро перейдете к их нетривиальному использованию).

Если вы выросли в англоязычной стране или изучали английский в формальной школьной обстановке, вы, вероятно, знакомы с основными правилами:

1. Если слово заканчивается буквами *S*, *X* или *Z*, следует добавить *ES*. *Bass* становится *basses*, *fax* становится *faxes*, а *waltz* — *waltzes*.
2. Если слово заканчивается звонкой *H*, следует добавить *ES*; если заканчивается глухой *H*, то нужно просто добавить *S*. Что такое звонкая *H*? Это такая, которая вместе с другими буквами объединяется в звук,

который вы можете слышать. Соответственно, *coach* становится *coaches*, и *rash* становится *rashes*, потому что вы слышите звуки *CH* и *SH*, когда произносите эти слова. Но *cheetah* становится *cheetahs*, потому что *H* здесь глухая.

3. Если слово заканчивается на *Y*, которая читается как *I*, то замените *Y* на *IES*; если *Y* объединена с гласной и звучит как-то по-другому, то просто добавьте *S*. Так что *vacancy* становится *vacancies*, но *day* становится *days*.
4. Если ни одно из правил не подходит, просто добавьте *S* и надейтесь на лучшее.

(Я знаю, существует множество исключений. *Man* становится *men*, а *woman* — *women*, но *human* становится *humans*. *Mouse* — *mice*, а *louse* — *lice*, но *house* во множественной числе — *houses*. *Knife* становится *knives*, а *wife* становится *wives*, но *lowlife* становится *lowlives*. И не заставляйте меня вглядываться в слова, которые не изменяются во множественном числе, как например *sheep*, *deer* или *haiku*).

Остальные языки, конечно, совершенно другие.

Давайте разработаем библиотеку на Python, которая автоматически образует множественное число английского слова. Мы начнем с этих четырех правил, но не забывайте, что вам неизбежно понадобится добавлять еще.

## Я знаю, используем регулярные выражения!

Итак, вы смотрите на слова, и, по крайней мере в английском, это означает, что вы смотрите на последовательности символов. У вас есть правила, которые говорят, что нужно искать разные комбинации символов, потом совершать с ними различные действия. Похоже, это работа для регулярных выражений!

```
import re
```

```
def plural(noun):
```

```
    if re.search('[sxz]$', noun):
```

□UNIQd2f4acf57a9a5326-ref-00000003-

QINU□

```
        return re.sub('$', 'es', noun)
```

□UNIQd2f4acf57a9a5326-ref-00000006-

QINU□

```
    elif re.search('[^aeioudgkprt]h$', noun):
```

```
        return re.sub('$', 'es', noun)
```

```
    elif re.search('[^aeiou]y$', noun):
```

```
        return re.sub('y$', 'ies', noun)
```

```
    else:
```

```
        return noun + 's'
```

1. ↑ Это регулярное выражение, но оно использует синтаксис, который вы не видели в главе Регулярные Выражения. Квадратные скобки означают «найти совпадения ровно с одним из этих символов». Поэтому [sxz] означает «s или x, или z», но только один из них. Символ \$ должен быть знаком вам. Он ищет совпадения с концом строки. Все регулярное выражение проверяет, заканчивается ли noun на s, x или z.
2. ↑ Упомянутая функция re.sub() производит замену подстроки на основе регулярного выражения.

Рассмотрим замену с помощью регулярного выражения внимательнее.

```
>>> import re
>>> re.search('[abc]', 'Mark') ①
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.sub('[abc]', 'o', 'Mark') ②
'Mork'
>>> re.sub('[abc]', 'o', 'rock') ③
'rook'
>>> re.sub('[abc]', 'o', 'caps') ④
'oops'
```

1. Содержит ли строка Mark символы a, b или c? Да, содержит a.
2. Отлично, теперь ищем a, b или c и заменяем на o. Mark становится Mork.
3. Та же функция превращает rock в rook.
4. Вы могли подумать, что этот код caps преобразует в oaps, но он этого не делает. re.sub заменяет все совпадения, а не только первое найденное. Так что, данное регулярное выражение превратит caps в oops, потому что оба символа c и a заменяются на o.

Вернемся снова к функции plural()...

```
def plural(noun):
    if re.search('[sxz]$', noun):
        return re.sub('$', 'es', noun) ①
    elif re.search('[^aeioudgkprt]h$', noun): ②
        return re.sub('$', 'es', noun)
    elif re.search('[^aeiou]y$', noun): ③
        return re.sub('y$', 'ies', noun)
    else:
        return noun + 's'
```

1. Здесь вы заменяете конец строки (найденный с помощью символа \$) на строку es. Другими словами, добавляете es к строке. Вы могли бы совершить то же самое с помощью конкатенации строк, например, как

noun + 'es', но я предпочел использовать регулярные выражения для каждого правила, по причинам которые станут ясны позже.

2. Взгляните-ка, это регулярное выражение содержит кое-что новое. Символ ^ в качестве первого символа в квадратных скобках имеет особый смысл: отрицание. [^abc] означает «любой отдельный символ кроме a, b или c». Так что [^aeiou dgkprt] означает любой символ кроме a, e, i, o, u, d, g, k, p, r или t. Затем за этим символом должен быть символ h, следом за ним — конец строки. Вы ищете слова, заканчивающиеся на H, которую можно услышать.
3. То же самое здесь: найти слова, которые заканчиваются на Y, в которых символ перед Y — не a, e, i, o или u. Вы ищете слова, заканчивающиеся на Y, которая звучит как I.

Давайте внимательнее рассмотрим регулярные выражения с участием отрицания.

```
>>> import re
>>> re.search('[^aeiou]y$', 'vacancy') ①
<_sre.SRE_Match object at 0x001C1FA8>
>>> re.search('[^aeiou]y$', 'boy') ②
>>>
>>> re.search('[^aeiou]y$', 'day')
>>>
>>> re.search('[^aeiou]y$', 'pita') ③
>>>
```

1. vacancy подходит, потому что оно заканчивается на су, и с — не a, e, i, o или u.
2. boy не подходит, потому что оно заканчивается на ou, а вы конкретно указали, что символ перед y не может быть o. day не подходит, потому что он заканчивается на ay.
3. pita не подходит, потому что оно не заканчивается на y.

```
>>> re.sub('y$', 'ies', 'vacancy') ①
'vacancies'
>>> re.sub('y$', 'ies', 'agency') 'agencies'
>>> re.sub('([^aeiou])y$', r'\1ies', 'vacancy') ②
'vacancies'
```

1. Это регулярное выражение преобразует vacancy в vacancies, а agency — в agencies, что вам и нужно. Заметьте, что оно бы преобразовало boy в boies, но этого в функции никогда не произойдет, потому что вы сначала сделали re.search с целью выяснить, следует ли делать re.sub.
2. Замечу заодно, что возможно объединить эти два регулярных выражения (одно чтобы выяснить применяется ли правило, а другое чтобы

собственно его применить) в одно регулярное выражение. Вот так выглядел бы результат. Большая часть должна быть вам знакома: вы используете запоминаемую группу, о которой вы узнали из Учебный пример: Разбор телефонного номера. Группа используется чтобы запомнить символ перед у. Затем в подстановочной строке, вы используете новый синтаксис, \1, который означает «эй, та первая группа, которую ты запомнил? положи ее сюда». Таким образом, вы помните с перед у; когда вы делаете подстановку, вы ставите с на место с, и ies на место у. (Если у вас более одной запоминаемой группы, можете использовать \2 и \3 и так далее.)

Замены с использованием регулярных выражений являются чрезвычайно мощным инструментом, а синтаксис \1 делает их еще более мощным. Но вся операция, объединенная в одно регулярное выражение, также становится сложной для чтения, кроме того такой способ не соотносится напрямую с тем, как вы изначально описали правила формирования множественного числа. Изначально вы спроектировали правила в форме «если слово заканчивается на S, X или Z, то добавьте ES». А если вы смотрите на функцию, то у вас — две строки кода, которые говорят «если слово заканчивается на S, X или Z, то добавьте ES». Еще ближе к оригинальному варианту приблизиться никак не получится.

## Список функций

Сейчас вы добавите уровень абстракции. Вы начали с определения списка правил: если верно это, сделай то, иначе обращайтесь к следующему правилу. Давайте временно усложним часть программы, так что вы сможете упростить другую ее часть.

```
import re
```

```
def match_sxz(noun):
    return re.search('[sxz]$', noun)
```

```
def apply_sxz(noun):
    return re.sub('$', 'es', noun)
```

```
def match_h(noun):
    return re.search('[^aeioudgkprt]h$', noun)
```

```
def apply_h(noun):
    return re.sub('$', 'es', noun)
```

```
def match_y(noun):①
    return re.search('[^aeiou]y$', noun)
```

```

def apply_y(noun):
    return re.sub('y$', 'ies', noun)

def match_default(noun):
    return True

def apply_default(noun):
    return noun + 's'

rules = ((match_sxz, apply_sxz),
         (match_h, apply_h),
         (match_y, apply_y),
         (match_default, apply_default)
        )

def plural(noun):
    for matches_rule, apply_rule in rules:
        if matches_rule(noun):
            return apply_rule(noun)

```

1. Теперь каждое правило-условие совпадения является отдельной функцией которая возвращает результаты вызова функции `re.search()`.
2. Каждое правило-действие также является отдельной функцией, которая вызывает функцию `re.sub()` чтобы применить соответствующее правило формирования множественного числа.
3. Вместо одной функции (`plural()`) с несколькими правилами у вас теперь есть структура данных `rules`, являющаяся последовательностью пар функций.
4. Поскольку правила развернуты в отдельной структуре данных, новая функция `plural()` может быть сокращена до нескольких строк кода. Используя цикл `for`, из структуры `rules` можно извлечь правила условия и замены одновременно. При первой итерации `for` цикла, `match_rule` станет `match_sxz`, а `apply_rule` станет `apply_sxz`. Во время второй итерации, если мы до нее дойдем, `matches_rule` будет присвоено `match_h`, а `apply_rule` станет `apply_h`. Функция гарантированно вернет что-нибудь по окончании работы, потому что последнее правило совпадения (`match_default`) просто возвращает `True`, подразумевая, что соответствующее правило замены (`apply_default`) всегда будет применено.

Причиной, по которой этот пример работает, является тот факт, что в Python все является объектом, даже функции. Структура данных `rules` содержит функции — не имена функций, а фактические функции-объекты. Когда они

присваиваются в for цикле, `matches_rule` и `apply_rule` являются настоящими функциями, которые вы можете вызывать. При первой итерации for цикла, это эквивалентно вызову `matches_sxz(noun)`, и если она возвращает совпадение, вызову `apply_sxz(noun)`.

-> Переменная «rules» — это последовательность пар функций. [war-robin.com]

Если этот дополнительный уровень абстракции сбивает вас с толку, попробуйте развернуть функцию, чтобы увидеть, что мы получаем то же самое. Весь цикл for эквивалентен следующему:

```
def plural(noun):
    if match_sxz(noun):
        return apply_sxz(noun)
    if match_h(noun):
        return apply_h(noun)
    if match_y(noun):
        return apply_y(noun)
    if match_default(noun):
        return apply_default(noun)
```

Преимуществом здесь является то, что функция `plural()` упрощена. Она принимает последовательность правил, определенных где-либо, и проходит по ним.

1. Получить правило совпадения
2. Правило срабатывает? Тогда применить правило замены и вернуть результат.
3. Нет совпадений? Начать с пункта 1.

Правила могут быть определены где угодно, любым способом. Для функции `plural()` абсолютно нет никакой разницы.

Итак, добавление этого уровня абстракции стоило того? Вообще-то пока нет. Попробуем представить, что потребуется для добавления нового правила в функцию. В первом примере этого потребовало бы добавить новую конструкцию `if` в функцию `plural()`. Во втором примере, это потребовало бы добавить две функции, `match_foo()` и `apply_foo()`, а затем обновить последовательность `rules` чтобы указать, когда новые правила совпадения и замены должны быть вызваны по отношению к остальным правилам.

Но на самом деле это только средство, чтобы перейти к следующей главе. Двигаемся дальше...



## Список шаблонов

Определение отдельных именованных функций для каждого условия и правила замены вовсе не является необходимостью. Вы никогда не вызываете их напрямую; вы добавляете их в последовательность `rules` и вызываете их через нее. Более того, каждая функция следует одному из двух шаблонов. Все функции совпадения вызывают `re.search()`, а все функции замены вызывают `re.sub()`. Давайте исключим шаблоны, чтобы объявление новых правил было более простым.

```
import re
```

```
def build_match_and_apply_functions(pattern, search, replace):
    def matches_rule(word):                                ①
        return re.search(pattern, word)
    def apply_rule(word):                                  ②
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)                     ③
```

1. `build_match_and_apply_functions()` — это функция, которая динамически создает другие функции. Она принимает `pattern`, `search` и `replace`, затем определяет функцию `matches_rule()`, которая вызывает `re.search()` с шаблоном `pattern`, переданный функции `build_match_and_apply_functions()` в качестве аргумента, и `word`, который передан функции `matches_rule()`, которую вы определяете.
2. Строим функцию `apply` тем же способом. Функция `apply` — это функция, которая принимает один параметр, и вызывает `re.sub()` с `search` и `replace` параметрами, переданными функции `build_match_and_apply_functions`, и `word`, переданным функции `apply_rule()`, которую вы создаете. Подход, заключающийся в использовании значений внешних параметров внутри динамической функции называется замыканиями. По сути вы определяете константы в функции замены: он принимает один параметр (`word`), но затем действует используя его и два других значения (`search` и `replace`), которые были установлены в момент определения функции замены.
3. В конце концов функция `build_match_and_apply_functions()` возвращает кортеж с двумя значениями, двумя функциями, которые вы только что создали. Константы, которые вы определили внутри тех функций (`pattern` внутри функции `match_rule()`), `search` и `replace` в функции `apply_rule()` остаются с этими функциями, даже. Это безумно круто.

Если это сбивает вас с толку (и так и должно быть, это весьма странное поведение), картина может проясниться, когда вы увидите как использовать этот подход.

```

patterns = \
(
    ('[sxz]$', '$', 'es'),
    ('^[^aeiou]d[kprt]h$', '$', 'es'),
    ('(qu|^[^aeiou])y$', 'y$', 'ies'),
    ('$', '$', 's')
)
rules = [build_match_and_apply_functions(pattern, search, replace)
         for (pattern, search, replace) in patterns]

```

1. Наши правила формирования множественного числа теперь определены как кортеж кортежей строк (не функций). Первая строка в каждой группе — это регулярное выражение, которое вы бы использовали в `re.search()` чтобы определить, подходит ли данное правило. Вторая и третья строки в каждой группе — это выражения для поиска и замены, которые вы бы использовали в `re.sub()` чтобы применить правило и преобразовать существительное во множественное число.
2. В альтернативном правиле есть небольшое изменение. В прошлом примере функция `match_default()` просто возвращает `True`, подразумевая, что если ни одно конкретное правило не применилось, код должен просто добавить `s` в конец данного слова. Функционально данный пример делает то же самое. Окончательное регулярное выражение узнает, заканчивается ли слово (`$` ищет конец строки). Конечно же, у каждой строки есть конец, даже у пустой, так что выражение всегда срабатывает. Таким образом, она служит той же цели, что и функция `match_default()`, которая всегда возвращала `True`: она гарантирует, что если нет других конкретных выполненных правил, код добавляет `s` в конец данного слова.
3. Это волшебная строка. Она принимает последовательность строк в `patterns` и превращает их в последовательность функций. Как? «Отображением» строк в функцию `build_and_apply_functions()`. То есть она берет каждую тройку строк и вызывает функцию `build_match_and_apply_functions()` с этими тремя строками в качестве аргументов. Функция `build_match_and_apply_functions()` возвращает кортеж из двух функций. Это означает, что `rules` в конце концов функционально становится эквивалентным предыдущему примеру: список кортежей, где каждый кортеж — это пара функций. Первая функция — это функция совпадения, которая вызывает `re.search()`, а вторая функция — применение правила (замена), которая вызывает `re.sub()`.

Завершим эту версию скрипта главной точкой входа, функцией `plural()`.

```

def plural(noun):
    for matches_rule, apply_rule in rules:
        if matches_rule(noun):
            return apply_rule(noun)

```

1. Поскольку список `rules` — тот же самый, что и в предыдущем примере (да, так и есть), нет ничего удивительного в том, что функция `plural()` совсем не изменилась. Она является полностью обобщенной; она принимает список функций-правил и вызывает их по порядку. Ее не волнует, как определены правила. В предыдущем примере они были определены как отдельные именованные функции. Теперь же они создаются динамически сопоставлением результата функции `build_match_and_apply_functions()` списку обычных строк. Это не играет никакой роли. функция `plural()` продолжает работать как и раньше.

## Файл шаблонов

Вы вынесли весь дублирующийся код и добавили достаточно абстракций для возможности хранить правила формирования множественного числа в списке строк. Следующий логический этап — взять эти строки и расположить их в отдельном файле, где они могут поддерживаться отдельно от использующего их кода.

Во-первых, давайте создадим текстовый файл, содержащий нужные нам правила. Никаких сложных структур данных, просто разделенные на три колонки данные. Назовем его `plural4-rules.txt`

```
[sxz]$          $ es
[^aeiou]h$      $ es
[^aeiou]y$      y$ ies
$              $ s
```

Теперь давайте посмотрим, как вы можете использовать этот файл с правилами.

```
import re
```

```
def build_match_and_apply_functions(pattern, search, replace): ①
    def matches_rule(word):
        return re.search(pattern, word)
    def apply_rule(word):
        return re.sub(search, replace, word)
    return (matches_rule, apply_rule)
```

```
rules = []
```

```
with open('plural4-rules.txt', encoding='utf-8') as pattern_file: ②
    for line in pattern_file: ③
        pattern, search, replace = line.split(None, 3) ④
        rules.append(build_match_and_apply_functions( ⑤
            pattern, search, replace))
```

1. Функция `build_match_and_apply_functions()` не изменилась. Вы по-прежнему используете замыкания, чтобы динамически создать две функции, которые будут использовать переменные из внешней функции.
2. Глобальная функция `open()` открывает файл и возвращает файловый объект. В данном случае файл, который мы открываем, содержит строки-шаблоны для правил формирования множественного числа. Утверждение `with` создает то, что называется контекстом: когда блок `with` заканчивается, Python автоматически закроет файл, даже если внутри блока `with` было выброшено исключение. Подробнее о блоках `with` и файловых объектах вы узнаете из главы Файлы.
3. Форма «`for line in <fileobject>`» читает данные из открытого файла построчно и присваивает текст переменной `line`. Подробнее про чтение файлов вы узнаете из главы Файлы.
4. Каждая строка в файле действительно содержит три значения, но они разделены пустым пространством (табуляцией или пробелами, без разницы). Чтобы разделить их, используйте строковый метод `split()`. Первый аргумент для `split()` — `None`, что означает «разделить любым символом свободного пространства (табуляцией или пробелом, без разницы)». Второй аргумент — `3`, что означает «разбить свободным пространством 3 раза, затем оставить остальную часть строки». Строка вида «`[sxz]$ $ es`» будет разбита и преобразована в список `['[sxz]$', '$', 'es']`, что означает что `pattern` станет `'[sxz]$',` `search` — `'$'`, а `replace` получит значение `'es'`. Это довольно мощно для одной маленькой строки кода.
5. В конце концов, вы передаете `pattern`, `search` и `replace` функции `build_match_and_apply_function()`, которая возвращает кортеж функций. Вы добавляете этот кортеж в список `rules`, и в завершении `rules` хранит список функций поиска совпадений и выполнения замен, который ожидает функция `plural()`.

Сделанное улучшение заключается в том, что вы полностью вынесли правила во внешний файл, так что он может поддерживаться отдельно от использующего его кода. Код — это код, данные — это данные, а жизнь хороша.

## Генераторы

Но ведь будет круто если обобщенная функция `plural()` будет разбирать файл с правилами? Извлеки правила, найди совпадения, примени соответствующие изменения, переходи к следующему правилу. Это все, что функции `plural()` придется делать, и больше ничего от нее не требуется.

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3)
```

```
yield build_match_and_apply_functions(pattern, search, replace)
```

```
def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename):
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {}'.format(noun))
```

Как черт возьми это работает? Давайте сначала посмотрим на пример с пояснениями.

```
>>> def make_counter(x):
...     print('entering make_counter')
...     while True:
...         yield x ①
...         print('incrementing x')
...         x = x + 1
...
>>> counter = make_counter(2) ②
>>> counter ③
<generator object at 0x001C9C10>
>>> next(counter) ④
entering make_counter
2
>>> next(counter) ⑤
incrementing x
3
>>> next(counter) ⑥
incrementing x
4
```

1. Присутствие ключевого слова `yield` в `make_counter` означает, что это не обычная функция. Это особый вид функции, которая генерирует значения по одному. Вы можете думать о ней как о продолжаемой функции. Её вызов вернёт генератор, который может быть использован для генерации последующих значений `x`.
2. Чтобы создать экземпляр генератора `make_counter`, просто вызовите его как и любую другую функцию. Заметьте, что фактически это не выполняет кода функции. Вы можете так сказать, потому что первая строка функции `make_counter()` вызывает `print()`, но ничего до сих пор не напечатано.
3. Функция `make_counter()` возвращает объект-генератор.
4. Функция `next()` принимает генератор и возвращает его следующее значение. Первый раз, когда вы вызываете `next()` с генератором `counter`,

он исполняет код в `make_counter()` до первого утверждения `yield`, затем возвращает значение, которое было возвращено `yield`. В данном случае, это будет 2, поскольку изначально вы создали генератор вызовом `make_counter(2)`.

5. Повторный вызов `next()` с тем же генератором продолжает вычисления точно там, где они были прерваны, и продолжает до тех пор, пока не встретит следующий `yield`. Все переменные, локальные состояния и т. д. сохраняются во время `yield`, и восстанавливаются при вызове `next()`. Следующая строка кода, ожидающая исполнения, вызывает `print()`, который печатает `incrementing x`. После этого следует утверждение `x = x + 1`. Затем снова исполняется цикл `while`, и первое, что в нём встречается — утверждение `yield x`, которое сохраняет все состояние и возвращает текущее значение `x` (сейчас это 3).
6. После второго вызова `next(counter)` происходит всё то же самое, только теперь `x` становится равным 4.

Поскольку `make_counter` входит в бесконечный цикл, вы бы теоретически могли заниматься этим бесконечно, а он продолжал бы увеличивать `x` и возвращать его значения. Но вместо этого давайте посмотрим на более продуктивное использование генераторов.

## Генератор последовательности Фибоначчи

```
def fib(max):
```

```
    a, b = 0, 1    ①
```

```
    while a < max:
```

```
        yield a    ②
```

```
        a, b = b, a + b    ③
```

1. Последовательность Фибоначчи — это последовательность чисел, в которой каждое число является суммой двух предыдущих. Она начинается с 0 и 1, сначала постепенно возрастает, потом растет все быстрее и быстрее. Чтобы начать последовательность, вам необходимо две переменные: `a` начинает с 0, а `b` — с 1.
2. `a` является начальным значением последовательности, поэтому её следует вернуть.
3. `b` является следующим числом последовательности, так что присвойте ее `a`, но так же посчитайте следующее значение (`a + b`) и присвойте его `b` для последующего использования. Заметьте, что это происходит одновременно. Если `a` равно 3, а `b` равно 5, то `a, b = b, a + b` установит `a` в 5 (предыдущее значение `b`) а `b` в 8 (сумма предыдущих значений `a` и `b`).

Так что теперь у вас есть функция выдает последовательные числа Фибоначчи. Конечно, вы могли бы сделать это с помощью рекурсии, но данная реализация проще читается. Помимо этого, она лучше работает с циклами `for`.

->yield приостанавливает функцию, next() снова запускает ее в том же состоянии

```
>>> from fibonacci import fib
>>> for n in fib(1000): ①
...     print(n, end=' ') ②
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> list(fib(1000)) ③
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

1. Вы можете использовать генератор типа fib() непосредственно в for цикле. Цикл for автоматически вызывает функцию next() чтобы получить значения из генератора fib() и присвоить их переменной for цикла n.
2. Каждый раз проходя for цикл, n принимает новое значение от yield в функции fib(), и все что вам нужно сделать — напечатать его. Как только fib() выходит за пределы чисел (а становится больше, чем max, которое в данном случае равно 1000), так сразу цикл for заканчивает работу.
3. Это полезная техника: отдайте генератор функции list(), и она пройдет в цикле весь генератор (точно так же, как и цикл for в предыдущем примере) и вернет список всех значений.

## A Plural Rule Generator

Давайте вернемся к plural5.py и посмотрим как работает эта версия функции plural().

```
def rules(rules_filename):
    with open(rules_filename, encoding='utf-8') as pattern_file:
        for line in pattern_file:
            pattern, search, replace = line.split(None, 3) ①
            yield build_match_and_apply_functions(pattern, search, replace) ②

def plural(noun, rules_filename='plural5-rules.txt'):
    for matches_rule, apply_rule in rules(rules_filename): ③
        if matches_rule(noun):
            return apply_rule(noun)
    raise ValueError('no matching rule for {}'.format(noun))
```

1. Никакой магии. Помните, что строки файла правил содержат по три значения, разделенных пустым пространством, так что вы используете line.split(None, 3) чтобы получить три «колонки» и присвоить их трем локальным переменным.
2. А затем вы вызываете yield. Что вы возвращаете? Две функции, созданные динамически вашим старым помощником, build\_match\_and\_apply\_functions(), который такой же как и в предыдущих



примерах. Другими словами, `rules()` — это генератор, который отдаёт правила совпадения и изменения по требованию.

3. Поскольку `rules()` — это генератор, вы можете использовать его напрямую в `for` цикле. При первом прохождении `for` цикла, вы вызовете функцию `rules()`, которая откроет файл шаблонов, прочитает первую строку, динамически построит функцию условия и модификации из шаблона в этой строке, и вернет динамически созданные функции. При прохождении через `loop` цикл второй раз, вы продолжите ровно с того места, в котором покинули `rules()` (это было внутри `for line in pattern_file` цикла). Первое, что он сделает — прочитает следующую строку файла (который до сих пор открыт), динамически создаст другие функции условия и модификации на основании шаблонов этой строки файла, и отдаст эти две функции.

Что вы приобрели по сравнению с вариантом 4? Меньшее время запуска. В варианте 4 когда вы импортировали модуль `plural4`, он читал весь файл шаблонов и строил список всех возможных правил ещё до того, как вы вообще подумали о вызове функции `plural()`. С генераторами вы можете выполнять все действия лениво: вы читаете первое правило и создаете функции и пробуете их, и если это срабатывает вы никогда не читаете остальной файл и не создаете другие функции.

В чем вы теряете? В производительности! Каждый раз когда вы вызываете функцию `plural()`, генератор `rules()` начинает всё с начала — что означает открытие заново файла шаблонов и чтение с самого начала построено.

Что если бы вы могли взять лучшее из двух миров: минимальные расходы на запуск (не исполнять никакого кода во время `import`) и максимальная производительность (не создавать одни и те же функции снова и снова). И да, вы по-прежнему хотите хранить правила в отдельном файле (потому что код — это код, а данные — это данные), настолько долго, пока вам вдруг не потребуется читать одну и ту же строку дважды.

Чтобы сделать это, вам необходимо будет построить свой собственный итератор. Но перед тем как вы сделаете это, вам необходимо изучить классы в Python.

## Материалы для дальнейшего чтения

- PEP 255: Simple Generators
  - Understanding Python's «with» statement
  - Closures in Python
  - Числа Фибоначчи
  - English Irregular Plural Nouns
-

# Классы и итераторы

## Погружение

Генераторы на самом деле — всего лишь частный случай итераторов. Функция, возвращающая (yields) значения, является простым и компактным способом получения функциональности итератора, без непосредственного *создания* итератора. Помните генератор чисел Фибоначчи? Вот набросок того, как мог бы выглядеть аналогичный итератор:

```
class Fib:
    "iterator that yields numbers in the Fibonacci sequence"

    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib
```

Давайте рассмотрим этот пример более детально:

```
class Fib:
```

```
class? Что такое класс?
```

## Определение классов

Python полностью объектно-ориентирован, то есть вы можете определять свои собственные классы, наследовать новые классы от своих или встроенных классов, и создавать экземпляры классов, которые уже определили.

Определить класс в Python просто. Также как и в случае с функциями, отдельное объявление интерфейса не требуется. Вы просто определяете класс и начинаете программировать. Определение класса в Python начинается с зарезервированного слова `class`, за которым следует имя (идентификатор) класса. Формально, это все, что необходимо, в случае, когда класс не должен быть унаследован от другого класса.

```
class ParayaWhip: [К 1]
```

```
    pass [К 2]
```

1. ↑ Определенный выше класс имеет имя `ParayaWhip` и не наследует никакой другой класс. Имена классов, как правило, пишутся с большой буквы, Например `ВотТак`, но это всего лишь соглашение, а не требование.
2. ↑ Вы наверное уже догадались, что каждая строка в определении класса имеет отступ, также как и в случае с функциями, оператором условного перехода `if`, циклом `for` или любым другим блоком кода. Первая строка без отступа находится вне блока `class`.

Класс `ParayaWhip` не содержит определений методов или атрибутов, но с точки зрения синтаксиса, тело класса не может оставаться пустым. В таких случаях используется оператор `pass`. В языке Python `pass` — зарезервированное слово, которое говорит интерпретатору: «идем дальше, здесь ничего нет». Это оператор не делающий ровным счетом ничего, но тем не менее являющийся удобным решением, когда вам нужно сделать заглушку для функции или класса.



Выражение `pass` в языке Python аналог пустого множества или фигурных скобок в языках Java или C++.

Многие классы наследуются от других классов, но не этот. Многие классы определяют свои методы, но не этот. Класс в Python не обязан иметь ничего, кроме имени. В частности, людям знакомым с C++ может показаться странным, что у класса в Python отсутствуют в явном виде конструктор и деструктор. Несмотря на то, что это не является обязательным, класс в Python может иметь нечто, похожее на конструктор: метод `__init__()`.

### Метод `__init__()`

В следующем примере демонстрируется инициализация класса `Fib`, с помощью метода `__init__()`.

```
class Fib:
    """iterator that yields numbers in the Fibonacci sequence"""
    def __init__(self, max):
```

[К 1]

[К 2]

1. ↑ Классы, по аналогии с модулями и функциями могут (и должны) иметь строки документации (docstrings).
2. ↑ Метод `__init__()` вызывается сразу же после создания экземпляра класса. Было бы заманчиво, но формально неверно, считать его «конструктором» класса. Заманчиво, потому что он напоминает конструктор класса в языке C++: *внешне* (общепринято, что метод `__init__()` должен быть первым методом, определенным для класса), и *в действии* (это первый блок кода, исполняемый в контексте только что созданного экземпляра класса). Неверно, потому что на момент вызова `__init__()` объект уже фактически является созданным, и вы можете оперировать корректной ссылкой на него (`self`)

Первым аргументом любого метода класса, включая метод `__init__()`, всегда является ссылка на текущий экземпляр класса. Принято называть этот аргумент `self`. Этот аргумент выполняет роль зарезервированного слова `this` в C++ или Java, но, тем не менее, в Python `self` не является зарезервированным. Несмотря на то, что это всего лишь соглашение, пожалуйста не называйте этот аргумент как либо еще.

В случае метода `__init__()`, `self` ссылается на только что созданный объект; в остальных методах — на экземпляр, метод которого был вызван. И, хотя вам необходимо явно указывать `self` при определении метода, при вызове этого не требуется; Python добавит его для вас автоматически.

## Создание экземпляров

Для создания нового экземпляра класса в Python нужно вызвать класс, как если бы он был функцией, передав необходимые аргументы для метода `__init__()`. В качестве возвращаемого значения мы получим только что созданный объект.

```
>>> import fibonacci2
>>> fib = fibonacci2.Fib(100)
>>> fib
<fibonacci2.Fib object at 0x00DB8810>
>>> fib.__class__
<class 'fibonacci2.Fib'>
>>> fib.__doc__
```

1

2

3

4

'iterator that yields numbers in the Fibonacci sequence'

1. Вы создаете новый экземпляр класса Fib (определенный в модуле fibonacci2) и присваиваете только что созданный объект переменной fib. Единственный переданный аргумент, 100, соответствует именованному аргументу max, в методе \_\_init\_\_() класса Fib.
2. fib теперь является экземпляром класса Fib
3. Каждый экземпляр класса имеет встроенный атрибут \_\_class\_\_, который указывает на класс объекта. Java программисты могут быть знакомы с классом Class, который содержит методы getName() и getSuperclass(), используемые для получения информации об объекте. В Python, метаданные такого рода доступны через соответствующие атрибуты, но используемая идея та же самая.
4. Вы можете получить строку документации (docstring) класса, по аналогии с функцией и модулем. Все экземпляры класса имеют одну и ту же строку документации.



Для создания нового экземпляра класса в Python, просто вызовите класс, как если бы он был функцией, явные операторы, как например new в C++ или Java, в языке Python отсутствуют.

## Переменные экземпляра

Перейдем к следующей строчке:

```
class Fib:
    def __init__(self, max):
        self.max = max
```

1.

1. Что такое self.max? Это переменная экземпляра. Она не имеет ничего общего с переменной max, которую мы передали в метод \_\_init\_\_() в качестве аргумента. self.max является «глобальной» для всего экземпляра. Это значит, что вы можете обратиться к ней из других методов.

```
class Fib:
    def __init__(self, max):
        self.max = max
```

1.

.

.

.

```
def __next__(self):
```

2.

```
fib = self.a
if fib > self.max:
```

1. self.max определена в методе `__init__()`...
2. ...и использована в методе `__next__()`.

Переменные экземпляра связаны только с одним экземпляром класса. Например, если вы создадите два экземпляра класса `Fib` с разными максимальными значениями, каждый из них будет помнить только свое собственное значение.

```
>>> import fibonacci2
>>> fib1 = fibonacci2.Fib(100)
>>> fib2 = fibonacci2.Fib(200)
>>> fib1.max
100
>>> fib2.max
200
```

## Итератор чисел Фибоначчи

Теперь вы готовы узнать как создать итератор. Итератор это обычный класс, который определяет метод `__iter__()`.

```
class Fib: ①
    def __init__(self, max): ②
        self.max = max

    def __iter__(self): ③
        self.a = 0
        self.b = 1
        return self

    def __next__(self): ④
        fib = self.a
        if fib > self.max:
            raise StopIteration ⑤
        self.a, self.b = self.b, self.a + self.b
        return fib ⑥
```

- ① Чтобы построить итератор с нуля, `Fib` должна быть классом, а не функцией.
-

# Подробнее об итераторах

## Погружение

HAWAII + IDAHO + IOWA + OHIO == STATES. Или, если записать это по-другому,  $510199 + 98153 + 9301 + 3593 == 621246$ . Думаете, я брежу? Нет, это просто головоломка.

Позвольте мне привести разгадку.

HAWAII + IDAHO + IOWA + OHIO == STATES  
 $510199 + 98153 + 9301 + 3593 == 621246$

H = 5  
A = 1  
W = 0  
I = 9  
D = 8  
O = 3  
S = 6  
T = 2  
E = 4

Такие головоломки называются криптоарифмами. Буквы составляют существующие слова, а если заменить каждую букву цифрой от 0 до 9, получится еще и правильное математическое равенство. Весь фокус в том, чтобы выяснить какая буква соответствует каждой цифре. Все вхождения каждой буквы должны заменяться одной и той же цифрой, одной цифре не могут соответствовать несколько букв и «слова» не могут начинаться с цифры 0.

В этой главе мы познакомимся с потрясающей программой на языке Python, написанной Рэймондом Хейттингером. Эта программа решает криптоарифмические головоломки и состоит всего из 14 строк кода.



## Наиболее известная криптоарифмическая головоломка $SEND + MORE = MONEY$ .

```
import re
import itertools

def solve(puzzle):
    words = re.findall('[A-Z]+', puzzle.upper())
    unique_characters = set(''.join(words))
    assert len(unique_characters) <= 10, 'Too many letters'
    first_letters = {word[0] for word in words}
    n = len(first_letters)
    sorted_characters = ''.join(first_letters) + \
        ''.join(unique_characters - first_letters)
    characters = tuple(ord(c) for c in sorted_characters)
    digits = tuple(ord(c) for c in '0123456789')
    zero = digits[0]
    for guess in itertools.permutations(digits, len(characters)):
        if zero not in guess[:n]:
            equation = puzzle.translate(dict(zip(characters, guess)))
            if eval(equation):
                return equation

if __name__ == '__main__':
    import sys
    for puzzle in sys.argv[1:]:
        print(puzzle)
        solution = solve(puzzle)
        if solution:
            print(solution)
```

Вы можете запустить программу из командной строки. В Linux это будет выглядеть так. (Выполнение программы может потребовать некоторого времени, зависящего от быстродействия вашего компьютера, а индикатор выполнения в программе отсутствует. Поэтому просто наберитесь терпения.)

```
you@localhost:~/diveintopython3/examples$ python3 alphametics.py "HAWAII + IDAHO + IOWA + OHIO == STATES"
```

```
HAWAII + IDAHO + IOWA + OHIO = STATES
```

```
510199 + 98153 + 9301 + 3593 == 621246
```

```
you@localhost:~/diveintopython3/examples$ python3 alphametics.py "I + LOVE + YOU == DORA"
```

I + LOVE + YOU == DORA

1 + 2784 + 975 == 3760

you@localhost:~/diveintopython3/examples\$ python3 alphametics.py "SEND + MORE == MONEY"

SEND + MORE == MONEY

9567 + 1085 == 10652

## Нахождение всех вхождений шаблона

Первая вещь, которую делает эта программа — это находит все слова (в оригинале — буквы, так как ищет она именно буквы, прим. перев) в головоломке.

```
>>> import re
```

```
>>> re.findall('[0-9]+', '16 2-by-4s in rows of 8') ①
```

```
['16', '2', '4', '8']
```

```
>>> re.findall('[A-Z]+', 'SEND + MORE == MONEY') ②
```

```
['SEND', 'MORE', 'MONEY']
```

① Модуль `re` реализует регулярные выражения в Питоне. В этом модуле есть удобная функция `findall()`, которая принимает в качестве параметров шаблон регулярного выражения и строку, и находит все подстроки, соответствующие шаблону. В этом случае шаблон соответствует последовательностям цифр. Функция `findall()` возвращает список всех подстрок, представляющих последовательность цифр.

② Здесь регулярное выражение соответствует последовательностям букв. И снова возвращаемое значение представляет собой список, каждый элемент которого — это строка, соответствующая шаблону регулярного выражения.

## Это самая трудная скороговорка на английском языке.

Вот другой пример, над которым вам, возможно, придется поломать голову.

```
>>> re.findall(' s.*? s', "The sixth sick sheikh's sixth sheep's sick.")
```

```
[' sixth s', " sheikh's s", " sheep's s"]
```

Удивлены? Приведенное регулярное выражение ищет пробел, за которым следуют буква `s`, кратчайшая возможная последовательность любых символов (`.*`), пробел и еще одна буква `s`.

1. The **sixth** sick sheikh's sixth sheep's sick.
2. The sixth **sick** sheikh's sixth sheep's sick.
3. The sixth sick **sheikh's s** sixth sheep's sick.

4. The sixth sick sheikh's **sixth s**heep's sick.
5. The sixth sick sheikh's sixth **sheep's s**ick.

Но функция `re.findall()` находит только три вхождения: первое, третье и пятое. Почему так? Потому что она не возвращает перекрывающиеся подстроки. Первая подстрока перекрывается со второй, поэтому и возвращается только первая строка, а вторая пропускается. Затем третья подстрока перекрывается с четвертой, поэтому возвращается только третья подстрока, а четвертая пропускается. Наконец возвращается пятая подстрока. Три совпадения, а не пять.

Это не имеет никакого отношения к решению криптарифмов, я просто подумал, что это интересно.

## Нахождение уникальных элементов в последовательностях

Множества делают задачу нахождения уникальных элементов в последовательности тривиальной.

```
>>> a_list = ['The', 'sixth', 'sick', "sheik's", 'sixth', "sheep's", 'sick']
>>> set(a_list) ①
{'sixth', 'The', "sheep's", 'sick', "sheik's"}
>>> a_string = 'EAST IS EAST'
>>> set(a_string) ②
{'A', ' ', 'E', 'I', 'S', 'T'}
>>> words = ['SEND', 'MORE', 'MONEY']
>>> ".join(words) ③
'SENDMOREMONEY'
>>> set(".join(words)) ④
{'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
```

① Получив список из нескольких строк, функция `set()` вернет множество уникальных строк из этого списка. Работа этой функции будет более понятной, если вы представите цикл `for`. Возьмите первый элемент из списка и добавьте его в множество. Второй. Третий. Четвертый. Пятый — постойте-ка, он уже есть в множестве, поэтому его не нужно добавлять, потому что множества в Питоне не позволяют иметь повторяющиеся элементы. Шестой. Седьмой — снова дубликат, пропускаем его. Каков результат? Все уникальные элементы исходного списка без дубликатов. Исходный список даже сортировать предварительно не нужно.

② Тот же подход работает и если функции `set()` передать строку, а не список, поскольку строка — это просто последовательность символов.

③ Получив список строк, функция `.join(a_list)` объединяет все эти строки в одну.

④ Этот фрагмент кода, получив список строк, возвращает все уникальные символы, встречающиеся во всех строках.

Наша программа для решения криптарифмов использует эту технику для получения множества всех уникальных букв, используемых в головоломке.

```
unique_characters = set(''.join(words))
```

## Проверка выполнения условий

Like many programming languages, Python has an assert statement. Here's how it works.

Подобно многим языкам программирования, Python имеет оператор подтверждения отсутствия ошибок. Вот как он работает.

```
>>> assert 1 + 1 == 2 ①
>>> assert 1 + 1 == 3 ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
>>> assert 2 + 2 == 5, "Only for very large values of 2" ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Only for very large values of 2
```

① За словом `assert` следует любое допустимое в Питоне выражение. В данном случае, выражение `1 + 1 == 2` возвращает значение `True`, поэтому `assert` ничего не делает.

② Однако если выражение возвращает `False`, `assert` выбрасывает исключение.

③ Вы также можете добавить информативное сообщение, которое будет выведено при возбуждении исключения `AssertionError`.

Поэтому, эта строка кода

```
assert len(unique_characters) <= 10, 'Too many letters'
```

эквивалентна

```
if len(unique_characters) > 10:
    raise AssertionError('Too many letters')
```

Программа для решения криптарифмов использует именно это выражение для того чтобы прекратить выполнение если в головоломке содержится более

десяти уникальных букв. Поскольку каждая буква соответствует цифре, а цифр всего десять, головоломка с более чем десятью уникальными буквами не может иметь решения.

## Выражения-генераторы

Выражения-генераторы, как генератор функций, но без функции.

```
>>> unique_characters = {'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
>>> gen = (ord(c) for c in unique_characters) ①
>>> gen ②
<generator object <genexpr> at 0x00BADC10>
>>> next(gen) ③
69
>>> next(gen)
68
>>> tuple(ord(c) for c in unique_characters) ④
(69, 68, 77, 79, 78, 83, 82, 89)
```

① Выражение-генератор, это как анонимная функция которая выдает значения. Само выражение выглядит как список, но он обернут не в квадратные, а в фигурные скобки.

② Выражение-генератор возвращает итератор.

③ Вызов `next(gen)` возвращает следующее значение итератора.

④ Если вам нравится, вы можете повторить через все возможные значения и вернуть кортеж, список или множество, направив выражение-генератор в `tuple()`, `list()`, или `set()`. В этих случаях вам не нужно дополнительных скобок - просто передайте "голые" выражение `ord(c) for c in unique_characters` в `tuple()` функцию, и Python понимает, что это выражение-генератор.



Использование выражений-генераторов вместо списка помогает сохранить `cpu` и `ram`. Если вы используете список, чтобы потом выбросить его (например передать в `tuple()` или `set()`), используйте генератор вместо него!

Вот еще один способ сделать то же самое, используя генератор функции:

```
def ord_map(a_string):
    for c in a_string:
        yield ord(c)
```

```
gen = ord_map(unique_characters)
```

Выражения-генераторы более компактны, но функционально равны.

# Тестирование

## (не) Погружение

(Данная страница находится на стадии перевода)

Современная молодежь. Избалованы быстрыми компьютерами и модными «динамическими» языками. Писать, потом предоставлять свой код, потом отлаживать (в лучшем случае). В наши дни была дисциплина. Я сказал, дисциплина! Нам приходилось писать программы вручную, на бумаге, и вводить их в компьютер на перфокартах. И нам это нравилось!

В этом разделе вы напишете и отладите набор вспомогательных функций для конвертирования в римскую систему и обратно. Вы видели, как конструируются и валидируются числа в римской системе в разделе «Учебный пример: Римские цифры». Вернемся немного назад и представим, как бы выглядела реализация в виде функции, производящей преобразование в обоих направлениях.

Правила формирования римских чисел приводят нас к нескольким интересным наблюдениям:

1. Существует только один правильный способ записать число римскими цифрами.
2. Обратное также верно: если строка символов является последовательностью римских символов, она представляет только одно число, то есть может быть интерпретирована единственным способом.
3. Диапазон чисел, которые могут быть записаны римскими цифрами, — от 1 до 3999. У римлян было несколько способов записывать более крупные числа, в частности, с помощью черты над числом, которая означала бы, что значение нужно умножить на 1000. Для целей этой главы нам достаточно ограничиться диапазоном 1 — 3999.
4. Нет способа представить 0 в римской системе.
5. Нет способа представить отрицательные числа в римской системе.
6. Нет способа представить дробные или нецелые числа в римской системе.

Попробуем отразить, что должен делать модуль `roman.py`. Он будет содержать две основные функции, `to_roman()` и `from_roman()`. Функция `to_roman()` должна

принимать целое число в диапазоне от 1 до 3999 и возвращать строку, содержащую римское представление этого числа...

Остановимся здесь. Давайте теперь сделаем кое-что неожиданное: опишем небольшой тестовый случай, который проверяет, работает ли функция `to_roman()` так, как мы этого ожидаем. Вы правильно поняли: мы собираемся написать код, который тестирует код, который еще не написан.

Это так называемая разработка через тестирование (*test-driven-development*, TDD). Набор из двух функций конвертации — `to_roman()`, `from_roman()` — может быть написан и протестирован как юнит, в отдельности от любой крупной программы, которая импортирует их. В Python'е есть фреймворк для юнит-тестирования, модуль с соответствующим названием `unittest`.

Юнит тестирование — важная часть всей стратегии разработки, основанной на тестировании. Если вы пишете юнит-тесты, необходимо писать их на ранних этапах и обновлять их по мере изменений кода и требований. Многие люди пропагандируют написание тестов до написания тестируемого кода, и именно этот подход я собираюсь продемонстрировать в этом разделе. Однако юнит-тесты выгодны вне зависимости от того, когда вы их пишете.

- До написания кода написание юнит тестов заставляет детализировать требования в удобном для их реализации виде
- Во время написания кода юнит тесты предохраняют вас от лишнего кодирования. Когда все тесты проходят, тестируемый юнит готов.
- Во время рефакторинга они помогают доказать, что новая версия ведет себя так же, как и старая.
- Во время поддержки кода существование юнит тестов прикроет вашу задницу, когда кто-то начнет кричать, что ваше последнее изменение сломало их код. («Но сэр, все тесты проходили успешно, когда я делал `commit`.»)
- Когда код пишется в команде, наличие всестороннего набора тестов значительно снижает риск того что ваш код сломает код другого разработчика, поскольку вы можете исполнить его юнит тесты. (Я видел как это работает на практике в `code sprints` (кодирование на скорость? :) ???). Команда разбивает задание, участники разбирают спецификации своих задач, пишут для них юнит тесты, затем обмениваются юнит тестами со всей командой. Так никто не зайдет слишком далеко в разработке кода который плохо пригоден для команды.)

## Единственный вопрос.

Один тестовый случай (*test case*) отвечает на один вопрос о тестируемом коде. Тестовый случай должен быть способен...

- ... запускаться самостоятельно, без ввода данных от человека. Юнит тестирование должно быть автоматизировано



- ... определять самостоятельно, прошла ли тестируемая функция тест или нет, без вмешательства человека с целью интерпретировать результаты
- ... запускаться в изоляции, отдельно от остальных тестовых случаев (даже если они тестируют те же функции)

Каждый тестовый случай — это остров.

Учитывая это, давайте составим тестовый случай (тест) для первого требования: 1. Функция `to_roman()` должна возвращать представление числа в римской системе счисления для всех чисел от 1 до 3999

Не сразу ясно, как этот скрипт делает... ну, хоть что-то. Он определяет класс, не содержащий метод `__init__()`. Класс содержит другой метод, который никогда не вызывается. Скрипт содержит блок `__main__`, но тот не ссылается на класс или его методы. Но кое-что он делает, поверьте мне.

```
import roman1
import unittest
class KnownValues(unittest.TestCase): ①
    known_values = ( (1, 'I'),
                     (2, 'II'),
                     (3, 'III'),
                     (4, 'IV'),
                     (5, 'V'),
                     (6, 'VI'),
                     (7, 'VII'),
                     (8, 'VIII'),
                     (9, 'IX'),
                     (10, 'X'),
                     (50, 'L'),
                     (100, 'C'),
                     (500, 'D'),
                     (1000, 'M'),
                     (31, 'XXXI'),
                     (148, 'CXLVIII'),
                     (294, 'CCXCIV'),
                     (312, 'CCCXII'),
                     (421, 'CDXXI'),
                     (528, 'DXXVIII'),
                     (621, 'DCXXI'),
                     (782, 'DCCLXXXII'),
                     (870, 'DCCCLXX'),
                     (941, 'CMXLI'),
                     (1043, 'MXLIII'),
```

```
(1110, 'MCX'),
(1226, 'MCCXXVI'),
(1301, 'MCCCI'),
(1485, 'MCDLXXXV'),
(1509, 'MDIX'),
(1607, 'MDCVII'),
(1754, 'MDCCLIV'),
(1832, 'MDCCCXXXII'),
(1993, 'MCMXCIII'),
(2074, 'MMLXXIV'),
(2152, 'MMCLII'),
(2212, 'MMCCXII'),
(2343, 'MMCCCXLIII'),
(2499, 'MMCDXCIX'),
(2574, 'MMDLXXIV'),
(2646, 'MMDCXLVI'),
(2723, 'MMDCCXXIII'),
(2892, 'MMDCCCXCII'),
(2975, 'MMCMLXXV'),
(3051, 'MMMLI'),
(3185, 'MMMCLXXXV'),
(3250, 'MMMCCCL'),
(3313, 'MMMCCCXIII'),
(3408, 'MMMCDVIII'),
(3501, 'MMMMDI'),
(3610, 'MMMDCX'),
(3743, 'MMMDCCXLIII'),
(3844, 'MMMDCCCXLIV'),
(3888, 'MMMDCCCLXXXVIII'),
(3940, 'MMMCMXL'),
(3999, 'MMMCMXCIX')) ②
```

```
def test_to_roman_known_values(self): ③
    """to_roman should give known result with known input"""
    for integer, numeral in self.known_values:
        result = roman1.to_roman(integer) ④
        self.assertEqual(numeral, result) ⑤
if __name__ == '__main__':
    unittest.main()
```

① Для описания тестового случая первым делом определим класс `TestCase` подкласс модуля `unittest`. Этот класс содержит много полезных методов, которые вы можете использовать в ваших тестах для определенных условий.

- ② Это множество пар "число/значение", определенных мной вручную. Оно включает минимальные 10 чисел, наибольшее (3999), все числа, которые в преобразованном виде состоят из одного символа, а также набор случайных чисел. Не нужно тестировать все возможные варианты, но все уникальные варианты протестировать нужно.
- ③ Каждый тест определен отдельным методом, который вызывается без параметров и не возвращает значения. Если метод завершается нормально, без выброса исключения - тест считается пройденным, если выброшено исключение - тест завален.
- ④ Здесь и происходит вызов тестируемой функции `to_roman()`. (Ну, функция еще не написана, но когда будет, это будет строка, которая ее вызовет.) Заметьте, что Вы только что определили интерфейс (API) функции `to_roman()`: она должна принимать число для конвертирования и возвращать строку (представление в виде Римского числа). Если API отличается от вышеуказанного, тест вернет ошибку. Также отметьте, что Вы не отлавливаете какие-либо исключения, когда вызываете `to_roman()`. Это сделано специально. `to_roman()` не должна возвращать исключение при вызове с правильными входными параметрами и правильными значениями этих параметров. Если `to_roman()` выбрасывает исключение, Тест считается проваленным.
- ⑤ Предполагая, что функция `to_roman()` определена корректно, вызвана корректно, выполнилась успешно, и вернула значение, последним шагом будет проверка правильности возвращенного значения. Это общий вопрос, поэтому используем метод `AssertEqual` класса `TestCase` для проверки равенства (эквивалентности) двух значений. Если возвращенный функцией `to_roman()` результат (`result`) не равен известному значению, которое Вы ожидаете (`numeral`), `assertEqual` выбросит исключение и тест завершится с ошибкой. Если значения эквиваленты, `assertEqual` ничего не сделает. Если каждое значение, возвращенное `to_roman()` совпадет с ожидаемым известным, `assertEqual` никогда не выбросит исключение, а значит `test_to_roman_known_values` в итоге выполнится нормально, что будет означать, что функция `to_roman()` успешно прошла тест.

Раз у Вас есть тест, Вы можете написать саму функцию `to_roman()`. Во-первых, Вам необходимо написать заглушку, пустую функцию и убедиться, что тест провалится. Если тест удачен, когда функция еще ничего не делает, значит тест не работает вообще! Unit testing это как танец: тест ведет, код следует. Пишете тест, который проваливается, потом - код, пока тест не пройдет.

```
# roman1.py
def to_roman(n):
    """convert integer to Roman numeral"""
    pass ①
```

- ① На этом этапе Вы определяете API для функции `to_roman()`, но пока не хотите писать ее код. (Для первой проверки теста.) Чтобы заглушить

функцию, используется зарезервированное слово Python - pass, которое... ничего не делает. Выполняем romantest1.py on в интерпретаторе для проверки теста. Если Вы вызвали скрипт с параметром -v, будет выведен подробности о работе скрипта (verbose), и Вы сможете подробно увидеть, что происходит в каждом тесте. Если повезло, увидите нечто подобное:

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues) ①
to_roman should give known result with known input ... FAIL ②
```

```
=====
FAIL: to_roman should give known result with known input
```

```
-----
Traceback (most recent call last):
```

```
File "romantest1.py", line 73, in test_to_roman_known_values
```

```
self.assertEqual(numeral, result)
```

```
AssertionError: 'I' != None ③
```

```
-----
Ran 1 test in 0.016s ④
```

```
FAILED (failures=1) ⑤
```

① Запущенный скрипт выполняет метод unittest.main(), который запускает каждый тестовый случай. Каждый тестовый случай - метод класса в romantest.py. Нет особых требований к организации этих классов; они могут быть как класс с методом для отдельного тестового случая, mfr и один класс + несколько методов для всех тестовых случаев. Необходимо лишь, чтобы каждый класс был наследником unittest.TestCase.

② Для каждого тестового случая модуль unittest выведет строку документации метода и результат - успех или провал. Как видно, тест провален.

③ Для каждого проваленного теста система выводит детальную информацию о том, что конкретно произошло. В данном случае вызов assertEquals() вызвал ошибку объявления (AssertionError), поскольку ожидалось возвращения 'I' от to\_roman(1), но этого не произошло. (Если у функции нет явного возврата, то она вернет None, значение null в Python.)

④ После детализации каждого тестового случая, unittest отображает суммарно, сколько тестов было выполнено и сколько это заняло времени.

⑤ В целом тест считается проваленным, если хоть один случай не пройден. Unittest различает ошибки и провалы. Провал вызывает метод assertXYZ, например assertEquals или assertRaises, который провалится, если объявленное условие неверно или ожидаемое исключение не выброшено. Ошибка - это другой тип исключения, который выбрасывается тестируемым кодом или тестовым юнитом и не является ожидаемым.

Наконец, мы можем написать функцию to\_roman().

```

roman_numeral_map = (('M', 1000),
                      ('CM', 900),
                      ('D', 500),
                      ('CD', 400),
                      ('C', 100),
                      ('XC', 90),
                      ('L', 50),
                      ('XL', 40),
                      ('X', 10),
                      ('IX', 9),
                      ('V', 5),
                      ('IV', 4),
                      ('I', 1)) ①
def to_roman(n):
    """convert integer to Roman numeral"""
    result = ""
    for numeral, integer in roman_numeral_map:
        while n >= integer: ②
            result += numeral
            n -= integer
    return result

```

① `roman_numeral_map` - это кортеж кортежей, определяющий три вещи: представление базовых символов римских цифр и популярных их сочетаний; порядок римских символов (в обратном направлении, от М и до I); значения римских цифр. Каждый внутренний кортеж - это пара значений (представление, число). И это не только односимвольные римские цифры; это также пары символов типа CM (“тысяча без сотни”). Это сильно упрощает код функции `to_roman()`.

② Вот здесь видно, в чем выигрыш такой структуры `roman_numeral_map`, поскольку не требуется какой-то хитрой логики при обработке вычитанием. Для конвертирования в римское число необходимо просто пройти в цикле `roman_numeral_map`, находя наименьшее число, в которое влезает остаток ввода. При нахождении такового, к возвращаемому значению функции добавляется соответствующее римское представление, ввод уменьшается на это число и далее операция повторяется для следующего кортежа.

Если все же не понятно, как работает функция `to_roman()`, добавим `print()` в конец цикла:

```

while n >= integer:
    result += numeral
    n -= integer
    print('subtracting {0} from input, adding {1} to output'.format(integer, numeral))

```

Этот отладочный вывод показывает следующее:

```
>>> import roman1
>>> roman1.to_roman(1424)
subtracting 1000 from input, adding M to output
subtracting 400 from input, adding CD to output
subtracting 10 from input, adding X to output
subtracting 10 from input, adding X to output
subtracting 4 from input, adding IV to output
'MCDXXIV'
```

Ну, функция `to_roman()` вроде бы работает, как и предполагалось в начале главы. Но пройдет ли она написанный ранее тест?

```
you@localhost:~/diveintopython3/examples$ python3 romantest1.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
```

```
-----
Ran 1 test in 0.016s
OK
```

1. Ура! Функция `to_roman()` прошла тест “known values”. Возможно не всесторонняя проверка, но в ее ходе проверены различные входные данные, включая числа, записываемые одним римским символом, наибольшее исходное значение (3999), и значение, дающее наибольшее римское число (3888). На этом этапе можно сказать, что функция корректно обрабатывает любые правильные исходные значения.

“Правильные” исходные значения? Хм. А как насчет неправильных?

### «Остановись и загорись»

Недостаточно проверить работу функции только с правильными входными данными; также необходимо убедиться, что функция выдаст ошибку при неправильном вводе. И не просто ошибку - а такую как ожидается.

```
>>> import roman1
>>> roman1.to_roman(4000)
'MMMM'
>>> roman1.to_roman(5000)
'MMMMM'
>>> roman1.to_roman(9000) ①
'MMMMMMMMMMM'
```

1. Это определенно не то, что ожидалось — это не правильные римские числа! По сути, все эти числа выходят за возможные пределы, но функция все равно возвращает результат, только фиктивный. Тихое возвращение неверного

значения - oooooooooочень неверно; если возникает ошибка, лучше чтобы программа завершалась быстро и шумно. "Остановись и загорись", как говорится. "Питоновский" способ остановиться и загореться - это выбросить исключение.

Спрашивается, как же учесть это в требованиях к тестированию? Для начинающих - вот так: функция `to_roman()` должна выбрасывать исключение типа `OutOfRangeError`, если ей передать число более 3999. Как будет выглядеть тест?

```
class ToRomanBadInput(unittest.TestCase): ①
    def test_too_large(self): ②
        """to_roman should fail with large input"""
        self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000) ③
```

1. Как и в предыдущем случае, создаем класс-наследник от `unittest.TestCase`. У Вас может быть более одного теста на класс (как Вы увидите дальше в этой главе), но я решил создать отдельный класс для этого, потому что этот случай отличается от предыдущих. Мы поместили все тесты на "положительный выход" в одном классе, а на ошибки - в другом.

2. Как и в предыдущем случае, тест - это метод, имя которого - название теста.

3. Класс `unittest.TestCase` предоставляет метод `assertRaises`, который принимает следующие аргументы: тип ожидаемого исключения, имя тестируемой функции и аргументы этой функции. (Если тестируемая функция принимает более одного аргумента, все они передаются методу `assertRaises` по порядку, как будто передаете их тестируемой функции.)

Обратите особое внимание на последнюю строку кода. Вместо вызова функции `to_roman()` и проверки вручную того, что она выбрасывает исключение (путем оберты в блок `try-catch`), метод `assertRaises` делает все это за нас. Все что Вы делаете - говорите, какой тип исключения ожидаете (`roman2.OutOfRangeError`), имя функции (`to_roman()`), и ее аргументы (4000). Метод `assertRaises` позаботится о вызове функции `to_roman()` и проверит, что она возвращает исключение `roman2.OutOfRangeError`.

Также заметьте, что Вы передаете функцию `to_roman()` как аргумент; Вы не вызываете ее и не передаете ее имя как строку. Кажись, я уже упоминал, что все в Python является объектом?

Что же происходит, когда Вы запускаете скрипт с новым тестом?

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ERROR ①
```

=====

ERROR: to\_roman should fail with large input

-----  
Traceback (most recent call last):

File "romantest2.py", line 78, in test\_too\_large  
self.assertRaises(roman2.OutOfRangeError, roman2.to\_roman, 4000)  
AttributeError: 'module' object has no attribute 'OutOfRangeError' ②

-----  
Ran 2 tests in 0.000s  
FAILED (errors=1)

1. Следовало ожидать этот провал (если конечно Вы не написали дополнительного кода), но... это не совсем "провал", скорее это ошибка. Это тонкое но очень важное различие. Тест может вернуть три состояния: успех, провал и ошибку. Успех, естественно, означает, что тест пройден — код делает что положено. «Провал» - то что вернул тест выше — код выполняется, но возвращает не ожидаемое значение. «Ошибка» означает, что Ваш код работает неправильно.

2. Почему код не выполняется правильно? Раскрутка стека все объясняет. Тестируемый модуль не выбрасывает исключение типа OutOfRangeError. То самое, которое мы скормили методу assertRaises(), потому что ожидаем его при вводе большого числа. Но исключение не выбрасывается, потому вызов метода assertRaises() провален. Без шансов - функция to\_roman() никогда не выбросит OutOfRangeError.

Решим эту проблему - определим класс исключения OutOfRangeError в roman2.py.

```
class OutOfRangeError(ValueError): ①
    pass ②
```

1. Исключения - это классы. Ошибка «out of range» это разновидность ошибки — аргумент выходит за допустимые пределы. Поэтому это исключение наследуется от исключения ValueError. Это не строго необходимо (по идее достаточно наследования от класса Exception), однако так правильно.

2. Исключение вообще-то ничего и не делает, но Вам нужна хотя бы одна строка в классе. Встроенная функция pass ничего не делает, однако необходима для минимального определения кода в Python.

Теперь запустим тест еще раз.

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... FAIL ①
```

=====



FAIL: to\_roman should fail with large input

---

Traceback (most recent call last):

```
File "romantest2.py", line 78, in test_too_large
    self.assertRaises(roman2.OutOfRangeError, roman2.to_roman, 4000)
AssertionError: OutOfRangeError not raised by to_roman ②
```

---

Ran 2 tests in 0.016s

FAILED (failures=1)

1. Тест по-прежнему не проходит, хотя уже и не выдает ошибку. Это прогресс! Значит, метод `assertRaises()` был выполнен и тест функции `to_roman()` был произведен.

2. Конечно, функция `to_roman()` не выбрасывает только что определенное исключение `OutOfRangeError`, так как Вы ее еще "не заставили". И это хорошие новости! Значит, тест работает, а проваливаться он будет, пока Вы не напишете условие его успешного прохождения.

Этим и займемся.

```
def to_roman(n):
    "convert integer to Roman numeral"
    if n > 3999:
        raise OutOfRangeError('number out of range (must be less than 4000)') ①
    result = ""
    for numeral, integer in roman_numeral_map:
        while n >= integer:
            result += numeral
            n -= integer
    return result
```

1. Все просто: если переданный параметр больше 3999, выбрасываем исключение `OutOfRangeError`. Тест не ищет текстовую строку, объясняющую причину исключения, хотя Вы можете написать тест для проверки этого (но учтите трудности, связанные с различными языками - длина строк или окружение могут отличаться).

Позволит ли это пройти тест? Узнаем:

```
you@localhost:~/diveintopython3/examples$ python3 romantest2.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok ①
```

---

Ran 2 tests in 0.000s  
OK

1. Ура! Оба теста пройдены. Так как Вы работали, переключаясь между кодированием и тестированием, то Вы с уверенностью можете сказать, что именно последние 2 строки кода позволили тесту вернуть "успех", а не "провал". Такая уверенность далась не дешево, но окупит себя с лихвой в дальнейшем.

## Больше СТОПов, больше "Огня"

Наряду с тестированием на слишком большой "ввод" необходимо протестировать и слишком маленький. Как было отмечено в требованиях к функциональности, римские цифры не могут быть меньше или равны 0.

```
>>> import roman2
>>> roman2.to_roman(0)
"
>>> roman2.to_roman(-1)
"
```

Не хорошо. Добавим тесты для каждого случая.

```
class ToRomanBadInput(unittest.TestCase):
    def test_too_large(self):
        """to_roman should fail with large input"""
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 4000) ①

    def test_zero(self):
        """to_roman should fail with 0 input"""
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0) ②

    def test_negative(self):
        """to_roman should fail with negative input"""
        self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1) ③
```

1. Метод `test_too_large()` не изменился. Я включил его сюда, чтобы показать схожесть кода.

2. Это новый тест: `test_zero()`. Как и `test_too_large()`, мы заставляем метод `assertRaises()`, определенный в `unittest.TestCase`, вызвать нашу функцию `to_roman()` с параметром "0", и проверить, что она выбрасывает соответствующее исключение, `OutOfRangeError`.

3. Метод `test_negative()` почти аналогичный, только передает -1 в функцию `to_roman()`. И ни один из этих методов не вернет ошибку `OutOfRangeError`

(потому что наша функция возвращает значение), и тест считается проваленным.

Теперь проверим, что тест провалится:

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... FAIL
```

```
=====
FAIL: to_roman should fail with negative input
```

```
-----
Traceback (most recent call last):
```

```
  File "romantest3.py", line 86, in test_negative
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, -1)
AssertionError: OutOfRangeError not raised by to_roman
```

```
=====
FAIL: to_roman should fail with 0 input
```

```
-----
Traceback (most recent call last):
```

```
  File "romantest3.py", line 82, in test_zero
    self.assertRaises(roman3.OutOfRangeError, roman3.to_roman, 0)
AssertionError: OutOfRangeError not raised by to_roman
```

```
-----
Ran 4 tests in 0.000s
```

```
FAILED (failures=2)
```

Великолепно. Оба теста провалены, как и ожидалось. А теперь обратимся к коду и посмотрим, что можно сделать для успешного прохождения теста.

```
def to_roman(n):
```

```
    """convert integer to Roman numeral"""
```

```
    if not (0 < n < 4000): ①
```

```
        raise OutOfRangeError('number out of range (must be 1..3999)') ②
```

```
    result = ""
```

```
    for numeral, integer in roman_numeral_map:
```

```
        while n >= integer:
```

```
            result += numeral
```

```
n -= integer
return result
```

1. Отличный пример сокращения Python: множественное сравнение в одну строку. Это эквивалентно выражению "Если не ((0 < n) и (n < 4000))", но читается проще. Этот однострочный код охватывает "плохой" диапазон входных данных.

2. Изменение условия требует изменения сообщения исключения. Тестовому фреймворку все равно, а вот при отладке вручную могут возникнуть трудности, если сообщение будет неправильно описывать ситуацию.

Я мог бы привести целый ряд примеров, чтобы показать, что однострочный код работает, но вместо этого я просто запущу тест.

```
you@localhost:~/diveintopython3/examples$ python3 romantest3.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok
```

```
-----
Ran 4 tests in 0.016s
OK
```

## И еще одна штука...

Еще одно требование к функциональности - обработка нецелых чисел.

```
>>> import roman3
>>> roman3.to_roman(0.5) ①
"
>>> roman3.to_roman(1.0) ②
'I'
```

1. О, это плохо.

2. О, а это еще хуже.

Оба случая должны выбросить исключение. Вместо этого функция возвращает ложное значение.

Тестирование не-чисел весьма сложно. Во-первых, определим исключение `NotIntegerError`.

```
# roman4.py
class OutOfRangeError(ValueError): pass
class NotIntegerError(ValueError): pass
```

Далее напишем тестовый случай для проверки выброса исключения NotIntegerError.

```
class ToRomanBadInput(unittest.TestCase):
    .
    .
    .
    def test_non_integer(self):
        """to_roman should fail with non-integer input"""
        self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
```

Убеждаемся, что тест провален.

```
you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... FAIL
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok
```

```
=====
FAIL: to_roman should fail with non-integer input
```

```
-----
Traceback (most recent call last):
```

```
  File "romantest4.py", line 90, in test_non_integer
    self.assertRaises(roman4.NotIntegerError, roman4.to_roman, 0.5)
AssertionError: NotIntegerError not raised by to_roman
```

```
-----
Ran 5 tests in 0.000s
```

```
FAILED (failures=1)
```

Пишем код для прохождения теста.

```
def to_roman(n):
    """convert integer to Roman numeral"""
    if not (0 < n < 4000):
        raise OutOfRangeError('number out of range (must be 1..3999)')
    if not isinstance(n, int): ①
```

```

    raise NotIntegerError('non-integers can not be converted') ②
result = ""
for numeral, integer in roman_numeral_map:
    while n >= integer:
        result += numeral
        n -= integer
return result

```

1. Встроенная функция `isinstance()` проверяет, принадлежит ли переменная определенному типу (точнее, технически - к наследнику типа).

2. Если аргумент `n` не число, выбрасываем наше новое исключение `NotIntegerError`.

Наконец, проверим код на тесте.

```

you@localhost:~/diveintopython3/examples$ python3 romantest4.py -v
test_to_roman_known_values (__main__.KnownValues)
to_roman should give known result with known input ... ok
test_negative (__main__.ToRomanBadInput)
to_roman should fail with negative input ... ok
test_non_integer (__main__.ToRomanBadInput)
to_roman should fail with non-integer input ... ok
test_too_large (__main__.ToRomanBadInput)
to_roman should fail with large input ... ok
test_zero (__main__.ToRomanBadInput)
to_roman should fail with 0 input ... ok

```

-----

Ran 5 tests in 0.000s

OK

Функция `to_roman()` успешно прошла все тесты, и больше тестов мне в голову не приходит, так что пора переходить к функции `from_roman()`.

## Приятная симметрия

Преобразование римского представления числа в десятичное выглядит более сложным, чем преобразование десятичной формы в римскую. Основная сложность заключается в валидации. Достаточно просто проверить, является ли целое число положительным; однако немного сложнее проверить, является ли строка корректным римским числом. К счастью, мы уже написали регулярное выражение, проверяющее римские числа.

Осталась задача преобразования самой строки. Как мы увидим через минуту, благодаря определенной нами структуре данных, ставящей в соответствие целым числам римские, реализация функции `from_roman()` является тривиальной задачей.

Но сначала тесты. Нам понадобятся известные значения для выборочной проверки правильности конвертирования. В качестве этих значений мы будем использовать описанный ранее набор `known_values`:

```
def test_from_roman_known_values(self):
    """from_roman should give known result with known input"""
    for integer, numeral in self.known_values:
        result = roman5.from_roman(numeral)
        self.assertEqual(integer, result)
```

Здесь мы наблюдаем интересную симметрию. Функции `to_roman()` и `from_roman()` являются взаимнообратными. Первая преобразует десятичное представление числа в римское, вторая же делает обратное преобразование. В теории мы должны иметь возможность "замкнуть круг", передав функции `to_roman()` число, затем передать результат выполнения функции `from_roman()`, возвращенное значение которой должно совпасть с исходным числом:

`n = from_roman(to_roman(n))` for all values of `n`

В этом случае "all values" означает любое число в интервале [1, 3999]. Напишем тест, который передает все числа из этого интервала функции `to_roman()`, затем вызывает `from_roman()` и проверяет соответствие результата исходному числу:

```
class RoundtripCheck(unittest.TestCase):
    def test_roundtrip(self):
        """from_roman(to_roman(n))==n for all n"""
        for integer in range(1, 4000):
            numeral = roman5.to_roman(integer)
            result = roman5.from_roman(numeral)
            self.assertEqual(integer, result)
```

Наши новые тесты пока не являются даже провальными - они завершились с ошибкой, так как мы еще не реализовали функцию `from_roman()`:

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
E.E....
```

```
=====
ERROR: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
-----
```

Traceback (most recent call last):

```
File "romantest5.py", line 78, in test_from_roman_known_values
    result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'
```

```
=====
```



```
ERROR: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
```

```
-----
Traceback (most recent call last):
File "romantest5.py", line 103, in test_roundtrip
result = roman5.from_roman(numeral)
AttributeError: 'module' object has no attribute 'from_roman'
```

```
-----
Ran 7 tests in 0.019s
FAILED (errors=2)
```

Создание заглушки функции решит эту проблему:

```
# roman5.py
def from_roman(s):
    """convert Roman numeral to integer"""
```

(Вы заметили? Я написал функцию, в которой нет ничего, кроме строки документации. Это нормально. Это Python. На самом деле, многие разработчики придерживаются именно такого стиля. “Не делай заглушек; документируй!”)

Теперь тесты действительно являются провальными:

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
F.F....
```

```
=====
FAIL: test_from_roman_known_values (__main__.KnownValues)
from_roman should give known result with known input
```

```
-----
Traceback (most recent call last):
File "romantest5.py", line 79, in test_from_roman_known_values
self.assertEqual(integer, result)
AssertionError: 1 != None
```

```
=====
FAIL: test_roundtrip (__main__.RoundtripCheck)
from_roman(to_roman(n))==n for all n
```

```
-----
Traceback (most recent call last):
File "romantest5.py", line 104, in test_roundtrip
self.assertEqual(integer, result)
AssertionError: 1 != None
```

```
-----
Ran 7 tests in 0.002s
FAILED (failures=2)
```

Теперь напомним функцию `from_roman()`:

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral: ①
            result += integer
            index += len(numeral)
    return result
```

Стиль написания здесь точно такой же, как и в функции `to_roman()`. Мы пробегаем все значения `roman_numeral_map`, но вместо того, чтобы брать максимальное целое число, пока это возможно, мы берем максимальное римское представление числа и ищем его в строке, пока это возможно.

Если вам еще не совсем понятно, как работает функция `from_roman()`, добавьте вывод в конце цикла:

```
def from_roman(s):
    """convert Roman numeral to integer"""
    result = 0
    index = 0
    for numeral, integer in roman_numeral_map:
        while s[index:index+len(numeral)] == numeral:
            result += integer
            index += len(numeral)
            print('found', numeral, 'of length', len(numeral), ', adding', integer)
>>> import roman5
>>> roman5.from_roman('MCMLXXII')
found M , of length 1, adding 1000
found CM of length 2, adding 900
found L of length 1, adding 50
found X of length 1, adding 10
found X of length 1, adding 10
found I of length 1, adding 1
found I of length 1, adding 1
1972
```

Перезапустим тесты:

```
you@localhost:~/diveintopython3/examples$ python3 romantest5.py
```

```
.....
```

```
-----
```

Ran 7 tests in 0.060s  
ОК

У меня есть для вас две новости. Обе хорошие. Во-первых, функция `from_roman()` работает для правильного ввода (по крайней мере, для известных значений); во-вторых, наш "круг" замкнулся. Эти два факта позволяют вам быть уверенным в том, что функции `to_roman()` и `from_roman()` работают правильно для всех корректных значений. (На самом деле, правильность работы не гарантирована. Теоретически, функция `to_roman()` может иметь баг в реализации, из-за которого получается неправильное представление числа в римской форме для некоторых входных данных, а функция `from_roman()` может иметь "обратный" баг, из-за которого результатом выполнения является число, по счастливой случайности совпадающее с исходным. Если вас это беспокоит, напишите более сложные тесты.)

## Больше плохих "вводов"

Now that the `from_roman()` function works properly with good input, it's time to fit in the last piece of the puzzle: making it work properly with bad input. That means finding a way to look at a string and determine if it's a valid Roman numeral. This is inherently more difficult than validating numeric input in the `to_roman()` function, but you have a powerful tool at your disposal: regular expressions. (If you're not familiar with regular expressions, now would be a good time to read the regular expressions chapter.) As you saw in Case Study: Roman Numerals, there are several simple rules for constructing a Roman numeral, using the letters M, D, C, L, X, V, and I. Let's review the rules: 1. Sometimes characters are additive. I is 1, II is 2, and III is 3. VI is 6 (literally, "5 and 1"), VII is 7, and VIII is 8. 2. The tens characters (I, X, C, and M) can be repeated up to three times. At 4, you need to subtract from the next highest fives character. You can't represent 4 as IIII; instead, it is represented as IV ("1 less than 5"). 40 is written as XL ("10 less than 50"), 41 as XLI, 42 as XLII, 43 as XLIII, and then 44 as XLIV ("10 less than 50, then 1 less than 5"). 3. Sometimes characters are... the opposite of additive. By putting certain characters before others, you subtract from the final value. For example, at 9, you need to subtract from the next highest tens character: 8 is VIII, but 9 is IX ("1 less than 10"), not VIIII (since the I character can not be repeated four times). 90 is XC, 900 is CM. 4. The fives characters can not be repeated. 10 is always represented as X, never as VV. 100 is always C, never LL. 5. Roman numerals are read left to right, so the order of characters matters very much. DC is 600; CD is a completely different number (400, "100 less than 500"). CI is 101; IC is not even a valid Roman numeral (because you can't subtract 1 directly from 100; you would need to write it as XCIX, "10 less than 100, then 1 less than 10"). Thus, one useful test would be to ensure that the `from_roman()` function should fail when you pass it a string with too many repeated numerals. How many is "too many" depends on the numeral.

```
class FromRomanBadInput(unittest.TestCase):
    def test_too_many_repeated_numerals(self):
        '''from_roman should fail with too many repeated numerals'''
        for s in ('MMMM', 'DD', 'CCCC', 'LL', 'XXXX', 'VV', 'IIII'):
            self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Another useful test would be to check that certain patterns aren't repeated. For example, IX is 9, but IXIX is never valid.

```
def test_repeated_pairs(self):
    '''from_roman should fail with repeated pairs of numerals'''
    for s in ('CMCM', 'CDCD', 'XCXC', 'XLXL', 'IXIX', 'IVIV'):
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

A third test could check that numerals appear in the correct order, from highest to lowest value. For example, CL is 150, but LC is never valid, because the numeral for 50 can never come before the numeral for 100. This test includes a randomly chosen set of invalid antecedents: I before M, V before X, and so on.

```
def test_malformed_antecedents(self):
    '''from_roman should fail with malformed antecedents'''
    for s in ('IIMXCC', 'VX', 'DCM', 'CMM', 'IXIV', 'MCMC', 'XCX', 'IVI', 'LM', 'LD', 'LC'):
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
```

Each of these tests relies the from\_roman() function raising a new exception, InvalidRomanNumeralError, which we haven't defined yet.

```
def test_malformed_antecedents(self):
# roman6.py
class InvalidRomanNumeralError(ValueError): pass
```

All three of these tests should fail, since the from\_roman() function doesn't currently have any validity checking. (If they don't fail now, then what the heck are they testing?)

```
you@localhost:~/diveintopython3/examples$ python3 romantest6.py
FFF.....
```

```
=====
FAIL: test_malformed_antecedents (__main__.FromRomanBadInput)
from_roman should fail with malformed antecedents
```

```
-----
Traceback (most recent call last):
```

```
File "romantest6.py", line 113, in test_malformed_antecedents
self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from_roman, s)
AssertionError: InvalidRomanNumeralError not raised by from_roman
```

```
=====
FAIL: test_repeated_pairs (__main__.FromRomanBadInput)
```

from\_roman should fail with repeated pairs of numerals

-----  
Traceback (most recent call last):

File "romantest6.py", line 107, in test\_repeated\_pairs

self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from\_roman, s)

AssertionError: InvalidRomanNumeralError not raised by from\_roman

=====

FAIL: test\_too\_many\_repeated\_numerals (\_\_main\_\_.FromRomanBadInput)

from\_roman should fail with too many repeated numerals

-----  
Traceback (most recent call last):

File "romantest6.py", line 102, in test\_too\_many\_repeated\_numerals

self.assertRaises(roman6.InvalidRomanNumeralError, roman6.from\_roman, s)

AssertionError: InvalidRomanNumeralError not raised by from\_roman

-----  
Ran 10 tests in 0.058s

FAILED (failures=3)

Good deal. Now, all we need to do is add the regular expression to test for valid Roman numerals into the from\_roman() function.

CD

И опять тестируем...

you@localhost:~/diveintopython3/examples\$ python3 romantest7.py

.....

-----  
Ran 10 tests in 0.066s

OK

И премия за главное разочарование в этом году достается... я так волнуюсь... слову "OK", выведенному как результат успешного прохождения теста.

---

# Рефакторинг

## Погружение

Нравится Вам или нет, но баги случаются. Несмотря на все усилия при создании полных модульных тестов, баги всё равно существуют. Что я подразумеваю под словом «баг»? Баг - это тестовый случай который ещё не написан.

```
>>> import roman7
>>> roman7.from_roman('') ①
0
```

① Собственно, баг. Вызов `from_roman` с параметром пустой строки (или любой другой последовательности символов не являющейся правильным римским числом) должен завершиться исключением `InvalidRomanNumeralError`.

После воспроизведения бага и до его фиксации следует написать тестовый случай завершающийся ошибкой, таким образом иллюстрируя баг.

```
class FromRomanBadInput(unittest.TestCase):
    .
    .
    .
    def testBlank(self):
        '''from_roman should fail with blank string'''
        self.assertRaises(roman6.InvalidRomanNumeralError, roman6.f
rom_roman, '') ①
```

① Всё предельно просто: вызываем `from_roman()` с пустой строкой и проверяем, что выбрасывается исключение `InvalidRomanNumeralError`. Самое сложное было найти баг; теперь, когда известно, что такая ошибка существует, кодирование проверки не займёт много времени.

Так как код содержит баг и мы имеем тест проверки этого бага, то данный тестовый случай завершается ошибкой:

```

you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... FAIL
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok

```

```

=====
FAIL: from_roman should fail with blank string
-----
Traceback (most recent call last):
  File "romantest8.py", line 117, in test_blank
    self.assertRaises(roman8.InvalidRomanNumeralError, roman8.from_
roman, '')
AssertionError: InvalidRomanNumeralError not raised by from_roman
-----

Ran 11 tests in 0.171s

FAILED (failures=1)

```

*Только теперь* Вы можете исправлять баг.

```

def from_roman(s):
    '''convert Roman numeral to integer'''

```



```

if not s:
    ①
        raise InvalidRomanNumeralError('Input can not be blank')
if not re.search(romanNumeralPattern, s):
    raise InvalidRomanNumeralError('Invalid Roman numeral: {}'.
format(s)) ②

result = 0
index = 0
for numeral, integer in romanNumeralMap:
    while s[index:index+len(numeral)] == numeral:
        result += integer
        index += len(numeral)
return result

```

① Требуется всего 2 строчки кода: явная проверка с пустой строкой и выброс исключения.

② Не уверен, упоминалось ли ранее в книге, поэтому пусть это будет последний трюк при форматировании строк. Начиная с Python 3.1 разрешается опускать числа при использовании индексов позиции в строке форматирования. То есть, вместо использования {0} для ссылки на первый параметр метода format(), Вы можете писать {} и Python заполнит соответствующий индекс позиции за Вас. Это выполняется для любого количества аргументов: первые скобки {} равносильны {0}, вторые скобки {} равносильны {1} и так далее.

```

you@localhost:~/diveintopython3/examples$ python3 romantest8.py -v
from_roman should fail with blank string ... ok ①
from_roman should fail with malformed antecedents ... ok
from_roman should fail with repeated pairs of numerals ... ok
from_roman should fail with too many repeated numerals ... ok
from_roman should give known result with known input ... ok
to_roman should give known result with known input ... ok
from_roman(to_roman(n))==n for all n ... ok
to_roman should fail with negative input ... ok

```

```
to_roman should fail with non-integer input ... ok
to_roman should fail with large input ... ok
to_roman should fail with 0 input ... ok
```

```
-----
Ran 11 tests in 0.156s
```

OK ②

- ① Тест на обработку пустой строки теперь проходит, значит баг исправлен.
- ② Все остальные тестовые случаи по прежнему выполняются без ошибок, а это значит, что при исправлении ошибки не мы не добавили новых. Самое время, чтобы остановиться править код!

Кодирование через написание тестов не облегчает процесс исправления багов. Для исправления простых ошибок (как в приведённом примере) необходимы простые тесты; сложные ошибки, конечно же, требуют сложных тестов. Если ведётся разработка проекта через тестирование, то *может показаться*, что фиксация бага займёт больше времени, так как Вам придётся найти строчки кода с багом (собственно, написать тестовый случай для проверки этих строчек) и затем исправить баг. Если тест опять завершается не успешно, то придётся разобраться верно ли был исправлен баг или сам тест содержит ошибки. Тем не менее, при длительной разработке эти исправления в коде-в тесте-в коде окупают себя, так как скорее всего баг будет исправлен с первого раза. Также, так как Вы можете легко перезапускать все тесты, включая новый, Вы навряд ли "испортите" старый код при фиксации бага. Сегодняшние модульные тесты завтра превратятся в тесты регрессии.

## Управляемся с изменением требований

### Рефакторинг

### Выводы

---

# Файлы

## 11. Файлы

**«девять миль ходьбы это не шутка, особенно во время дождя.»** —Harry Kemelman, The Nine Mile Walk

На моём Windows ноутбуке было 38493 файла, прежде чем я установил одно приложение. Установка Python 3 добавила почти 3000 файлов к общему объёму. Файлы представляют собой первичную парадигму хранения информации в основных операционных системах; эта концепция настолько укоренилась, что большинство людей не воспримут нечто другое альтернативное. Образно говоря, Ваш компьютер тонет в море файлов.

### 11.2 Чтение из текстовых файлов

Прежде чем читать из текстового файла, его требуется открыть. Открытие файла в Python легко выполнить:

```
a_file = open('examples/chinese.txt', encoding='utf-8')
```

Python имеет встроенную функцию `open()`, которой передается имя файла в качестве аргумента. В примере имя файла `'examples/chinese.txt'` и в нём есть 5 интересных вещей:

1. Это не просто имя файла, это комбинация пути к каталогу и имя файла. Гипотетически, в функцию открытия файла можно было бы передать два параметра: путь к файлу и имя файла, но в функцию `open()` можно передать только один. В Python, когда это необходимо вы можете включать все или некоторые пути к каталогу.
2. При указании пути к каталогу используется `/` (прямая обратная черта, слэш, правый слэш), не обсуждая какая операционная система используется. Windows использует `\` (обратную косую черту, обратный слэш, слэш влево) для указания пути к каталогам, а операционные системы Linux и MacOS используют `/` (прямая обратная черта, слэш, правый слэш). В Python прямой слэш просто работает всегда, даже на Windows.

3. Путь каталога не начинается с косой черты (слэша) или буквы, это называется относительным путем. Относительно чего? Имей терпение, кузнечик!
4. Это строки. Все современные операционные системы (включая Windows) используют Unicode для хранения имён файлов и директорий. Python 3 полностью поддерживает не-ascii пути.
5. Файл не обязательно должен находиться на локальных дисках. Вы можете использовать сетевые диски. Этот файл может быть объектом виртуальной файловой системы (/proc в linux). Если ваш компьютер считает это за файл и даёт возможность обращаться к этому как к файлу, то Python сможет открыть этот файл.

Вызов функции `open()` не ограничивается передачей параметра пути к файлу и его имени. Имеется ещё один параметр, называющийся `encoding`. О да, дорогой читатель, это звучит воистину ужасно!

### 11.2.1 Особенности кодировки показывают своё страшное лицо

Байты байт; символы абстракции. Строка представляет собой последовательность символов в кодировке Юникод. Но файлы на диске не являются последовательностью символов в кодировке Unicode, а являются последовательностью байтов. Если вы читаете текстовый файл с диска, то как Python преобразует эту последовательность байт в последовательность символов? Он декодирует байт по определенному алгоритму кодировки и возвращает последовательность символов в кодировке Unicode (т. е. в виде строки).

```
>> file = open('examples/chinese.txt')
...>>> a_string = file.read()
...Traceback (most recent call last):
... File «<stdin>», line 1, in <module>
... File «C:\Python31\lib\encodings\cp1252.py», line 23, in decode
... return codecs.charmap_decode(input, self.errors, decoding_table)[
0]
...UnicodeDecodeError: 'charmap' codec can't decode byte 0x8f in posi
tion 28: character maps to <undefined>
```

[[Категория:Погружение в Python 3]]

# XML

## Погружение

Большинство глав в этой книге строятся на отрывках, примерах кода. Но xml это больше данные, нежели код. Один из способов применения xml это «синдикация контента» такого, как последние статьи с блога, форума или других часто обновляемых сайтов. Большинство популярного ПО для ведения блогов может создавать ленты (фиды) и обновлять их, когда новые статьи, темы публикуются. Вы можете следить за блогом подписавшись на его канал, также вы можете следить за несколькими блогами при помощи «программ-агрегаторов» таких, как Google Reader [1]

Итак, ниже представлены XML данные с которыми мы будем работать в этой главе. Это фид формата Atom syndication feed

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  <subtitle>currently between addictions</subtitle>
  <id>tag:diveintomark.org,2001-07-29:/</id>
  <updated>2009-03-27T21:56:07Z</updated>
  <link rel='alternate' type='text/html' href='http://diveintomark.org/'/>
  <link rel='self' type='application/atom+xml' href='http://diveintomark.org/feed/'/>
  <entry>
    <author>
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Dive into history, 2009 edition</title>
    <link rel='alternate' type='text/html'
```

```

    href='http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition'/>
    <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>
    <updated>2009-03-27T21:56:07Z</updated>
    <published>2009-03-27T17:20:42Z</published>
    <category scheme='http://diveintomark.org' term='diveintopython' />
    <category scheme='http://diveintomark.org' term='docbook' />
    <category scheme='http://diveintomark.org' term='html' />
    <summary type='html'>Putting an entire chapter on one page sounds bloated, but consider this &mdash; my longest chapter so far would be 75 printed pages, and it loads in under 5 seconds&hellip;
    On dialup.</summary>
</entry>
<entry>
    <author>
        <name>Mark</name>
        <uri>http://diveintomark.org/</uri>
    </author>
    <title>Accessibility is a harsh mistress</title>
    <link rel='alternate' type='text/html'
        href='http://diveintomark.org/archives/2009/03/21/accessibility-is-a-harsh-mistress' />
    <id>tag:diveintomark.org,2009-03-21:/archives/20090321200928</id>
    <updated>2009-03-22T01:05:37Z</updated>
    <published>2009-03-21T20:09:28Z</published>
    <category scheme='http://diveintomark.org' term='accessibility' />

```

```

    <summary type='html'>The accessibility orthodoxy does not permit people to
        question the value of features that are rarely useful and rarely used.</summary>
</entry>
<entry>
    <author>
        <name>Mark</name>
    </author>
    <title>A gentle introduction to video encoding, part 1: container formats</title>
    <link rel='alternate' type='text/html'
        href='http://diveintomark.org/archives/2008/12/18/give-part-1-container-formats' />
    <id>tag:diveintomark.org,2008-12-18:/archives/20081218155422</id>
    <updated>2009-01-11T19:39:22Z</updated>
    <published>2008-12-18T15:54:22Z</published>
    <category scheme='http://diveintomark.org' term='asf' />
    <category scheme='http://diveintomark.org' term='avi' />
    <category scheme='http://diveintomark.org' term='encoding' />
    <category scheme='http://diveintomark.org' term='flv' />
    <category scheme='http://diveintomark.org' term='GIVE' />
    <category scheme='http://diveintomark.org' term='mp4' />
    <category scheme='http://diveintomark.org' term='ogg' />
    <category scheme='http://diveintomark.org' term='video' />
    <summary type='html'>These notes will eventually become part of
a
        tech talk on video encoding.</summary>
</entry>
</feed>

```

## 5-минутное введение в XML

Если Вы уже знакомы с XML, то можете пропустить эту главу.

XML — это язык разметки для описания иерархии структурированных данных. XML документ содержит один или более *элементов* разделённых *открывающими* и *закрывающими тегами*. Это правильный, хотя и неинтересный, XML документ:

```
<foo>    ①
</foo>   ②
```

① Это *открывающий (начальный)* тег элемента foo.

② Это соответствующий *закрывающий (конечный)* тег элемента foo. Как в математике и языках программирования каждая открывающая скобка должна иметь соответствующую закрывающую, в XML каждый открывающий тег должен быть *закрыт* соответствующим закрывающим.

Элементы могут быть неограниченно *вложены* друг в друга. Так как элемент bar вложен в элемент foo, то его называют *подэлементом* или *дочерним элементом* элемента foo.

```
<foo>
  <bar></bar>
</foo>
```

Первый элемент каждого XML документа называется корневым. XML документ может содержать только один корневой элемент. Пример представленный ниже **не является XML документом**, так как он имеет два корневых элемента:

```
<foo></foo>
<bar></bar>
```

Элементы могут иметь *атрибуты* состоящие из пары имя-значение. Атрибуты перечисляются внутри открывающего тега элемента и разделяются пробелами. [war-robin.com] *Имена атрибутов* не могут повторяться внутри одного элемента. *Значения атрибутов* должны быть обрамлены одинарными или двойными кавычками.

```
<foo lang='en'>                                ①
  <bar id='papayawhip' lang="fr"></bar>         ②
</foo>
```



- ① Элемент `foo` имеет один атрибут именованный как `lang`. Значению атрибута `lang` присваивается строка `en`.
- ② Элемент `bar` имеет два атрибута: `id` и `lang`. Значение `lang` есть `fr`. Это не приводит к конфликту с атрибутом `lang` элемента `foo`, так как каждый элемент имеет свой набор атрибутов.

Если элемент имеет больше чем один атрибут, то порядок атрибутов не играет роли. Атрибуты элементов есть неупорядоченный набор ключей и значений подобно словарям в Python. Для каждого элемента можно указать неограниченное число атрибутов.

Элементы могут иметь *текст (текстовое содержание)*.

```
<foo lang='en'>
  <bar lang='fr'>PapayaWhip</bar>
</foo>
```

Элементы которые не содержат текста и дочерних элементов называются *пустыми*.

```
<foo></foo>
```

Существует сокращённая запись пустого элемента. Поместив знак дроби / в конце открывающего тега, вы можете пропустить закрывающий тег. XML документ предыдущего примера с пустым элементов может быть записан следующим образом:

```
<foo/>
```

Подобно тому как функции Python могут быть объявлены в разных *модулях*, XML элементы могут быть объявлены в разных *пространствах имён (namespaces)*. Пространства имён обычно выглядят как URL-пути. Для объявления *пространства имён по умолчанию* используется директива `xmlns`. Объявление пространства имён очень похоже на атрибут, но имеет специальное значение.

```
<feed xmlns='http://www.w3.org/2005/Atom'> ①
  <title>dive into mark</title>              ②
</feed>
```

- ① Элемент `feed` находится в пространстве имён `http://www.w3.org/2005/Atom`.
- ② Элемент `title` также находится в пространстве имён `http://www.w3.org/2005/Atom`. Пространство имён применяется как к элементу в котором оно было определено так и ко всем дочерним элементам.

Вы можете объявлять пространство имён `xmlns:prefix` и ставить ему в соответствие *префикс* `prefix`. Тогда каждый элемент в данном пространстве имён должен быть явно объявлен с указанием префикса `prefix`.

```
<atom:feed xmlns:atom='http://www.w3.org/2005/Atom'> ①
  <atom:title>dive into mark</atom:title> ②
</atom:feed>
```

① Элемент `feed` находится в пространстве имён `http://www.w3.org/2005/Atom`.

② Элемент `title` также находится в пространстве имён `http://www.w3.org/2005/Atom`.

С точки зрения синтаксического анализатора XML, предыдущие два XML документа идентичны. Пара «пространство имён» + «имя элемента» задают XML идентичность. Префиксы используются только для ссылки на пространство имён, но не изменяют имени атрибута. Если пространства имён совпадают, имена элементов совпадают, атрибуты (или их отсутствие) совпадают и тексты элементов совпадают, то XML документы одинаковы.

И, наконец, XML документы могут содержать информацию о кодировке символов в первой строке до корневого элемента. (Если Вам интересно как документ может содержать информацию которая должна быть известна XML-анализатору до анализа XML документа, то смотрите Catch-22 раздел F XML спецификации)

```
<?xml version='1.0' encoding='utf-8'?>
```

Теперь Вы знаете об XML достаточно чтобы «вынести» следующие разделы главы!

## Структура формата синдикации фида Atom

Рассмотрим блог (weblog) или любой сайт с часто обновляемым контентом, например CNN.com. Сайт содержит заголовок («CNN.com»), подзаголовок («Breaking News, U.S., World, Weather, Entertainment & Video News»), дату последнего изменения («обновлено 12:43 p.m. EDT, Sat May 16, 2009») и список статей опубликованных в разное время. Каждая статья в свою очередь также имеет заголовок, дату первой публикации (и, возможно, дату последнего обновления, в случае если статья была корректирована) и уникальный URL.

Формат синдикации Atom разработан с целью хранить информацию подобного рода стандартным образом. Мой блог и CNN.com абсолютно разные по дизайну, содержанию и посетителям сайты, но оба имеют сходную структуру. Оба сайта имеют заголовки и публикуют статьи.

На верхнем уровне фид Atom должен иметь корневой элемент по имени feed находящийся в пространстве имен <http://www.w3.org/2005/Atom>.

```
<feed xmlns='http://www.w3.org/2005/Atom' ①
      xml:lang='en'> ②
```

① <http://www.w3.org/2005/Atom> - пространство имён Atom

② Каждый элемент может содержать атрибут xml:lang который определяет язык элемента и его дочерних элементов. В данном случае атрибут xml:lang объявленный в корневом элементе задаёт английский язык для всего фида.

Фид Atom содержит дополнительную информацию о себе в дочерних элементах корневого элемента:

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into mark</title>
  ①
  <subtitle>currently between addictions</subtitle>
  ②
  <id>tag:diveintomark.org,2001-07-29:/</id>
  ③
  <updated>2009-03-27T21:56:07Z</updated>
  ④
  <link rel='alternate' type='text/html' href='http://diveintomark.
  org/'/> ⑤
```

① Заголовок title содержит текст 'dive into mark'.

② Подзаголовок subtitle фида есть строка 'currently between addictions'.

③ Каждый фид должен иметь глобальный уникальный идентификатор. RFC 4151 содержит информацию как создавать такие идентификаторы.

④ Данный фид был обновлён последний раз 27 марта 2009 в 21:56 GMT. Обычно элемент updated эквивалентен дате последнего изменения какой-либо из статей на сайте.

⑤ А вот здесь начинается самое интересное. Элемент ссылки link не имеет текстового содержания, но имеет три атрибута: rel, type и href. Значение атрибута rel говорит о том какого типа ссылка. rel='alternate' значит, что это альтернативная ссылка этого фида. Атрибут type='text/html' говорит, что это ссылка на HTML страницу. И, собственно, путь ссылки содержится в атрибуте href.

Теперь мы знаем, что представленный выше фид получен с сайта «dive into mark». Сайт доступен по адресу <http://diveintomark.org/> и последний раз был обновлён 27 марта 2009.



Хотя в некоторых XML документах порядок элементов может иметь значение, в фидах Atom порядок элементов - произвольный.

Продолжим дальше рассматривать строение фида: после метаинформации о фиде идёт список последних статей. Статья выглядит следующим образом:

```
<entry>
  <author>
    ①
      <name>Mark</name>
      <uri>http://diveintomark.org/</uri>
    </author>
    <title>Dive into history, 2009 edition</title>
    ②
    <link rel='alternate' type='text/html'
    ③
      href='http://diveintomark.org/archives/2009/03/27/dive-into-his-
      tory-2009-edition' />
    <id>tag:diveintomark.org,2009-03-27:/archives/20090327172042</id>
    ④
    <updated>2009-03-27T21:56:07Z</updated>
    ⑤
    <published>2009-03-27T17:20:42Z</published>
    <category scheme='http://diveintomark.org' term='diveintopython' /
    >
      ⑥
    <category scheme='http://diveintomark.org' term='docbook' />
    <category scheme='http://diveintomark.org' term='html' />
    <summary type='html'>Putting an entire chapter on one page sounds
    ⑦
      bloated, but consider this &mdash; my longest chapter so fa
      r
      would be 75 printed pages, and it loads in under 5 seconds&
      hellip;
      On dialup.</summary>
```

```
</entry>
```

⑧

- ① Элемент `author` сообщает о том, кто написал статью: некоторый парень по имени Марк (Mark), который валяет дурака на сайте <http://diveintomark.org/> (В данном случае ссылка на сайт автора совпадает с альтернативной ссылкой в метаинформации о фиде, но это не всегда правда, так как многие блоги имеют несколько авторов, у каждого из которых — свой сайт.)
- ② Элемент `title` содержит заголовок статьи «Dive into history, 2009 edition».
- ③ Как и с альтернативной ссылкой на фид, в элементе `link` находится адрес HTML версии данной статьи.
- ④ Элемент `entry`, подобно фидам, имеет уникальный идентификатор.
- ⑤ Элемент `entry` имеет две даты: дату первой публикации и дату последнего изменения.
- ⑥ Элементы `entry` могут иметь произвольное количество категорий `category`. Рассматриваемая статья попадёт в категории `diveintopython`, `docbook` и `html`.
- ⑦ Элемент `summary` даёт краткий обзор статьи. (Бывает также не представленный здесь элемент содержания `content` предназначенный для включения в фид полного текста статьи.) Данный элемент `summary` содержит специфичный для фидов Atom атрибут `type='html'` указывающий что содержимое элемента есть текст в формате HTML. Это важно, так как HTML-объекты `&mdash;` и `&hellip;` присутствующие в элементе должны отображаться как «—» и «...», а не печататься «как есть».
- ⑧ И, наконец, закрывающий тег элемента `entry` говорит о конце метаданных для этой статьи.

## Синтаксический разбор XML

В Python документы XML могут быть обработаны с использованием разных библиотек. Язык имеет обычные синтаксические анализаторы DOM и SAX, но я буду использовать другую библиотеку `ElementTree`.

```
>>> import xml.etree.ElementTree as etree ①
>>> tree = etree.parse('examples/feed.xml') ②
>>> root = tree.getroot() ③
>>> root ④
<Element {http://www.w3.org/2005/Atom}feed at cd1eb0>
```

- ① Модуль `ElementTree` входит в стандартную библиотеку Python, путь для импорта `xml.etree.ElementTree`.
- ② Функция `parse()` — это базовая функция модуля `ElementTree`. Функция принимает имя файла или файлоподобный объект. Эта функция выполняет синтаксический анализ документа за раз. Если разрабатываемая программа должна экономить память, то можно анализировать XML документ частями.
- ③ Функция `parse()` возвращает объект, который является представлением всего документа. Однако объект `tree` не является корневым элементом. Чтобы получить ссылку на корневой элемент, необходимо вызвать метод `getroot()`.
- ④ Как и следовало ожидать, корневой элемент есть элемент фида в пространстве имён `http://www.w3.org/2005/Atom`. Строковое представление объекта `root` ещё раз подчёркивает важный момент: XML элемент — это комбинация пространства имён и его имени-тега (так же называемого локальным именем). Каждый элемент в данном документе находится в пространстве `Atom`, поэтому корневой элемент представлен как `{http://www.w3.org/2005/Atom}feed`.



Модуль `ElementTree` всегда представляет элементы XML как '{пространство имён}локальное имя'. Вам неоднократно предстоит использовать этот формат при использовании API `ElementTree`.

## Элементы XML есть списки Python

В API `ElementTree` элементы представляются встроенным типом Python - списком. Каждый из элементов списка представляет собой дочерние XML элементы.

```
# продолжение предыдущего примера
>>> root.tag                                ①
'{http://www.w3.org/2005/Atom}feed'
>>> len(root)                                ②
8
>>> for child in root:                        ③
...     print(child)                          ④
...
<Element {http://www.w3.org/2005/Atom}title at e2b5d0>
<Element {http://www.w3.org/2005/Atom}subtitle at e2b4e0>
<Element {http://www.w3.org/2005/Atom}id at e2b6c0>
<Element {http://www.w3.org/2005/Atom}updated at e2b6f0>
```

```
<Element {http://www.w3.org/2005/Atom}link at e2b4b0>
<Element {http://www.w3.org/2005/Atom}entry at e2b720>
<Element {http://www.w3.org/2005/Atom}entry at e2b510>
<Element {http://www.w3.org/2005/Atom}entry at e2b750>
```

- ① Продолжим предыдущий пример: корневой элемент root - {http://www.w3.org/2005/Atom}feed
- ② "Длина" корневого элемента есть количество дочерних элементов root.
- ③ Вы можете использовать элемент как итератор по всем дочерним элементам.
- ④ Из сообщений видно, что в элементе root 8 дочерних элементов: 5 элементов с метаданной о фиде (title, subtitle, id, updated и link) и 3 элемента со статьями entry.

Вы, должно быть, уже догадались, но я хочу явно указать на следующее: список дочерних элементов содержит только прямые дочерние элементы. В свою очередь каждый дочерний элемент entry может содержать свои дочерние элементы, но они не будут включены в список. Они будут включены в список элемента entry, а не в список подэлементов элемента feed. Найти определённые элементы любого уровня вложенности можно несколькими способами; ниже мы рассмотрим 2 из них.

## Атрибуты XML есть словари Python

Напомним, что документ XML это не только набор элементов; каждый элемент так же имеет набор атрибутов. Имея конкретный XML элемент, Вы можете легко получить его атрибуты как словарь Python.

```
# продолжение предыдущего примера
>>> root.attrib                                ①
{'{http://www.w3.org/XML/1998/namespace}lang': 'en'}
>>> root[4]                                    ②
<Element {http://www.w3.org/2005/Atom}link at e181b0>
>>> root[4].attrib                              ③
{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}
>>> root[3]                                    ④
<Element {http://www.w3.org/2005/Atom}updated at e2b4e0>
```

```
>>> root[3].attrib
{}
⑤
```

① Свойство `attrib` возвращает словарь атрибутов элемента. Исходная разметка XML была следующая `<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>`. Префикс `xml:` ссылается на стандартное пространство имён, которое любой XML документ может использовать без объявления.

② Пятый подэлемент есть элемент `link` (используется индекс `[4]`, так как списки Python индексируются начиная с 0).

③ Подэлемент `link` имеет три атрибута `href`, `type` и `rel`.

④ Четвёртый подэлемент (с индексом `[3]` в списке начинающемся с 0) — это элемент `updated`.

⑤ Подэлемент `updated` не имеет атрибутов, следовательно свойство `.attrib` возвращает пустой словарь.

## Поиск узлов в XML документе

До настоящего момента мы рассматривали XML документ «сверху вниз», начиная с корневого элемента, далее к его дочерним элементам и так вглубь всего документа. Однако во многих случаях при работе с XML Вам необходимо искать конкретные элементы. Etree справится и с этой задачей.

```
>>> import xml.etree.ElementTree as etree
>>> tree = etree.parse('examples/feed.xml')
>>> root = tree.getroot()
>>> root.findall('{http://www.w3.org/2005/Atom}entry')    ①
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
>>> root.tag
'{http://www.w3.org/2005/Atom}feed'
>>> root.findall('{http://www.w3.org/2005/Atom}feed')    ②
[]
>>> root.findall('{http://www.w3.org/2005/Atom}author')   ③
[]
```



- ① Метод `findall()` выполняет поиск дочерних элементов удовлетворяющих запросу. (Формат запроса рассматривается ниже.)
- ② Все элементы (включая корневой и дочерние) имеют метод `findall()`. Метод находит все элементы среди дочерних соответствующие запросу. Почему же метод вернул пустой список? Хотя это может показаться неочевидным, данный запрос ищет только в дочерних элементах. Так как корневой элемент `feed` не имеет дочерних элементов по имени `feed`, то запрос возвращает пустой список.
- ③ Этот результат также может Вас удивить. В документе XML действительно есть элемент `author`; на самом деле, их даже три (по одному в каждом элементе `entry`). Но эти элементы `author` не являются *прямыми подэлементами* (*direct children*) корневого элемента; они — «подподэлементы» (подэлементы подэлемента). Если Вам нужно найти элементы `author` любого уровня вложенности, то придётся изменить строку запроса.

```
>>> tree.findall('{http://www.w3.org/2005/Atom}entry')    ①
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
>>> tree.findall('{http://www.w3.org/2005/Atom}author')    ②
[]
```

① Для удобства объект `tree` (который возвращает функция `etree.parse()`) имеет несколько методов идентичных методам корневого элемента. Результаты функции такие же как при вызове метода `tree.getroot().findall()`.

② Наверное, удивлены, однако этот запрос не находит элемента `author` в данном документе. Почему же? Потому что, этот вызов идентичен вызову `tree.getroot().findall('{http://www.w3.org/2005/Atom}author')`, что значит «найти все элементы `author`, которые являются подэлементами корневого элемента». Элементы `author` не являются дочерними для корневого элемента; они подэлементы элементов `entry`. Таким образом, при выполнении запроса совпадений не найдено.

Помимо метода `findall()` есть метод `find()` который возвращает только первый найденный элемент. Метод может быть полезен в случаях когда в результате поиска Вы ожидаете только один элемент или Вам важен только первый элемент из списка найденных.

```
>>> entries = tree.findall('{http://www.w3.org/2005/Atom}entry')
①
>>> len(entries)
```

3

```
>>> title_element = entries[0].find('{http://www.w3.org/2005/Atom}title') ②
>>> title_element.text
'Dive into history, 2009 edition'
>>> foo_element = entries[0].find('{http://www.w3.org/2005/Atom}foo')
>>> foo_element
>>> type(foo_element)
<class 'NoneType'>
```

① Как Вы видели в предыдущем примере `findall()` возвращает список элементов `atom:entry`.

② Метод `find()` принимает запрос `ElementTree` и возвращает первый удовлетворяющий запросу элемент.

③ Во элементе `foo` отсутствуют дочерние элементы, поэтому `find()` возвращает объект `None`.



Здесь необходимо отметить заковыку при использовании метода `find()`. В логическом контексте объекты элементов `ElementTree` не содержащие дочерних элементов равны значению `False` (т.е. `if len(element)` вычисляется как 0). Код `if element.find('...')` проверяет не то, что нашёл ли метод `find()` удовлетворяющий запросу элемент; код проверяет содержит ли найденный элемент дочерние элементы! Для того чтобы проверить нашёл ли метод `find()` элемент необходимо использовать `if element.find('...') is not None`.

Рассмотрим поиск внутри дочерних элементов, т.е. подэлементов, подподэлементов и так далее любого уровня вложенности.

```
>>> all_links = tree.findall('//{http://www.w3.org/2005/Atom}link')
>>> all_links
[<Element {http://www.w3.org/2005/Atom}link at e181b0>,
 <Element {http://www.w3.org/2005/Atom}link at e2b570>,
 <Element {http://www.w3.org/2005/Atom}link at e2b480>,
 <Element {http://www.w3.org/2005/Atom}link at e2b5a0>]
>>> all_links[0].attrib
```

```
{'href': 'http://diveintomark.org/',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[1].attrib
③
{'href': 'http://diveintomark.org/archives/2009/03/27/dive-into-his-
tory-2009-edition',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[2].attrib
{'href': 'http://diveintomark.org/archives/2009/03/21/accessibility-
-is-a-harsh-mistress',
 'type': 'text/html',
 'rel': 'alternate'}
>>> all_links[3].attrib
{'href': 'http://diveintomark.org/archives/2008/12/18/give-part-1-c
ontainer-formats',
 'type': 'text/html',
 'rel': 'alternate'}
```

① Этот запрос — `//{http://www.w3.org/2005/Atom}link` — очень похож на запросы из предыдущих примеров. Отличие заключается в двух символах косой черты `//` в начале строки запроса. Символы `//` обозначают «Я хочу найти все элементы независимо от уровня вложенности, а не только непосредственные дочерние элементы». Поэтому метод возвращает список из четырёх элементов, а не из одного.

② Первый элемент результата — прямой подэлемент корневого элемента. Как мы видим из его атрибутов, это альтернативная ссылка уровня фид, которая указывает на html версию вебсайта на котором располагается фид.

③ Остальные три элемента результата есть альтернативные ссылки уровня элементов `entry`. Каждый из элементов `entry` имеет по одному подэлементу `link`. Так как запрос `findall()` содержал символы двойной черты в начале запроса, то результат поиска содержит все подэлементы `link`.

В целом, метод `findall()` библиотеки `ElementTree` довольно мощный инструмент поиска, однако формат запроса может быть немного непредсказуем. Официально формат запросов `ElementTree` описан как «ограниченная поддержка выражений XPath». XPath это стандарт организации W3C для построения

запросов поиска внутри XML документа. С одной стороны формат запросов ElementTree достаточно похож на формат XPath для выполнения простейших поисков. С другой стороны он отличается настолько, что может начать раздражать если Вы уже знаете XPath. Далее мы рассмотрим сторонние библиотеки XML позволяющие расширить API ElementTree до полной поддержки стандарта XPath.

## Работаем с lxml

lxml это сторонняя библиотека с открытым кодом основанная на известном синтаксическом анализаторе libxml2. Библиотека обеспечивает стопроцентную совместимость с API ElementTree, полностью поддерживает XPath 1.0 и имеет несколько других приятных фишек. Для Windows можно скачать инсталлятор; пользователям Linux следует проверить наличие скомпилированных пакетов в репозиториях дистрибутива (например, используя инструменты yum или apt-get). В противном случае придётся устанавливать lxml вручную.

```
>>> from lxml import etree ①
>>> tree = etree.parse('examples/feed.xml') ②
>>> root = tree.getroot() ③
>>> root.findall('{http://www.w3.org/2005/Atom}entry') ④
[<Element {http://www.w3.org/2005/Atom}entry at e2b4e0>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b510>,
 <Element {http://www.w3.org/2005/Atom}entry at e2b540>]
```

① При импорте lxml предоставляет абсолютно такой же API как встроенная библиотека ElementTree.

② Функция parse(): такая же как в ElementTree.

③ Метод getroot(): такой же.

④ Метод findall(): точно такой же.

При обработке больших XML документов lxml значительно быстрее чем встроенная библиотека ElementTree. Если Вы используете функции только из API ElementTree и хотите чтобы обработка выполнялась как можно быстрее, то можно попробовать импортировать библиотеку lxml и, в случае её отсутствия, использовать ElementTree.

```
try:
    from lxml import etree
except ImportError:
```

```
import xml.etree.ElementTree as etree
```

Однако, lxml не только быстрее чем ElementTree: метод findall() поддерживает более сложные запросы.

```
>>> import lxml.etree
①
>>> tree = lxml.etree.parse('examples/feed.xml')
>>> tree.findall('//{http://www.w3.org/2005/Atom}*[@href]')
②
[<Element {http://www.w3.org/2005/Atom}link at eeb8a0>,
 <Element {http://www.w3.org/2005/Atom}link at eeb990>,
 <Element {http://www.w3.org/2005/Atom}link at eeb960>,
 <Element {http://www.w3.org/2005/Atom}link at eeb9c0>]
>>> tree.findall("//{http://www.w3.org/2005/Atom}*[@href='http://di
veintomark.org/']") ③
[<Element {http://www.w3.org/2005/Atom}link at eeb930>]
>>> NS = '{http://www.w3.org/2005/Atom}'
>>> tree.findall('//{NS}author[{NS}uri]'.format(NS=NS))
④
[<Element {http://www.w3.org/2005/Atom}author at eebe80>,
 <Element {http://www.w3.org/2005/Atom}author at eebeba0>]
```

① В этом примере я импортирую объект lxml.etree (вместо объекта etree: from lxml import etree) чтобы подчеркнуть, что описываемые возможности реализуемы только с lxml.

② Этот запрос найдёт все элементы в пространстве имён Atom (любой вложенности), которые имеют атрибут href. Символы // в начале запроса обозначают «элементы любой вложенности, а не только потомки корневого элемента». {http://www.w3.org/2005/Atom} обозначает «только элементы пространства имён Atom». Символ \* значит «элементы с любым локальным именем». И [@href] обозначает «элемент имеет атрибут href».

③ В результате запроса найдены все элементы Atom с атрибутом href равным http://diveintomark.org/.

④ После преобразования строки (иначе эти запросы становятся невероятно длинны) данный запрос ищет элементы Atom author имеющие подэлементы Atom uri. Запрос возвращает только 2 элемента author: в первом и во втором

элементах entry. В последнем элементе entry элемент author содержит только имя name, но не uri.

Вам мало? lxml имеет встроенную поддержку для выражений XPath 1.0. Мы не будем детально рассматривать синтаксис XPath, так как это тема для отдельной книги. Однако мы рассмотрим пример использования XPath в lxml.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed.xml')
>>> NSMAP = {'atom': 'http://www.w3.org/2005/Atom'}
①
>>> entries = tree.xpath("//atom:category[@term='accessibility']/..",
②
...     namespaces=NSMAP)
>>> entries
③
[<Element {http://www.w3.org/2005/Atom}entry at e2b630>]
>>> entry = entries[0]
>>> entry.xpath('./atom:title/text()', namespaces=NSMAP)
④
['Accessibility is a harsh mistress']
```

① Чтобы выполнить XPath запрос элементов из пространства имён, необходимо определить отображение префикса этого пространства. На самом деле это обычный словарь Python.

② А вот и XPath запрос. Данное выражение выполняет поиск элементов category (пространства имён Atom) содержащие атрибут с парой имя-значение term='accessibility'. Но это не совсем то, что возвращает запрос. Вы заметили символы /.. в конце строки запроса? Это обозначает «верни не найденный элемент, а его родителя». И так, одним запросом мы найдём все элементы entry с дочерними элементами <category term='accessibility'>.

③ Функция xpath() возвращает список объектов ElementTree. В анализируемом документе всего один элемент entry с атрибутом term='accessibility'.

④ Выражение XPath не всегда возвращает список элементов. Формально, DOM разобранного документа XML не содержит элементов, она содержит узлы (nodes). В зависимости от их типа узлы могут быть элементами, атрибутами или даже текстом. Результатом запроса XPath всегда является список узлов. Этот запрос возвращает список текстовых узлов: текст text() элемента title (atom:title) есть подэлемент текущего элемента (/).

## Создание XML

ElementTree умеет не только разбирать существующие XML документы, но и создавать их «с нуля».

```
>>> import xml.etree.ElementTree as etree
>>> new_feed = etree.Element('{http://www.w3.org/2005/Atom}feed',
①
...     attrib={'{http://www.w3.org/XML/1998/namespace}lang': 'en'})
②
>>> print(etree.tostring(new_feed))
③
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

① Для создания нового элемента необходимо создать объект класса Element. В качестве первого параметра в конструктор мы передаём имя элемента (пространство имён и локальное имя). Данное выражение создаёт элемент feed в пространстве Atom. Этот будет корневой элемент нашего нового документа XML.

② Для того чтобы добавить атрибуты к создаваемому элементу мы передаём словарь имён атрибутов и их значений в втором аргументе attrib. Заметьте, что имена атрибутов должны задаваться в формате ElementTree {пространство\_имён}локальное\_имя.

③ В любой момент Вы можете сериализовать элемент и его подэлементы используя функцию tostring() библиотеки ElementTree.

Вы удивлены результату сериализации new\_feed? Формально ElementTree сериализует XML элементы правильно, но не оптимально. Пример XML документа в начале главы определён в *пространстве по умолчанию* xmlns='http://www.w3.org/2005/Atom'. Определение пространства по умолчанию полезно для документов (например, фидов Atom), где все элементы принадлежат одному пространству, то есть Вы можете объявить пространство один раз, а на элементы ссылаться используя локальное имя (<feed>, <link>, <entry>). Если Вы не собираетесь объявлять элементы из другого пространства имён, то нет необходимости использовать префикс пространства по умолчанию.

Синтаксический анализатор XML не «замечит» разницы между документом XML с пространством по умолчанию и документом с использованием префикса пространства имён перед каждым элементом. Результирующая модель DOM данной сериализации выглядит как

```
<ns0:feed xmlns:ns0='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

что равнозначно

```
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

Единственная разница в том, что второй вариант на несколько символов короче. Если мы переделаем наш пример с использованием префикса ns0: в каждом открывающем и закрывающем тэгах, это добавило бы 4 символа на открывающий тэг × 79 тэгов + 4 символа на объявление собственно пространства имён, всего 320 символов. В кодировке UTF-8 это составило бы 320 байт. (После архивации gzip разница уменьшается до 21 байта; однако 21 байт это 21 байт). Возможно, Вы бы не обратили внимания на эти десятки байтов, но для фидов Atom, которые загружаются тысячу раз при изменении, выигрыш нескольких байт на одном запросе быстро превращается в килобайты.

Ещё одно преимущество lxml: в отличие от стандартной библиотеки ElementTree lxml предоставляет более тонкое управление сериализацией элементов.

```
>>> import lxml.etree
>>> NSMAP = {None: 'http://www.w3.org/2005/Atom'}
①
>>> new_feed = lxml.etree.Element('feed', nsmap=NSMAP)
②
>>> print(lxml.etree.tounicode(new_feed))
③
<feed xmlns='http://www.w3.org/2005/Atom'/>
>>> new_feed.set('{http://www.w3.org/XML/1998/namespace}lang', 'en')
④
>>> print(lxml.etree.tounicode(new_feed))
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'/>
```

① Для начала определим пространство имён используя словарь. Значения словаря и есть пространство имён; ключи словаря - задаваемый префикс. Используя объект None в качестве префикса мы задаём пространство имён по умолчанию.

② При создании элемента мы передаём специфичный для lxml аргумент nsmap, используемый для передачи префиксов пространств имён.

③ Как и ожидали, при сериализации определено пространство имён по умолчанию Atom и объявлен один элемент feed без префикса пространства имён.



④ Опа, мы забыли добавить атрибут `xml:lang`. Используя метод `set()`, можно всегда добавить атрибут к любому элементу. Метод принимает два аргумента: имя атрибута в стандартном формате `ElementTree` и значение атрибута. (Данный метод есть и в библиотеке `ElementTree`. Единственное отличие `lxml` и `ElementTree` в данном примере это передача аргумента `nsmap` для указания префиксов пространств имён.)

Разве наши документы ограничены только одним элементом? Конечно, нет. Мы можем запросто создать дочерние элементы.

```
>>> title = lxml.etree.SubElement(new_feed, 'title',           ①
...     attrib={'type':'html'})                                ②
>>> print(lxml.etree.tounicode(new_feed))                     ③
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'><title type
='html' /></feed>
>>> title.text = 'dive into &hellip;'                          ④
>>> print(lxml.etree.tounicode(new_feed))                     ⑤
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'><title type
='html'>dive into &hellip;</title></feed>
>>> print(lxml.etree.tounicode(new_feed, pretty_print=True))  ⑥
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
<title type='html'>dive into&hellip;</title>
</feed>
```

① Для создания подэлемента существующего элемента необходимо создать объект класса `SubElement`. В конструктор класса передаются элемент родителя (в данном случае `new_feed`) и имя нового элемента. Мы не объявляем заново пространство имён для создаваемого потомка, так как он наследует пространство имён от родителя.

② Также мы передаём словарь с атрибутами для элемента. В качестве имён атрибутов выступают ключи словаря, в качестве значений атрибутов - значения словаря.

③ Неудивительно, что новый элемент `title` был создан в пространстве `Atom` и является подэлементом элемента `feed`. Так как элемент `title` не имеет текстового содержания и подэлементов, то `lxml` сериализует его как пустой элемент и закрывает символами `/>`.

④ Для того чтобы добавить текстовое содержание, мы задаём свойство `.text`.

⑤ Теперь элемент `title` сериализуется с только что заданным текстовым содержанием. Если в тексте содержатся знаки «меньше чем» `<` или

«амперсанд» ', то при сериализации они должны быть экранированы escape-последовательностью. Такие ситуации lxml обрабатывает автоматически.

⑥ При сериализации Вы можете применить «приятную печать» («pretty printing»), при которой вставляется разрыв строки после закрывающего тэга или открывающего тэга элементов с подэлементами но без текстового содержания. С технической точки зрения lxml добавляет незначащие пробелы и переносы строк («insignificant whitespace») чтобы вывести XML более читаемым.



Вам, возможно, будет интересно попробовать ещё одну стороннюю библиотеку `xmlwitch`, которая повсеместно использует оператор Python `with` для того чтобы сделать код создания XML более читаемым.

## Синтаксический разбор нецелых XML

XML спецификация предписывает, что все XML синтаксические анализаторы должны выполнять «драконову (строгую) обработку ошибок». То есть, при обнаружении в XML документе формальной ошибки или не«правильнопостроенности» (*wellformedness*) анализаторы должны сразу же прервать анализ и «вспыхнуть». Ошибки правильнопостроенности включают несогласованность открывающих и закрывающих тэгов, неопределённые элементы, неправильные символы Юникод и другие эзотерические ситуации. Такая обработка ошибок сильно контрастирует на фоне других известных форматов, например, HTML — браузер не останавливается отрисовывать web-страницу если в странице забыт закрывающий HTML тэг или значение атрибута тэга содержит неэкранированный амперсанд. (Существует распространённое заблуждение, что в формате HTML не оговорена обработка ошибок. На самом деле, обработка HTML ошибок отлично документирована, но она гораздо сложнее чем просто «остановиться и загореться на первой ошибке».)

Некоторые считают (и я в том числе), что это было ошибкой со стороны разработчиков формата XML заставлять так строго обрабатывать ошибки. Не поймите меня неправильно, я конечно же за упрощение правил обработки ошибок. Однако, на практике понятие «правильнопостроенности» оказывается коварнее чем кажется, особенно для XML документов которые публикуются в интернете и передаются по протоколу HTTP (например, фиды Atom). Несмотря на зрелость XML, который стандартизовал драконову обработку ошибок в 1997, исследования постоянно показывают, что значительная часть фидов Atom в интернете содержат ошибки правильнопостроенности.

Итак, у меня есть и теоретические и практические причины обрабатывать XML документы «любой ценой», то есть не останавливаться и взрываться при первой ошибке. Если Вы окажетесь в похожей ситуации, то lxml может помочь.

Ниже приведён фрагмент «битого» XML документа.

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into &hellip;</title>
  ...
</feed>
```

В фиде ошибка, так как последовательность &hellip; не определена в формате XML (она определена в HTML). Если попробовать разобрать битый фид с настройками по умолчанию, то lxml споткнётся на неопределённом вхождении hellip.

```
>>> import lxml.etree
>>> tree = lxml.etree.parse('examples/feed-broken.xml')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "lxml.etree.pyx", line 2693, in lxml.etree.parse (src/lxml/lxml.etree.c:52591)
    File "parser.pxi", line 1478, in lxml.etree._parseDocument (src/lxml/lxml.etree.c:75665)
    File "parser.pxi", line 1507, in lxml.etree._parseDocumentFromURL (src/lxml/lxml.etree.c:75993)
    File "parser.pxi", line 1407, in lxml.etree._parseDocFromFile (src/lxml/lxml.etree.c:75002)
    File "parser.pxi", line 965, in lxml.etree._BaseParser._parseDocFromFile (src/lxml/lxml.etree.c:72023)
    File "parser.pxi", line 539, in lxml.etree._ParserContext._handleParseResultDoc (src/lxml/lxml.etree.c:67830)
    File "parser.pxi", line 625, in lxml.etree._handleParseResult (src/lxml/lxml.etree.c:68877)
    File "parser.pxi", line 565, in lxml.etree._raiseParseError (src/lxml/lxml.etree.c:68125)
lxml.etree.XMLSyntaxError: Entity 'hellip' not defined, line 3, column 28
```

Для того чтобы обрабатывать XML документ с ошибками, необходимо создать новый синтаксический анализатор XML.

```

>> parser = lxml.etree.XMLParser(recover=True) ①
>>> tree = lxml.etree.parse('examples/feed-broken.xml', parser) ②
>>> parser.error_log ③
examples/feed-broken.xml:3:28:FATAL:PARSER:ERR_UNDECLARED_ENTITY: Entity 'hellip' not defined
>>> tree.findall('{http://www.w3.org/2005/Atom}title')
[<Element {http://www.w3.org/2005/Atom}title at ead510>]
>>> title = tree.findall('{http://www.w3.org/2005/Atom}title')[0]
>>> title.text ④
'dive into '
>>> print(lxml.etree.tounicode(tree.getroot())) ⑤
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en'>
  <title>dive into </title>
.
. [остальной вывод сериализации пропущен для краткости]
.

```

① Для того чтобы создать новый анализатор мы создаём новый класс `lxml.etree.XMLParser`. Хотя он может принимать много разных параметров, для нас представляет интерес только один — аргумент восстановления `recover`. При присвоении аргументу значения `True` `lxml` будет из кожи вон лезть чтобы восстановить ошибки правильнопостроенности.

② Для того чтобы разобрать XML документ новым анализатором мы передаём объект `parser` в качестве второго аргумента в функцию `parse()`. На этот раз `lxml` не выбрасывает исключительную ситуацию при неопределённой последовательности `&hellip;`.

③ Анализатор содержит сообщения обо всех найденных ошибках. (На самом деле эти сообщения сохраняются независимо от параметра `recover`.)

④ Так как анализатор не знает что делать с неопределённым `&hellip;`, то он просто выбрасывает слово. Текстовое содержание элемента `title` превращается в `'dive into '`.

⑤ И ещё раз: после сериализации последовательность `&hellip;` исчезла, `lxml` её выбросил.

Важно отметить, что **нет никакой гарантии переносимости восстановления ошибок** у XML анализаторов. Другой анализатор может быть умнее и распознать что `&hellip;` является валидной последовательностью HTML и

восстановить её как амперсанд. «Лучше» ли это? Возможно. Является ли это «более правильным»? Нет, так как оба решения с точки зрения формата XML неверны. Правильное поведение (согласно XML спецификации) прекратить обработку и загореться. Если же необходимо не следовать спецификации, то Вы делаете это на свой страх и риск.

## Материалы для дальнейшего чтения

[XML на Википедии](#)

[The ElementTree XML API\(англ.\)](#)

[Elements and Element Trees - Элементы и деревья элементов\(англ.\)](#)

[XPath Support in ElementTree - Поддержка XPath в ElementTree\(англ.\)](#)

[The ElementTree iterparse Function - Функция iterparse в ElementTree\(англ.\)](#)

[lxml\(англ.\)](#)

[Parsing XML and HTML with lxml - обработка XML и HTML в lxml\(англ.\)](#)

[XPath and XSLT with lxml - XPath и XSLT в lxml\(англ.\)](#)

[xmlwitch\(англ.\)](#)

---

# Сериализация объектов Python

## Погружение

С первого взгляда, идея сериализации проста. У вас есть структура данных в памяти, которую вы хотите сохранить, использовать повторно, или отправить кому либо. Как вам это сделать? Это зависит от того как вы ее сохраните, как вы ее хотите использовать, и кому вы ее хотите отправить. Многие игры позволяют вам сохранять ваш прогресс перед выходом и возобновлять игру после запуска. (Вообще, многие неигровые приложения также позволяют это делать). В этом случае, структура, которая хранит ваш прогресс в игре, должна быть сохранена на диске, когда вы закрываете игру, и загружена с диска, когда вы ее запускаете. Данные предназначены только для использования той же программой что и создала их, никогда не посылаются по сети, и никогда не читаются ничем кроме программы их создавшей. Поэтому проблемы совместимости ограничены тем, чтобы более поздние версии программы могли читать данные созданные ранними версиями.

Для таких случаев модуль `pickle` идеален. Это часть стандартной библиотеки Python, поэтому он всегда доступен. Он быстрый, большая часть написана на C, как и сам интерпретатор Python. Он может сохранять совершенно произвольные комплексные структуры данных Python.

Что может сохранять модуль `pickle`?

- Все встроенные типы данных Python: тип `boolean`, `Integer`, числа с плавающей точкой, комплексные числа, строки, объекты `bytes`, массивы байт, и `None`.

- Списки, кортежи, словари и множества, содержащие любую комбинацию встроенных типов данных
- Списки, кортежи, словари и множества, содержащие любую комбинацию списков, кортежей, словарей и множеств содержащий любую комбинацию встроенных типов данных (и так далее, вплоть до максимального уровня вложенности, который поддерживает Python).
- Функции, классы и экземпляры классов (с caveats).

Если для вас этого мало, то модуль `pickle` еще и расширяем. Если вам интересна эта возможность, то смотрите ссылки в разделе «Дальнейшее чтение» в конце этой главы.

### Маленькая заметка о примерах в этой главе.

Эта часть повествует о двух Python консолях. Все примеры в этой главе — часть одной большей истории. Вам нужно будет переключаться назад и вперед между двумя консолями для демонстрации модулей `pickle` и `json`.

Для того чтобы не запутаться откройте консоль Python и определите следующую переменную:

```
>>> shell = 1
```

Оставьте это окно открытым. И откройте еще одну консоль Python и определите следующую переменную:

```
>>> shell = 2
```

В этой главе я буду использовать переменную `shell` для того чтобы показать какую именно консоль Python я использую в каждом примере.

### Сохранение данных в файл Pickle.

Модуль `Pickle` работает со структурами данных. Давайте создадим одну.

```
>>> shell 1
>>> entry = {}
>>> entry['title'] = 'Dive into history, 2009 edition'
>>> entry['article_link'] = 'http://diveintomark.org/archives/2009/03/27/dive-into-
history-2009-edition'
>>> entry['comments_link'] = None
>>> entry['internal_id'] = b'\xDE\xD5\xB4\xF8'
>>> entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> entry['published'] = True
>>> import time
>>> entry['published_date'] = time.strptime('Fri Mar 27 22:20:42 2009')
>>> entry['published_date'] = time.struct_time(tm_year=2009, tm_mon=3,
```

```
tm_mday=27, tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86,
tm_isdst=-1)
```

① Все дальнейшее происходит в консоли Python #1.

② Идея в том чтобы создать словарь, который будет представлять что-нибудь полезное, например элемент рассылки Atom. Также я хочу быть уверенным, что он содержит несколько разных типов данных, чтобы раскрыть возможности модуля pickle. Не вчитывайтесь слишком сильно в эти переменные.

③ Модуль time содержит структуру данных (struct\_time) для представления момента времени (вплоть до миллисекунд) и функции для работы с этими структурами. Функция strptime() принимает на вход форматированную строку и преобразует ее в struct\_time. Эта строка в стандартном формате, но вы можете контролировать ее при помощи кодов форматирования. Для более подробного описания загляните в модуль time.

Теперь у нас есть замечательный словарь. Давайте сохраним его в файл.

```
>>> shell                                ①
1
>>> import pickle
>>> with open('entry.pickle', 'wb') as f:  ②
...     pickle.dump(entry, f)             ③
...
```

① Мы все еще в первой консоли

② Используйте функцию open() для того чтобы открыть файл. Установим режим работы с файлом в 'wb' для того чтобы открыть файл для записи в двоичном режиме. Оборнем его в конструкцию with для того чтобы быть уверенным в том что файл закроется автоматически, когда вы завершите работу с ним.

③ Функция dump() модуля pickle принимает сериализуемую структуру данных Python, сериализует ее в двоичный, Python-зависимый формат использует последнюю версию протокола pickle и сохраняет ее в открытый файл.

Последнее предложение было очень важным.

- Протокол pickle зависит от Python; здесь нет гарантий совместимости с другими языками. Вы возможно не сможете взять entry.pickle файл, который только что сделали и как — либо с пользой его использовать при помощи Perl, PHP, Java или любого другого языка программирования
- Не всякая структура данных Python может быть сериализована модулем Pickle. Протокол pickle менялся несколько раз с добавлением новых типов данных в язык Python, и все еще у него есть ограничения.
- Как результат, нет гарантии совместимости между разными версиями Python. Новые версии Python поддерживают старые форматы



сериализации, но старые версии Python не поддерживают новые форматы (поскольку не поддерживают новые форматы данных)

- Пока вы не укажете иное, функции модуля `pickle` будут использовать последнюю версию протокола `pickle`. Это сделано для уверенности в том, что вы имеете наибольшую гибкость в типах данных, которые вы можете сериализовать, но это также значит, что результирующий файл будет невозможно прочитать при помощи старых версий Python, которые не поддерживают последнюю версию протокола `pickle`.
- Последняя версия протокола `pickle` это двоичный формат. Убедитесь, что открываете файлы `pickle` в двоичном режиме, или данные будут повреждены при записи.

## Загрузка данных из файла `pickle`.

Теперь переключитесь во вторую консоль Python — т. е. не в ту где вы создали словарь `entry`.

```
>>> shell ①
2
>>> entry ②
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import pickle
>>> with open('entry.pickle', 'rb') as f: ③
...     entry = pickle.load(f) ④
...
>>> entry ⑤
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link':
 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27,
 tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
```

① Это вторая консоль Python

② Здесь не определена переменная `entry`. Вы определяли переменную `entry` в первой консоли Python, но это полностью отличное окружение со своим собственным состоянием.

③ Откроем entry.pickle файл, который вы создали в первой консоли Python. Модуль pickle использует двоичный формат представления данных, поэтому вам всегда нужно открывать файл в двоичном режиме.

④ Функция pickle.load() принимает на вход поток, читает сериализованные данные из потока, создает новый объект Python, восстанавливает сериализованные данные в новый объект Python, и возвращает новый объект Python.

⑤ Теперь переменная entry — это словарь со знакомыми ключами и значениями.

Результат цикла pickle.dump()/pickle.load() это новая структура данных эквивалентная оригинальной структуре данных.

```
>>> shell                                ①
1
>>> with open('entry.pickle', 'rb') as f: ②
...     entry2 = pickle.load(f)           ③
...
>>> entry2 == entry                       ④
True
>>> entry2 is entry                       ⑤
False
>>> entry2['tags']                        ⑥
('diveintopython', 'docbook', 'html')
>>> entry2['internal_id']
b'\xDE\xD5\xB4\xF8'
```

① Переключитесь обратно в первую консоль Python.

② Откройте entry.pickle файл

③ Загрузите сериализованные данные в новую переменную entry2

④ Python подтверждает, что эти два словаря(entry и entry2) эквивалентны. В этой консоли вы создали entry с нуля, начиная с пустого словаря вручную присваивая значения ключам. Вы сериализовали этот словарь и сохранили в файле entry.pickle. Теперь вы считали сериализованные данные из этого файла и создали совершенную копию оригинальной структуры.

⑤ Эквивалентность не значит идентичности. Я сказал, что вы создали \_идеальную копию\_ оригинальной структуры данных, и это правда. Но это все же копия.

⑥ По причинам которые станут ясны в дальнейшем, я хочу указать, что значения ключа 'tags' это кортеж, и значение 'internal\_id' это объект bytes.



Много статей о модуле Pickle ссылаются на cPickle. В Python 2 существует две реализации модуля pickle одна написана на чистом Python а другая на C(но все же вызываемая из Python). В Python 3 эти два модуля были объединены поэтому вам следует всегда использовать `import pickle`. Вам могут быть полезны эти статьи но следует игнорировать устаревшую информацию о cPickle.

## Используем Pickle без файлов

Пример из предыдущей секции показал как сериализовать объект напрямую в файл на диске. Но что если он вам не нужен или вы не хотели использовать файл? Вы можете сериализовать в объект `bytes` в памяти.

```
>>> shell
1
>>> b = pickle.dumps(entry)    ①
>>> type(b)                    ②
<class 'bytes'>
>>> entry3 = pickle.loads(b)    ③
>>> entry3 == entry            ④
True
```

① Функция `pickle.dumps()` (обратите внимание на 's' в конце имени функции) делает ту же самую сериализацию что и функция `pickle.dump()`. Вместо того чтобы принимать на вход поток и писать сериализованные данные на диск, она просто возвращает сериализованные данные

② Поскольку протокол pickle использует двоичный формат данных, функция `pickle.dumps()` возвращает объект типа `bytes`.

③ Функция `pickle.loads()` (снова заметьте 's' в конце имени функции) делает ту же самую десериализацию что и функция `pickle.load()`. Но вместо того чтобы принимать на вход поток и читать сериализованные данные из файла, она принимает на вход объект типа `bytes` содержащий сериализованные данные, такие как возвращаемые функцией `pickle.dumps()`

④ Конечный результат таков же: идеальная копия оригинального словаря.

## Байты и строки снова вздымают свои уродливые головы

Протокол pickle существует уже много лет, и он развивался вместе с тем как развивался сам Python. Сейчас существует четыре различных версии протокола pickle.

- Python 1.x породил две версии протокола, основанный на тексте формат (версия 0) и двоичный формат (версия 1)

- Python 2.3 ввел новый протокол pickle(версия 2) для того чтобы поддерживать новый функционал в классах Python. Он двоичный.
- Python 3.0 ввел еще один протокол pickle(версия 3) с полной поддержкой объектов типа bytes и массивов байт. Он так же двоичный.

Важно отметить, разница между строками и байтами снова вздымает свою уродливую голову. (Если вы удивлены, вы не уделяли достаточно внимания.) На практике это значит, что в то время, как Python 3 может читать данные сохраненные при помощи протокола версии 2, Python 2 не может читать данные сохраненные при помощи протокола версии 3.

## Отладка файлов pickle

Как выглядит протокол pickle? Давайте ненадолго отложим консоль python и взглянем в файл entry.pickle, который мы создали. Для не вооруженного взгляда он выглядит как тарабарщина.

```
you@localhost:~/diveintopython3/examples$ ls -l entry.pickle
-rw-r--r-- 1 you you 358 Aug 3 13:34 entry.pickle
you@localhost:~/diveintopython3/examples$ cat entry.pickle
comments_linkqNXtagsqXdiveintopythonqXdocbookqXhtmlq?qX publishedq?
XlinkXJhttp://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition
q Xpublished_dateq
ctime
struct_time
?qRqXtitleqXDive into history, 2009 editionqu.
```

Не слишком то полезно. Вы можете видеть строки, но остальные типы данных выглядят как непечатаемые (или как минимум не читаемые) символы. Поля даже не разделены хотя бы табуляцией или пробелами. Это не тот формат, который вы бы захотели отлаживать вручную.

```
>>> shell
1
>>> import pickletools
>>> with open('entry.pickle', 'rb') as f:
...     pickletools.dis(f)
0: \x80 PROTO      3
2: }  EMPTY_DICT
3: q  BININPUT    0
5: (  MARK
6: X      BINUNICODE 'published_date'
25: q      BININPUT    1
27: c      GLOBAL      'time struct_time'
45: q      BININPUT    2
```

```

47: ( MARK
48: M BININT2 2009
51: K BININT1 3
53: K BININT1 27
55: K BININT1 22
57: K BININT1 20
59: K BININT1 42
61: K BININT1 4
63: K BININT1 86
65: J BININT -1
70: t TUPLE (MARK at 47)
71: q BINPUT 3
73: } EMPTY_DICT
74: q BINPUT 4
76: \x86 TUPLE2
77: q BINPUT 5
79: R REDUCE
80: q BINPUT 6
82: X BINUNICODE 'comments_link'
100: q BINPUT 7
102: N NONE
103: X BINUNICODE 'internal_id'
119: q BINPUT 8
121: C SHORT_BINBYTES 'þŒ'ø'
127: q BINPUT 9
129: X BINUNICODE 'tags'
138: q BINPUT 10
140: X BINUNICODE 'diveintopython'
159: q BINPUT 11
161: X BINUNICODE 'docbook'
173: q BINPUT 12
175: X BINUNICODE 'html'
184: q BINPUT 13
186: \x87 TUPLE3
187: q BINPUT 14
189: X BINUNICODE 'title'
199: q BINPUT 15
201: X BINUNICODE 'Dive into history, 2009 edition'
237: q BINPUT 16
239: X BINUNICODE 'article_link'
256: q BINPUT 17
258: X BINUNICODE 'http://diveintomark.org/archives/2009/03/27/dive-into-

```

```

history-2009-edition'
337: q      BINPUT    18
339: X      BINUNICODE 'published'
353: q      BINPUT    19
355: \x88   NEWTRUE
356: u      SETITEMS  (MARK at 5)
357: .      STOP
highest protocol among opcodes = 3

```

Самая интересная часть информации в дизассемблере находится на последней строке, потому что она включает версию протокола, при помощи которого данный файл был сохранен. Не существует явного маркера протокола pickle. Чтобы определить какую версию протокола использовали для сохранения файла Pickle, вам необходимо заглянуть в маркеры («opcodes») внутри сохраненных данных и использовать вшитую информацию о том какие маркеры были введены, в какой версии протокола Pickle. Функция `pickletools.dis()` делает именно это, и она печатает результат в последней строке дизассемблированного вывода. Вот функция, которая возвращает только номер версии, без вывода данных:

```

import pickletools

def protocol_version(file_object):
    maxproto = -1
    for opcode, arg, pos in pickletools.genops(file_object):
        maxproto = max(maxproto, opcode.proto)
    return maxproto

```

И вот она же в действии:

```

>>> import pickleversion
>>> with open('entry.pickle', 'rb') as f:
...     v = pickleversion.protocol_version(f)
>>> v
3

```

## Сериализация объектов Python для чтения при помощи других языков

Формат данных используемый модулем pickle Python-зависимый. Он не пытается быть совместимым с другими языками программирования. Если межязыковая совместимость есть среди ваших потребностей, вам следует присмотреться к форматам сериализации. Один из таких форматов JSON. «JSON» это аббревиатура от «JavaScript Object Notation», но не позволяйте

имени обмануть вас — JSON был наверняка разработан для использования многими языками программирования.

Python 3 включает модуль `json` в стандартную библиотеку. Как и модуль `pickle`, модуль `json` имеет функции для сериализации структур данных, сохранения сериализованных данных на диск, загрузки сериализованных данных с диска, и десериализации данных обратно в новый объект Python. Так же существует несколько важных различий. Первое, формат данных `json` текстовый, а не двоичный. RFC 4627 определяет формат `json` и то, как различные типы данных должны быть преобразованы в текст. Например, логическое значение сохраняется как пяти символьная строка `'false'` или четырех символьная строка `'true'`. Все значения в `json` регистрочувствительные.

Во — вторых, как и с любым текстовым форматом, существует проблема пробелов. JSON позволяет вставлять произвольное количество пробелов (табуляций, переводов строк, и пустых строк) между значениями. Пробелы в нем «незначачие», что значит, кодировщики JSON могут добавлять так много или так мало пробелов как захотят, и декодировщики JSON будут игнорировать пробелы между значениями. Это позволяет вам использовать красивый вывод (`pretty-print`) для отображения ваших данных в формате JSON, удобно отображать вложенные значения различными уровнями отступа так чтобы вы могли читать все в стандартном просмотрщике или текстовом редакторе. Модуль `json` в Python имеет опции красивого вывода во время кодирования данных.

В — третьих, существует многолетняя проблема кодировок. JSON хранит значения как обычный текст, но, как вы знаете, не существует таких вещей как «обычный текст». JSON должен быть сохранен в кодировке Unicode (UTF-32, UTF-16, или стандартной UTF-8), и секция 3 из RFC 4627 определяет то, как указать используемую кодировку.

## Сохранение данных в файл JSON

JSON выглядит удивительно похожим на структуру данных, которую вы могли бы определить вручную в JavaScript. Это не случайно, вы действительно можете использовать функцию `eval()` из JavaScript чтобы «декодировать» данные сериализованные в `json`. (Обычные протесты против не доверенного ввода принимаются, но дело в том, что `json` это корректный JavaScript). По существу, JSON может быть уже хорошо знаком вам.

```
>>> shell
1
>>> basic_entry = {}
>>> basic_entry['id'] = 256
>>> basic_entry['title'] = 'Dive into history, 2009 edition'
>>> basic_entry['tags'] = ('diveintopython', 'docbook', 'html')
>>> basic_entry['published'] = True
```

①

```
>>> basic_entry['comments_link'] = None
>>> import json
>>> with open('basic.json', mode='w', encoding='utf-8') as f: ②
...     json.dump(basic_entry, f) ③
```

① Мы собираемся создать новую структуру данных вместо того чтобы использовать уже имеющуюся структуру данных entry. Позже в этой главе мы увидим что случится, когда мы попробуем кодировать более общую структуру данных в JSON.

② JSON это текстовый формат, это значит, что вы должны открыть файл в текстовом режиме и указать кодировку. Вы никогда не ошибетесь, используя UTF-8.

③ Как и модуль pickle, модуль json определяет функцию dump() которая принимает на вход структуру данных Python и поток для записи. Функция dump() сериализует структуру данных Python и записывает ее в объект потока. Раз мы делаем это в конструкции with, мы можем быть уверенными, что файл будет корректно закрыт, когда мы завершим работу с ним.

Ну и как выглядит результат сериализации в формат json?

```
you@localhost:~/diveintopython3/examples$ cat basic.json
{"published": true, "tags": ["diveintopython", "docbook", "html"], "comments_link":
null,
"id": 256, "title": "Dive into history, 2009 edition"}
```

Это несомненно, намного более читаемо, чем файл pickle. Но json может содержать произвольное количество пробелов между значениями, и модуль json предоставляет простой путь для создания еще более читаемого файла json.

```
>>> shell
1
>>> with open('basic-pretty.json', mode='w', encoding='utf-8') as f:
...     json.dump(basic_entry, f, indent=2) ①
```

① Если вы передадите параметр indent функции Json.dump() она сделает результирующий файл json более читаемым в ущерб размеру файла. Параметр indent это целое число. 0 значит «расположить каждое значение на отдельной строке». Число больше 0 значит «расположить каждое значение на отдельной строке, и использовать number пробелов для отступов во вложенных структурах данных».

И вот результат:

```
you@localhost:~/diveintopython3/examples$ cat basic-pretty.json
{
    "published": true,
```



```

"tags": [
    "diveintopython",
    "docbook",
    "html"
],
"comments_link": null,
"id": 256,
"title": "Dive into history, 2009 edition"
}

```

## Соответствие типов данных Python к JSON

Поскольку JSON разрабатывался не только для Python, есть некоторые недочеты в покрытии типов данных Python. Некоторые из них просто различие в названии типов, но есть два важных типа данных Python, которые полностью упущены из виду. Посмотрим, сможете ли вы заметить их:

Пометки JSON	PYTHON 3
object	dictionary
array	list
string	string
integer	integer
real number	float
* true	True
* false	False
* null	None
* Текст ячейки	Текст ячейки

- - Все переменные в JavaScript регистрозависимые

Вы заметили что потеряно? Кортежи и байты! В JSON есть тип - массив, который модуль JSON ставит в соответствие типу список в Python, но там нет отдельного типа для "статичных массивов» (кортежей). И также в JSON есть хорошая поддержка строк, но нет поддержки объектов типа bytes или массивов байт.

## Сериализация типов данных не поддерживаемых JSON

То что в JSON нет встроенной поддержки типа bytes, не значит что вы не сможете сериализовать объекты типа bytes. Модуль json предоставляет расширяемые хуки для кодирования и декодирования неизвестных типов данных. (Под "неизвестными" я имел в виду "не определенные в json". Очевидно, что модуль json знает о массивах байт, но он создан с учетом ограничений спецификации json). Если вы хотите закодировать тип bytes или другие типы данных, которые json не поддерживает, вам необходимо предоставить особые кодировщики и декодировщики для этих типов данных.

```
>>> shell
1
>>> entry
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ('diveintopython', 'docbook', 'html'),
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27,
tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}
>>> import json
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f)
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "C:\Python31\lib\json\encoder.py", line 170, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: b'\xDE\xD5\xB4\xF8' is not JSON serializable
```

① Хорошо, настало время вновь обратиться к структуре данных entry. Там есть все: логические значения, пустое значение, строка, кортеж строк, объект типа bytes, и структура хранящая время.

② Я знаю, что говорил это ранее, но повторяюсь еще раз: json это текстовый формат. Всегда открывайте файлы json в текстовом режиме с кодировкой utf-8.

③ Чтож... ЭТО не хорошо. Что произошло?

А вот что: функция `json.dump()` попробовала сериализовать объект `bytes b'\xDE\xD5\xB4\xF8'`, но ей не удалось, потому что в json нет поддержки объектов `bytes`. Однако, если сохранение таких объектов важно для вас, вы можете определить свой "мини формат сериализации".

```
def to_json(python_object):
    if isinstance(python_object, bytes):
        return {'__class__': 'bytes',
                '__value__': list(python_object)}
    raise TypeError(repr(python_object) + ' is not JSON serializable')
```

① Чтобы определить свой собственный "мини формат сериализации" для тех типов данных, что json не поддерживает из коробки, просто определите функцию, которая принимает объект Python как параметр. Этот объект Python будет именно тем объектом, который функция `json.dump()` не сможет сериализовать сама - в данном случае это объект `bytes b'\xDE\xD5\xB4\xF8'`

② Вашей специфичной функции сериализации следует проверять тип объектов Python, которые передала ей функция `json.dump()`. Это не обязательно, если ваша функция сериализует только один тип данных, но это делает кристально ясным какой случай покрывает данная функция, и делает более простым улучшение функции, если вам понадобится сериализовать больше типов данных позже

③ В данном случае я решил конвертировать объект `bytes` в словарь. Ключ `__class__` будет содержать название оригинального типа данных, а ключ `__value__` будет хранить само значение. Конечно, это не может быть объекты типа `bytes`, поэтому нужно преобразовать его во что-нибудь сериализуемое при помощи json. Объекты типа `bytes` это просто последовательность чисел, каждое число будет где-то от 0 до 255. Мы можем использовать функцию `list()` чтобы преобразовать объект `bytes` в список чисел. Итак `b'\xDE\xD5\xB4\xF8'` становится `[222, 213, 180, 248]`. (Посчитайте! Это работает! Байт `\xDE` в шестнадцатеричной системе это 222 в десятичной, `\xD5` это 213, и так далее.)

④ Это строка важна. Структура данных, которую вы сериализуете может содержать типы данных которых нет в json, и которые не обрабатывает ваша функция. В таком случае, ваш обработчик должен `raise` ошибку `TypeError` чтобы функция `json.dump()` узнала, что ваш обработчик не смог распознать тип данных.

Вот оно, больше вам ничего не нужно. Действительно, определенная вами функция обработчик возвращает словарь Python, не строку. Вы не пишете сериализацию в json полностью сами, вы просто делаете конвертацию-в-поддерживаемый-тип-данных. Функция `json.dump()` сделает остальное за вас.

```

>>> shell
1
>>> import customserializer
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f, default=customserializer.to_json)
...
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
    json.dump(entry, f, default=customserializer.to_json)
  File "C:\Python31\lib\json\__init__.py", line 178, in dump
    for chunk in iterable:
  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
    for chunk in _iterencode_dict(o, _current_indent_level):
  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
    for chunk in chunks:
  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
    o = _default(o)
  File "/Users/pilgrim/diveintopython3/examples/customserializer.py", line 12, in
to_json
    raise TypeError(repr(python_object) + ' is not JSON serializable')
TypeError: time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22,
tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1) is not JSON
serializable

```

- ① Модуль customserializer, это то где вы только что определили функцию to\_json() в предыдущем примере
- ② Текстовый режим, utf-8, тра-ля-ля. (Вы забудете! Я иногда забываю! И все работает замечательно, пока в один момент не сломается, и тогда оно начинает ломаться еще театральнее)
- ③ Это важный кусок: чтобы встроить вашу функцию обработчик преобразования в функцию json.dump() передайте вашу функцию в json.dump() в параметре default. (Ура, все в Python - объект!)
- ④ Замечательно, это и правда работает. Но посмотрите на исключение. Теперь функция json.dump() больше не жалуется о том, что не может сериализовать объект bytes. Теперь она жалуется о совершенно другом объекте: time.struct\_time.

Хоть получить другое исключение и не выглядит как прогресс, на самом деле это так. Нужно просто добавить пару строк кода, чтобы и это работало.

```
import time
```

```
def to_json(python_object):
```

```

if isinstance(python_object, time.struct_time): ①
    return {'__class__': 'time.asctime',
            '__value__': time.asctime(python_object)} ②
if isinstance(python_object, bytes):
    return {'__class__': 'bytes',
            '__value__': list(python_object)}
raise TypeError(repr(python_object) + ' is not JSON serializable')

```

① Добавляя в уже существующую функцию `customserializer.to_json()` мы должны проверить, что объект Python (с которым у функции `json.dump()` проблемы) на самом деле `time.struct_time`.

② Если так, мы сделаем нечто похожее на конвертацию, что мы делали с объектом `bytes`: преобразуем объект `time.struct_time` в словарь который содержит только те типы данных что можно сериализовать в json. В данном случае, простейший путь преобразовать дату в значение которое можно сериализовать в json это преобразовать ее к строке при помощи функции `time.asctime()`. Функция `time.asctime()` преобразует отвлратительно выглядящую `time.struct_time` в строку `'Fri Mar 27 22:20:42 2009'`.

С этими двумя особыми преобразованиями, структура данных `entry` должна сериализовать полностью без каких либо проблем.

```
>>> shell
```

```
1
```

```
>>> with open('entry.json', 'w', encoding='utf-8') as f:
...     json.dump(entry, f, default=customserializer.to_json)
...
```

```
you@localhost:~/diveintopython3/examples$ ls -l example.json
```

```
-rw-r--r-- 1 you you 391 Aug 3 13:34 entry.json
```

```
you@localhost:~/diveintopython3/examples$ cat example.json
```

```

{"published_date": {"__class__": "time.asctime", "__value__": "Fri Mar 27 22:20:42
2009"},
"comments_link": null, "internal_id": {"__class__": "bytes", "__value__": [222, 213,
180, 248]},
"tags": ["diveintopython", "docbook", "html"], "title": "Dive into history, 2009 edition",
"article_link": "http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-
edition",
"published": true}

```

## Загрузка данных из файла json

Как и в модуле `pickle` в модуле `json` есть функция `load()`, которая принимает на вход поток, читает из него данные в формате json и создает новый объект Python, который будет копией структуры данных записанной в json файле.

```

>>> shell
2
>>> del entry
>>> entry
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'entry' is not defined
>>> import json
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f)
...
>>> entry
{'comments_link': None,
 'internal_id': {'__class__': 'bytes', '__value__': [222, 213, 180, 248]},
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-
edition',
 'published_date': {'__class__': 'time.asctime', '__value__': 'Fri Mar 27 22:20:42
2009'},
 'published': True}

```

① Для демонстрации переключитесь во вторую консоль Python и удалите структуру данных entry которую вы создали ранее в этой главе при помощи модуля Pickle.

② В простейшем случае функция json.load() работает так же как и функция pickle.load(). Вы передаете ей объект потока, а получаете в результате новый объект Python.

③ У меня есть хорошие и плохие новости. Хорошие новости в том, что функция json.load() успешно прочитала файл entry.json, который вы создали в первой консоли Python и создала новый объект Python, который содержит данные. А теперь плохие новости: она не воссоздала оригинальную структуру данных entry. Два значения 'internal\_id' и 'published\_date' были созданы как словари - а именно, как словари со значениями которые совместимы с json (именно их вы создали в функции преобразования to\_json() )

Функция json.load() ничего не знает о функции преобразования, которую вы могли передать в json.dump(). Теперь вам нужно создать функцию, обратную to\_json() — функцию, которая примет выборочно преобразованный json объект и преобразует его обратно в оригинальный объект Python.

```
# add this to customserializer.py
```

```
def from_json(json_object):
```

```
    if '__class__' in json_object:
```

```

if json_object['__class__'] == 'time.asctime':
    return time.strptime(json_object['__value__']) ③
if json_object['__class__'] == 'bytes':
    return bytes(json_object['__value__']) ④
return json_object

```

① Эта функция преобразования так же принимает один параметр и возвращает одно значение. Но параметр который она принимает - не строка, это объект Python - результат десериализации строки JSON, в которую был преобразован объект Python.

② Все что вам нужно, так это проверить, содержит ли данный объект ключ `'__class__'`, который создала функция `to_json()`. Если так, то значение найденное по этому ключу, расскажет вам, как декодировать этот объект обратно в оригинальный объект Python

③ Чтобы декодировать строку времени возвращаемую функцией `time.asctime()` вам нужно использовать функцию `time.strptime()`. Эта функция принимает параметром форматированную строку времени(в определяемом формате, но по умолчанию этот формат совпадает с форматом `time.asctime()`) и возвращает `time.struct_time`

④ Для преобразования списка чисел обратно в объекты `bytes` вы можете использовать функцию `bytes()`

Вот и все, всего два типа данных которые были обработаны функцией `to_json()` и теперь эти же типы данных были обработаны функцией `from_json()`. Вот результат:

```

>>> shell
2
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry = json.load(f, object_hook=customserializer.from_json) ①
...
>>> entry ②
{'comments_link': None,
 'internal_id': b'\xDE\xD5\xB4\xF8',
 'title': 'Dive into history, 2009 edition',
 'tags': ['diveintopython', 'docbook', 'html'],
 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-
edition',
 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27,
tm_hour=22, tm_min=20, tm_sec=42, tm_wday=4, tm_yday=86, tm_isdst=-1),
 'published': True}

```

① Чтобы встроить функцию `from_json()` в процесс десериализации, передайте ее в параметре `object_hook` в вызове функции `json.load()`. Функции которые принимают функции, как удобно!

② Структура данных `entry` теперь содержит ключ `'internal_id'` со значением типа `bytes`. Также она содержит ключ `'published_date'` со значением `time.struct_time`.

Хотя остался еще один глюк.

```
>>> shell
1
>>> import customserializer
>>> with open('entry.json', 'r', encoding='utf-8') as f:
...     entry2 = json.load(f, object_hook=customserializer.from_json)
...
>>> entry2 == entry                                ①
False
>>> entry['tags']                                    ②
('diveintopython', 'docbook', 'html')
>>> entry2['tags']                                   ③
['diveintopython', 'docbook', 'html']
```

① Даже после встраивания функции `to_json()` в сериализацию и функции `from_json()` в десериализацию мы все еще не получили полную копию оригинальной структуры данных. Почему нет?

② В оригинальной структуре данных `entry` значение по ключу `'tags'` было кортежем строк.

③ Но в воссозданной структуре данных `entry2` значение по ключу `'tags'` - это список строк. JSON не видит различий между кортежами и списками, в нем есть только один похожий на список тип данных, массив, и модуль `json` по-тихому преобразует и списки и кортежи в массивы `json` во время сериализации. Для большинства случаев, вы можете проигнорировать различие между списками и кортежами, но это нужно помнить, когда работаешь с модулем `json`.

## Материалы для дальнейшего чтения

- О модуле `pickle`:
  - PEP 238: Изменение оператора деления
- О формате JSON и модуле `json`:
  - `json` — JavaScript Object Notation Serializer
  - JSON encoding and decoding with custom objects in Python



# HTTP и веб-сервисы

## Погружение

HTTP веб-сервисы являются программными способами передачи и получения данных от удаленных серверов, не используя ничего кроме операций HTTP. Если вы хотите получить данные с сервера используйте HTTP GET; если вы хотите отправить данные на сервер, используйте HTTP POST. Более продвинутые функции API HTTP веб-сервиса позволяют создавать, модифицировать, и удалять данные, используя HTTP PUT и HTTP DELETE. Иными словами, «глаголы» встроенные в HTTP протокол (GET, POST, PUT и DELETE) могут отображаться на операции уровня приложения для получения, создания, модифицирования и удаления данных.

HTTP web services are programmatic ways of sending and receiving data from remote servers using nothing but the operations of HTTP. If you want to get data from the server, use HTTP GET; if you want to send new data to the server, use HTTP POST. Some more advanced HTTP web service APIs also allow creating, modifying, and deleting data, using HTTP PUT and HTTP DELETE. In other words, the «verbs» built into the HTTP protocol (GET, POST, PUT, and DELETE) can map directly to application-level operations for retrieving, creating, modifying, and deleting data.

Главное преимущество такого подхода это простота, и эта простота оказалась популярной. Данные — обычно XML или JSON — могут быть построены или сохранены статически, или динамически сгенерированы скриптом на стороне сервера, а все современные языки (включая Python, конечно же!) включают в себя HTTP библиотеку для их загрузки. Отладка также проста; потому что каждый ресурс в HTTP веб-сервисе имеет уникальный адрес (в форме URL), вы можете загрузить его в ваш веб браузер и немедленно увидеть сырые данные.

*The main advantage of this approach is simplicity, and its simplicity has proven popular. Data — usually XML or JSON — can be built and stored statically, or generated dynamically by a server-side script, and all major programming languages (including Python, of course!) include an HTTP library for downloading it. Debugging is also easier; because each resource in an HTTP web service has a unique address (in the form of a URL), you can load it in your web browser and immediately see the raw data.*

Примеры HTTP веб-сервисов:

- \* Google Data APIs позволяют вам взаимодействовать с широким набором сервисов Google, включая Blogger и YouTube.
- \* Flickr Services позволяет вам загружать и выгружать фотографии на Flickr.
- \* Twitter API позволяет вам публиковать обновления статусов на Twitter.
- \* и много других

*Examples of HTTP web services:*

- \* Google Data APIs allow you to interact with a wide variety of Google services, including Blogger and YouTube.
- \* Flickr Services allow you to upload and download photos from Flickr.
- \* Twitter API allows you to publish status updates on Twitter.
- \* ...and many more

Python3 располагает двумя библиотеками для взаимодействия с HTTP веб-сервисами: `http.client` — это низкоуровневая библиотека, реализующая RFC 2616 — протокол HTTP. `urllib.request` — это уровень абстракции, построенный поверх `http.client`. Она предоставляет стандартный API для доступа к HTTP и FTP серверам, автоматически следует HTTP редиректам (перенаправлениям) и умеет работать с некоторыми распространенными формами HTTP аутентификации.

Python 3 comes with two different libraries for interacting with HTTP web services: `http.client` is a low-level library that implements RFC 2616, the HTTP protocol. `urllib.request` is an abstraction layer built on top of `http.client`. It provides a standard

API for accessing both HTTP and FTP servers, automatically follows HTTP redirects, and handles some common forms of HTTP authentication.

Итак, какую же из них следует использовать? Никакую. Вместо них лучше использовать `httplib2` — стороннюю библиотеку с открытым кодом. Она более полно по сравнению с `http.client` реализует HTTP и в то же время предоставляет лучшую чем в `urllib.request` абстракцию.

So which one should you use? Neither of them. Instead, you should use `httplib2`, an open source third-party library that implements HTTP more fully than `http.client` but provides a better abstraction than `urllib.request`.

Чтобы понять, почему вашим выбором должна стать `httplib2`, сначала нужно понять протокол HTTP.

To understand why `httplib2` is the right choice, you first need to understand HTTP.

## 14.2 Особенности HTTP

### 14.2 Features of HTTP

Есть пять важных особенностей, которые все HTTP клиенты должны поддерживать.

There are five important features which all HTTP clients should support.

#### 14.2.1 Кэширование

##### 14.2.1 Caching

Самая важная вещь в понимании о любом веб-сервисе это то, что доступ в сеть невероятно дорог. Я не имею в виду «доллары и центы» (хотя ширина канала не является бесплатной). Я говорю о том, что для открытия соединения, отправки запроса, и получения ответа от удаленного сервера требуется очень много времени. Даже на быстром ширококанальном соединении, задержка (время на передачу запроса и началом получения ответных данных) может по прежнему быть выше, чем вы ожидали. Ошибки роутеров, пропажа пакетов, хакерские атаки на промежуточные прокси — в интернет не бывает ни одной спокойной минуты, и с этим ничего нельзя поделать.

The most important thing to understand about any type of web service is that network access is incredibly expensive. I don't mean «dollars and cents» expensive (although bandwidth ain't free). I mean that it takes an extraordinary long time to open a connection, send a request, and retrieve a response from a remote server. Even on the fastest broadband connection, latency (the time it takes to send a request and start retrieving data in a response) can still be higher than you anticipated. A router misbehaves, a packet is dropped, an intermediate proxy is under attack — there's never a dull moment on the public internet, and there may be nothing you can do about it.

HTTP был спроектирован с учетом кэширования. Есть целый класс устройств (т. н. «кэширующие прокси»), единственной задачей которых является находиться между вами и остальным миром и минимизировать сетевой трафик. В вашей компании или у вашего Интернет-провайдера почти наверняка имеются кэширующие прокси, даже если вы об этом не знаете. Их работа основана на кэшировании, встроенном в протокол HTTP.

HTTP is designed with caching in mind. There is an entire class of devices (called «caching proxies») whose only job is to sit between you and the rest of the world and minimize network access. Your company or ISP almost certainly maintains caching proxies, even if you're unaware of them. They work because caching built into the HTTP protocol.

Вот конкретный пример работы кэширования. С помощью своего браузера вы посетили сайт [diveintomark.org](http://diveintomark.org). На этом сайте имеется картинка [wearehugh.com/m.jpg](http://wearehugh.com/m.jpg). Когда ваш браузер загружает эту картинку, сервер включает в свой ответ следующие HTTP заголовки:

Here's a concrete example of how caching works. You visit [diveintomark.org](http://diveintomark.org) in your browser. That page includes a background image, [wearehugh.com/m.jpg](http://wearehugh.com/m.jpg). When your browser downloads that image, the server includes the following HTTP headers:

```
HTTP/1.1 200 OK
Date: Sun, 31 May 2009 17:14:04 GMT
Server: Apache
Last-Modified: Fri, 22 Aug 2008 04:28:16 GMT
ETag: "3075-ddc8d800"
Accept-Ranges: bytes
Content-Length: 12405
```

Cache-Control: max-age=31536000, public  
Expires: Mon, 31 May 2010 17:14:04 GMT  
Connection: close  
Content-Type: image/jpeg

---

# Пример: перенос chardet на Python 3

## Содержание

- 1 Погружение
- 2 Что такое автоматическое определение кодировки символов?
  - 2.1 Разве это возможно?
  - 2.2 Такой алгоритм существует?
- 3 Введение в модуль chardet
  - 3.1 UTF-N с МПБ
  - 3.2 Экранированные кодировки
  - 3.3 Многобайтные кодировки
  - 3.4 Однобайтные кодировки
  - 3.5 windows-1252

## Погружение

Вопрос: что является первой причины тарабарщины на страницах интернета, с вашем почтовом ящике, и в любой компьютерной системе когда либо написанной? Это кодировка символов. В главе про строки, я рассказывал о истории кодировок и создании Юникода, "одной кодировки правящей всеми". Я бы полюбил ее если бы никогда больше не видел тарабарщины в вебе, потому что все системы сохраняют верную информацию о кодировке, все протоколы

передачи данных поддерживают Юникод, и каждая система работы с текстом придерживалась идеальной верности когда конвертировала из одной кодировки в другую.

Еще я люблю пони.

Пони юникода.

Юникодопони, так сказать.

Я останавлиюсь на автоматическом определении кодировки символов.

## Что такое автоматическое определение кодировки символов?

Это значит взять последовательность байт в неизвестной кодировке, и попытаться определить кодировку так чтобы вы могли читать текст. Это как взломать когда у вас нет ключа для расшифровки.

### Разве это возможно?

Вообще, да. Однако, некоторые кодировки оптимизированны для конктерных языков, и языки не случайны. Некоторые последовательности символов встречаются постоянно, когда другие очень редки. Человек спокойно говорящий по английски открывая газету и найдя там “txzqJv 2!dasd0a QqdKjvz” сразу определит что это не английский(даже если состоит полностью из английских букв). При помощи изучения большого количества "обычного" текста, компьютерный алгоритм может имитировать знание языка и обучиться делать предположения о языке текста.

Другими словами, определение кодировки это на самом деле определение языка, объединенное со знанием какой язык использовать с какой кодировкой.

### Такой алгоритм существует?

Черт побери да! Все основные браузеры имеют встроенное автоопределение кодировки, поскольку интернет полон страниц на которых кодировка вообще не указана. Mozilla Firefox включает библиотеку автоопределения кодировки символов с открытым исходным кодом. Я портировал ее на Python2 и продублировал в модуле chardet. Эта глава покажет вам пошагово весь процесс переноса модуля chardet с Python 2 до Python 3.

## Введение в модуль chardet

Прежде чем начать переносить код, я помогу вам понять как этот код работает! Это краткий инструктаж по навигации в исходном коде. Библиотека chardet слишком большая чтобы включать ее здесь полностью, но вы можете скачать ее с [chardet.feedparser.org](http://chardet.feedparser.org).

Точкой входа алгоритма определения является `universaldetector.py`, в котором есть один класс `UniversalDetector`. (Вы могли подумать что точка входа это

функция `detect` в `chardet/__init__.py`, но на самом деле это просто удобная функция которая создает объект `UniversalDetector`, вызывает его, и возвращает его результат)

Вот пять категорий кодировок которые поддерживает `UniversalDetector`:

- UTF-N с меткой порядка байт(МПБ). Это включает в себя UTF-8, оба большой-индийский и малый-индийский вариант UTF-16, и все 4 варианта UTF-32 зависящих от порядка байт.
- Экранированные последовательности, которые полностью совместимы с 7-битным ASCII, где символы не входящие в семибитную кодировку ASCII начинаются с символа экранирования. Например: ISO-2022-JP(Японская) и HZ-GB-2312(Китайская)
- Многобайтовые кодировки, где каждый символ представлен различным количеством байт. Например: BIG5(Китайская), SHIFT\_JIS(Японская), и TIS-620(Тайская)
- Однобайтовые кодировки, где каждый символ представлен одним байтом. Например: KOI8-R(Русская), WINDOWS-1266(Иврит), и TIS-620(Тайская)
- WINDOWS-1252, которая в основном используется в Microsoft Windows менеджерами среднего звена которые не хотят задумываться о кодировке символов сидя в своей норе.

## UTF-N с МПБ

Если текст начинается с МПБ, мы можем разумно заключить что текст закодирован при помощи кодировки UTF-8, UTF-16, или UTF-32. (МБП расскажет нам какой именно; именно для этого она и служит.) Это поддерживается в `UniversalDetector`, который вернет результат сразу без каких-либо дальнейших изысканий.

## Экранированные кодировки

Если текст содержит распознаваемую экранированную последовательность то это может быть индикатором экранированной кодировки, `UniversalDetector` создаст `EscCharSetProber`(определенный в `escprober.py`) и отдаст текст на обработку.

`EscCharSetProber` создаст ряд конечных автоматов, основанных на моделях HZ-GB-2312, ISO-2022-CN, ISO-2022-JP, и ISO-2022-KR (определенных в `escsm.py`). `EscCharSetProber` пропустит текст через каждый конечный автомат, побайтово. Если только один из автоматов даст положительный результат проверки, `EscCharSetProber` незамедлительно вернет его в `UniversalDetector`, который, в свою очередь, отдаст его вызвавшему его процессу. Если любой из конечных автоматов наткнется на недопустимую последовательность, он останавливается и далее продолжается обработка при помощи другого конечного автомата.



## Многобайтные кодировки

Основываясь на МПБ, UniversalDetector проверяет содержит ли текст символы со старшим байтом. Если так, то он создает набор «исследователей» для определения многобайтных кодировок, однобайтных кодировок, и в качестве последнего средства windows-1252.

Исследователь для многобайтных кодировок, MBCSGroupProber(определенный в mbcsgroupprober.py), на самом деле просто консоль которая управляет группой других исследователей, по одному на каждую многобайтную кодировку: Big5, GB2312, EUC-TW, EUC-KR, EUC-JP, SHIFT\_JIS, и UTF-8. MBCSGroupProber отдает текст каждому из этих кодировкозависимых исследователей и проверяет результат. Если исследователь сообщает что нашел недопустимую последовательность, он исключается из дальнейшей обработки(так что любые последующие вызовы UniversalDetector.feed() пропустят этого исследователя). Если исследователь сообщает что он достаточно уверен в том что определил кодировку, MBCSGroupProber сообщает о положительном результате в UniversalDetector, который передает результат вызвавшему его процессу.

Большинство исследователей многобайтных кодировок наследованы от MultiByteCharSetProber(определенном в mbcharsetprober.py) и просто включают в себя подходящий конечный автомат и анализатор распределения а остальную работу выполняет MultiByteCharSetProber. MultiByteCharSetProber пропускает текст через зависимый от кодировки конечный автомат побайтно, чтобы найти последовательность байт которая бы указала на положительный или отрицательный результат. В то же время, MultiByteCharSetProber пропускает текст через зависимый от кодировки анализатор распределения.

Анализатор распределения(каждый определен в chardistribution.py) использует модель в которой указано в каком языке какие символы встречаются чаще. Как только MultiByteCharSetProber отдал достаточно текста для анализа, вычисляется рейтинг схожести основанный на числе часто используемых символов, общем количестве символов, и коэффициенте распределения специфичном для языка. Если уверенность достаточно высока, MultiByteCharSetProber возвращает результат в MBCSGroupProber, который возвращает результат в UniversalDetector, а он в свою очередь вызвавшему его процессу.

Тяжелее всего разобраться с японским. Односимвольного анализатора распределения не всегда достаточно чтобы различить EUC-JP и SHIFT\_JIS, поэтому SJISProber(определенный в sjisprober.py) также использует двухсимвольный анализатор распределения. SJISContextAnalysis и EUCJPContextAnalysis (оба определенные в jpcntx.py и оба наследованные от класса JapaneseContextAnalysis) проверяют частоту повторения символов Хироганы в тексте. Как только достаточно текста было обработано, он возвращает уровень уверенности в SJISProber, который проверяет оба анализатора и возвращает результат на уровень выше в MBCSGroupProber.

## Однобайтные кодировки

Серьезно, где мой пони юникода?

Исследователь для однобайтных кодировок, `SBCSGroupProber` (определенный в `sbcsgroupprober.py`), так же просто консоль которая управляет группой исследователей, по одному на каждую комбинацию однобайтной кодировки и языка: `windows-1251`, `KOI8-R`, `ISO-8859-5`, `MacCyrillic`, `IBM855`, и `IBM866` (Русский); `ISO-8859-7` и `windows-1253` (Греческий); `ISO-8859-5` и `windows-1251` (Болгарский); `ISO-8859-2` и `windows-1250` (Венгерский); `TIS-620` (Тайский); `windows-1255` и `ISO-8859-8` (Иврит).

`SBCSGroupProber` отдает текст каждому такому исследователю специфичному для языка и кодировки и проверяет результат. Все эти исследователи реализованы как один класс, `SingleByteCharSetProber` (определенный в `sbcharsetprober.py`), который принимает модель языка в качестве аргумента. Модель языка определяет как часто встречаются различные двухсимвольные последовательности в обычном тексте. `SingleByteCharSetProber` обрабатывает текст и отмечает наиболее часто используемые двухсимвольные последовательности. Как только было обработано достаточно текста, он вычисляет уровень схожести основанный на количестве часто встречающихся последовательностей, общем количестве символов, и специфичным для языка коэффициентом распределения.

Иврит обрабатывается по особому. Если при помощи анализа двухсимвольного распределения выясняется что текст может быть на Иврите, `HebrewProber` (определенный в `hebrewprober.py`) пробует различить Визуальный Иврит (где каждая строка исходного текста хранится «наоборот», и потом отображается так же чтобы она могла быть прочтена с права на лево) и Логический Иврит (где текст сохранен в порядке чтения и после этого отображается в клиенте справа на лево). Поскольку некоторые символы кодируются различно в зависимости от положения в слове, мы можем сделать обоснованное предположение о направлении исходного текста, и определить нужную кодировку (`windows-1255` для Логического Иврита или `ISO-8859-8` для Визуального Иврита)

## windows-1252

Если `UniversalDetector` определяет символы со старшим байтом в тексте, но ни один из других многобайтных или однобайтных исследователей не вернул положительный результат, создается `Latin1Prober` (определенный в `latin1prober.py`) чтобы попытаться определить английский текст в кодировке `windows-1252`. Это будет изначально не надежным анализом, потому что английские символы закодированы таким же способом как и во многих различных кодировках. Единственный способ определить `windows-1252` это обратить внимание на часто используемые символы как умные кавычки, выющиеся апострофы, символы копирайта и т. д. `Latin1Prober` автоматически

уменьшает свой уровень уверенности чтобы другие, более достоверные, исследователи могли выиграть если возможно.

---

# Создание пакетов библиотек»

---

# Перенос кода на Python 3 с помощью 2to3»

---

# Особые названия методов

Мы уже обнаружили несколько специальных наименований методов повсюду в книге — магические методы которые питон вызывает когда вы используете определенный синтаксис. Используя специальные методы ваши классы могут работать как последовательности, как словари, как функции, как итераторы или даже как числа. Аппендикс служит как справочник по специальным методам которые мы уже видели и короткое введение в некоторые более эзотерические из них.

## Основы

Если вы читали введение в классы вы уже видели самые общие специальные методы: метод `__init__()`. Многообразие классов которые я написал требуют некоторой инициализации. Также есть некоторые другие специальные методы которые особенно полезны для отлаживания ваших пользовательских классов.

1. Метод `__init__()` вызывается после того как экземпляр создан. Если вы хотите контролировать процесс создания используйте метод `__new__()`.
2. По соглашению метод `__repr__()` должен возвращать строку которая является действительным питоновским выражением.
3. Метод `__str__()` также вызывается когда используется `print(x)`.
4. Новое в Python3, был введен новый тип `bytes`.
5. По соглашению, `format_сpec` должен удовлетворять Format Specification Mini-Language `decimal.py` в стандартной библиотеке Python в которой имеется свой метод `__format__()`.

Классы, которые ведут себя как итераторы.

В главе про итераторы вы видели как построить итератор с нуля используя методы `__iter__()` и `__next__()`.

1. Метод `__iter__()` вызывается когда вы создаете новый итератор. Это хорошее место для инициализации итератора начальными значениями.
2. Метод `__next__()` вызывается когда вы получаете следующее значение из итератора.
3. Метод `__reversed__()` является . Он получает существующую последовательность и возвращает итератор который производит элементы в последовательности в обратном порядке, от последнего к первому.

Как вы видели в главе Итераторы, цикл `for` может быть применен к итератору. В этом цикле:

```
for x in seq:
    print(x)
```

Python 3 будет вызывать `seq.__iter__()` для создания итератора, затем вызовет метод `__next__()` для этого итератора для получения каждого значения `x`.

Когда метод `__next__()`

# Куда пойти

## Это стоит прочитать

Существует некоторое количество тем недостаточно раскрытых в этой книге, однако для их раскрытия существуют открытые ресурсы.

Декораторы:

- Декораторы Функций от Ariel Ortiz
- Подробнее о Декораторах Функций от Ariel Ortiz
- Очаровательный Python: Магия Декораторов это просто от David Mertz
- Определение Функций в официальной документации Python

Свойства:

- The Python propertybuiltin от Adam Goma
- Getters/Setters/Fuxors от Ryan Tomayko
- property() функция в официальной документации Python

#### Дескрипторы:

- How-To руководство по Дескрипторам от Raymond Hettinger
- Очаровательный Python: Элегантность и недостатки Python, Часть 2 от David Mertz
- Дескрипторы Python от Mark Summerfield
- Вызов Дескрипторов в официальной документации Python

#### Мультипоточность & многопроцессорность:

- threading модуль
- threading Управление конкурирующими потоками
- multiprocessing модуль
- multiprocessing Управление процессами как потоками
- Потоки Python и Global Interpreter Lock от Jesse Noller
- Внутри Python GIL (видео) от David Beazley

#### Метаклассы:

- Программирование метаклассов в Python от David Mertz and Michele Simionato
- Программирование метаклассов в Python, Часть 2 от David Mertz and Michele Simionato
- Программирование метаклассов в Python, Часть 3 от David Mertz and Michele Simionato

И в дополнение Doug Hellman™ Python Модуль недели, это фантастическое руководство к большинству модулей для стандартной библиотеки Python.

#### Где искать совместимый с Python 3 код.

Так как Python 3 относительно новый, существует довольно мало совместимых библиотек. Вот некоторые из мест, где Вы могли бы их отыскать.

- Python Package Index: список пакетов Python 3
- Python Cookbook: список рецептов для Python 3
- Google Project Hosting: список проектов Python3
- SourceForge: список проектов базирующихся на Python 3
- GitHub: список проектов Python 3 (а также, список проектов Python 3)
- BitBucket: список проектов python3 (а также Python 3)

# Устранение проблем

## Использование командной строки

Во время чтения этой книги вы видели примеры, в которых программа запускалась из командной строки. Но откуда ее взять? В Linux, найдите в меню Приложений (Applications) программу под названием Терминал (Terminal). (Эта программа может быть и в каком-либо из подменю) В Mac OS X, вы можете найти Терминал (Terminal) в папке /Applications/Utilities/. Чтобы зайти в нее, кликните по рабочему столу, откройте меню Перейти (Go), выберите Перейти в папку... (Go to folder...) и введите /Applications/Utilities/. Затем, дважды кликните по программе Терминал (Terminal). В Windows, нажмите Пуск, выберите пункт меню Выполнить... (Run), введите cmd и нажмите ENTER.

## Запуск программ из командной строки

Как только вы запустили командную строку, вы получаете возможность запустить *интерактивную оболочку Python*. В терминале Linux или Mac OS X наберите python3 и нажмите ENTER. В командной строке Windows введите c:\python31\python и нажмите ENTER. Если все пройдет удачно, то вы увидите что-то, похожее на это

```
you@localhost:~$ python3
Python 3.1 (r31:73572, Jul 28 2009, 06:52:23)
[GCC 4.2.4 (Ubuntu 4.2.4-1ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(Введите exit() и нажмите ENTER, чтобы закрыть интерактивную оболочку Python и вернуться в командную строку. Это работает во всех операционных системах.)

Если вы видите ошибку "command not found" ("**Команда не найдена**"), то это может означать, что Python 3 не установлен.



```
you@localhost:~$ python3  
bash: python3: command not found
```

---

# О книге

«Погружение в Python 3» включает оригинальные тексты и изображения, распространяемые на условиях лицензии CC-BY-SA 3.0 [1]. Иллюстрации из Open Clip Art Library [2] являются общественным достоянием.

Библиотека `chardet` распространяется на условиях LGPL 2.1 или более поздней версии. Решатель криптарифмов — порт программы Рэймонда Эттингера (Raymond Hettinger), выпущенной под лицензией MIT. Некоторые главы содержат код из стандартной библиотеки Python, выпущенной под лицензией PSF 2.0. Все остальные оригинальные исходные коды распространяются под лицензией MIT.

Сайт `diveintopython3.org` использует jQuery [3], выпущенную под лицензиями MIT и GPL. Расцветка синтаксиса в оригинальном тексте (но не в этом переводе) производилась при помощи `prettify.js` и `highlighter.js`; обе библиотеки выпущены под лицензией Apache License 2.0.

Исправления и отзывы по оригинальному тексту посылайте на `mark@diveintomark.org`. Замечания по переводу — Сыру Российскому.

---

# О переводе

## Участники

- Сыр Российский (координатор)
- 9e9names
- Blogytalky



- Bobry
- Ichiro
- K0sh
- Ls
- Pumbatwarek

## Благодарности

- Множеству анонимных переводчиков.

## Трудности перевода

### Терпение, кузнечик

<http://youneedtoknow.com/self-improvement/patience-grasshopper/>

### Постижение генератора

Мне кажется, в главе про comprehensions (генераторы) Марк взял первый попавшийся афоризм со словом «comprehend» (постигать). При переводе игру слов воспроизвести не удалось, вышло как-то совсем ни рыба ни мясо. // Сыр Российский 09:50, 4 января 2012 (UTC)

---

# ВЫХОДНЫЕ данные

Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte. ~ Это письмо получилось таким длинным потому, что у меня не было времени написать его короче.

*Блез Паскаль*

*Примечание переводчика: Значительная часть этой страницы относится только к оригинальной английской HTML-версии.*

## Погружение

Эта книга, как и все книги, сделана с любовью. Конечно, я получил немножко баксов за неё, но никто не пишет техническую литературу ради денег. И поскольку эта книга доступна как на бумаге, так и на вебсайте, я потратил кучу времени на всякие вебовские штучки вместо того, чтобы писать.



Онлайн-версия загружается максимально эффективно. Эффективность никогда не приходит сама, я потратил на неё много часов. Может быть, слишком много часов. Да, почти наверняка слишком много часов. Никогда не недооценивайте глубину того болота, которое затягивает писателей, откладывающих работу в долгий ящик.

Не буду надоедать вам перечислением всех деталей. Нет, погодите. Я буду надоедать вам перечислением всех деталей. Но пока вот вам короткая версия.

1. HTML-код сокращён, и при выдаче сжимается.
2. Скрипты и стили сокращены при помощи YUI Compressor [1] (и тоже выдаются в сжатом виде).
3. Скрипты собраны вместе, чтобы уменьшить количество запросов HTTP.
4. Стили собраны вместе и частично встроены в текст, чтобы уменьшить количество запросов HTTP.
5. Неиспользуемые селекторы и свойства CSS постранично удалены при небольшой помощи ruquery.
6. HTTP-кэширование и прочие серверные опции оптимизированы на основе рекомендаций YSlow [2] и Page Speed.
7. Где возможно, вместо изображений используются юникодные символы [3].
8. Изображения оптимизированы с помощью OptiPNG [4].
9. Вся книга была с любовью написана руками на HTML 5, чтобы не было мусора в коде разметки.

## Типографика

вертикальный ритм, самый красивый амперсанд, фигурные кавычки и апострофы и ещё много чего с [webtypography.net](http://webtypography.net)

## Графика

Юникод, выноски, работа над font-family в Windows

## Быстродействие

«Dive Into History 2009 edition», с минимизацией CSS + JS + HTML, встроенными CSS, с оптимизацией изображений

## Забавные штуки

Цитаты, неровный почерк (?),<sup>[1]</sup> МуссиИзПапайи (ParayaWhip)

## Материалы для дальнейшего чтения

- Применение типографских элементов в Вебе
- Привязка шрифта к линиям сетки в Вебе
- Набор текста с соблюдением вертикального ритма
- Использование самого красивого амперсанда
- Поддержка Юникода в HTML, шрифтах и веб-браузерах
- YSlow [5] для Firebug [6]
- Рекомендации по ускорению веб-сайтов
- 14 правил быстрой загрузки веб-сайтов
- YUI Compressor [7]
- Google Page Speed
- Использование Google Page Speed
- OptiPNG [8]

## Примечания

1. Вопросительный знак принадлежит автору, расшифровывайте его сами. А «constrained writing» может быть не только «неровным почерком», но и «напряжённым стилем изложения». Смысл несильно, но меняется. — *Прим. пер.*
-