ГЛАВА 1

# Знакомство с IPython и Jupyter

Разрабатывая код на Python для исследования данных, я обычно переключаюсь между тремя режимами работы: использую оболочку IPython для тестирования коротких последовательностей команд, Jupyter Notebook — для более продолжительного интерактивного анализа и обмена результатами исследований с другими, а также интерактивные среды разработки (Interactive Development Environments, IDE), такие как Emacs или VSCode, для создания многократно используемых пакетов Python. В этой главе основное внимание уделяется первым двум режимам: оболочке IPython и Jupyter Notebook. Интерактивная среда разработки (IDE) — очень важный третий инструмент в арсенале любого специалиста по обработке и анализу данных, но здесь мы не будем обращаться к нему напрямую.

## Запуск командной оболочки IPython

Данная глава, как и бóльшая часть книги, не предназначена для пассивного чтения. Я рекомендую вам экспериментировать с описываемыми инструментами и синтаксисом: формируемая при этом мышечная память принесет намного больше пользы, чем простое чтение. Начнем с запуска интерпретатора оболочки IPython путем ввода команды `ipython` в командной строке. Если вы установили один из таких дистрибутивов, как Anaconda или EPD, то у вас, возможно, уже есть средство запуска для вашей операционной системы.

После этого вы должны увидеть приглашение к вводу:

```
Python 3.9.2 (v3.9.2:1a79785e3e, Feb 19 2021, 09:06:10)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.21.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Теперь вы можете активно следовать за примерами в книге.

# Запуск Jupyter Notebook

Jupyter Notebook — браузерный графический интерфейс для командной оболочки Ipython, предлагающий богатый набор возможностей динамической визуализации. Помимо выполнения операторов Python/IPython, блокноты позволяют пользователям вставлять форматированный текст, статические и динамические диаграммы, математические уравнения, виджеты JavaScript и многое другое. Более того, эти документы можно сохранять, благодаря чему другие смогут открывать и выполнять их в своих системах.

Хотя просмотр и редактирование блокнотов Jupyter осуществляется в окне браузера, они должны подключаться к запущенному процессу Python для выполнения кода. Для запуска этого процесса (называемого «ядром») выполните следующую команду в командной строке вашей операционной системы:

```
$ jupyter lab
```

Эта команда запустит локальный веб-сервер, доступный браузеру, и сразу же начнет журналировать выполняемые действия. Журнал будет выглядеть следующим образом:

```
$ jupyter lab
[ServerApp] Serving notebooks from local directory: /Users/jakevdp/ \
PythonDataScienceHandbook
[ServerApp] Jupyter Server 1.4.1 is running at:
[ServerApp] http://localhost:8888/lab?token=dd852649
[ServerApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
```

Эта команда должна автоматически запустить браузер по умолчанию и открыть в нем указанный в выводе локальный URL; точный адрес зависит от вашей системы. Если браузер не запускается автоматически, то запустите его вручную и перейдите по указанному в выводе адресу (в примере это http://localhost:8888/lab).

# Справка и документация в IPython

Если вы не читали другие разделы в данной главе, прочитайте хотя бы этот. Обсуждаемые здесь утилиты (из IPython) внесли наибольший вклад в мой ежедневный процесс разработки.

Когда человека с техническим складом ума просят помочь другу, родственнику или коллеге решить проблему с компьютером, чаще всего речь идет об умении быстро найти неизвестное решение. В науке о данных все точно так же: веб-ресурсы с поддержкой поиска, такие как онлайн-документация, дискуссии в почтовых рассылках и ответы на сайте Stack Overflow, содержат массу информации, даже

если речь идет о теме, информацию по которой вы уже искали. Уметь эффективно исследовать данные означает скорее не запоминание утилит или команд, которые нужно использовать в каждой из возможных ситуаций, а знание того, как эффективно искать неизвестную пока информацию: посредством поиска в Интернете или с помощью других средств.

Одна из самых полезных возможностей IPython/Jupyter заключается в сокращении разрыва между пользователями и типом документации и поиска, что должно помочь им эффективнее выполнять свою работу. Хотя поиск в Интернете все еще играет важную роль в ответе на сложные вопросы, большое количество информации можно найти, используя саму оболочку IPython. Вот несколько примеров вопросов, на которые IPython может помочь ответить буквально с помощью нескольких нажатий клавиш.

- Как вызвать эту функцию? Какие параметры она имеет?
- Как выглядит исходный код этого объекта Python?
- Что имеется в импортированном мной пакете?
- Какие атрибуты или методы есть у этого объекта?

Далее мы обсудим инструменты IPython для быстрого доступа к этой информации, а именно символ `?` для просмотра документации, символы `??` для просмотра исходного кода и клавишу `Tab` для автодополнения.

## Доступ к документации с помощью символа ?

Язык программирования Python и его экосистема для исследования данных ориентированы на потребности клиента, и в значительной степени это проявляется в доступе к документации. Каждый объект Python содержит ссылку на строку, именуемую *docstring* (сокращение от *documentation string* — «строка документации»), которая в большинстве случаев будет содержать краткое описание объекта и способ его использования. В языке Python имеется встроенная функция `help`, позволяющая обращаться к этой информации и выводить результат. Например, вот как можно посмотреть документацию по встроенной функции `len`:

```
In [1]: help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container[1].
```

В зависимости от интерпретатора информация будет отображена в виде встраиваемого текста или в отдельном всплывающем окне.

[1] Возвращает количество элементов в контейнере. — *Здесь и далее примеч. пер.*

Поскольку поиск справочной информации по объекту — очень распространенное действие, оболочка IPython предоставляет символ ? для быстрого доступа к документации и другой соответствующей информации:

```
In [2]: len?
Signature: len(obj, /)
Docstring: Return the number of items in a container[1].
Type:      builtin_function_or_method
```

Данная нотация подходит практически для чего угодно, включая методы объектов:

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Signature: L.insert(index, object, /)
Docstring: Insert object before index[2].
Type:      builtin_function_or_method
```

или даже сами объекты с описанием их типов:

```
In [5]: L?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified[3].
```

Эта возможность поддерживается даже для созданных пользователем функций и других объектов! В следующем фрагменте кода мы опишем маленькую функцию с docstring:

```
In [6]: def square(a):
  ....:     """Return the square of a."""[4]
  ....:     return a ** 2
  ....:
```

Обратите внимание: чтобы создать docstring с описанием нашей функции, мы просто вставили в начало определения строковый литерал. Поскольку docstring обычно занимает несколько строк, в соответствии с соглашениями мы использовали тройные кавычки — нотацию языка Python для многострочных docstring.

---

[1] Возвращает количество элементов в контейнере.

[2] Вставляет object перед index.

[3] При вызове без аргументов конструктор создает новый пустой список. Передаваемый аргумент должен быть итерируемым объектом.

[4] Возвращает квадрат числа a.

Теперь воспользуемся знаком ? для поиска этой строки docstring:

```
In [7]: square?
Signature: square(a)
Docstring: Return the square of a.
File:      <ipython-input-6>
Type:      function
```

Быстрый доступ к документации через строки docstring — одна из причин, почему желательно приучить себя добавлять подобную встроенную документацию в создаваемый код!

## Доступ к исходному коду с помощью символов ??

Поскольку код на языке Python читается очень легко, простота доступа к исходному коду интересующего вас объекта может обеспечить более глубокое его понимание. Оболочка IPython предоставляет сокращенную форму получения исходного кода — двойной знак вопроса (??):

```
In [8]: square??
Signature: square(a)
Source:
def square(a):
    """Return the square of a."""
    return a ** 2
File:      <ipython-input-6>
Type:      function
```

Для подобных простых функций двойной знак вопроса позволяет быстро вникнуть в особенности внутренней реализации.

Немного поэкспериментировав, вы можете заметить, что иногда добавление ?? в конце не приводит к отображению исходного кода: обычно это объясняется тем, что рассматриваемый объект реализован не на языке Python, а на C или каком-либо другом компилируемом языке. В подобных случаях добавление ?? приводит к такому же результату, что и добавление ?. Вы столкнетесь с этим при исследовании многих встроенных объектов и типов Python, например упомянутой выше функции len:

```
In [9]: len??
Signature: len(obj, /)
Docstring: Return the number of items in a container[1].
Type:      builtin_function_or_method
```

Использование ? и/или ?? — простой способ для быстрого поиска информации о работе любой функции или модуля языка Python.

[1] Возвращает количество элементов в контейнере.

# Исследование содержимого модулей с помощью функции автодополнения

Другой удобный интерфейс оболочки IPython — клавиша Tab и вызываемая ею функция автодополнения, позволяющая исследовать содержимое объектов, модулей и пространств имен. В следующих примерах мы будем применять обозначение <TAB> там, где необходимо нажать клавишу Tab.

## Использование автодополнения для исследования содержимого объектов

Каждый объект в Python имеет множество различных атрибутов и методов. Помимо упоминавшейся выше функции `help`, в языке Python есть встроенная функция `dir`, возвращающая их список, но на практике интерфейс автодополнения по клавише Tab гораздо удобнее. Чтобы получить список всех доступных атрибутов объекта, необходимо набрать имя объекта, символ точки (`.`) и нажать клавишу Tab:

```
In [10]: L.<TAB>
           append() count    insert   reverse
           clear    extend   pop      sort
           copy     index    remove
```

Чтобы сократить список, можно набрать первый символ или несколько символов нужного имени и нажать клавишу Tab, после чего будут отображены соответствующие атрибуты и методы:

```
In [10]: L.c<TAB>
            clear() count()
            copy()

In [10]: L.co<TAB>
             copy()  count()
```

Если имеется только один вариант, нажатие клавиши Tab приведет к автодополнению строки. Например, эта последовательность символов будет немедленно заменена на `L.count`:

```
In [10]: L.cou<TAB>
```

В языке Python отсутствует четкое разграничение между открытыми/внешними и закрытыми/внутренними атрибутами, поэтому по соглашениям для обозначения закрытых методов их имена принято начинать с символа подчеркивания. По умолчанию закрытые, а также специальные методы исключаются из списка вариантов автодополнения, но их можно вывести, набрав знак подчеркивания:

```
In [10]: L._<TAB>
           __add__              __delattr__     __eq__
           __class__            __delitem__     __format__()
           __class_getitem__() __dir__()        __ge__             >
           __contains__         __doc__         __getattribute__
```

Для краткости я показал только несколько первых строк вывода. Большинство этих методов имеют специальное значение в языке Python, и их имена начинаются с двойного подчеркивания (на сленге называются dunder[1]-методами).

## Автодополнение в инструкциях импортирования

Функцию автодополнения удобно использовать также в инструкциях импорта для импортирования объектов из пакетов. Воспользуемся этой возможностью для поиска всех элементов с именами, начинающимися с `co`, доступных для импорта из пакета `itertools`:

```
In [10]: from itertools import co<TAB>
         combinations()                  compress()
         combinations_with_replacement() count()
```

Точно так же можно использовать функцию автодополнения по Tab для просмотра пакетов, доступных в системе для импорта (у вас результат может отличаться от приведенного ниже в зависимости от того, какие сторонние сценарии и модули являются видимыми в данном сеансе Python):

```
In [10]: import <TAB>
             abc                 anyio
             activate_this       appdirs
             aifc                appnope        >
             antigravity         argon2

In [10]: import h<TAB>
             hashlib html
             heapq   http
             hmac
```

## Помимо автодополнения: поиск с использованием шаблонного символа

Автодополнение по клавише Tab удобно, когда известны первые несколько символов в имени искомого объекта или атрибута. Однако эта функция малопригодна, когда нужно найти соответствие по символам, находящимся в середине или конце имени. На этот случай оболочка IPython предлагает возможность поиска соответствий с использованием шаблонного символа *.

---

[1] Игра слов: одновременно сокращение от double underscore — «двойное подчеркивание» и dunderhead — «тупица», «болван».

Например, вот как можно получить список всех объектов с именами, оканчивающимися на `Warning`:

```
In [10]: *Warning?
BytesWarning                  RuntimeWarning
DeprecationWarning            SyntaxWarning
FutureWarning                 UnicodeWarning
ImportWarning                 UserWarning
PendingDeprecationWarning     Warning
ResourceWarning
```

Обратите внимание, что символ * соответствует любой строке, включая пустую.

Аналогично предположим, что мы ищем строковый метод, содержащий где-то в имени слово `find`. Отыскать его можно так:

```
In [11]: str.*find*?
str.find
str.rfind
```

Я обнаружил, что подобный гибкий поиск с помощью шаблонных символов очень удобен для поиска нужной команды при знакомстве с новым пакетом или обращении после перерыва к уже знакомому.

# Горячие клавиши в командной оболочке IPython

Вероятно, все, кто проводит время за компьютером, используют в своей работе горячие клавиши. Наиболее известные — Cmd+C и Cmd+V (или Ctrl+C и Ctrl+V), применяемые для копирования и вставки в различных программах и системах. Опытные пользователи выбирают популярные текстовые редакторы, такие как Emacs, Vim и другие, позволяющие выполнять множество операций посредством замысловатых сочетаний клавиш.

В командной оболочке IPython также имеются горячие клавиши для быстрой навигации при наборе команд. Хотя некоторые из них работают в блокноте для браузера, данный раздел в основном касается горячих клавиш именно в командной оболочке IPython.

Привыкнув к сочетаниям горячих клавиш, вы сможете использовать их для быстрого выполнения команд без изменения исходного положения рук на клавиатуре. Если вы пользователь Emacs или имеете опыт работы с Linux-подобными командными оболочками, некоторые сочетания горячих клавиш покажутся вам знакомыми. Мы сгруппируем их в несколько категорий: *навигационные горячие клавиши*, *горячие клавиши ввода текста*, *горячие клавиши для истории команд* и *прочие горячие клавиши*.

## Навигационные горячие клавиши

Использовать стрелки «влево» (←) и «вправо» (→) для перемещения назад и вперед по строке вполне естественно, но есть и другие возможности, не требующие изменения исходного положения рук на клавиатуре (табл. 1.1).

**Таблица 1.1.** Навигационные горячие клавиши

| Комбинация клавиш | Действие |
|---|---|
| Ctrl+A | Перемещает курсор в начало строки |
| Ctrl+E | Перемещает курсор в конец строки |
| Ctrl+B (или стрелка «влево») | Перемещает курсор назад на один символ |
| Ctrl+F (или стрелка «вправо») | Перемещает курсор вперед на один символ |

## Горячие клавиши ввода текста

Для удаления предыдущего символа привычно использовать клавишу Backspace, несмотря на то что требуется небольшая гимнастика для пальцев, чтобы до нее дотянуться. Эта клавиша удаляет только один символ за раз. В оболочке IPython имеется несколько сочетаний горячих клавиш для удаления различных частей набираемого текста. Наиболее полезные из них — команды для удаления сразу целых строк текста (табл. 1.2). Вы поймете, что привыкли к ним, когда поймаете себя на использовании сочетания Ctrl+B и Ctrl+D вместо Backspace для удаления предыдущего символа!

**Таблица 1.2.** Горячие клавиши для ввода текста

| Комбинация клавиш | Действие |
|---|---|
| Backspace | Удаляет предыдущий символ в строке |
| Ctrl+D | Удаляет следующий символ в строке |
| Ctrl+K | Вырезает текст, начиная от курсора и до конца строки |
| Ctrl+U | Вырезает текст с начала строки до курсора |
| Ctrl+Y | Вставляет предварительно вырезанный текст |
| Ctrl+T | Меняет местами предыдущие два символа |

## Горячие клавиши для истории команд

Вероятно, наиболее важные из обсуждаемых здесь горячих клавиш в IPython — сочетания для навигации по истории команд. Данная история команд распространяется за пределы текущего сеанса оболочки IPython: полная история команд хранится в базе данных SQLite в каталоге с профилем IPython.

Простейший способ получить к ним доступ — с помощью стрелок «вверх» (↑) и «вниз» (↓) для пошагового перемещения по истории, но есть и другие варианты (табл. 1.3).

**Таблица 1.3.** Горячие клавиши для истории команд

| Комбинация клавиш | Действие |
|---|---|
| Ctrl+P (или стрелка «вверх») | Доступ к предыдущей команде в истории |
| Ctrl+N (или стрелка «вниз») | Доступ к следующей команде в истории |
| Ctrl+R | Поиск в обратном направлении по истории команд |

Особенно полезным может оказаться поиск в обратном направлении. Как вы помните, в предыдущем разделе мы определили функцию `square`. Выполним поиск в обратном направлении по нашей истории команд Python в новом окне оболочки IPython и найдем это описание снова. После нажатия Ctrl+R в терминале IPython вы должны увидеть следующее приглашение командной строки:

```
In [1]:
(reverse-i-search)`':
```

Если начать вводить символы в этом приглашении, IPython автоматически будет дополнять их, подставляя недавние команды, соответствующие этим символам, если такие существуют:

```
In [1]:
(reverse-i-search)`sqa': square??
```

Вы можете в любой момент добавить символы для уточнения поискового запроса или снова нажать Ctrl+R, чтобы отыскать следующую команду, соответствующую запросу. Если в процессе чтения предыдущего раздела вы выполняли все описанные там действия, то нажмите Ctrl+R еще два раза, и вы получите:

```
In [1]:
(reverse-i-search)`sqa': def square(a):
    """Return the square of a"""
    return a ** 2
```

Найдя искомую команду, нажмите Enter, и поиск завершится. После этого можно использовать найденную команду и продолжить работу в сеансе:

```
In [1]: def square(a):
    """Return the square of a"""
    return a ** 2

In [2]: square(2)
Out[2]: 4
```

Обратите внимание, что также можно использовать сочетания клавиш Ctrl+P/Ctrl+N или стрелки вверх/вниз для поиска по истории команд, но только по совпадающим символам в начале строки. Если вы введете `def` и нажмете Ctrl+P, в истории будет найдена последняя команда, начинающаяся с символов `def`, если таковая имеется.

## Прочие горячие клавиши

Имеется еще несколько сочетаний клавиш, не относящихся ни к одной из предыдущих категорий, но заслуживающих упоминания (табл. 1.4).

**Таблица 1.4.** Дополнительные горячие клавиши

| Комбинация клавиш | Действие |
|---|---|
| Ctrl+L | Очистить экран терминала |
| Ctrl+C (или стрелка «вниз») | Прервать выполнение текущей команды Python |
| Ctrl+D | Выйти из сеанса Ipython[1] |

Сочетание Ctrl+C особенно удобно при случайном запуске очень долго работающего задания.

Хотя использование некоторых сочетаний горячих клавиш может показаться утомительным, вскоре у вас появится соответствующая мышечная память и вы будете жалеть, что эти команды недоступны в других программах.

[1] В отличие от упомянутого выше идентичного сочетания горячих клавиш, это сочетание работает при пустой строке приглашения к вводу.