## ГЛАВАЗ.

**IPython: интерактивные**

**вычисления и среда разработки**

*Действуй, не делая. Работай, не прилагая усилий. Дмай о малом, как о большом, и о нескольких, как о многих. Работай иад трудной задачей, пока она еще проста; реализуй великую цель через множество небольших шагов.*

Лао Цзы

Меня часто спрашивают: «в какой среде разработки на Python вы работаете?». Почти всегда я отвечаю «IPython и текстовый редактор». Вы можете заменить тектовый редактор интегрированной средой разработки (IDE), если хотите иметь графические инструменты и средства автоматического завершения кода. Но и в этом случае я советую не отказываться от IPython. Некоторые IDE даже включают интеграцию с IPython, так что вы получаете лучшее из обоих миров.

Проект *IPython* в 2001 году основал Фернандо Перес как побочный продукт по ходу создания усовершенствованного интерактивного интерпретатора Python. Впоследствии он превратился в один из самых важных инструментов в арсенале ученых, работающих на Python. Сам по себе он не предлагает ни вычислительных, ни аналитических средств, но изначально спроектирован с целью повысить продуктивность интерактивных вычислений и разработки ПО. В его основе лежит последовательность действий «выполни и посмотри» вместо типичной для многих языков «отредактируй, откомпилируй и запусти». Он также очень тесно интегрирован с оболочкой операционной системы и с файловой системой. Посколь­ ку анализ данных подразумевает исследовательскую работу, применение метода проб и ошибок и итеративный подход, то IPython почти во всех случаях позволяет ускорить выполнение работы.

Разумеется, сегодняшний IPython - это куда больше, чем просто усовершен­ спюванная интерактивная оболочка Python. В его состав входит развитая графи­ ческая консоль с встроенными средствами построения графиков, интерактивный веб-блоюют и облегченный движок для быстрых параллельных вычислений. И подобно многим другим инструментам, созданным программистами для программистов, он настраивается в очень широких пределах. Некоторые из вышеупо­ мянутых возможностей будут рассмотрены ниже.

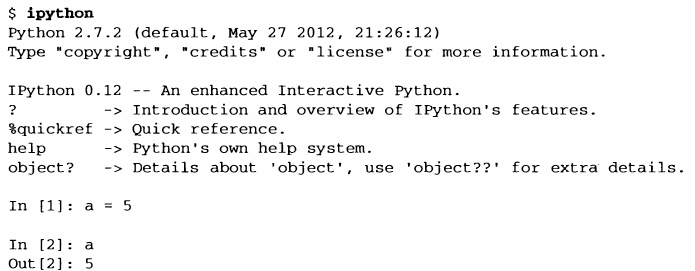
Поскольку интерактивность неотъемлема от IPython, некоторые описываемые далее возможности трудно в полной мере продемонстрировать без «живой» консоли. Если вы читаете об IPython впервые, я рекомендую параллельно прорабатывать примеры, чтобы понять, как все это работает на практике. Как и в любой среде, основанной на активном использовании клавиатуры, для полного освоения инструмента необходимо запомнить комбинации клавиш для наиболее употребительных команд.



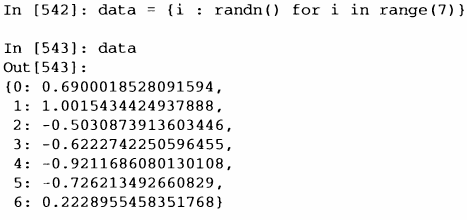
При первом чтении многие части этой главы (к примеру, профилирование и отладка) можно пропустить без ущерба для понимания последующего материала. Эта глава задуман как независимый, достаточно полный обзор функциональности IPython.

##### Основы IPython

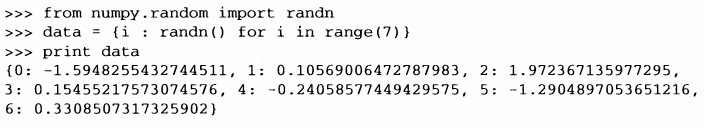
IPython можно запустить из командной строки, как и стандартный интерпретатор Python, только для этого служит команда ipython:



Чтобы выполнить произвольное предложение Python, нужно ввести его и на­ жать клавишу Enter. Если ввести только имя переменной, то IPython выведет строковое представление объекта:

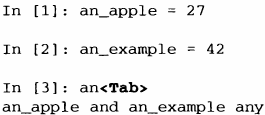


Многие объекты Python форматируются для удобства чтения; такая красивая печать отличается от обычного представления методом print. Тот же словарь, на­ печатанный в стандартном интерпретаторе Python, выглядел бы куда менее презентабельно:

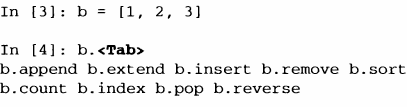
 IPython предоставляет также средства для исполнения произвольных блоков кода (путем копирования и вставки) и целых Руthоn-скриптов. Эти вопросы будут рассмотрены чуть ниже.

***Завершение по нажатию клавиши Таb***

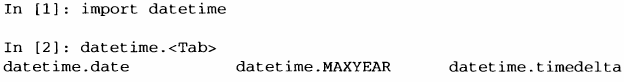
На первый взгляд, оболочка IPython очень похожа на стандартный интерпрета­ тор Python с мелкими косметическими изменениями. Пользователям программы Mathematica знакомы пронумерованные строки ввода и вывода. Одно из суще­ ственных преимуществ над стандартной оболочкой Python - завершение по нажатию клавиши ТаЬ, функция, реализованная в большинстве интерактивных сред анализа данных. Если во время ввода выражения нажать <Таb>, то оболочка про­ изведет поиск в пространстве имен всех переменных (объектов, функций и т. д.), имена которых начинаются с введенной к этому моменту строки:



Обратите внимание, что IPython вывел обе определенные выше переменные, а также ключевое слово Python and и встроенную функцию any. Естественно, можно также завершать имена методов и атрибутов любого объекта, если предварительно ввести точку:



То же самое относится и к модулям:

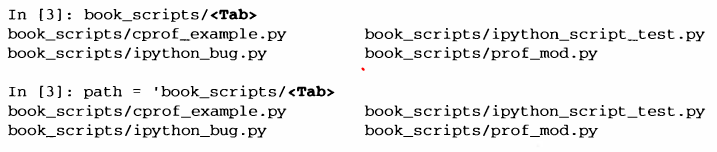


.



Отметим, что JPython по умолчанию скрывает методы и атрибуты, начинающиеся знаком подчеркивания, например магические методы и внутренние «закрытые» методы и атрибуты, чтобы не загромождать экран (и не смущать неопытных пользователей). На них автозавершение также распространяется, нужно только сначала набрать знак подчеркивания. Если вы предпочитаете всегда видеть такие методы при автозавершении, измените соответствующий режим в конфигурационном файле JPython.

Завершение по нажатию Таb работает во многих контекстах, помимо поиска в интерактивном пространстве имен и завершения атрибутов объекта или модуля. Если нажать <Таb> при вводе чего-то, похожего на путь к файлу (даже внутри строки Python), то будет произведен поиск в файловой системе:

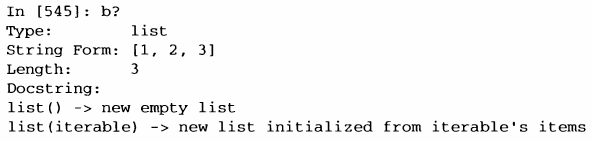


В сочетании с командой %run (см. ниже) эта функция несомненно позволит вам меньше лупить по клавиатуре.

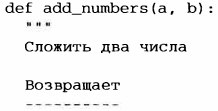
Автозавершение позволяет также сэкономить время при вводе именованных аргументов функции (в том числе и самого знака=).

***Интроспекция***

Если ввести вопросительный знак (?) до или после имени переменной, то будет напечатана общая информация об объекте:

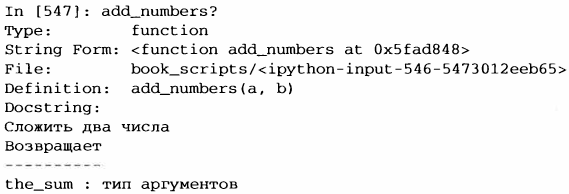


Это называется интроспекцией обьекта*.* Если объект предствляет собой функцию или метод экземпляра, то будет показана строка документации, если она существует. Допустим, мы написали такую функцию:

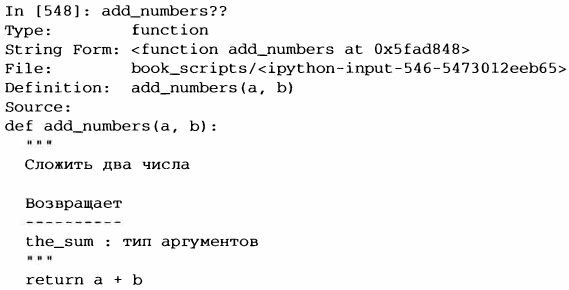




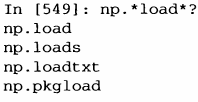
Тогда при вводе знака? мы увидим строку документации:



Два вопросительных знака ?? покажут также исходный код функции, если это возможно:



И последнее применение? - поиск в пространстве имен IPython по аналогии со стандартной командной строкой UNIX или Windows. Если ввести несколько символов в сочетании с метасимволом \*, то будут показаны все имена по указанной маске. Например, вот как можно получить список всех функций в пространстве имен верхнего уровня NumPy, имена которых содержат строку load:



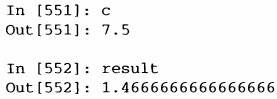
***Команда %run***

Команда %run позволяет выполнить любой файл как Руthоn-программу в кон­ тексте текущего сеанса IPython. Предположим, что в файле ipython\_script\_test .ру хранится такой простенький скрипт:

Этот скрипт можно выполнить, передав имя файла команде % run:



Скрипт выполняется в *nycmoм пространстве имен* (в которое ничего пе импор тировано и в котором не определены пикакие переменные), поэтому его поведение должно быть идентично тому, что получается при запуске программы из командной строки командой python script.ру. Вес переменные (импортированные, функции, глобальные объекты), определенные в файле (до момента исключения, если таковое произойдет), будут доступны оболочке IPython:



Если Руthоn-скрипт ожидает передачи аргументов из командной строки (которые должны попасть в массив sys.argv), то их можно перечислить после пути к файлу, как в командной строке.



Если вы хотите дать скрипту доступ к переменным, уже определенным в интерактивном пространстве имен IPython, используйте команду

%run - i, а не просто %run.

**Прерывание выполняемой программы**

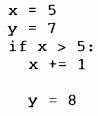
Нажатие <Ctrl-C> во время выполнения кода, запущенного с помощью %run, или просто долго работающей программы, приводит к возбуждению исключе­ ния Keyboardinterrupt. В этом случае почти все Руthоn-программы немедленно прекращают работу, если только не возникло очень редкое стечение обстоя­ тельств.

|  |  |
| --- | --- |
|  | Если Руthоn-код вызвал откомпилированный модуль расширения, то нажатие <Сtrl-С> не всегда приводит к немедленному завершению. В таких случаях нужно либо дождаться возврата управления интерпретатору Python, либо - если случилось что-то ужасное - принудительно снять процесс Python с помощью диспетчера задач ОС. |

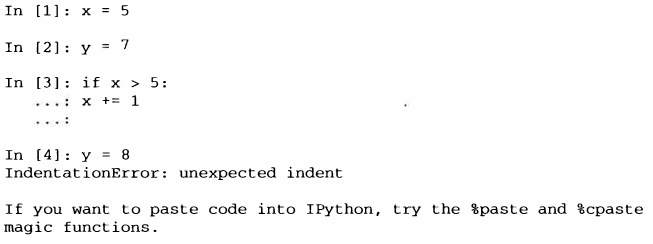
***Исполнение кода из буфера обмена***

Когда нужно быстро и без хлопот выполнить код в IPython, можно просто взять его из буфера обмена. На первый взгляд, неряшливо, но на практике очень полезно. Например, при разработке сложного или долго работающего приложения иногда желательно исполнять скрипт по частям, останавливаясь после каждого шага, чтобы проверить загруженные данные и результаты. Или, допустим, вы наш­ ли какой-то фрагмент кода в Интернете и хотите поэкспериментировать с ним, не создавая новый ру-файл.

Для извлечения фрагмента кода из буфера обмена во многих случаях достаточ­ но нажать <Ctrl-Shift-V>. Отметим, что это не стопроцентно надежный способ, потому в таком режиме имитируется ввод каждой строки в IPython, а символы но­ вой строки трактуются как нажатие <Enter>. Это означает, что если извлекается код, содержащий блок с отступом, и в нем присутствует пустая строка, то IPython будет считать, что блок закончился. При выполнении следующей строки в блоке возникнет исключение IndentationErтor. Например, такой код:

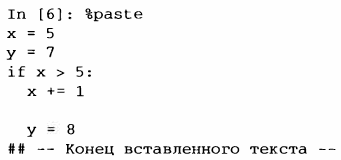


после вставки из буфера обмена работать не будет:



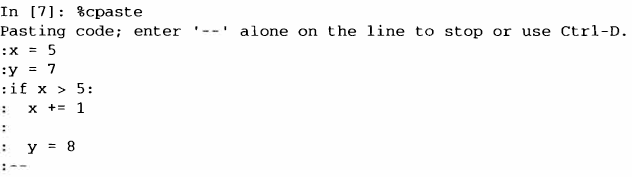
В сообщении об ошибке предлагается использовать магические функции

%paste и %cpaste. Функция %paste принимает текст, находящийся в буфере обмена, и исполняет его в оболочке как единый блок:



|  |  |
| --- | --- |
|  | В зависимости от платформы и способа установки Python может случиться, что функция %paste откажется работать, хотя это маловероятно. В пакетных дистрибутивах типа EPDFree (описан во введении) такой проблемы быть не должно. |

Функция %cpaste аналогична, но выводит специальное приглашение для вставки кода:

 При использовании %cpaste вы можете вставить сколько угодно кода, перед тем как начать его выполнение. Например, %cpaste может пригодиться, сели вы хотите посмотреть па вставленный код до выполнения. Если окажется, что слу­ чайно вставлен не тот код, то из %cpaste можно выйти нажатием <Ctrl-C>.

Ниже мы познакомимся с НТМL-блокнотом IPython, который выводит по­ блочный анализ на новый уровень - с помощью работающего в браузсре блокнота с исполняемыми ячейками, содержащими код.

**Взаимодействие IPython с редакторами и IDE**

Для некоторых текстовых редакторов, например Emacs и vim, существуют рас­ ширения, позволяющие отправлять блоки кода напрямую из редактора в запу­ щенную оболочку IPython. Для получения дополнительных сведений зайдите на сайт IPython или поищите в Интернете.

Для некоторых IDE существуют подключаемые модули, например PyDev для Eclipse или Python Tools для Visual Studio от Microsoft (а, возможно, и для дру­ гих), обеспечивающие интеграцию с консольным приложением IPython. Если вы хотите работать в IDE, не отказываясь от консольных функций IPython, то это может стать удачным решением.

***Комбинации клавиш***

В IPython есть много комбинаций клавиш для навигации по командной стро­ ке (они знакомы пользователям текстового редактора Emacs или оболочки UNIX bash) и взаимодействия с историей команд (см. следующий раздел). В

табл. 3.1 перечислены наиболее употребительные комбинации, а на рис. 3.1 некоторые из них, например перемещение курсора, проиллюстрированы.

|  |  |
| --- | --- |
| **Команда** | **Описание** |
| Ctrl-Р или стрелка-вверх | Просматривать историю команд назад в поисках команд, начинающихся с введенной строки |
| Ctrl-N или стрелка-вниз | Просматривать историю команд вперед в поисках команд, начинающихся с введенной строки |
| Ctrl-R | Обратный поиск в истории в духе Readline (частичное соответствие) |
| Ctrl-Shift-V | Вставить текст из буфера обмена |
| Ctrl-C | Прервать исполнение программы |
| Ctrl-A | Переместить курсор в начало строки |
| Ctrl- Е | Переместить курсор в конец строки |
| Ctrl- K | Удалить текст от курсора до конца строки |
| Ctrl-U | Отбросить весь текст в текущей строке |
| Ctrl-F | Переместить курсор на один символ вперед |
| Ctrl -B | Переместить курсор на один символ назад |
| Ctrl-L | Очистить экран |

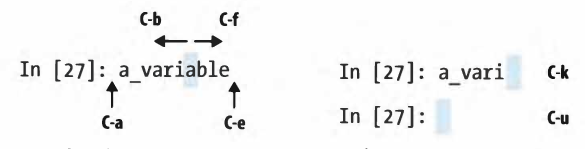


Рис. 3.1. Иллюстрация некоторых комбинаций клавиш IPython

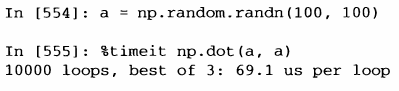
***Исключения и обратная трассировка***

Если при выполнении любого предложения или скрипта, запущенного коман­ дой %run, возникнет исключение, то по умолчанию IPython напечатает все содержимое стека вызовов (обратную трассировку), сопроводив каждый вызов несколькими строками контекста.

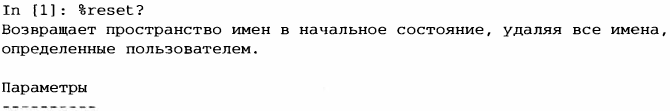
Наличие дополнительного контекста уже само по себе является большим пре­ имуществом по сравнению со стандартным интерпретатором Python (который не выводит контекст). Объемом контекста можно управлять с помощью магической команды %xmode - от минимального (как в стандартном интерпретаторе Python) до подробного (когда включаются значения аргументов функции и другая информация). Ниже в этой главе мы увидим, что можно перемещаться по стеку (с по­ мощью магических команд %debug и %pdb) после ошибки, это дает возможность производить посмертную отладку.

***Магические команды***

В IPython есть много специальных команд, называемых «магическими». цель которых - упростить решение типичных задач и облегчить контроль над поведением всей системы IPython. Магической называется команда, которой предшествует знак процента %. Например, магическая функция %timeit (мы подробно рассмотрим ее ниже) позволяет замерить время выполнения любого предложения Python, например умножения матриц:



Магические команды можно рассматривать как командные утилиты, исполня­ емые внутри IPython. У многих из них имеются дополнительные параметры «командной строки». список которых можно распечатать с помощью? (вы ведь так и думали, правда?)':



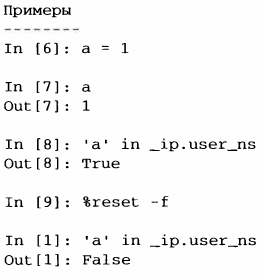
1 Сообщения выводятся на английском языке, но для удобства читателя переведены. - ***При.м. перев.***

**Основы IPython**

-f принудительная очистка без запроса подтверждения.

-s 'Мягкая' очистка: очищается только ваше пространство имен, а история остается. Ссылки на объекты можно удержать. По умолчанию (без этого параметра) выполняется 'жесткая' очистка, в результате чего вы получаете новый сеанс, а все ссылки на объекты в текущем сеансе удаляются.

Примеры



Магические функции по умолчанию можно использовать и без знака процента, если только нигде не определена переменная с таким же именем, как у магической функции. Этот режим называется *,* его можно включить или выключить с помощью функции %automagic.

Поскольку к документации по IPython легко можно обратиться из системы, я рекомендую изучить все имеющиеся специальные команды, набрав

%quickref или %magic. Я расскажу только о нескольких наиболее важных для продуктивной работы в области интерактивных вычислений и разработки в среде IPython.

Таблица 3.2. Часто используемые магические команды IPython

**Команда**

**Описание**

%quickref

%magic

%debug

%hist

%pdb

%paste

%cpaste

Вывести краткую справку по IPython

Вывести подробную документацию по всем имеющимся магическим командам

Войти в интерактивный отладчик в точке последнего вызова, показанного в обратной трассировке исключения

Напечатать историю введенных команд (по желанию вместе с результатами)

Автоматически входить в отладчик после любого исключения

Выполнить отформатированный Python-кoд, находящийся в буфере обмена

Открыть специальное приглашение для ручной вставки Python­ кoдa, подлежащего выполнению

**Команда**

%reset

**Описание**

Удалить все переменные и прочие имена, определенные в интерактивном пространстве имен

%page *OBJECT* Сформировать красиво отформатированное представление

объекта и вывести его постранично

%run *script* .ру Выполнить Руthоn-скрипт из IPython

%prun *предложение* Выполнить *предложение* под управлением cProfile и вывести

результаты профилирования

% time *предложение* Показать время выполнения одного предложения

%timeit *предложение* Выполнить предложение несколько раз и усреднить время

выполнения. Полезно для хронометража кода, который выполняется очень быстро

%who, %who\_ls, %whos Вывести переменные, определенные в интерактивном

пространстве имен, с различной степенью детализации

%xdel *переменная* Удалить переменную и попытаться очистить все ссылки на объект во внутренних структурах данных IPython

***Графическая консоль на базе Qt***

Команда IPython разработала графическую консоль на базе библиотеки Qt (рис. 3.2), цель которой - скрестить возможность чисто консольного приложения со средствами, предоставляемыми виджетом обогащенного текста, в том числе встраиваемыми изображениями, режимом многострострочного редактирования и подсветкой синтаксиса. Если на вашей машине установлен пакет PyQt или PySide, то можно запустить приложение с встроенным построением графиков:

*ipython qtconsole --pylab=inline*

Qt-консоль позволяет запускать несколько процессов IPython в отдельных вкладках и тем самым переключаться с одной задачи на другую. Она также может разделять процесс с НТМL-блокнотом IPython, о котором я расскажу ниже.

***Интеграция* с *matplotlib и режим ру/аЬ***

IPython так широко используется в научном сообществе отчасти потому, что он спроектирован как дополнение к библиотекам типа matplotlib и другим графи­ ческим инструментам. Если вы раньше никогда не работали с matplotlib, ничего страшного; ниже мы обсудим эту библиотеку во всех подробностях. Создав окно графика matplotlib в стандартной оболочке Python, вы будете неприятно пораже­ ны тем, что цикл обработки событий ГИП «перехватывает контроль» над сеансом

Python до тех пор, пока окно не будет закрыто. Для интерактивного анализа данных и визуализации это не годится, поэтому в IPython реализована специальная логика для всех распространенных библиотек построения ГИП, так чтобы обеспе­ чить органичную интеграцию с оболочкой

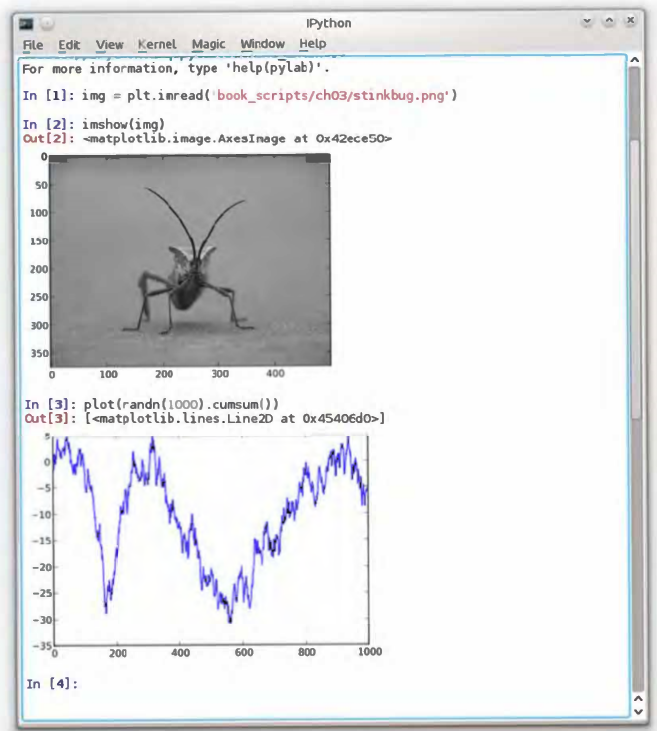


Рис. 3.2. Qt-консоль IPython

Для запуска IPython в режиме интеграции с matplotlib достаточно просто добавить флаг --pylab (дефисов должно быть два).

$ ipython --pylab

При этом произойдет несколь ко вещей. Во-первых, IPython запустится в ре­ жиме интеграции с ГИП по умолчанию, что позволит без проблем создавать окна графиков matplotlib. Во-вторых, в интерактивное пространство имен верхнего уровня будет импортирована большая часть NumPy и matplotlib, в результате чего создается среда интерактивных вычислений, напоминающая MATLAB и другие

предметно-ориентированные среды (рис. 3.3). Такую же конфигурацию можно за дать и вручную, воспользовавшись командой %gui (наберите %gui ?, чтобы узнать, как).

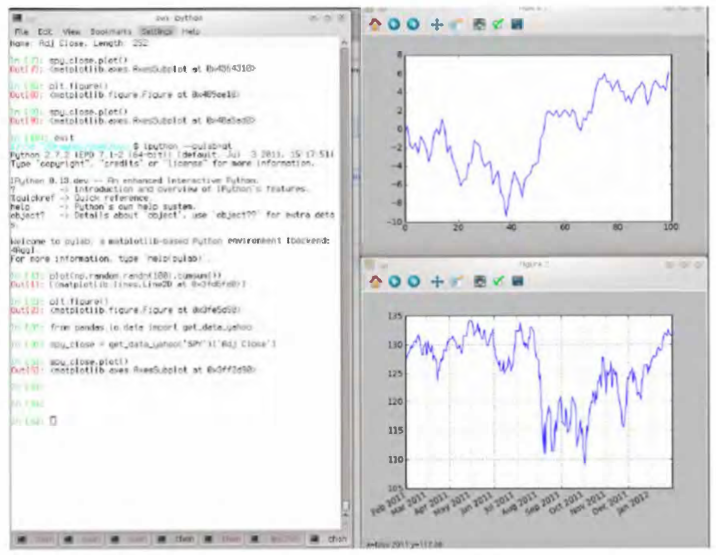


Рис. 3.3. Режим Pylab: IPython с окнами matplotlib

История команд

IPython поддерживает небольшую базу данных на диске, в которой хранятся тек сты всех выполненных команд. Она служит нескольким целям:

* поиск, автозавершение и повторное выполнение ранее выполненных команд с минимальными усилиями;
* сохранение истории команд между сеансами;
* протоколирование истории ввода-вывода в файле.

***Поиск в истории команд и повторное***

***выполнение***

Возможность искать и повторно выполнять предыдущие команды для многих является самой полезной функцией. Поскольку IPython рассчитан на итеративную и интерактивную разработку кода, мы часто повторяем одни и те же команды, например %run. Допустим вы выполнили такую команду:

In[7]: %run first/second/third/data\_script.py

и, ознакомившись с результатами работы скрипта (в предположении, что он за­ вершился успешно), обнаружили ошибку в вычислениях. Разобравшись, в чем проблема, и исправив скрипт data\_script.ру, вы можете набрать несколько первых букв команды %run и нажать <Ctrl-P> или клавишу <стрелка вверх>. В ответ IPython найдет в истории команд первую из предшествующих команд, начинающуюся введенными буквами. При повторном нажатии <Ctrl-P> или

<стрелки вверх> поиск будет продолжен. Если вы проскочили мимо нужной ко­ манды, ничего страшного. По истории команд можно перемещаться и *вперед* с по­ мощью клавиш <Ctrl-N> или <стрелка вниз>. Стоит только попробовать, и вы начнете нажимать эти клавиши, не задумываясь.

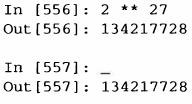
Комбинация клавиш <Ctrl-·R> дает ту же возможность частичного инкремент­ ного поиска, что подсистема readl ine, применяемая в оболочках UNIX, например bash. В Windows функциональность readline реализуется самим IPython. Что­ бы воспользоваться ей, нажмите <Ctr 1-R>, а затем введите несколько символов, встречающихся в искомой строке ввода:



Нажатие <Ctrl-R> приводит к циклическому просмотру истории в поисках строк, соответствующих введенным символам.

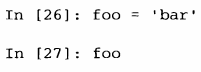
***Входные и выходные переменные***

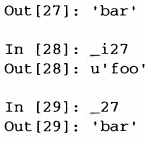
Забыв присвоить результат вызова функции, вы можете горько пожалеть об этом. По счастью, IPython сохраняет ссылки как на входные команды (набранный вами текст), так и на выходные объекты в специальных переменных. Последний и предпоследний выходной объект хранятся соответственно в переменных \_ (один подчерк) и\_ (два подчерка):



Входные команды хранятся в переменных с именами вида \_ix, где х - номер входной строки. Каждой такой входной переменной соответствует выходная пере­менная \_х. Поэтому после ввода строки 27 будут созданы две новых переменных

\_27 (для хранения выходного объекта) и \_i27 (для хранения входной команды).





Поскольку входные переменные - это строки, то их можно повторно вычислить с помощью ключевого слова Python ехес:



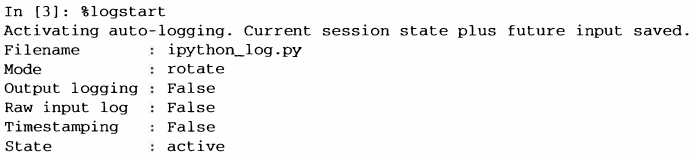
Есть несколько магических функций, позволяющих работать с историей ввода и вывода. Функция %hist умеет покаэывать историю ввода полностью или частично, с номерами строк или без них. Функция %reset очищает интерактивное пространство имен и факультативно кэши ввода и nьшода. Функция %xdel уда­ ляет все ссылки на коикретпый объект из внутренних структур данных IPython. Подробнее см. документацию по этим функциям.

|  |  |
| --- | --- |
|  | Работая с очень большими наборами данных, имейте в виду, что объекты, хранящиеся в истории ввода-вывода IPython, не могут быть удалены из памяти сборщиком мусора - даже если вы удалите соответствующую пе­ ременную из интерактивного пространства имен встроенным оператором del. В таких случаях команды %xdel и %reset помогут избежать проблем с памятью. |

***Протоколирование ввода-вывода***

IPython умеет протоколировать весь консольный сеанс, в том числе ввод и вы­

вод. Режим протоколирования включается командой %logstart:



Включить протоколирование IPython можно в любое время, и записан будет весь сеанс (в том числе и ранее выполненные команды). Таким образом, если в процессе работы вы решите сохранить все сделанное к этому моменту, достаточно будет включить протоколирование. Дополнительные параметры описаны в строке документации по функции %logstart (в частности, как изменить путь к файлу журнала) и связанным с ней функциям %logoff, %logon, %logstate и

%logstop.

#### Взаимодействие с операционной системой

Еще одна важная особенность IPython - очень тесная интеграция с оболочкой операционной системы. Среди прочего это означает, что многие стандартные дей­ ствия в командной строке можно выполнять в точности так же, как в оболочке Windows или UNIX (Linux, OS Х), не выходя из IPython. Речь идет о выполнении команд оболочки, смене рабочего каталога и сохранении результатов команды в объекте Python (строке или списке). Существуют также простые средства для за­ дания псевдонимов команд оболочки и создания закладок на каталоги.

Перечень магических функций и синтаксис вызова команд оболочки представ­ лены в табл. 3.3. В следующих разделах я кратко расскажу о них.

**Таблица 3.3.** Команды IPython, относящиеся к операционной системе

**Команда**

!cmd

output = !cmd args

%alias alias\_name cmd

%bookmark

%cd каталог

%pwd

%pushd каталог

%popd

%dirs

%dhist

%env

**Описание**

Выполнить команду в оболочке системы

Выполнить команду и сохранить в объекте output все выведенное на стандартный вывод

Определить псевдоним команды оболочки Воспользоваться системой закладок IPython Сделать указанный каталог рабочим Вернуть текущий рабочий каталог

Поместить текущий каталог в стек и перейти в указанный

каталог

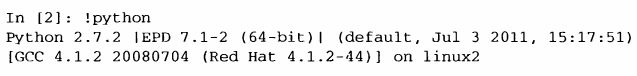
Извлечь каталог из стека и перейти в него

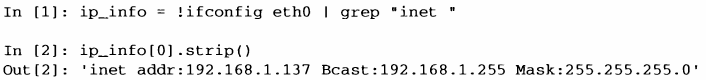
Вернуть список, содержащий текущее состояние стека каталогов

Напечатать историю посещения каталогов Вернуть переменные среды в виде слова

***Команды оболочки и псевдонимы***

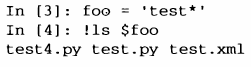
Восклицательный знак ! в начале командной строки IPython означает, что все следующее за ним следует выполнить n оболочке системы. Таким образом можно удалять файлы (командой rm или del в зависимости от ОС), изменять рабочий каталог или исполнять другой процесс. Можно даже запустить процесс, который перенимает управление у IPython, даже еще один интерпретатор Python:

  Все, что команда выводит на консоль, можно сохранить в переменной, присвоив ей значение выражения, начинающегося со знака ! . Например, на своей Linuх­ машине, подключенной к Интернету Еthеrnеt-кабелем, я могу следующим обра­ зом записать в переменную Python свой IР-адрес:

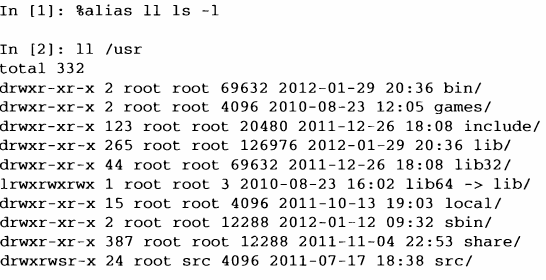


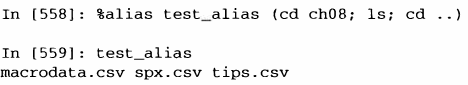
Возвращенный объект Python ip\_info - это специализированный список, со­ держащий различные варианты вывода на консоль.

IPython умеет также подставлять в команды, начинающиеся знаком ! , значения переменных Python, определенных в текущем окружении. Для этого имени пере­ менной нужно предпослать знак $:



Магическая функция %alias позволяет определять собственные сокращения для команд оболочки, например:

 Несколько команд можно выполнить, как одну, разделив их точками с запятой:



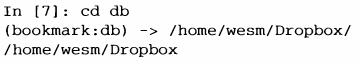
Обратите внимание, что IPython «забывает» все определенные интерактивно псевдонимы после закрытия сеанса. Чтобы создать постоянные псевдонимы, нужно прибегнуть к системе конфигурирования. Она описывается ниже в этой главе

***Система закладок на каталоги***

В IPython имеется простая система закладок, позволяющая создавать псевдо­ нимы часто используемых каталогов, чтобы упростить переход в них. Например, я регулярно захожу в каталог DгорЬох, поэтому могу определить закладку, которая даст возможность быстро перейти в него:



После этого с помощью магической команды %cd я смогу воспользоваться ранее определенными закладками:



Если имя закладки конфликтует с именем подкаталога вашего текущего рабо­ чего каталога, то с помощью флага - ь можно отдать приоритет закладке. Команда

%bookmark с флагом -1 вьшодит список всех закладок:



Закладки, в отличие от псевдонимов, автоматически сохраняются после закры­ тия сеанса.

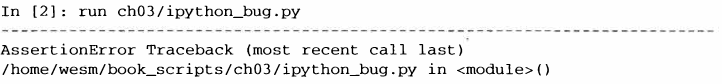
Средства разработки программ

IPyt]юn не только является удобной средой для интерактивных вычислений и исследования данных, но и прекрасно оснащен для разработки программ. В при­ ложениях для анализа данных; прежде всего, важно, чтобы код был *правилъным.* К счастью, в IPython встроен отлично интегрированный и улучшенный отладчик Python pdb. Кроме того, код должен быть *быстрым.* Для этого в IPython имеются простые в использовании средства хронометража и профилирования. Ниже я рас­ скажу об этих инструментах подробнее.

***Интерактивный отладчик***

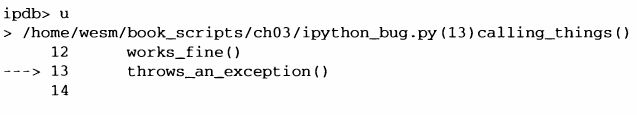
Отладчик IPython дополняет pdb завершением по нажатию клавиши ТаЬ, подсветкой синтаксиса и контекстом для каждой строки трассировки исключения. Отлаживать программу лучше всего сразу после возникновения ошибки. Команда

%debug, выполненная сразу после исключения, вызывает «посмертный» отладчик и переходит в то место стека вызовов, где было возбуждено исключение:





Находясь в отладчике, можно выполнять произвольный Python-кoд и просма­ тривать все объекты и данные (которые интерпретатор «сохранил живыми») в каждом кадре стека. По умолчанию отладчик оказывается на самом нижнем уров­ не - там, где произошла ошибка. Клавиши u (вверх) и d (вниз) позволяют пере­ ходить с одного уровня стека на другой:

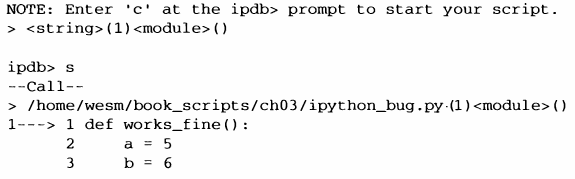


Команда %pdb устанавливает режим, в котором IPython автоматически вызы­ вает отладчик после любого исключения, многие считают этот режим особенно полезным.

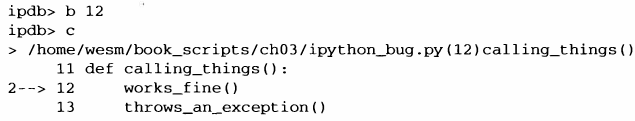
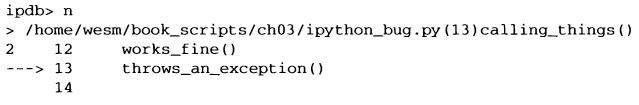
Отладчик также помогает разрабатывать код, особенно когда хочется расста­ вить точки останова либо пройти функцию или скрипт в пошаговом режиме, из­ учая состояния после каждого шага. Сделать это можно несколькими способами. Первый - воспользоваться функцией %run с флагом -d, которая вызывает отлад­ чик, перед тем как начать выполнение кода в

переданном скрипте. Для входа в скрипт нужно сразу же нажать s (step - пошаговый режим):

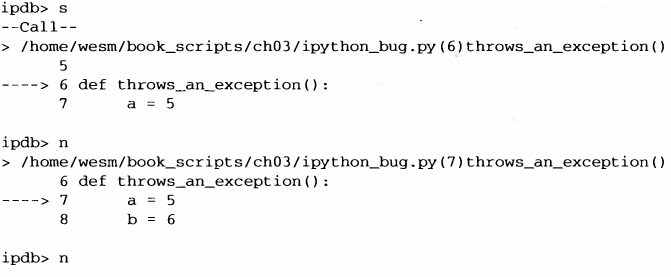


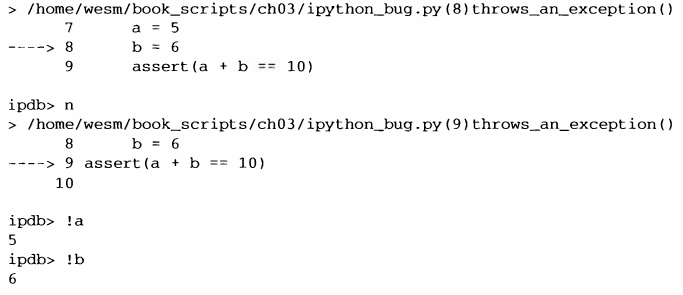


После этого вы сами решаете, каким образом работать с файлом. Например, в приведенном выше примере исключения можно было бы поставить точку оста­ нова прямо перед вызовом метода works\_fine и выполнить программу до этой точки, нажав с (continue - продолжить):

 В этот момент можно войти внутрь works\_fine ( ) командой step или выполнить works\_fine() без захода внутрь, т. е. перейти к следующей строке, нажав n (next - дальше): 

Далее мы можем войти внутрь throws\_an\_exception, дойти до строки, где воз­ никает ошибка, и изучить переменные в текущей области видимости. Отметим, что у команд отладчика больший приоритет, чем у имен переменных, поэтому для просмотра переменной с таким же именем, как у команды, необходимо предпо­ слать ей знак ! .





Уверенное владение интерактивным отладчиком приходит с опытом и практи­кой. В табл. 3.3 приведен полный перечень команд отладчика. Если вы привыкли к IDE, то консольный отладчик на первых порах может показаться неуклюжим, но со временем это впечатление рассеется. В большинстве IDE для Python имеются отличные графические отладчики, но обычно отладка в самом IPython оказывает­ ся намного продуктивнее.

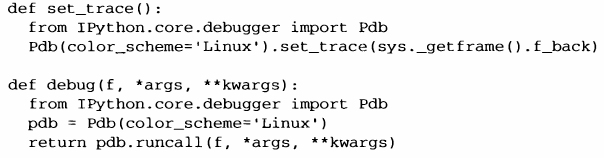
**Таблица 3.4.** Команда отладчика **(l)Python**

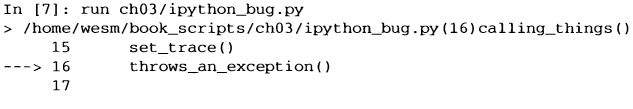
|  |  |
| --- | --- |
| **Команда** | **Действие** |
| h(elp) | Вывести список команд |
| help команда | Показать документацию по команде |
| c(ontinue) | Продолжить выполнение программы |
| q(uit) | Выйти из отладчика, прекратив выполнение кода |
| b(reak) номер | Поставить точку остановка на строке с указанным номером в текущем файле |
| b путъ/к/файлу.ру:номер | Поставить точку остановка на строке с указанным номером в указанном файле |
| s(tep) | Войти внутрь функции |
| n(ext) | Выполнить текущую строку и перейти к следующей на текущем уровне |
| u (р) / d (own) | Перемещение вверх и вниз по стеку вызовов |
| a(rgs) | Показать аргументы текущей функции |
| debug предложение | Выполнить предложение в новом (вложенном) отладчике |
| l(ist) предложение | Выполнить предложение в новом (вложенном) отладчике |
| w(here) | Распечатать весь стек в контексте текущей позиции |

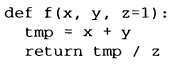
**Средства разработки программ**

**Другие способы работы с отладчиком**

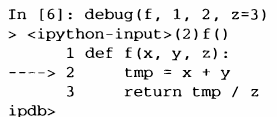
Существует еще два полезных способа вызова отладчика. Первый - восполь­ зоваться специальной функцией set\_trace (названной так по аналогии с pdb. set\_trace), которая по существу является упрощенным вариантом точки оста­ нова. Вот два небольших фрагмента, которые вы можете сохранить где-нибудь и использовать в разных программах (я, например, помещаю их в свой профиль IPython):

 Первая функция, set\_trace, совсем простая. Вызывайте ее в той точке кода, где хотели бы остановиться и оглядеться (например, прямо перед строкой, в которой происходит исключение):

 При нажатии с (продолжить) выполнение программы возобновится без каких­ либо побочных эффектов. Функция debug позволяет вызвать интерактивный отладчик в момент обращения к любой функции. Допустим, мы написали такую функцию:

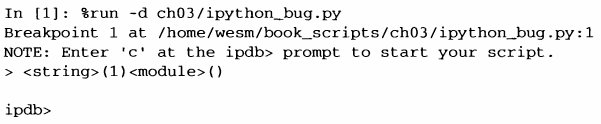


и хотели бы пройти ее в пошаговом режиме. Обычно f используется пример­ но так: f ( 1, 2, z = З). А чтобы войти в эту функцию, передайте f в качестве первого аргумента функции debug, а затем ее позиционные и именованные ар­ гументы:



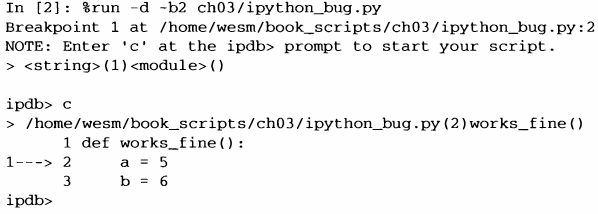
Мне эти две простенькие функции ежедневно экономят уйму времени. Нако­ нец, отладчик можно использовать в сочетании с функцией %run. Запустив скрипт

командой %run -d, вы попадете прямо в отладчик и сможете расставить точки останова и начать выполнение:



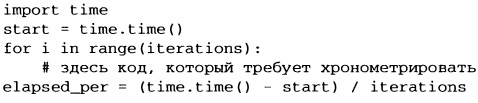
Если добавить еще флаг -b, указав номер строки, то после входа в отладчик на

этой строке уже будет стоять точка останова:

***Хронометраж программы: %time и %timeit***

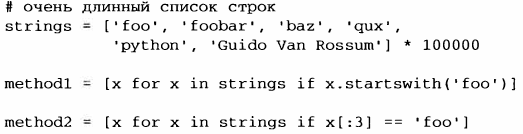
Для больших или долго работающих аналитических приложений бывает же­ лательно измерить время выполнения различных участков кода или даже от­ дельных предложений или вызовов функций. Интересно получить отчет о том, какие функции занимают больше всего времени в сложном процессе. По счастью, IPython позволяет без труда получить эту информацию по ходу разработки и тестирования программы.

Ручной хронометраж с помощью встроенного модуля time и его функций time. clock и time. time зачастую оказывается скучной и утомительной процедурой, поскольку приходится писать один и тот же неинтересный код:

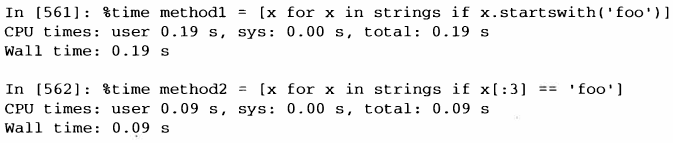


Поскольку эта операция встречается очень часто, в IPython есть две магические функции, %time и %timeit, которые помогают автоматизировать процесс. Функция %time выполняет предложение один раз и сообщает, сколько было затрачено времени. Допустим, имеется длинный список строк, и мы хотим сравнить различные методы выбора всех строк, начинающихся с

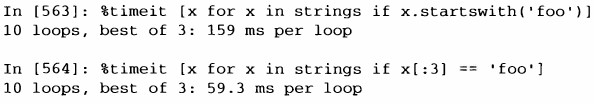
заданного префикса. Вот простой список, содержащий 700 000 строк, и два метода выборки тех, что начи­ наются с 'foo':



На первый взгляд, производительность должна быть примерно одинаковой, верно? Проверим с помощью функции %time:

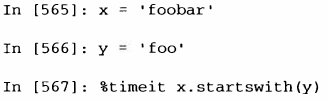


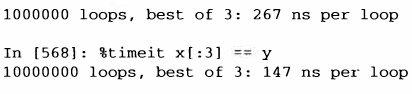
Наибольший интерес представляет величина Wall time (фактическое время). Похоже, первый метод работает в два раза медленнее второго, но это не очень точное измерение. Если вы несколько раз сами замерите время работы этих двух предложений, то убедитесь, что результаты варьируются. Для более точного измерения воспользуемся магической функцией %tiineit. Она получает произвольное предложение и, применяя внутренние эвристики, выполняет его столько раз, сколько необходимо для получения сравнительно точного среднего времени:



Этот, на первый взгляд, безобидный пример показывает, насколько важно хо­ рошо понимать характеристики производительности стандартной библиотеки Python, NumPy, pandas и других используемых в книге библиотек. В больших при­ ложениях для анализа данных из миллисекунд складываются часы!

Функция %timeit особенно полезна для анализа предложений и функций, ра­ ботающих очень быстро, порядка микросекунд (10^-6) или наносекунд ( 10-9 секунд). Вроде бы совсем мизерные промежутки времени, но если функцию, работающую 20 микросекунд, вызвать миллион раз, то будет потрачено на 15 секунд больше, чем если бы она работала всего 5 микросекунд. В примере выше можно сравнить две операции со строками напрямую, это даст отчетливое представление об их характеристиках в плане производительности:

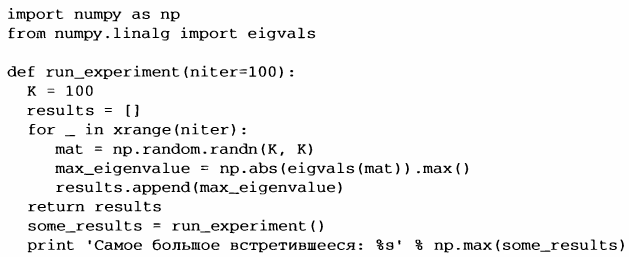




***Простейшее профилирование: %prun* и *%run -р***

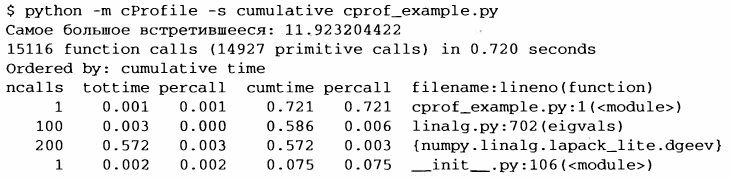
Профилирование кода тесно связано с хронометражем, только отвечает на вопрос, где шtенно тратится время. В Python основное средство профилирования - модуль cProfile, который предназначен отнюдь не только для IPython. cProfile исполняет программу или произвольный блок кода и следит за тем, сколько вре­ мени проведено в каждой функции.

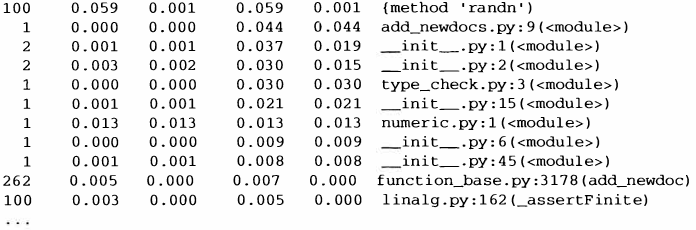
Обычно cProfile запускают из командной строки, профилируют программу целиком и выводят агрегированные временные характеристики каждой функции. Пусть имеется простой скрипт, который выполняет в цикле какой-нибудь алгоритм линейной алгебры (скажем, вычисляет максимальное по абсолютной величине собственное значение для последовательности матриц размерности 100 х 100):

 Незнание NumPy пусть вас не пугает. Это скрипт можно запустить под управ­ лением cProfile из командной строки следующим образом:



Попробуйте и убедитесь, что результаты отсортированы по имени функции. Та­ кой отчет не позволяет сразу увидеть, где тратится время, поэтому обычно порядок сортировки задают с помощью флага - s:

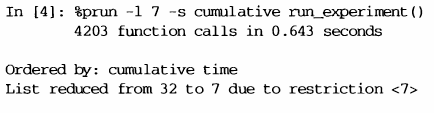


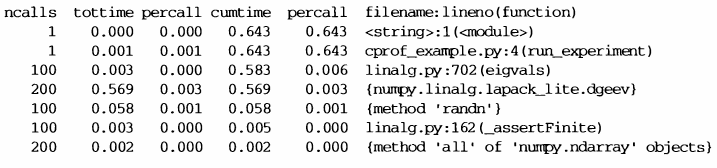


Показаны только первые 15 строк отчета. Читать его проще всего, просматривая сверху вниз столбец cumtime, чтобы понять, сколько времени было проведено внутри каждой функции. Отметим, что если одна функция вызывает другую, то тай­

.мер не останавливается. cProfile запоминает моменты начала и конца каждого вызова функции и на основе этих данных создает отчет о затраченном времени.

сProfilе можно запускать не только из командной строки, но и программно для профилирования работы произвольных блоков кода без порождения нового процесса. В IPython имеется удобный интерфейс к этой функциональности в виде команды%prun и команды%run с флагом -р. Команда%prun принимает те же «аргументы командной строки», что и cProfile, но профилирует произвольное предложение Python, а не ру-файл:

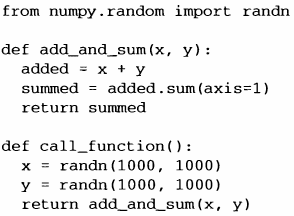


Аналогично команда%run -р -s cumulative срrоf\_ехаmрlе.рудает тотже результат, что рассмотренный выше запуск из командной строки, только не при­ ходится выходить из IPython.

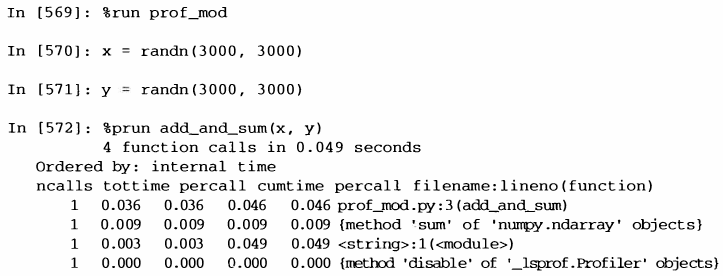
***Построчное профилирование функции***

Иногда информации, полученной от %prun (или добытой иным способом профилирования на основе cProfile), недостаточно, чтобы составить полное пред­ ставление о времени работы функции. Или она настолько сложна, что результаты, агрегированные по имени функции, с трудом поддаются интерпретации. На такой случай есть небольшая библиотека line\_profiler (ее поможет установить PyPI или любой другой инструмент управления пакетами). Она содержит расширение IPython, включающее новую магическую функцию %lprun, которая строит по­ строчный профиль выполнения одной или нескольких функций. Чтобы подключить это расширение, нужно модифицировать конфигурационный файл IPython (см. документацию по IPython или раздел, посвященный конфигурированию, ниже), добавив такую строку:

 Библиотеку line\_profiler можно использовать из программы (см. полную документацию), но, пожалуй, наиболее эффективна интерактивная работа с ней в IPython. Допустим, имеется модуль prof\_rnod, содержащий следующий код, в котором выполняются операции с массивом NumPy:



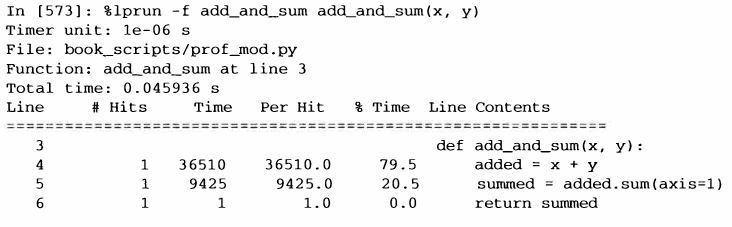
Если бы нам нужно было оценить производительность функции add\_and\_surn, то команда %prun дала бы такие результаты:



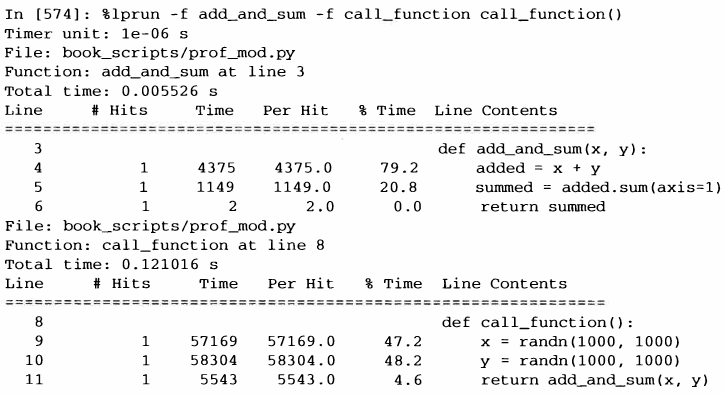
Не слишком полезно. Но после активации расширения IPython line\_profiler становится доступна новая команда %lprun. От %prun она отличается только тем, что мы указываем, какую функцию (или функции) хотим профилировать. Поря­ док вызова такой:



В данном случае мы хотим профилировать функцию add\_and\_sum, поэтому пи­шем:



мы профилировали ту же функцию, которая составляла предложение. Но можно было бы вызвать функ­ цию call\_function из показанного выше модуля и профилировать ее наряду с add\_and\_sum, это дало бы полную картину производительности кода:



Обычно я предпочитаю использовать %prun (cProfile) для «макропрофили­ рования», а %lprun (line\_profiler) - для «микропрофилирования». Полезно освоить оба инструмента.



Явно указывать имена подлежащих профилированию функций в команде

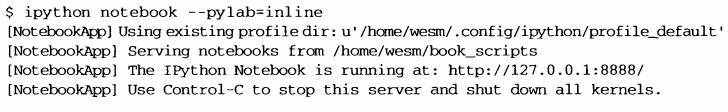
%lprun необходимо, потому что накладные расходы на «трассировку» времени выполнения каждой строки весьма значительны. Трассировка функций, не представляющих интереса, может существенно изменить ре­ зультаты профилирования.

НТМL-блокнот в IPython

В 2011 году команда разработчиков IPython, возглавляемая Брайаном Грейндже­ ром, приступила к разработке основанного на веб-технолоrиях формата интерактивного вычислительного документа под названием блокнот IPython (IPython Notebook). Со временем он превратился в чудесный инструмент для интерак­ тивных вычислений и идеальное средство для воспроизводимых исследований и преподавания. Я пользовался им при написании большинства примеров для этой книги, призываю и вас не пренебрегать им.

Формат iрunb-документа основан нa JSON и позволяет легко обмениваться ко­ дом, результатами и рисунками. На недавних конференциях по Python получил широкое распространение такой подход к демонстрациям: создать iрunb -файлы в блокноте и разослать их всем желающим для экснериментов.

Блокнот реализован в виде облегченного серверного процесса, запускаемого из командной строки, например:

 На большинстве платформ автоматически откроется браузер, подразумевае­ мый по умолчанию, и в нем появится информационная панель блокнота (рис. 3.4). Иногда приходится переходить на нужный URL-aдpec самостоятельно. После этого можно создать новый блокнот и приступить к исследованиям.

Поскольку блокнот работает внутри браузера, серверный процесс можно запу­ стить где угодно. Можно даже организовать безопасное соединение с блокнотами, работающими в облаке, например Amazon ЕС2. На момент написания этой книги уже существовал новый проект NotebookCloud (*http.//notebookcloud.appspot.com*), который позволяет без труда запускать блокноты в ЕС2.

Советы по продуктивной разработке кода с использованием IPython

Создание кода таким образом, чтобы его можно было разрабатывать, отлаживать и в конечном счете *исполъзоватъ* интерактивно, многим может показаться сменой парадигмы. Придется несколько изменить подходы к таким процедурным деталям, как перезагрузка кода, а также сам стиль кодирования.

Поэтому изложенное в этом разделе - скорее искусство, чем наука, вы должны будете экспериментально определить наиболее эффективный для себя способ написания Python-кoдa. Конечная задача - структурировать код так, чтобы с ним было легко работать интерактивно и изучать результаты прогона всей программы или отдельной функции с наименьшими усилиями. Я пришел к выводу, что программу, спроектированную в расчете на IPython, использовать проще, чем анало-

гичную, но построенную как автономное командное приложение. Это становится особенно важно, когда возникает какая-то проблема и нужно найти ошибку в коде, написанном вами или кем-то еще несколько месяцев или лет назад.

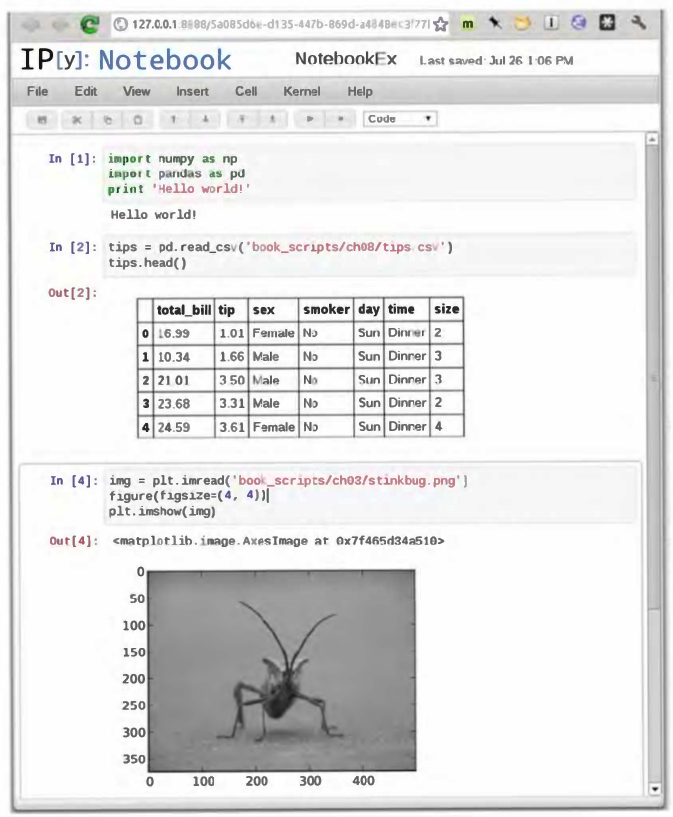
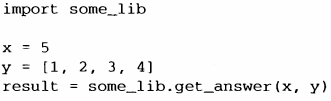


Рис. 3.4. Блокнот IPython

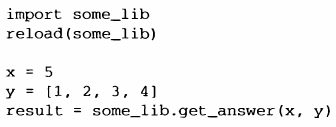
***Перезагрузка зависимостей модуля***

Когда в Руthоn-программе впервые встречается предложение import some lib, выполняется код из модуля some\_lib и все переменные, функции и импор­ тированные модули сохраняются во вновь созданном пространстве имен модуля

some\_lib. При следующей обработке предложения import some\_lib будет воз­ вращена ссылка на уже существующее пространство имен модуля. При интерак­ тивной разработке кода возникает проблема: как быть, когда, скажем, с помощью команды %run выполняется скрипт, зависящий от другого модуля, в который вы внесли изменения? Допустим, в файле test\_script .ру находится такой код:



Если выполнить %run test\_script .ру, а затем изменить some\_lib.py, то при следующем выполнении %run test\_script. ру мы получим старую версию some\_ 1ib из-за принятого в Python механизма однократной загрузки. Такое поведение отличается от некоторых других сред анализа данных, например MATLAB, в ко­ торых изменения кода распространяются автоматически. Справиться с этой про­ блемой можно двумя способами. Во-первых, использовать встроенную в Python функцию reload, изменив test\_script. ру следующим образом:



При этом гарантируется получение новой копии some\_lib при каждом запуске test\_script.ру. Очевидно, что если глубина вложенности зависимостей боль­ ше единицы, то вставлять reload повсюду становится утомительно. Поэтому в IPython имеется специальная функция dreload (не магическая), выполняющая

«глубокую» (рекурсивную) перезагрузку модулей. Если бы я написал import

some\_lib, а затем dreload ( some\_lib), то был бы перезагружен как модуль some\_ 1ib, так и все его зависимости. К сожалению, это работает не во всех случаях, но если работает, то оказывается куда лучше перезапуска всего IPython.

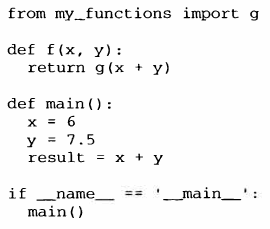
***Советы по проектированию программ***

Простых рецептов здесь нет, но некоторыми общими соображениями, которые лично мне кажутся эффективными, я все же поделюсь.

Сохраняйте ссылки на нужные объекты и данные Программы, рассчитанные на запуск из командной строки, нередко структури­

руются, как показано в следующем тривиальном примере:

Поскольку модуль или пакет может импортироваться в нескольких местах программы, Python кэширует код модуля при первом импортировании, а не выполняет eгo каждый раз. В противном случае следование принципам модульности и правильной организации кода могло бы поставить под угрозу эффективность приложения.



Вы уже видите, что случится, если эту программу запустить в IPython? После ее завершения все результаты или объекты, определенные в функции main, будут недоступны в оболочке IPython. Лучше, если любой код, находящийся в main, будет исполняться прямо в глобальном пространстве имен модуля (или в блоке if ‘\_name\_’ == ‘ \_main\_ ‘: , если вы хотите, чтобы и сам модуль был импортируемым). Тогда после выполнения кода командой %run вы сможете просмотреть все переменные, определенные в main. В таком простом примере это неважно, но далее в книге будут рассмотрены сложные задачи анализа данных, в которых уча­ ствуют большие наборы, и их исследование может оказаться весьма полезным.

**Плоское лучше вложенного**

Глубоко вложенный код напоминает мне чешуи луковицы. Сколько чешуй придется снять при тестировании или отладке функции, чтобы добраться до ин­ тересующего кода? Идея «плоское лучше вложенного>.'> - часть «Свода мудрости Python>.'>, применимая и к разработке кода, предназначенного для интерактивного использования. Чем более модульными являются классы и функции и чем мень­ ше связей между ними, тем проще их тестировать (если вь1 пишете автономные тесты), отлаживать и использовать интерактивно.

**Перестаньте бояться длинных файлов**

Если вы раньше работали с Java (или аналогичным языком), то, наверное, вам говорили, что чем файл короче, тем лучше. Во многих языках это разумный совет; длинный файл несет в себе дурной «запашок>.'> и наводит на мысль о необходи­ мости рефакторинга или реорганизации. Однако при

разработке кода в IPython наличие 10 мелких (скажем, не более 100 строчек) взаимосвязанных файлов с большей вероятностью вызовет проблемы, чем при работе всего с одним, двумя или тремя файлами подлиннее. Чем меньше файлов, тем меньше нужно переза­ гружать модулей и тем реже приходится переходить от файла к файлу в процессе редактирования. Я пришел к выводу, что сопровождение крупных модулей с высо­ кой степенью внутренней сцепленности гораздо полезнее и лучше соответствует духу Python. По мере приближения к окончательному решению, возможно, имеет смысл разбить большие файлы на более мелкие.

Понятно, что я не призываю бросаться из одной крайности в другую, т. е. по­ мещать весь код в один гигантский файл. Для отыскания разумной и интуитив

но очевидной структуры модулей и пакеtов, составляющих большую программу, нередко приходится потрудиться, но при коллективной работе это очею, важно. Каждый модуль должен обладать внутренней сцеплешюстыо, а местонахождение функций и классов, относящихся к каждой области функциональности, должно быть как можно более очевидным.

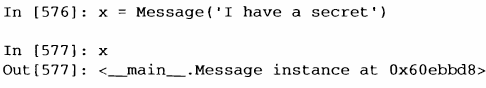
Дополнительные возможности IPython

***Делайте классы дружественными к Ipython***

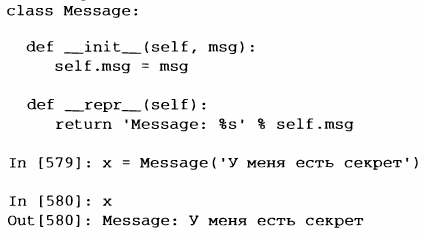
В IPython предпринимаются все меры к тому, чтобы вывести на консоль понят­ ное строковое представление инспектируемых объектов. Для многих объектов, в частности словарей, списков и кортежей, красивое форматирование обеспечива­ ется за счет встроенного модуля pprint. Однако в классах, определенных поль­ зователем, порождение строкового представ.тiсния возлагается на автора. Рассмо­ трим такой простенький класс:



Вы будете разочарованы тем, как такой класс распечатывается по умолчанию:



IPython принимает строку, возвращенную магическим методом \_repr\_ (вы­ полняя предложение output = repr ( obj) ), и выводит ее па консоль. Но раз так, то мы можем включить в класс простой метод \_repr\_, который создает более полезное представление:



***Профили* и *конфигурирование***

Многие аспекты внешнего вида (цвета, приглашение, расстояние между стро­ ками и т. д.) и поведения оболочки IPython настраиваются с помощью развитой системы конфигурирования. Приведем лишь несколько примеров того, что можно сделать.

* Изменить цветовую схему.
* Изменить вид приглашений ввода и вывода или убрать пустую строку, пе­чатаемую после out и перед следующим приглашением In.
* Выполнить список произвольных предложений Python. Это может быть,

например, импорт постоянно используемых модулей или вообще все, что должно выполняться сразу после запуска IPython.

* Включить расширения IPython, например магическую функцию %lprun в модуле line\_profiler.
* Определить собственные магические функции или псевдонимы систем­

ных.

Все эти параметры задаются в конфигурационном файле ipython\_config .ру, находящемся в каталоге -/.conf ig/ipython/ в UNIХ-системе или в каталоге

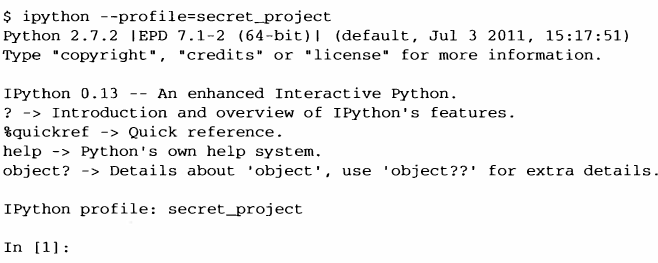
%НОМЕ%/.ipython/ в Windows. Где находится ваш домашний каталог, зависит от системы. Конфигурирование производится на основе конкретного профиля. При обычном запуске IPython загружается профиль по у.молчанию, который хранится в каталоге profile\_default. Следовательно, в моей Linux-cиcтeмe полный путь к конфигурационному файлу IPython по умолчанию будет таким:

*/home/wesm/.config/ipython/profile\_default/ipython\_config.py*

Не стану останавливаться на технических деталях содержимого этого файла. По счастью, все параметры в нем подробно прокомментированы, так что оставляю их изучение и изменение читателю. Еще одна полезная возможность - помержка сразу нескольких профилей. Допустим, имеется альтернативная конфигурация IPython для конкретного приложения или проекта. Чтобы создать новый профиль, нужно всего лишь ввести такую строку:

*ipython profile create secret\_project*

Затем отредактируйте конфигурационные файлы во вновь созданном каталоге profile\_secret\_project и запустите IPython следующим образом:



Как всегда, дополнительные сведения о профилях и конфигурировании можно найти в документации по IPython в сети.

Благодарности

Материалы этой главы частично были заимствованы из великолепной документации, подготовленной разработчиками IPython. Я испытываю к ним бесконечную благодарность за создание этого восхитителыюго набора инструментов.