# Оглавление

# ГЛАВА 1

# 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, поставляемой 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 поздней[1] 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 Jupiter

Отпочковавшийся от проекта IPython проект Jupiter позволяет работать со специальными документами, включающими текст, код и визуализации. Эти документы дают возможность сочетать выполнение кода, вывод результатов работы и форматирование текста. Jupiter позволяет сопровождать код документацией. Он стал популярен повсеместно, особенно в мире науки о данных. Установить Jupiter 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. Второй 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
>>>
```