

Оглавление

Предисловие	19
Что DevOps означает для авторов.....	20
Как пользоваться этой книгой.....	22
Условные обозначения.....	23
Использование примеров кода.....	24
Благодарности	25
От издательства	28
Глава 1. Основы Python для DevOps	29
Установка и запуск Python.....	30
Командная оболочка Python	30
Блокноты Jupiter	31
Процедурное программирование	32
Переменные.....	32
Основные математические операции	33
Комментарии.....	33
Встроенные функции.....	34
Print	34
Range	35

Контроль выполнения	35
if/elif/else	36
Циклы for	37
Циклы while	38
Обработка исключений	39
Встроенные объекты	40
Что такое объект	40
Методы и атрибуты объектов	41
Последовательности	42
Функции	55
Синтаксис функции	55
Функции как объекты	57
Анонимные функции	57
Регулярные выражения	58
Поиск	59
Наборы символов	60
Классы символов	61
Группы	61
Поименованные группы	61
Найти все	62
Поисковый итератор	62
Подстановка	63
Компиляция	63
Отложенное вычисление	64
Генераторы	64
Генераторные включения	65
Дополнительные возможности IPython	66
Выполнение инструкций командной оболочки Unix с помощью IPython	66
Упражнения	68

Глава 2. Автоматизация работы с файлами и файловой системой	69
Чтение и запись файлов	69
Поиск в тексте с помощью регулярных выражений	79
Обработка больших файлов.....	81
Шифрование текста.....	82
Хеширование с помощью пакета hashlib	82
Шифрование с помощью библиотеки cryptography	83
Модуль os	85
Управление файлами и каталогами с помощью os.path.....	86
Обход дерева каталогов с помощью os.walk	90
Пути как объекты: библиотека pathlib.....	91
Глава 3. Работа с командной строкой	93
Работа с командной оболочкой.....	93
Взаимодействие с интерпретатором с помощью модуля sys	93
Взаимодействие с операционной системой с помощью модуля os	94
Порождение процессов с помощью модуля subprocess.....	95
Создание утилит командной строки	97
Использование sys.argv	99
argparse	101
click.....	106
fire	110
Реализация плагинов.....	115
Ситуационный анализ: разгоняем Python с помощью утилит командной строки.....	116
Динамический компилятор Numba	117
Использование GPU с помощью CUDA Python	119
Многоядерное многопоточное выполнение кода Python с помощью Numba	120
Кластеризация методом k-средних.....	122
Упражнения.....	123

Глава 4. Полезные утилиты Linux	124
Дисковые утилиты.....	125
Измерение быстродействия	126
Разделы диска	128
Получение информации о конкретном устройстве.....	129
Сетевые утилиты.....	131
SSH-туннелирование	131
Оценка быстродействия HTTP с помощью Apache Benchmark (ab)	132
Нагрузочное тестирование с помощью molotov	133
Утилиты для получения информации о загрузке CPU.....	136
Просмотр процессов с помощью htop	136
Работаем с Bash и ZSH	138
Настройка командной оболочки Python под свои нужды	140
Рекурсивные подстановки	140
Поиск и замена с запросами подтверждения	141
Удаление временных файлов Python	143
Вывод списка процессов и его фильтрация.....	143
Метка даты/времени Unix.....	144
Комбинирование Python с Bash и ZSH.....	144
Генератор случайных чисел.....	145
Существует ли нужный мне модуль?	146
Переходим из текущего каталога по пути к модулю	146
Преобразование CSV-файла в JSON.....	147
Однострочные сценарии Python	148
Отладчики	148
Быстро ли работает конкретный фрагмент кода?	149
strace.....	150
Вопросы и упражнения	153
Задача на ситуационный анализ	153

Глава 5. Управление пакетами	154
Почему пакетная организация программ так важна	155
Случаи, когда пакетная организация программ не нужна.....	155
Рекомендации по пакетной организации программ.....	156
Информативный контроль версий.....	156
Журнал изменений	158
Выбор стратегии	159
Решения для создания пакетов.....	159
Нативные пакеты Python.....	160
Создание пакетов для Debian	166
Создание пакетов RPM	174
Диспетчеризация с помощью systemd.....	180
Долгоживущие процессы	181
Настройка	182
Юниты systemd	183
Установка юнита.....	185
Управление журналами.....	186
Вопросы и упражнения	188
Задача на ситуационный анализ	188
Глава 6. Непрерывная интеграция и непрерывное развертывание	189
Ситуационный анализ примера из практики: перевод плохо работавшего сайта с WordPress на Hugo.....	189
Настройка Hugo	191
Преобразование WordPress в посты Hugo.....	192
Создание поискового индекса Algolia и его обновление	194
Координация с помощью Makefile.....	196
Развертывание с помощью AWS CodePipeline	196
Ситуационный анализ примера из практики: развертывание приложения Python App Engine с помощью Google Cloud Build.....	197
Ситуационный анализ примера из практики: NFSOPS.....	205

Глава 7. Мониторинг и журналирование	207
Ключевые понятия создания надежных систем.....	207
Неизменные принципы DevOps	208
Централизованное журналирование	209
Ситуационный анализ: база данных при промышленной эксплуатации разрушает жесткие диски	209
Производить или покупать?	210
Отказоустойчивость.....	211
Мониторинг	213
Graphite	214
StatsD.....	214
Prometheus	215
Телеметрия	219
Соглашения о наименованиях	222
Журналирование	224
Почему это так трудно	224
basicconfig.....	225
Углубляемся в конфигурацию.....	226
Распространенные паттерны.....	231
Стек ELK	232
Logstash.....	233
Elasticsearch и Kibana.....	235
Вопросы и упражнения	239
Задача на ситуационный анализ	239
Глава 8. Pytest для DevOps	240
Сверхспособности тестирования фреймворка pytest	240
Начало работы с pytest	241
Тестирование с помощью pytest.....	242
Отличия от unittest.....	244

Возможности pytest.....	245
conftest.py	246
Этот замечательный оператор assert	247
Параметризация	248
Фикстуры.....	250
Приступим	250
Встроенные фикстуры.....	252
Инфраструктурное тестирование	255
Что такое проверка системы.....	256
Введение в Testinfra	257
Подключение к удаленным узлам	258
Фикстуры и особые фикстуры.....	261
Примеры.....	263
Тестирование блокнотов Jupyter с помощью pytest	266
Вопросы и упражнения	267
Задача на ситуационный анализ	267
Глава 9. Облачные вычисления	268
Основы облачных вычислений.....	269
Типы облачных вычислений.....	271
Типы облачных сервисов.....	272
Инфраструктура как сервис	272
«Железо» как сервис	277
Платформа как сервис.....	278
Бессерверная обработка данных.....	278
Программное обеспечение как сервис.....	282
Инфраструктура как код	283
Непрерывная поставка.....	283
Виртуализация и контейнеры	283
Аппаратная виртуализация	283
Программно определяемые сети	284

Программно определяемое хранилище	285
Контейнеры	285
Трудные задачи и потенциальные возможности распределенной обработки данных.....	286
Конкурентное выполнение на Python, быстроедействие и управление процессами в эпоху облачных вычислений	289
Управление процессами	289
Процессы и дочерние процессы	289
Решение задач с помощью библиотеки multiprocessing.....	292
Ветвление процессов с помощью Pool()	293
Функция как сервис и бессерверная обработка данных.....	295
Повышение производительности Python с помощью библиотеки Numba	295
Динамический компилятор Numba	295
Высокопроизводительные серверы.....	296
Резюме.....	297
Вопросы	298
Вопросы на ситуационный анализ.....	298
Глава 10. Инфраструктура как код	299
Классификация инструментов автоматизации выделения инфраструктуры	301
Выделение инфраструктуры вручную.....	302
Автоматическое выделение инфраструктуры с помощью Terraform.....	304
Выделение корзины S3	304
Предоставление SSL-сертификата с помощью ACM AWS	307
Выделение раздачи Amazon CloudFront.....	308
Создание записи DNS Route 53	311
Копирование статических файлов в корзину S3.....	312
Удаление всех ресурсов AWS, выделенных с помощью Terraform.....	313

Автоматическое выделение инфраструктуры с помощью Pulumi	313
Создание нового проекта Pulumi на Python для AWS.....	314
Создание значений параметров конфигурации для стека staging	319
Создаем SSL-сертификат ACM.....	319
Выделение зоны Route 53 и записей DNS	320
Выделение раздачи CloudFront.....	323
Создание записи DNS Route 53 для URL сайта.....	324
Создание и развертывание нового стека.....	325
Упражнения.....	327
Глава 11. Контейнерные технологии: Docker и Docker Compose.....	328
Что такое контейнер Docker	329
Создание, сборка, запуск и удаление образов и контейнеров Docker.....	330
Публикация образов Docker в реестре Docker.....	334
Запуск контейнера Docker из одного образа на другом хост-компьютере	335
Запуск нескольких контейнеров Docker с помощью Docker Compose	337
Портирование сервисов docker-compose на новый хост-компьютер и операционную систему.....	350
Упражнения.....	354
Глава 12. Координация работы контейнеров: Kubernetes.....	355
Краткий обзор основных понятий Kubernetes	356
Создание манифестов Kubernetes на основе файла docker_compose.yaml с помощью Kompose	357
Развертывание манифестов Kubernetes на локальном кластере Kubernetes, основанном на minikube	359
Запуск кластера GKE Kubernetes в GCP с помощью Pulumi.....	374
Развертывание примера приложения Flask в GKE.....	377
Установка чартов Helm для Prometheus и Grafana.....	383
Удаление кластера GKE.....	389
Упражнения.....	390

Глава 13. Технологии бессерверной обработки данных.....	391
Развертывание одной и той же функции Python в облака большой тройки поставщиков облачных сервисов.....	394
Установка фреймворка Serverless.....	394
Развертывание функции Python в AWS Lambda.....	395
Развертывание функции Python в Google Cloud Functions.....	397
Развертывание функции на Python в Azure.....	403
Развертывание функции на Python на самохостируемых FaaS-платформах.....	408
Развертывание функции на Python в OpenFaaS.....	408
Выделение таблиц DynamoDB, функций Lambda и методов API Gateway с помощью AWS CDK.....	416
Упражнения.....	438
Глава 14. MLO и разработка ПО для машинного обучения.....	439
Что такое машинное обучение.....	439
Машинное обучение с учителем.....	440
Моделирование.....	442
Экосистема машинного обучения языка Python.....	445
Глубокое обучение с помощью PyTorch.....	445
Платформы облачного машинного обучения.....	449
Модель зрелости машинного обучения.....	451
Основная терминология машинного обучения.....	452
Уровень 1. Очерчивание рамок задачи и области определения, а также формулировка задачи.....	453
Уровень 2. Непрерывная поставка данных.....	453
Уровень 3. Непрерывная поставка очищенных данных.....	455
Уровень 4. Непрерывная поставка разведочного анализа данных.....	457
Уровень 5. Непрерывная поставка обычного ML и AutoML.....	457
Уровень 6. Цикл обратной связи эксплуатации ML.....	458

Приложение sklearn Flask с использованием Docker и Kubernetes.....	459
Разведочный анализ данных.....	463
Моделирование	464
Тонкая настройка масштабированного GBM.....	465
Подгонка модели.....	466
Оценка работы модели.....	466
adhoc_predict.....	467
Технологический процесс JSON.....	468
Масштабирование входных данных	468
adhoc_predict на основе выгрузки	470
Масштабирование входных данных	470
Вопросы и упражнения	471
Задача на ситуационный анализ	471
Вопросы на проверку усвоения материала	471
Глава 15. Инженерия данных	472
Малые данные.....	473
Обработка файлов малых данных	474
Запись в файл	474
Чтение файла.....	474
Конвейер с генератором для чтения и обработки строк.....	475
YAML	476
Большие данные	477
Утилиты, компоненты и платформы для работы с большими данными.....	479
Источники данных.....	480
Файловые системы	480
Хранение данных.....	481
Ввод данных в режиме реального времени	483
Ситуационный анализ: создание доморощенного конвейера данных	484
Бессерверная инженерия данных.....	485

AWS Lambda и события CloudWatch	486
Журналирование Amazon CloudWatch для AWS Lambda.....	486
Наполнение данными Amazon Simple Queue Service с помощью AWS Lambda	487
Подключение срабатывающего по событию триггера CloudWatch.....	492
Создание событийно-управляемых функций Lambda	493
Чтение событий Amazon SQS из AWS Lambda.....	493
Резюме.....	498
Упражнения.....	498
Задача на ситуационный анализ	498
Глава 16. Истории из практики DevOps и интервью.....	499
Киностудия не может снять фильм	500
Разработчик игр не может обеспечить поставку игрового ПО.....	503
Сценарии Python, запуск которых требует 60 секунд	505
Решаем горячие проблемы с помощью кэша и интеллектуальной телеметрии.....	506
Доавтоматизироваться до увольнения.....	507
Антипаттерны DevOps.....	509
Антипаттерн: отсутствие автоматизированного сервера сборки	509
Работать вслепую	509
Сложности координации как постоянная проблема.....	510
Отсутствие командной работы	511
Интервью.....	517
Гленн Соломон	517
Эндрю Нгуен	518
Габриэлла Роман	520
Ригоберто Рош	521
Джонатан Лакур.....	523
Вилле Туулос	525
Джозеф Рис.....	527

Тейо Хольцер	529
Мэтт Харрисон	531
Майкл Фоорд.....	533
Рекомендации.....	536
Вопросы	537
Интересные задачи.....	537
Дипломный проект.....	538
Об авторах	539
Об иллюстрации на обложке	541

Основы Python для DevOps

DevOps — сочетание разработки программного обеспечения с IT-задачами — в последнее десятилетие обрело большую популярность. Традиционные границы между разработкой программного обеспечения, его развертыванием, сопровождением и контролем качества все больше размываются, приводя к тесной интеграции различных групп специалистов. Python — язык программирования, популярный как в среде традиционных IT-задач, так и в DevOps благодаря сочетанию гибкости, широких возможностей и простоты использования.

Язык программирования Python появился в начале 1990-х и изначально предназначался для системного администрирования. В этой сфере он приобрел большую популярность и применяется очень широко. Python представляет собой универсальный язык программирования и используется практически во всех предметных областях, даже в киноиндустрии для создания спецэффектов. А совсем недавно он стал фактически основным языком науки о данных и машинного обучения. Он присутствует повсюду, от авиации до биоинформатики. Python включает обширный инструментарий, охватывающий все нужды его пользователей. Изучение стандартной библиотеки Python, предоставляемой с любым дистрибутивом Python-функциональности, потребовало бы огромных усилий. А изучить все сторонние библиотеки, оживляющие экосистему Python, — поистине необъятная задача. К счастью, этого не требуется. Можно с большим успехом практиковать DevOps, изучив лишь малую толику Python.

В этой главе мы, основываясь на многолетнем опыте применения Python для DevOps, рассмотрим только необходимые элементы этого языка. Некоторые части Python составляют инструментарий, необходимый для решения ежедневных задач DevOps. После изучения этих основ вы сможете в следующих главах перейти к более продвинутым инструментам.

Установка и запуск Python

Чтобы опробовать в работе код из этой главы, вам понадобятся Python версии 3.7 или более поздней¹ и доступ к командной оболочке. В macOS X, Windows и большинстве дистрибутивов Linux для доступа к командной оболочке достаточно открыть терминал. Чтобы узнать используемую версию Python, откройте командную оболочку и наберите команду `python --version`:

```
$ python --version
Python 3.8.0
```

Скачать установочные пакеты Python можно непосредственно с сайта Python.org (<https://www.python.org/downloads>). Можно также воспользоваться системой управления пакетами, например Apt, RPM, MacPorts, Homebrew, Chocolatey и др.

Командная оболочка Python

Простейший способ работы с Python — воспользоваться встроенным интерактивным интерпретатором. Просто набрав в командной оболочке `python`, вы сможете интерактивно выполнять операторы Python. Для выхода из командной оболочки наберите `exit()`:

```
$ python
Python 3.8.0 (default, Sep 23 2018, 09:47:03)
[Clang 9.0.0 (clang-900.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 2
3
>>> exit()
```

Сценарии Python

Код Python выполняется из файла с расширением `.py`:

```
# Мой первый сценарий Python
print('Hello world!')
```

Сохраните этот код в файле `hello.py`. Для вызова этого сценария наберите в командной оболочке `python` с последующим именем файла:

```
$ python hello.py
Hello world!
```

¹ По состоянию на сентябрь 2021 г. текущая стабильная версия — 3.9.7. — *Примеч. пер.*

Большая часть кода Python в промышленной эксплуатации выполняется именно в виде сценариев Python.

IPython

Помимо встроенной интерактивной командной оболочки, код Python позволяет выполнять несколько сторонних интерактивных командных оболочек. Одна из наиболее популярных — IPython (<https://ipython.org>). Возможности IPython включают *интроспекцию* (динамическое получение информации об объектах), подсветку синтаксиса, специальные *магические* команды (которые мы обсудим далее в этой главе) и многое другое, превращая изучение Python в сплошное удовольствие. Для установки IPython можно использовать систему управления пакетами Python `pip`:

```
$ pip install ipython
```

Ее запуск аналогичен запуску встроенной интерактивной командной оболочки, описанному в предыдущем разделе:

```
$ ipython
Python 3.8.0 (default, Sep 23 2018, 09:47:03)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.5.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]: print('Hello')
Hello

In [2]: exit()
```

Блокноты Jupyter

Отпочковавшийся от проекта IPython проект Jupyter позволяет работать со специальными документами, включающими текст, код и визуализации. Эти документы дают возможность сочетать выполнение кода, вывод результатов работы и форматирование текста. Jupyter позволяет сопровождать код документацией. Он стал популярен повсеместно, особенно в мире науки о данных. Установить Jupyter и запустить блокноты можно вот так:

```
$ pip install jupyter
$ jupyter notebook
```

Эта команда открывает веб-браузер и отображает текущий рабочий каталог. А там уже вы сможете открывать имеющиеся блокноты разрабатываемого проекта или создавать новые.

Процедурное программирование

Если вы хоть немного сталкивались с программированием, то наверняка слышали термины «объектно-ориентированное программирование» (ООП) и «функциональное программирование» — так называются две различные архитектурные парадигмы организации программ. В качестве стартовой точки парадигма процедурного программирования, одна из самых простых, подходит прекрасно. *Процедурное программирование* (procedural programming) означает описание инструкций компьютеру в виде упорядоченной последовательности:

```
>>> i = 3
>>> j = i + 1
>>> i + j
7
```

Как видите, в этом примере приведены три оператора, выполняемых по порядку, от первой строки до последней. Каждый из них использует сформированное предыдущими операторами состояние. В данном случае первый оператор присваивает переменной `i` значение 3. Во втором операторе значение этой переменной применяется для присваивания значения переменной `j`, а в третьем значения обеих переменных складываются. Пусть вас не волнуют детали синтаксиса этих операторов, обратите внимание только на то, что они выполняются по порядку и зависят от состояния, сформированного предыдущими операторами.

Переменные

Переменная — это имя, указывающее на какое-то значение. В предыдущем примере встречались переменные `i` и `j`. Переменным в языке Python можно присваивать новые значения:

```
>>> dog_name = 'spot'
>>> dog_name
'spot'
>>> dog_name = 'rex'
>>> dog_name
'rex'
>>> dog_name = 't-' + dog_name
>>> dog_name
't-rex'
>>>
```

Типизация переменных языка Python динамическая. На практике это означает, что им можно повторно присваивать значения, относящиеся к другим типам или классам:


```
>>> big = 'large'
>>> big
'large'
>>> big = 1000*1000
>>> big
1000000
>>> big = {}
>>> big
{}
```

В данном случае одной и той же переменной присваиваются строковое значение, числовое, а затем ассоциативный массив. Переменным можно повторно присваивать значения любого типа.

Основные математические операции

Для основных математических операций — сложения, вычитания, умножения и деления — существуют встроенные математические операторы:

```
>>> 1 + 1
2
>>> 3 - 4
-1
>>> 2*5
10
>>> 2/3
0.6666666666666666
```

Символ `//` служит для целочисленного деления. Символ `**` означает возведение в степень, а `%` — оператор взятия остатка от деления:

```
>>> 5/2
2.5
>>> 5//2
2
>>> 3**2
9
>>> 5%2
1
```

Комментарии

Комментарии — текст, который интерпретатор Python игнорирует. Они удобны для документирования кода, некоторые сервисы умеют собирать их для создания отдельной документации. Однострочные комментарии отделяются указанием перед ними символа `#` и могут начинаться как в начале строки, так

и в любом месте в ней. Все символы от # до символа новой строки — это часть комментария:

```
# Комментарий
1 + 1 # Комментарий, следующий за оператором
```

Многострочные комментарии заключаются в блоки, начинающиеся и заканчивающиеся символами `"""` или `'''`:

```
"""
Многострочный
комментарий.
"""

...
Этот оператор также представляет собой
многострочный комментарий
...
```

Встроенные функции

Функции — это сгруппированные операторы. Вызывается функция путем ввода ее имени с последующими скобками. Внутри них указываются принимаемые функцией аргументы (при их наличии). В языке Python есть множество встроенных функций. Две из числа наиболее часто используемых встроенных функций — `print` и `range`.

Print

Функция `print` генерирует видимую пользователям программы информацию. В интерактивной среде она не очень полезна, но при написании сценариев Python это важнейший инструмент. В предыдущем примере аргумент функции `print` выводится в консоль при запуске сценария:

```
# Мой первый сценарий Python
print("Hello world!")
```

```
$ python hello.py
Hello world!
```

С помощью функции `print` можно просматривать значения переменных или предоставлять обратную связь о состоянии программы. Функция `print` обычно выводит информацию в стандартный поток вывода, отображаемый в командной оболочке.

Range

Хотя `range` — одна из встроенных функций, формально это вообще не функция, а тип, служащий для представления последовательности чисел. При вызове конструктора `range()` возвращается представляющий последовательность чисел объект. Функция `range` принимает до трех целочисленных аргументов. При указании лишь одного аргумента последовательность состоит из чисел от нуля до этого аргумента, не включая его. Вторым аргумент (при его наличии) отражает точку, с которой вместо нуля начинается последовательность. Третий аргумент служит для указания шага последовательности, по умолчанию равен 1:

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(5, 10, 3))
[5, 8]
>>>
```

Требования к оперативной памяти функции `range` невелики даже для больших последовательностей, ведь хранить нужно только значения начала, конца и шага последовательности. Функция `range` может проходить по большим последовательностям чисел без снижения быстродействия.

Контроль выполнения

В языке Python есть множество конструкций для управления потоком выполнения операторов. Операторы, которые нужно выполнять вместе, можно группировать в блоки кода, которые можно выполнять многократно с помощью циклов `for` и `while` либо только при определенных условиях с помощью оператора `if`, цикла `while` или блоков `try-except`. Применение подобных конструкций — первый шаг к подлинной реализации возможностей программирования. В различных языках программирования разграничение блоков кода определяется разными соглашениями. Во многих C-подобных языках (очень важный язык, использовавшийся при написании операционной системы Unix) блок описывается путем заключения группы операторов в круглые скобки. В Python для этой цели применяются отступы. Операторы группируются по числу отступов в блоки, выполняемые как единое целое.



Интерпретатору Python неважно, формируете вы отступы кода пробелами или символами табуляции, главное — единообразие. Впрочем, руководство PEP-8 по стилю написания кода Python (<https://oreil.ly/b5yU4>) рекомендует отделять уровни отступов с помощью четырех пробелов.

if/elif/else

Операторы `if/elif/else` — распространенное средство выбора веток кода. Следующий непосредственно за оператором `if` блок кода выполняется, если значение условия оператора равно `True`:

```
>>> i = 45
>>> if i == 45:
...     print('i is 45')
...
...
i is 45
>>>
```

Здесь использовался оператор `==`, возвращающий `True`, если его операнды равны между собой, и `False` в противном случае. За этим блоком может следовать необязательный оператор `elif` или `else` со своим блоком, который в случае оператора `elif` выполняется, только если условие `elif` равно `True`:

```
>>> i = 35
>>> if i == 45:
...     print('i is 45')
... elif i == 35:
...     print('i is 35')
...
...
i is 35
>>>
```

При желании можно расположить друг за другом несколько выражений `elif`. Это напоминает множественный выбор с помощью операторов `switch` в других языках программирования. Добавление оператора `else` в конце позволяет выполнять блок только в том случае, если ни одно из предыдущих условий не равно `True`:

```
>>> i = 0
>>> if i == 45:
...     print('i is 45')
... elif i == 35:
...     print('i is 35')
... elif i > 10:
```

```
...     print('i is greater than 10')
... elif i%3 == 0:
...     print('i is a multiple of 3')
... else:
...     print('I don't know much about i...')
...
...
i is a multiple of 3
>>>
```

Операторы `if` могут быть вложенными, с содержащими операторы `if` блоками, которые выполняются только в том случае, если условие во внешнем операторе `if` равно `True`:

```
>>> cat = 'spot'
>>> if 's' in cat:
...     print("Found an 's' in a cat")
...     if cat == 'Sheba':
...         print("I found Sheba")
...     else:
...         print("Some other cat")
... else:
...     print(" a cat without 's'")
...
...
Found an 's' in a cat
Some other cat
>>>
```

Циклы `for`

Циклы `for` позволяют повторять выполнение блока операторов (блока кода) столько раз, сколько содержится элементов в *последовательности* (упорядоченной группе элементов). При проходе в цикле по последовательности блок кода обращается к текущему элементу. Чаще всего циклы используются для прохода по объекту `range` для выполнения какой-либо операции заданное число раз:

```
>>> for i in range(10):
...     x = i*2
...     print(x)
...
...
0
2
4
6
```

```
8  
10  
12  
14  
16  
18  
>>>
```

В этом примере блок кода имеет следующий вид:

```
...     x = i*2  
...     print(x)
```

Этот код выполняется десять раз, причем переменной `i` каждый раз присваивается следующий элемент последовательности целых чисел из диапазона 0–9. Циклы `for` подходят для обхода любых типов последовательностей Python, как вы увидите далее в этой главе.

continue

Оператор `continue` позволяет пропустить один шаг в цикле и перейти сразу к очередному элементу последовательности:

```
>>> for i in range(6):  
...     if i == 3:  
...         continue  
...     print(i)  
...  
...  
0  
1  
2  
4  
5  
>>>
```

Циклы while

Цикл `while` повторяет выполнение блока кода до тех пор, пока условное выражение равно `True`:

```
>>> count = 0  
>>> while count < 3:  
...     print(f"The count is {count}")  
...     count += 1  
...
```



```
...
The count is 0
The count is 1
The count is 2
>>>
```

Главное — задать условие выхода из такого цикла, в противном случае он будет выполняться до тех пор, пока программа не завершится аварийно. Для этого можно, например, задать такое условное выражение, которое в конце концов окажется равным `False`. Либо воспользоваться оператором `break` для выхода из цикла с помощью вложенного условного оператора:

```
>>> count = 0
>>> while True:
...     print(f"The count is {count}")
...     if count > 5:
...         break
...     count += 1
...
...
The count is 0
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
>>>
```

Обработка исключений

Исключения — ошибки, которые могут привести к фатальному сбою программы, если их не обработать должным образом (перехватить). Благодаря их перехвату с помощью блока `try-except` программа может продолжить работу. Для создания такого блока необходимо добавить отступы к блоку, в котором может возникнуть исключение, поместить перед ним оператор `try`, а после него — оператор `except`. За ним будет следовать блок кода, который должен выполняться при возникновении ошибки:

```
>>> thinkers = ['Plato', 'PlayDo', 'Gumby']
>>> while True:
...     try:
...         thinker = thinkers.pop()
...         print(thinker)
...     except IndexError as e:
...         print("We tried to pop too many thinkers")
```

```
...     print(e)
...     break
...
...
...
Gumby
PlayDo
Plato
We tried to pop too many thinkers
pop from empty list
>>>
```

Существует множество встроенных исключений, например `IOError`, `KeyError` и `ImportError`. Во многих сторонних пакетах также описаны собственные классы исключений, указывающих на серьезные проблемы, так что перехватывать их имеет смысл, только если вы уверены, что соответствующая проблема не критична для вашего приложения. Можно указывать явным образом, какое исключение перехватывается. По возможности следует перехватывать как можно меньше видов исключений (в нашем примере исключение `IndexError`).

Встроенные объекты

В этом кратком обзоре мы не станем стремиться охватить ООП в целом. Впрочем, в языке Python есть немало встроенных классов.

Что такое объект

В ООП данные (состояние) и функциональность объединены. Для работы с объектами необходимо разобраться с такими понятиями, как *создание экземпляров классов* (class instantiation) — создание объектов на основе классов и *синтаксис с использованием точки* (dot syntax), служащий для доступа к атрибутам и методам объектов. В классе описываются атрибуты и методы, совместно используемые всеми его объектами. Его можно считать чем-то вроде чертежа автомобиля. На основе такого «чертежа» можно затем создавать экземпляры этого класса. Экземпляр класса (объект) — это конкретный автомобиль, построенный по такому чертежу.

```
>>> # Описываем класс для воображаемого описания воображаемых автомобилей
>>> class FancyCar():
...     pass
...
>>> type(FancyCar)
<class 'type'>
```

```
>>> # Создаем экземпляр воображаемого автомобиля
>>> my_car = FancyCar()
>>> type(my_car)
<class '__main__.FancyCar'>
```

Не волнуйтесь пока насчет создания собственных классов. Просто запомните, что любой объект представляет собой экземпляр какого-либо класса.

Методы и атрибуты объектов

Данные объектов хранятся в их атрибутах, представляющих собой прикрепленные к объектам или классам объектов переменные. Функциональность объектов описывается в *методах объекта* (методах, описанных для всех объектов класса) и *методах класса* (методах, относящихся к классу и совместно используемых всеми объектами данного класса), представляющих собой связанные с объектом функции.



В документации Python прикрепленные к объектам и классам функции называются методами.

У этих функций есть доступ к атрибутам объекта, они могут модифицировать и использовать его данные. Для вызова методов объекта или доступа к его атрибутам используется синтаксис с применением точки:

```
>>> # Описываем класс для воображаемого описания воображаемых автомобилей
>>> class FancyCar():
...     # Добавляем переменную класса
...     wheels = 4
...     # Добавляем метод
...     def driveFast(self):
...         print("Driving so fast")
...
...
>>> # Создаем экземпляр воображаемого автомобиля
>>> my_car = FancyCar()
>>> # Обращаемся к атрибуту класса
>>> my_car.wheels
4
>>> # Вызываем метод
>>> my_car.driveFast()
Driving so fast
>>>
```