Muslimov Arthur

(since 2.01.20)

*IPython: за пределами обычного Python*

Если сам Python – это механизм решения задач, то IPython – это его пульт управления.

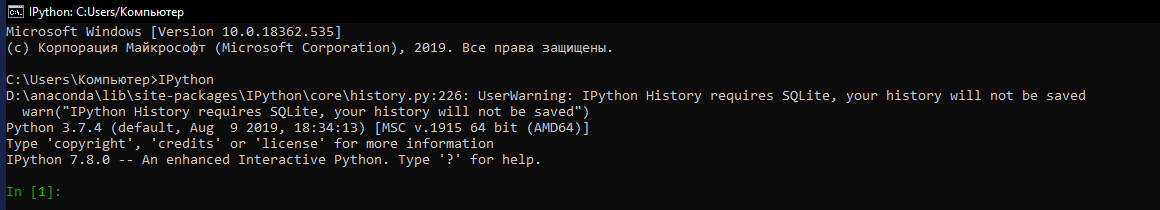
Эта оболочка тесно связана с проектом Jupyter, представляющий из себя блокнот с полезными инструментами. IPython позволяет эффективно использовать Python для работы с большими данными, совместным использованием кода и может интерактивно его выполнять.

*Командная строка или блокнот*

Работать с IPython можно двумя способами: с помощью командной строки или блокнота Jupyter.

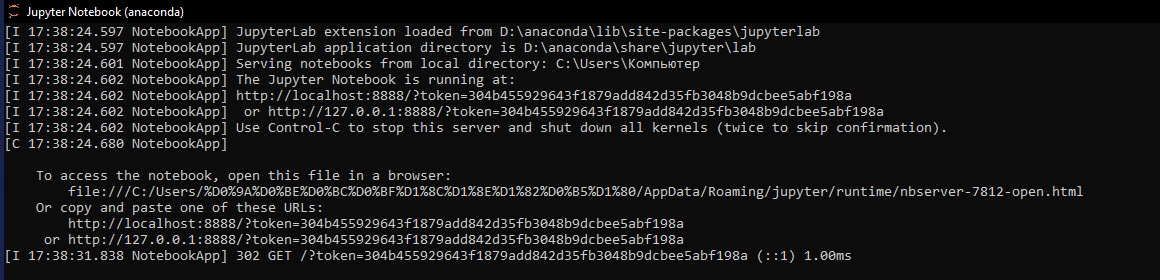
*Запуск командной оболочки IPython*

IPython у вас уже должен стоять, если вы установили дистрибутив Anaconda. Для запуска достаточно ввести в консоль IPython, после чего вы увидите приглашение.



*Запуск блокнота Jupiter*

В терминале всего лишь надо ввести *jupyter notebook*, после чего скрипт сам запустит хост и откроет браузер с вашим блокнотом (но у меня ничего просто-так не работает, и я вставляю в браузер то, что предлагает jupyter). Jupyter представляет из себя браузерный графический интерфейс для IPython со множеством разных плюшек, вроде динамической визуализации.

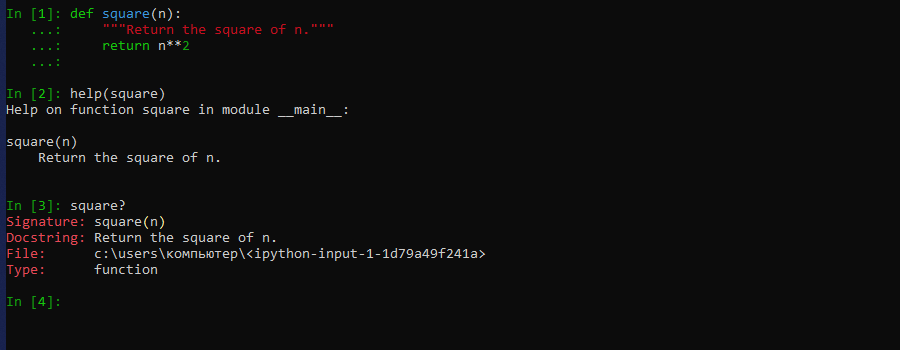


*Справка и документация в оболочке IPython*

Одна из самых полезных особенностей IPython/Jupyter – это легкий доступ к информации. И, хотя всё можно найти в интернете, наиболее частые вопросы можно решить в несколько кликов не выходя из IPython:

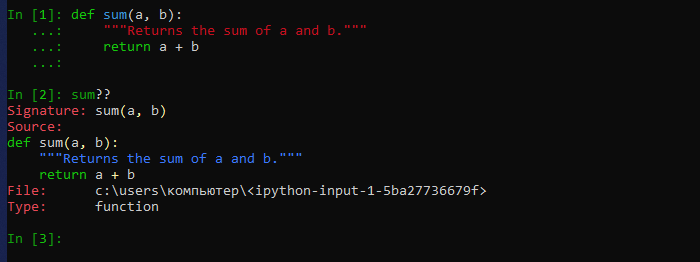
* Как вызвать эту функцию? Какие аргументы и параметры у неё есть?
* Как выглядит код этого объекта?
* Что есть в этом пакете?

*Доступ к документации с помощью символа ?*

Как мы знаем (но я не знал), в Python есть встроенная функция help(), печатающая docstring этого объекта на экран. Т.к. это, видимо, довольно частая практика, создатели IPython ввели продвинутую версию этой функции. Теперь надо всего лишь поставить знак ? после объекта, и IPython любезно расскажет вам что это такое. 

*Доступ к исходному коду с помощью символов ??*

Чтение краткой документации объекта - это конечно хорошо. Но что может быть лучше, чем самим взглянуть на его код? IPython даёт эту возможность – простое добавление ?? после имени объекта, и вы увидите его изнутри.



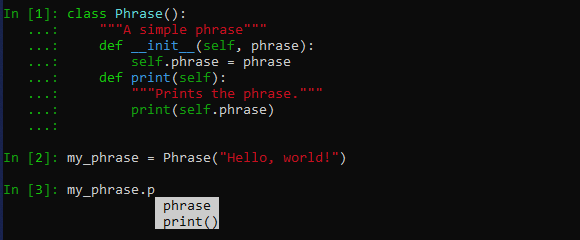
Использование ?/?? – это простой способ добычи информации о работе функции или модуля. Но ?? иногда может выдавать то, что должен выдавать простой ?. Это связано с тем, что функция, к которой ты обращаешься написана на C, а не на Python.

*Просмотр содержимого модулей с помощью Tab-автодополнения*

Другой удобный интерфейс IPython – использование Tab для автодополнения и просмотра содержимого объектов, модулей, пространств имён.

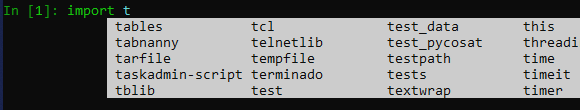
*Автодополнение названий содержимого объектов с помощью клавиши Tab*

Как мы знаем (но я опять не знал), в Python есть встроенная функция dir(), в выводящая все внутренности (атрибуты и методы) объекта или класса. IPython предлагает её облегчённую (от слово легко, просто) версию этой функции. Надо всего лишь поставить точку сразу после имени объекта (класса) и нажать на клавишу Tab на клавиатуре, после чего оболочка выведет аккуратный интерактивный список его атрибутов и методов без излишеств, вроде \_\_reduce\_\_().



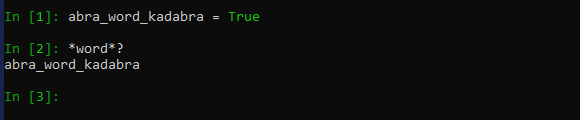
*Автодополнение с помощью клавиши Tab во время импорта*

По такому же принципу Tab-автодополнения работает с модулями, выводя функции и классы, определённые в них. Важно заметить, что вы можете ввести только начальные символы того, чего хотите найти, и IPython будет предлагать объекты, судя по ним. Можно вообще ничего не вводить, и оболочка выведет весь ваш арсенал.



*Помимо автодополнения табуляцией, подбор по джокерному символу*

Иногда бывает так, что ты помнишь только часть имени. Можно конечно перерывать весь список, выдаваемый через Tab, но есть способ лучше – поставить знак \* в забытой части имени, а знак ? в конец. Так IPython поймёт, что ты что-то потерял.



*Сочетания горячих клавиш в командной оболочке IPython*

Горячие клавиши известны тем, что позволяют сократить время при выполнении сложных операций. В IPython они также присутствуют, но они предоставляются не от самой IPython, а от её зависимости – GNU Readline, поэтому на разных компьютерах комбинации горячих клавиш выглядят немного по-разному. Автор рекомендует пользоваться продвинутыми текстовыми редакторами, вроде Emacs или Vim, где подобных комбинаций навалом, и они могут сильно облегчить жизнь.

*Навигационные горячие клавиши*

Перемещение курсора стрелочками присутствует и тут. Но есть и другие возможности.

|  |  |
| --- | --- |
| **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 вперёд на один символ |

*Горячие клавиши ввода текста*

Я думаю, что это самые полезные комбинации, т.к. пригодиться они могут довольно часто. Вы поймёте, что привыкли к ним, когда заметите, что только что использовали не привычный Backspace, а сочетания Ctrl+B и Ctrl+D.

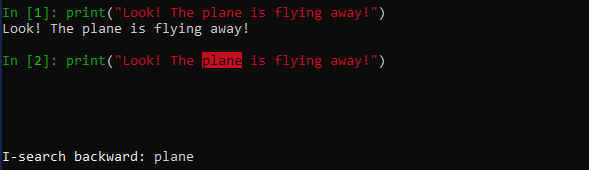
|  |  |
| --- | --- |
| **Комбинация клавиш** | **Действие** |
| Backspace | Удаляет предыдущий символ в строке |
| Ctrl+D | Удаляет следующий символ в строке |
| Ctrl+K | Вырезает текст, начиная от курсора до конца строки |
| Ctrl+U | Вырезает текст с начала строки до курсора |
| Ctrl+Y | Вставляет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

*Горячие клавиши для истории команд*

|  |  |
| --- | --- |
| **Комбинация клавиш** | **Действие** |
| Ctrl+P (или стрелка вверх) | Доступ к предыдущей команде по истории |
| Ctrl+N (или стрелка вниз) | Доступ к следующей команде по истории |
| Ctrl+R | Поиск по истории |

Вероятно, наиболее важные сочетания – это горячие клавиши навигации по истории команд. Вроде как она сохраняется в отдельной SQLite базе данных в папке с IPython, что позволяет вести поиск по командам, введённым в прошлых сеансах.

Наиболее полезным может оказаться Ctrl+R.



*Прочие горячие клавиши*

|  |  |
| --- | --- |
| **Комбинация клавиш** | **Действие** |
| Ctrl+L | Очистить экран терминала |
| Ctrl+C | Остановка текущей программы |
| Ctrl+D | Выйти из сеанса IPython (если нет ввода) |

Имеется ещё несколько сочетаний клавиш, не относящихся ни к одной из предыдущих категорий, но заслуживающих упоминания.

Использование всех этих комбинаций клавиш может быть утомительно, но если вы приноровитесь, то позже будете жалеть, что их нет в других программах.

*«Магические» команды IPython*

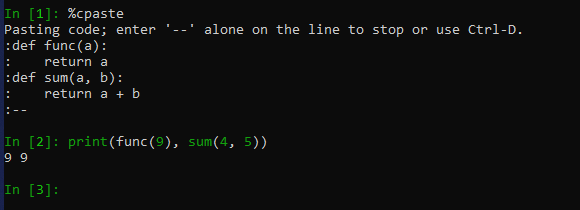
«Магические» команды, как предыдущие расширения, призваны ускорить вашу работу. Отличаются они от других, тем, что относятся уже не к коду Python, а к дополнениям оболочки IPython. Есть два вида «магических» команд. Строчные работают только с одной строкой кода и обозначаются символом %. Блочные же работают, исходя из названия, с целыми блоками и обозначаются символами %%.

*Вставка блоков кода: %paste и %cpaste*

Есть одна распространённая проблема – при вставке многострочных блоков текста IPython может быть сбит с толку лишними символами. Например, если копировать код из стандартного IDE Python. Эту проблему решает «магическое» слово %paste.

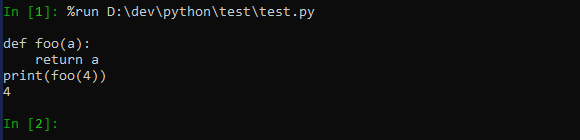


Как ты видишь, три стрелки перед определением функции не напугали IPython.Если ты хочешь вставить код в несколько заходов, то лучше использовать %cpaste. Эта команда даёт возможность вставлять код постепенно.



*Выполнение внешнего кода: %run*

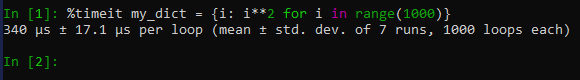
Часто приходится работать с уже написанными программами. Чтобы постоянно не копировать весь их код, создатели оболочки придумали %run, выполняющую его прямо в оболочке. Это «магическое» слово также выполняет роль import’а, т.к. вам становятся доступны функции данной программы.



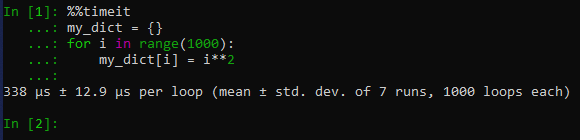
Стоит заметить, что символы ? и ?? также работают и с «магическими» командами.

*Длительность выполнения кода: %timeit*

Это уже довольно полезная «магическая» функция. Она проводит несколько тестов для быстрых функций или команд, чтобы определить их среднее время выполнения.



Не забываем, что символы %% дают возможность применять «магическую» команду для целого блока кода.



Здесь мы видим, что цикл каким-то боком оказался быстрее генератора.

*Справка по «магическим» функциям: ?, %magic и %lsmagic*

Как я уже и говорил, вы можете использовать символы ? и ?? для получения информации о «магических» функциях. Чтобы получить общие сведения о них можно ввести %magic. Чтобы просто увидеть список всех доступных «магических» функций, достаточно ввести %lsmagic. При желании, можно ввести свою «магическую» функцию, но для этого следует обратиться к документации.

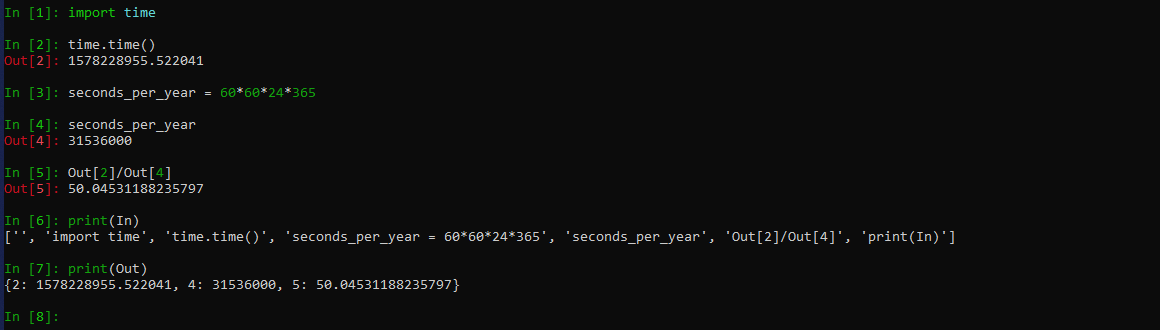
*История ввода и вывода*

Мы уже знаем, что стрелками можно возвращать предыдущие команды. Но IPython также даёт возможность получать выводы этих команд, и даже их строковую версию.

Объекты In и Out оболочки IPython

Приглашения In[1] и Our[1] – это не только красивый интерфейс, но ещё и хранилища.

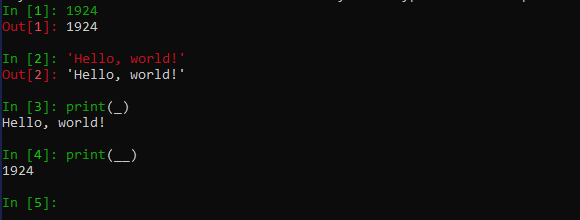
In представляет из себя список всех введённых команд, который можно вывести на экран. Out же представляет из себя словарь, ключи которого – номера вводов команд, а значение – выводы этих команд, которые позже можно будет использовать. Это особенно полезно, когда требуется результат долгих вычислений.



Как мы видим, уже прошло чуть более, чем 50 лет с 1970 года. Обратите внимание, что ввод начинается с In[1]. В списке In[0] – это всего лишь пустышка. Также мы видим, что вывод print() не входит в Out. И это правильно, ведь вывод print()’a – просто None.

*Быстрый доступ к предыдущим выводам с помощью знака подчеркивания*

Стандартный Python даёт возможность получить предыдущий вывод с помощью команды print(\_). IPython же немного расширяет эту возможность. Теперь вы можете вывести до 3-ёх последних выводов, вставляя по знаку, как индекс, \_ в print().

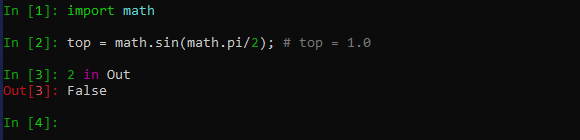


Но только до 3-ёх. Дальше оболочка уже не позволит, т.к. будет трудно уследить за всеми знаками \_. Здесь уже удобнее использовать Out[n], или же, его сокращение - \_n.



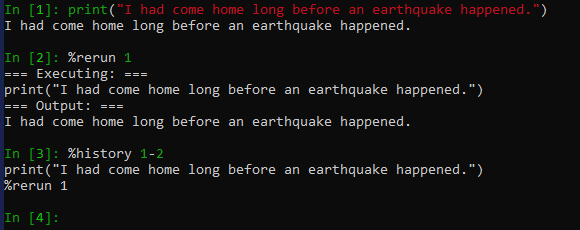
*Подавление вывода*

Иногда бывает так, что вы хотите, чтобы вывод следующей команды не сохранялся в истории (т.е. в Out). Для этого достаточно просто добавить символ ; в конец команды. Сама команда при этом выполняется как-бы «бесшумно».



*Соответствующие «магические команды»*

Если вы хотите получить сразу несколько вводов, то это случай функции %history. Если вы хотите заново выполнить какую-то часть команд, то вам к %rerun. Если вы желаете сохранить какую-то часть ввода, то обратитесь к %save.



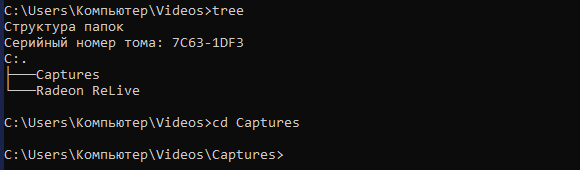
*Оболочка IPython и использование системного командного процессора*

Вы наверное встречались с проблемой стандартной IDE Python, когда вам приходилось переключаться между несколькими окнами для работы с cmd. Здесь же достаточно ввести знак ! и команду, которую оболочка как-бы пошлёт в cmd.

*Краткое введение в использование командного процессора*

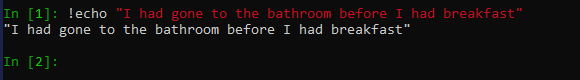
Начиная с середины 80-х годов прошлого века начали появляться операционные системы с графическим оформлением. С тех самых пор люди и приучились к мышке, как основным пультом управления своего компьютера. Но ОС были и раньше, просто выглядели они по-другому. Вместо красивых окошек – фактически голый терминал. Это и были предшественники современных командных процессоров.

Если вы не работали с cmd раньше, то вы наверно будете озадачены вопросом – «А зачем мне это вообще нужно, если у меня есть мышь и окна?». На самом деле, работать через cmd может быть проще. Вот пример манипуляции с директориями.



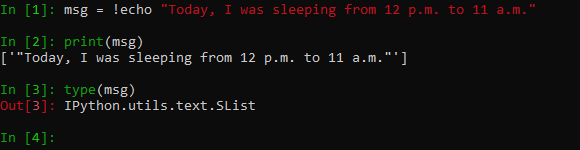
*Инструкции командного процессора в оболочке IPython*

Как я уже и говорил, с помощью знака ! вы можете посылать команды прямо из IPython’а прямо в cmd и получать вывод.



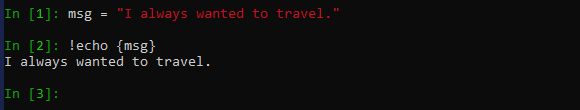
*Передача значений в командный процессор и из него*

Через знак = и ! мы можем получать вывод cmd в свою переменную.



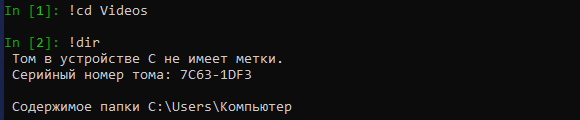
Заметьте, что мы получаем не обычный список, а его специальную модификацию IPython’а, в которой есть несколько дополнительных методов и свойств.

Ещё мы можем отправлять свои данные в терминал через знаки {}.

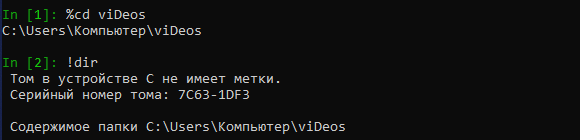


*«Магические» команды для командного процессора*

Если вы попробуете поиграться с инструкциями cmd, то заметите, что !cd не работает.



Это связано с тем, что IPython как-бы создаёт временную cmd каждый раз, при использовании !. Если вы хотите, чтобы ваши действия сохранялись, то вам следует использовать «магические» версии этих команд. Конкретно здесь - %cd.



На самом деле знак % здесь можно даже и не использовать. Команда %cd – это «автомагическая» функция. Такое поведение настраивается с помощью %automagic. Вам также доступны и другие «автомагические» функции: %cat, %cp, %env, %ls, %man, %mkdir, %more, %mv, %pwd, %rm и %rmdir. Это Unix-функции, но, как минимум, некоторые из них выполняют схожие Windows-функции.



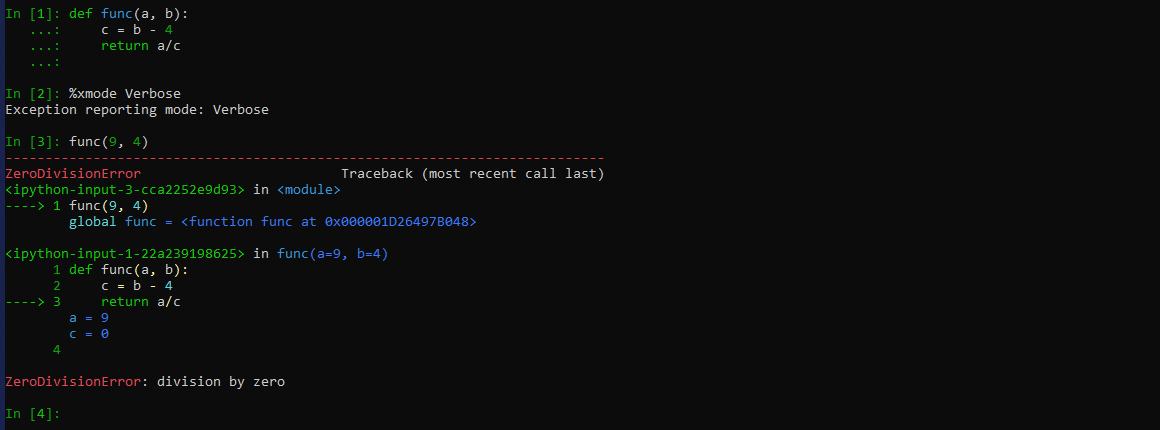
Таким образом, IPython превращается в обычный системный терминал.

*Ошибки и отладка*

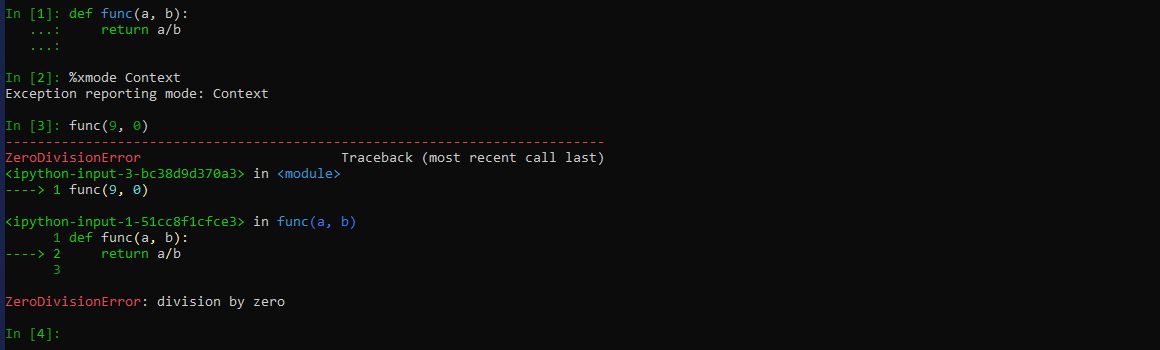
Разработка кода и анализ всегда требуют некоторого количества ошибок. Даже Бьярне Страуструп говорил, что создание полностью готовой программы без единой ошибки в процессе – это целый праздник, т.к. случается такое реже, чем раз в год. В IPython есть некоторые инструменты, которые упростят вам процесс отладки.

*Управление исключениями: %xmode*

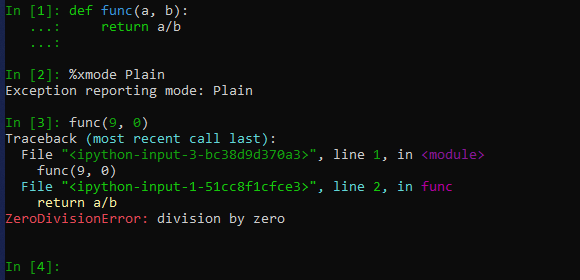
Простой Python при исключении почти всегда выдаёт трассировку (т.е. путь, который привёл к ошибке), чтобы ты мог разобраться с проблемой. IPython же даёт возможность управлять степенью выводимой информации. Функция %xmode принимает только один аргумент – режим. Всего режимов три: Plain (простой), Context (такой как у Python) и Verbose (Расширенный). Их можно переключать по очереди, просто вводя %xmode. Эта особенность в книге не описывалась, я сам её заметил.



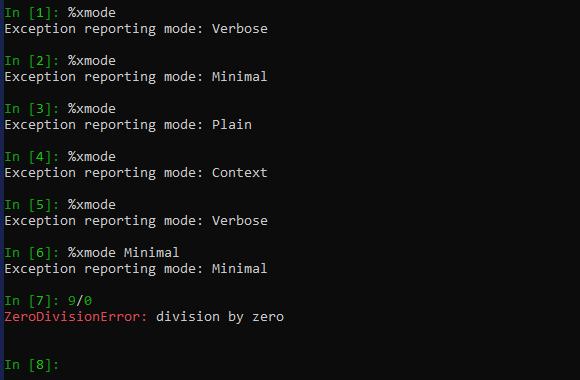
Режим Verbose выводит значения наиболее важных переменных, но даже так вы можете запутаться во всех этих данных. Выбирайте наиболее удобный режим!



Режим Context вроде-как никак не отличается от привычной трассировки Python.



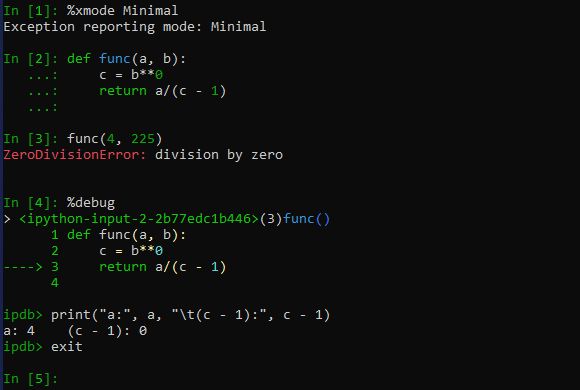
Режим Plain минимум информации, чтобы ты мог отследить причину исключения.



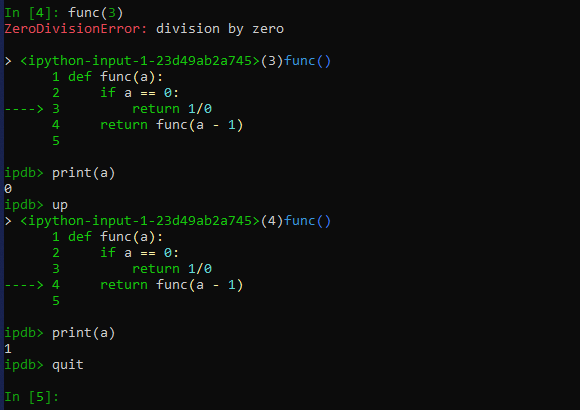
Как оказалось, в IPython есть ещё один режим – Minimal, выводящий только само название исключение. В книге говорилось, что режимов только три.

*Отладка: что делать, если чтения трассировок недостаточно*

В Python есть стандартная утилита для интерактивной отладки – pdb. В IPython есть её расширенная версия – ipdb. Существует множество способов их запуска. Если интересно, читай документацию. Вероятно, самый удобный вход в неё – функция %debug. Если её вызвать после исключения, то откроется интерактивная командная строка, где вы можете с помощью встроенных функций (их, благо, не много) отслеживать ход выполнения вашей программы. Вот простенький пример.



Однако, ipdb способен на большее, чем показать слепок последнего момента.



Как вы видите, можно путешествовать по стёку. И это ещё не все возможности!

|  |  |
| --- | --- |
| **Команда** | **Описание** |
| list | Отображает текущее место в файле (стрелкой слева, как на скриншоте) |
| h(elp) | Отображает список команд или справку по конкретной команде (?) |
| q(uit) | Выход из отладчика и программы (?) |
| c(ontinue) | Выход из отладчика, продолжение выполнения программы |
| n(ext) | Переход к следующему шагу команды (?) |
| <enter> | Повтор предыдущей команды |
| p(rint) | Вывод значений переменных |
| s(tep) | Вход в подпрограмму (?) |
| r(eturn) | Возврат из подпрограммы |

Наконец, если у вас есть сценарий, который вы хотели бы запустить в интерактивном режиме, то достаточно запустить его с префиксом %run –d использовать next.

*Профилирование и мониторинг скорости выполнения кода*

В процессе разработки кода и создания конвейеров обработки данных всегда присутствуют компромиссы между различными реализациями. Лучше следовать знаменитому афоризму Дональда Кнута: «Лучше не держать в голове подобные “малые” вопросы производительности, скажем, 97% времени: преждевременная оптимизация – корень всех зол».

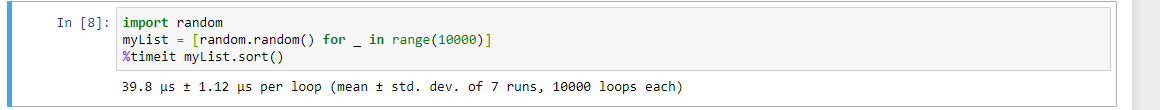
Однако, как только ваш код начинает работать, будет полезно заняться его производительностью. Здесь вам и помогут некоторые команды IPython’а:

* %time – длительность выполнения отдельного оператора;
* %timeit – длительность выполнения отдельного оператора при неоднократном повторе, для большей точности (о нём говорилось ранее);
* %prun – выполнение кода с использованием профилировщика;
* %lprun – пошаговое выполнение кода с применением профилировщика;
* %memit – оценка использования оперативной памяти для отдельного оператора;
* %mprun – пошаговое выполнение кода с применением профилировщика памяти.

Последние четыре (на самом деле три) команды не включены в стандартный пакет IPython – для их использования необходимо установить line\_profiler и memory\_profiler. Ну или вы установили готовый пакет Anaconda, и радуетесь жизнью.

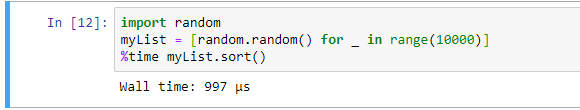
*Оценка времени выполнения фрагментов кода: %timeit и %time*

Как ты помнишь, %timeit выполняет быстрые команды множество раз. Как оказалось, это не всегда хорошо. Вот допустим у нас есть список, который нужно отсортировать.



Как ты видишь, это уже не командная строка. Это нечто более страшное – Jupyter. Автор не стал уделять внимание на него (чего я ждал), он просто перечислил его достоинства в начале книги. Всё было хорошо, пока он не стал показывать многострочные примеры в одной ячейке. Это был знак, что я уже должен был пользоваться блокнотами Jupyter’а. Как оказалось, в них нет привычных мне комбинаций клавиш (их перехватывает браузер) и меня пугают неизвестные мне инструменты на панели сверху. Также я заметил, что блочные «магические» функции (которые начинаются с %%) тут работают как-то по-другому. Ещё здесь странная система ячеек: с каждым новым запуском кода, индекс ячейки меняется, а как отмотать его назад я не знаю. Всё это значит, что мне придётся уделять время на освоение Jupyter. И это печально, т.к. книгу я покупал как-раз для того, чтобы автор мне всё разжевал, а не просто отсылал к документации.

Вроде бы ничего необычного здесь нет, пока ты не вспомнишь, что сортировка уже отсортированного списка проходит гораздо быстрее. %timeit множество раз прогнал этот код (в нашем случае аж больше 10к раз). Но если использовать «магическую» команду %time, прогоняющую код только один раз, то результат будет совсем другим.



997 us у %time против 39,8 us у %timeit. Чувствуешь разницу? И я не почувствовал бы, но если именно на эту часть кода вы будете использовать в основном цикле программы, то общая производительность будет очень даже ощутимо ниже, чем вы планировали. Ещё вам стоит знать, что %timeit использует некоторые уловки для ускорения вашего кода. Например она не использует сборщик мусора Python. Подробности опять же следует искать в документации.

*Профилирование сценариев целиком:*

Часто бывает полезно измерить производительность всей программы в целом. Для этого и придумано профилирование. Есть правило 10/90 – это когда 10% кода занимают 90% времени. Именно на эти 10% и укажет профилировка. В общем, она указывает на наиболее долгие или востребованные участки кода – их называют Hot\_points. В Python есть собственный профилировщик, а наиболее удобный способ вызвать его (из IPython, конечно же) – команда %prun.



Как мы видим, результат выведен в пейджере (окошке снизу). Я почитал немного про профилировщики и понял, что этот для меня не самый удобный, т.к. оформление функций не явное. Но отсюда можно понять, что наибольшее количество времени в этой маленькой функции отнял метод включения список (ну, так автор написал).

*Пошаговое профилирование с помощью %lprun*

Конечно можно попытаться вытянуть информацию из того, что выдаёт %prun, но, как оказалось, есть вариант более удобного представления отсчёта профилирования. Но тут мы уже выходим за пределы Python’a, и даже IPython’а. Вам следует установить пакет line\_profiler через pip, как говорит автор, но на самом деле, если вы используете conda (да, conda входит в дистрибутив anaconda), то нужно использовать следующую команду. Так вы избежите проблемы совместимости (это моя догадка).

conda install  line\_profiler

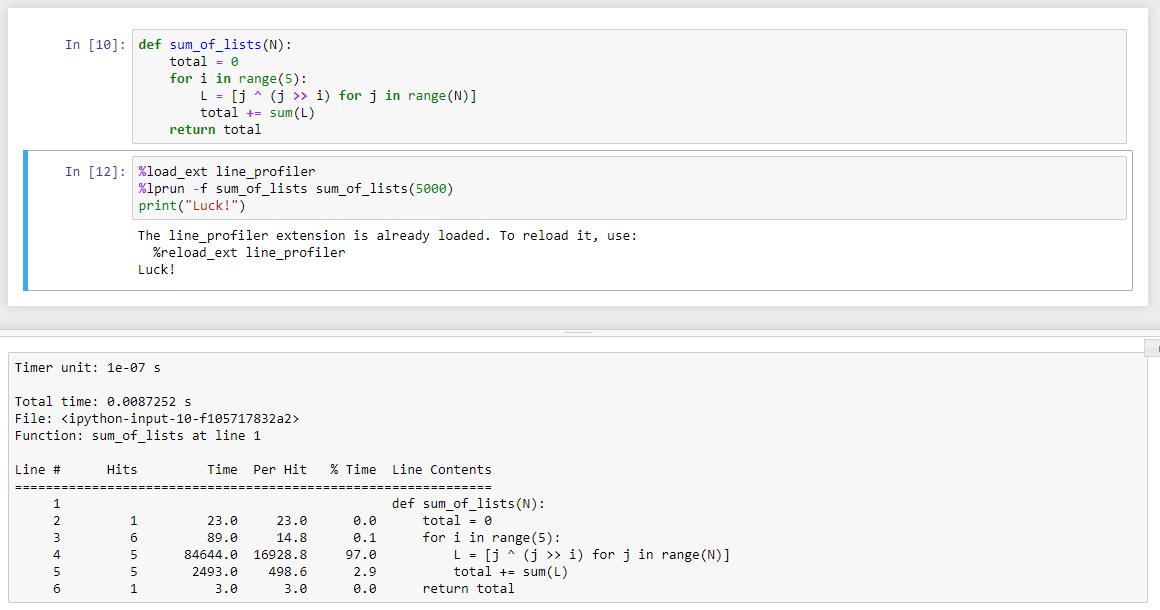
Как ты видишь, я додумался выделять команды. Так читать гораздо приятнее. Ещё я не потратил время на скриншот.

Дальше следует уведомить IPython о новой команде. В оболочке вводим это:

%load\_ext line\_profiler

**Наверное именно так и вводятся собственные магические функции. Вводить эту команду нужно при каждой новой сессии IPython, т.к. он не запоминает её.**

Доступ к новому профилировщику тоже идёт «через магическую» команду. Чтобы использовать его, надо явно тыкнуть ему на функцию.



Вот это уже более цивилизованное оформление. В начале пейджера нас осведомляют, что за единицу времени берут десятую часть микросекунды. Ну а дальше ты и сам прочесть можешь. Как мы видим, 97% времени занимает 4 строчка функции – т.е. генератор. Если у тебя много времени – можешь ввести %lprun? и почитать.

*Профилирование использования памяти: %memit и %mprun*

Есть ещё один аспект профилирования – счёт памяти. Для него опять нужно обратиться к расширениям IPython. Опять используем conda.

conda install memory\_profiler

И также опять придётся осведомлять IPython о новых «магических» командах.

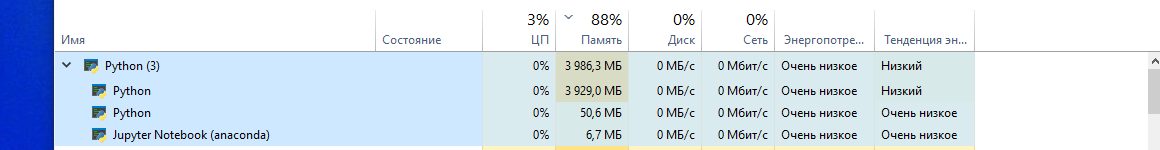
%load\_ext memory\_profiler

Это расширение даёт нам две полезные функции. Одна из них - %memit - этакий аналог %timeit, но считающий память и выводящий максимум его потребления.



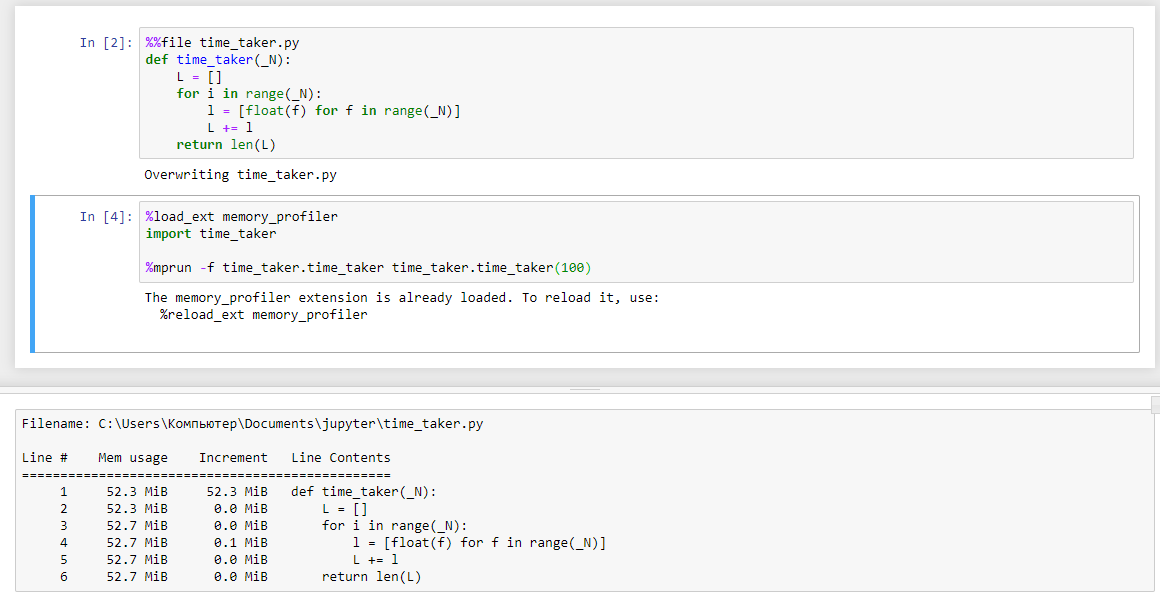
**Изначально, это была функция для %lprun. Поэтому её имя – time\_taker. Позже я переписал её, но забыл переименовать и она попала на скриншот.**

Как мы видим, наша функция достигала размера в 4.5 гибибайта, из 8 доступных на моём средненьком компьютере. Ты наверное заметил, что я вставил в функцию input(“”). Так, у меня появилось время, чтобы сделать следующий скриншот.



Да, тут показатель меньше, но не забывай, что %memit показывает самый пик.

Вторая функция – %mprun – аналог %lprun. Ей также надо явно указать функцию, но работать она умеет только с файлами. Это повод использовать команду %%file. Не волнуйся, эта функция, как и многие другие, интуитивно понятна.



**Спустя, наверное, неделю попыток я так и не смог запустить этот профилировщик. Я, конечно, пытался найти замену, но другие, по разным причинам, я не стал использовать (какие-то выводили потребление в общем, какие-то в неудобном виде). В итоге я сдался, и не стал останавливаться на этой проблеме. Если нужно – почитай о проекте guppy.**

***Дополнительные источники информации об оболочке IPython***

**В данной главе мы захватили лишь верхушку айсберга по использованию языка Python для**

**работы с данными. Гораздо больше информации доступно как в других книга, так и в интернете. Вот несколько рекомендаций от автора.**

**Веб-ресурсы:**

* **Сайт IPython (**<http://ipython.org/>**) – в общем то, сам сайт разработчиков с документацией, примеров и множеством других ресурсов.**
* **Сайт nbviewer (**<http://nbviewer.ipython.org/>**) – кладезь блокнотов Jupyter для изучения. Знайте! Большую часть времени мы будем смотреть на чужой код!**
* **Галерея интересных блокнотов оболочки IPython - (**[http://github.com/ipython/ipython/wiki/A-gallery -of-interesting-ipython-Notebooks/](http://github.com/ipython/ipython/wiki/A-gallery%20-of-interesting-ipython-Notebooks/)**). Поддерживается nbviewer, непрерывно растёт, «демонстрирует глуби ну размах численного анализа» © Автор. Ещё здесь есть целые книги и туториалы в блокнотах Jupyter.**
* **Видеоруководства. В интернете есть немало видеокурсов, обучающих IPython (да и всему угодно). Автор рекомендует Фернандо Переса и Брайана Грейнджера – создателей Ipython и Jupyter – с конференций PyCon, SciPy and PyData.**

**Книги:**

* **Python for Data Analysis («Python и анализ данных»). Эта книга Уэса Маккинли также включает в себя разбор IPython в отдельной главе, что значит, будет во многом пересекаться с данной, но вторая точка зрения никогда не помешает.**
* **Learning Ipython for Interactive Computing and Data Visualization. Эта книга Цириллы Россан предлагает неплохое введение в использование IPython.**
* **IPython Interactive Computing and Visualization Cookbook. Эта ещё одна книга Цириллы Россан, представляет из себя что-то вроде справочника, но с дополнительной информацией широкого спектра проблем науки о данных.**

**Ну и конечно же, не помни символ ?, который уже обсуждался, и пользуйся им как только встретишь что-то новое и незнакомое. Не стесняйся!**